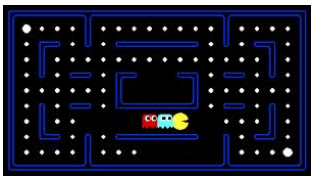
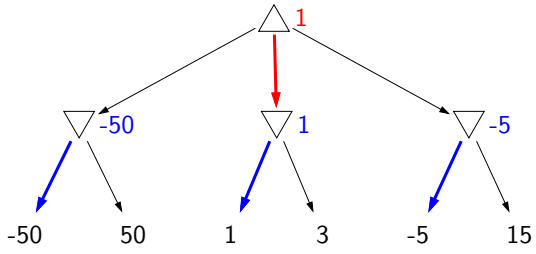
Lecture 11.1: Games II



Review: minimax

agent (max) versus opponent (min)



- Recall that the central object of study is the game tree. Game play starts at the root (starting state) and descends to a leaf (end state), where at each node s (state), the player whose turn it is (Player(s)) chooses an action $a \in$ Actions(s), which leads to one of the children Succ(s, a).
- The **minimax principle** provides one way for the agent (your computer program) to compute a pair of minimax policies for both the agent and the opponent $(\pi^*_{\text{agent}}, \pi^*_{\text{opp}})$.
- For each node s, we have the minimax value of the game $V_{\text{minmax}}(s)$, representing the expected utility if both the agent and the opponent play optimally. Each node where it's the agent's turn is a max node (right-side up triangle), and its value is the maximum over the children's values. Each node where it's the opponent's turn is a min node (upside-down triangle), and its value is the minimum over the children's values.
- Important properties of the minimax policies: The agent can only decrease the game value (do worse) by changing his/her strategy, and the opponent can only increase the game value (do worse) by changing his/her strategy.

Roadmap

Expectiminimax

Evaluation functions

TD learning

Alpha-beta pruning

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).

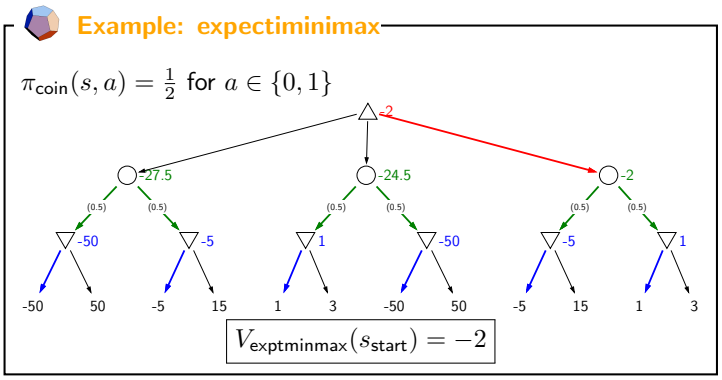
I choose a number from that bin.

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.

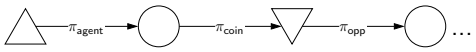
Expectiminimax example



- 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.

Value recurrences

Primitives: max nodes, chance nodes, min nodes

Composition: alternate nodes according to model of game

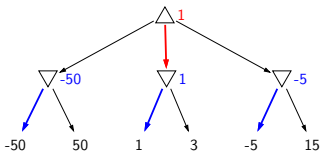
Value function $V_{...}(s)$: recurrence for expected utility

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?

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

25515520672986852924121150151425587630190414488161019324176778440771467258239937365843732987043555789782336195637736653285543297897675074636936187744140625

- Thus far, we've only touched on the modeling part of games. Now we will turn to the question of 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.

Roadmap

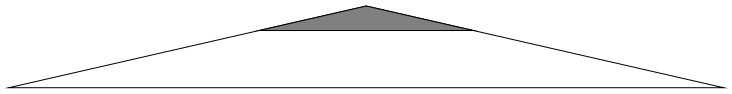
Expectiminimax

Evaluation functions

TD learning

Alpha-beta pruning

Depth-limited search



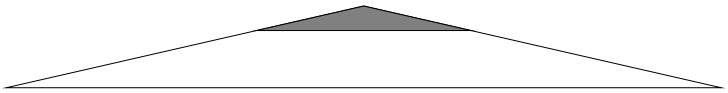
Limited depth tree search (stop at maximum depth d_{max}):

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

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

Convention: decrement depth at last player's turn

Evaluation functions



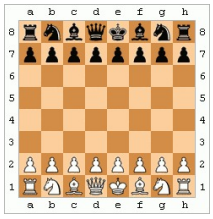
Definition: Evaluation function

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

Analogy: FutureCost(s) in search problems

- 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** Eval(s), which is supposed to approximate the value of $V_{\text{minmax}}(s)$ (just like the heuristic $h(s)$ approximated 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

Eval(s) = material + mobility + king-safety + center-control

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

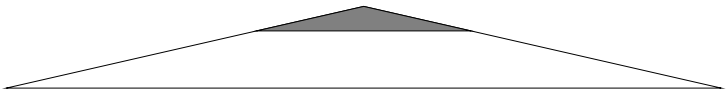
$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

- Now what is this mysterious evaluation function 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.

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.

Roadmap

Expectiminimax

Evaluation functions

TD learning

Alpha-beta pruning

Evaluation function

Old: hand-crafted

Example: chess

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

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

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

...

New: learn from data

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

- Having a good evaluation function is one of the most important components of game playing. So far we've shown how one can manually specify the evaluation function by hand. However, this can be quite tedious, and moreover, how does one figure out to weigh the different factors? Next, we will see a method for learning this evaluation function automatically from data.
- The three ingredients in any machine learning approach are to determine the (i) model family (in this case, what is $V(s; \mathbf{w})$?), (ii) where the data comes from, and (iii) the actual learning algorithm. We will go through each of these in turn.

Model for evaluation functions

Linear:

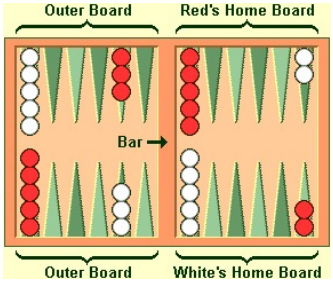
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

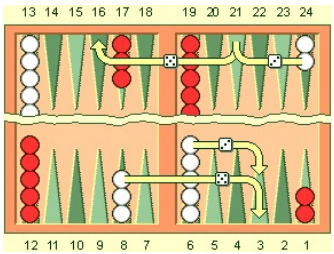
Neural network:

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

- When we looked at Q-learning, we considered linear evaluation functions (remember, linear in the weights $\mathbf{w}$). This is the simplest case, but it might not be suitable in some cases.
- But the evaluation function can really be any parametrized function. For example, the original TD-Gammon program used a neural network, which allows us to represent more expressive functions that capture the non-linear interactions between different features.
- Any model that you could use for regression in supervised learning you could also use here.

Example: Backgammon

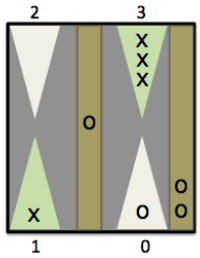




- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to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

Features for Backgammon

state s



Features $\phi(s)$:

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

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

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

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

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

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

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

Generating data

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

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

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

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

Generate episode:

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

- The second ingredient of doing learning is generating the data. As in reinforcement learning, we will generate a sequence of states, actions, and rewards by simulation — that is, by playing the game.
- In order to play the game, we need two exploration policies: one for the agent, one for the opponent. The policy of the dice is fixed to be uniform over $\{1, \ldots, 6\}$ as expected.
- A natural policy to use is one that uses our current estimate of the value $V(s; \mathbf{w})$. Specifically, the agent's policy will consider all possible actions from a state, use the value function to evaluate how good each of the successor states are, and then choose the action leading to the highest value. Generically, we would include $\text{Reward}(s, a, \text{Succ}(s, a))$, but in games, all the reward is at the end, so $r_t = 0$ for $t < n$ and $r_n = \text{Utility}(s_n)$. Symmetrically, the opponent's policy will choose the action that leads to the lowest possible value.
- Given this choice of π_{agent} and π_{opp}, we generate the actions $a_t = \pi_{\text{Player}(s_{t-1})}(s_{t-1})$, successors $s_t = \text{Succ}(s_{t-1}, a_t)$, and rewards $r_t = \text{Reward}(s_{t-1}, a_t, s_t)$.
- In reinforcement learning, we saw that using an exploration policy based on just the current value function is a bad idea, because we can get stuck exploiting local optima and not exploring. In the specific case of Backgammon, using deterministic exploration policies for the agent and opponent turns out to be fine, because the randomness from the dice naturally provides exploration.

Learning algorithm

Episode:

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

A small piece of experience:

$$(s, a, r, s')$$

Prediction:

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

Target:

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

- With a model family $V(s; \mathbf{w})$ and data $s_0, a_1, r_1, s_1, \ldots$ in hand, let's turn to the learning algorithm.
- A general principle in learning is to figure out the **prediction** and the **target**. The prediction is just the value of the current function at the current state s, and the target uses the data by looking at the immediate reward r plus the value of the function applied to to the successor state s' (discounted by γ). This is analogous to the SARSA update for Q-values, where our target actually depends on a one-step lookahead prediction.

General framework

Objective function:

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

Gradient:

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

Update:

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

- Having identified a prediction and target, the next step is to figure out how to update the weights. The general strategy is to set up an objective function that encourages the prediction and target to be close (by penalizing their squared distance).
- Then we just take the gradient with respect to the weights $\mathbf{w}$.
- Note that even though technically the target also depends on the weights $\mathbf{w}$, we treat this as constant for this derivation. The resulting learning algorithm by no means finds the global minimum of this objective function. We are simply using the objective function to motivate the update rule.

Temporal difference (TD) learning

Algorithm: TD learning

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

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

For linear functions:

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

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

- Plugging in the prediction and the target in our setting yields the TD learning algorithm. For linear functions, recall that the gradient is just the feature vector.

Comparison

Algorithm: TD learning

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

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

Algorithm: Q-learning

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

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

Comparison

Q-learning:

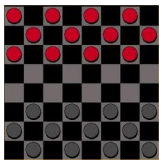
- Operate on $\hat{Q}_{\text{opt}}(s, a; \mathbf{w})$
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions $T(s, a, s')$

TD learning:

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

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

Learning to play checkers

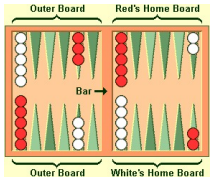


Arthur Samuel's checkers program [1959]:

- Learned by playing itself repeatedly (self-play)
- Smart features, linear evaluation function, use intermediate rewards
- Reach human amateur level of play
- IBM 701: 9K of memory!

- The idea of using machine learning for game playing goes as far back as Arthur Samuel's checkers program. Many of the ideas (using features, alpha-beta pruning) were employed, resulting in a program that reached a human amateur level of play. Not bad for 1959!

Learning to play Backgammon

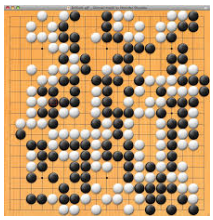


Gerald Tesauro's TD-Gammon [1992]:

- Learned weights by playing itself repeatedly (1 million times)
- Dumb features, neural network, no intermediate rewards
- Reached human expert level of play, provided new insights into opening

- Tesauro refined some of the ideas from Samuel with his famous TD-Gammon program provided the next advance, using a variant of TD learning called TD(λ). It had dumber features, but a more expressive evaluation function (neural network), and was able to reach an expert level of play.

Learning to play Go



AlphaGo Zero [2017]:

- Learned by self play (4.9 million games)
- Dumb features (stone positions), neural network, no intermediate rewards, Monte Carlo Tree Search
- Beat AlphaGo, which beat Le Sedol in 2016
- Provided new insights into the game

- Very recently, self-play reinforcement learning has been applied to the game of Go. AlphaGo Zero uses a single neural network to predict winning probability and actions to be taken, using raw board positions as inputs. Starting from random weights, the network is trained to gradually improve its predictions and match the results of an approximate (Monte Carlo) tree search algorithm.

Summary so far

- Parametrize evaluation functions using features
- TD learning: learn an evaluation function

$$(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Roadmap

Expectiminimax

Evaluation functions

TD learning

Alpha-beta pruning

Pruning principle

Choose A or B with maximum value:

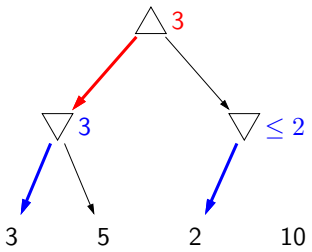
A: [3, **5**] B: [**5**, 100]

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

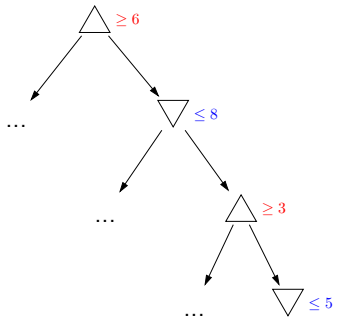
Since branch doesn't affect root value, can safely prune

- 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

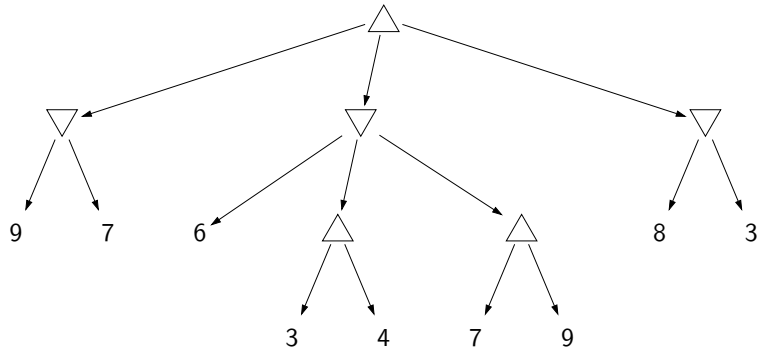
The optimal path is path that minimax policies take. Values of all nodes on path are the same.



- 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 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.

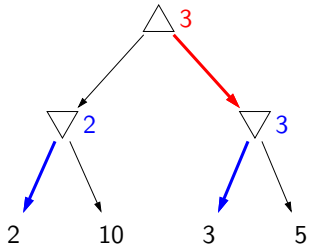
Alpha-beta pruning example



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 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?

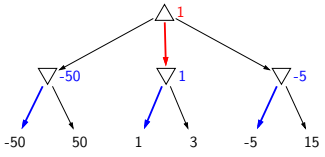
- Worst ordering: $O(b^{2 \cdot d})$ time
- Best ordering: $O(b^{2 \cdot 0.5d})$ time
- Random ordering: $O(b^{2 \cdot 0.75d})$ time

In practice, can use evaluation function 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 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



- Evaluation functions: domain-specific, approximate
- TD-learning: learn evaluation function through gameplay
- Alpha-beta pruning: domain-general, exact