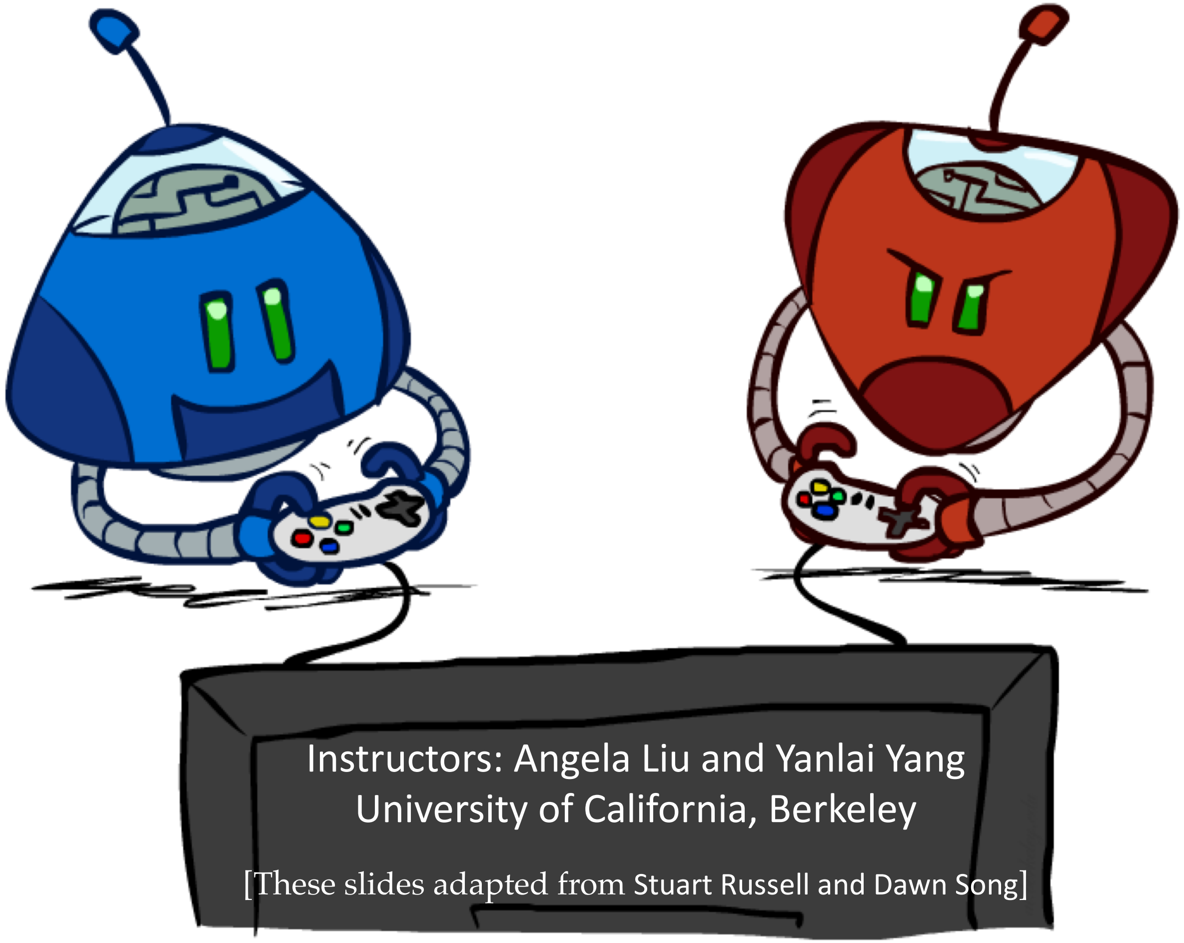
CS 188: Artificial Intelligence

Adversarial Search II



Instructors: Angela Liu and Yanlai Yang
University of California, Berkeley

[These slides adapted from Stuart Russell and Dawn Song]

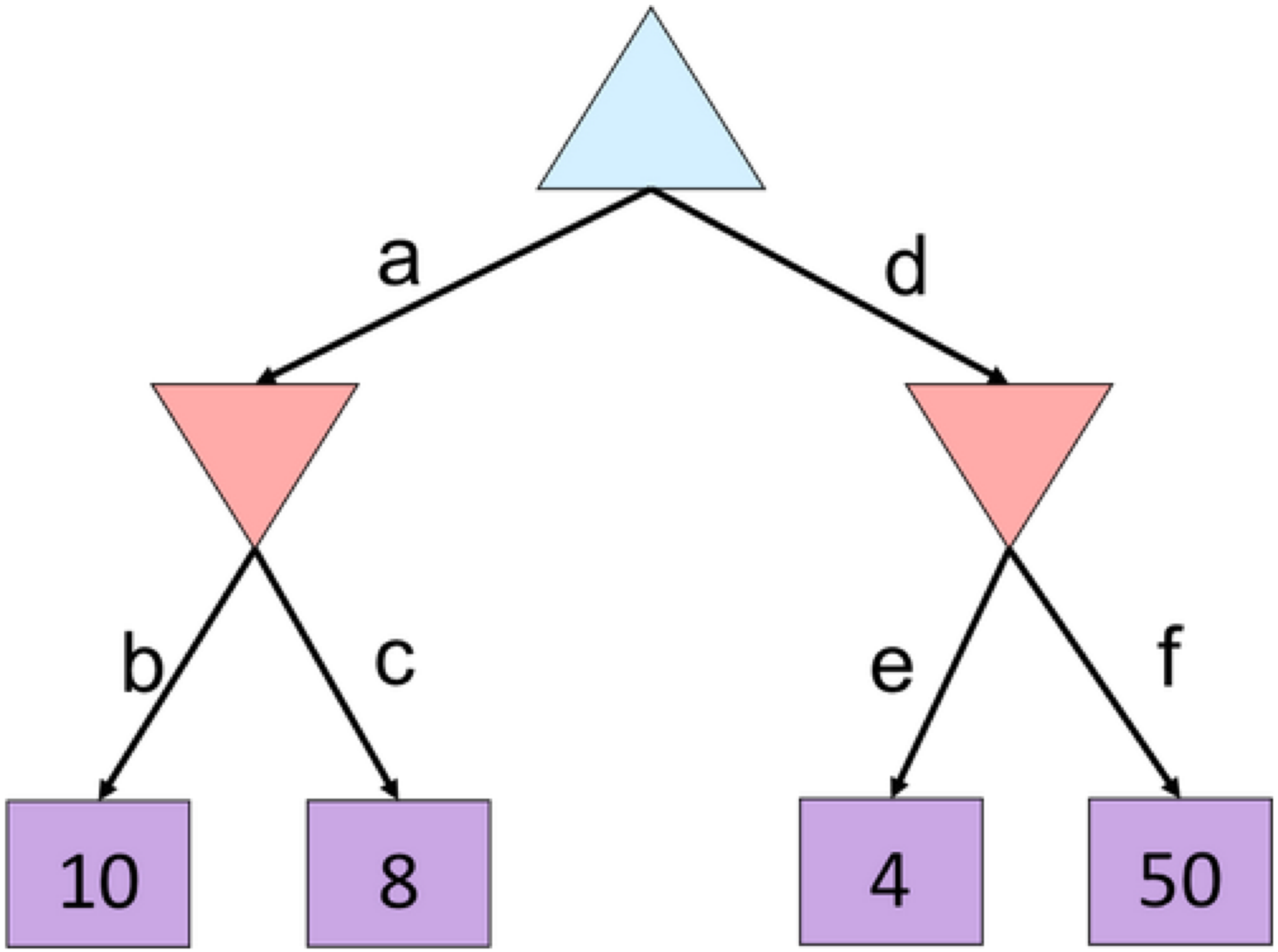
Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor, α,β))
        if v ≥ β
            return v
        α = max(α, v)
    return v
```

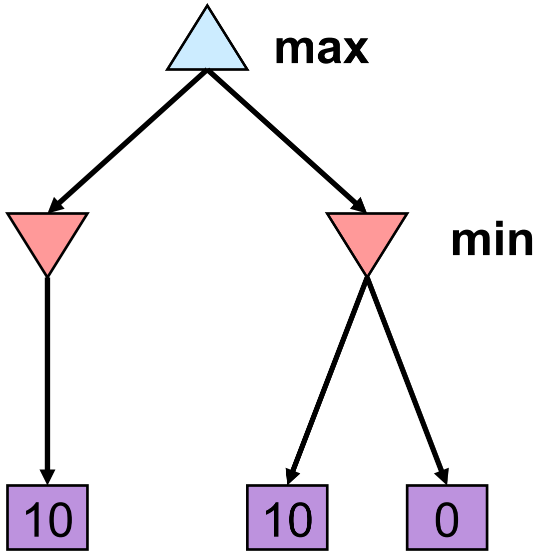
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor, α, β))
        if v ≤ α
            return v
        β = min(β, v)
    return v
```

Alpha-Beta Quiz



Alpha-Beta Pruning Properties

- Theorem: This pruning has ***no effect*** on minimax value computed for the root!

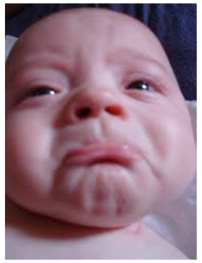


- Good child ordering improves effectiveness of pruning
  - Iterative deepening helps with this

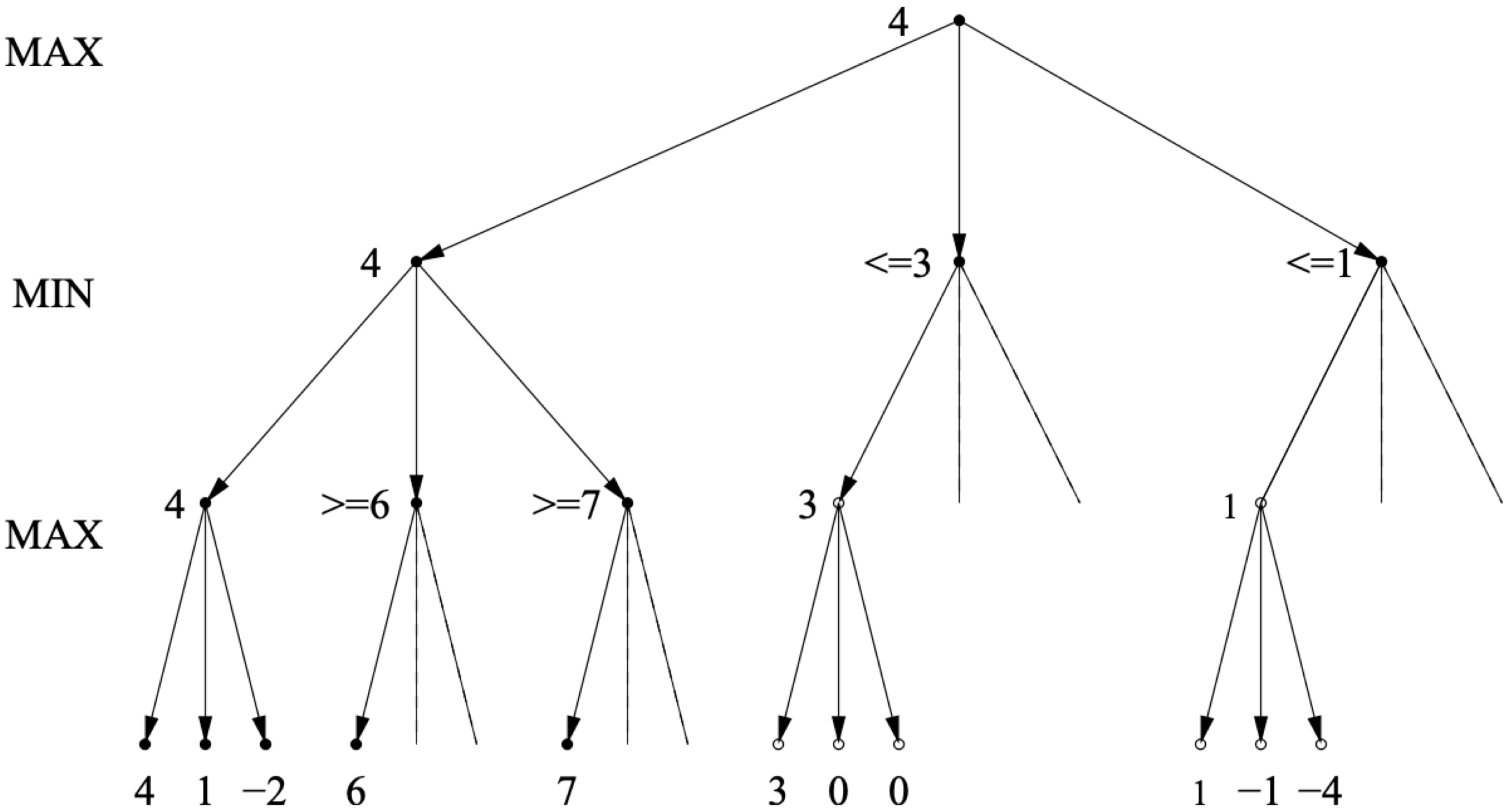
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!

- This is a simple example of metareasoning (reasoning about reasoning)

- For chess: only 35^{50} instead of 35^{100}!! Yaaay!!!!!



Alpha-Beta Best-Case Analysis



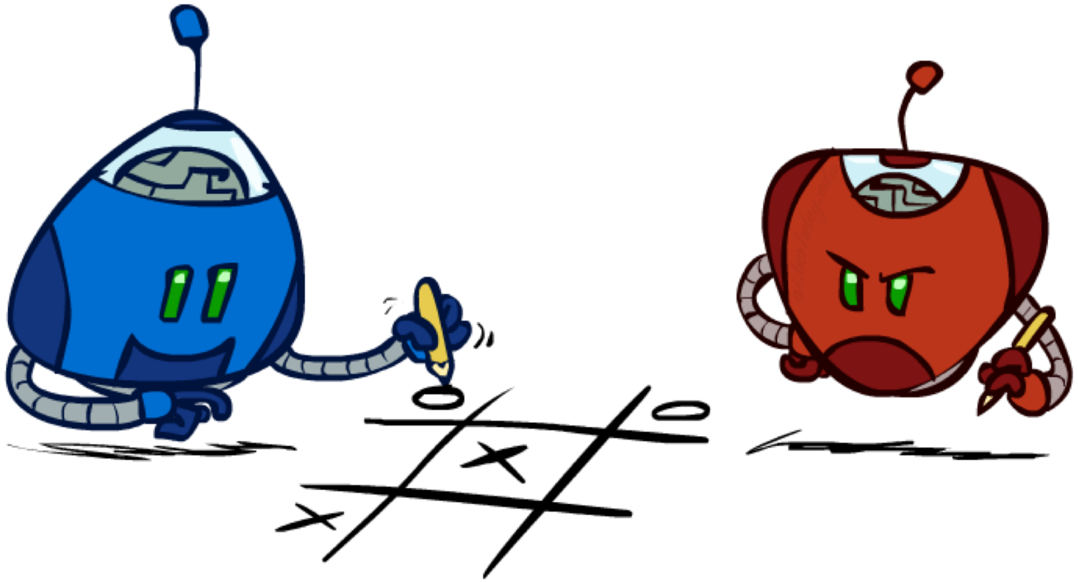
http://www.cs.utsa.edu/~bylander/cs5233/a-b-analysis.pdf

Summary

- Games are decision problems with $\geq$ 2 agents
  - Huge variety of issues and phenomena depending on details of interactions and payoffs
- For zero-sum games, optimal decisions defined by minimax
  - Simple extension to n-player "rotating" max with vectors of utilities
  - Implementable as a depth-first traversal of the game tree
  - Time complexity $O(b^m)$, space complexity $O(bm)$
- Alpha-beta pruning
  - Preserves optimal choice at the root
  - Alpha/beta values keep track of best obtainable values from any max/min nodes on path from root to current node
  - Time complexity drops to $O(b^{m/2})$ with ideal node ordering
- Exact solution is impossible even for "small" games like chess

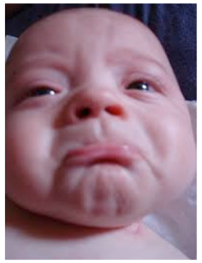
Outline

- Finite lookahead and evaluation
- Games with chance elements
- Monte Carlo tree search



The story so far…

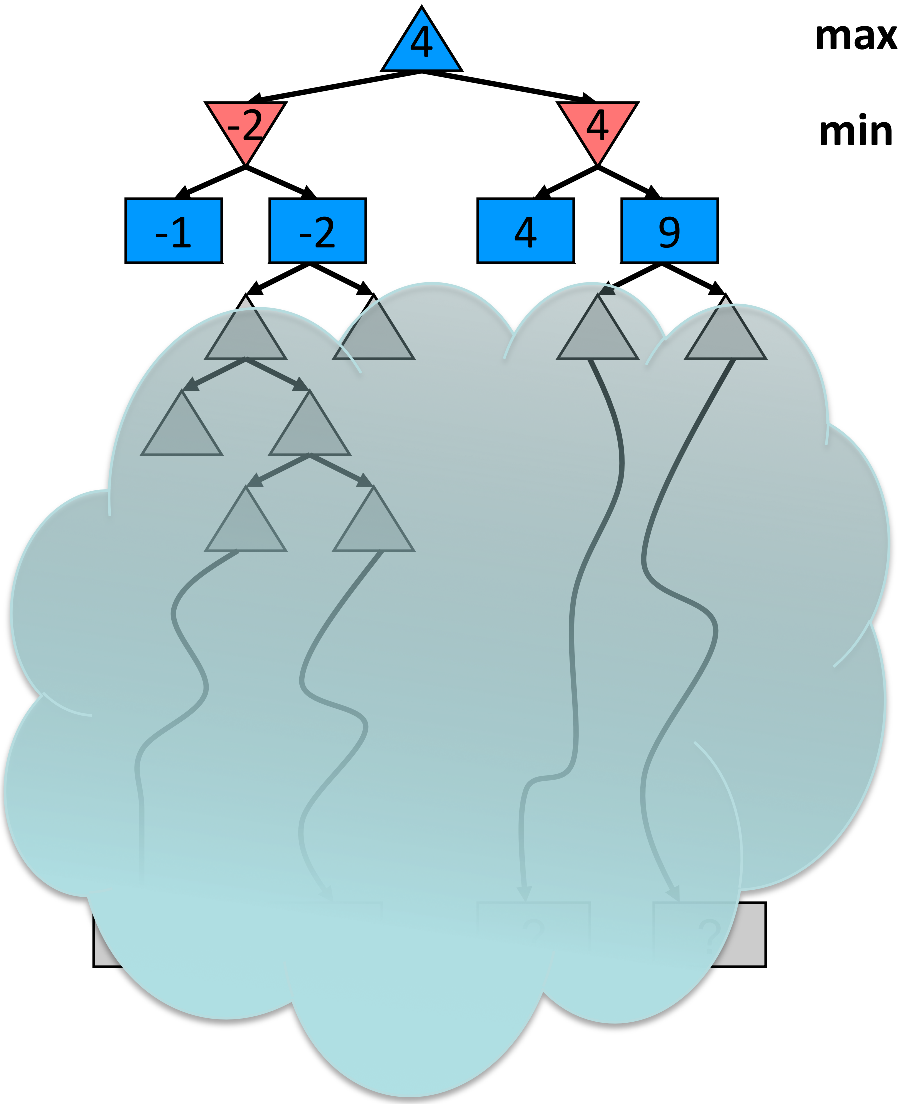
- Focus on two-player, zero-sum, deterministic, observable, turn-taking games
- Minimax defines rational behavior
- Recursive DFS implementation: space complexity $O(bm)$, time complexity $O(b^m)$
- Alpha-beta pruning with good node ordering reduces time complexity to $O(b^{m/2})$
- Still nowhere close to solving chess, let alone Go or StarCraft



Resource Limits

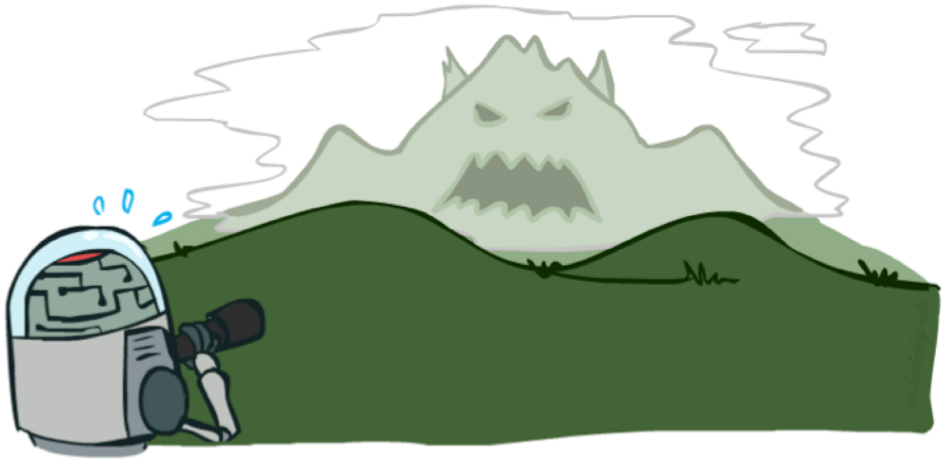
Resource Limits

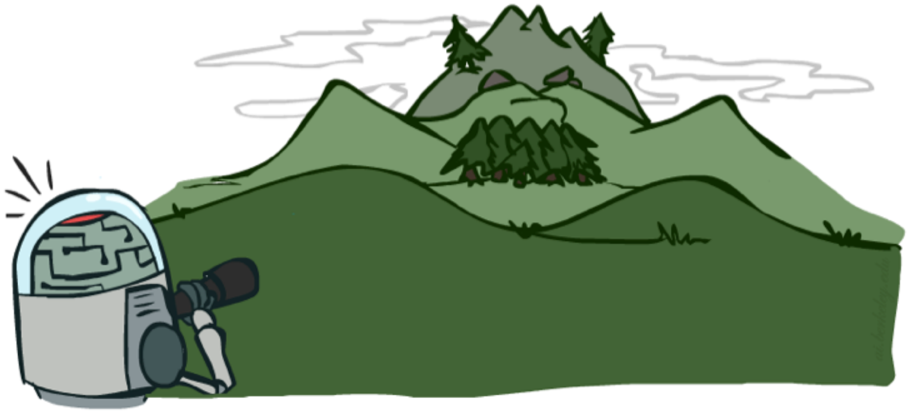
- Problem: In realistic games, cannot search to leaves!
- Solution (Shannon, 1950): Bounded lookahead
  - Search only to a preset ***depth limit*** or ***horizon***
  - Use an ***evaluation function*** for non-terminal positions
- Guarantee of optimal play is gone
- Example:
  - Suppose we can explore 1M nodes per move
  - Chess with alpha-beta, $35^{(8/2)}$ =~ 1M; depth 8 is quite good



Depth Matters

- Evaluation functions are always imperfect
- Deeper search => better play (usually)
- Or, deeper search gives same quality of play with a less accurate evaluation function
- An important example of the tradeoff between complexity of features and complexity of computation





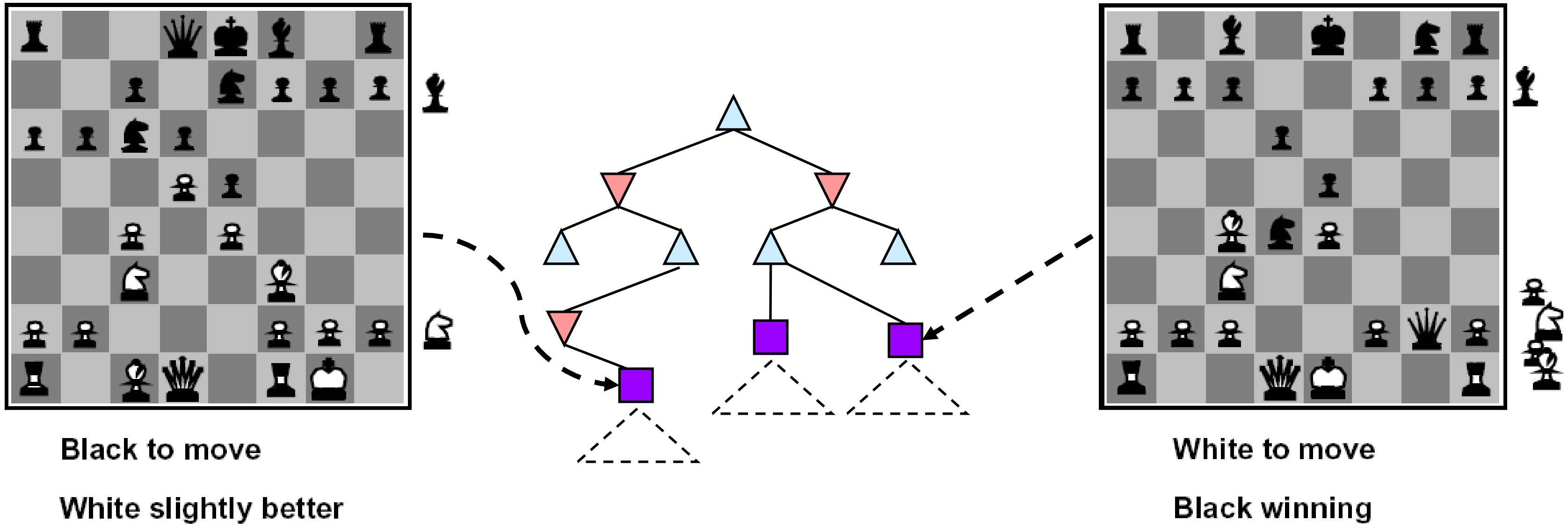
[Demo: depth limited (L6D4, L6D5)]

Pacman with Depth-2 Lookahead

Pacman with Depth-10 Lookahead

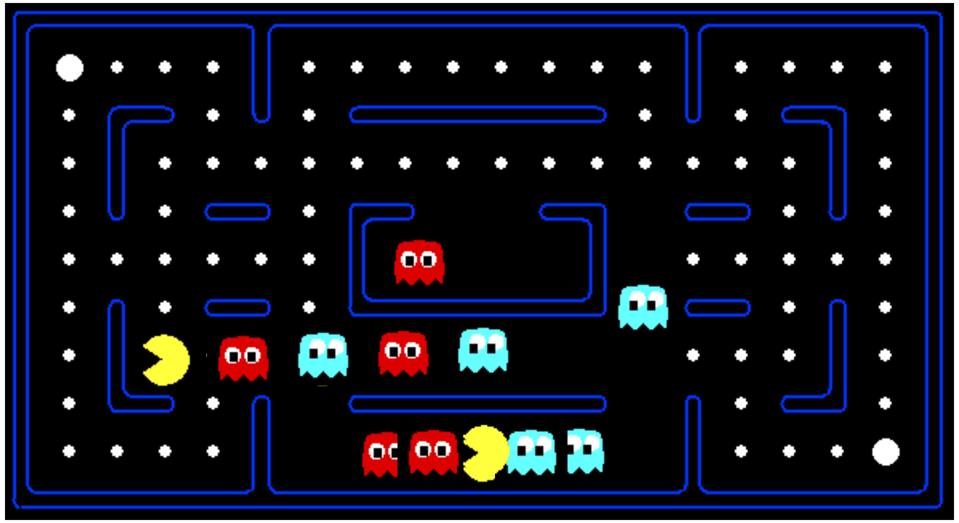
Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search

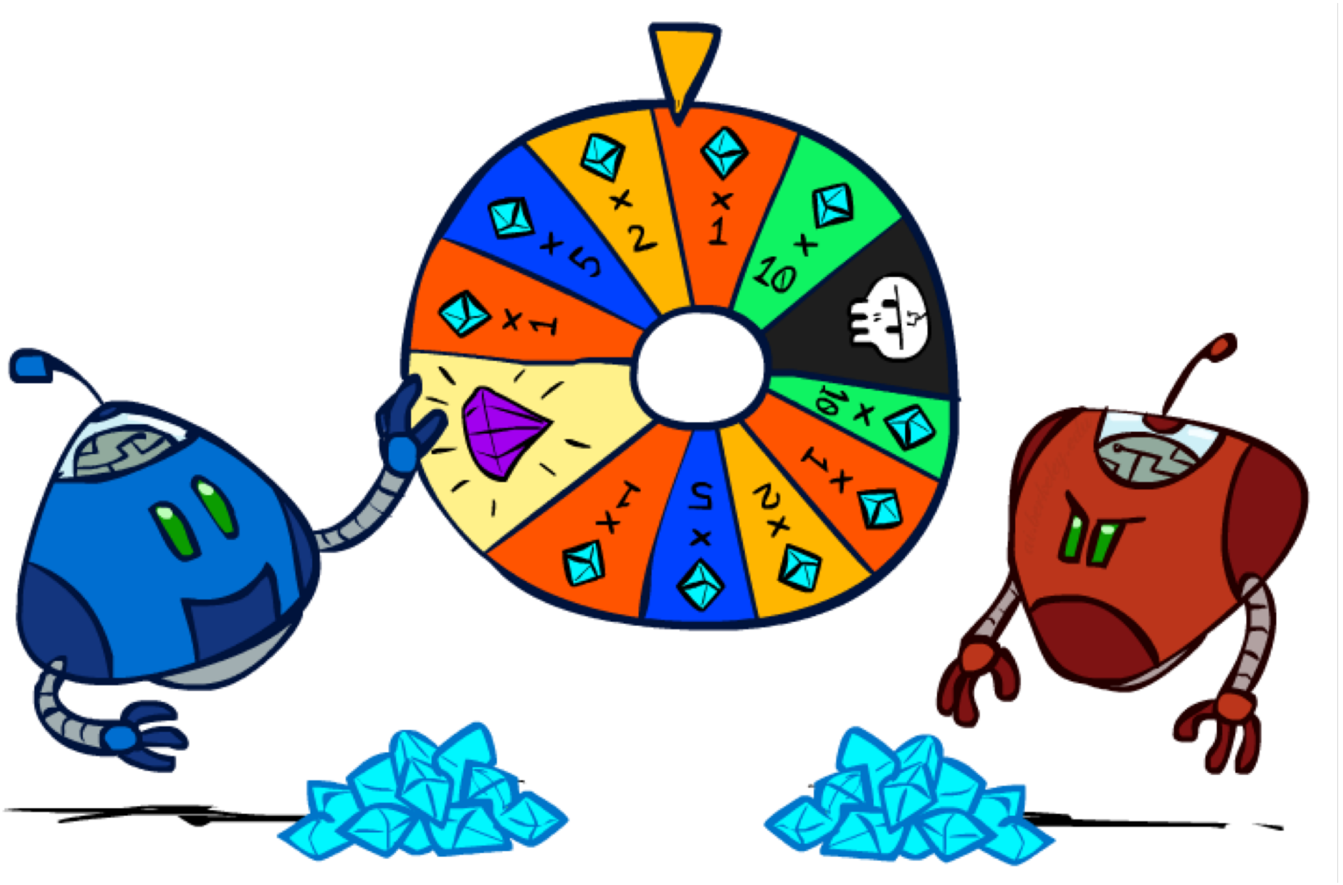


- Typically weighted linear sum of features:
  - EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \ldots. + w_n f_n(s)$
  - E.g., $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.
- Or a more complex nonlinear function (e.g., NN) trained by self-play RL
- Terminate search only in ***quiescent*** positions, i.e., no major changes expected in feature values

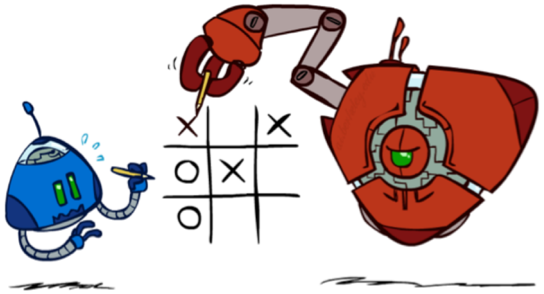
Evaluation for Pacman

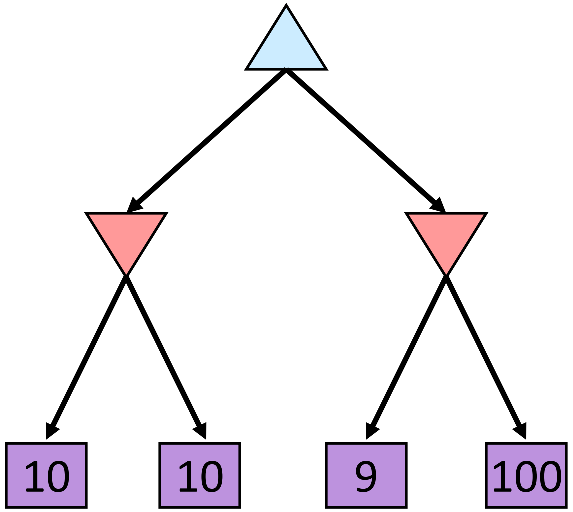


Games with uncertain outcomes



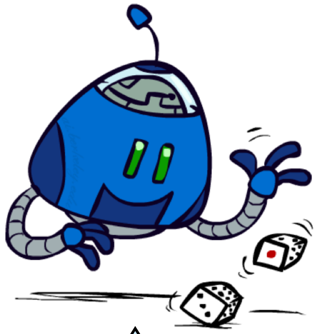
Chance outcomes in trees

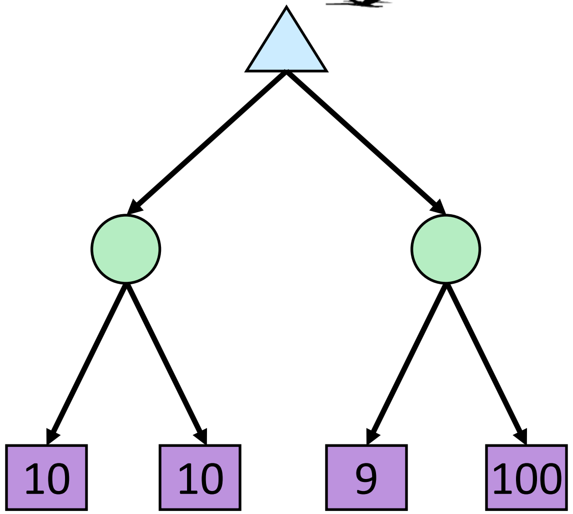




Tictactoe, chess

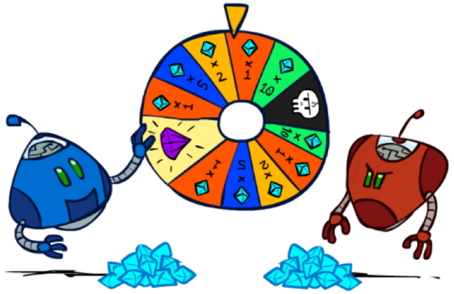
Minimax

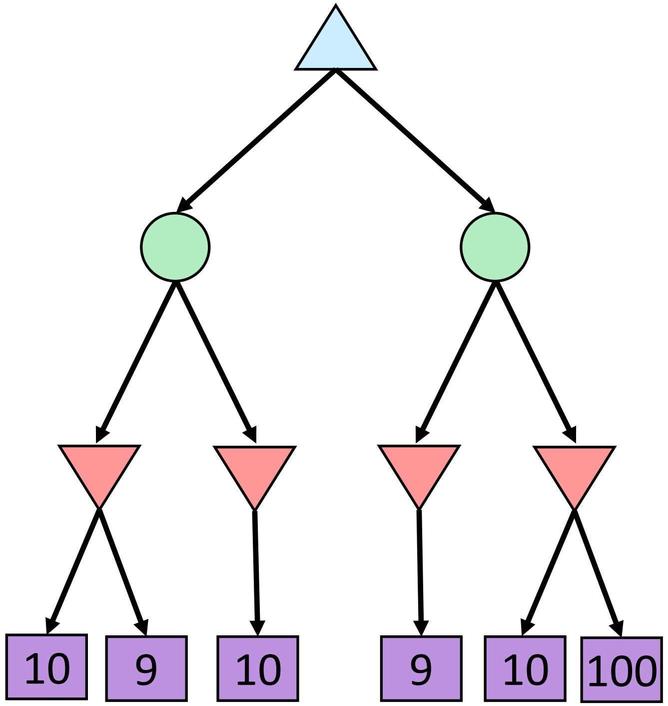




Tetris, roulette

Expectimax





Backgammon, Monopoly

Expectiminimax

Minimax

function decision(s) returns an action

    return the action a in Actions(s) with the highest
     minimax_value(Result(s,a))

function minimax_value(s) returns a value
    if Terminal-Test(s) then return Utility(s)
    if Player(s) = MAX   then return $\max_{a \text{ in Actions}(s)}$ minimax_value(Result(s,a))
    if Player(s) = MIN   then return $\min_{a \text{ in Actions}(s)}$ minimax_value(Result(s,a))

Expectiminimax

```
function decision(s) returns an action
    return the action a in Actions(s) with the highest
     value(Result(s,a))
```

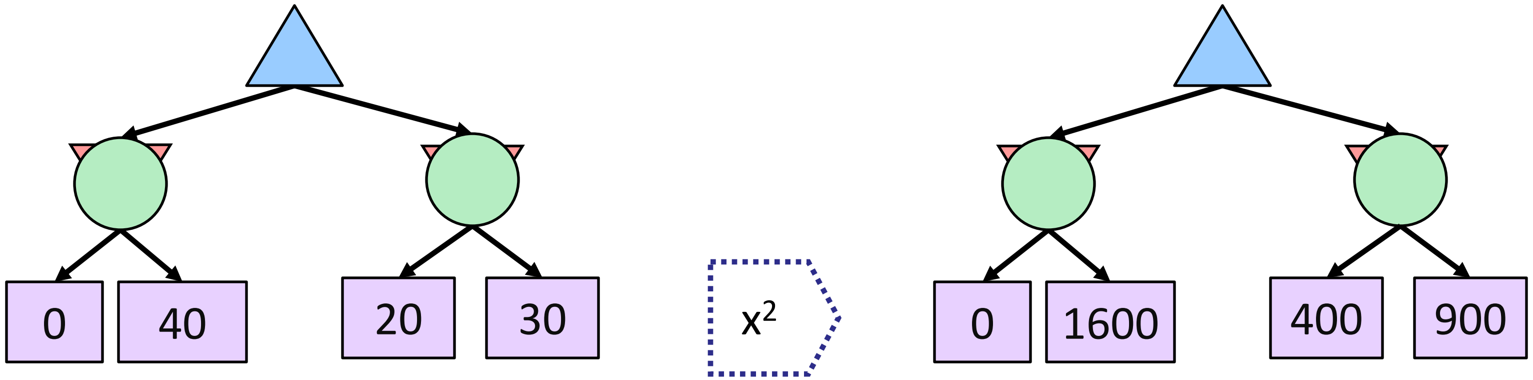
```
function value(s) returns a value
    if Terminal-Test(s) then return Utility(s)
    if Player(s) = MAX      then return max_{a in Actions(s)} value(Result(s,a))
    if Player(s) = MIN      then return min_{a in Actions(s)} value(Result(s,a))
    if Player(s) = CHANCE then return sum_{a in Actions(s)} Pr(a) * value(Result(s,a))
```

Example: Backgammon

- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves
  - 4 plies = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…
- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

Image: Wikipedia

What Values to Use?



x>y => f(x)>f(y)

f(x) = Ax+B where A>0

- For worst-case minimax reasoning, evaluation function scale doesn't matter
  - We just want better states to have higher evaluations (get the ordering right)
  - Minimax decisions are ***invariant with respect to monotonic transformations on values***

- Expectiminimax decisions are ***invariant with respect to positive affine transformations***
- Expectiminimax evaluation functions have to be aligned with actual win probabilities!

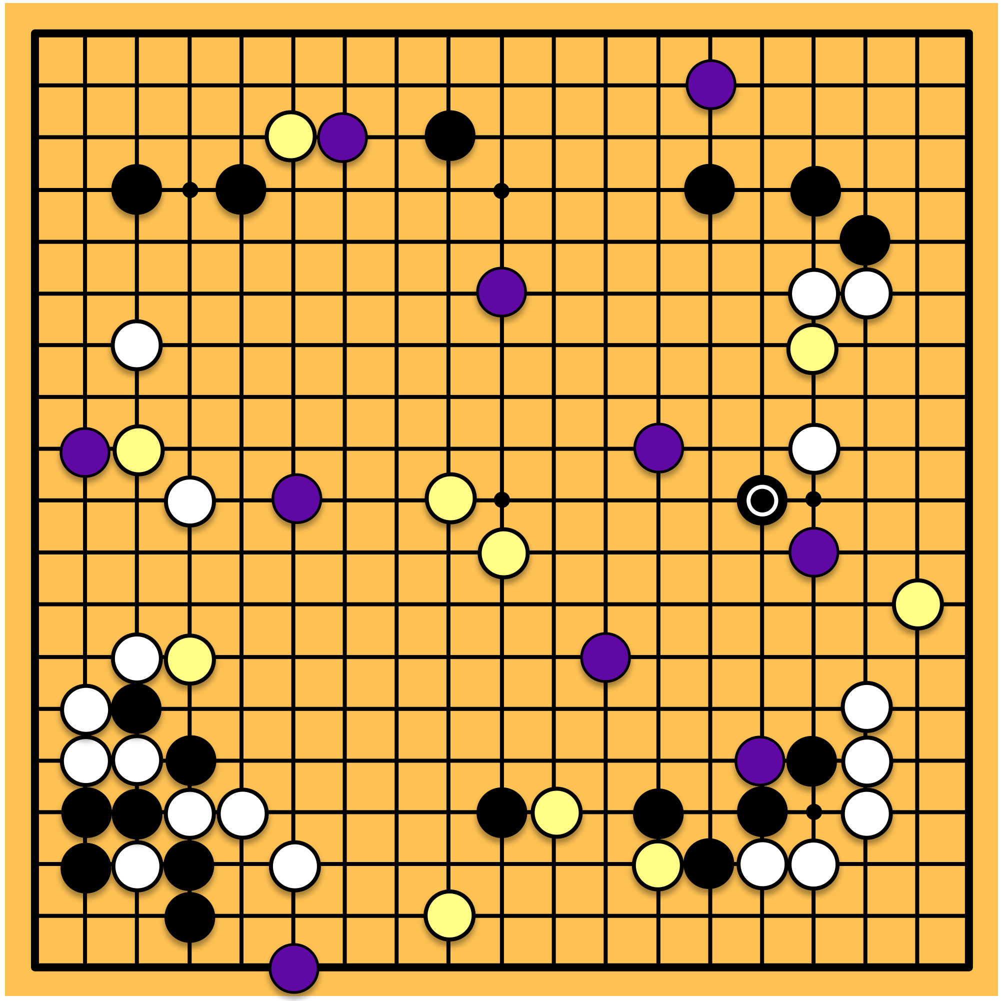
Monte Carlo Tree Search

- Methods based on alpha-beta search assume a fixed horizon
  - Pretty hopeless for Go, with $b > 300$
- MCTS combines two important ideas:
  - ***Evaluation by rollouts*** – play multiple games to termination from a state s (using a simple, fast rollout policy) and count wins and losses
  - ***Selective search*** – explore parts of the tree that will help improve the decision at the root, regardless of depth

Rollouts

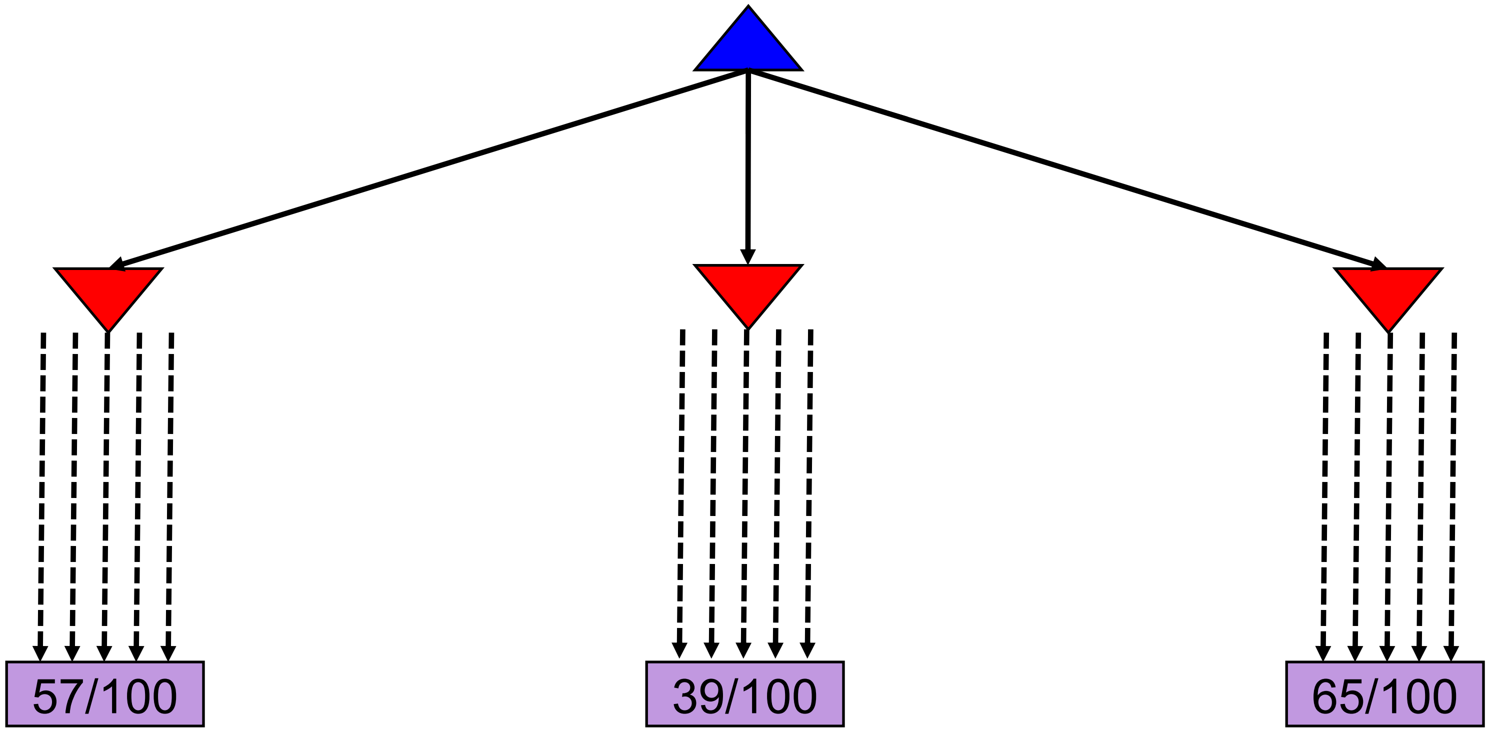
- For each rollout:
  - Repeat until terminal:
    - Play a move according to a fixed, fast rollout policy
  - Record the result
- Fraction of wins correlates with the true value of the position!
- Having a “better” rollout policy helps

“Move 37”



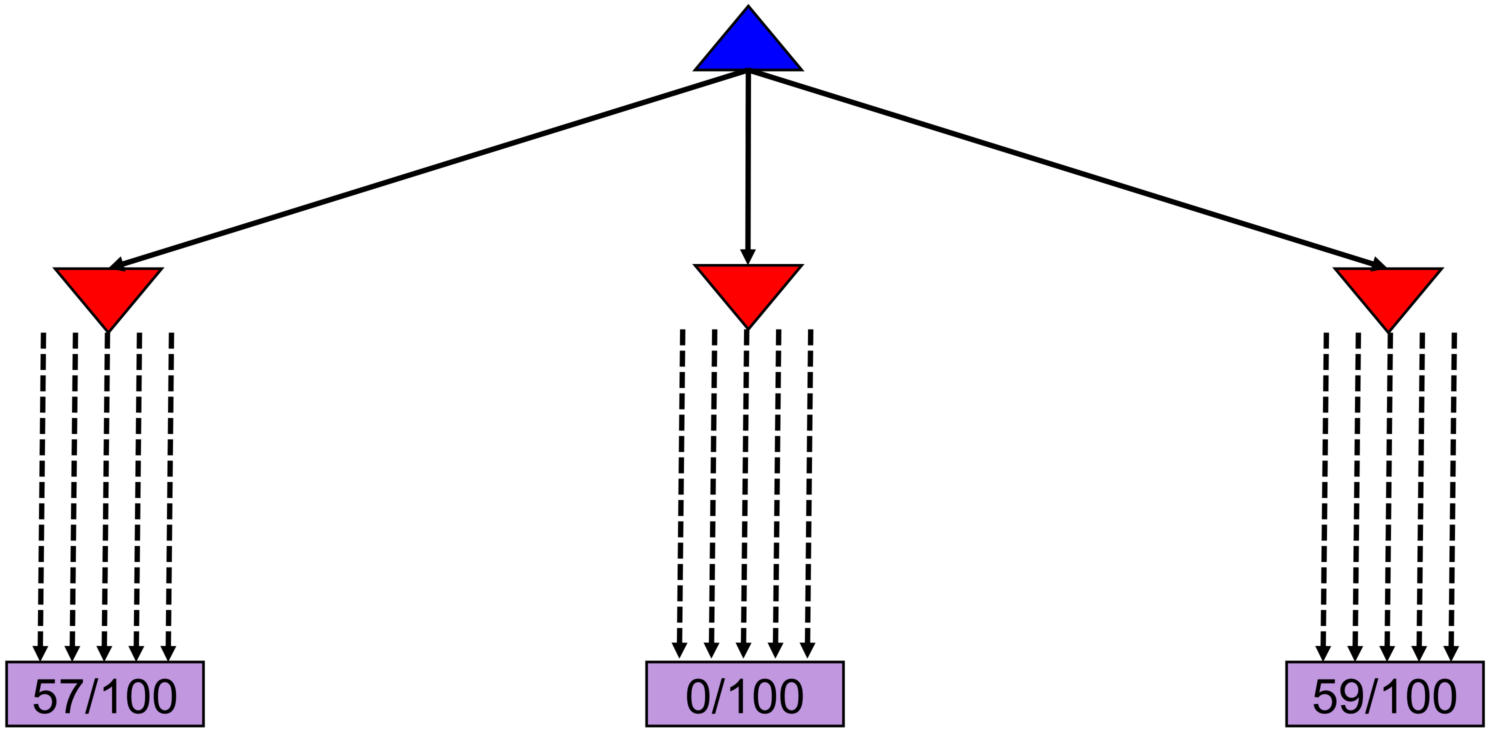
MCTS Version 0

- Do *N* rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric



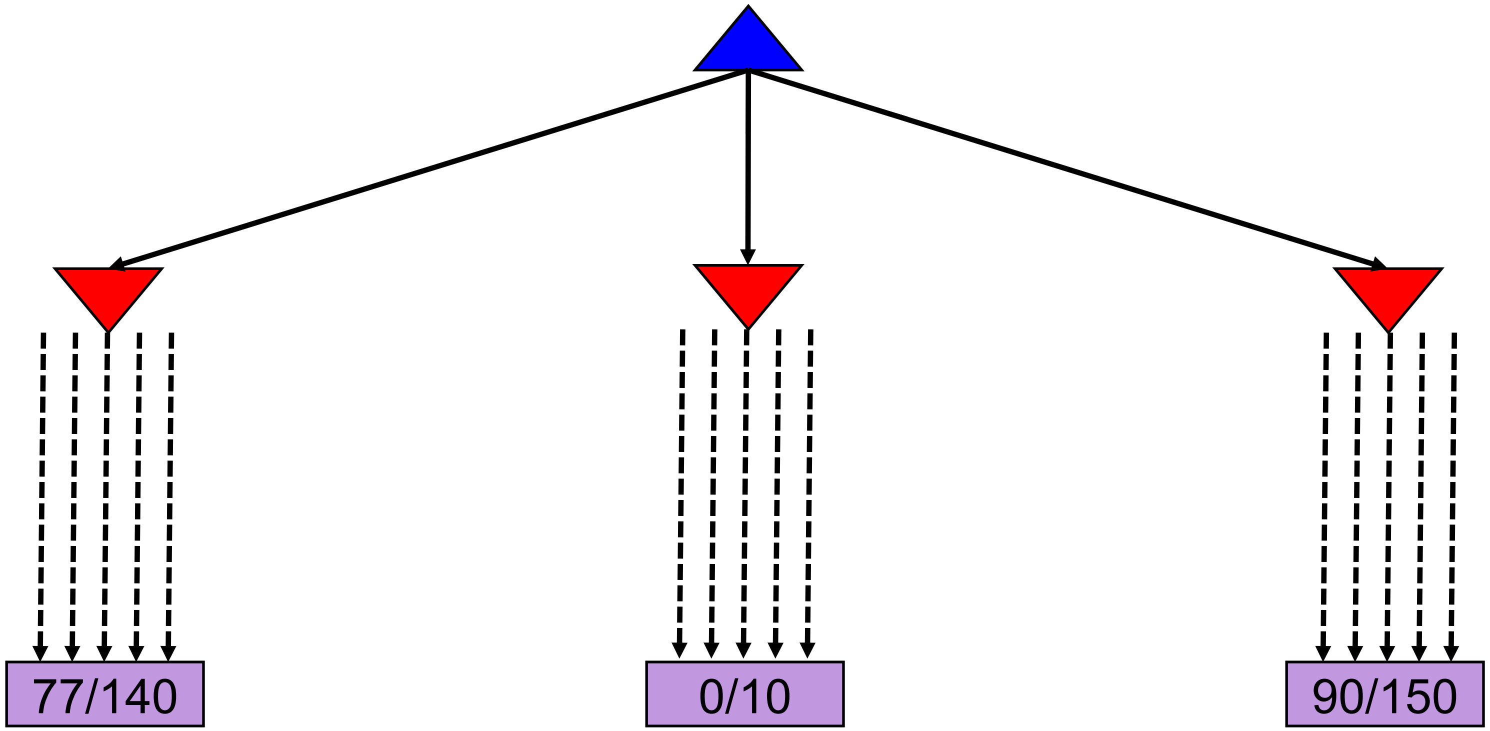
MCTS Version 0

- Do *N* rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric



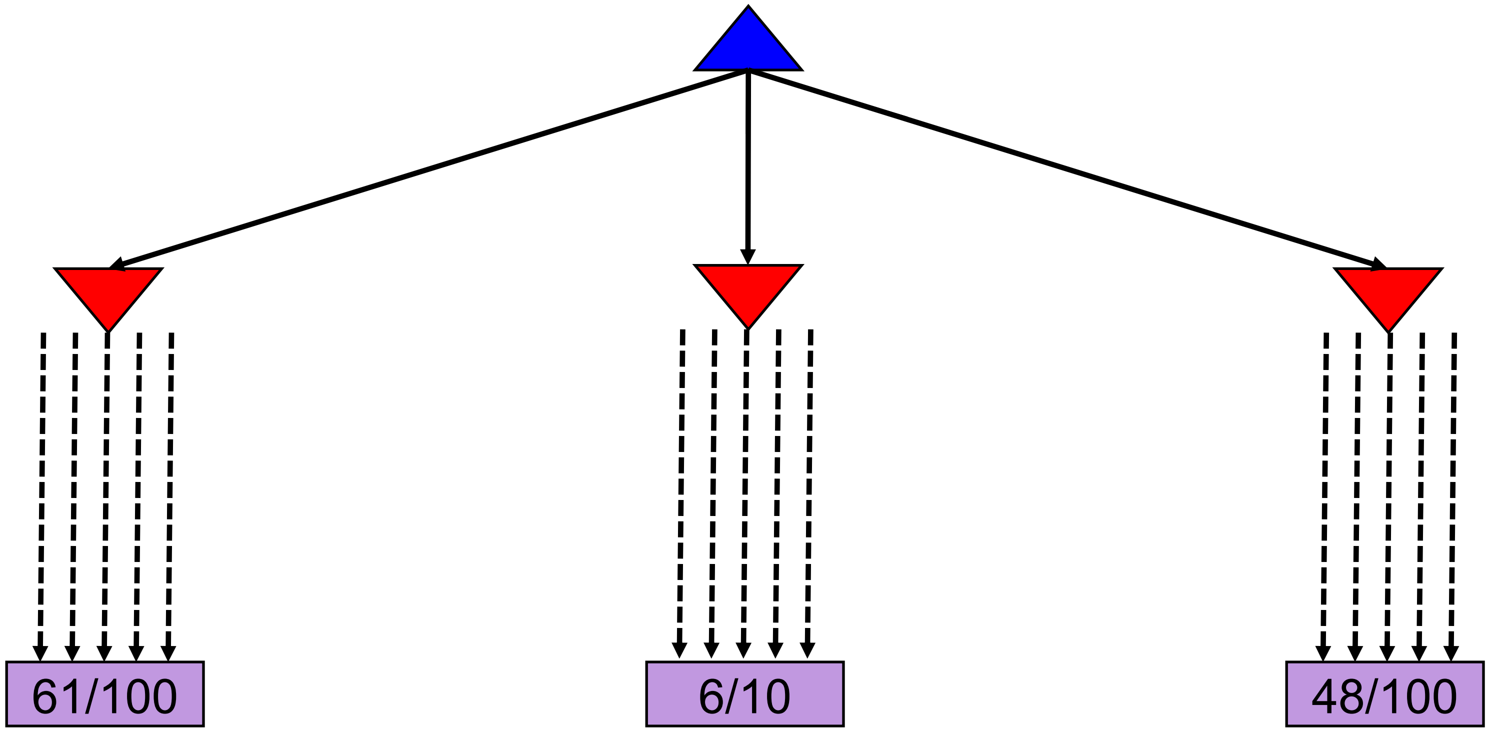
MCTS Version 0.9

- Allocate rollouts to more promising nodes



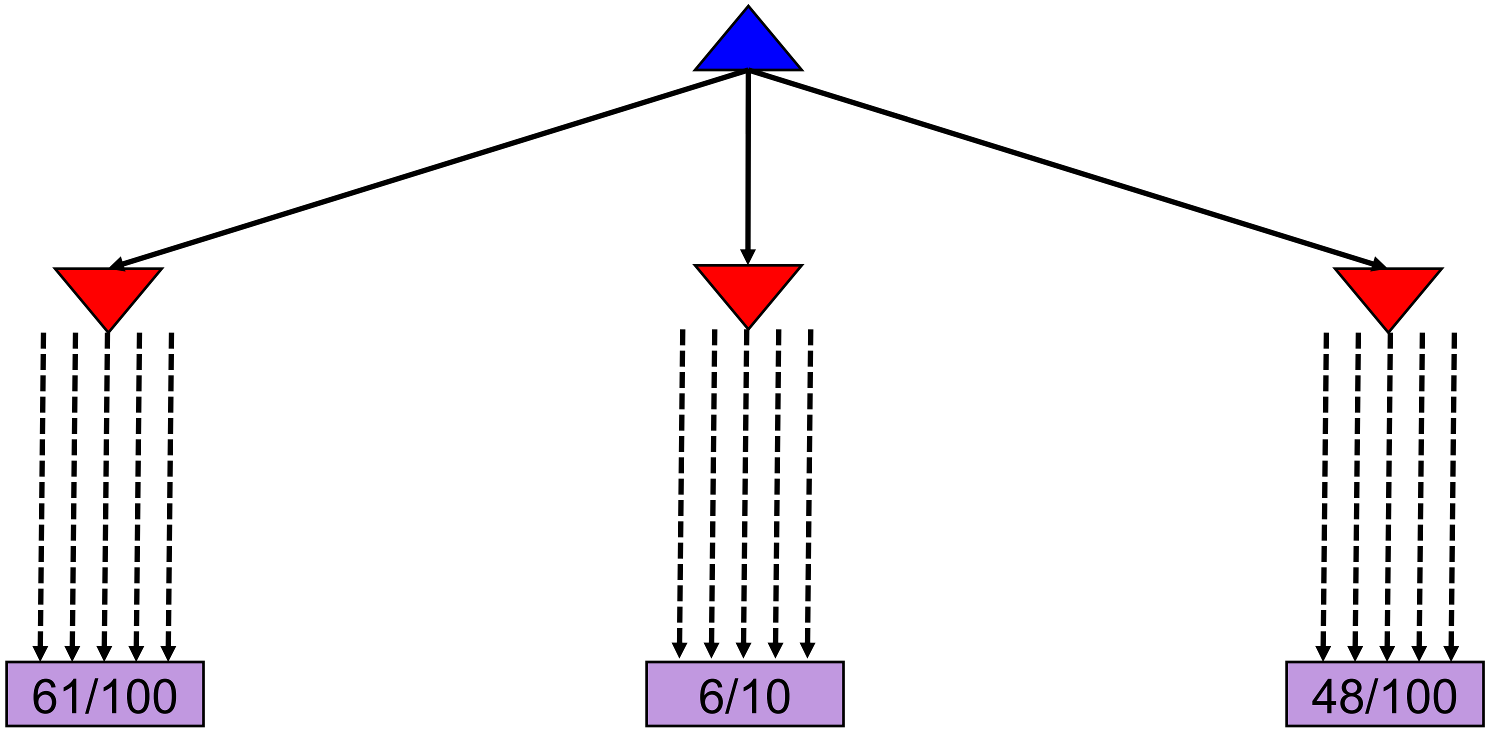
MCTS Version 0.9

- Allocate rollouts to more promising nodes



MCTS Version 1.0

- Allocate rollouts to more promising nodes
- Allocate rollouts to more uncertain nodes



UCB heuristics

- UCB1 formula combines “promising” and “uncertain”:

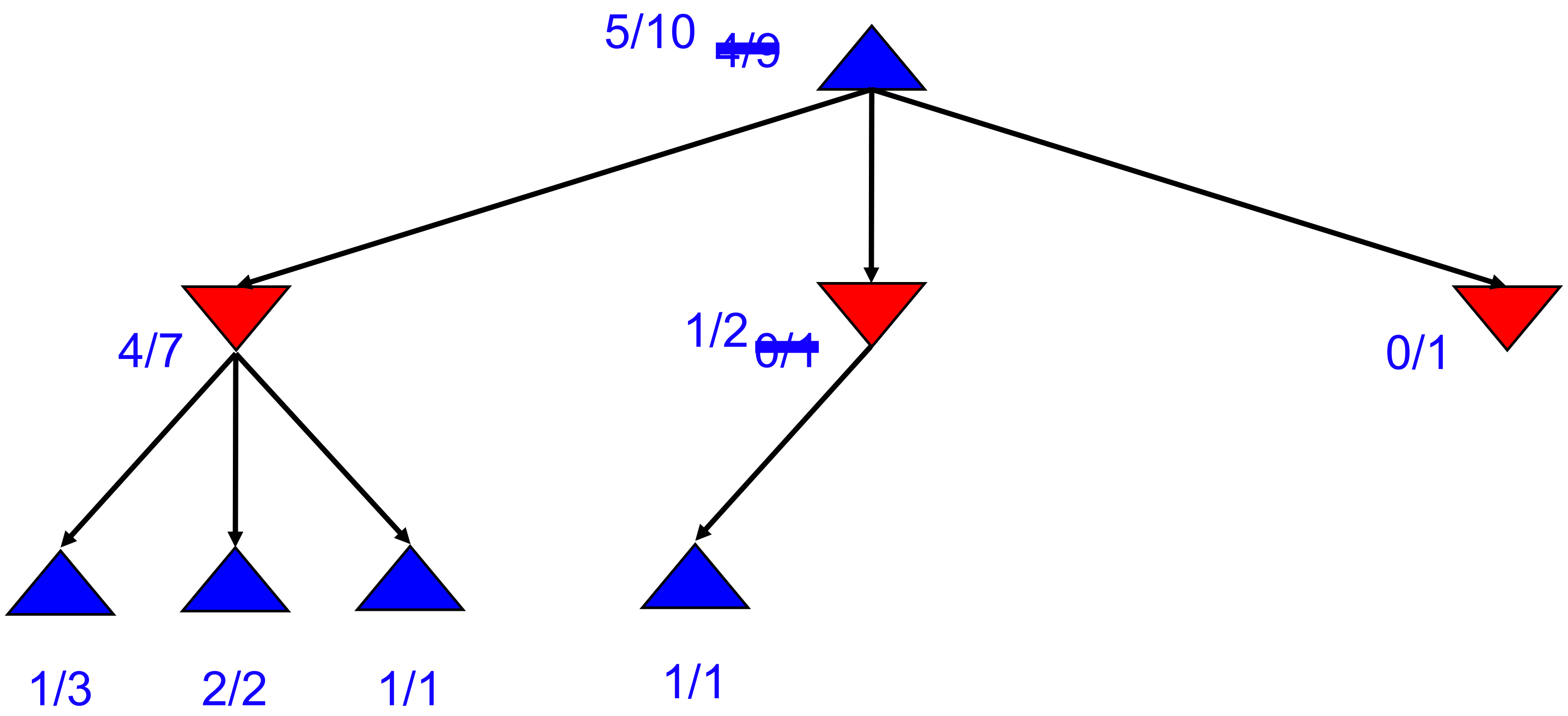
$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- $N(n)$ = number of rollouts from node n
- $U(n)$ = total utility of rollouts (e.g., # wins) for Player(Parent(n))
- A provably not terrible heuristic for ***bandit problems***
  - (which are not the same as the problem we face here!)

MCTS Version 2.0: UCT

- Repeat until out of time:
  - Given the current search tree, recursively apply UCB to choose a path down to a leaf (not fully expanded) node *n*
  - Add a new child *c* to *n* and run a rollout from *c*
  - Update the win counts from *c* back up to the root
- Choose the action leading to the child with highest *N*

UCT Example



Why is there no min or max?????

- "Value" of a node, $U(n)/N(n)$, is a weighted ***sum*** of child values!
- Idea: as $N \to \infty$, the vast majority of rollouts are concentrated in the best child(ren), so weighted average $\to$ max/min
- Theorem: as $N \to \infty$ UCT selects the minimax move
  - (but N never approaches infinity!)

Summary

- Games require decisions when optimality is impossible
  - Bounded-depth search and approximate evaluation functions
- Games force efficient use of computation
  - Alpha-beta pruning, MCTS
- Game playing has produced important research ideas
  - Reinforcement learning (checkers)
  - Iterative deepening (chess)
  - Rational metareasoning (Othello)
  - Monte Carlo tree search (chess, Go)
  - Solution methods for partial-information games in economics (poker)
- Video games present much greater challenges – lots to do!
  - $b = 10^{500}$, $|S| = 10^{4000}$, $m = 10{,}000$, partially observable, often > 2 players