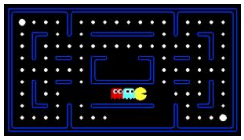
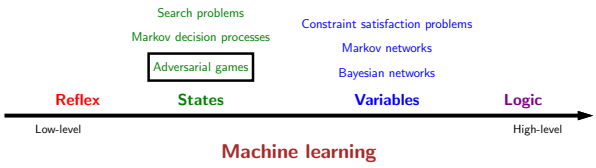
Games I



Course plan



- 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

Modeling

Modeling Games

Learning

Temporal Difference Learning

Algorithms

Game Evaluation

Expectimax

Minimax

Expectiminimax

Evaluation Functions

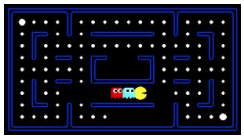
Alpha-Beta Pruning

Other Topics

Simultaneous Games

Non-Zero-Sum Games

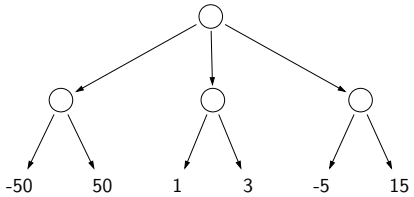
Games: modeling



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

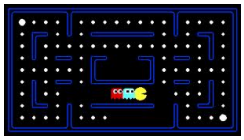
Stochastic policies $\pi_p(s, a) \in [0, 1]$:

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 $p \in$ Players. We require that π_p only be defined when it's p's turn; that is, for states s such that 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: $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$.

Games: game evaluation

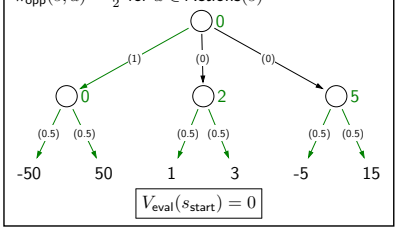


Game evaluation example

Example: game evaluation

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

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

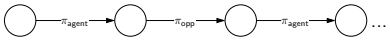


$V_{\text{eval}}(s_{\text{start}}) = 0$

- 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

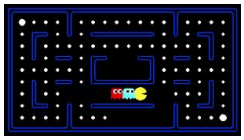


Value of the game:

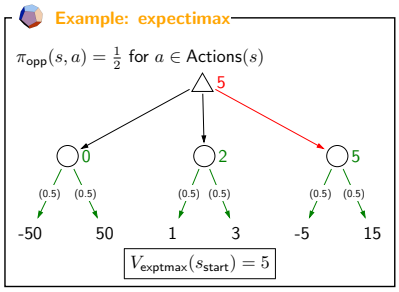
$$V_{\text{eval}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s)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

- 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.

Games: expectimax



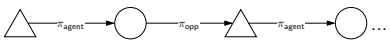
Expectimax example



- 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

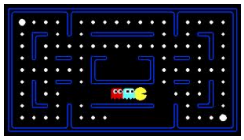
Analogy: recurrence for value iteration in MDPs



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

- 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.

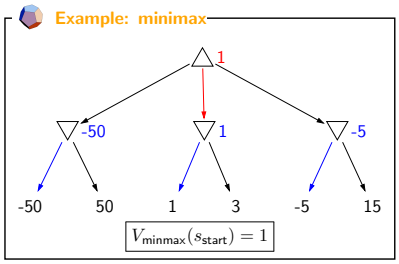
Games: minimax



Problem: don't know opponent's policy

Approach: assume the worst case

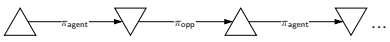
Minimax example



- 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:



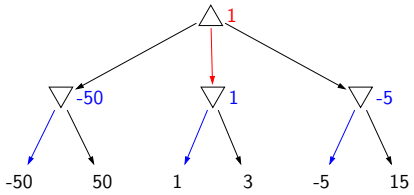
$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- 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:

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

$V_{\text{minmax}} \Rightarrow \pi_{\max}, \pi_{\min}$

Play policies against each other:

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

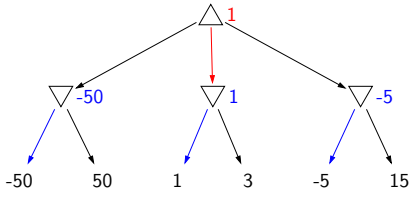
What's the relationship between these values?

- 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}

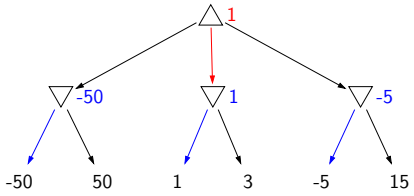


- Recall that $\pi_{\max}$ and $\pi_{\min}$ are the minimax agent and opponent policies, respectively. The first property is if the agent were to change her policy to any π_{agent}, then the agent would be no better off (and in general, worse off).
- From the example, it's intuitive that this property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

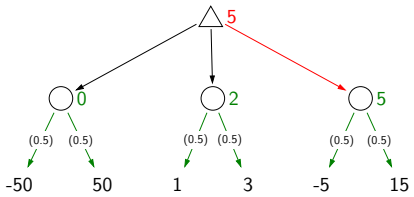


- The second property is the analogous statement for the opponent: if the opponent changes his policy from $\pi_{\min}$ to π_{opp}, then he will be no better off (the value of the game can only increase).
- From the point of the view of the agent, this can be interpreted as guarding against the worst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

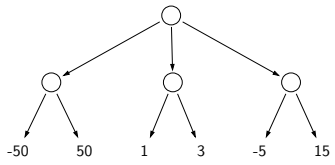
Proposition: not optimal if opponent is known

$V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax(7)}}, \pi_7)$ 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.

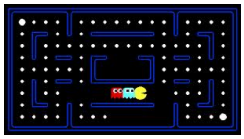
Relationship between game values



	$\pi_{\min}$		π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$	$\leq$	$V(\pi_{\max}, \pi_7)$
	1		2
	|\/		|\/
$\pi_{\text{exptmax(7)}}$	$V(\pi_{\text{exptmax(7)}}, \pi_{\min})$		$V(\pi_{\text{exptmax(7)}}, \pi_7)$
	-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.

Games: expectiminimax



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).

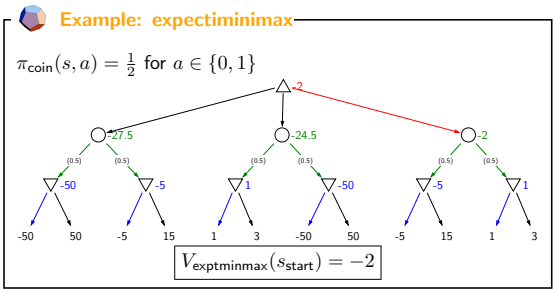
I choose a number from that bin.

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.

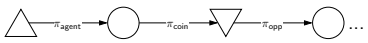
Expectiminimax example



- 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. The agent really wants to avoid ending up on A, in which case the opponent could deliver a deadly -50 utility.

Expectiminimax recurrence

Players = {agent, opp, coin}



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

- The resulting game is modeled using **expectiminimax**, where we introduce a third player (called coin), which always follows a known stochastic 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

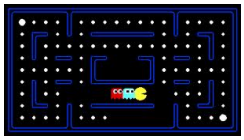
Value function $V_{...}(s)$: recurrence for expected utility

Scenarios to think about:

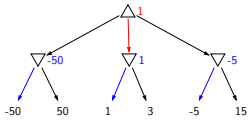
- What if you are playing against multiple opponents?
- What if you and your partner have to take turns (table tennis)?
- Some actions allow you to take an extra turn?

- 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).

Games: evaluation functions



Computation



Approach: tree search

Complexity:

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

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

25515526672986852924121150151425587630190414488161019324767784407714672582399373658437326870435557897823616956377366532855432976789767507463693618774414062 5

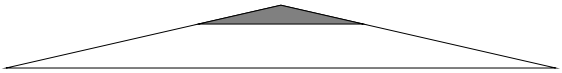
- Thus far, we've only touched on the modeling part of games. The rest of the lecture will be about how to actually compute (or approximately compute) the values of games.
- The first thing to note is that we cannot avoid exhaustive search of the game tree in general. Recall that a state is a summary 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 we 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.
- 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.

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.

Depth-limited search



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

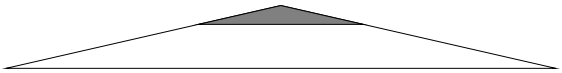
$$V_{\text{minmax}}(s,d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a), d-1) & \text{Player}(s)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

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

Convention: decrement depth at last player's turn

- 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 is 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 $V_{\text{minmax}}(s)$ (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



Definition: Evaluation function

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

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

- Now what is this mysterious evaluation function $\text{Eval}(s)$ that serves as a substitute for the horrendously hard V_{minmax} 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.

Evaluation functions

Example: chess

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

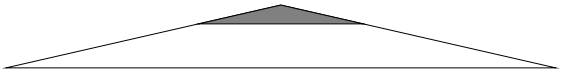
$\text{material} = 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P')$

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

...

Summary: evaluation functions

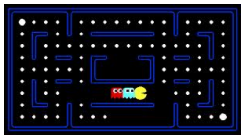
Depth-limited exhaustive search: $O(b^{2d})$ time



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

- 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**, and using a cheaper mechanism for estimating the value at a node rather than search its entire subtree.

Games: alpha-beta pruning



Pruning principle

Choose A or B with maximum value:

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

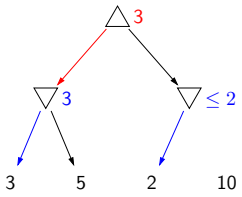
Key idea: branch and bound

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

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

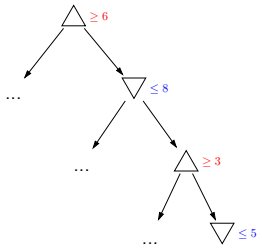
Since branch doesn't affect root value, can safely prune

- 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

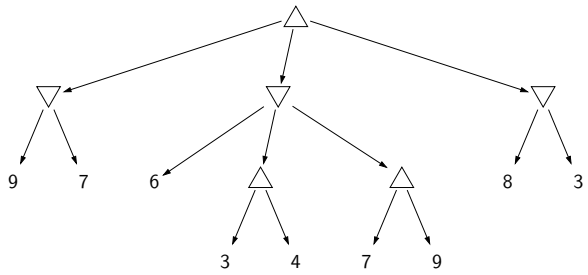
The optimal path is path that minimax policies take. Values of all nodes on path are the same.



- 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.

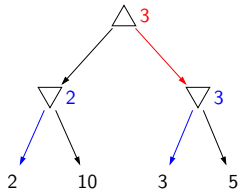
Alpha-beta pruning example



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?

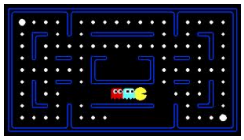
- Worst ordering: $O(b^{2 \cdot d})$ time
- Best ordering: $O((\sqrt{b - \frac{3}{4}} + \frac{1}{2})^{2 \cdot d}) \simeq O(b^{2 \cdot 0.5d})$ time
- Random ordering: $O(b^{2 \cdot 0.75d})$ time when $b = 2$
- Random ordering: $O((\frac{b-1+\sqrt{b^2+14b+1}}{4})^{2 \cdot d})$ for general b

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

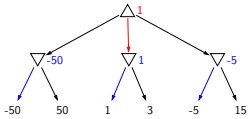
- Max nodes: order successors by decreasing Eval(s)
- Min nodes: order successors by increasing Eval(s)

- In the worst case, we don't get any savings.
- If we use the best possible ordering, then we roughly 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.

Games: recap



Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: domain-specific, approximate
- Alpha-beta pruning: domain-general, exact