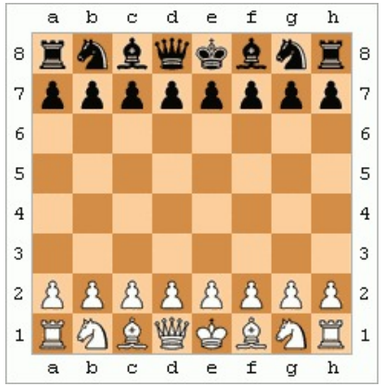
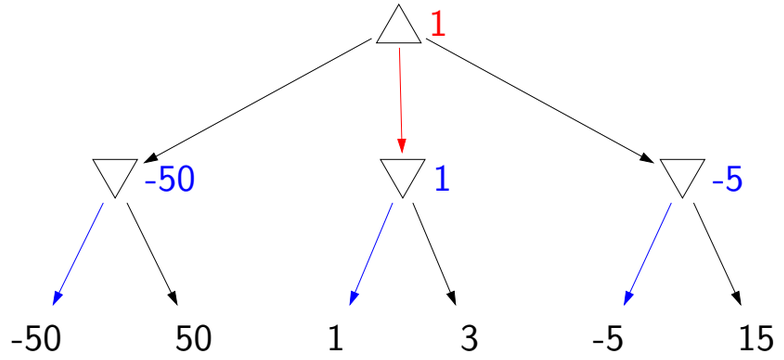
Computation





Approach: tree search

Complexity:

- branching factor b, depth d ($2d$ plies)
- $O(d)$ space, $O(b^{2d})$ time

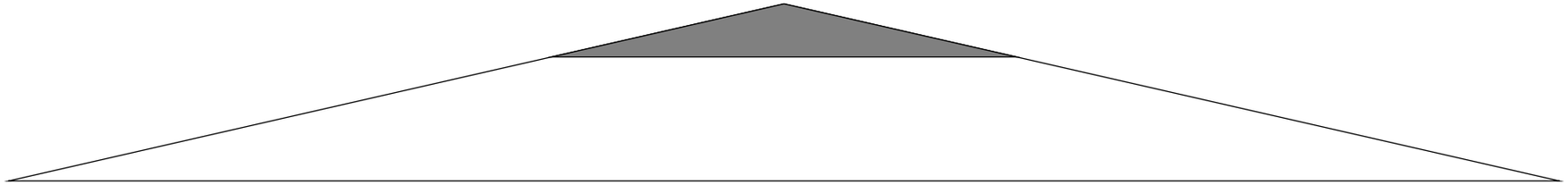
Chess: $b \approxeq 35$, $d \approxeq 50$

255155206729868529241211501514255876301904144881610193241767784407714672582399373658437329870435557897823361956377366532855432978976750746369361877441406 25

Speeding up minimax

- Evaluation functions: use domain-specific knowledge, compute approximate answer
- Alpha-beta pruning: general-purpose, compute exact answer

Depth-limited search



Limited depth tree search (stop at maximum depth d_{max}):

$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d-1) & \text{Player}(s) = \text{opp} \end{cases}$$

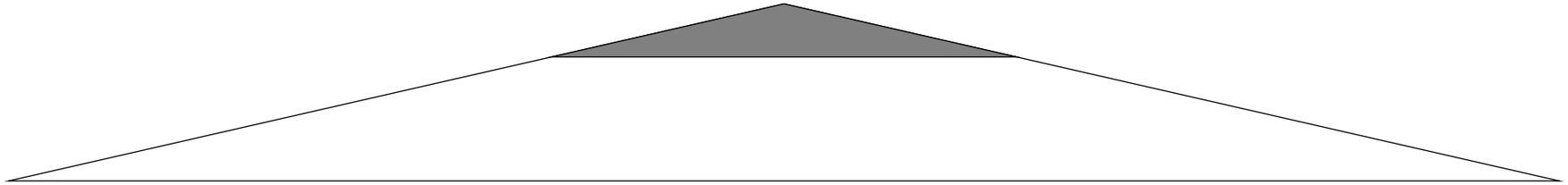
start at $d > 0$ at root, then decrement every time we go down

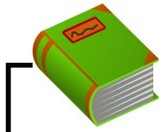
Use: at state s, call $V_{\text{minmax}}(s, d_{\text{max}})$

go though plies

Convention: decrement depth at last player's turn

Evaluation functions





Definition: Evaluation function

An evaluation function Eval(s) is a (possibly very weak) estimate of the value $V_{\text{minmax}}(s)$.

Analogy: FutureCost(s) in search problems

↑ like $h(s)$

Evaluation functions

tells you how good a state is
→ estimate using features

Example: chess

$$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$$

$$\text{material} = 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P')$$

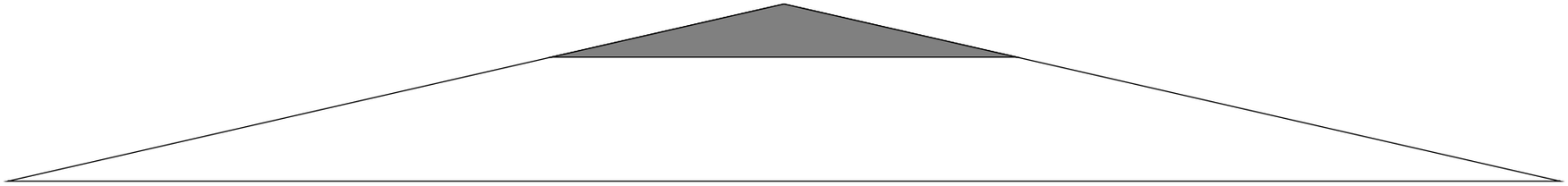
$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

pieces on board, # of kings, etc

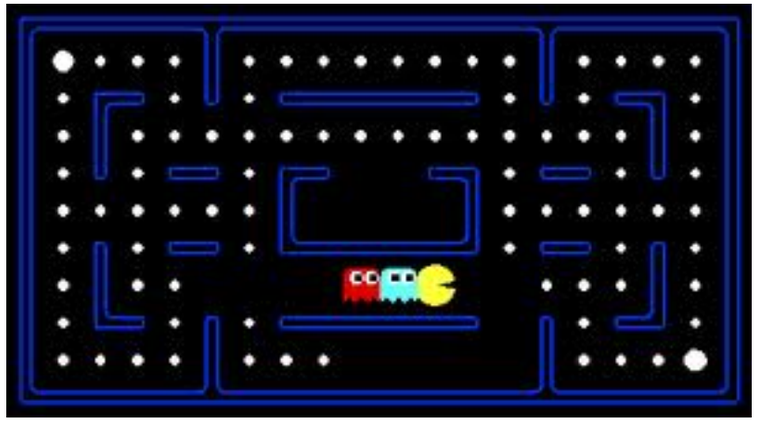
Summary: evaluation functions

Depth-limited exhaustive search: $O(b^{2d})$ time



- Eval(s) attempts to estimate $V_{\text{minmax}}(s)$ using domain knowledge
- No guarantees (unlike A*) on the error from approximation

Games: alpha-beta pruning



Pruning principle

Choose A or B with maximum value:

→ we'd never choose A, because the min of B is the max of A

A: [3, **5**] B: [**5**, 100]

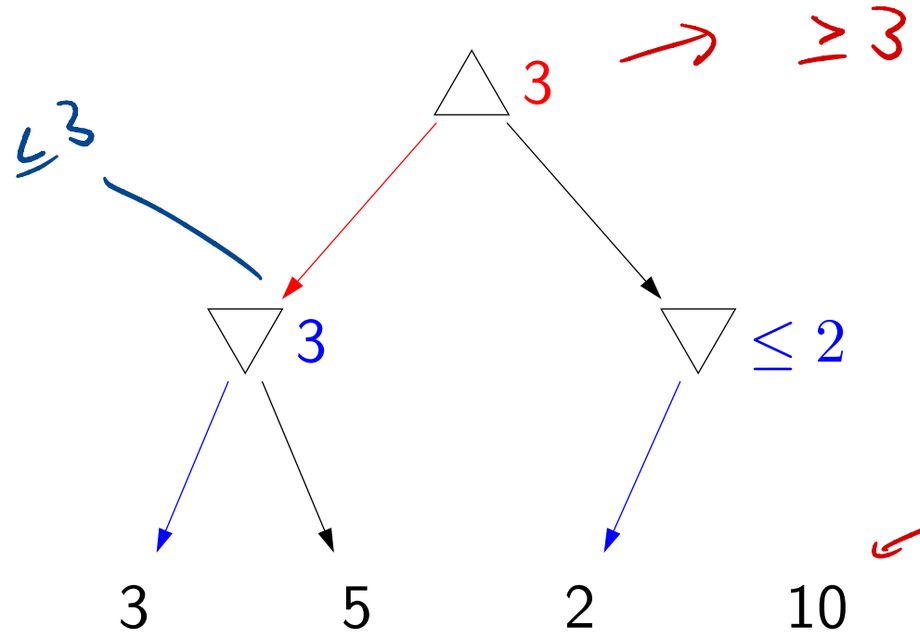
Key idea: branch and bound

Maintain lower and upper bounds on values.

If intervals don't overlap non-trivially, then can choose optimally without further work.

* a_s: lower bound on max nodes →  → max node $\geq a_s$
* b_s: upper bound on min nodes →  → min node $\leq b_s$

Pruning game trees



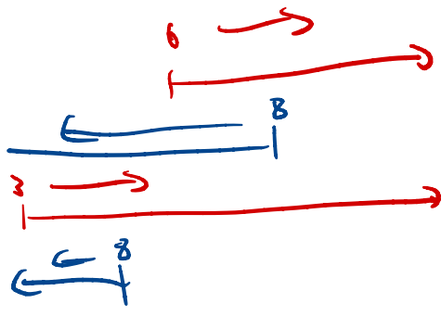
Once see 2, we know that value of right node must be ≤ 2

Root computes $\max(3, \leq 2) = 3$

Since branch doesn't affect root value, can safely prune

- In the context of minimax search, we note that the root node is a max over its children.
- Once we see the left child, we know that the root value must be at least 3.
- Once we get the 2 on the right, we know the right child has to be at most 2.
- Since those two intervals are non-overlapping, we can prune the rest of the right subtree and not explore it.

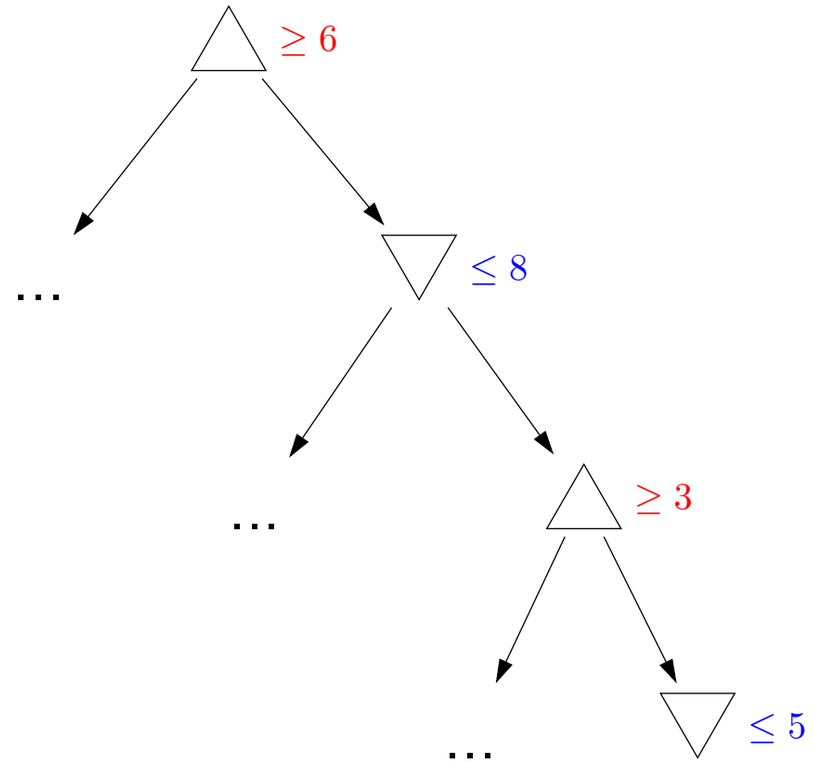
Alpha-beta pruning



Key idea: optimal path

The optimal path is path that minimax policies take.

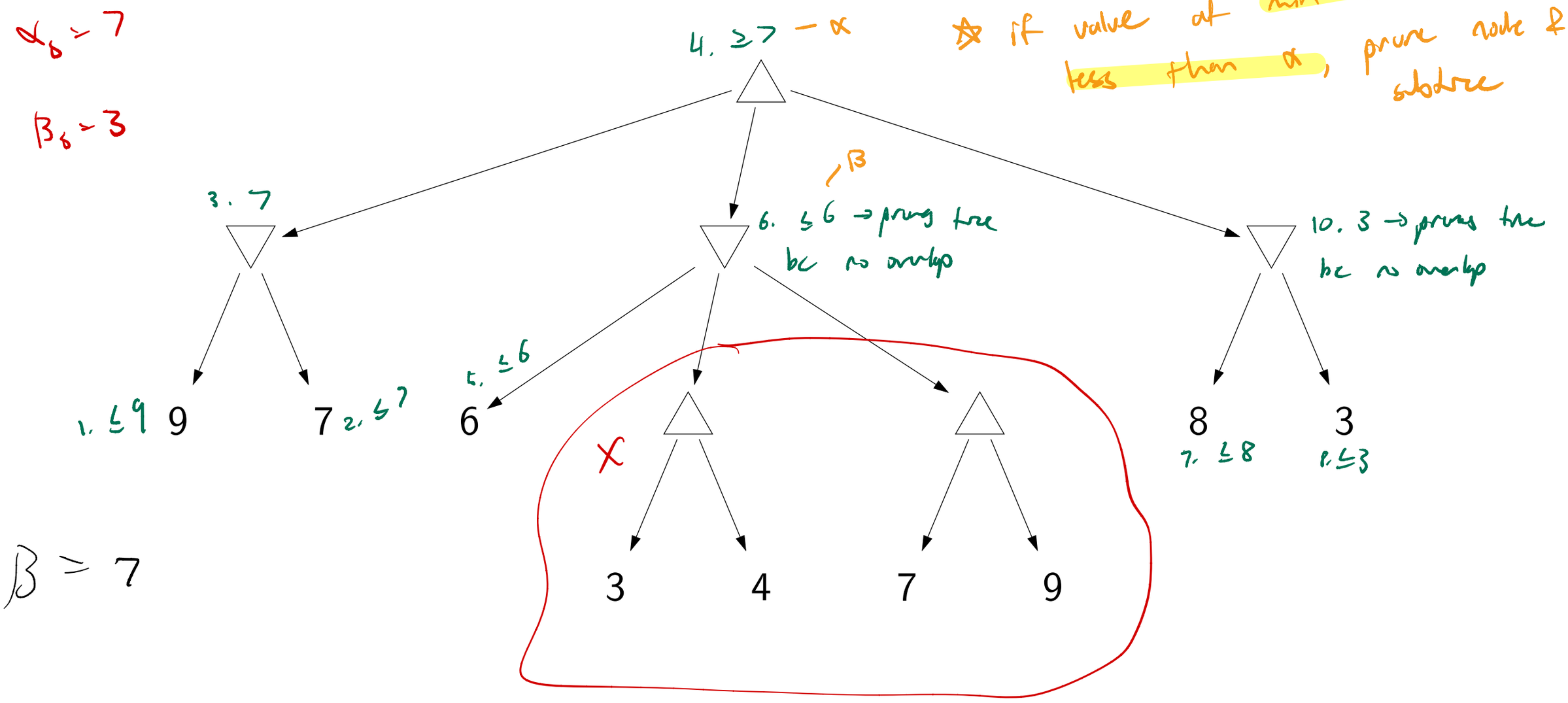
Values of all nodes on path are the same.



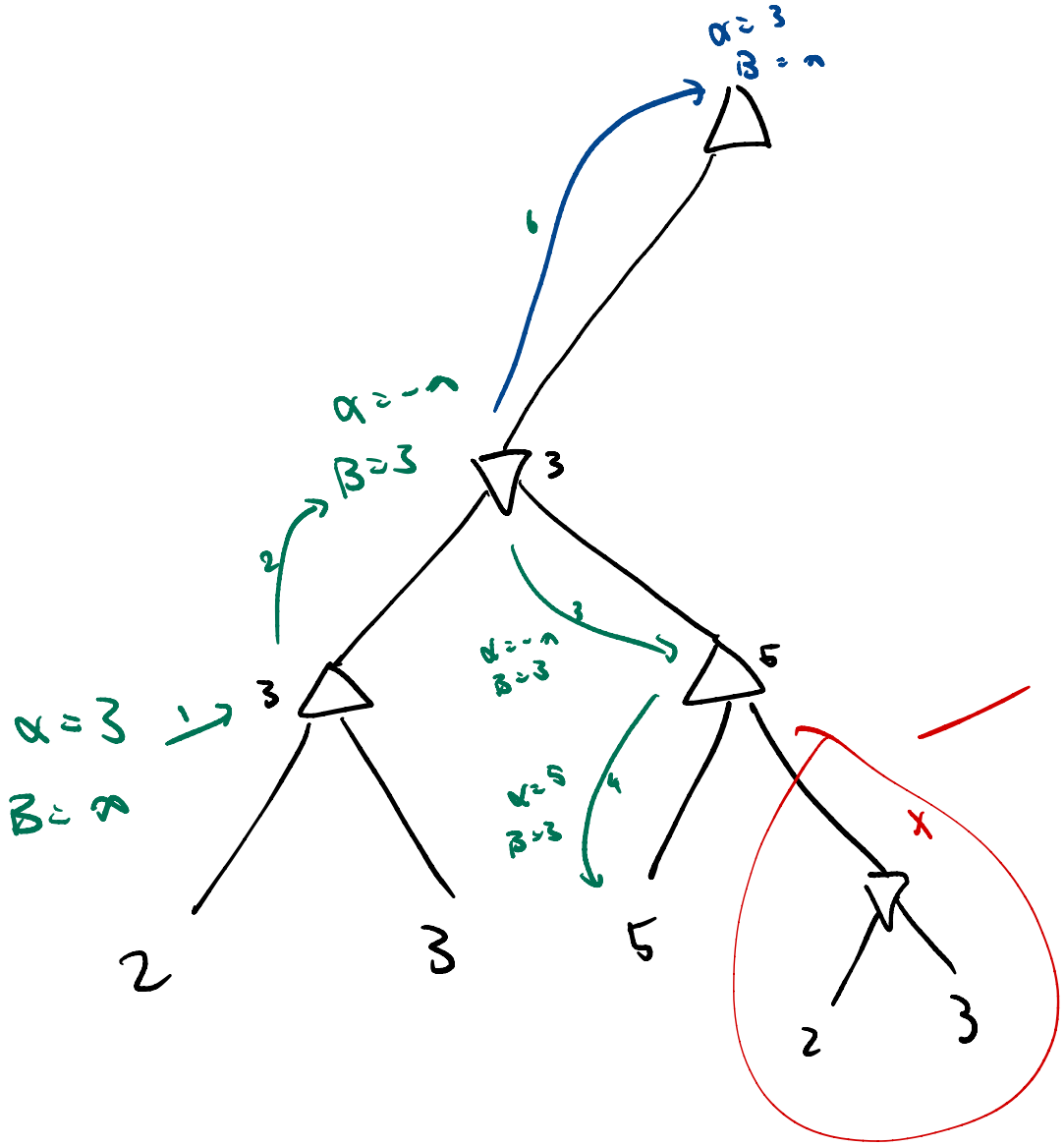
- a_s: lower bound on value of max node s
- b_s: upper bound on value of min node s
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store $\alpha_s = \max_{s' \preceq s} a_{s'}$ and $\beta_s = \min_{s' \preceq s} b_{s'}$)

- In general, let's think about the minimax values in the game tree. The value of a node is equal to the utility of at least one of its leaf nodes (because all the values are just propagated from the leaves with min and max applied to them). Call the first path (ordering by children left-to-right) that leads to the first such leaf node the **optimal path**. An important observation is that the values of all nodes on the optimal path are the same (equal to the minimax value of the root).
- Since we are interested in computing the value of the root node, if we can certify that a node is not on the optimal path, then we can prune it and its subtree.
- To do this, during the depth-first exhaustive search of the game tree, we think about maintaining a lower bound ($\geq a_s$) for all the max nodes s and an upper bound ($\leq b_s$) for all the min nodes s.
- If the interval of the current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node. In the example, we've determined the root's node must be ≥ 6. Once we get to the node on at ply 4 and determine that node is ≤ 5, we can prune the rest of its children since it is impossible that this node will be on the optimal path (≤ 5 and ≥ 6 are incompatible). Remember that all the nodes on the optimal path have the same value.
- Implementation note: for each max node s, rather than keeping a_s, we keep α_s, which is the maximum value of $a_{s'}$ over s and all its max node ancestors. Similarly, for each min node s, rather than keeping b_s, we keep β_s, which is the minimum value of $b_{s'}$ over s and all its min node ancestors. That way, at any given node, we can check interval overlap in constant time regardless of how deep we are in the tree.

Alpha-beta pruning example



★ if value at min node less than α, prune node & subtree



as soon as left subtree is computed w/ value 5, we have value $\geq \beta$ at max node, so prune the rest of the children

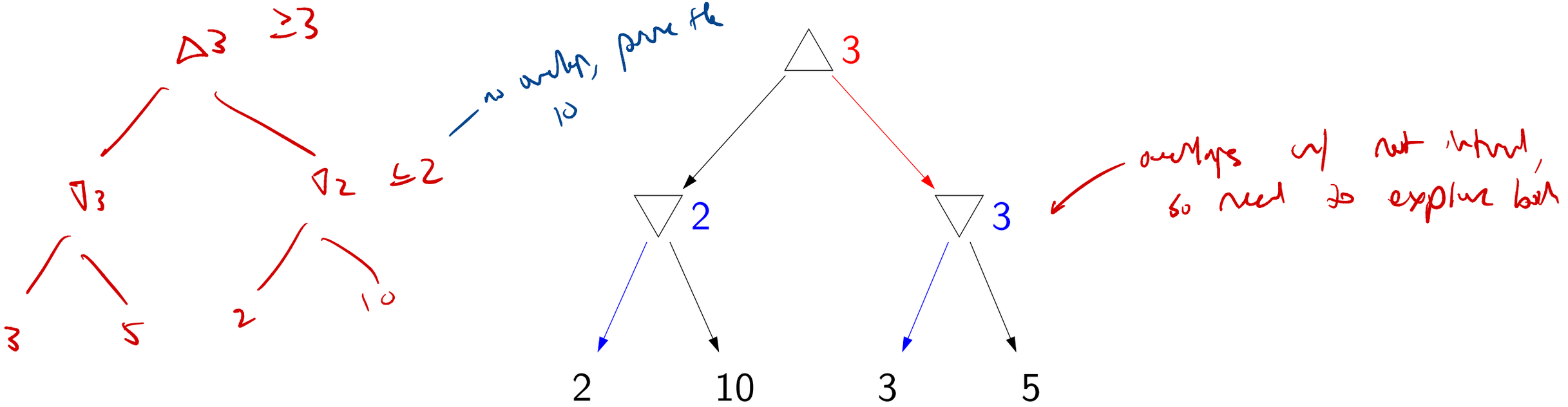
$\Rightarrow$ val of min node $\leq \beta$

$\Rightarrow$ if value $> \beta$, no need to explore further

Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



Move ordering

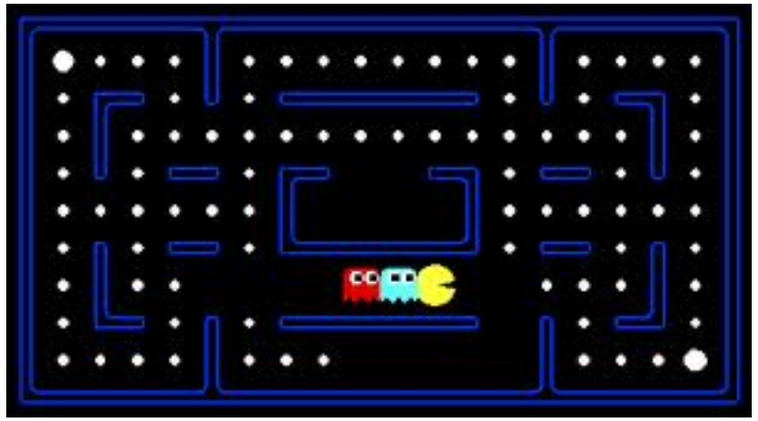
Which ordering to choose?

- Worst ordering: $O(b^{2 \cdot d})$ time
- Best ordering: $O(b^{2 \cdot 0.5d})$ time
- Random ordering: $O(b^{2 \cdot 0.75d})$ time when $b = 2$
- Random ordering: $O((\frac{b-1+\sqrt{b^2+14b+1}}{4})^{2 \cdot d})$ for general b

In practice, can use evaluation function Eval(s):

- Max nodes: order successors by decreasing Eval(s)
- Min nodes: order successors by increasing Eval(s)

Games: TD-learning



learn evaluation automatically from data

Evaluation function

Old: hand-crafted

Example: chess

$$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$$

$$\text{material} = 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P')$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

New: learn from data

$$\text{Eval}(s) = V(s; \mathbf{w})$$

model family

Model for evaluation functions

Linear:

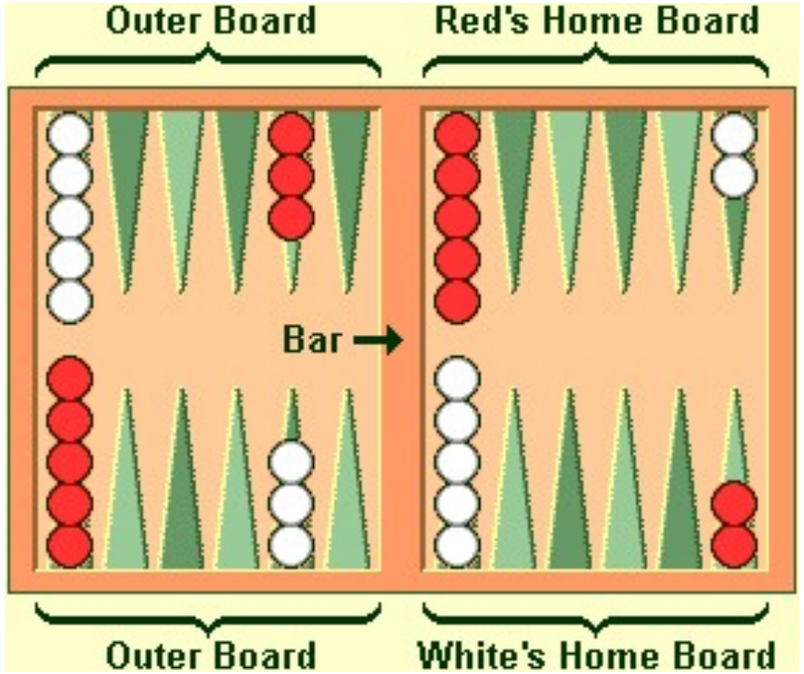
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

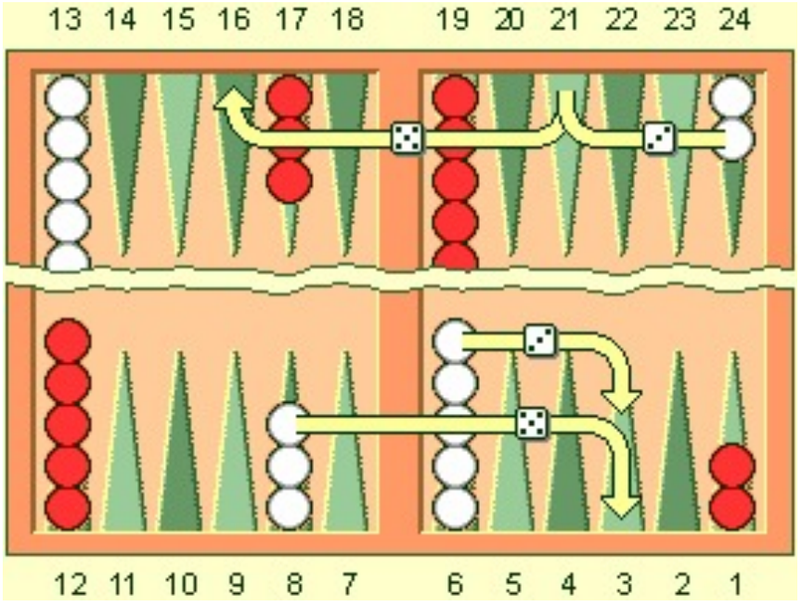
* learn what $\mathbf{w}$ is

Neural network:

$$V(s; \mathbf{w}, \mathbf{v}_{1:k}) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(s))$$

Example: Backgammon

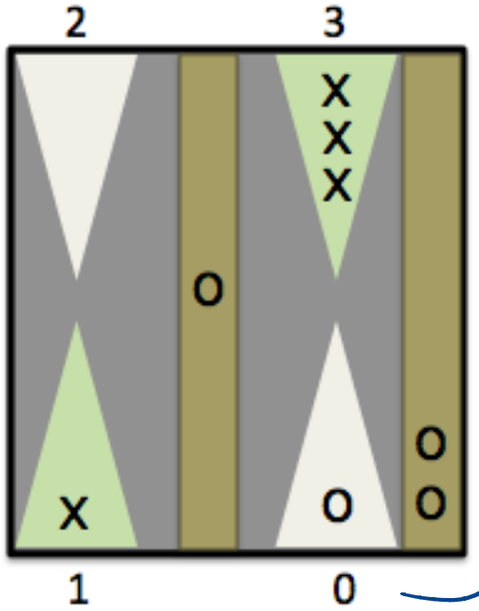




- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to move forward that many positions.
- You cannot land on a position containing more than one opponent piece. If you land on exactly one opponent piece, then that piece goes on the bar and has start over from the beginning. (See the Wikipedia article for the full rules.).

Features for Backgammon

state s



Features $\phi(s)$:

$[(\#\text{ o in column } 0) = 1]$: 1

$[(\#\text{ o on bar})]$: 1

$[(\text{fraction o removed})]$: $\frac{1}{2}$

$[(\#\text{ x in column } 1) = 1]$: 1

$[(\#\text{ x in column } 3) = 3]$: 1

$[(\text{is it o's turn})]$: 1

- As an example, we can define the following features for Backgammon, which are inspired by the ones used by TD-Gammon.
- Note that the features are pretty generic; there is no explicit modeling of strategies such as trying to avoid having singleton pieces (because it could get clobbered) or preferences for how the pieces are distributed across the board.
- On the other hand, the features are mostly **indicator** features, which is a common trick to allow for more expressive functions using the machinery of linear regression. For example, instead of having one feature whose value is the number of pieces in a particular column, we can have multiple features for indicating whether the number of pieces is over some threshold.

Generating data

Generate using policies based on current $V(s; \mathbf{w})$:

$$\pi_{\text{agent}}(s; \mathbf{w}) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

$$\pi_{\text{opp}}(s; \mathbf{w}) = \arg \min_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

use these to get later

Note: don't need to randomize (ϵ-greedy) because game is already stochastic (backgammon has dice) and there's function approximation

Generate episode:

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

s_0, take action a_1 and get reward r_1, end up in s_1

Learning algorithm

Episode:

$s_0; a_1, r_1, s_1; a_2, r_2, s_2, a_3, r_3, s_3; \ldots, a_n, r_n, s_n$

A small piece of experience:

(s, a, r, s')

Prediction:

$V(s; \mathbf{w})$

Target:

$r + \gamma V(s'; \mathbf{w})$

General framework

Objective function:

$$\text{Loss}(\mathbf{w}) = \frac{1}{2}(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Gradient:

$$(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})$$

Update:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})}_{\text{gradient}}$$

gradient descent

Temporal difference (TD) learning

Algorithm: TD learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{V(s;\mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma V(s';\mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} V(s;\mathbf{w})$$

For linear functions:

$$V(s;\mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

$$\nabla_{\mathbf{w}} V(s;\mathbf{w}) = \phi(s)$$

Linear case:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\,(\text{Pred}(w) - \text{Target}) \cdot \nabla_{\mathbf{w}} \text{Pred}(w)$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\,(V(s;w) - (r + \gamma V(s';w)) \nabla_{\mathbf{w}} V(s;w)$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\,(\mathbf{w} \cdot \phi(s) - (r + \gamma\, \mathbf{w} \cdot \phi(s')) \cdot \phi(s)$$

Example of TD learning

Step size $\eta = 0.5$, discount $\gamma = 1$, reward is end utility

$w \leftarrow w - \eta(p - t) \cdot \phi(s)$

$p = w \cdot \phi(s)$

$t = r + \gamma\, w \cdot \phi(s')$

Example: TD learning

S1	r:0	S4	r:0	S8	r:1	S9
$\phi: \begin{pmatrix}0\\1\end{pmatrix}$	1 step	$\phi: \begin{pmatrix}1\\0\end{pmatrix}$		$\phi: \begin{pmatrix}1\\2\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$
$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:0 p-t:0	$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:0 p-t:0	$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:1 p-t:-1	$\mathbf{w}: \begin{pmatrix}0.5\\1\end{pmatrix}$

episode 1

S1	r:0	S2	r:0	S6	r:0	S10
$\phi: \begin{pmatrix}0\\1\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$		$\phi: \begin{pmatrix}0\\0\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$
$\mathbf{w}: \begin{pmatrix}0.5\\1\end{pmatrix}$	p:1 t:0.5 p-t:0.5	$\mathbf{w}: \begin{pmatrix}0.5\\0.75\end{pmatrix}$	p:0.5 t:0 p-t:0.5	$\mathbf{w}: \begin{pmatrix}0.25\\0.75\end{pmatrix}$	p:0 t:0.25 p-t:-0.25	$\mathbf{w}: \begin{pmatrix}0.25\\0.75\end{pmatrix}$

episode 2

- Here's an example of TD learning in action. We have two episodes: [S1, 0, S4, 0, S8, 1, S9] and [S1, 0, S2, 0, S6, 0, S10].
- In games, all the reward comes at the end and the discount is 1. We have omitted the action because TD learning doesn't depend on the action.
- Under each state, we have written its feature vector, and the weight vector before updating on that state. Note that no updates are made until the first non-zero reward. Our prediction is 0, and the target is $1 + 0$, so we subtract $-0.5[1, 2]$ from the weights to get $[0.5, 1]$.
- In the second row, we have our second episode, and now notice that even though all the rewards are zero, we are still making updates to the weight vectors since the prediction and targets computed based on adjacent states are different.

Comparison

Algorithm: TD learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$

Algorithm: Q-learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

Comparison

Q-learning:

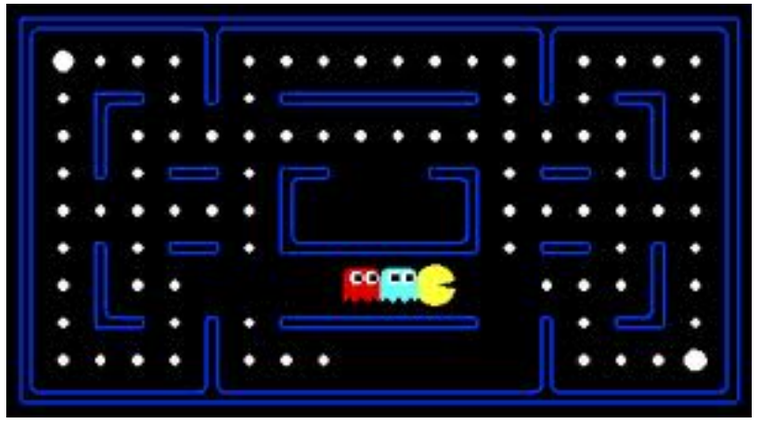
- Operate on $\hat{Q}_{\text{opt}}(s, a; \mathbf{w})$ — dependent on action
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions $T(s, a, s')$

TD learning:

- Operate on $\hat{V}_\pi(s; \mathbf{w})$
- On-policy: value is based on exploration policy (usually based on $\hat{V}_\pi$)
- To use, need to know rules of the game $\text{Succ}(s, a)$

- TD learning is very similar to Q-learning. Both algorithms learn from the same data and are based on gradient-based weight updates.
- The main difference is that Q-learning learns the Q-value, which measures how good an action is to take in a state, whereas TD learning learns the value function, which measures how good it is to be in a state.
- Q-learning is an off-policy algorithm, which means that it tries to compute Q_{opt}, associated with the optimal policy (not Q_π), whereas TD learning is on-policy, which means that it tries to compute V_π, the value associated with a fixed policy π. Note that the action a does not show up in the TD updates because a is given by the fixed policy π. Of course, we usually are trying to optimize the policy, so we would set π to be the current guess of optimal policy $\pi(s) = \arg\max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$.
- When we don't know the transition probabilities and in particular the successors, the value function isn't enough, because we don't know what effect our actions will have. However, in the game playing setting, we do know the transitions (the rules of the game), so using the value function is sufficient.

Games: simultaneous games



☆ no ordering of player moves

answer in chat

Question

For a simultaneous two-player zero-sum game (like rock-paper-scissors), can you still be optimal if you reveal your strategy?

yes

no

you can reveal that you are playing stochastically / randomly, while still be optimal

Two-finger Morra

Example: two-finger Morra

Players A and B each show 1 or 2 fingers.

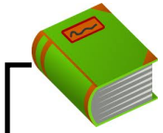
If both show 1, B gives A 2 dollars.

If both show 2, B gives A 4 dollars.

Otherwise, A gives B 3 dollars.

[play with a partner]

Payoff matrix



Definition: single-move simultaneous game

Players $= \{\text{A}, \text{B}\}$

Actions: possible actions

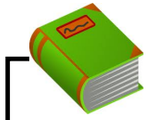
$V(a, b)$: **A's utility** if A chooses action a, B chooses b

(let V be **payoff matrix**)

Example: two-finger Morra payoff matrix

A \ B	1 finger	2 fingers
1 finger	2	-3
2 fingers	-3	4

Strategies (policies)

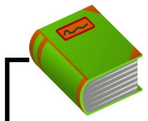


Definition: pure strategy

A pure strategy is a single action:

$a \in \text{Actions}$

→ always take a single action deterministically



Definition: mixed strategy

A mixed strategy is a probability distribution

$0 \leq \pi(a) \leq 1$ for $a \in \text{Actions}$

→ stochastically play actions

Example: two-finger Morra strategies

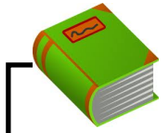
$\pi(1)$ — $\pi(2)$

Always 1: $\pi = [1, 0]$

Always 2: $\pi = [0, 1]$

Uniformly random: $\pi = [\frac{1}{2}, \frac{1}{2}]$

Game evaluation



Definition: game evaluation

The **value** of the game if player A follows π_A and player B follows π_B is

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)$$

→ sum over possibilities

$V(\pi_A, \pi_B) = \pi_A^T V \pi_B$

Example: two-finger Morra

Player A always chooses 1: $\pi_A = [1, 0]$

Player B picks randomly: $\pi_B = [\frac{1}{2}, \frac{1}{2}]$

Value: $\boxed{-\frac{1}{2}}$

[whiteboard: matrix]

$\frac{1}{2}(2) + 0 + 0 + \frac{1}{2}(-3) = -\frac{1}{2}$

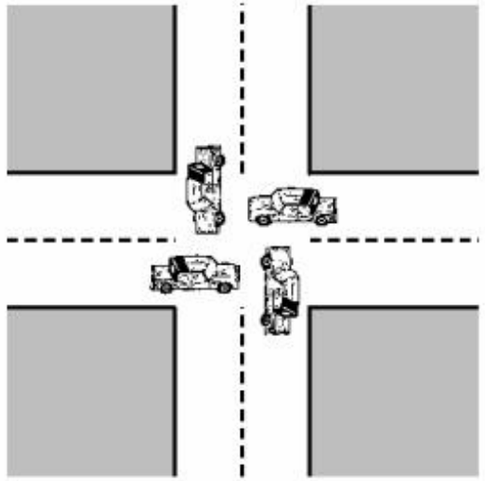
How to optimize?

Game value:

$$V(\pi_A, \pi_B)$$

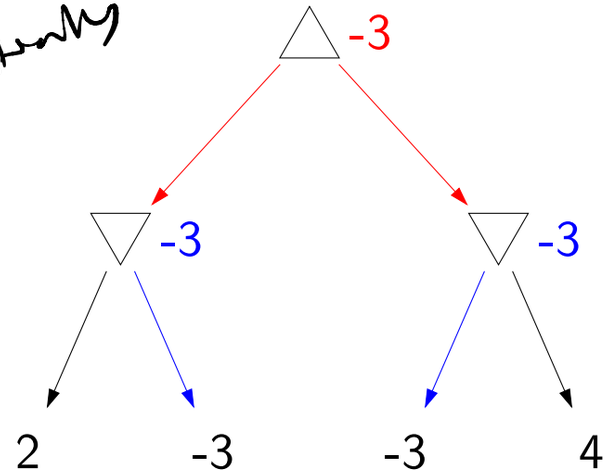
Challenge: player A wants to maximize, player B wants to minimize...

simultaneously

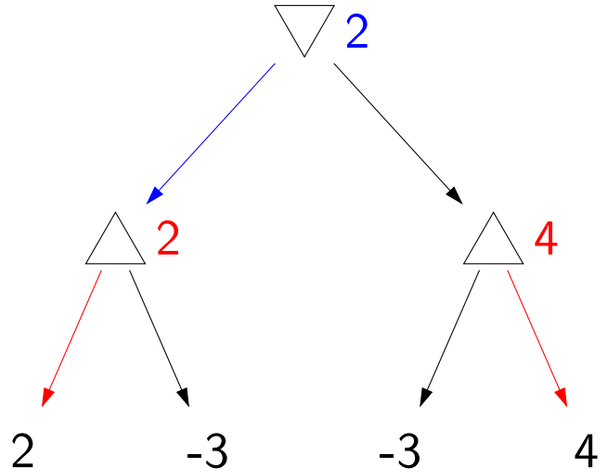


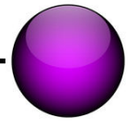
Pure strategies: who goes first?

Player A goes first:



Player B goes first:





Proposition: going second is no worse

$$\max_a \min_b V(a,b) \leq \min_b \max_a V(a,b)$$

Mixed strategies

Example: two-finger Morra

Player A reveals: $\pi_A = [\frac{1}{2}, \frac{1}{2}]$

Value $V(\pi_A, \pi_B) = \pi_B(1)(-\frac{1}{2}) + \pi_B(2)(+\frac{1}{2})$

Optimal strategy for player B is $\pi_B = [1, 0]$ (**pure!**)

Proposition: second player can play pure strategy

For any fixed mixed strategy π_A:

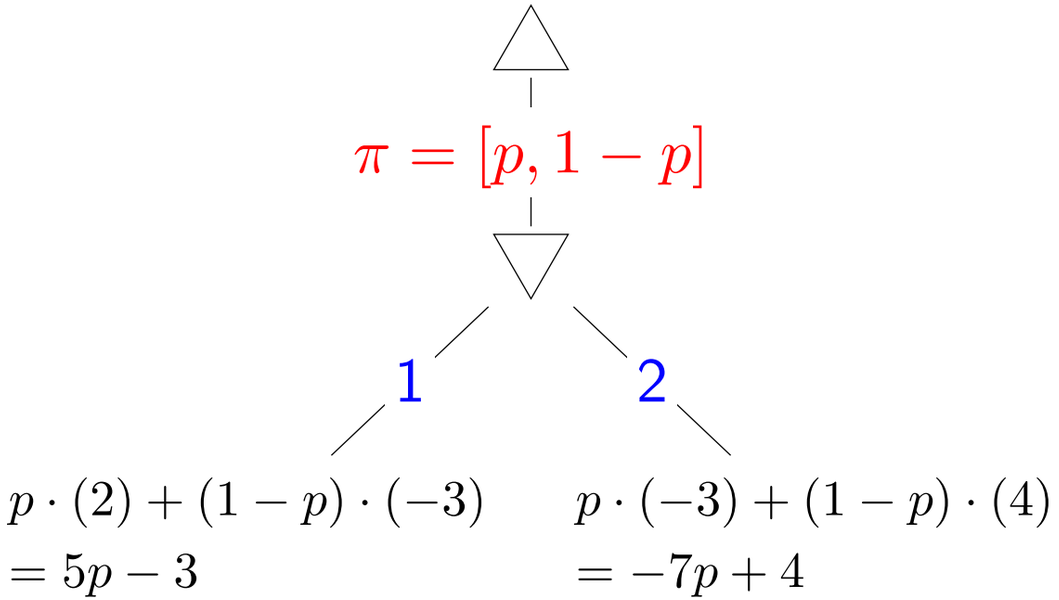
$$\min_{\pi_B} V(\pi_A, \pi_B)$$

can be attained by a pure strategy.

* once stochastic π_A is fixed, π_B is pure

Mixed strategies

Player A first reveals his/her mixed strategy



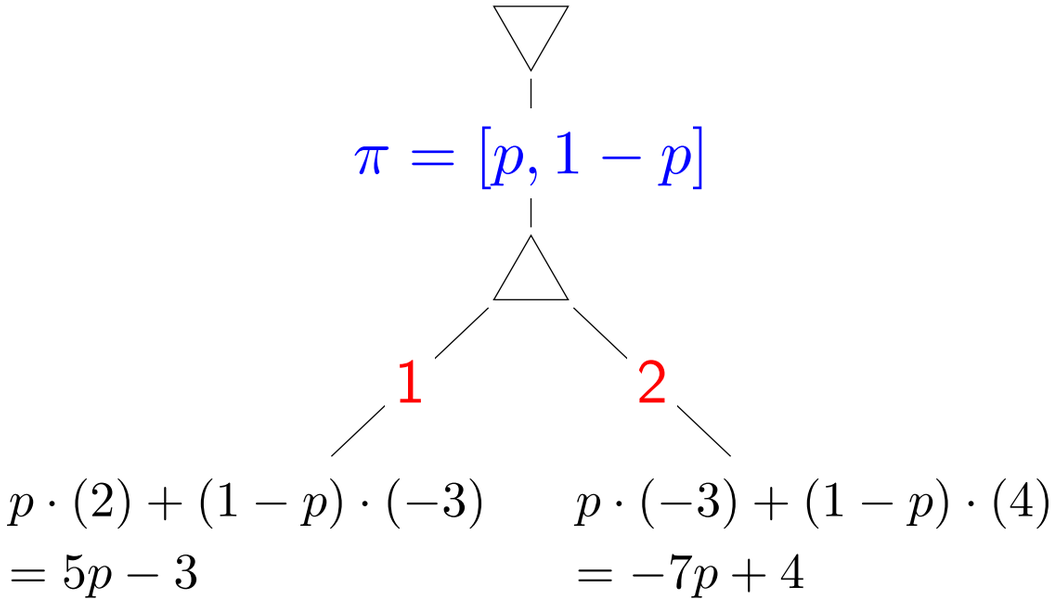
Minimax value of game:

$$\max_{0 \le p \le 1} \min\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \tfrac{7}{12}\text{)}$$

- Now let us try to draw the minimax game tree where the player A first chooses a mixed strategy, and then player B chooses a pure strategy.
- There are an uncountably infinite number of mixed strategies for player A, but we can summarize all of these actions by writing a single action template $\pi = [p, 1-p]$.
- Given player A's action, we can compute the value if player B either chooses 1 or 2. For example, if player B chooses 1, then the value of the game is $5p - 3$ (with probability p, player A chooses 1 and the value is 2; with probability $1 - p$ the value is -3). If player B chooses action 2, then the value of the game is $-7p + 4$.
- The value of the min node is $F(p) = \min\{5p - 3, -7p + 4\}$. The value of the max node (and thus the minimax value of the game) is $\max_{0 \leq 1 \leq p} F(p)$.
- What is the best strategy for player A then? We just have to find the p that maximizes $F(p)$, which is the minimum over two linear functions of p. If we plot this function, we will see that the maximum of $F(p)$ is attained when $5p - 3 = -7p + 4$, which is when $p = \frac{7}{12}$. Plugging that value of p back in yields $F(p) = -\frac{1}{12}$, the minimax value of the game if player A goes first and is allowed to choose a mixed strategy.
- Note that if player A decides on $p = \frac{7}{12}$, it doesn't matter whether player B chooses 1 or 2; the payoff will be the same: $-\frac{1}{12}$. This also means that whatever mixed strategy (over 1 and 2) player B plays, the payoff would also be $-\frac{1}{12}$.

Mixed strategies

Player B first reveals his/her mixed strategy

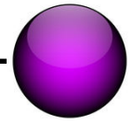


Minimax value of game:

$$\min_{p \in [0,1]} \max\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \tfrac{7}{12}\text{)}$$

- Now let us consider the case where player B chooses a mixed strategy $\pi = [p, 1-p]$ first. If we perform the analogous calculations, we'll find that we get that the minimax value of the game is exactly the same ($-\frac{1}{12}$)!
- Recall that for pure strategies, there was a gap between going first and going second, but here, we see that for mixed strategies, there is no such gap, at least in this example.
- Here, we have been computed minimax values in the conceptually same manner as we were doing it for turn-based games. The only difference is that our actions are mixed strategies (represented by a probability distribution) rather than discrete choices. We therefore introduce a variable (e.g., p) to represent the actual distribution, and any game value that we compute below that variable is a function of p rather than a specific number.

General theorem



Theorem: minimax theorem [von Neumann, 1928]

For every simultaneous two-player zero-sum game with a finite number of actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B),$$

where π_A, π_B range over **mixed strategies**.

Upshot: revealing your optimal mixed strategy doesn't hurt you!

Proof: linear programming duality

Algorithm: compute policies using linear programming

- It turns out that having no gap is not a coincidence, and is actually one of the most celebrated mathematical results: the von Neumann minimax theorem. The theorem states that for any simultaneous two-player zero-sum game with a finite set of actions (like the ones we've been considering), we can just swap the min and the max: it doesn't matter which player reveals his/her strategy first, as long as their strategy is optimal. This is significant because we were stressing out about how to analyze the game when two players play simultaneously, but now we find that both orderings of the players yield the same answer. It is important to remember that this statement is true only for mixed strategies, not for pure strategies.
- This theorem can be proved using linear programming duality, and policies can be computed also using linear programming. The sketch of the idea is as follows: recall that the optimal strategy for the second player is always deterministic, which means that the $\max_{\pi_A} \min_{\pi_B} \cdots$ turns into $\max_{\pi_A} \min_b \cdots$. The min is now over n actions, and can be rewritten as n linear constraints, yielding a linear program.
- As an aside, recall that we also had a minimax result for turn-based games, where the max and the min were over agent and opponent policies, which map states to actions. In that case, optimal policies were always deterministic because at each state, there is only one player choosing.

Summary

- Challenge: deal with simultaneous min/max moves
- Pure strategies: going second is better
- Mixed strategies: doesn't matter (von Neumann's minimax theorem)