COMP7015 Artificial Intelligence (S1, 2024-25)

Lecture 3 Part B: Adversarial Search

Instructor: Dr. Kejing Yin (cskjyin@hkbu.edu.hk)

Department of Computer Science
Hong Kong Baptist University

September 20, 2024

Adversarial Search

- Two-Player Zero-Sum Games
- The Minimax Search Algorithm
- Alpha-Beta Pruning
- Heuristic Alpha–Beta Tree Search
- Monte Carlo Tree Search

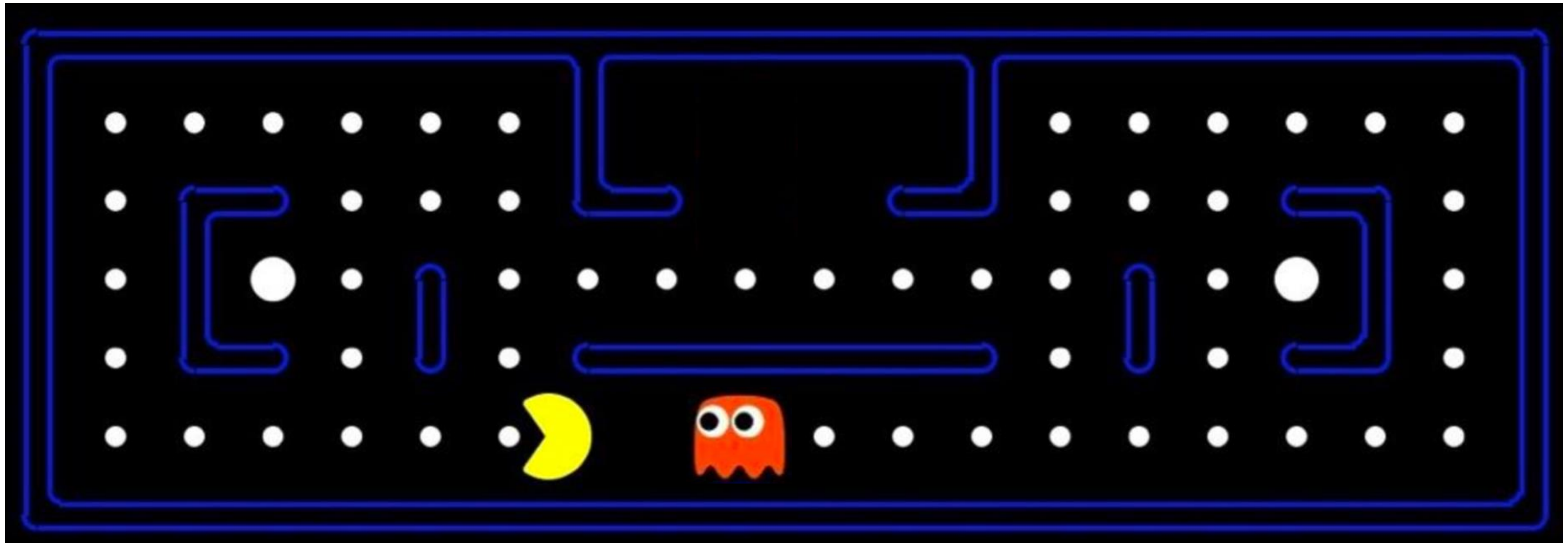
Let’s Play A Game First

(1)	(2)	(3)
+50	+20	+2
-20	-10	+1

- There are three boxes, each with two numbers.
- You pick a box, and I pick the number for you. The one with a larger number wins.

- Which box will you pick?
- If I play optimally (against you), which box will you pick?
- If I play randomly, which box will you pick?
- You best move depends on your mental modeling of your opponent (me)!

Example of A Game: PacMan with A Ghost



PacMan with no ghosts: The agent use searching algorithms to find solutions

Adding a ghost: The ghost plays against the PacMan (tries to eat it)

We focus on games, but multi-agent settings are common in many AI subfields.

Games

- Different types of games:
  - Deterministic or stochastic?
  - Number of players: 1, 2, or even more?
  - Zero sum?
  - Turn-taking?
  - Perfect information (all players see everything)?

- Our objectives: finding a strategy (policy) that selects a move for each state.

Deterministic Games

- Formulation of a deterministic game:
  - States: S (starts at the initial state S_0)
  - Players: $p \in \{1, \dots, N\}$
  - Action function $\mathrm{Actions}(s)$: The set of legal moves in state s
  - Transition function $\mathrm{Result}(s, a)$: The results of taking action a in state s.
  - Terminal test $\mathrm{IsTerminal}(s)$: True when the game is over and false otherwise.
  - Utility function $\mathrm{Utility}(s, p)$: A value on outcome to player p when the game ends in state s.

  - Example of utility: In chess, the outcome is a win, loss, or draw. Utility=1, 0, or 1/2.

- Solution for a player is a policy: $S \rightarrow A$ (action to take at each state)

Zero-Sum Games

- Purely competitive (adversarial)
- One agent wins means that the other agent losses.
- Agents have opposite utilities

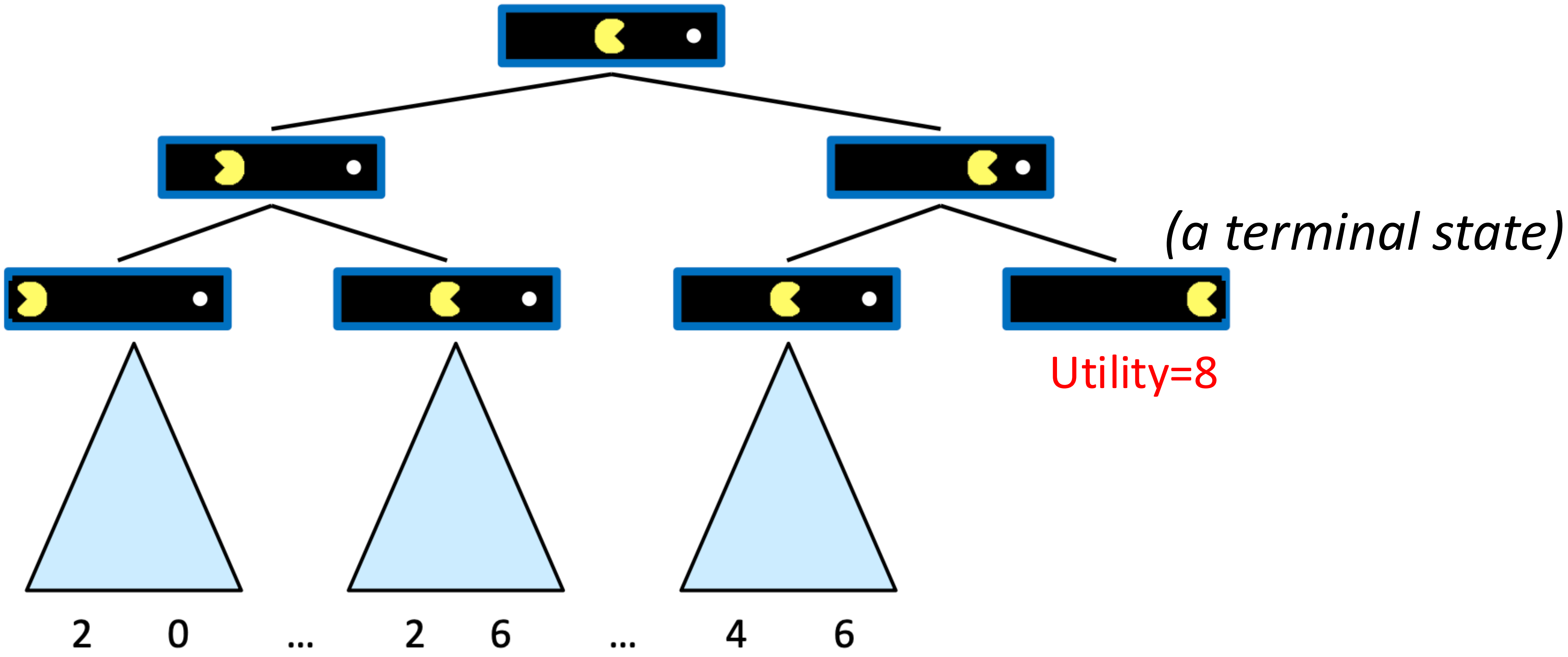
- For now, we focus on deterministic, two-player, turn-taking, perfect information, and zero-sum games.

Another Example of Multiple Agents

- Hide and seek game https://openai.com/blog/emergent-tool-use/

Single-Agent Search Trees

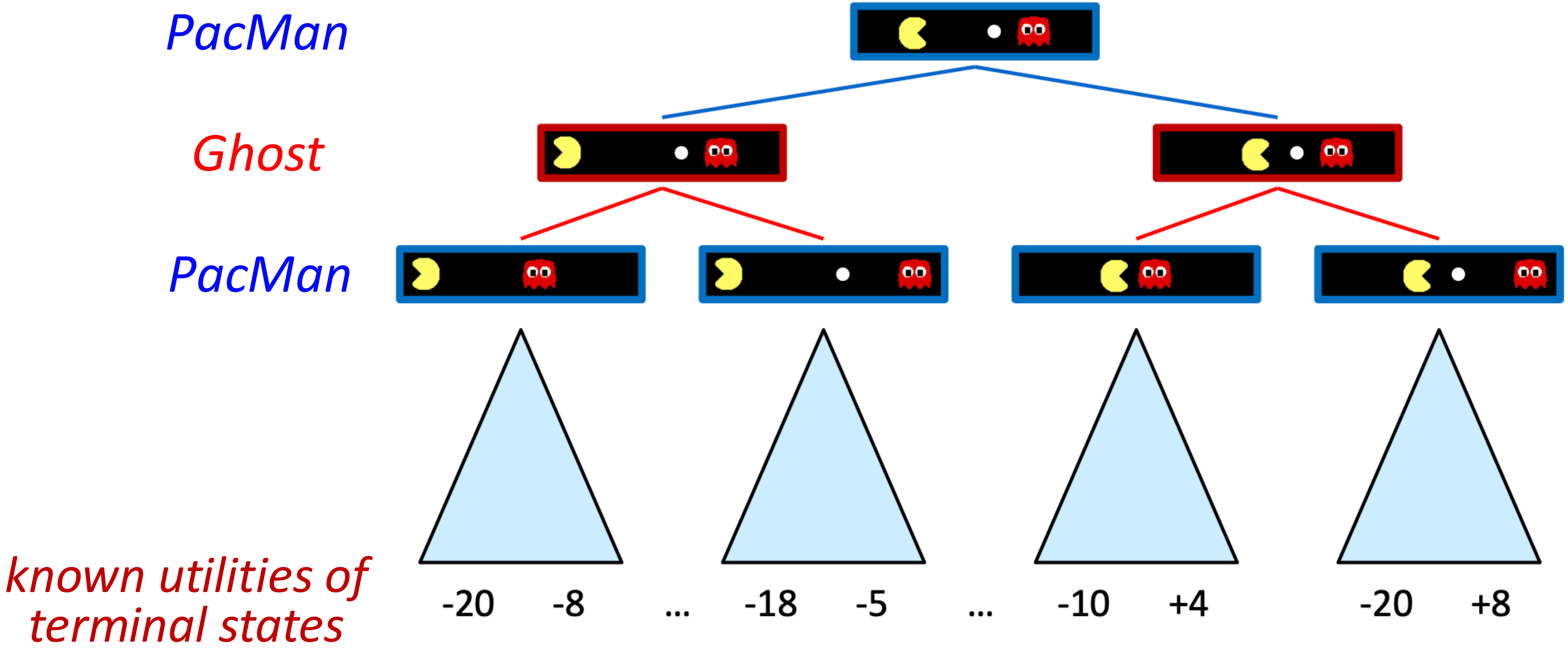
- Each action costs 1 score.
- Eating the dot gets 10 scores.



Example from CS188@UCB

Adversarial Game Trees

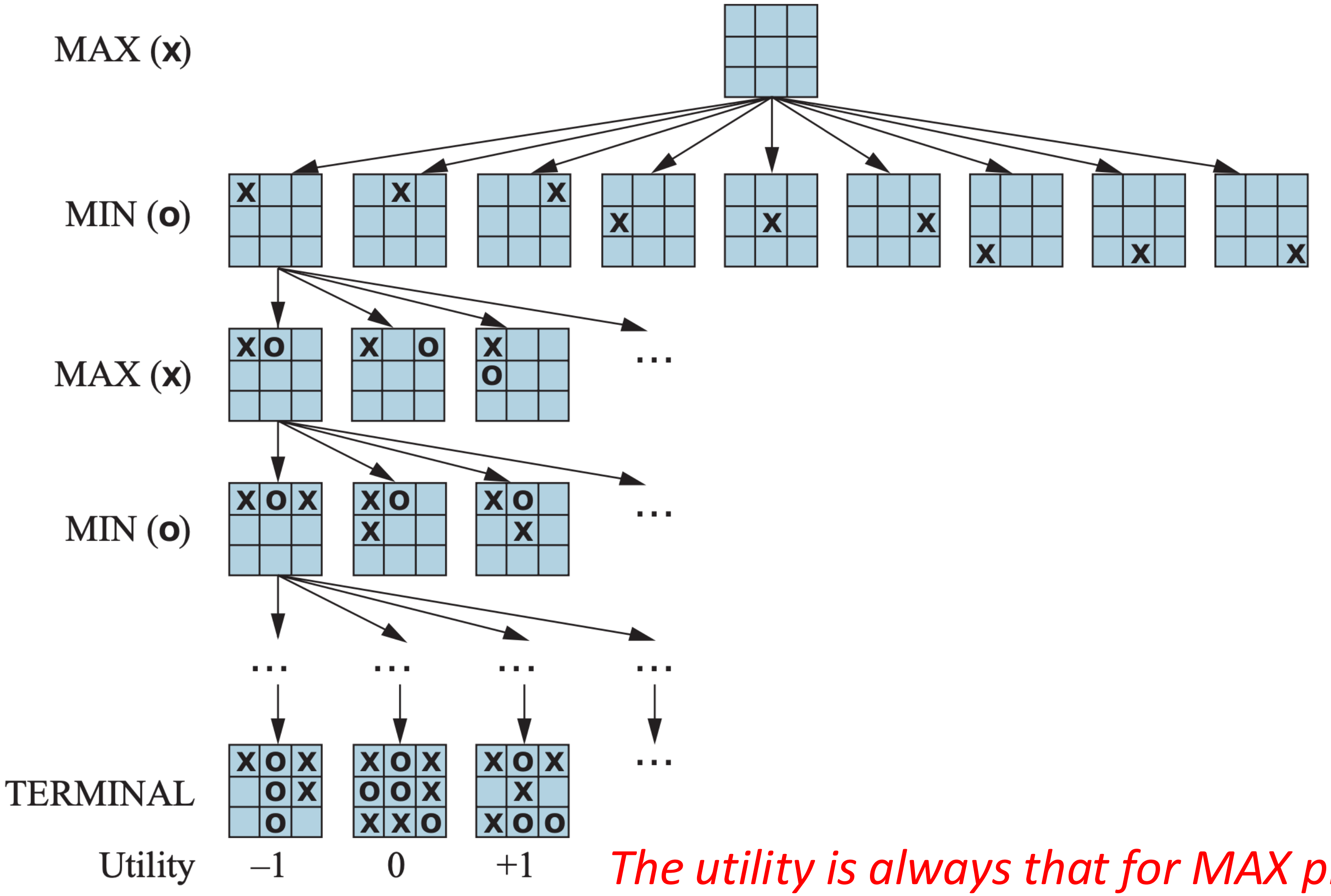
- Similar as search trees, we can construct game trees for adversarial games.
- Players take turns to make actions.



Example from CS188@UCB

Adversarial Game Trees

- Similar as search trees, we can construct game trees for adversarial games.
- *Another example: Two players (MAX and MIN) playing tic-tac-toe.*

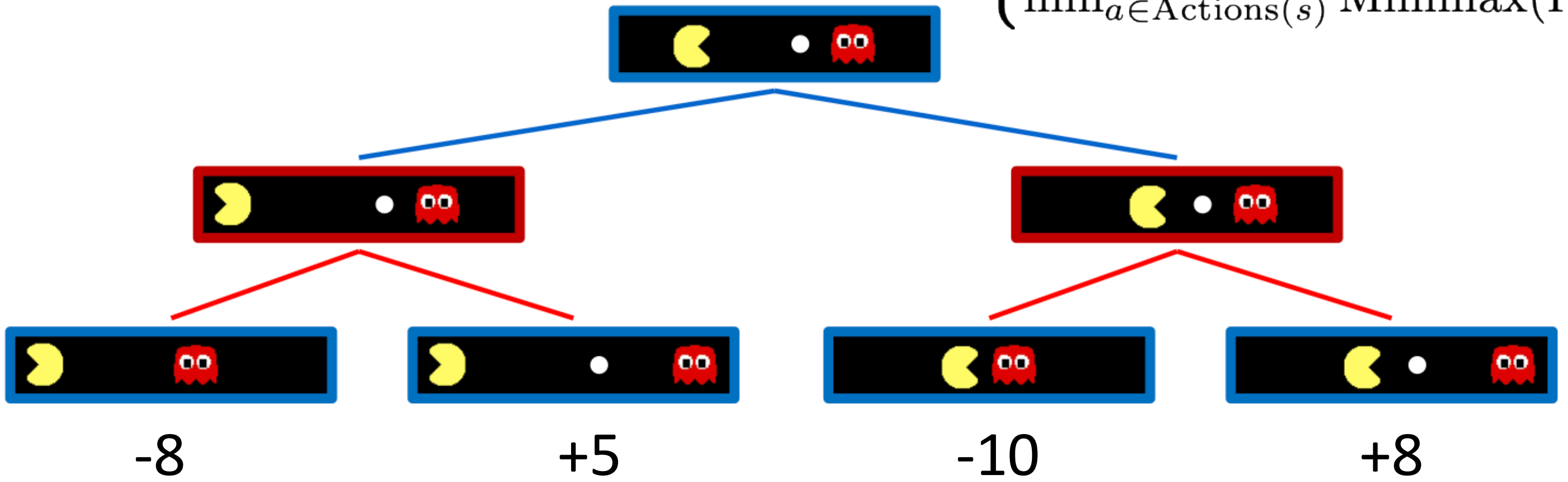


The utility is always that for MAX player

Minimax Values

- MAX tries to maximize its utility while MIN tries to minimize it.
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game.

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s, \text{MAX}) & \text{if IsTerminal}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if Player}(s) = \text{MIN} \end{cases}$$



Terminal states are known

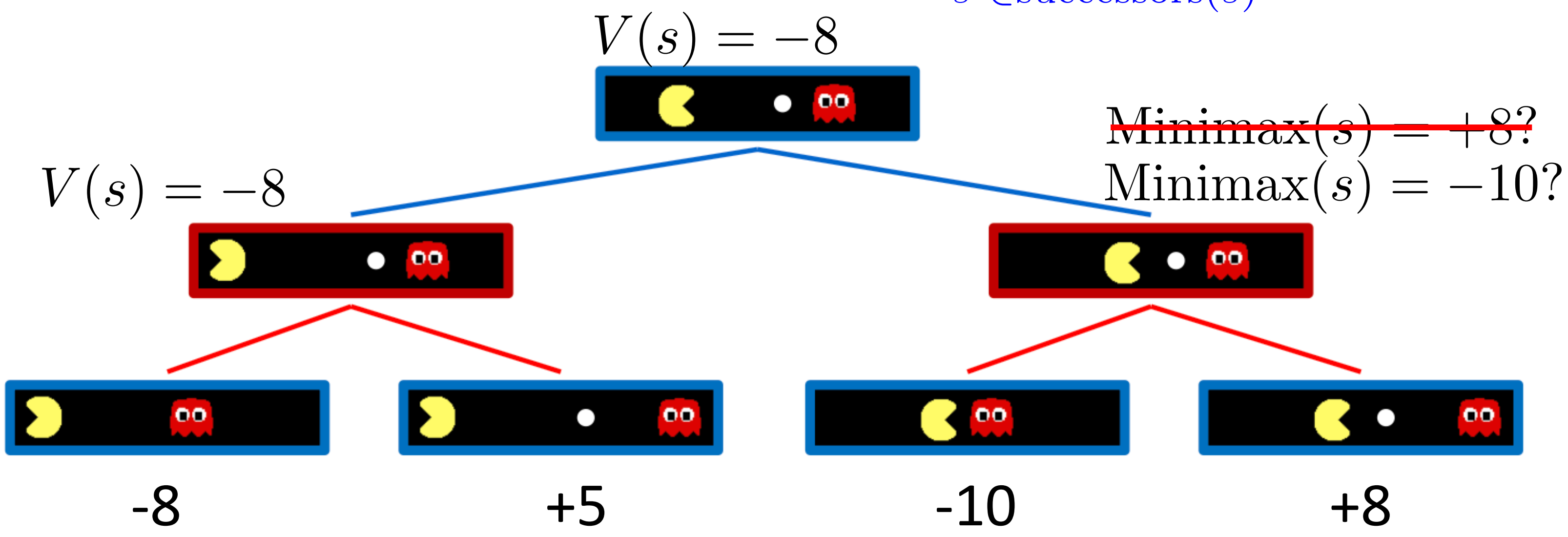
(some random numbers in this example for illustration)

Example from CS188@UCB

Minimax Values

- MAX tries to maximize its utility while MIN tries to minimize it.
- The minimax value is the utility (for MAX) of being in that state, *assuming that both players play optimally from there to the end of the game*.

States Under MAX's Control: $\mathrm{V}(s) = \max_{s' \in \text{successors}(s)} \mathrm{V}(s')$



States Under MIN's Control:

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

(use V as a shorthand notation for minimax value)

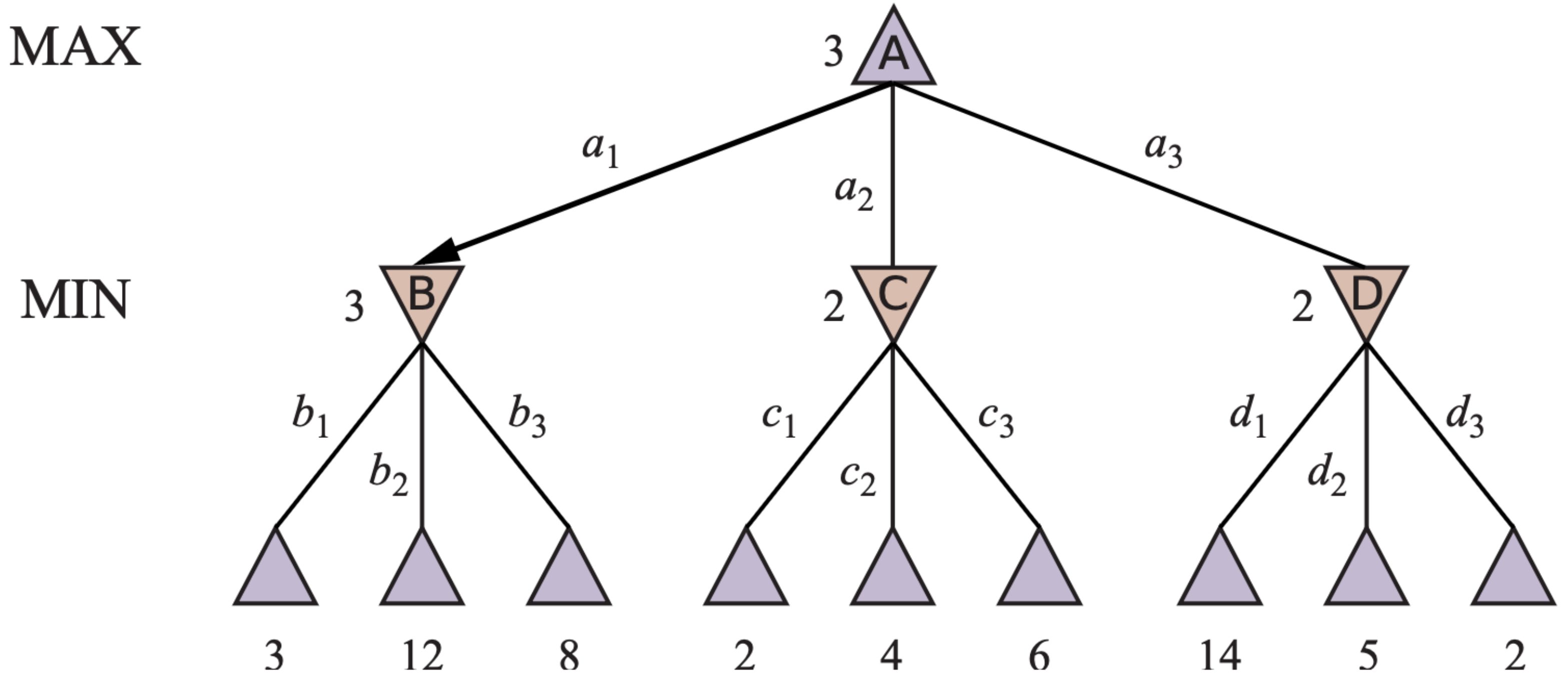
Terminal states are known

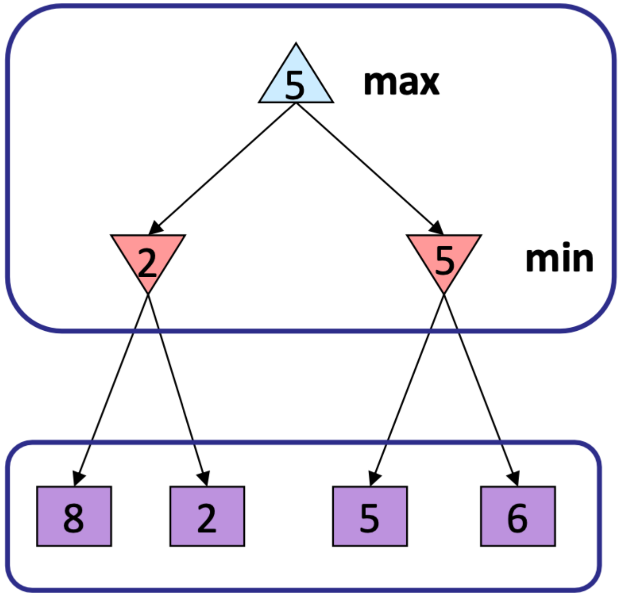
(some random numbers in this example for illustration)

Example from CS188@UCB

Minimax Values of Game Trees

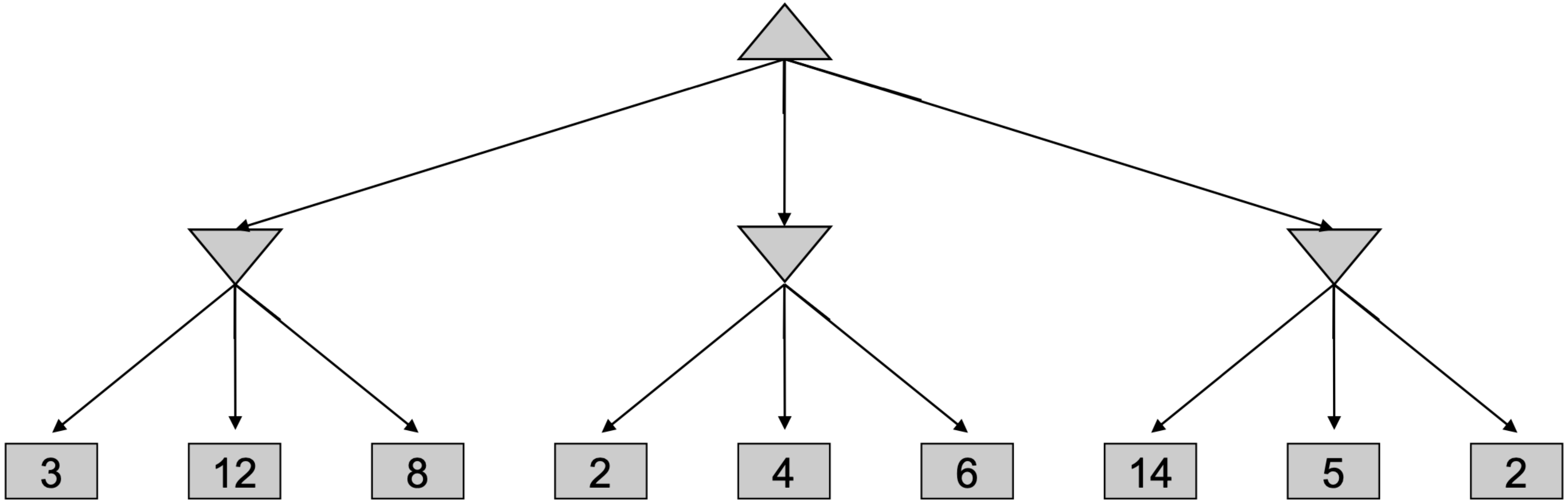
- Convention: △ for MAX and ▽ for MIN





Terminal utilities are defined by the game

Exercise: Determine The Minimax Values



Minimax Search Algorithm

- Best move for MAX: the action whose resulting state has the highest Minimax value.
- Best move for MIN: the action whose resulting state has the lowest Minimax value.

- Computing the values:

```
def Minmax-value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

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

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

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

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

Exercise: The Halving Game

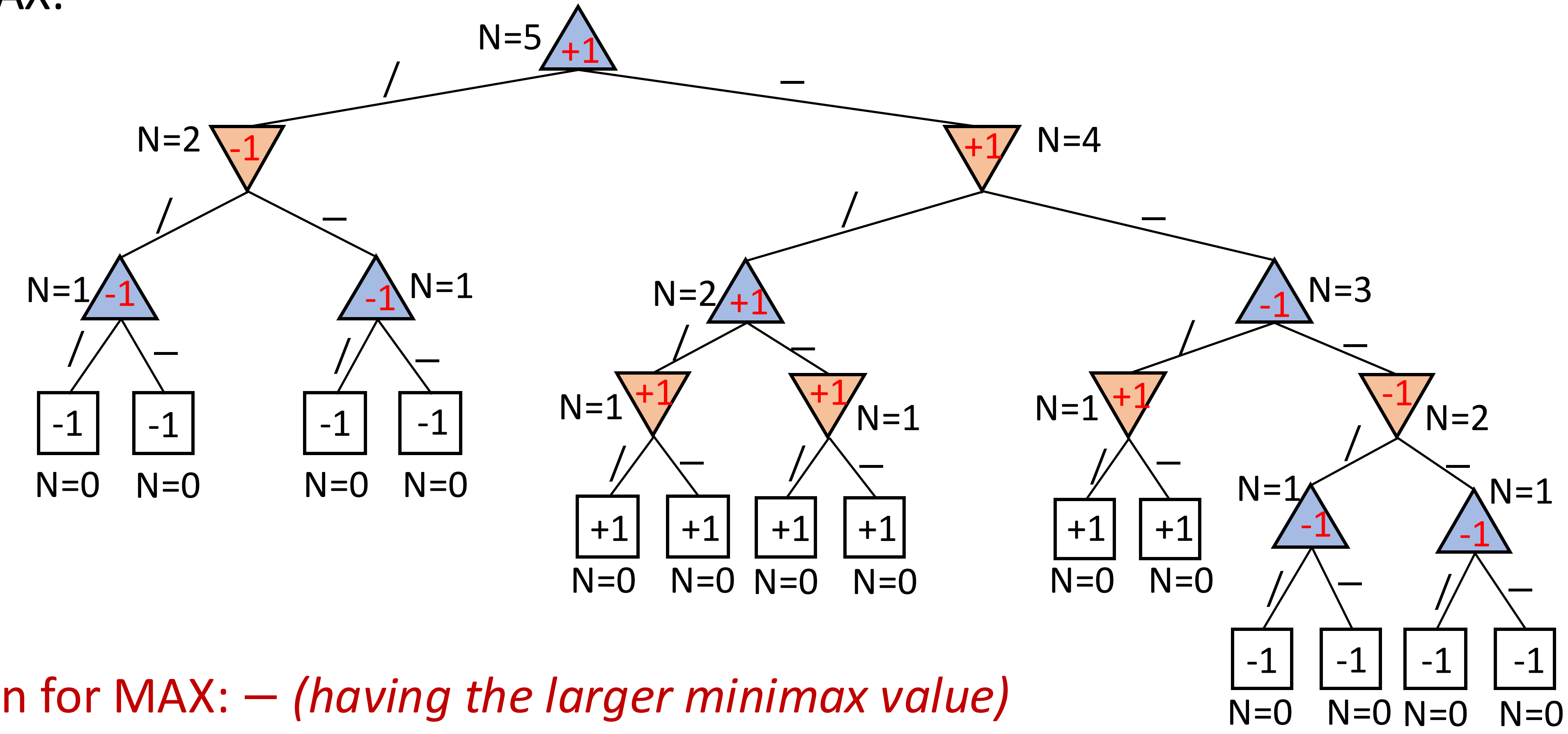
Starting from a number N, two players take turns to replace N with either $\left\lfloor \frac{N}{2} \right\rfloor$ or $N - 1$. The player left with zero wins. Assume N=5 and MAX plays first, perform minimax search to find the best move for MAX.

- Utility:
  - +1 for wining
  - -1 for losing
- Actions:
  - /: $\left\lfloor \frac{N}{2} \right\rfloor$
  - $-$: $N - 1$

Exercise: The Halving Game

Starting from a number N, two players take turns to replace N with either $\left\lfloor \frac{N}{2} \right\rfloor$ or $N-1$. The player left with zero wins. Assume N=5 and MAX plays first, perform minimax search to find the best move for MAX.

- Utility:
  - +1 for wining
  - -1 for losing
- Actions:
  - /: $\left\lfloor \frac{N}{2} \right\rfloor$
  - –: $N-1$



Action for MAX: – *(having the larger minimax value)*

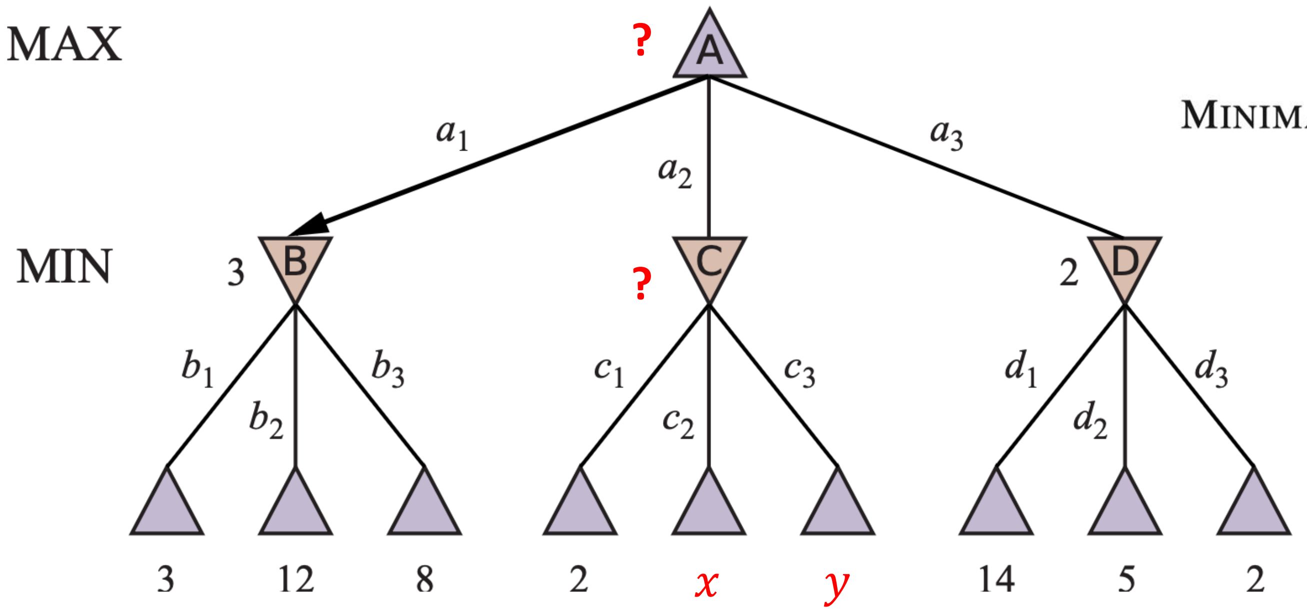
Efficiency or Feasibility of Minimax

- If $N = 100$, can we still run a minimax search?
- Number of game states is exponential in the depth of the tree.
- For tic-tac-toe: fewer than $9! = 362{,}880$ terminal nodes.
- For chess: over 10^{40} nodes.

- Impossible to exhaust for most games.

Alpha-Beta Pruning

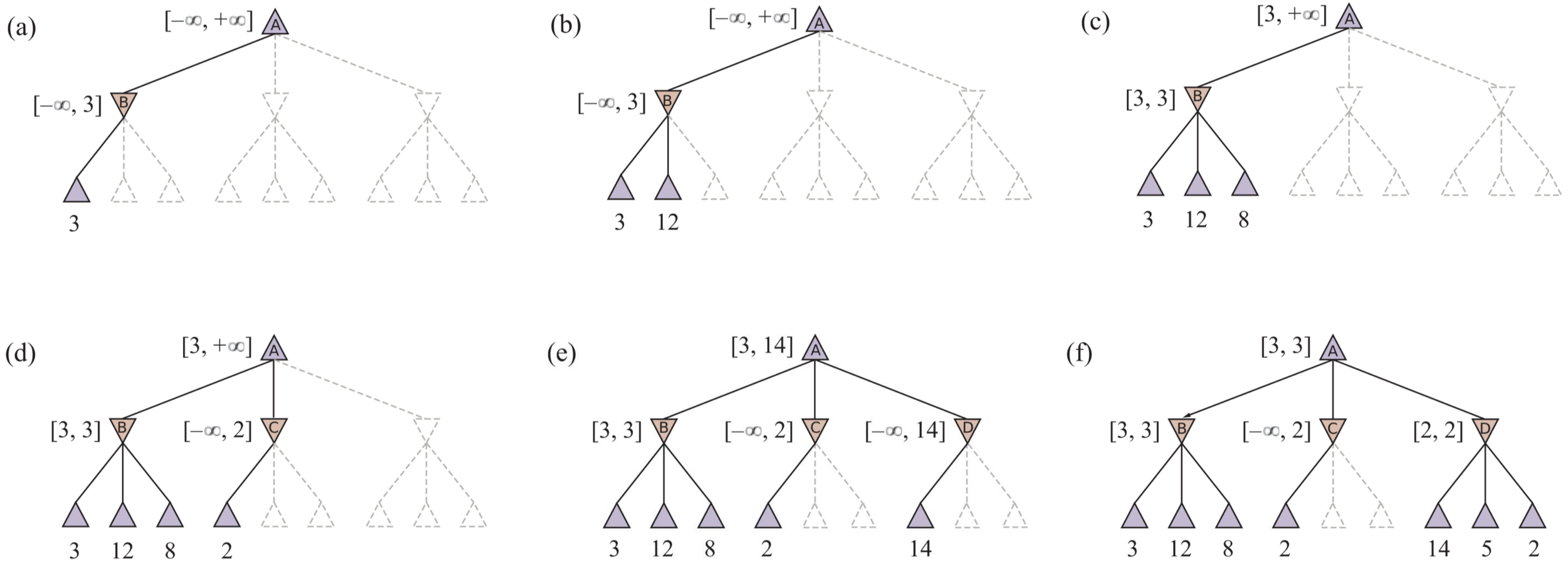
- Computing the correct minimax decision without examining every state?



$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2)) \\
&= \max(3,\min(2,x,y),2) \\
&= \max(3,z,2) \qquad \text{where } z=\min(2,x,y)\le 2 \\
&= 3.
\end{aligned}
$$

Can we compute Minimax(root) without evaluating x and y? *They can be pruned.*

Minimax Pruning: Detailed Steps



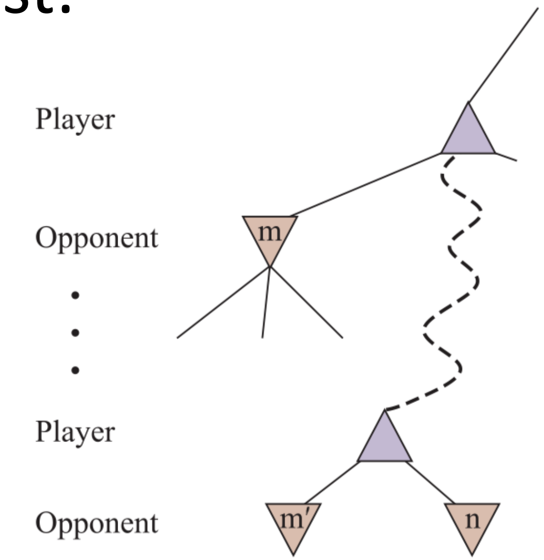
Alpha-Beta Pruning

Two extra parameters:

- α: the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: α = "at least."
- β: the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: β = "at most."

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

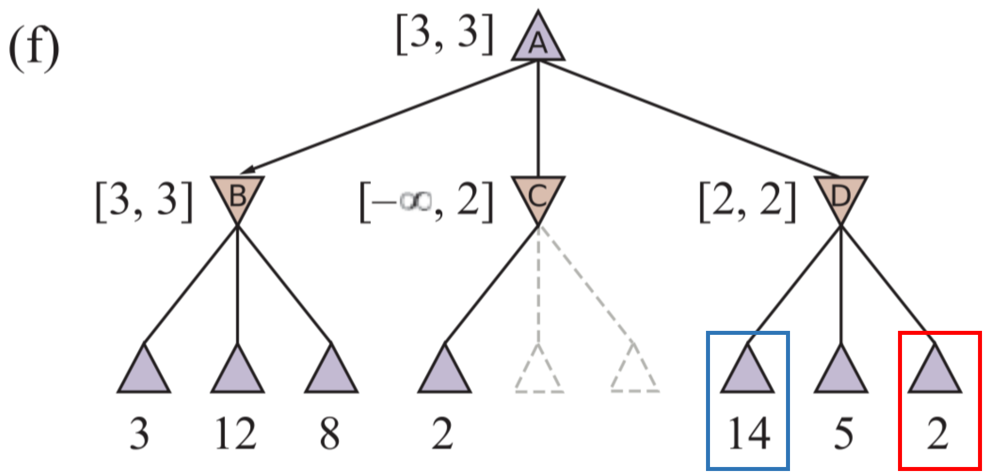
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```



If m or m' is better than n for Player, we will never get to n in play.

Move Ordering

The effectiveness of alpha–beta pruning depends on the order of examining the states.



If we examined the node with value 2 first, we would have pruned the nodes with values 14 and 5.

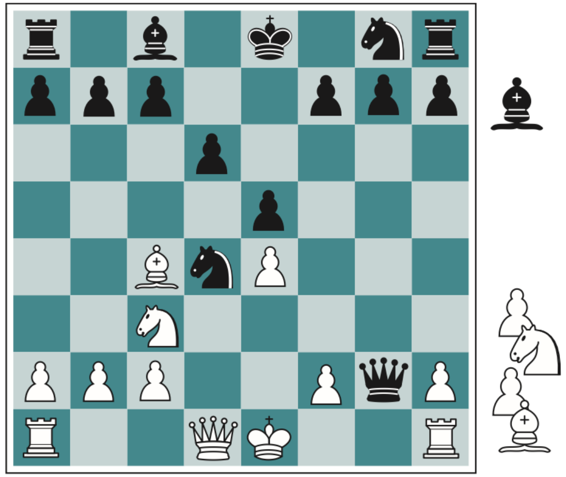
We cannot prune any successors of D because the worst successors (for MIN) was generated first.

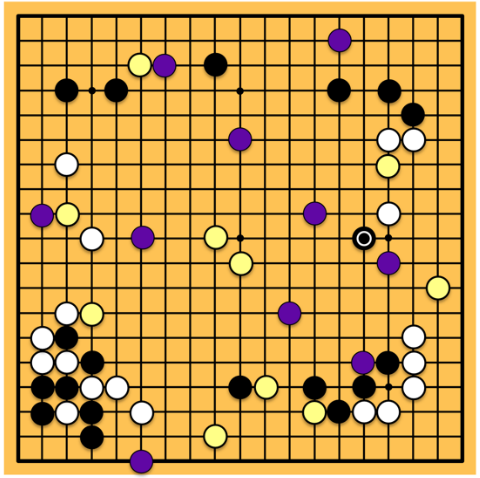
It might be worthwhile to try to first examine the successors that are likely to be the best.

- For chess, a simple ordering function (such as trying *captures* first, then *threats*, then *forward moves*, and then *backward moves*) could reduce number of nodes from $O(b^m)$ to $O(b^{m/2})$.
- With random move ordering, the number of nodes is roughly $O(b^{3m/4})$.
- Another scheme: try first the moves that were found to be best in the past.

Solving *Chess* and *Go*?

- Can we solve chess & Go using the minimax search with alpha-beta pruning?

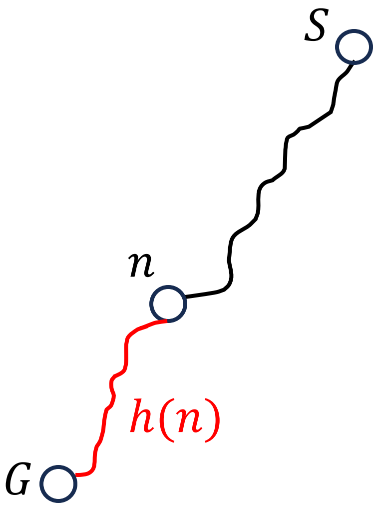


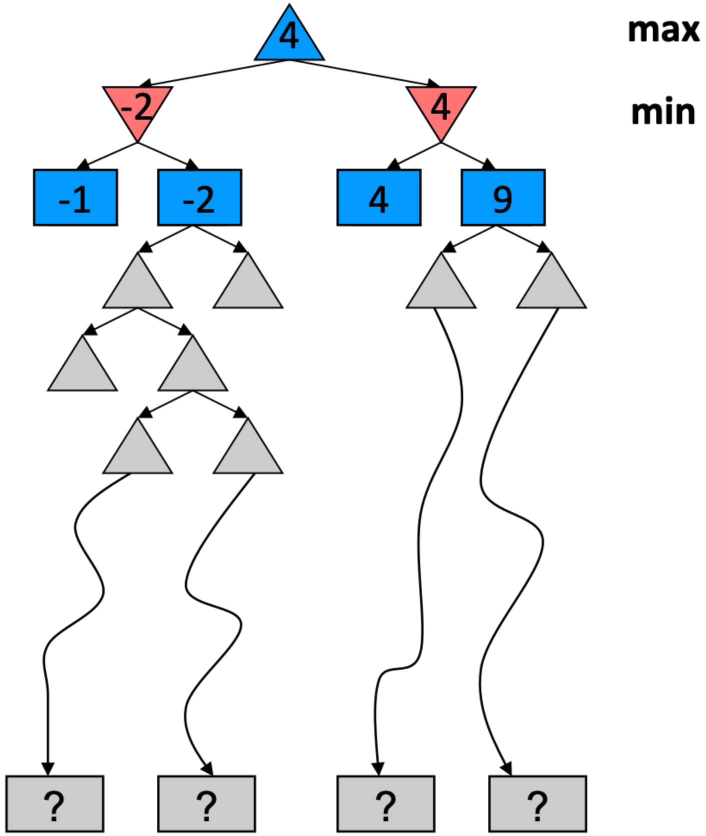


- If done perfectly, alpha-beta examines $O(b^{m/2})$ nodes, instead of $O(b^m)$ for minimax.
- Expanding eight plies:
  - Minimax: Chess: $35^8 \approx 10^{12}$ Go: $300^8 \approx 10^{19}$
  - Alpha-beta: Chess: $35^{8/2} \approx$ 1Million Go: $300^{8/2} \approx$ 8 Billion
- In reality, we cannot reach the leaves! *(i.e., cannot compute values)*

Heuristic Alpha–Beta Tree Search

- Recall in standard search problems (e.g., 8-puzzle), how do we speed up?
  - Use heuristic functions to *estimate the cost from state n to a goal state.*





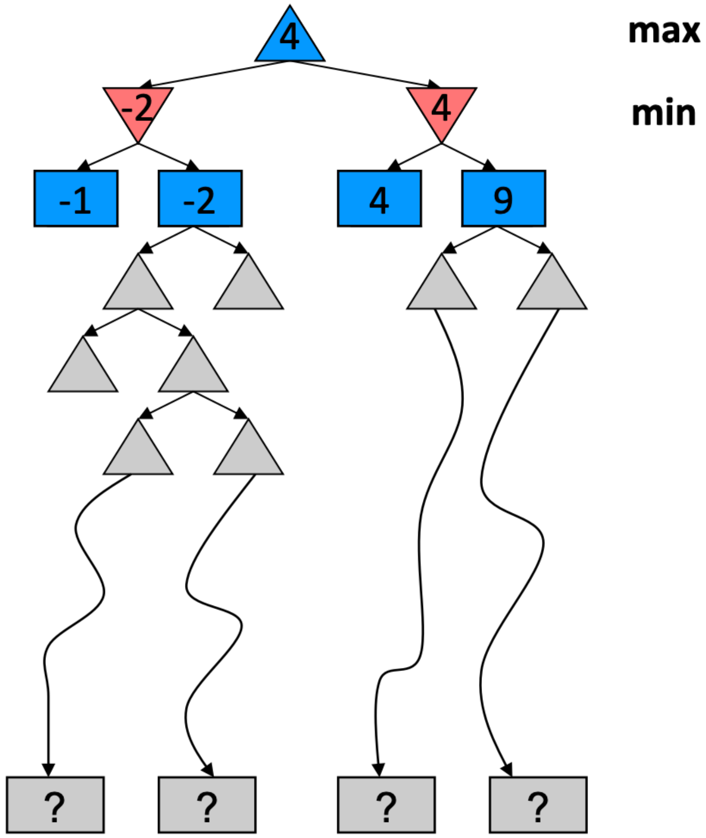
- Apply the same idea: *estimate the values*.

Example from CS188@UCB

Cutting Off Search

- Search to a limited depth in the game tree.
- Use a heuristic evaluation function to estimate the value of non-terminal nodes.

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$



- Suppose we have 100 seconds / move:
  - We can explore 10K nodes / second;
  - We can check ~1M nodes / move;
  - Alpha-beta reaches about depth 8 – gives us a descent chess program

Example from CS188@UCB

Evaluation Function

Formally, $Eval(s,p)$ returns an estimate of the value of state s to player p.

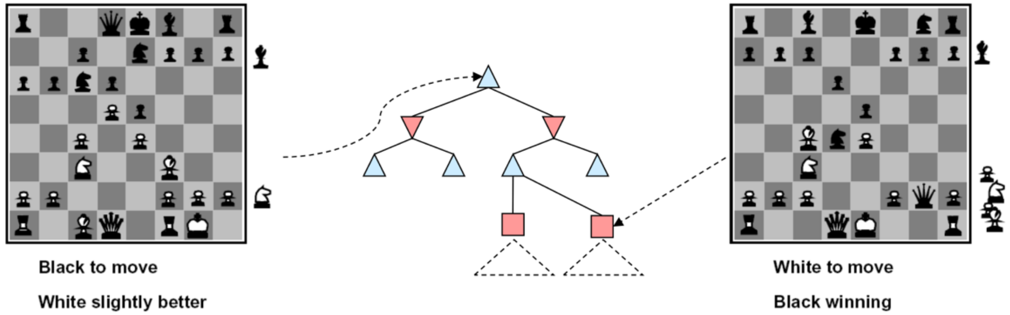
- For terminal states, it must be that $Eval(s,p) = Utility(s,p)$
- For non-terminal states, $Utility(loss,p) \leq Eval(s,p) \leq Utility(win,p)$

- Computation must be fast!
- The evaluation function should be strongly correlated with the actual chances of winning.

Evaluation Function

- A typical and practical form: weighted linear sum of features:

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



- Examples:

  $f_1(s)$ = (number of black queens – number of white queens) *(larger f_1 → higher chance to win)*

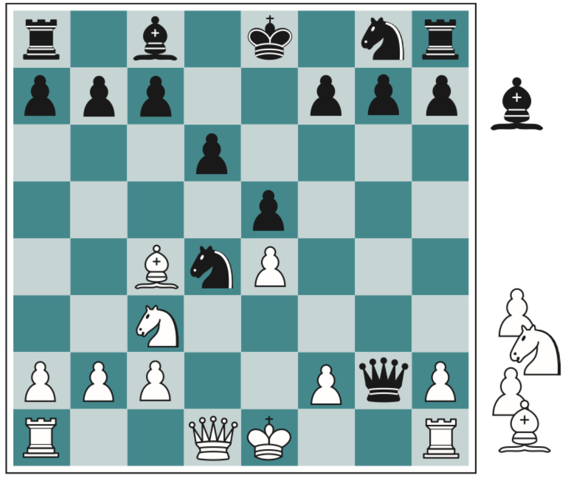
  $f_2(s)$ = 5 if black king is well protected, 0 if threatened *(larger f_2 → higher chance to win)*

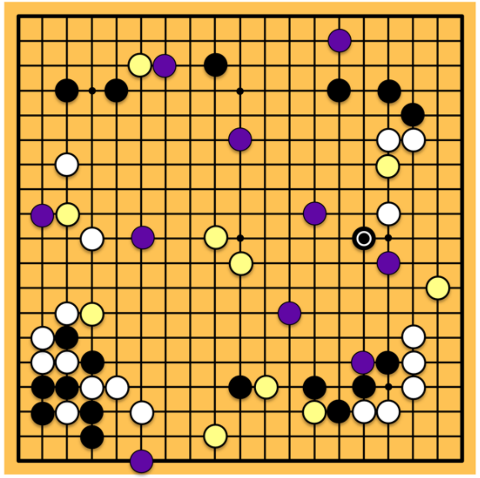
  $f_3(s)$ = …

- More advanced technique: learning the features using machine learning.

Solving *Chess* and *Go*?

- Can we solve chess & Go using the minimax search with alpha-beta pruning?





- If done perfectly, alpha-beta examines $O(b^{m/2})$ nodes, instead of $O(b^m)$ for minimax.
- Expanding eight plies:
  - Minimax: Chess: $35^8 \approx 10^{12}$ Go: $300^8 \approx 10^{19}$
  - Alpha-beta: Chess: $35^{8/2} \approx$ 1Million Go: $300^{8/2} \approx$ 8 Billion

For chess: costly but manageable

Has some good evaluation functions

For Go: still infeasible even with cut off

Hard to define evaluation functions

Monte Carlo Tree Search: The Main Idea

- ~~Evaluation function~~ → Simulations (playout or rollout)
  - Play multiple games to termination from a state s and count wins/losses.
  - Value of s = "win percentage" (do not use heuristic functions).
  - Need a *playout policy* that plays the game to termination.
  - A simple policy: play completely randomly.

- ~~Many evaluations per move~~→ Strategically select nodes to expand
  - Need a *selection policy* that selectively focuses the computational resources on the important parts of the game tree.

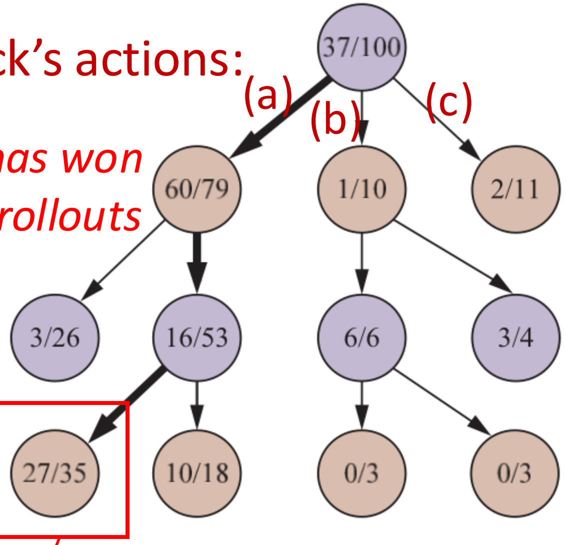
Monte Carlo Tree Search: The Main Idea

- Four steps in one iteration: select, expand, simulate, and back-propagate.

white has won 37 out of 100 rollouts

black's actions:

black has won 60/79 rollouts



(a) Selection

A simple selection policy: select the most promising node.

We have simulated 100 times from this node or its descendants. Among them, white wins 37 times.

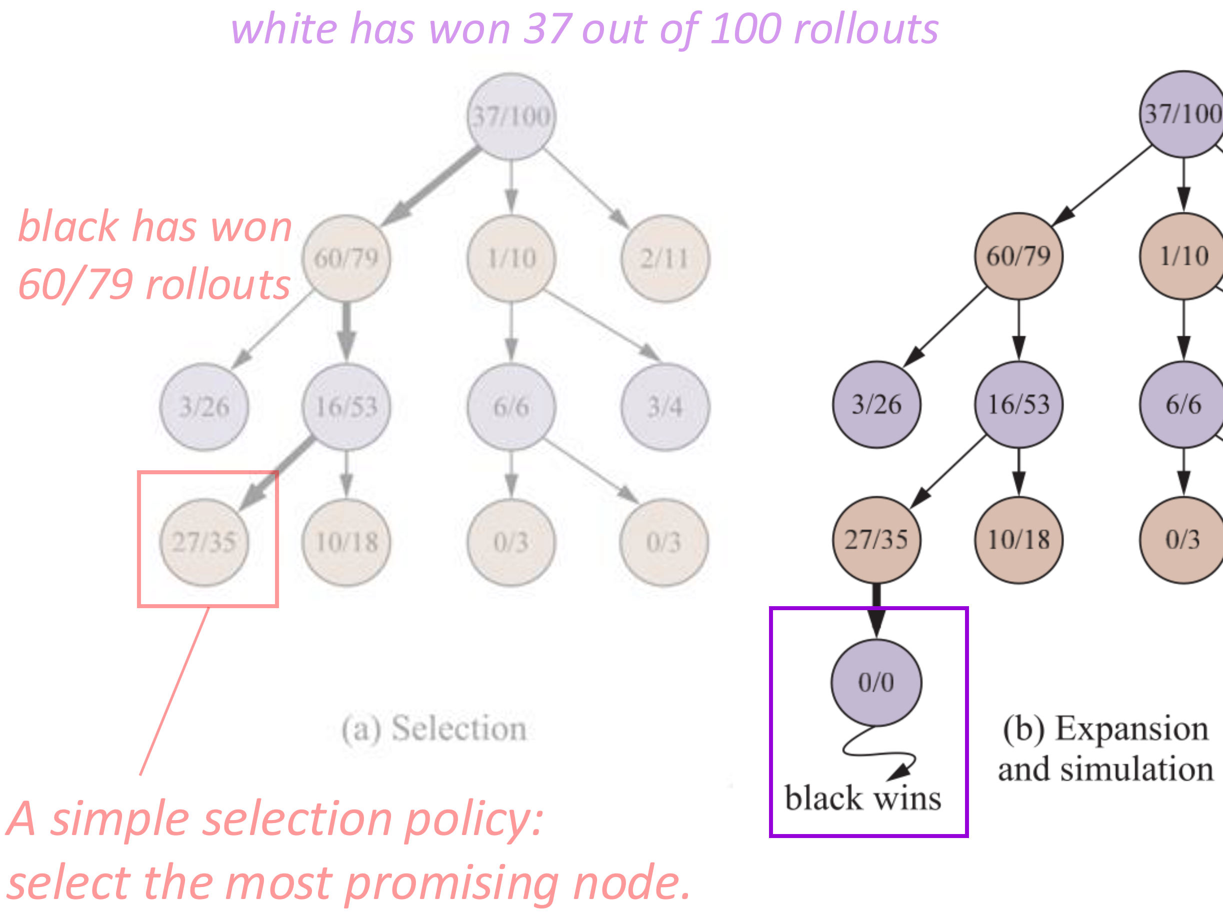
Select:

- A simple selection policy is to select the most promising leaf.
- Win chances for white's first move: (a) 0.76, (b) 0.1, (c) 0.18 → select (a)
- Win chances for black's move: 16/53 = 0.3 > 0.12 = 3/26

- We will look at other selection policies later.

Monte Carlo Tree Search: The Main Idea

- Four steps in one iteration: select, expand, simulate, and back-propagate.



Expand:

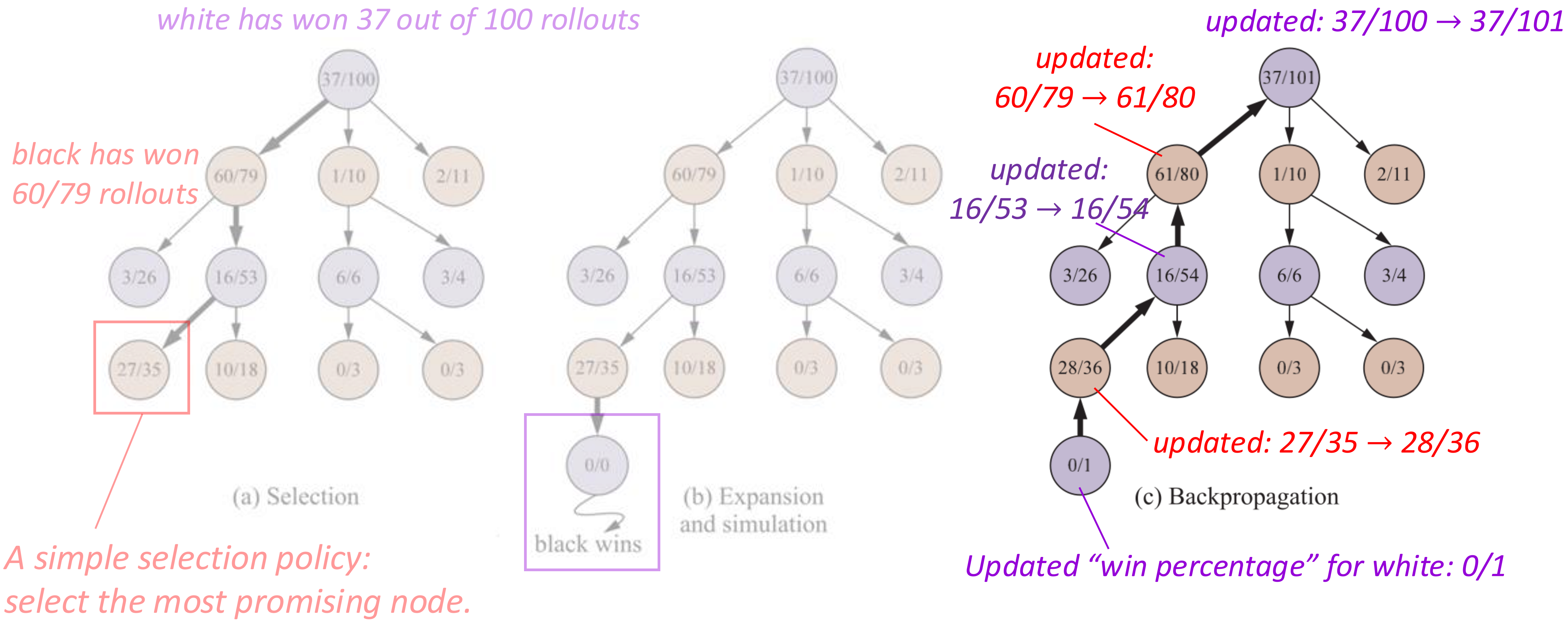
- We grow the search tree by generating a new child.
- "0/0": simulated 0 times and won 0 times.

Simulate:

- Perform a playout from the new node.
- Choose moves for both players according to the playout policy.
- The simplest playout policy: move randomly.
- The moves are NOT recorded in the tree.
- We only record results: *black wins*.

Monte Carlo Tree Search: The Main Idea

- Four steps in one iteration: select, expand, simulate, and back-propagate.



Backpropagation: update win percentage for all ascendants

Monte Carlo Tree Search: The Main Idea

- Monte Carlo Tree Search Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

- *Repeat this process until we consume all time available.*
- Questions: 1) Can we better select a leaf? 2) Can we better simulate?

Selection Policy: Upper Confidence Bounds Applied to Trees (UCT)

- Selection policy is very important: we want to focus on the important parts!
- An effective selection policy: upper confidence bounds applied to trees (UCT). It ranks each possible move based on *an upper confidence bound formula*, **UCB1**.

UCB1

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

$U(n)$: total utility of all playouts what went through node n.
$N(n)$: number of playouts through node n.
$\text{Parent}(n)$: the parent node of n.
C: a constant that balances exploitation and exploration.

$\frac{U(n)}{N(n)}$: average utility (exploitation; if C=0 → select the current best)

$\sqrt{\log \frac{N(\text{Parent}(n))}{N(n)}}$: large when n is small, close to zero when n is large (exploration)

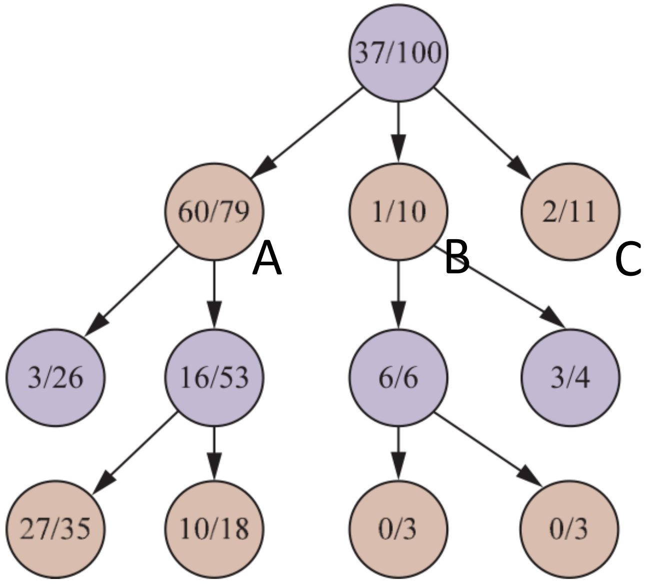
Example of UCT

UCB1

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

$U(n)$: total utility of all playouts what went through node n.
$N(n)$: number of playouts through node n.
$\text{Parent}(n)$: the parent node of n.
C: a constant that balances exploitation and exploration.

- Example: Select among A, B, C using the UCT selection metric (suppose C=1.4).



$$U(A) = 60, N(A) = 79, N(\text{Parent}(A)) = 100$$

$$UCB1(A) = \frac{60}{79} + 1.4 \times \sqrt{\frac{\log 100}{79}} = 1.098$$ *node A will be selected.*

$$UCB1(B) = \frac{1}{10} + 1.4 \times \sqrt{\frac{\log 100}{10}} = 1.05$$

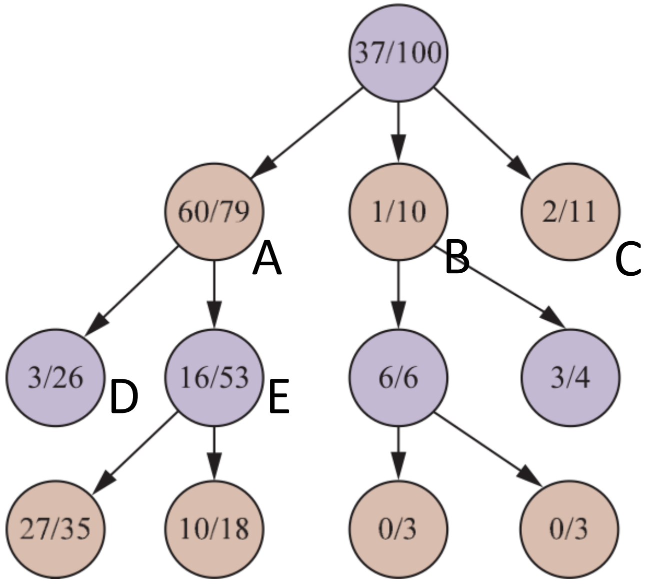
$$UCB1(C) = \frac{2}{11} + 1.4 \times \sqrt{\frac{\log 100}{11}} = 1.088$$

Exercise

UCB1

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

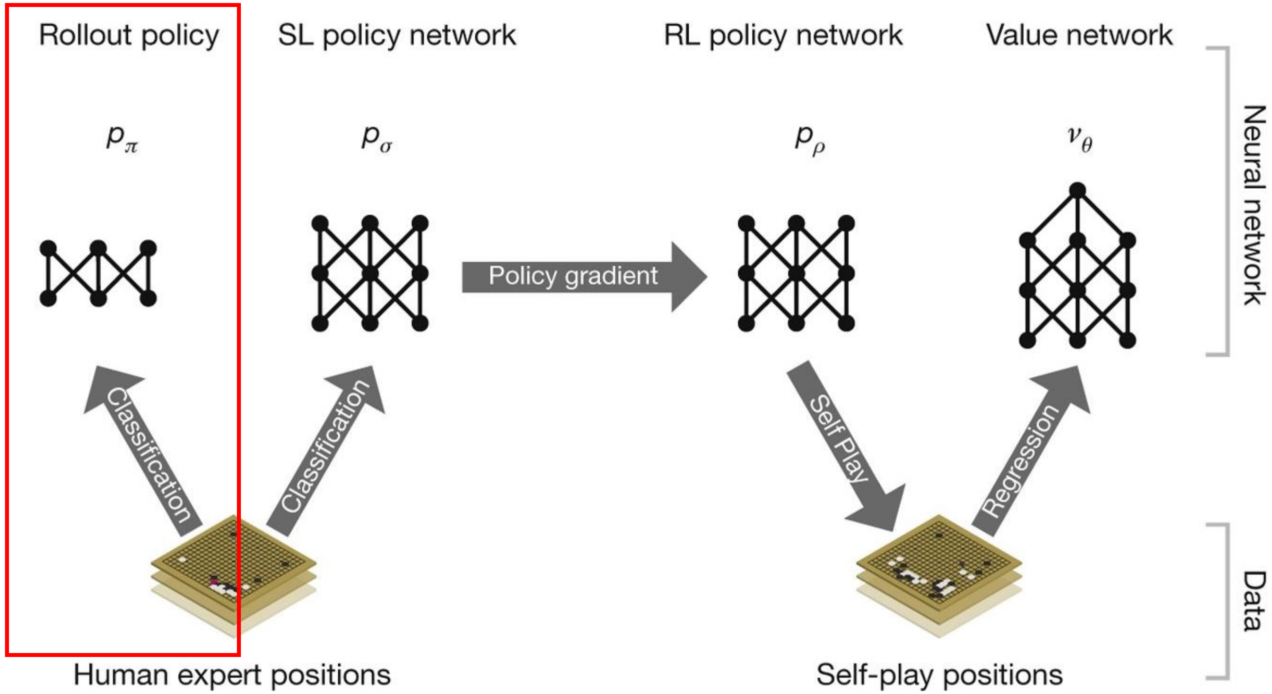
$U(n)$: total utility of all playouts what went through node n.
$N(n)$: number of playouts through node n.
$\text{Parent}(n)$: the parent node of n.
C: a constant that balances exploitation and exploration.



- Exercise:
  - Using UCT with C=1.4, continue the selection process until a leaf is selected.
  - If we use C=1.5, which leaf will be selected?
  - Can you explain the reasons of differences in the selection result when using C=1.4 and C=1.5?

Playout Policy

- The modern way: learning the playout policy by neural network



Example: AlphaGo trains neural networks for a fast rollout policy ρ_π.
It is trained by predicting human moves in a dataset of positions.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.

Demystifying AlphaGo

- AlphaGo is based on MCTS and reinforcement learning.
- Rollout policy: a neural network trained to predict human moves.

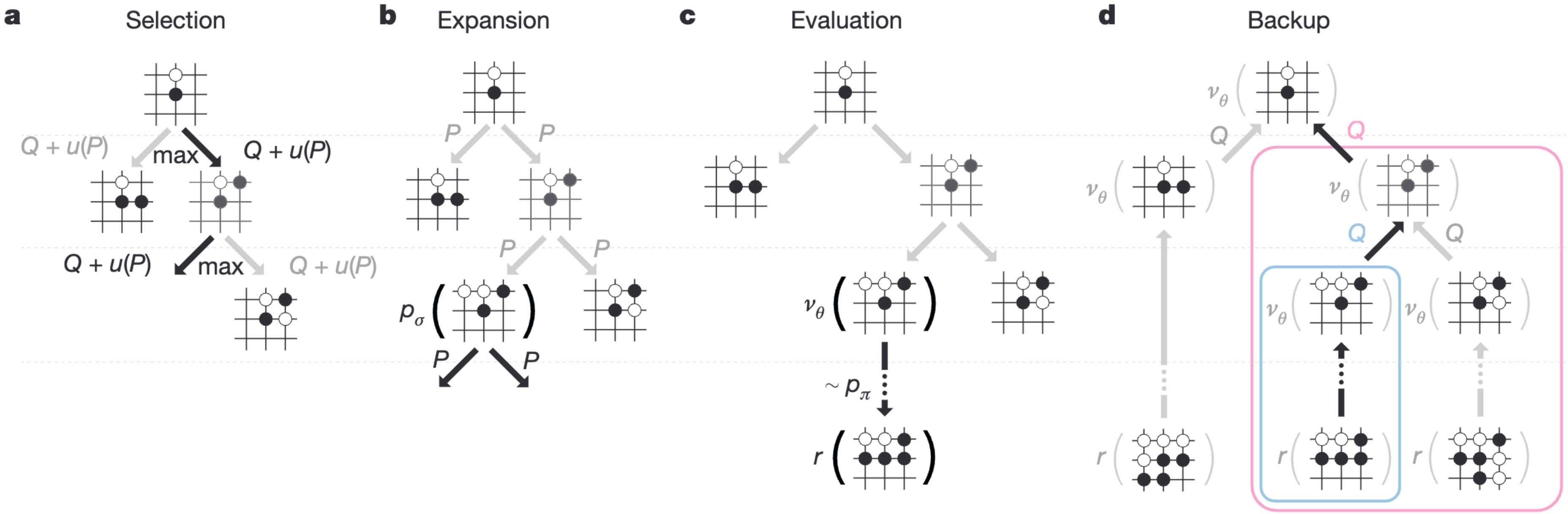


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

Summary of Adversarial Search

- Our journey of playing Go:
  - Formulation of two-player zero-sum deterministic games and game tree.
  - Minimax search algorithm: not feasible to build the entire tree!
  - Alpha-beta pruning: more efficient, but still need to reach to the leaf nodes.
  - Heuristic alpha-beta search:
    - Cut off the tree and use heuristic functions, no need to reach leaf nodes.
    - Manageable for chess, but still not feasible for Go.
    - Need to define good heuristic functions!
  - Monte Carlo Tree Search:
    - Does not use heuristic functions.
    - Four steps: select, expand, simulate, and backpropagate.
    - Select policy: UCT metric and UCB1 formula.

Let your voice be heard!

Thank you for your feedback! 🙌