1 Adversarial Search

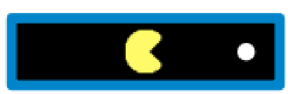
Adversarial search problems, more commonly known as **games** where our agents have one or more **adversaries**, and take actions with either deterministic or **stochastic** (probabilistic) outcomes, can have any variable number of players, and may or may not be **zero-sum**. As opposed to normal search, which returns a comprehensive plan, adversarial search returns a **strategy**, or **policy**, which simply recommends the best possible move given some configuration of our agent(s) and their adversaries. Giving rise to behavior through computation.

The standard game formulation consists of the following definitions:

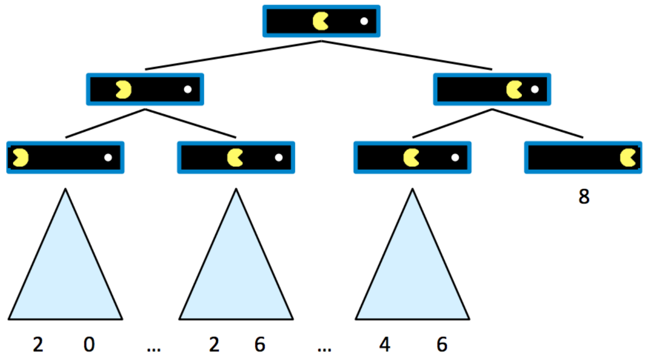
- Initial state, s_0
- Players, $Players(s)$ denote whose turn is
- Actions, $Actions(s)$ available actions for the player
- Transition model $Result(s, a)$
- Terminal test, $Terminal - test(s)$
- Terminal values, $Utility(s, player)$

1.1 Minimax

minimax runs under the motivating assumption that the opponent we face behaves optimally, and will always perform the move that is worst for us. To introduce this algorithm, we must first formalize the notion of **terminal utilities** and **state value**. The value of a state is the optimal score attainable by the agent which controls that state. In order to get a sense of what this means, observe the following trivially simple Pacman game board:



Assume that Pacman starts with 10 points and loses 1 point per move until he eats the pellet, at which point the game arrives at a **terminal state** and ends. We can start building a **game tree** for this board as follows, where children of a state are successor states just as in search trees for normal search problems:



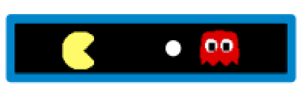
It's evident from this tree that if Pacman goes straight to the pellet, he ends the game with a score of 8 points, whereas if he backtracks at any point, he ends up with some lower-valued score. Now that we've generated a game tree with several terminal and intermediary states, we're ready to formalize the meaning of the value of any of these states.

A state's value is defined as the best possible outcome (**utility**) an agent can achieve from that state. Simply, think of an agent's utility as its score or number of points it attains. The value of a terminal state, called a **terminal utility**, is always some deterministic known value and an inherent game property. In our Pacman example, the value of the rightmost terminal state is simply 8, the score Pacman gets by going straight to the pellet. Also, in this example, the value of a non-terminal state is defined as the maximum of the values of its children. Defining $V(s)$ as the function defining the value of a state s, we can summarize the above discussion:

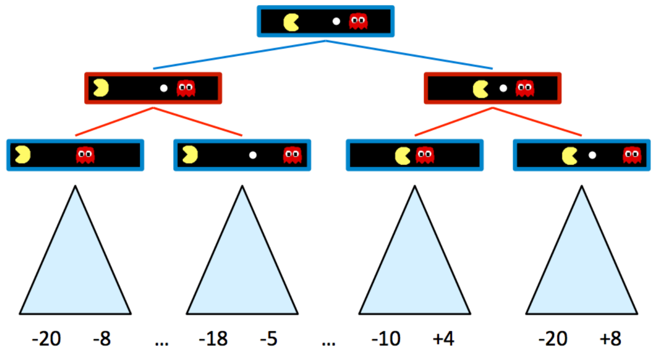
$$\begin{aligned} \forall \text{ non-terminal states,} \quad V(s) &= \max_{s' \in successors(s)} V(s') \\ \forall \text{ terminal states,} \quad V(s) &= \text{known} \end{aligned}$$

This sets up a simple recursive rule, from which it should make sense that the value of the root node's direct right child will be 8, and the root node's direct left child will be 6, since these are the maximum possible scores the agent can obtain if it moves right or left, respectively, from the start state. It follows that by running such computation, an agent can determine that it's optimal to move right, since the right child has a greater value than the left child of the start state.

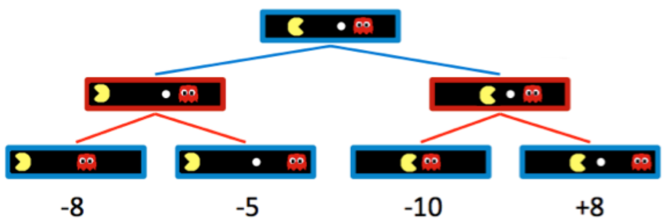
Let's now introduce a new game board with an adversarial ghost that wants to keep Pacman from eating the pellet.



The two agents take turns making moves, leading to a game tree where the two agents switch off/on layers of the tree that they "control." Here's the game tree that arises from the new two-agent game board above:



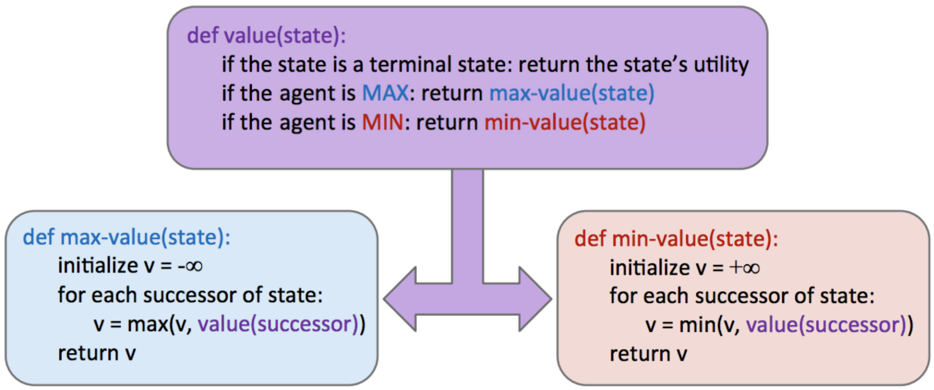
Blue nodes correspond to nodes that Pacman controls and can decide what action to take, while red nodes correspond to ghost-controlled nodes. Note that all children of ghost-controlled nodes are nodes where the ghost has moved either left or right from its state in the parent, and vice versa for Pacman-controlled nodes. For simplicity purposes, let's truncate this game tree to a depth-2 tree, and assign spoofed values to terminal states as follows:



Naturally, adding ghost-controlled nodes changes the move Pacman believes to be optimal, and the new optimal move is determined with the minimax algorithm. Instead of maximizing the utility over children at every level of the tree, the minimax algorithm only maximizes over the children of nodes controlled by Pacman, while minimizing over the children of nodes controlled by ghosts. Hence, the two ghost nodes above have values of $\min(-8,-5) = -8$ and $\min(-10,+8) = -10$ respectively. Correspondingly, the root node controlled by Pacman has a value of $\max(-8,-10) = -8$. Since Pacman wants to maximize his score, he'll go left and take the score of -8 rather than trying to go for the pellet and scoring -10. This is a prime example of the rise of behavior through computation - though Pacman wants the score of $+8$ he can get if he ends up in the rightmost child state, through minimax he "knows" that an optimally-performing ghost will not allow him to have it. In order to act optimally, Pacman is forced to hedge his bets and counterintuitively move away from the pellet to minimize the magnitude of his defeat. We can summarize the way minimax assigns values to states as follows:

$$\begin{aligned}
\forall \text{ agent-controlled states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\
\forall \text{ opponent-controlled states}, \quad V(s) &= \min_{s' \in successors(s)} V(s') \\
\forall \text{ terminal states}, \quad V(s) &= \text{known}
\end{aligned}$$

In implementation, minimax behaves similarly to depth-first search, computing values of nodes in the same order as DFS would, starting with the the leftmost terminal node and iteratively working its way rightwards. More precisely, it performs a **postorder traversal** of the game tree. The resulting pseudocode for minimax is both elegant and intuitively simple, and is presented below. Note that minimax will return an action, which corresponds to the root node's branch to the child it has taken its value from.

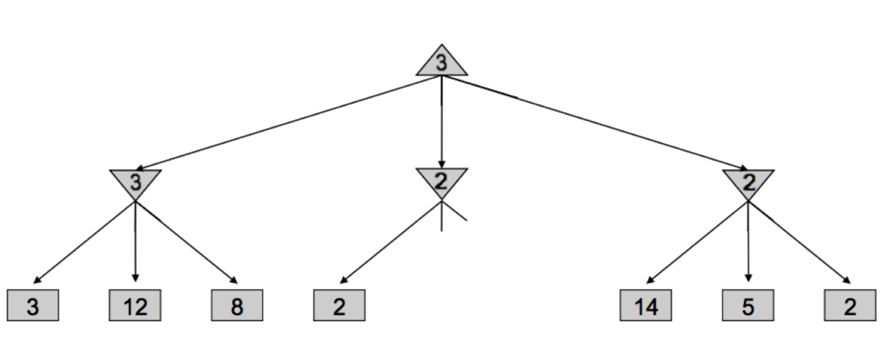


Alpha-Beta Pruning

Minimax seems just about perfect - it's simple, it's optimal, and it's intuitive. However its time complexity, similar to depth-first search, is a dismal $O(b^m)$. Recalling that b is the branching factor and m is the approximate tree depth at which terminal nodes can be found, this yields far too great a runtime for many games. For example, chess has a branching factor $b \approx 35$ and tree depth $m \approx 100$. To help mitigate this issue, minimax has an optimization - **alpha-beta pruning**.

Conceptually, alpha-beta pruning is this: if you're trying to determine the value of a node n by looking at its successors, stop looking as soon as you know that n's value can at best equal the optimal value of n's parent. Consider the following game tree, with square nodes corresponding to terminal states, downward-pointing triangles corresponding to minimizing nodes, and upward-pointing triangles corresponding to maximizer nodes:

Let's walk through how minimax derived this tree - it began by iterating through the nodes with values 3, 12, and 8, and assigning the value $\min(3, 12, 8) = 3$ to the leftmost minimizer. Then, it assigned $\min(2, 4, 6) = 2$ to the middle minimizer, and $\min(14, 5, 2) = 2$ to the rightmost minimizer, before finally assigning $\max(3, 2, 2) = 3$ to the maximizer at the root. However, if we think about this situation, we can come to the realization that as soon as we visit the child of the middle minimizer with value 2, we no longer need to look at the middle minimizer's other children. Why? Since we've seen a child of the middle minimizer with value 2, we know that no matter what values the other children hold, the value of the middle minimizer can be at most 2. Now that this has been established, let's think one step further still - the maximizer at the root is deciding between the value of 3 of the left minimizer, and the value that's ≤ 2, it's guaranteed to prefer the 3 returned by the left minimizer over the value returned by the middle minimizer, regardless of the values of its remaining children. This is precisely why we can **prune** the search tree, never looking at the remaining children of the middle minimizer:



Implementing such pruning can reduce our runtime to as good as $O(b^{m/2})$, effectively doubling our "solvable" depth and is implemented as follows:

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Take some time to compare this with the pseudocode for vanilla minimax, and note that we can now return early without searching through every successor.

Evaluation Functions

These are functions that take in a state and output an estimate of the true minimax value of that node. Typically, this is plainly interpreted as "better" states being assigned higher values by a good evaluation function than "worse" states. Evaluation functions are widely employed in **depth-limited minimax**, where we treat non-terminal nodes located at our maximum solvable depth as terminal nodes, giving them mock terminal utilities as determined by a carefully selected evaluation function. Because evaluation functions can only yield estimates of the values of non-terminal utilities, this removes the guarantee of optimal play when running minimax.

Typically, domain knowledge and experimentation are used for the selection of an evaluation function when designing an agent that runs minimax, and the better the evaluation function is, the closer the agent will come to behaving optimally. Additionally, going deeper into the tree before using an evaluation function also tends to give us better results. These functions serve a very similar purpose in games as heuristics do in standard search problems.

The most common design for an evaluation function is a linear combination of **features**.

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

Each $f_i(s)$ corresponds to a feature extracted from the input state s, and each feature is assigned a corresponding **weight** w_i. Features are simply some element of a game state that we can extract and assign a numerical value. For example, in a game of checkers we might construct an evaluation function with 4 features: number of agent pawns, number of agent kings, number of opponent pawns, and number of opponent kings. We'd then select appropriate weights based loosely on their importance. It makes sense to select positive weights for our agent's pawns/kings and negative weights for our opponents pawns/kings. Furthermore, we might decide that since kings are more valuable pieces in checkers than pawns, the features corresponding to our agent's/opponent's kings deserve weights with greater magnitude than the features concerning pawns. Below is a possible evaluation function that conforms to the features and weights we've just brainstormed:

$$Eval(s) = 2 \cdot agent\_kings(s) + agent\_pawns(s) - 2 \cdot opponent\_kings(s) - opponent\_pawns(s)$$

Evaluation function design can be quite free-form and doesn't necessarily have to be linear functions either. For example, nonlinear evaluation functions based on neural networks are very common in Reinforcement Learning applications.

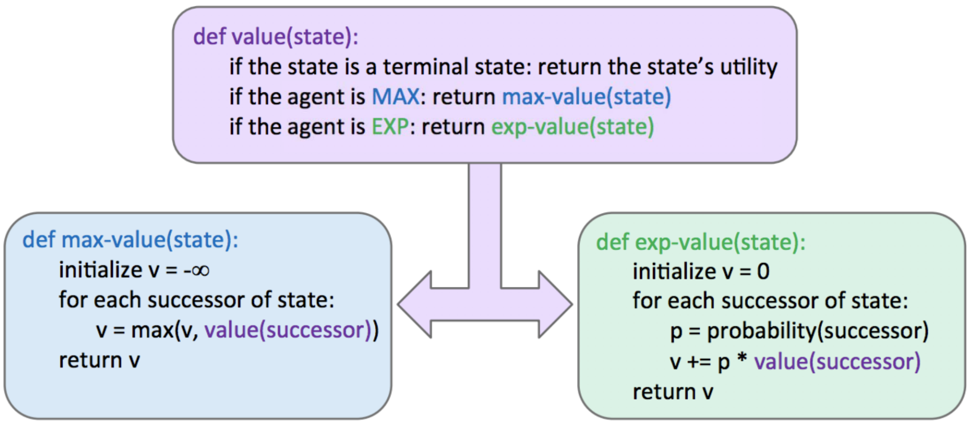
1.2 Expectimax

Minimax is often overly pessimistic in situations where optimal responses to an agent's actions are not guaranteed. Such situations include scenarios with inherent randomness such as card or dice games or unpredictable opponents that move randomly or suboptimally. This randomness can be represented through a generalization of minimax known as **expectimax**. Expectimax introduces *chance nodes* into the game tree, which instead of considering the worst case scenario as minimizer nodes do, considers the *average case*. More specifically, while minimizers simply compute the minimum utility over their children, chance nodes compute the **expected utility** or expected value. Our rule for determining values of nodes with expectimax is as follows:

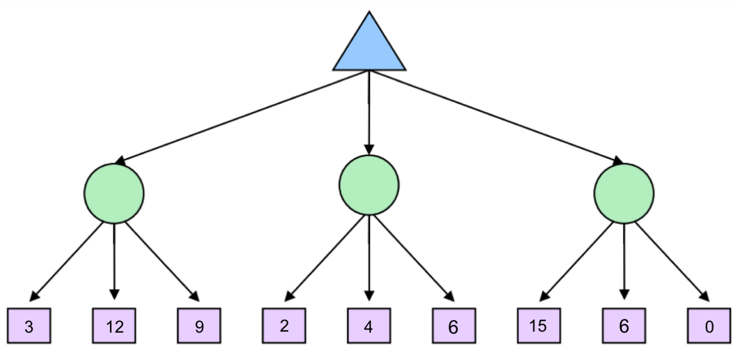
$$\begin{aligned}
\forall \text{ agent-controlled states,} \quad V(s) &= \max_{s' \in successors(s)} V(s') \\
\forall \text{ chance states,} \quad V(s) &= \sum_{s' \in successors(s)} p(s'|s)V(s') \\
\forall \text{ terminal states,} \quad V(s) &= \text{known}
\end{aligned}$$

In the above formulation, $p(s'|s)$ refers to either the probability that a given nondeterministic action results in moving from state s to s', or the probability that an opponent chooses an action that results in moving from state s to s', depending on the specifics of the game and the game tree under consideration. From this definition, we can see that minimax is simply a special case of expectimax. Minimizer nodes are simply chance nodes that assign a probability of 1 to their lowest-value child and probability 0 to all other children.

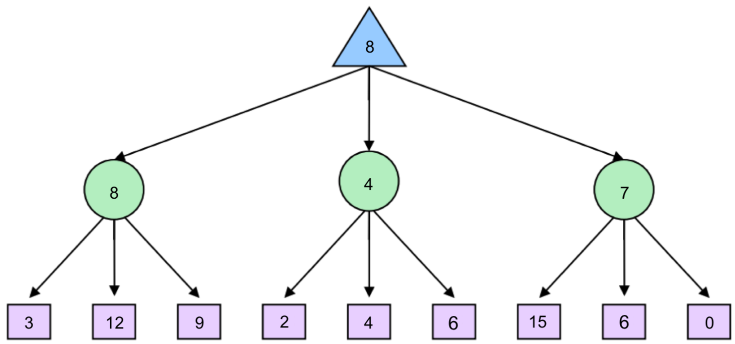
The pseudocode for expectimax is quite similar to minimax, with only a few small tweaks to account for expected utility instead of minimum utility, since we're replacing minimizing nodes with chance nodes:



Consider the following expectimax tree, where chance nodes are represented by circular nodes instead of the upward/downward facing triangles for maximizers/minimizers.



Assume for simplicity that all children of each chance node have a probability of occurrence of $\frac{1}{3}$. Hence, from our expectimax rule for value determination, we see that from left to right the 3 chance nodes take on values of $\frac{1}{3}\cdot 3+\frac{1}{3}\cdot 12+\frac{1}{3}\cdot 9 = \boxed{8}$, $\frac{1}{3}\cdot 2+\frac{1}{3}\cdot 4+\frac{1}{3}\cdot 6 = \boxed{4}$, and $\frac{1}{3}\cdot 15+\frac{1}{3}\cdot 6+\frac{1}{3}\cdot 0 = \boxed{7}$. The maximizer selects the maximimum of these three values, $\boxed{8}$, yielding a filled-out game tree as follows:



As a final note on expectimax, it's important to realize that, in general, it's necessary to look at all the children of chance nodes – we can't prune in the same way that we could for minimax. Unlike when computing minimums or maximums in minimax, a single value can skew the expected value computed by expectimax arbitrarily high or low. However, pruning can be possible when we have known, finite bounds on possible node values.

1.3 Summary

In this note, we discussed adversarial search problems, specifically the introduction of other agents (random, or deterministically adversarial), algorithms that produce policies/strategies dynamically based on the state of the agent, and adversaries.

Depending on the game attributes i.e the players, transition function, etc. we can employ a variety of search techniques, including: MinMax Search and ExpectiMax Search, or a combination of both depending on the agents and their objectives in the game. We discussed how Alpha-beta pruning can help reduce the time complexity of minimax search by leveraging the adversarial bounds of opponent agents in zero-sum settings, having the following effect:

- Does not affect the minimax value at the root node.
- Intermediate nodes might be wrong.
- Good ordering of child nodes improves the effectiveness of pruning.

We also discussed good evaluation functions (similar to heuristics in informed search, which require experimentation and domain knowledge of the game world/objectives) combined with depth-limited minimax search can further improve search time albeit at the the cost of no optimal play guarantee. Note: that the deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters, depicting a tradeoff between the complexity of features and the complexity of computation.

Finally, we discussed expectimax - a generalization of minimax wherein adversaries act non-deterministically, hence, minimax can be viewed as a special case of expectimax, wherein minimizer nodes are simply chance nodes that assign a probability of 1 to their lowest-value child and probability 0 to all other children.