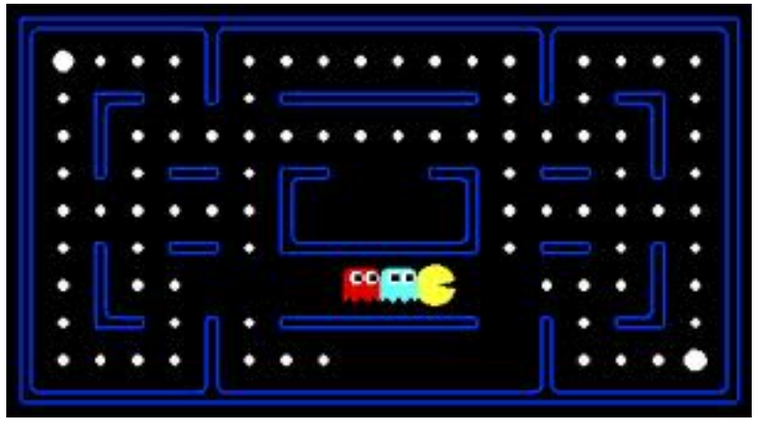
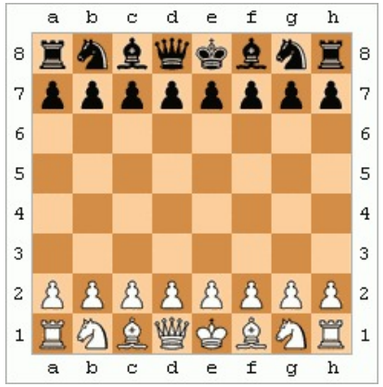
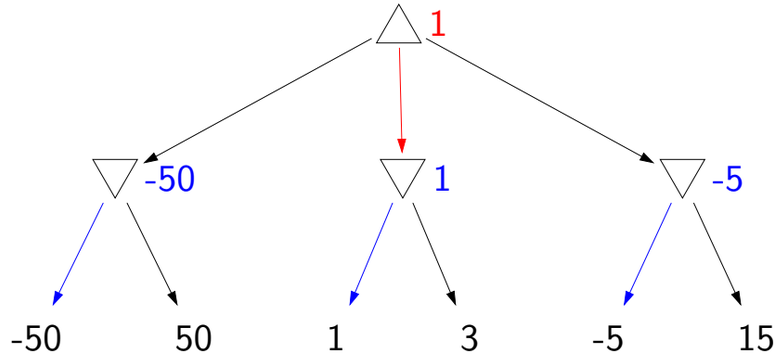
Games: evaluation functions



Computation





Approach: tree search

Complexity:

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

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

25515520672986852924121150151425587630190414488161019324176778440771467258239937365843732987043555789782336195637736653285543297897675074636936187744140625

- Thus far, we've only touched on the modeling part of games. The rest of the lecture will be about how to actually compute (or approximately compute) the values of games.
- The first thing to note is that we cannot avoid exhaustive search of the game tree in general. Recall that a state is a summary of the past actions which is sufficient to act optimally in the future. In most games, the future depends on the exact position of all the pieces, so 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.

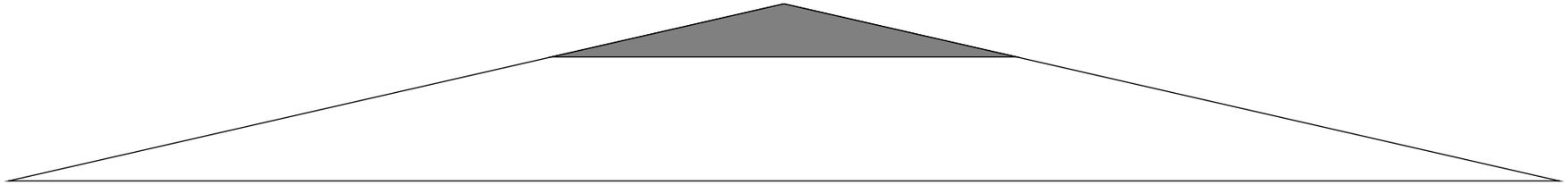
Speeding up minimax

- 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



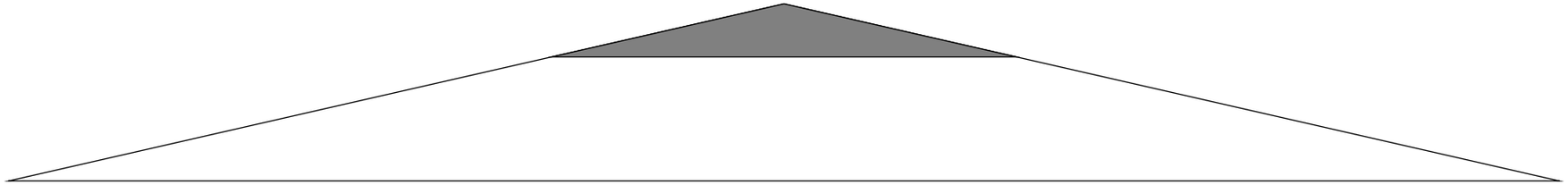
Limited depth tree search (stop at maximum depth d_{max}):

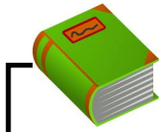
$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s)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

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

Convention: decrement depth at last player's turn

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

- 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

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

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

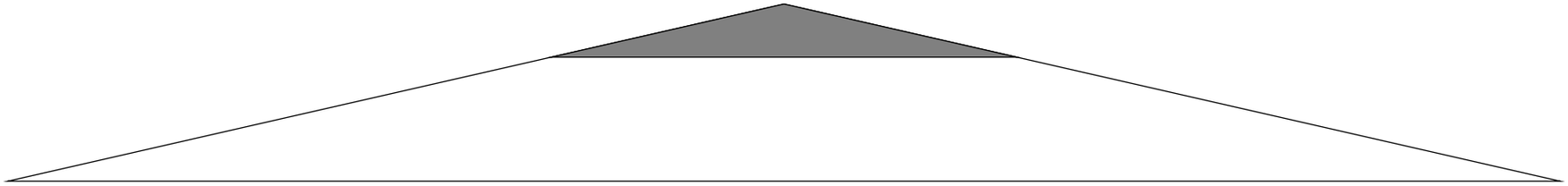
$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

- Now what is this mysterious evaluation function 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 $V_{\mathsf{minmax}}(s)$ 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.