COMS W4701: Artificial Intelligence

Lecture 4: Adversarial Search and Games

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Where are we now?

- Planning and identification (assignment) problems
- Applications: Pathing, motion planning, scheduling, …

- Solution: General-purpose search algorithms
  - NP-hard (exponential) in the worst / naïve case
  - Different strategies to make search more efficient

- We have only considered **single-agent** problems

Today

- Adversarial search problems
- Minimax algorithm
- Alpha-beta pruning, evaluation functions, move ordering
- Stochastic games

AI and Games

- 1950s and 60s: First checkers programs, some from reinforcement learning
- 1994: *Chinook* declared computer champion in checkers against Tinsley
- 2007: Checkers is solved (completely predictable)

- 1997: *Deep Blue* defeats chess world champion Kasparov
- 2017, 2019: *AlphaZero*, *Leela Chess Zero* defeat *Stockfish*, AI chess champ

- 2016: *AlphaGo* defeats go world champion Lee Sedol
- 2018-2020: *AlphaZero*, *MuZero* achieve state-of-the-art in chess, go, shogi

Adversarial Search

- Several approaches for competitive multi-agent environments

- Economics: Study aggregate system, no consideration of individual agents

- Can consider other agents as part of a *nondeterministic* environment
- Does not consider goals and motivations of adversaries

- Incorporate other agents into unified **adversarial search**
- Will often have to operate real-time, accept suboptimal solutions

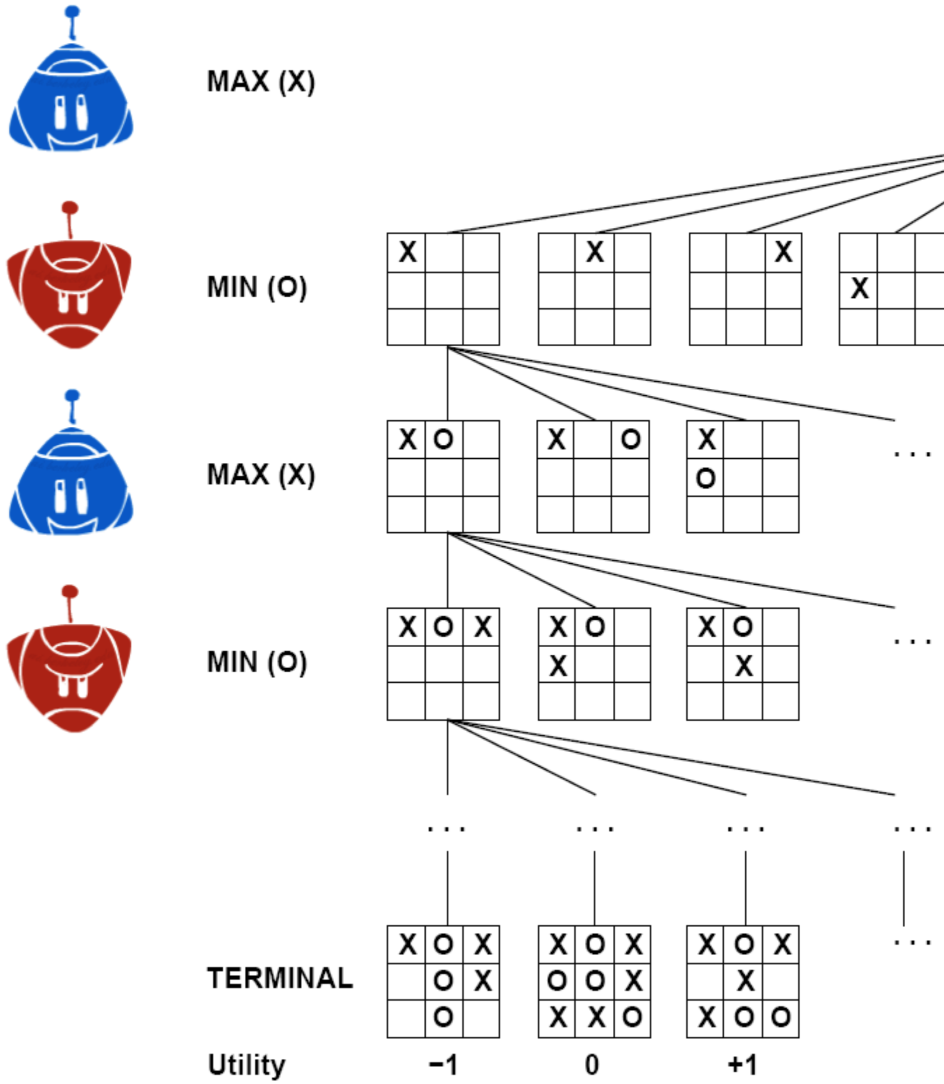
Two-player Zero-sum Games

- Simple adversarial search problem: Two agents, turn-taking, deterministic, perfect information (fully observable), zero-sum (p1 wins = p2 loses and vice-versa)

- States (positions): Current state of the game
- Actions: Set of legal moves in a state s
- Transition model: Mapping from (state, action) to a new state
- **Terminal test**: Is the current state a *terminal state* (is the game over)?

- **Utility function**: A player's "score" or payoff at a terminal state
- Since games are zero-sum, we should have $Utility(s, P1) = -Utility(s, P2)$
  - Ex: Tic-tac-toe. X has three in a row, $Utility(s, \mathrm{X}) = 1, Utility(s, \mathrm{O}) = -1$

Game Trees



- As with single-agent search, we can define state space graphs and search trees

- **Game tree**: A complete search tree from an initial state to a terminal state

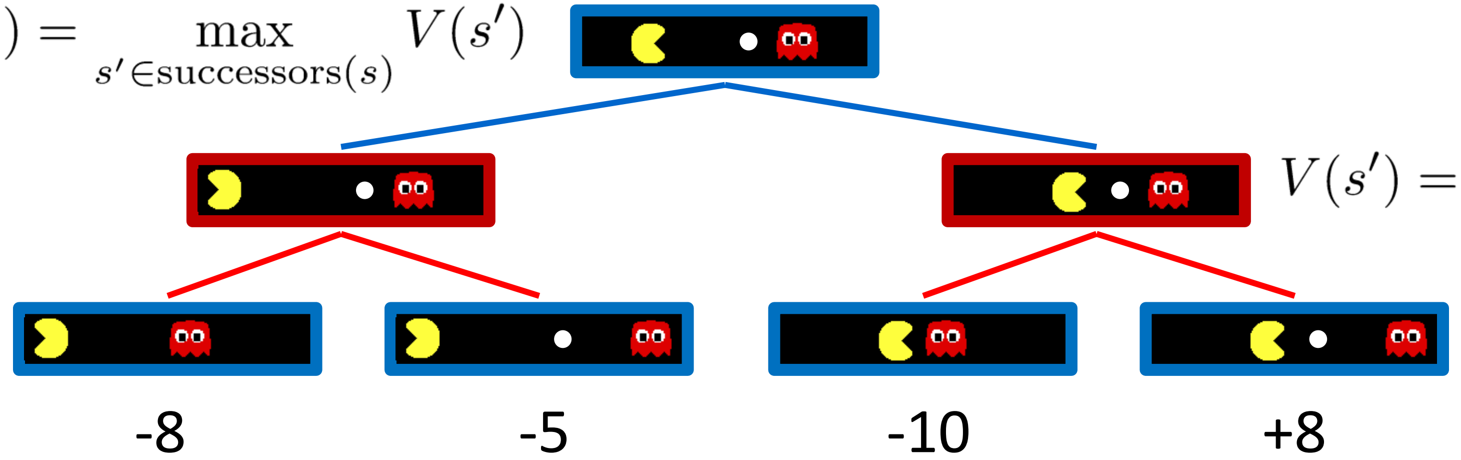
- Trees may be huge or infinite!
- Chess: Over 10^{108} nodes in the tree

State Utilities

- Both players want to maximize their own utilities
- Equivalently, we can just use MAX's utilities, which player MIN wants to minimize

- Each player should plan its move based on expectation of opponent
- If non-terminal states also have utilities, decision-making would be easy

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$



$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

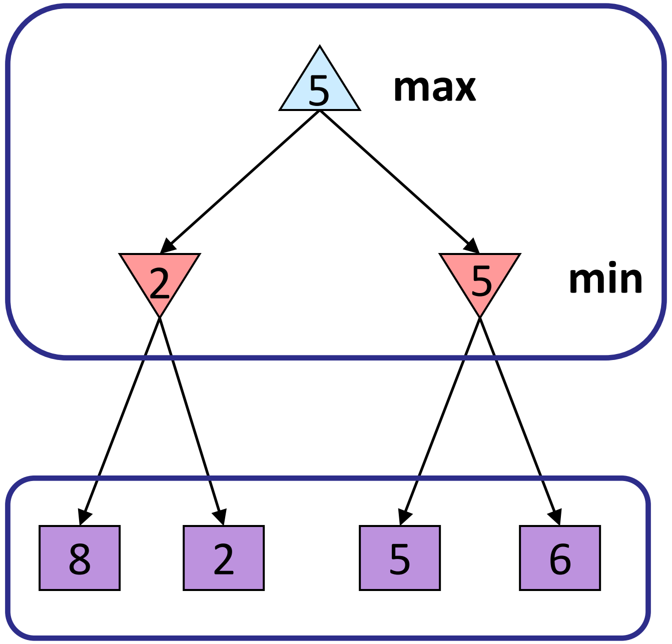
Terminal States: $V(s) = \text{known}$

Minimax Values

- Value of each state depends on each player's move
- If we know that each player plays optimally, the entire game tree's values are known!
- **Minimax value**: Utility of a state assuming *both* players play optimally until the end of the game

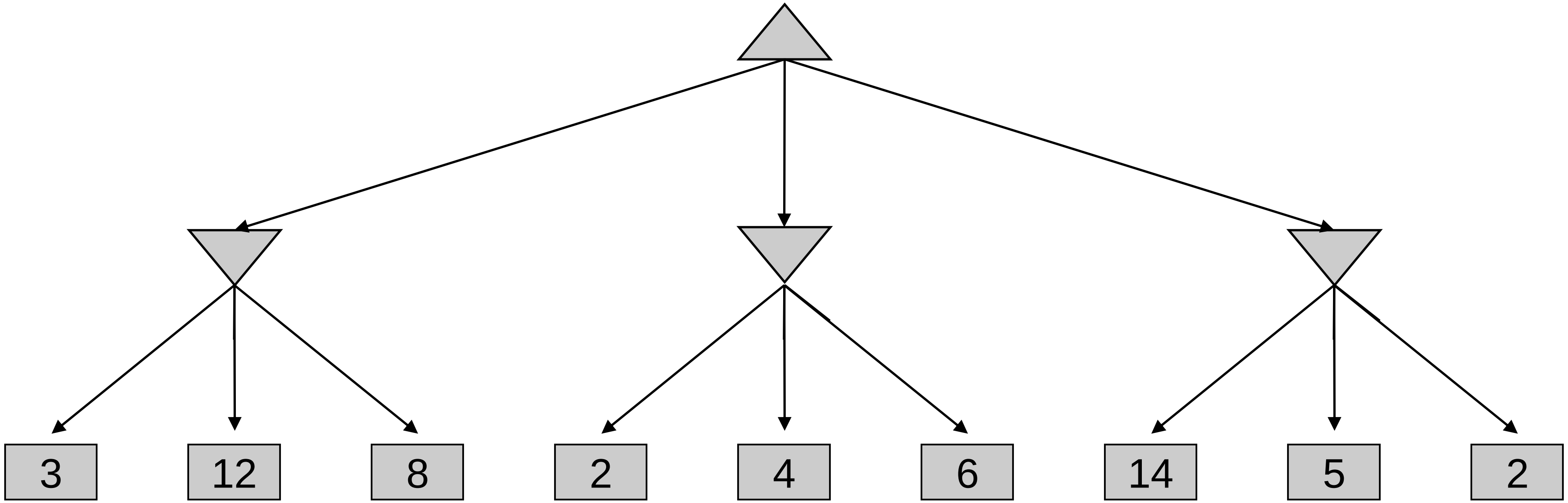
$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s)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

Minimax values: computed recursively



Terminal values: part of the game

Minimax Example



Minimax Search Algorithm

```
function MINIMAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) 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))
      if v2 > v then
         v, move ← 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 ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

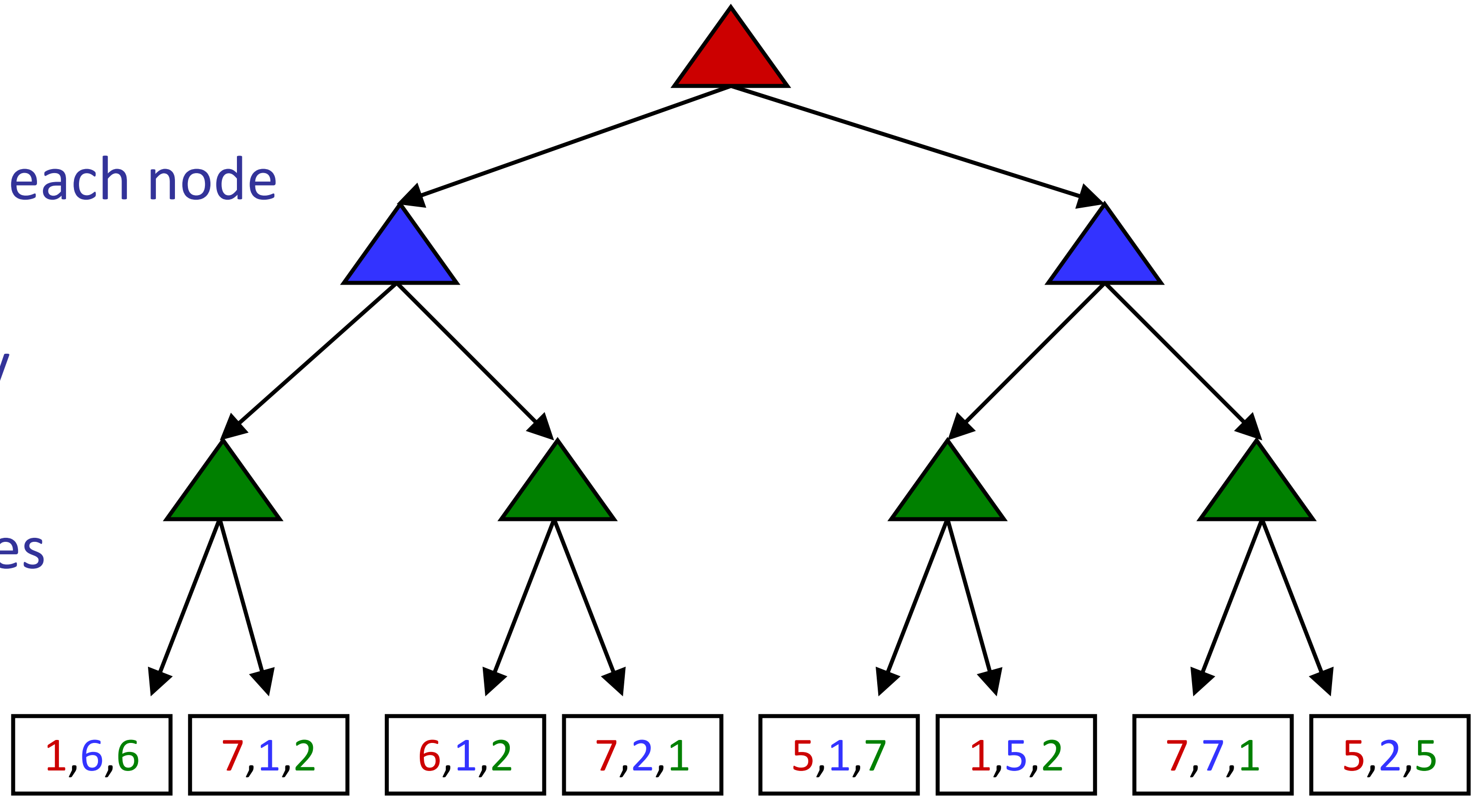
Assuming root is MAX

MAX calls MIN

MIN calls MAX

Multiplayer Games

- We can generalize minimax to non-zero-sum or multiplayer games
- Keep track of *vector* of utilities at each node
- Each player maximizes own utility
- Can produce cooperation, alliances



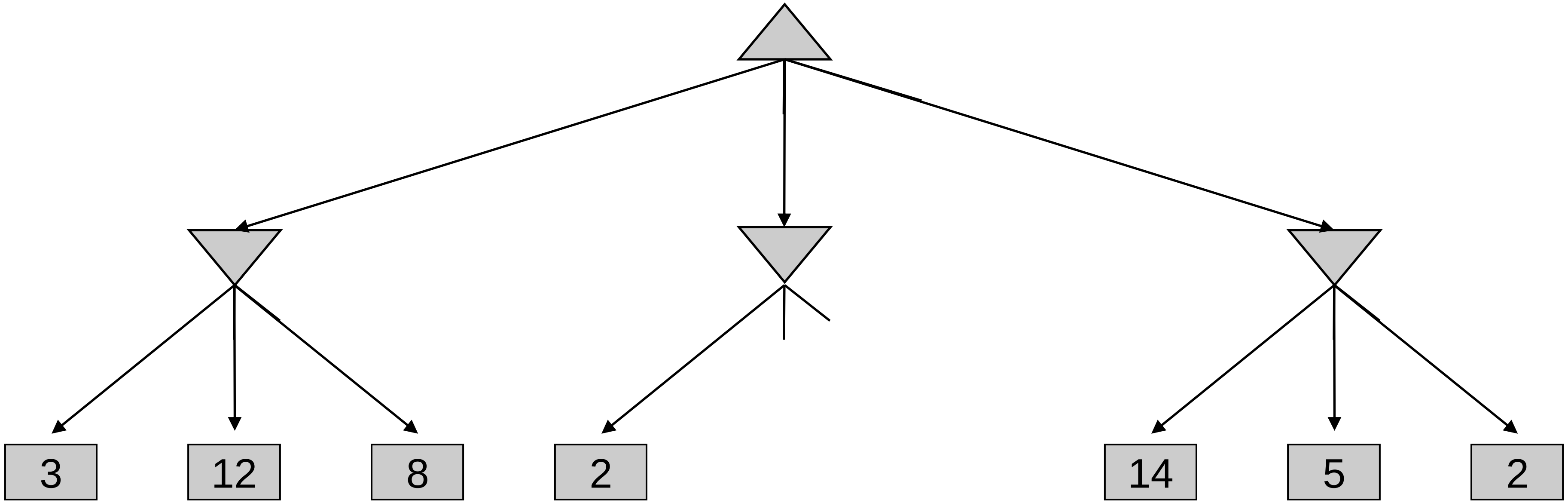
Improving Minimax

- Minimax executes a DFS-style search of the game tree
- Time complexity $O(b^d)$, space complexity $O(bd)$
- Optimal if both players play perfectly, complete if game eventually ends

- Example: Chess has $b \approx 35, d \approx 70$
- Completely infeasible to solve entirely!

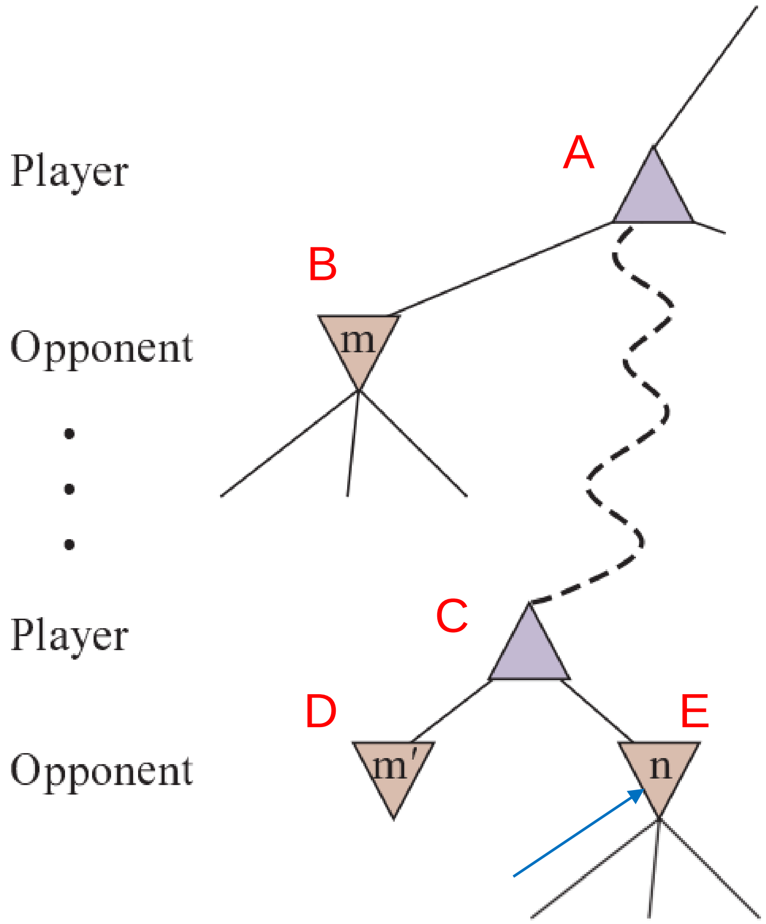
- Do we always need to expand entire tree?
- Improvements: Pruning, move ordering, cutting off search

Pruning



Pruning

- Best value for node E so far is n (from one of its children)
- If $n \leq m'$, MAX would never choose to go from C to E
  - $\max(m', \min(n, \dots), \dots) \geq m'$
- Similarly, if $n \leq m$ from node B, MAX would never choose the sequence of actions going from A to E
- In both cases, any remaining children of E should be pruned away

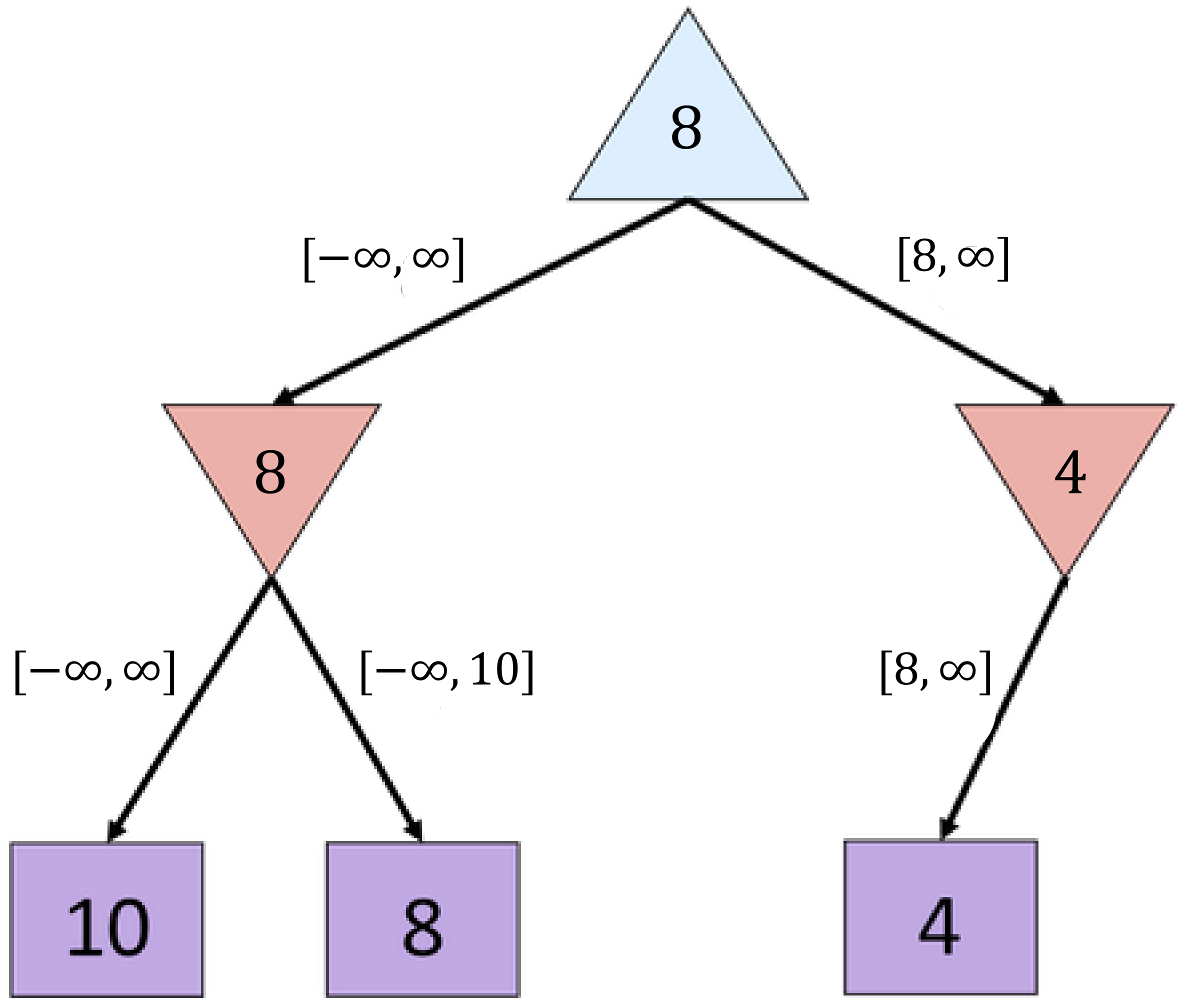


Alpha-Beta Pruning

- General idea: Keep track of highest (α) and lowest (β) values seen so far by MAX and MIN nodes, respectively

- Skip remaining children (prune) if:
- MAX sees value higher than β
- MIN sees value lower than α

- Know that root node would never choose path to current node



Alpha-Beta Search

function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
  player ← *game*.TO-MOVE(*state*)
  value, *move* ← MAX-VALUE(*game*, *state*, $-\infty$, $+\infty$)
  return *move*

Assuming root is MAX

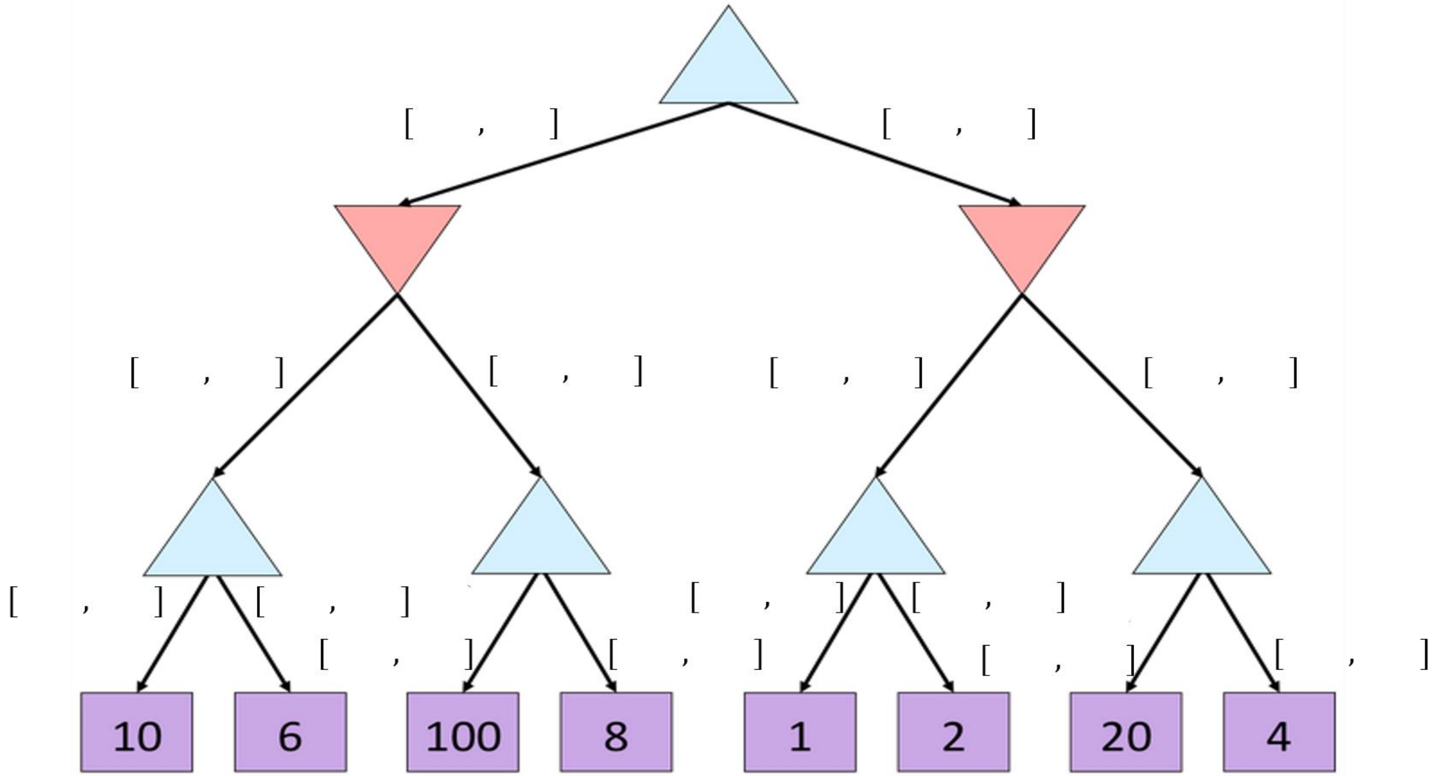
function MAX-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* ← MIN-VALUE(*game*, *game*.RESULT(*state*,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

MAX updates α, compares against β

function MIN-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  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*
      $\beta \leftarrow$ MIN(β, v)
    if $v \leq \alpha$ **then return** *v*, *move*
  return *v*, *move*

MIN updates β, compares against α

Alpha-Beta Example



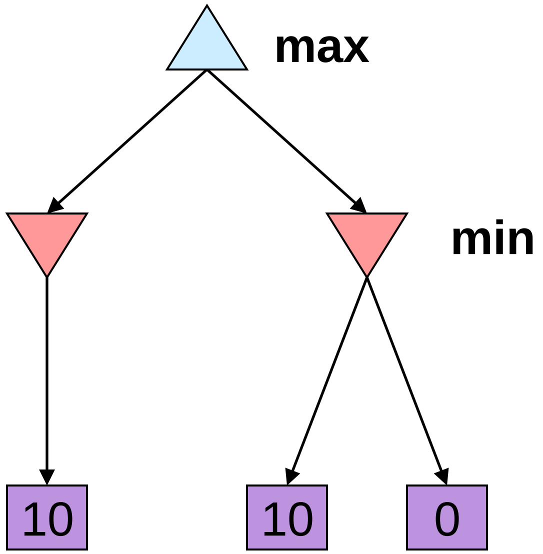
- Applet: http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html

Alpha-Beta Properties

- Pruning does not change true minimax value of the 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 ~$O(b^{0.75d})$

- "Perfect ordering" gets us to $O(b^{0.5d})$, doubling solvable depth
- Usually requires domain knowledge
  - Simple chess ordering function: captures, threats, forward moves, backward moves
  - Try moves first that were good in past moves

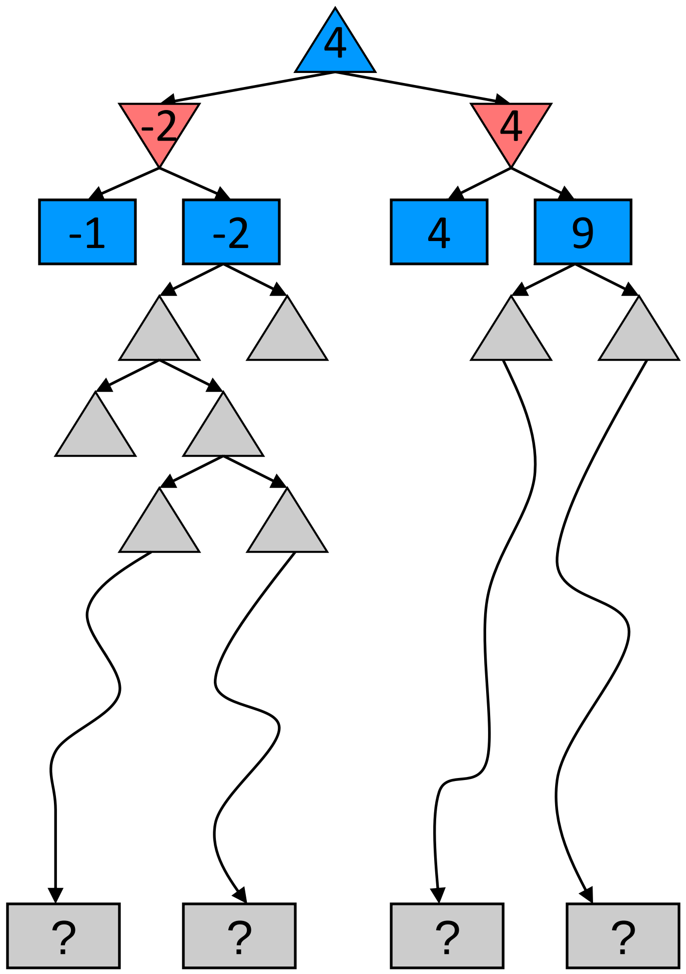
- If using depth-limited search, iterative deepening can also inform move ordering
- Evaluations at depth 1 inform ordering at depth 2, those inform ordering at depth 3, and so on

Imperfect Decisions

- Problem: Most game trees still too big
- α-β can help but still need to find terminal nodes

- Heuristic: Turn non-terminal nodes into terminals!
- **Evaluation function** returns an *estimate* of this "terminal" state's utility

- **Cutoff test** decides when to do this
- No more guarantee of optimality



Evaluation Functions

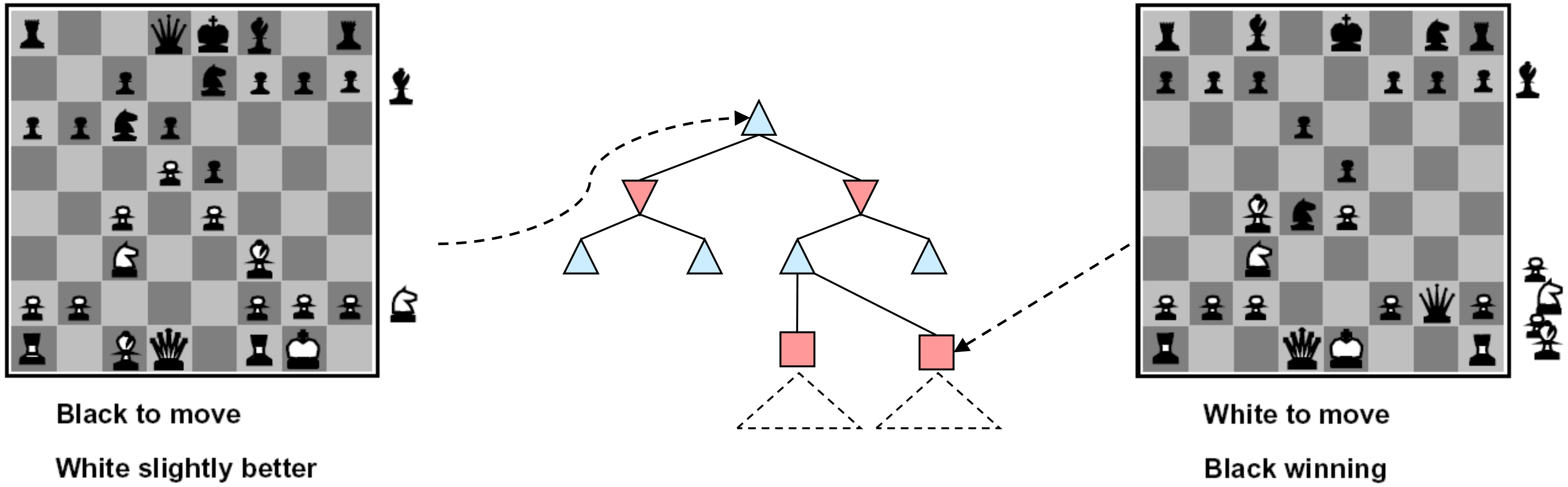
- Evaluation functions are *estimates* of a state's utility

- Agent's performance depends strongly on eval function quality
  - Evaluation of terminals should be the same as that of true utility function
  - Evaluation of non-terminals should have *some* correlation with winning
  - Computation must be efficient

- One common eval function: weighted linear sum of game **features**

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

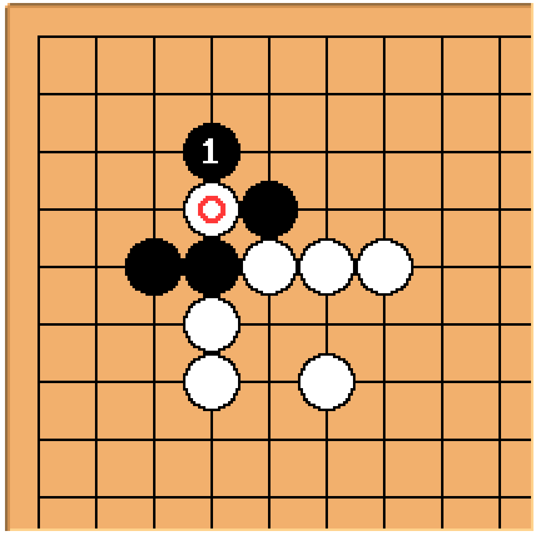
Example: Chess

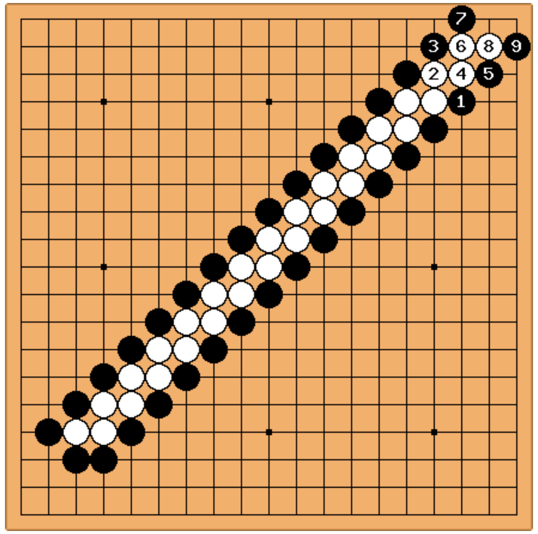


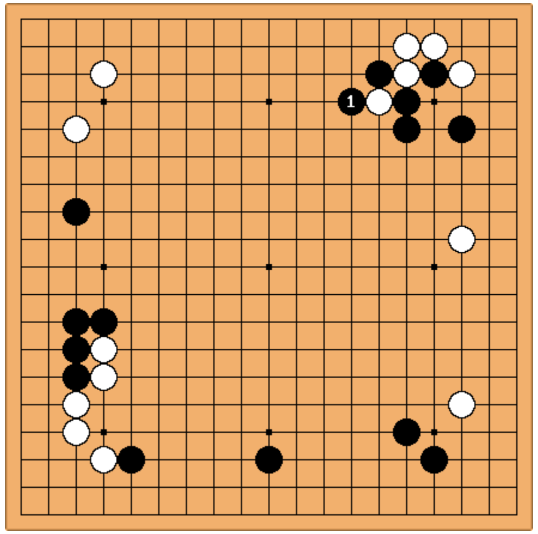
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Features may be derived from expert knowledge of common *categories* of states
- E.g., one feature for each type of piece, attack formations, king safety positions, etc.
- Weights correspond to *material values* of each feature
- Linear weighting assumes features are independent of each other

Example: Go







$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Example features in Go:
  - Num white pieces – num black pieces
  - Buildup of potential “ladders”
  - Territorial “spread” of pieces

Depth Limits

- Now that we have eval function, when to cut off search and apply it?
- Simple approach: Use a fixed depth limit, or use iterative deepening
- Transposition table especially effective with latter approach

- Problem: Cutting off search at volatile positions can lead to loss of information
- **Horizon effect**: Agent may favor moves that push danger "over the horizon", appears to have been mitigated but actually just delayed

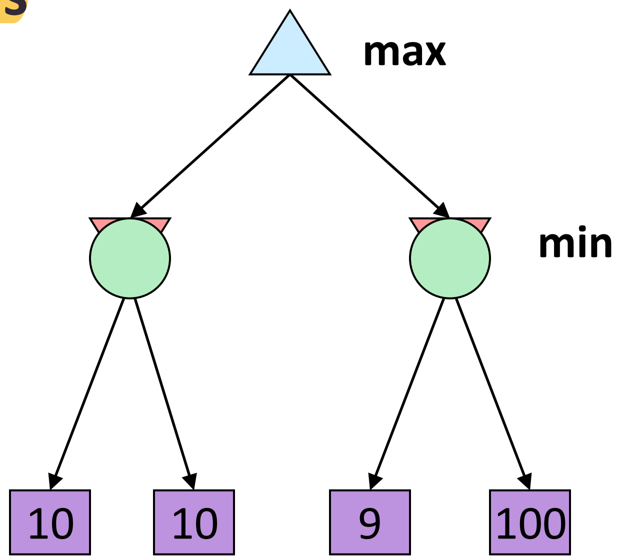
- If possible, only cut off search at **quiescent** (quiet) positions
- **Quiescence search**: Extend search at volatile positions until we find quieter ones
- "Adaptive" search depth based on domain knowledge

Stochastic Games

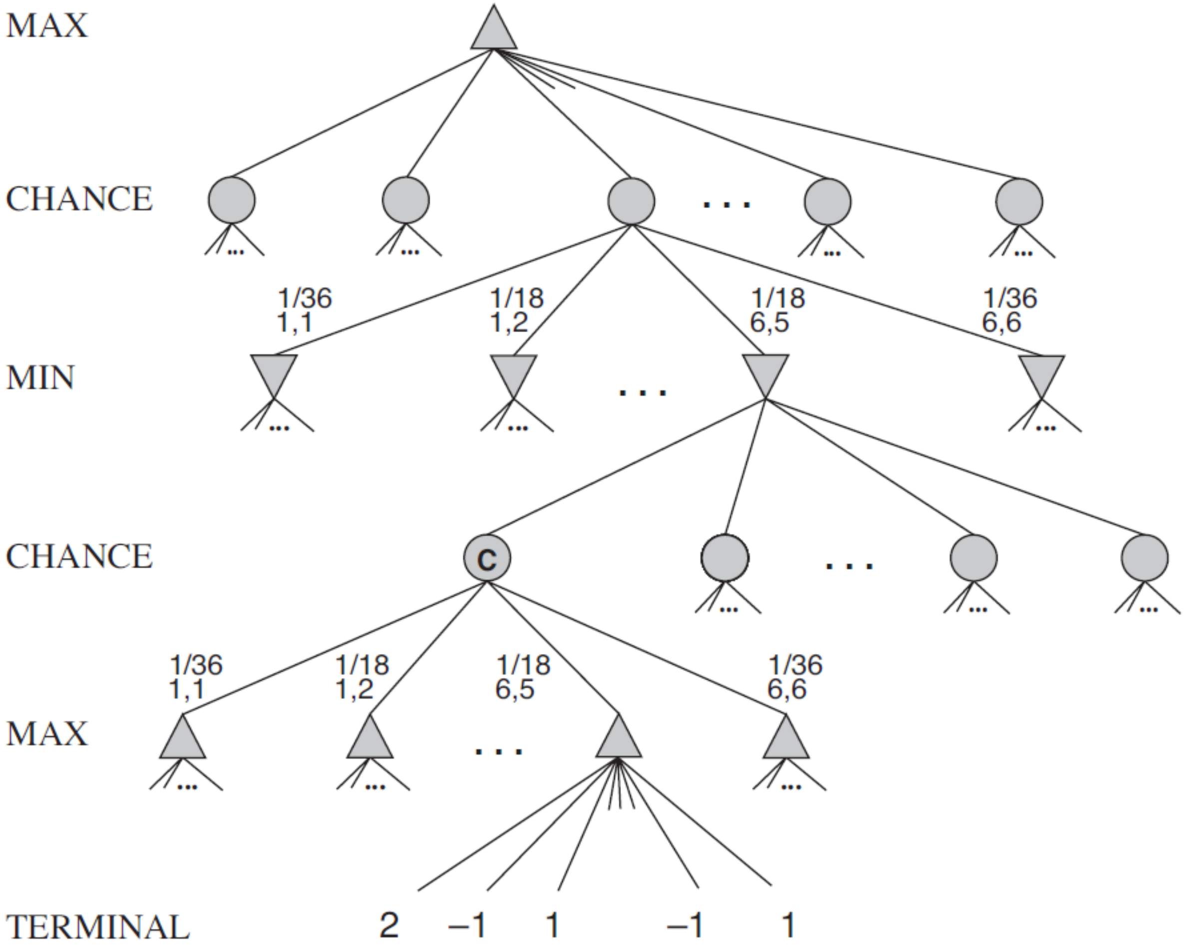
- Many games contain *stochasticisty*
- Opponents playing suboptimally (e.g., inexperienced) or randomly
- Explicit random elements (e.g., dice rolling)

- Instead of worst-case scenarios, we consider **expected values**
- Chance nodes have **expectiminimax values**

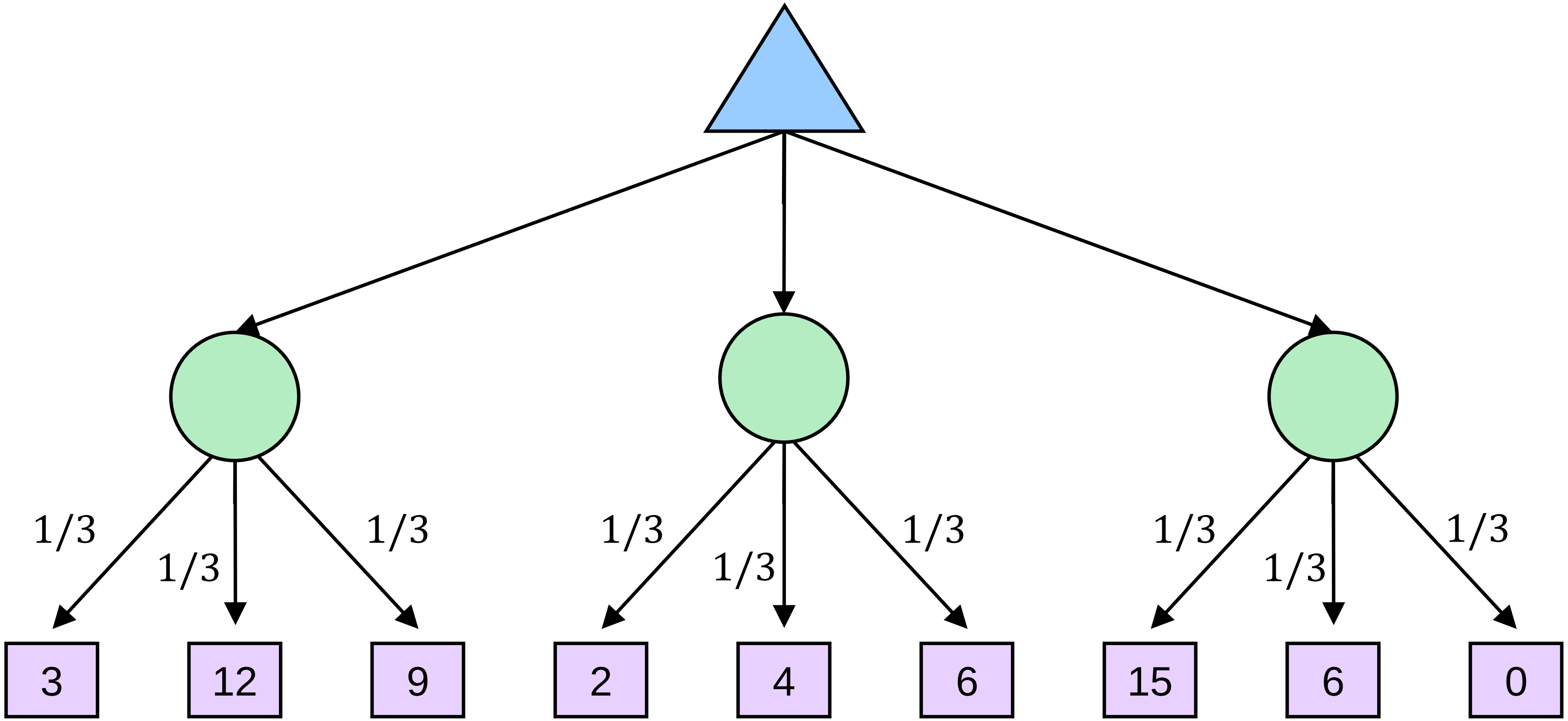
$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$



Example: Backgammon



Expectiminimax Example

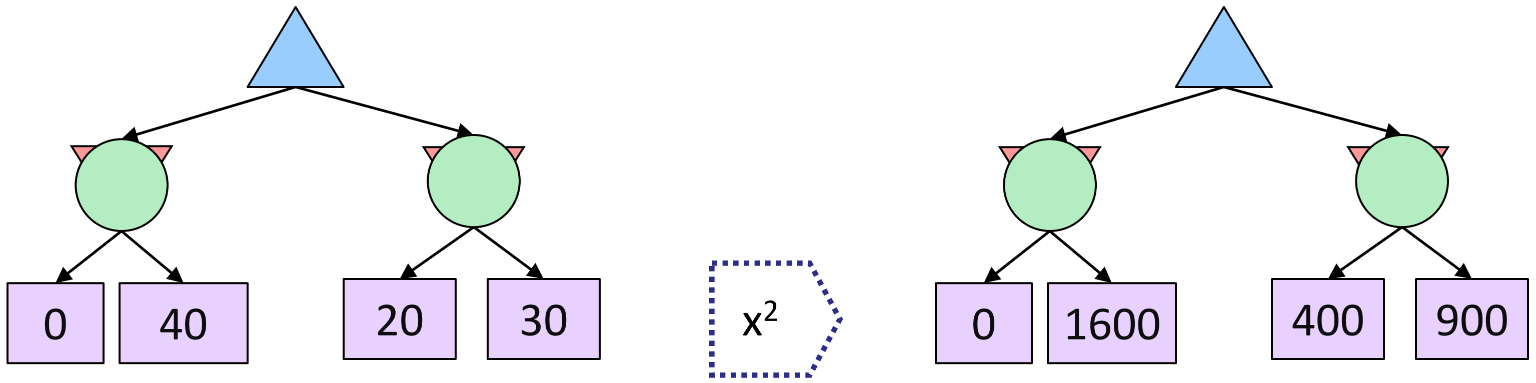


Expectiminimax Utilities

- Decisions in stochastic games are sensitive to choice of eval function

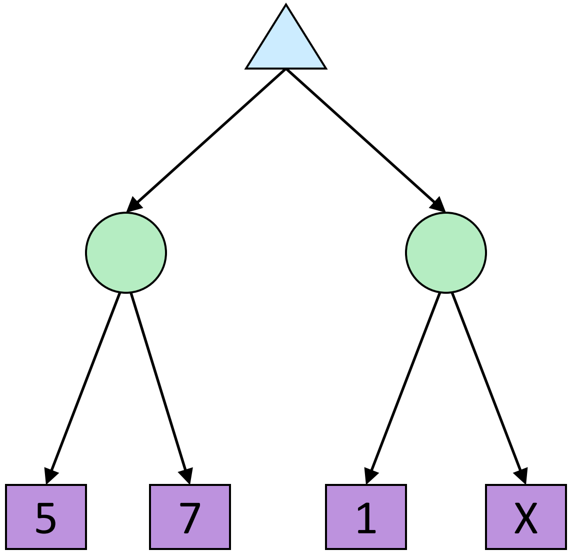
- Minimax: Decisions unchanged as long as relative ordering of node values is the same
- Expectiminimax: No guarantee that decisions stay the same if values change

- To fix, must ensure that eval function is a *positive linear transformation* of true utilities



Expectiminimax Pruning

- Pruning is more difficult than in deterministic games
- Cannot eliminate possibilities if we cannot predict results of stochastic transitions
- Pruning *can* occur if we have explicit bounds on utility values overall
- If we can accept imperfect decisions, we can perform forward pruning to examine fewer branches



All utilities ≤ 10

Monte Carlo Tree Search*

- Minimax and its variants are mostly **type A** strategies: search wide but shallow

- For games like Go with large branching factors, **type B** strategies like MCTS work better: search deep but narrow

- Idea: Run many simulations from current game state to a terminal
- No heuristic evaluations; we record end game results in all those simulations
- Results are tabulated and averaged; moves are chosen to maximize chances of winning

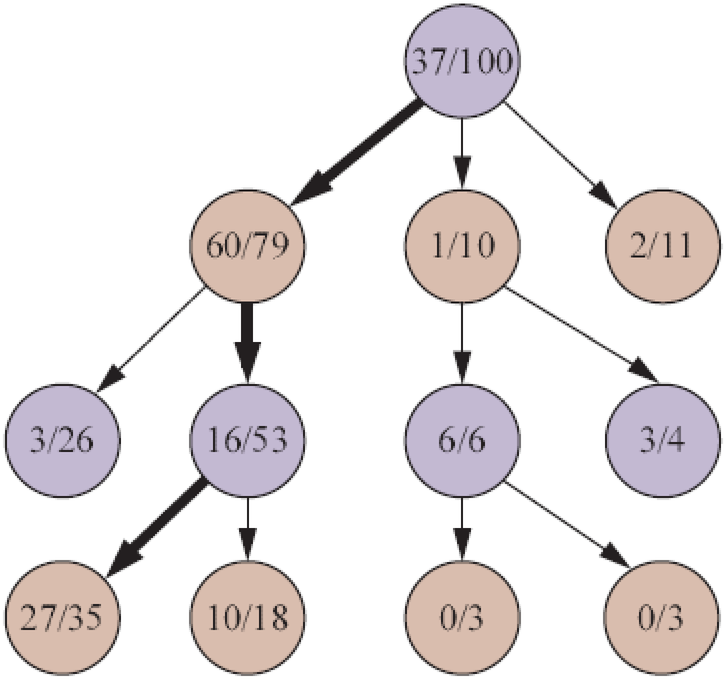
- Simulations may be informed by **selection policy** and **playout policy**

Selection Policy

- We grow a tree starting with current game state at the root
- Keep track of how many plays and wins we have simulated from each node

- A predetermined **selection policy** guides us down the tree to a leaf

- Goal: Obtain more simulation results for promising nodes (*exploitation*) but also for lesser known nodes (*exploration*)

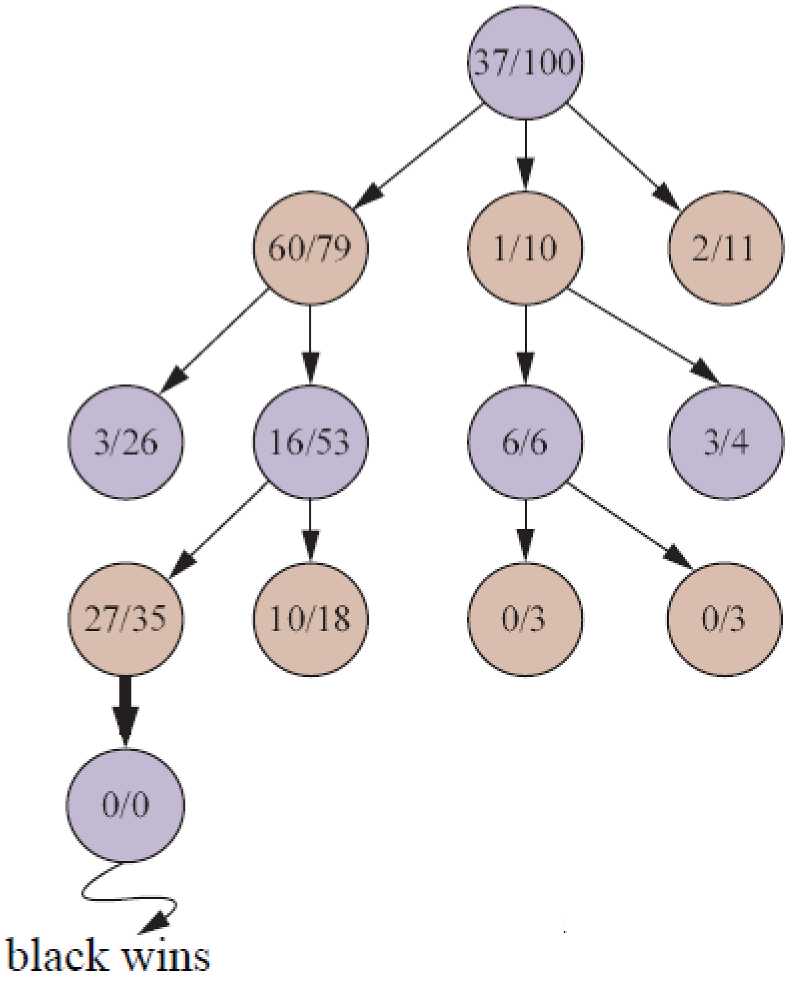


Playout Policy

- From selected leaf, add a single child node to the tree
- Simulate a complete game from the child state
- Encountered game states are *not* recorded!

- How do we perform a simulation in our head?
- Follow a predetermined **playout policy**

- Generally biases moves toward good or clever ones
- May incorporate game-specific heuristics
- May be obtained from deep learning (e.g. AlphaGo)

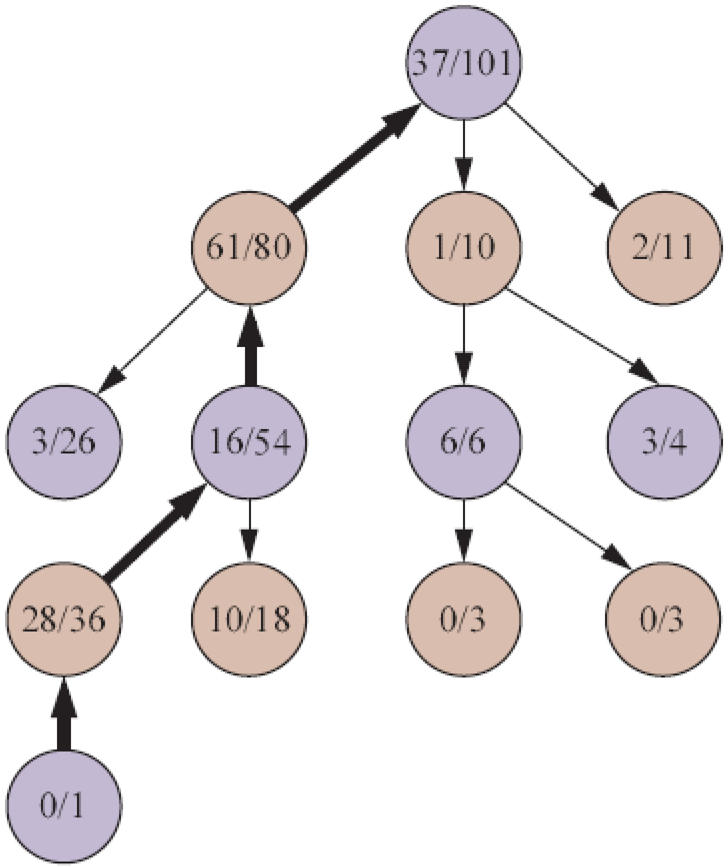


Backpropagation

- Once a simulation is done, the result is returned up the tree
- All nodes along path get updated

- Process iterates between growing and updating search tree
- When done, return the most simulated move

- Why not return move with highest win rate?
- We must also be wary of uncertainty!



MCTS vs Alpha-Beta

- MCTS simulations are linear in game depth
- For a game with branching factor of 32 and depth of 100, alpha-beta search down to 12 ply deep is equivalent to 10^7 MCTS simulations

- MCTS tends to do better when branching factor is high
- Also less sensitive than α-β to inaccurate eval functions
- Also good for brand new games with no predefined eval functions at all!

- Stochastic nature of MCTS: no guarantee of exploring all good moves

Summary

- Adversarial search problems consider the goals of multiple agents
- Two-player zero-sum games can be solved via minimax search
- Improvements: Alpha-beta pruning, move ordering, transposition tables, evaluation functions and depth limits for imperfect decisions
- Can be extended to stochastic games with expectiminimax search
- Generally much harder, need to consider many more search possibilities