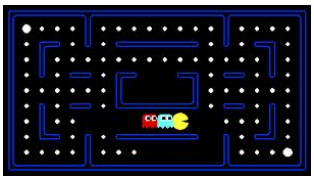
Lecture 10.1: Games I



Course plan

Search problems

Markov decision processes

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 agents, whose utility is not generally aligned with ours. Thus, the optimal strategy (policy) for us will depend on the strategies of these agents. Moreover, their strategies are often unknown and adversarial. How do we reason about this?

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 the other player (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 uniformly at random, then you should pick C so that on average things are reasonable (get 5 in expectation).

Roadmap

Games

Expectimax

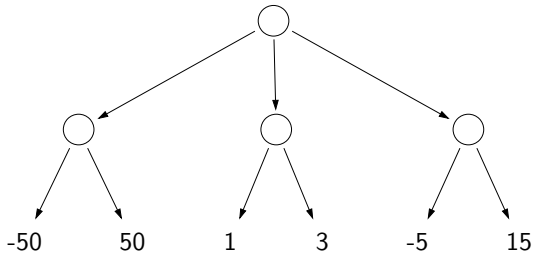
Minimax

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

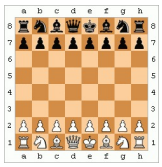
IsEnd(s): whether s is an end state (game over)

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-based 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, $-\infty$ 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 $+\infty$, $-\infty$, 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 states, who is probably out to get us.

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 player that is left with 0 wins.

[live solution: `HalvingGame`]

Policies

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

action that player p takes in state s

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

probability of player p taking action a in state s

[live solution: policies, main loop]

- Following our presentation of MDPs, we revisit the notion of a **policy**. Instead of having a single policy π, we have a policy π_p for each player $p \in \text{Players}$. We require that π_p only be defined when it's p's turn; that is, for states s such that $\text{Player}(s) = p$.
- It will be convenient to allow policies to be stochastic. 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: $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$.

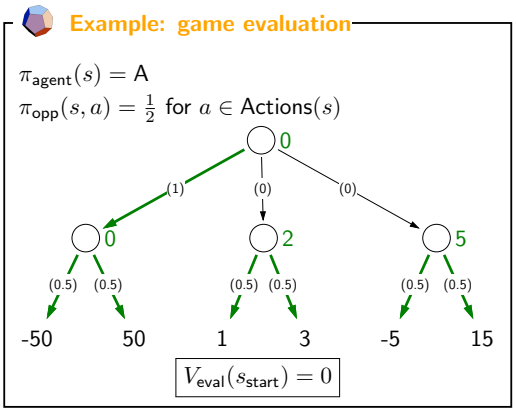
Roadmap

Games

Expectimax

Minimax

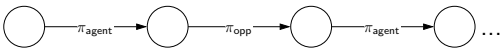
Game evaluation example



- Given two policies π_{agent} and π_{opp}, what is the (agent's) expected utility? That is, if the agent and the opponent were to play their (possibly stochastic) policies a large number of times, what would be the average utility? Remember, since we are working with zero-sum games, the opponent's utility is the negative of the agent's utility.
- Given the game tree, we can recursively compute the value (expected utility) of each node in the tree. The value of a node is the weighted average of the values of the children where the weights are given by the probabilities of taking various actions given by the policy at that node.

Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs

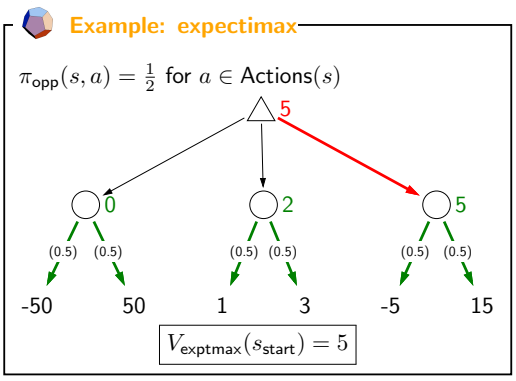


Value of the game:

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

- More generally, we can write down a recurrence for $V_{\text{eval}}(s)$, which is the **value** (expected utility) of the game at state s.
- There are three cases: 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 $\pi_{\text{agent}}(s,a)$. If it's the opponent's turn, we compute the expectation with respect to π_{opp} instead.

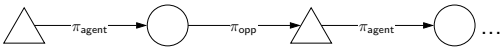
Expectimax example



- 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.
- In the game tree, we will now 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 take the max of the values of the children.
- This computation produces the **expectimax value** $V_{\text{exptmax}}(s)$ for a state s, which is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy π_{opp}.

Expectimax recurrence

Analogy: recurrence for value iteration in MDPs



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

- The recurrence for the expectimax value V_{exptmax} is exactly the same as the one for the game value V_{eval}, 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 analogue of policy evaluation for MDPs, expectimax is the analogue of value iteration.

Problem: don't know opponent's policy

Approach: assume the worst case

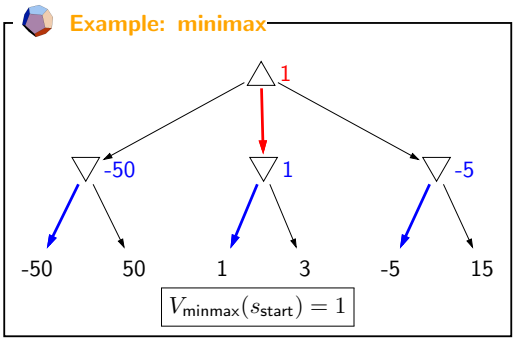
Roadmap

Games

Expectimax

Minimax

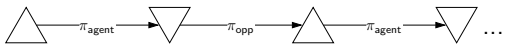
Minimax example



- 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 everything to minimize the agent's utility.
- In the game tree, 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:



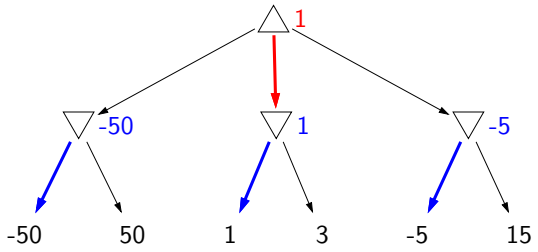
$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The general 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

$$\pi_{\max}(s) = \arg\max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a))$$

$$\pi_{\min}(s) = \arg\min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a))$$



- Having computed the minimax value V_{minmax}, we can extract the minimax policies $\pi_{\max}$ and $\pi_{\min}$ 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).

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 player that is left with 0 wins.

[live solution: `minimaxPolicy`]

Face off

Recurrences produce policies:

$$V_{\text{exptmax}} \Rightarrow \pi_{\text{exptmax(7)}}, \pi_7 \text{ (some opponent)}$$
$$V_{\text{minmax}} \Rightarrow \pi_{\max}, \pi_{\min}$$

Play policies against each other:

	$\pi_{\min}$	π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$	$V(\pi_{\max}, \pi_7)$
$\pi_{\text{exptmax(7)}}$	$V(\pi_{\text{exptmax(7)}}, \pi_{\min})$	$V(\pi_{\text{exptmax(7)}}, \pi_7)$

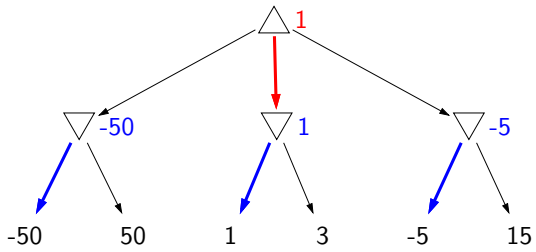
What's the relationship between these values?

- So far, we have seen how expectimax and minimax recurrences produce policies.
- The expectimax recurrence computes the best policy $\pi_{\text{exptmax(7)}}$ against a fixed opponent policy (say π_7 for concreteness).
- The minimax recurrence computes the best policy $\pi_{\max}$ against the best opponent policy $\pi_{\min}$.
- Now, whenever we take an agent policy π_{agent} and an opponent policy π_{opp}, we can play them against each other, which produces an expected utility via game evaluation, which we denote as $V(\pi_{\text{agent}}, \pi_{\text{opp}})$.
- How do the four game values of different combination of policies relate to each other?

Minimax property 1

Proposition: best against minimax opponent

$V(\pi_{\max}, \pi_{\min}) \geq V(\pi_{\text{agent}}, \pi_{\min})$ for all π_{agent}

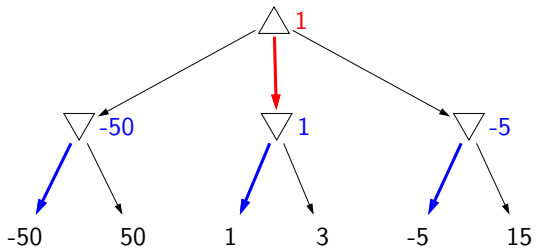


- Recall that $\pi_{\max}$ and $\pi_{\min}$ are the minimax agent and opponent policies, respectively. The first property is if the agent were to change her policy to any π_{agent}, then the agent would be no better off (and in general, worse off).
- From the example, it's intuitive that this property 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

$V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\text{opp}})$ for all π_{opp}

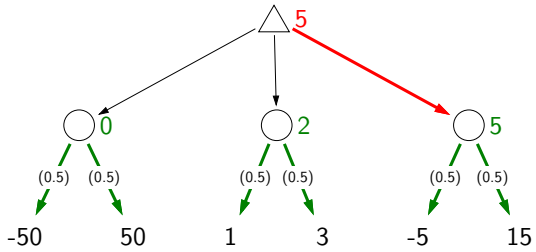


- The second property is the analogous statement for the opponent: if the opponent changes his policy from $\pi_{\min}$ to π_{opp}, then he will be no better off (the value of the game can only increase).
- From the point of the view of the agent, this can be interpreted as guarding against the worst 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 property 3

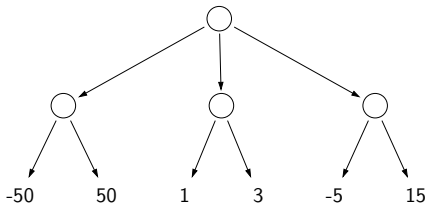
Proposition: not optimal if opponent is known

$V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax(7)}}, \pi_7)$ for opponent π_7



- However, following the minimax policy might not be optimal for the agent if the opponent is known to be not playing the adversarial (minimax) policy.
- Consider the running example where the agent chooses A, B, or C and the opponent chooses a bin. Suppose the agent is playing $\pi_{\max}$, but the opponent is playing a stochastic policy π_7 corresponding to choosing an action uniformly at random.
- Then the game value here would be 2 (which is larger than the minimax value 1, as guaranteed by property 2). However, if we followed the expectimax $\pi_{\text{exptmax(7)}}$, then we would have gotten a value of 5, which is even higher.

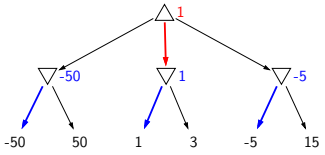
Relationship between game values



	$\pi_{\min}$		π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$	$\leq$	$V(\pi_{\max}, \pi_7)$
	1		2
	$\vee$		$\wedge$
$\pi_{\text{exptmax(7)}}$	$V(\pi_{\text{exptmax(7)}}, \pi_{\min})$		$V(\pi_{\text{exptmax(7)}}, \pi_7)$
	-5		5

- Putting the three properties together, we obtain a chain of inequalities that allows us to relate all four game values.
- We can also compute these values concretely for the running example.

Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Next time: scaling up to real games