Minimax

Team Chairun

November 27, 2020

Review of State Spaces

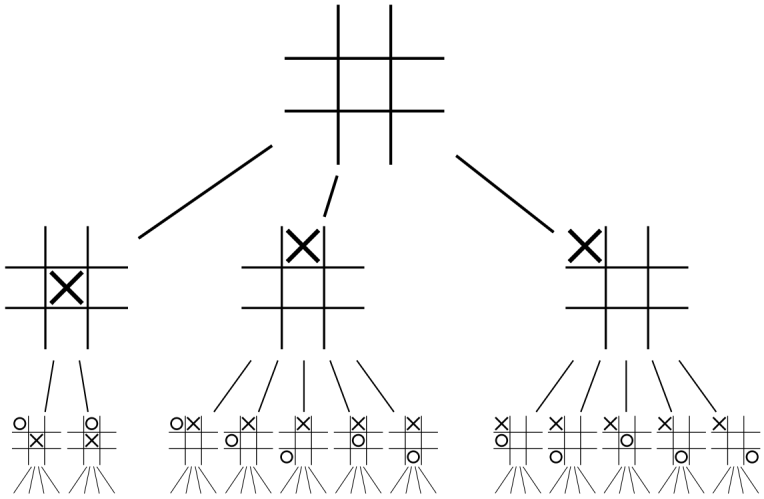
Previously, you may have been introduced to the idea of a **state space**. The motivation behind defining a **state** for a system is that it contains all the information we care about, allowing us to ignore previous states. This idea can be formalized by the Markov property, that the future evolution of the system or choices among actions depend only on the current state.

Note that dependence on previous states can often be eliminated by simply augmenting our definition of the state to include relevant information from previous states. For example, if we care about the direction of motion of a helicopter, we might include its previous position in the state so that this could be determined. Constructing a state appropriately allows us to analyze a system more conveniently.

Game Trees

Games are often represented by state spaces since there are usually only a few important variables. For example in Tic-Tac-Toe, the state can be described by the board - all other information is irrelevant.

To pick an action after defining a game state, we should understand the next state. Since each state has a "parent" state that it came from, the possible states that can evolve from a given state can be visualized as a game tree where "parent" states point to "child" states.



Tic-Tac-Toe game tree from Wikipedia

In the game tree above, the nodes are game states and the edges are possible actions that players can take to move from one state to another. For two player games like Tic-Tac-Toe or Chess, the moving player alternates as we go down each level of nodes and edges. For an n-player game where players take turns in order, the levels would cycle through the players from 1 to n. The root of the tree is the starting position we search from, in this case the beginning of the game.

Minimax

Now that we know how states evolve, how can we figure out the best move?

To simplify our analysis, we will focus on two-player **zero-sum** games. A two-player zero-sum game is one where **reward** values for players sum to 0. A common class of zero-sum games are those where we reward the winning player $+1$ and the losing player -1 so that the sum of rewards is 0.

In this setting, there is a very common algorithm for finding the best move that is known as **minimax**. After the first player moves, the second player wishes to maximize their reward from the new state. Since the game is zero-sum, the first player wishes to minimize the reward of the second player in order to maximize their own. Then the first player wishes to minimize the maximum reward that the second player can obtain after their move. Iterating this logic, the second player will want to maximize the minimum reward that they can get after the first player's next move.

These alternating turns of minimizing and maximizing a reward function lead to the name minimax. The algorithm is also known as **maximin** if the first player instead maximizes and the second player minimizes but these are conceptually identical and are used interchangeably.

Minimax Example

	B Cooperates	B Defects
A Cooperates	0	-4
A Defects	-1	-3

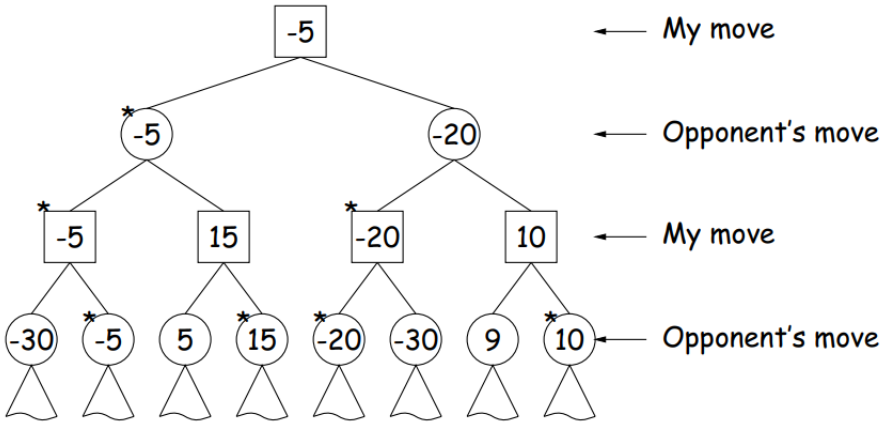
Table 1: Rewards for Player A in a Zero-Sum Game

Next we will look at a simple example of how minimax can be applied to find the optimal move. Consider the game above between two players A and B where player A moves first and each player can either cooperate or defect. The rewards for player A are shown above and the rewards for player B are the opposite values since the game is zero-sum.

Now A wants to decide what to do and knows B will respond by picking the state with the minimum reward for A. If A chooses to cooperate, B will minimize over $\{0, -4\}$ and will choose to defect, leaving A with -4. If A instead chooses to defect, B minimizes over $\{-1, -3\}$ and defects and A receives a reward of -3. Since A wants to maximize between these two options, A chooses to defect and gets a reward of $\max(\min(0, -4), \min(-1, -3)) = \max(-4, -3) = -3$. The alternating maxes and mins in this expression arise naturally from the competing goals of players A and B.

Minimax Tree

For larger games, the minimax algorithm is more easily visualized via a tree.



Minimax Tree Example from CS 61B

We begin with leaf nodes corresponding to ending game states with known reward. From these states, the rewards at the states one level above can be determined. At this level, you are the moving player so you look at the two child leaf nodes of each node and pick the one with maximum value. This yields the values $\max(-30, -5) = -5, \max(5, 15) = 15, \max(-20, -30) = -20, \max(9, 10) = 10$. At the level above this, it is your opponent's move and they minimize over the children of each node, yielding $\min(-5, 15) = -5, \min(-20, 10) = -20$. Finally at the top level, you calculate $\max(-5, -20) = -5$.

This example illustrates how minimax can be used to evaluate intermediate game states. Furthermore, by storing the calculated optimal actions inside each node, you can easily execute the optimal strategy for a game. Due to the dependence of parent nodes on their child nodes, the minimax algorithm can be described via the following recursive procedure.

Minimax Algorithm

1. If the game is over, return the value for this state.
2. Otherwise, loop over each possible action.
3. Execute the minimax algorithm on each of the resulting states with the opposite player to calculate their values.
4. Return the action with maximum/minimum value if you are the maximizing/minimizing player.

The games we have mentioned so far (Chess, Tic-Tac-Toe, the two games above) all are deterministic games. For stochastic games, there is an extension of minimax called **expectiminimax**, which instead minimizes/maximizes over the expected reward to choose actions and calculate expected rewards of earlier states. The reward function must be defined carefully with this algorithm. If reward for a checkers game was chosen to be number of leftover pieces, the algorithm would potentially maximize margin of victory at the cost of winning chances. In contrast, defining reward as ± 1 for a win or loss would optimize winning probability. Expectiminimax is quite powerful and often performs well in the CS61A Hog Contest.

Heuristics

The above method requires that we iterate over all game states starting from the ending states and building upwards. But this is computationally infeasible when there are too many states. For instance, the board game Othello has about 10^{28} states and a laptop computer wouldn't finish iterating over this many states before the end of the universe.

To address this issue, we can cut off our tree at some depth and try to get a reasonable estimate of the values of nodes at this depth. Then we can simply apply minimax to this truncated tree to calculate the optimal actions based on our estimates.

How can we get these estimated values of states? The function that is used to map from states to estimated values is called a **heuristic**, and will of course vary from game to game. Heuristics usually combine together various parameters of the state into estimates. For example a simple Tic-Tac-Toe heuristic might be (X's in a row) $-$ (O's in a row) since we might expect the first player to be more likely to win if there are more X's and less likely to win if there are more O's. The problem of choosing such a heuristic is a good candidate for machine learning since it is effectively a regression problem and training data is easily obtained.

Interestingly, minimax can itself be thought of as searching up to a given depth and then evaluating a heuristic function which happens to require searching further. This suggests that with a sufficiently good heuristic, searching up to a given depth can be almost as good as minimax. For moderately large games, this strategy can often outperform humans.

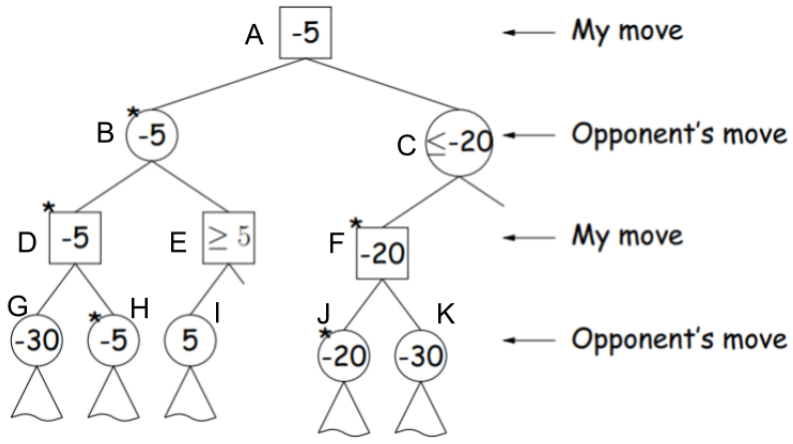
Alpha-Beta Pruning

Though heuristics allow us to come up with estimated values from limited depth searches, they do not address the issue of severely reduced search depth for larger games. A board state in Chess has about 35 legal moves on average so the number of possible states increases exponentially with search depth. The number 35 is called the **branching factor** since each state branches out into ~ 35 new states. A search 6 moves deep must consider $\sim 35^6 \approx 2$ billion moves which is already pushing the limits of a typical computer. Some good Chess players are able to see well beyond 6 moves ahead, so how are computers able to vastly outperform the best players in the world?

Aside from sophisticated evaluation functions, the answer is a more efficient search algorithm known as **alpha-beta pruning**. The key idea of alpha beta pruning is that we can can skip or "prune" a subtree if a player is guaranteed a better outcome by choosing a different action in some ancestor. In each state, we maintain two values α and β which are the minimum and maximum values that players can achieve in the current state or its ancestors. If these values are updated so that $\beta < \alpha$, we will know that a player can achieve a better value in one of the ancestors and can prune this current state's subtree.

Alpha-Beta Pruning Example

To gain more intuition, we demonstrate alpha-beta pruning with the same game tree as above.



Alpha-Beta Pruning Example from CS 61B

We begin from A and initialize worst case values of $\alpha = -\infty, \beta = \infty$. We move left down the tree and reach D. At D, we first see G has value -30. It is the maximizing player's turn so we update $\alpha = \max(-\infty, -30) = -30$. Then we see H, update $\alpha = \max(-30, -5) - -5$, and return -5.

Now at B the minimizing player can force a value of -5 by moving to D so β at B is updated to $\min(\infty, -5) = -5$. We move to B's right child, E, with this new β. Child I has value 5 so the maximizing player can achieve $\alpha = \max(-\infty, 5) = 5$ at E. However, $\beta < \alpha$ since the minimizing player can guarantee a lower value at D so we prune E's subtree and B's value becomes -5.

After α is updated to -5 at A, we move to the right subtree and find F has value $\max(-20, -30) = -20$. The minimizing player can move to F from C so $\beta = \min(\infty, -20) = -20$ at C. However, C was passed $\alpha = -5$ from A and $\beta < \alpha$ since the maximizing player will always choose -5 over -20

at A. So C's subtree is pruned and we find that the starting state A has value -5.

With alpha-beta pruning, we can search almost twice as deep as minimax in roughly the same time.

The alpha-beta pruning algorithm can be summarized as follows.

Alpha-Beta Pruning

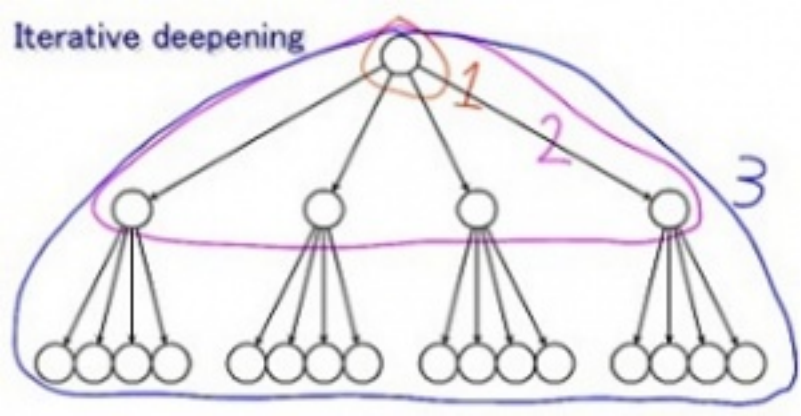
1. If the current state is at our search depth limit or is an ending state, return its heuristic value.
2. Otherwise, iterate over each possible action.
3. For each of the resulting states, run alpha-beta pruning for the opposite player with the current α and β values.
4. Update α with the state value if you are the maximizing player or β if you are the minimizing player.
5. If $\beta < \alpha$ after updating, return the best action and exit.
6. Otherwise, if the current action has better value than the best action so far, update the best action.
7. After the iteration is finished, return the best action.

Iterative Deepening

In practice, we usually want the computer to calculate a reasonable move in some fixed amount of time. We could easily do this if we knew how long it would take to search at a given depth. But this can vary based on the branching factor of the game at various points and how often we are able to prune using alpha-beta pruning. Furthermore, it's important to finish searching at a given depth before searching deeper. Otherwise, we could be searching suboptimal moves very deeply while neglecting the best move in another part of the tree.

An initial approach that you might come up with is breadth first search. Since this search gradually increases depth, we could keep searching deeper and deeper until time runs out and then return the best move at the highest depth we have finished searching. Unfortunately, breadth first search requires that the previous level of nodes is stored in a queue and the number of nodes in a level increases exponentially so we will quickly run out of memory. Furthermore, alpha beta pruning involves searching one subtree deeper in order to bound state values and avoid searching other subtrees. But if we are only increasing depth by 1 each iteration, there is little opportunity for this.

An idea of **iterative deepening** is used to resolve this issue. We start by running a search at a small depth like $d = 1$ and keep rerunning the entire search, increasing d by 1 each time. If we use depth first search for this, our memory is only the $O(d)$.



Iterations of Iterative Deepening from Chess Wiki

This might seem extremely inefficient - why are we redoing all of our previous work every time we search? Surprisingly, since the number of nodes in a game tree usually increases exponentially at each level, all of the previous search times become negligible compared to the search time for the last level for a sufficiently large branching factor. Iterative deepening ends up having the same asymptotic runtime as breadth first search while having vastly better space and using alpha-beta pruning on each individual search. As a result, iterative deepening allows for adaptable game engines that can perform well in time-limited settings.

Transposition Tables

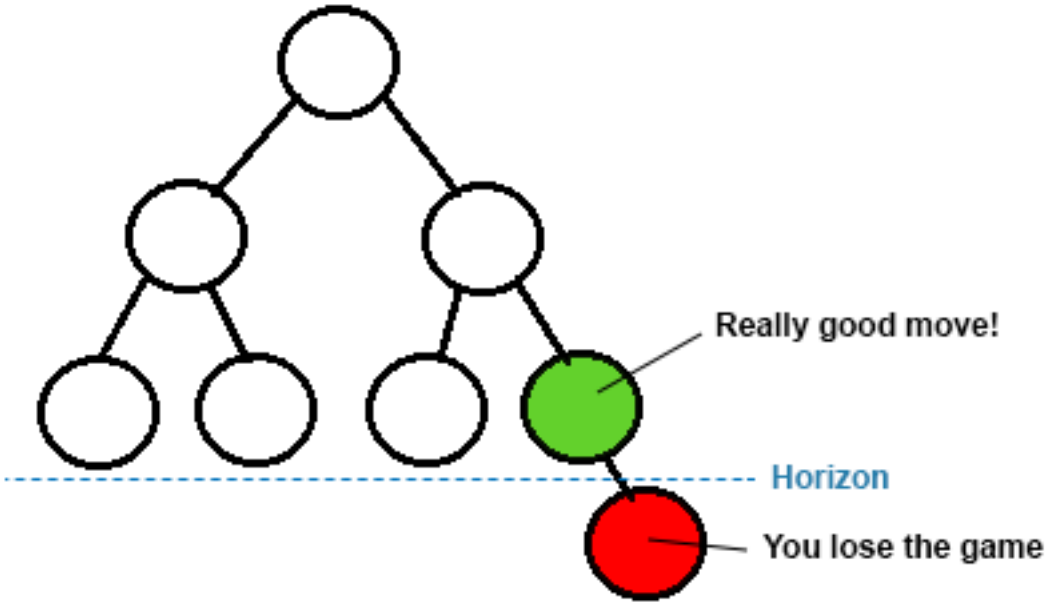
When searching a game's state space, we may run into positions that we have seen before. This can occur more often than expected because rearrangements of moves can result in the same position. For example in Go and Othello, placing pieces in different orders may result in identical board states. These reorderings of moves are called **transpositions**. In such cases, we might want to save the repeated board states so that we don't waste time redoing searches.
As usual, hashing is an excellent way to store a table of search results. This table is known as a **transposition table**. Some common pieces of information that might be stored in a position entry are search depth, estimated evaluation of the state at that depth, and the best responding move, although this can vary from game to game. When we run into a position again, we can check if it already has an entry of sufficient depth and if so, we can reuse this entry. Otherwise, we evaluate at the current depth and store the results depending on our policy for storing positions. In the notebook, you'll see that transposition tables may slow down searching when there are many moves but are incredibly helpful in Chess endgames when there are many repeated states.
Of course, we have limited space in our table so eventually we must choose how to replace, or "evict" table entries. There are some criteria to trade off here: deeply-searched positions will save more time but recently-seen positions or positions near leaves may be encountered more often. In practice, the best performing strategies usually balance these objectives.

Move Ordering

When we encounter a good position in alpha-beta pruning, we are able to prune away many worse positions since we have already found a better option. A real life example where this occurs is college applications: after getting accepted to an excellent school like Berkeley you're able to "prune" safety schools like Stanford and UCLA. This could save you a lot of time filling out applications and likewise it saves a huge amount of time searching inferior positions in game trees.
As a concrete example of how we would implement this, consider the board game Connect 4 where the objective is to get 4 pieces connected in a row. Moves that create a line of 4 pieces should be searched first since these win immediately. In general, moves that create runs of many pieces are likely to be better so searching these moves earlier may allow us to prune more states later on. To implement this, we can use specific knowledge of which moves are likely to be better but can also consider good moves already saved in our transposition table. This allows us to achieve faster runtime and potentially search important states more deeply, giving us a significant advantage.

The Horizon Effect and Quiescence Search

A key problem with doing limited depth searches is that events beyond the depth boundary or "horizon" of our search will be undetected. We might search up to depth d with our AI and not see that right beyond the horizon at depth $d+1$, we will lose the game.



A Loss Lurking Just Beyond the Horizon from krystman.itch.io

Humans usually search these volatile positions more deeply until the valuation of the position has settled and they are more confident about their estimate. Imitating this strategy for computers leads to the idea of **quiescence search**.
To implement quiescence search, we first require some estimate of how volatile a position is. Certain

move types such as captures may be more volatile which might lead us to implement a **capture search** where we search through all captures. Alternatively, we might look at heuristic valuations of nearby states to estimate the volatility of our current state and search deeper if there is too much variation.
At the largest depth of minimax, rather than returning a static evaluation of the state we call quiescence search on the state. Quiescence search does another search of a new depth, for only a particular limited set of moves. Quiescence search terminates either when the position is quiet, or non-volatile, or when we search the adequate depth. Quiescence search makes it more likely we will not stop our search on nodes which will have drastic changes in evaluation in the next moves. While we are unable to fully overcome the horizon effect, we can mitigate it so its effects are not as potent on our search, especially at lower depths.

References

- CS189 course slides
- CS61B course slides
- Chessprogramming wiki
- Wikipedia