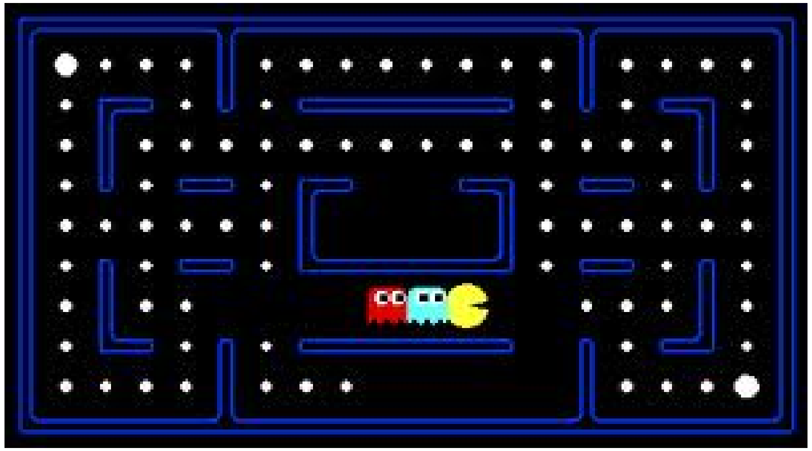
Lecture 9: Games I



cs221.stanford.edu/q

Question

Is it possible to define a policy that is optimal against all possible opponents, however adversarial?

yes
no

Sentiment competition results

1. **Kevin Chen**

   [74.5%] bigrams/trigrams, stemming

2. **Shutong Zhang**

   [74.4%] bigrams, stemming, remove stop words

3. **Vaibhav Singh**

   [73.9%] bigrams, stemming

Course plan

Search problems

Markov decision processes

$\boxed{\text{Adversarial games}}$

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning

- This lecture will be about games, which have been one of the main testbeds for developing AI programs since the early days of AI. Games are distinguished from the other tasks that we've considered so far in this class in that they make explicit the presence of other parties. Thus, the optimal strategy (policy) for us will depend on the strategies of the opponents, and moreover, their strategies are often unknown and adversarial. This makes life difficult!

A simple game

Example: game 1

You choose one of the three bins.

I choose a number from that bin.

Your goal is to maximize the chosen number.

A	**B**	**C**
-50 50	1 3	-5 15

- Which bin should you pick? Depends on your mental model of me.
- If you think I'm working with you (unlikely), then you should pick A in hopes of getting 50. If you think I'm against you (likely), then you should pick B as to guard against the worst case (get 1). If you think I'm just acting randomly, then you should pick C so that on average things are reasonable (get 5 in expectation).

Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

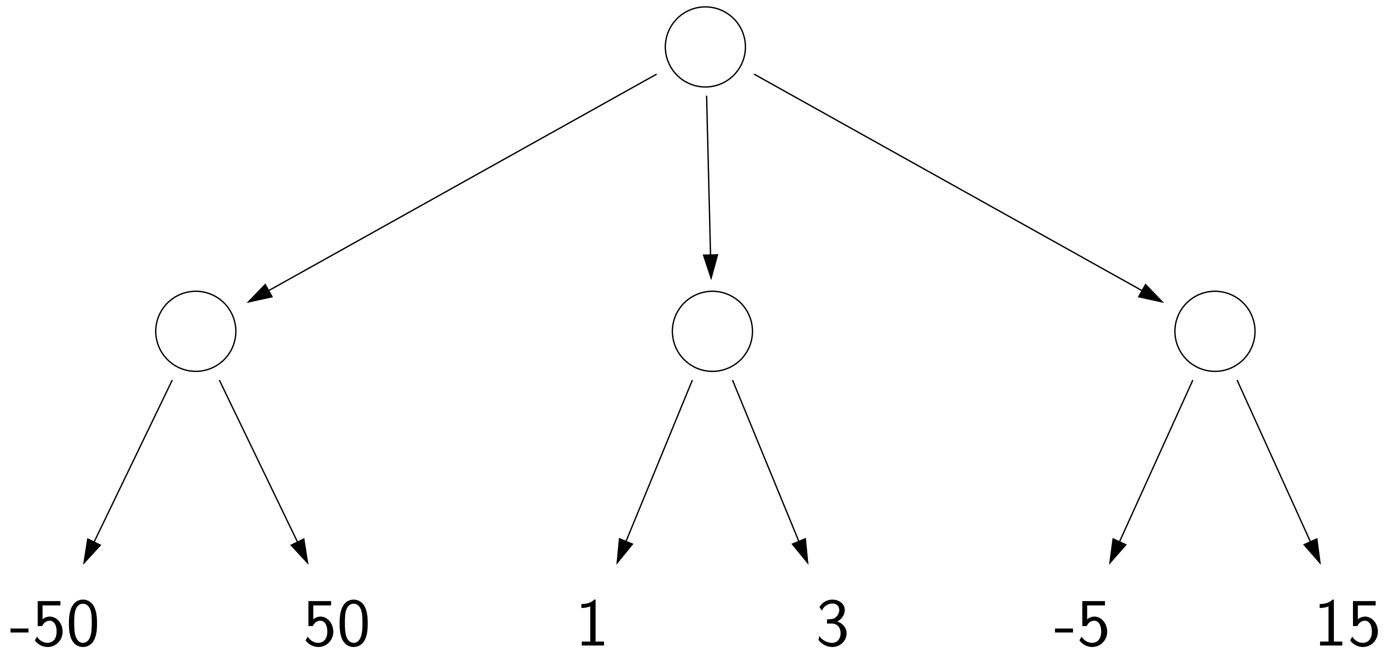
Alpha-beta pruning

Game tree

Key idea: game tree

Each node is a decision point for a player.

Each root-to-leaf path is a possible outcome of the game.



- Just as in search problems, we will use a tree to describe the possibilities of the game. This tree is known as a **game tree**.
- Note: We could also think of a game graph to capture the fact that there are multiple ways to arrive at the same game state. However, all our algorithms will operate on the tree rather than the graph since games generally have enormous state spaces, and we will have to resort to algorithms similar to backtracking search for search problems.

Two-player zero-sum games

Players $= \{\text{agent}, \text{opp}\}$

Definition: two-player zero-sum game

s_{start}: starting state

Actions(s): possible actions from state s

Succ(s, a): resulting state if choose action a in state s

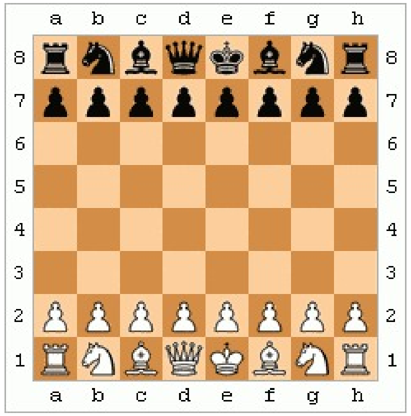
IsEnd(s): whether s is an end state (game over)

Utility(s): agent's utility for end state s

Player$(s) \in$ Players: player who controls state s

- In this lecture, we will specialize to **two-player zero-sum** games, such as chess. To be more precise, we will consider games in which people take turns (unlike rock-paper-scissors) and where the state of the game is fully-observed (unlike poker, where you don't know the other players' hands). By default, we will use the term **game** to refer to this restricted form.
- We will assume the two players are named agent (this is your program) and opp (the opponent). Zero-sum means that the utility of the agent is negative the utility of the opponent (equivalently, the sum of the two utilities is zero).
- Following our approach to search problems and MDPs, we start by formalizing a game. Since games are a type of state space model, much of the skeleton is the same: we have a start state, actions from each state, a deterministic successor state for each state-action pair, and a test on whether a state is at the end.
- The main difference is that each state has a designated Player(s), which specifies whose turn it is. A player p only gets to choose the action for the states s such that Player$(s) = p$.
- Another difference is that instead of having edge costs in search problems or rewards in MDPs, we will instead have a utility function Utility(s) defined only at the end states. We could have used edge costs and rewards for games (in fact, that's strictly more general), but having all the utility at the end states emphasizes the all-or-nothing aspect of most games. You don't get utility for capturing pieces in chess; you only get utility if you win the game. This ultra-delayed utility makes games hard.

Example: chess



Players $= \{\text{white}, \text{black}\}$

State s: (position of all pieces, whose turn it is)

Actions(s): legal chess moves that Player(s) can make

IsEnd(s): whether s is checkmate or draw

Utility(s): $+\infty$ if white wins, 0 if draw, $--$ if black wins

- Chess is a canonical example of a two-player zero-sum game. In chess, the state must represent the position of all pieces, and importantly, whose turn it is (white or black).
- Here, we are assuming that white is the agent and black is the opponent. White moves first and is trying to maximize the utility, whereas black is trying to minimize the utility.
- In most games that we'll consider, the utility is degenerate in that it will be $+\infty$, $--$, or 0.

Characteristics of games

- All the utility is at the end state

- Different players in control at different states

- There are two important characteristics of games which make them hard.
- The first is that the utility is only at the end state. In typical search problems and MDPs that we might encounter, there are costs and rewards associated with each edge. These intermediate quantities make the problem easier to solve. In games, even if there are cues that indicate how well one is doing (number of pieces, score), technically all that matters is what happens at the end. In chess, it doesn't matter how many pieces you capture, your goal is just to checkmate the opponent's king.
- The second is the recognition that there are other people in the world! In search problems, you (the agent) controlled all actions. In MDPs, we already hinted at the loss of control where nature controlled the chance nodes, but we assumed we knew what distribution nature was using to transition. Now, we have another player that controls certain nodes, who is probably out to get us.

Policies

Deterministic policies: $\pi_p(s) \in \text{Actions}(s)$

action that player p takes in state s

Random policies $\pi_p(s, a) \in [0, 1]$:

probability of player p taking action a in state s

Note: π_p only defined for states s such that $\text{Player}(s) = p$

- Following our presentation of MDPs, we revisit the notion of a policy. We will use π to represent the collection of policies, one for each player. Specifically, $\pi = (\pi_p)_{p \in \text{Player}}$, where π_p is the policy for player p. Usually, we will have $\pi = (\pi_{\text{agent}}, \pi_{\text{opp}})$.
- It will be convenient to allow policies to be random. In this case, we will use $\pi_p(s, a)$ to denote the probability of player p choosing action a in state s.
- We can think of an MDP as a game between the agent and nature. The states of the game are all MDP states s and all chance nodes (s, a). It's the agent's turn on the MDP states s, and the agent acts according to π_{agent}. It's nature's turn on the chance nodes. Here, the actions are successor states s', and nature chooses s' with probability given by the transition probabilities of the MDP: $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$.

Game evaluation

Definition: game value function

The value $V_\pi(s)$ is the expected utility if agent follows π_{agent} and opponent follows π_{opp}.

Define $\mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}] = V_\pi(s_{\text{start}})$.

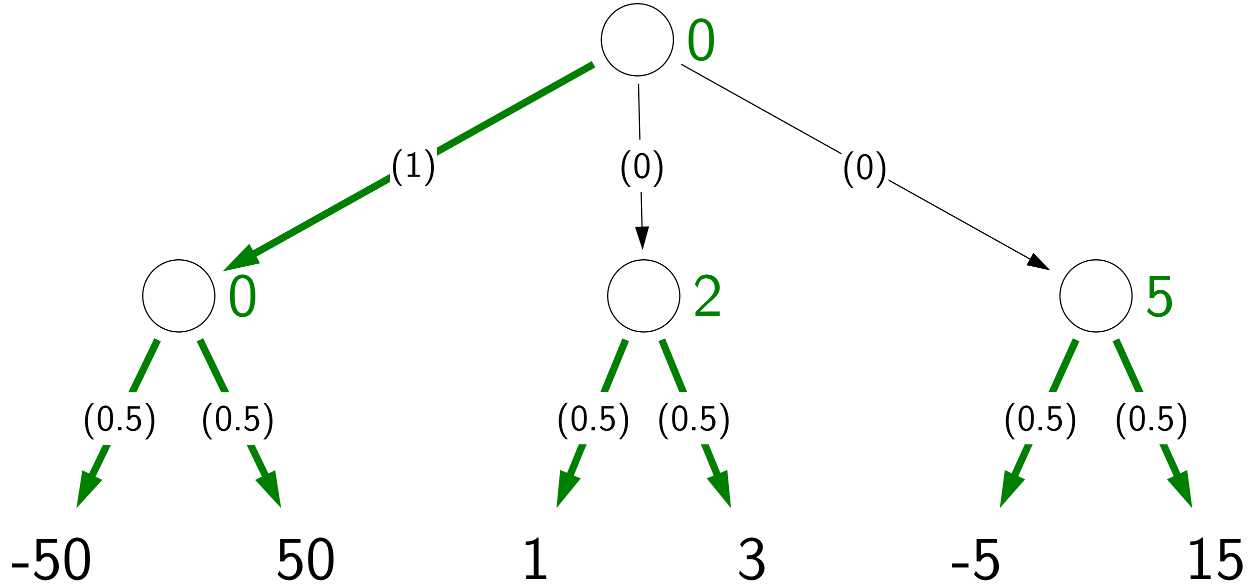
- The first step is to formalize the value of the game V_π, which is the expected utility that the agent obtains from state s (remember, since this is a zero-sum game, the opponent's utility is the negative of the agent's utility). Recall that $\pi = (\pi_{\text{agent}}, \pi_{\text{opp}})$ includes the policies of both agent and opponent. We will soon define a recurrence for this quantity, just like we did for MDPs.
- In addition, we use the $\mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}]$ notation, which defaults to the starting state and makes the policies explicit. This quantity will be useful for talking about what happens in the game when players vary their strategies.

Game evaluation example

Example: game evaluation

$\pi_{\text{agent}}(s) = \text{A}$

$\pi_{\text{opp}}(s, a) = \frac{1}{2}$ for $a \in \text{Actions}(s)$

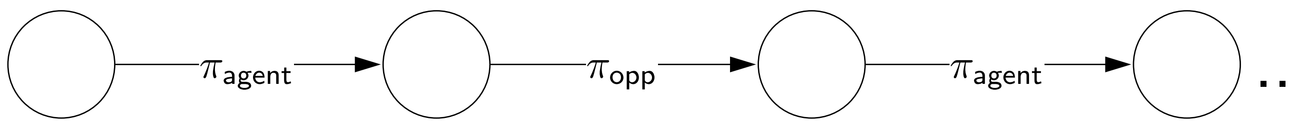


$$\boxed{V_\pi(s_{\text{start}}) = 0}$$

- We can visualize the computation of the values on the game tree. The value of a node is the values of the children combined according to the policy probabilities on the edges.

Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs



$$V_\pi(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s,a) V_\pi(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_\pi(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- More formally, we can write down a recurrence for $V_\pi(s)$, which includes three cases depending on whose turn it is. If the game is over ($\text{IsEnd}(s)$), then the value is just the utility $\text{Utility}(s)$. If it's the agent's turn, then we compute the expectation over the value of the successor resulting from the agent choosing an action according to $\pi(s, a)$. The opponent's case is exactly analogous.

The halving game

Problem: halving game

Start with a number N.

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The person that first reaches 0 wins.

[live solution]

Expectimax

Definition: expectimax value function

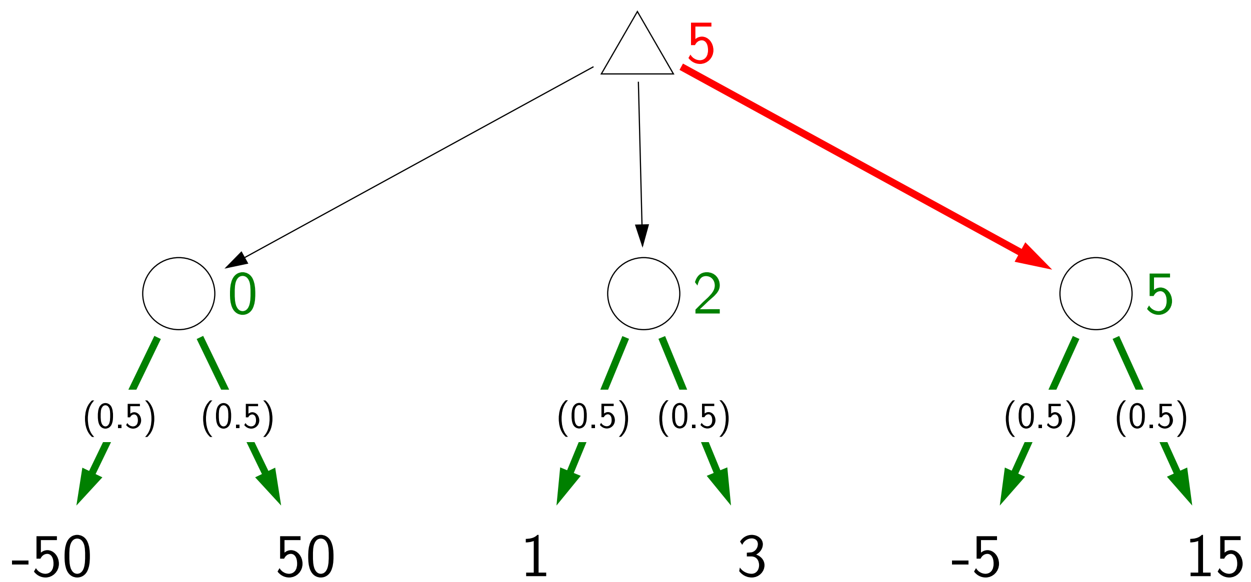
The value $V_{\text{opt},\pi}(s)$ is the expected utility if the agent follows the best policy assuming the opponent follows π_{opp}.

- Game evaluation just gave us the value of the game with two fixed policies π_{agent} and π_{opp}. But we are not handed a policy π_{agent}; we are trying to find the best policy. Expectimax gives us exactly that.
- We define the expectimax value $V_{\text{opt},\pi}$ with respect to a fixed opponent policy π_{opp} to be the expected utility if the agent follows the best policy.

Expectimax example

Example: expectimax

$\pi_{\text{opp}}(s, a) = \frac{1}{2}$ for $a \in \text{Actions}(s)$

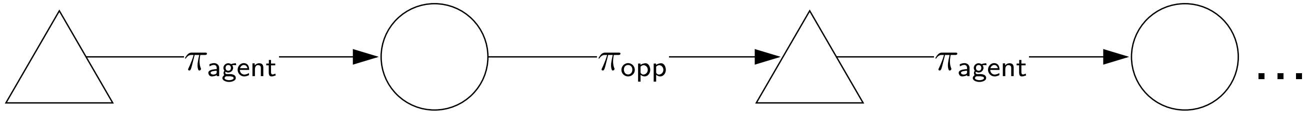


$\boxed{V_{\text{opt},\pi}(s_{\text{start}}) = 5}$

- We will use an upward-pointing triangle to denote states where the player is maximizing over actions (we call them **max nodes**).
- At max nodes, instead of averaging with respect to a policy, we simply take the max of the values of the children.

Expectimax recurrence

Analogy: recurrence for value iteration in MDPs



$$V_{\text{opt},\pi}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{opt},\pi}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{opt},\pi}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The recurrence for the expectimax value $V_{\text{opt},\pi}$ is exactly the same as the one for the game value V_π, except that we maximize over the agent's actions rather than following a fixed agent policy (which we don't know now).
- Where game evaluation was the analog of policy evaluation for MDPs, expectimax is the analog of value iteration.

Problem: don't know opponent's policy

Approach: assume the worst case

Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

Minimax

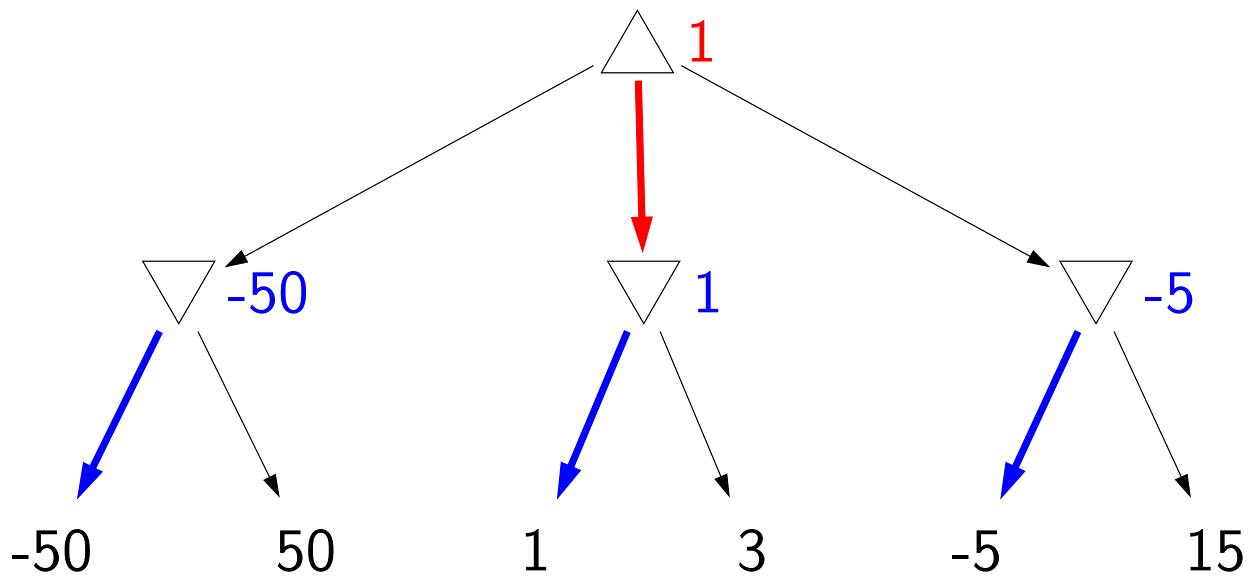
Definition: value function

The value $V_{\text{opt}}(s)$ is the utility if the agent follows the best policy and the opponent follows the **adversarial** policy.

- If we could perform some mind-reading and discover the opponent's policy, then we could maximally exploit it. However, in practice, we don't know the opponent's policy. So our solution is to assume the **worst case**, that is the opponent is doing all he can to minimize the agent's utility. We say that the opponent is **adversarial** in this case.

Minimax example

Example: minimax

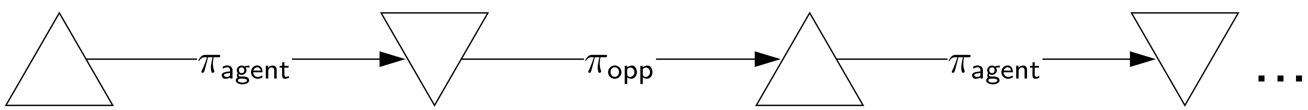


$$\boxed{V_{\text{opt}}(s_{\text{start}}) = 1}$$

- We use an upside-down triangle to represent **min nodes**, in which the player minimizes the value over possible actions.
- Note that the policy for the agent changes from choosing the rightmost action (expectimax) to the middle action. Why is this?

Minimax recurrence

No analogy in MDPs:



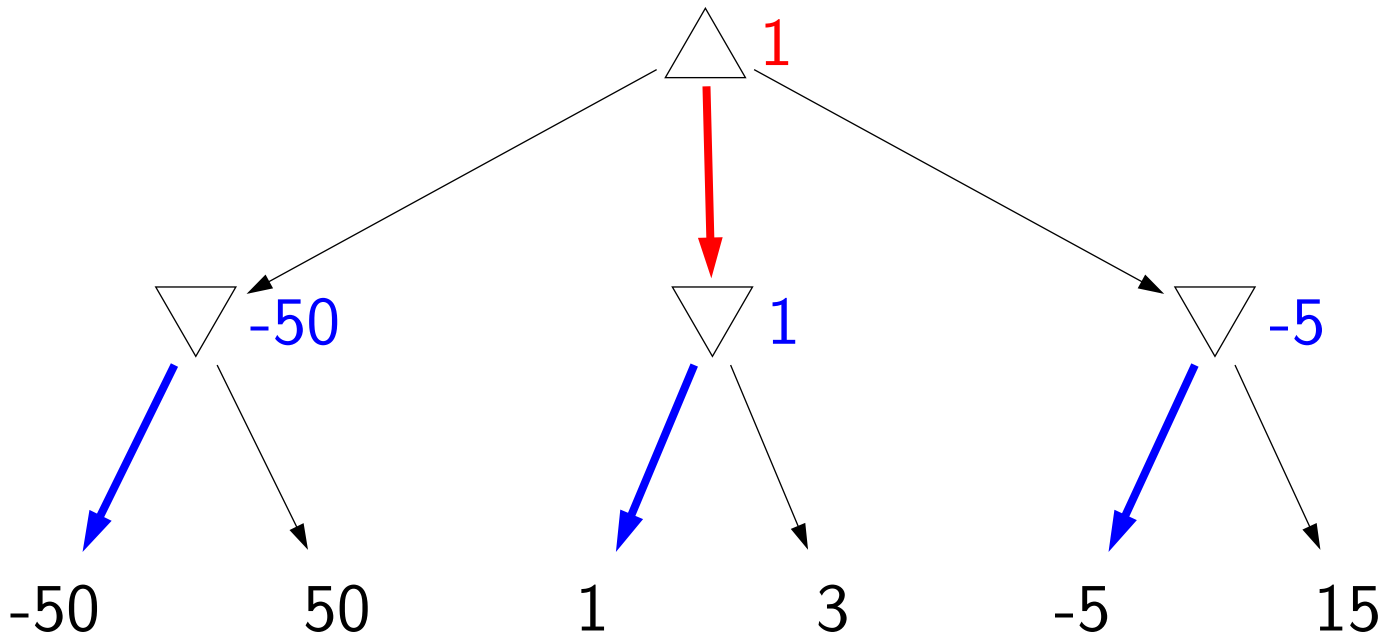
$$V_{\text{opt}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{opt}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{opt}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The recurrence for the minimax value is the same as expectimax, except that the expectation over the opponent's policy is replaced with a minimum over the opponent's possible actions. Note that the minimax value does not depend on any policies at all: it's just the agent and opponent playing optimally with respect to each other.

Extracting minimax policies

$$\pi^*_{\textsf{agent}}(s) = \arg \max_{a \in \textsf{Actions}(s)} V_{\textsf{opt}}(\textsf{Succ}(s,a))$$

$$\pi^*_{\textsf{opp}}(s) = \arg \min_{a \in \textsf{Actions}(s)} V_{\textsf{opt}}(\textsf{Succ}(s,a))$$

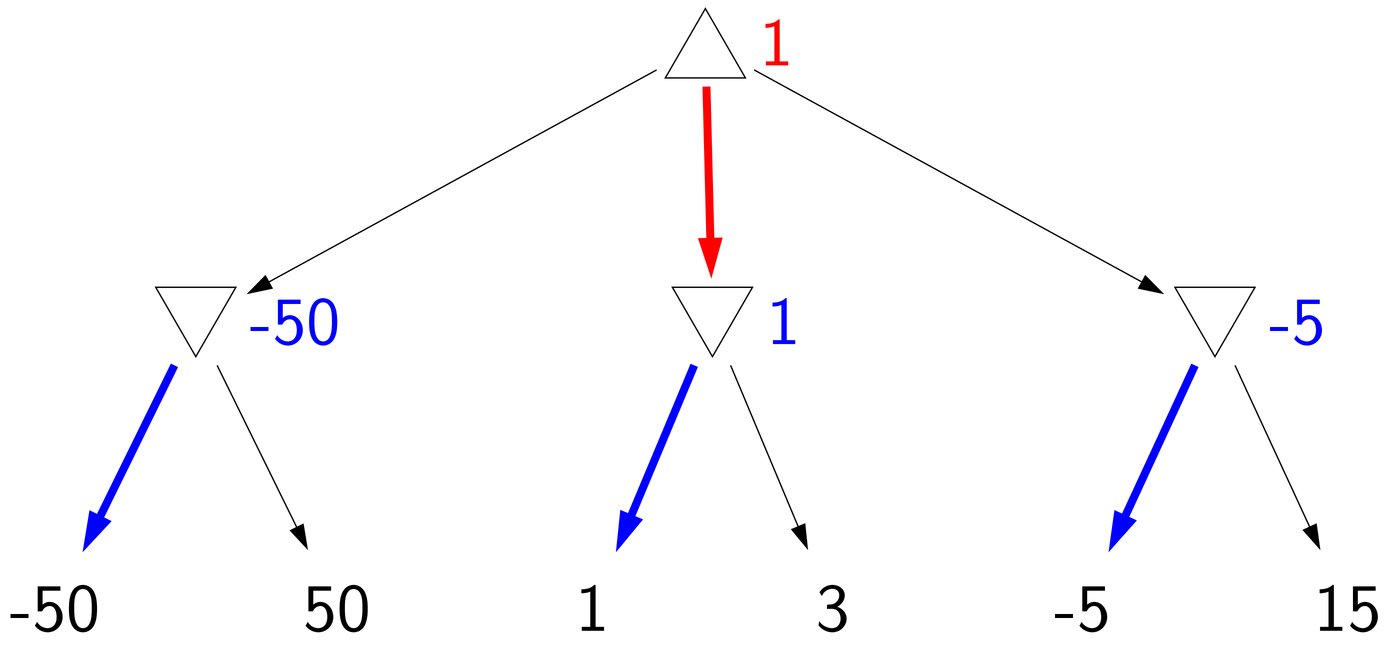


- Having computed the minimax value V_{opt}, we can extract the minimax policies π^*_{agent} and π^*_{opp} by just taking the action that leads to the state with the maximum (or minimum) value. In general, having a value function tells you which states are good, from which it's easy to set the policy to move to those states (provided you know the transition structure, which we assume we know here).

Minimax property 1

Proposition: best against adversarial opponent

$\mathbf{V}[\pi^*_{\text{agent}}, \pi^*_{\text{opp}}] \geq \mathbf{V}[\pi_{\text{agent}}, \pi^*_{\text{opp}}]$ for all π_{agent}

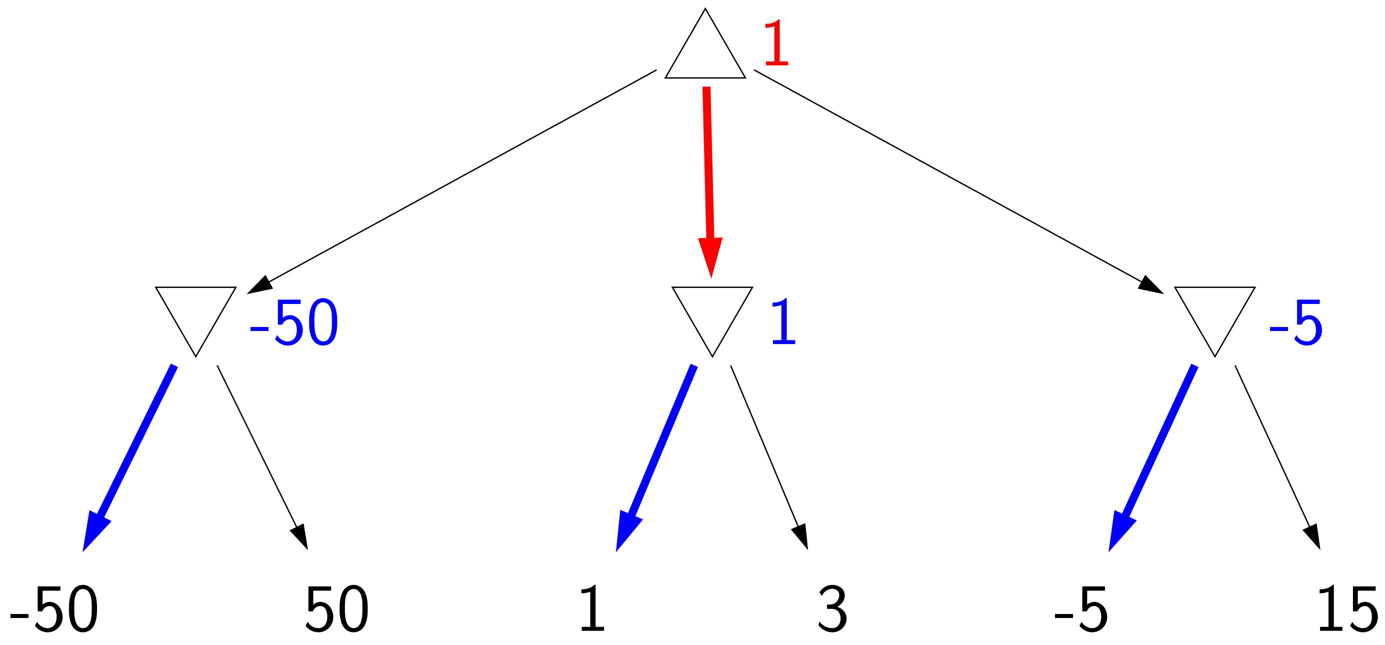


- Recall that π^*_{agent} and π^*_{opp} are the minimax policies. The first property is if the agent were to change her policy to any π_{agent}, then she would be no better off (and in general, worse off).
- From the example, it's intuitive that this property should hold. To prove it, we can perform induction starting from the leaves of the game tree, and show that the minimax value of each node is the highest over all possible policies.

Minimax property 2

Proposition: lower bound against any opponent

$\mathbf{V}[\pi^*_{\text{agent}}, \pi^*_{\text{opp}}] \le \mathbf{V}[\pi^*_{\text{agent}}, \pi_{\text{opp}}]$ for all π_{opp}



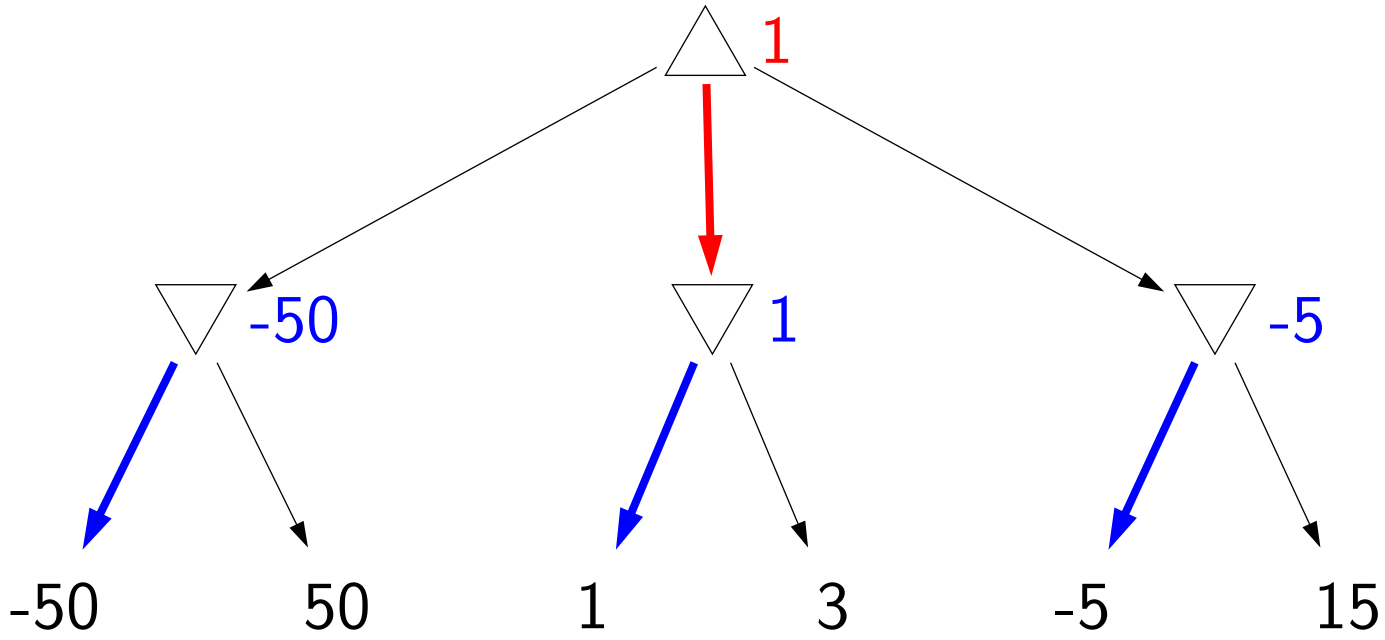
- The second property is the analogous statement for the opponent: if the opponent changes his policy from π^*_{opp} to π_{opp}, then he will be no better off (the value of the game can only increase).
- From the point of the view of the agent, this can be interpreted as guarding against the worst case. In other words, if we get a minimax value of 1, that means no matter what the opponent does, the agent is guaranteed at least a value of 1. As a simple example, if the minimax value is $+\infty$, then the agent is guaranteed to win, provided it follows the minimax policy.

Minimax non-property 3

Proposition: not necessarily best

Suppose opponent policy is π_{opp}.

$\mathbf{V}[\pi^*_{\text{agent}}, \pi_{\text{opp}}] \not\geq \mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}]$ for all π_{agent}



- However, following the minimax policy might not be optimal for the agent if the opponent is not playing the adversarial (minimax) policy.
- In this simple example, suppose the agent is playing π^*_{agent}, but the opponent is playing a random policy π_{opp}. Then the game value here would be 2 (larger than 1, as guaranteed by the second minimax property). However, if we followed the policy π_{agent} corresponding to expectimax, then we would have gotten a value of 5, which is even higher.
- To summarize, let π_{agent} be the expectimax policy against the random opponent π_{opp}, and π^*_{agent} and π^*_{opp} be the minimax policies. Then we have the following values for the example game tree:
- Agent's minimax policy against opponent's minimax policy: $\mathbf{V}[\pi^*_{\text{agent}}, \pi^*_{\text{opp}}] = 1$.
- Agent's expectimax policy against opponent's minimax policy: $\mathbf{V}[\pi_{\text{agent}}, \pi^*_{\text{opp}}] = -5$.
- Agent's minimax policy against opponent's random policy: $\mathbf{V}[\pi^*_{\text{agent}}, \pi_{\text{opp}}] = 2$.
- Agent's expectimax policy against opponent's random policy: $\mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}] = 5$.
- The four game values are related as follows: $\mathbf{V}[\pi_{\text{agent}}, \pi^*_{\text{opp}}] \le \mathbf{V}[\pi^*_{\text{agent}}, \pi^*_{\text{opp}}] \le \mathbf{V}[\pi^*_{\text{agent}}, \pi_{\text{opp}}] \le \mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}]$. Make sure you understand this.

A modified game

Example: game 2

You choose one of the three bins.

Flip a coin; if heads, then move one bin to the left (with wrap around).

I choose a number from that bin.

Your goal is to maximize the chosen number.

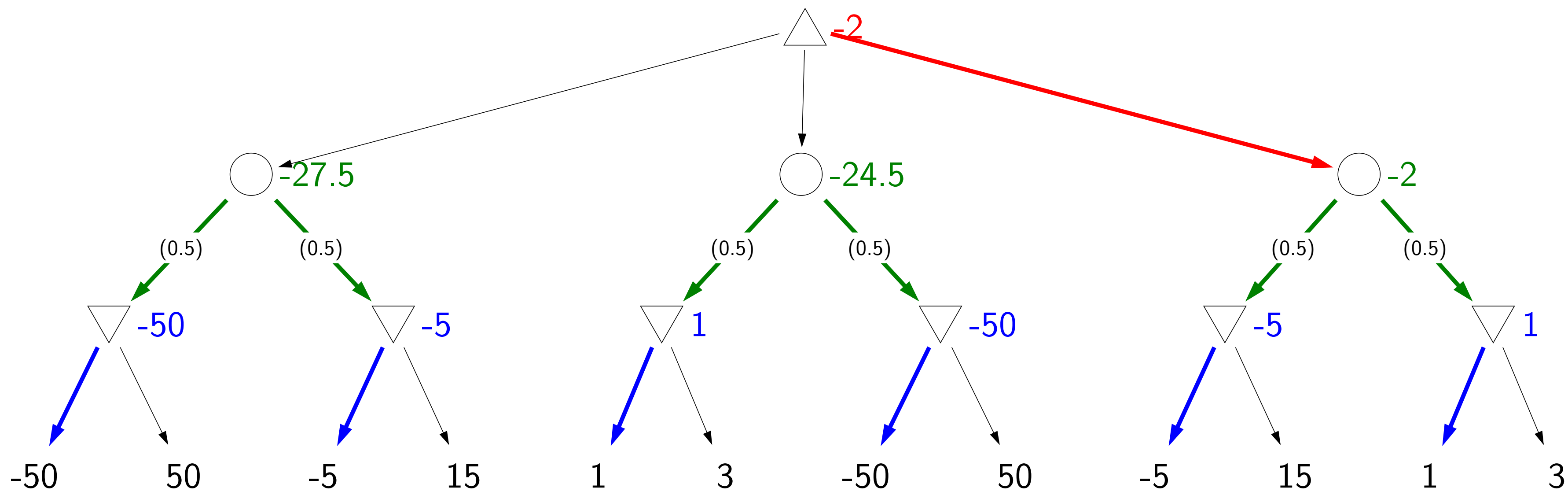
A		B		C	
-50	50	1	3	-5	15

- Now let us consider games that have an element of chance that does not come from the agent or the opponent. Or in the simple modified game, the agent picks, a coin is flipped, and then the opponent picks.
- It turns out that handling games of chance is just a straightforward extension of the game framework that we have already.

Expectiminimax example

Example: expectiminimax

$\pi_{\text{coin}}(s, a) = \frac{1}{2}$ for $a \in \{0, 1\}$



$\boxed{V_{\text{opt}}(s_{\text{start}}) = -2}$

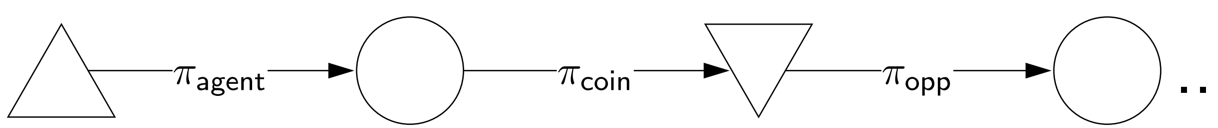
- In the example, notice that the minimax optimal policy has shifted from the middle action to the rightmost action, which guards against the effects of the randomness (of getting A).

Expectiminimax recurrence

Players $= \{\text{agent}, \text{opp}, \text{coin}\}$

Definition: value function

The value $V_{\text{opt}}(s)$ is the utility if the agent follows the **best** policy, the opponent follows the **adversarial** policy, the coin follows a fixed policy.



$$V_{\text{opt}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{coin}}(s,a) V_{\text{opt}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{coin} \\ \max_{a \in \text{Actions}(s)} V_{\text{opt}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{opt}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The resulting game is modeled using **expectiminimax**, where introduce a third player (called coin), which always follows a known random policy. We are using the term *coin* as just a metaphor for any sort of natural randomness.
- To handle coin, we simply add a line into our recurrence that sums over actions when it's coin's turn.

Summary so far

Primitives: max nodes, chance nodes, min nodes

Composition: alternate nodes according to model of game

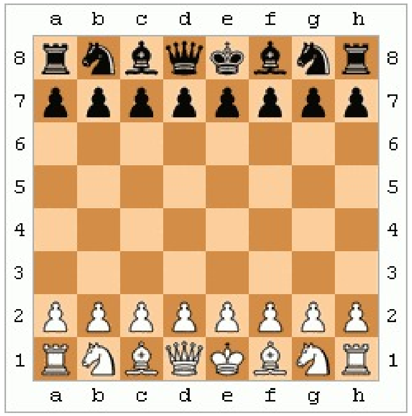
Value function $V(s)$: recurrence for expected utility

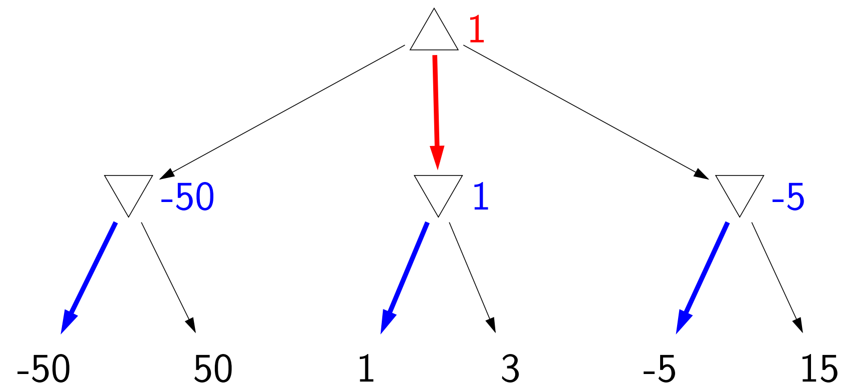
Scenarios to think about:

- What if you are playing against multiple opponents?
- What if you and your partner have to take turns (table tennis)?
- Some actions allow you to take an extra turn?

- In summary, so far, we've shown how to model a number of games using game trees, where each node of the game tree is either a max, chance, or min node depending on whose turn it is at that node and what we believe about that player's policy.
- Using these primitives, one can model more complex turn-taking games involving multiple players with heterogeneous strategies and where the turn-taking doesn't have to strictly alternate. The only restriction is that there are two parties: one that seeks to maximize utility and the other that seeks to minimize utility, along with other players who have known fixed policies (like coin).

Computation





Approach: tree search

Complexity:

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

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

25515520672986852924121150151425587630190414488161019324176778440771467258239937365843732987043555789782336195637736653285543297897675074636936187744140625

- Thus far, we've only touched on the modeling part of games. The rest of the lecture will be about how to actually compute (or approximately compute) the values of games.
- The first thing to note is that we cannot avoid exhaustive search of the game tree in general. Recall that a state is a summary of the past actions which is sufficient to act optimally in the future. In most games, the future depends on the exact position of all the pieces, so cannot forget much and exploit dynamic programming.
- Second, game trees can be enormous. Chess has a branching factor of around 35 and go has a branching factor of up to 361 (the number of moves to a player on his/her turn). Games also can last a long time, and therefore have a depth of up to 100 depth.
- A note about terminology specific to games: A game tree of depth d corresponds to a tree where each player has moved d times. Each level in the tree is called a **ply**. The number of plies is the depth times the number of players.

The halving game

Problem: halving game

Start with a number N.

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The person that first reaches 0 wins.

[live solution]

Speeding up minimax

- Evaluation functions: use domain-specific knowledge, compute approximate answer

- Alpha-beta pruning: general-purpose, compute exact answer

- The rest of the lecture will be about how to speed up the basic minimax search using two ideas: evaluation functions and alpha-beta pruning.

Roadmap

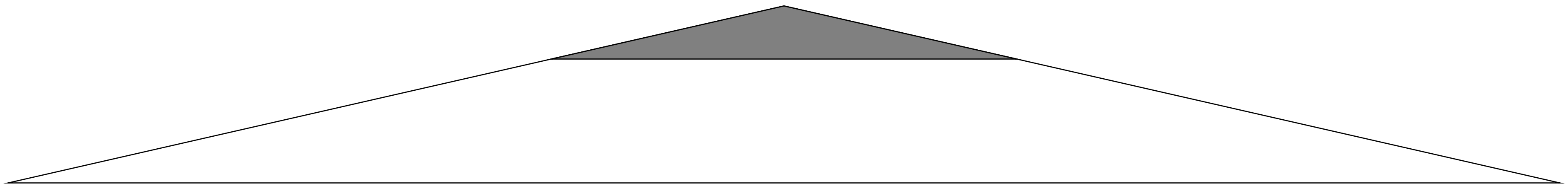
Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

Depth-limited search



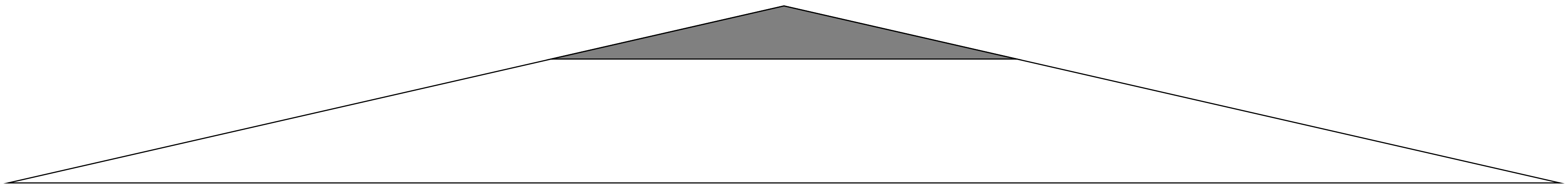
Limited depth tree search (stop at maximum depth d_{max}):

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

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

Convention: decrement depth at last player's turn

Evaluation functions



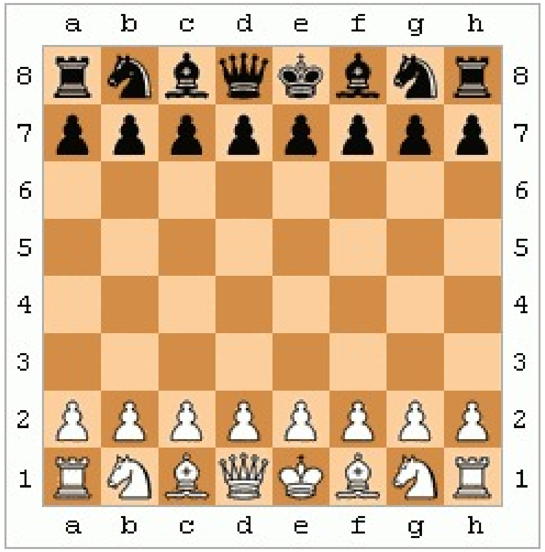
Definition: Evaluation function

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

Analogy: $\text{FutureCost}(s)$ in search problems

- The first idea on how to speed up minimax is to search only the tip of the game tree, that is down to depth d_{max}, which is much smaller than the total depth of the tree D (for example, d_{max} might be 4 and $D = 50$).
- We modify our minimax recurrence from before by adding an argument d, which the maximum depth that we are willing to descend from state s. If $d = 0$, then we don't do any more search, but fall back to an **evaluation function** $\text{Eval}(s)$, which is supposed to approximate the value of $V_{\text{opt}}(s)$ (just like the heuristic $h(s)$ approximated $\text{FutureCost}(s)$ in A* search).
- If $d > 0$, we recurse, decrementing the allowable depth by one at only min nodes, not the max nodes. This is because we are keeping track of the depth rather than the number of plies.

Evaluation functions



Example: chess

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

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

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

...

- Now what is this mysterious evaluation function $\text{Eval}(s)$ that serves as a substitute for the horrendously hard V_{opt} that we can't compute?
- Just as in A*, there is no free lunch, and we have to use domain knowledge about the game. Let's take chess for example. While we don't know who's going to win, there are some features of the game that are likely indicators. For example, having more pieces is good (material), being able to move them is good (mobility), keeping the king safe is good, and being able to control the center of the board is also good. We can then construct an evaluation function which is a weighted combination of the different properties.
- For example, $K - K'$ is the difference in the number of kings that the agent has over the number that the opponent has (losing kings is really bad since you lose then), $Q - Q'$ is the difference in queens, $R - R'$ is the difference in rooks, $B - B'$ is the difference in bishops, $N - N'$ is the difference in knights, and $P - P'$ is the difference in pawns.

Function approximation

Key idea: parameterized evaluation functions

$\text{Eval}(s; \mathbf{w})$ depends on weights $\mathbf{w} \in \mathbb{R}^d$

Feature vector: $\phi(s) \in \mathbb{R}^d$

$$\phi_1(s) = K - K'$$
$$\phi_2(s) = Q - Q'$$
$$\cdots$$

Linear evaluation function:

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

- Whenever you have written down a function that includes a weighted combination of different terms, there might be an opportunity for using machine learning to automatically tune these weights.
- In this case, we can take all the properties of the state, such as the difference in number of queens, as features $\phi(s)$. Note that in Q-learning with function approximation, we had a feature vector on each state-action pair. Here, we just need a feature vector on the state.
- We can then define the evaluation function as a dot product between a weight vector $\mathbf{w}$ and the feature vector $\phi(s)$.

Approximating the true value function

If knew optimal policies $\pi^*_{\text{agent}}, \pi^*_{\text{opp}}$, game tree evaluation provides best evaluation function:

$$\text{Eval}(s) = \mathbf{V}[\pi^*_{\text{agent}}, \pi^*_{\text{opp}}] = V_{\text{opt}}(s)$$

Intractable!

Two approximations:

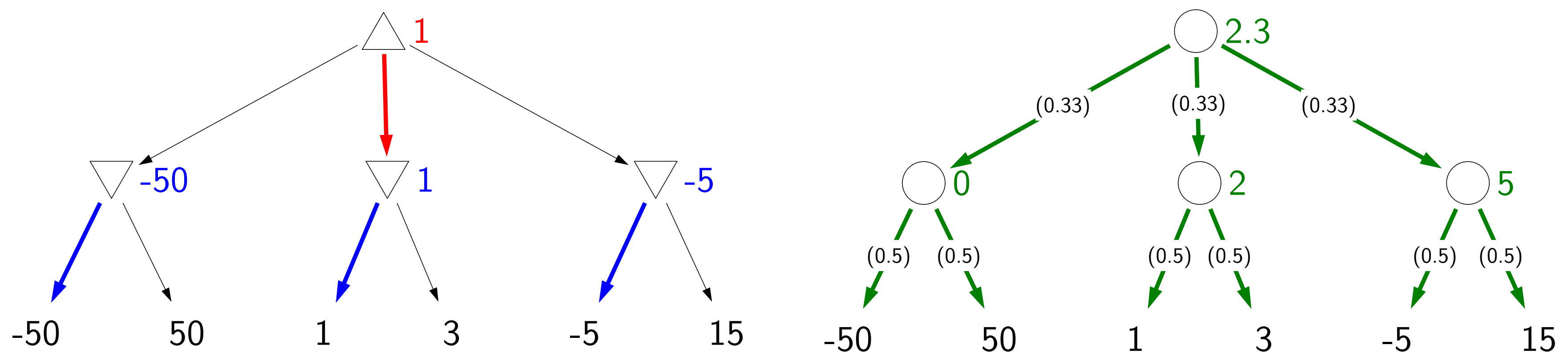
- Replace optimal policies with heuristic (random) policies
- Use Monte Carlo approximation

- Recall that the minimax value $V_{\text{opt}}(s)$ is the game value where the agent and the opponent both follow the minimax policies π^*_{agent} and π^*_{opp}. This is clearly intractable to compute. So we will approximate this value in two ways.

Approximation 1: random policies

Replace $\pi^*_{\text{agent}}, \pi^*_{\text{opp}}$ with random $\pi_{\text{agent}}, \pi_{\text{opp}}$:

Example: game 1



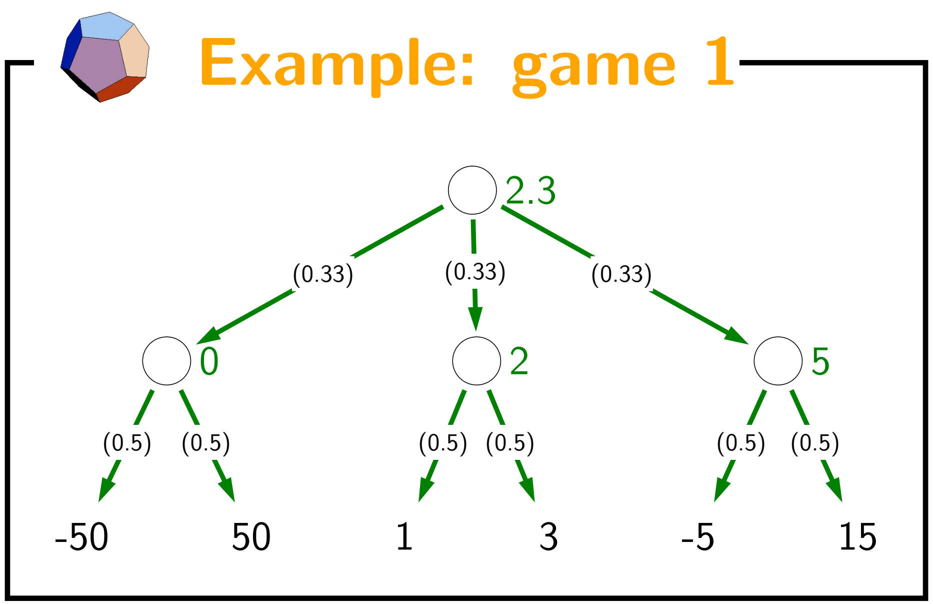
$\text{Eval}(s) = \mathbf{V}[\pi_{\text{agent}}, \pi_{\text{opp}}]$ is still hard to compute...

- First, we will simply replace the minimax policies with some random policies π_{agent} and π_{opp}. A naive thing would be to use policies that choose actions uniformly at random (as in the example), but in practice, we would want to choose better actions with higher probability. After all, these policies are supposed to be approximations of the minimax policies.
- In the example, the correct value is 1, but our approximation gives 2.3.
- Unfortunately, following even random policies is difficult to compute because we have to enumerate all nodes in the tree.

Approximation 2: Monte Carlo

Approach:

- Simulate n random paths by applying the policies
- Average the utilities of the n paths

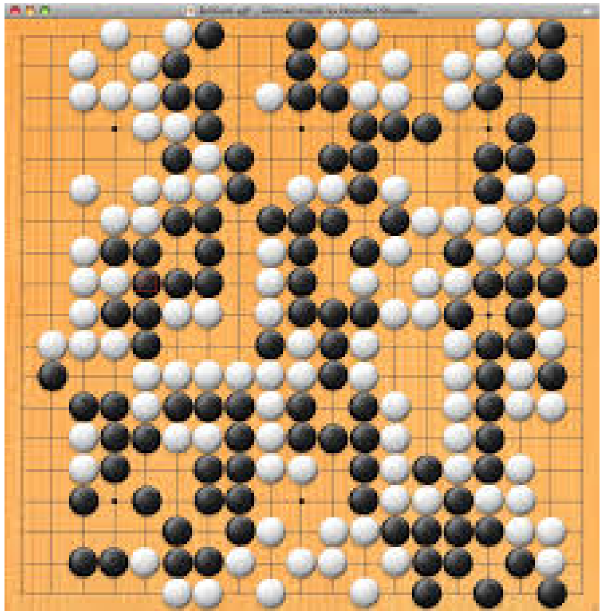


$$\text{Eval}(s) = \hat{\mathbf{V}}[\pi_{\text{agent}}, \pi_{\text{opp}}] =$$

$$\frac{1}{10}[(3) + (3) + (50) + (1) + (3) + (50) + (1) + (-5) + (50) + (3)] = 15.9$$

- However, moving to a fixed random policy sets the stage for the second approximation. Recall that Monte Carlo is a very powerful tool that allows us to approximate an expectation with samples.
- In this context, we will simply have the two policies play out the game n times, resulting in n paths (episodes). Each path has an associated utility. We then just average the n utilities together and call that our estimate $\hat{\mathbf{V}}[\pi_{\text{agent}}, \pi_{\text{opp}}]$ of the game value.
- From the example, you'll see that the values obtained by sampling are centered around the true value of 2.3, but have some variance, which will decrease as we get more samples (n increases).

Monte Carlo Go



Go has branching factor of 361, depth of 361

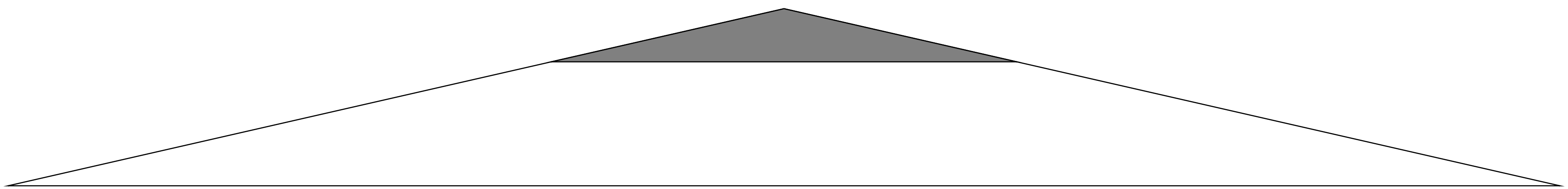
Example heuristic policy: if stone is threatened, try to save it; otherwise move randomly

Monte Carlo is responsible for recent successes

- Minimax search with hand-tuned evaluation functions was quite successful for producing chess-playing programs. However, these traditional methods worked horribly for Go, because the branching factor of Go was 361, much larger than Chess's 35.
- In recent years, researchers have made a ton of progress on Go, largely thanks to the use of Monte Carlo methods for creating evaluation functions. It should be quite surprising that the result obtained by moving under a simple random heuristic is actually helpful for determining the result obtained by playing carefully.

Summary: evaluation functions

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



Rely on evaluation function:

- Function approximation: parameterize by $\mathbf{w}$ and features
- Monte Carlo approximation: play many games heuristically (randomize)

- To summarize, this section has been about how to make naive exhaustive search over the game tree to compute the minimax value of a game faster.
- The methods so far have been focused on taking shortcuts: only searching up to depth d and relying on an **evaluation function**, a cheaper mechanism for estimating the value at a node rather than search its entire subtree.
- Function approximation allows us to use prior knowledge about the game in the form of features. Monte Carlo approximation allows us to look at thin slices of the subtree rather than looking at the entire tree.

Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

Pruning principle

Choose A or B with maximum value:

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

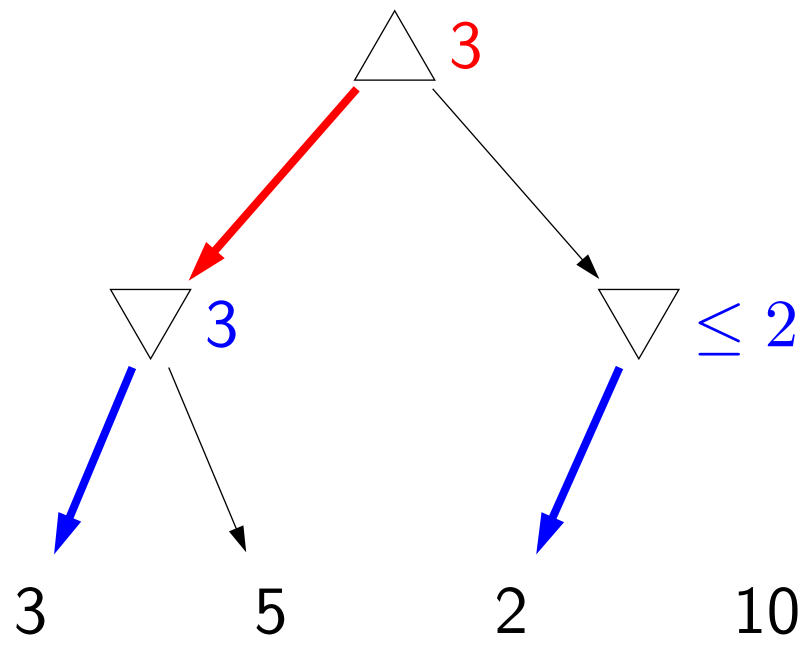
Key idea: branch and bound

Maintain lower and upper bounds on values.

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

- We continue on our quest to make minimax run faster based on **pruning**. Unlike evaluation functions, these are general purpose and have theoretical guarantees.
- The core idea of pruning is based on the branch and bound principle. As we are searching (branching), we keep lower and upper bounds on each value we're trying to compute. If we ever get into a situation where we are choosing between two options A and B whose intervals don't overlap or just meet at a single point (in other words, they do not **overlap non-trivially**), then we can choose the interval containing larger values (B in the example). The significance of this observation is that we don't have to do extra work to figure out the precise value of A.

Pruning game trees



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

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

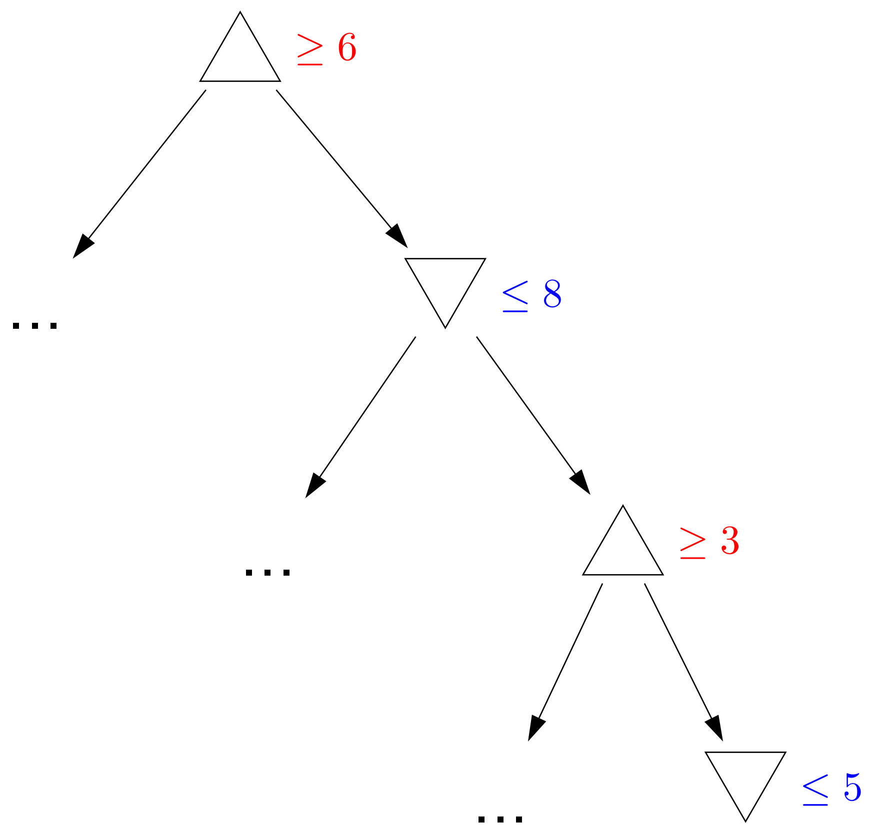
Since branch doesn't affect root value, can safely prune

- 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

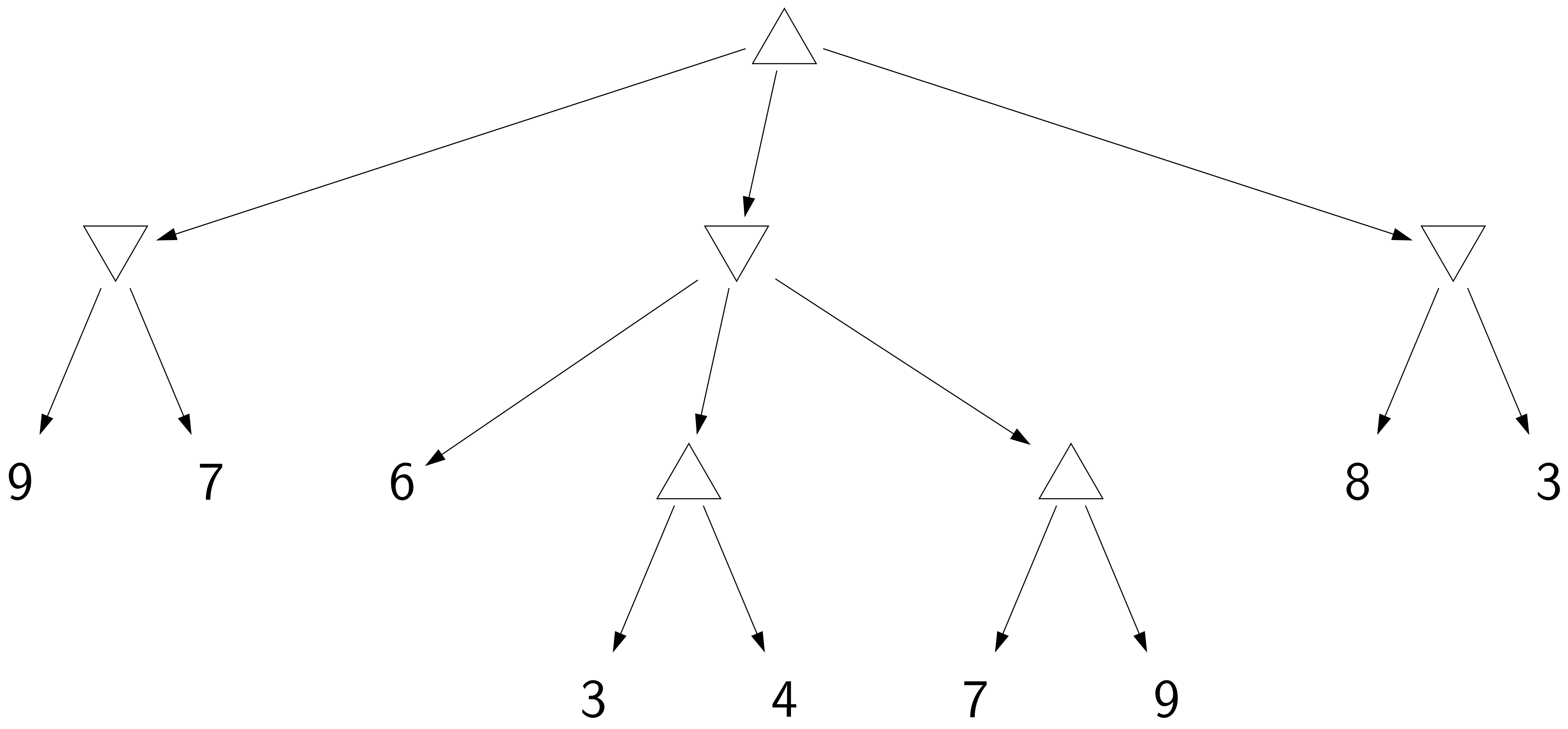
The optimal path is path that minimax policies take. Values of all nodes on path are the same.



- 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 prove 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.

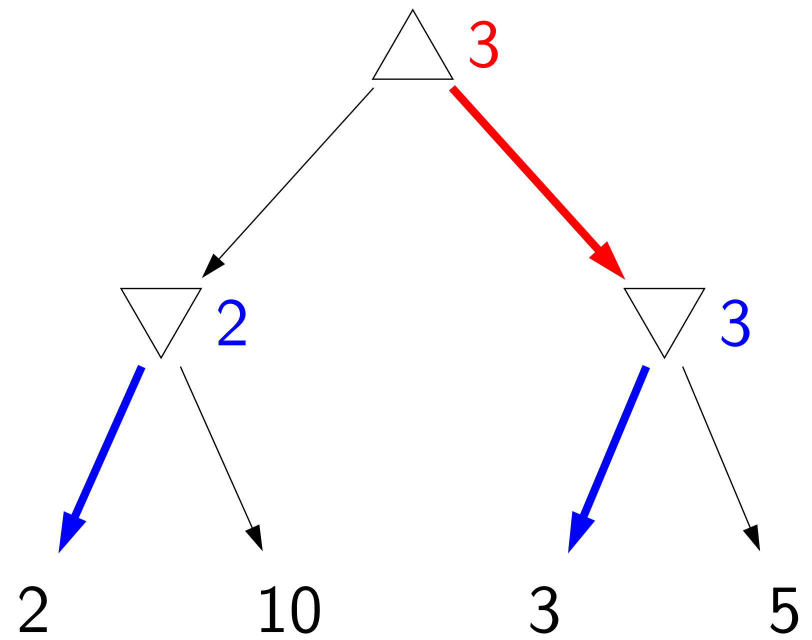
Alpha-beta pruning example



Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



- We have so far shown that alpha-beta pruning correctly computes the minimax value at the root, and seems to save some work by pruning subtrees. But how much of a savings do we get?
- The answer is that it depends on the order in which we explore the children. This simple example shows that with one ordering, we can prune the final leaf, but in the second, we can't.

Move ordering

Which ordering to choose?

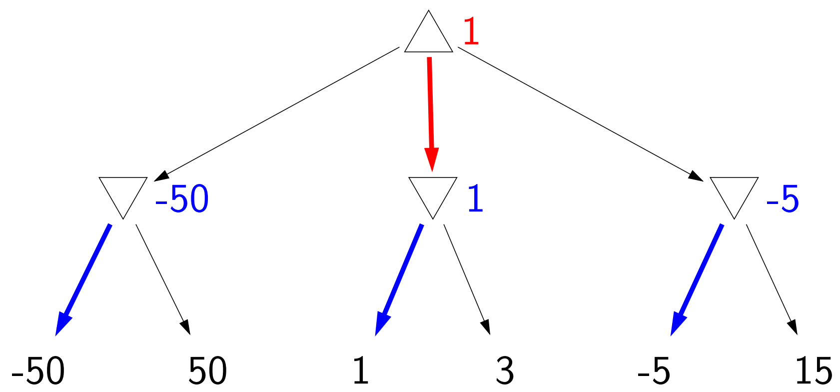
- Worst ordering: $O(b^{2\cdot d})$ time
- Best ordering: $O(b^{2\cdot 0.5d})$ time
- Random ordering: $O(b^{2\cdot 0.75d})$ time

In practice, can use evaluation function $\text{Eval}(s)$:

- Max nodes: order successors by decreasing $\text{Eval}(s)$
- Min nodes: order successors by increasing $\text{Eval}(s)$

- In the worst case, we don't get any savings.
- If we use the best possible ordering, then we save half the exponent, which is *significant*. This means that if could search to depth 10 before, we can now search to depth 20, which is truly remarkable given that the time increases exponentially with the depth.
- In practice, of course we don't know the best ordering. But interestingly, if we just use a random ordering, that allows us to search 33 percent deeper.
- We could also use a heuristic ordering based on a simple evaluation function. Intuitively, we want to search children that are going to give us the largest lower bound for max nodes and the smallest upper bound for min nodes.

Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: function approximation, Monte Carlo
- Alpha-beta pruning: increases depth