Games

Daniel Gigliotti

1 Games

Games have played a pivotal role in the development of artificial intelligence by providing structured environments for testing and advancing AI algorithms and techniques. In the context of AI, games refer to various types of competitive and strategic activities where multiple players or agents interact according to predefined rules. These interactions make games an excellent domain to study decision-making, planning, reasoning, and optimization.

2 Minmax

The most common search technique in game playing is the minmax algorithm. It is a depth-first, depth-limited, general-purpose search procedure, but it can be used in games to determine the optimal decision to make in an adversarial setting. An adversarial game is a game in which you and your opponent have opposite goals: your goal is for you to win and for your opponent to lose, your opponent's goal is for them to win and for you to loose. Using the minmax algorithm we suppose both players are playing optimally: playing optimally is about ensuring the best outcome for yourself even in the worst case scenario.

The central idea behind the minmax algorithm is to create a game tree that represents all possible moves and counter-moves, up to a certain depth, from the current game state. Each node in the tree represents a game state, and the edges represent possible moves that lead to subsequent game states. The leaves of the tree are evaluated using a heuristic function that estimates the desirability of the final outcomes from that state.

Here's how the minmax algorithm works:

1. **Build the Game Tree**: Starting from the current game state, generate all possible moves and create a new node for each resulting game state. This process continues for a certain depth or until a terminal state (win, lose, or draw) is reached.

2. **Evaluate Terminal Nodes**: If a terminal state is reached, assign a value to that node based on the outcome of the game (e.g., +1 for a win, -1 for a loss, 0 for a draw).

3. **Propagate Values Upwards**: Backpropagate the values from the terminal nodes to their parent nodes. For nodes representing the current player's turn, choose the maximum value of its child nodes. For nodes representing the opponent's turn, choose the minimum value of its child nodes. This alternation of maximizing and minimizing gives the algorithm its name, minmax.

4. **Choose the Best Move**: Once the values have been propagated up to the root node, the algorithm chooses the move that corresponds to the child node with the highest value. This is the move that the current player should make to maximize their chances of winning.

The minmax algorithm assumes that both players play optimally, which means that the current player makes decisions to maximize their own outcome, while assuming that the opponent will make decisions to minimize the current player's outcome. However, in practice, the minmax algorithm can be computationally expensive, especially in complex games like chess, due to the exponential growth of the game tree.

To alleviate this issue, optimizations like Alpha-Beta Pruning are often applied. Alpha-Beta Pruning helps in reducing the number of nodes evaluated by eliminating branches that are guaranteed to be worse than previously evaluated alternatives.

```
minmax(s):

    if Terminal(s):
        return Value(s)

    if Player(s) == MAX:
        value = -inf
        for a in Actions(s):
            value = max(value, minmax(Result(s, a)))
        return value

    if Player(s) == MIN:
        value = inf
        for a in Actions(s):
            value = min(value, minmax(Result(s, a)))
        return value
```

- Terminal(state) will tell us if the state is terminal (no more moves left or one of the two players won);
- Value(state) will assign a value to the state; the value should be something that the MAX player want to maximize and that the MIN player want to minimize;
- Player(state) will tell us which player needs to do the move;
- Actions(state) will tell us which actions are available to the player that must do the move;
- Result(state, action) takes a state and an action and returns the new state resulted from applying the action to the old state.

The minmax algorithm is:

- Complete (if the tree is finite)
- Optimal (if both players play optimally)

and has:

- Time complexity $O(b^m)$
- Space complexity $O(bm)$ (depth-first exploration)

The problem with the minmax is that the number of states it has to examine is exponential in depth of the tree.

3 $\alpha - \beta$ pruning

Alpha-Beta Pruning is an optimization technique used in the minmax algorithm to significantly reduce the number of nodes that need to be evaluated in the search tree. It aims to improve the efficiency of the search process by eliminating branches of the game tree that are guaranteed to not affect the final decision. This optimization is particularly valuable in complex games with a large branching factor, such as chess or Go, where the full minmax algorithm would be computationally expensive.

The basic idea of Alpha-Beta Pruning is to maintain two values at each node of the game tree:

- **Alpha**: The best value that the maximizing player has found so far along the path from the root to the current node.
- **Beta**: The best value that the minimizing player has found so far along the same path.

During the minmax search, when evaluating nodes, the algorithm compares the values of alpha and beta to determine whether certain branches of the tree can be pruned (ignored) without affecting the final decision.

4 Monte Carlo Tree Search (MCTS)

The Monte Carlo Tree Search (MCTS) algorithm is a heuristic search technique used for decision-making in games and other domains with a large state space and uncertain outcomes. MCTS has gained significant popularity and success in various AI applications, particularly in playing complex games like Go, chess, and poker. It was introduced as a way to address some of the limitations of traditional search algorithms like Minimax, which struggle with high branching factors and vast state spaces.

The algorithm iteratively does simulations (rollout) of complete games to estimate the heuristic values of moves using Monte Carlo simulations. It keeps playing against itself and compute the win percentage.

- **Simulations (Rollouts) of Complete Games**: MCTS simulates entire sequences of moves, referred to as **rollouts** or **playouts**, to evaluate the potential outcome of a given move. These simulations start from the current game state and proceed by selecting random moves for both players until a terminal state (win, lose, or draw) is reached. By performing these simulations, MCTS gains insight into the potential outcomes of various moves.
- **Estimating Heuristic Values of Moves**: During each simulation, the final outcome of the game is used to estimate the value or quality of the moves made during the simulation. For example, if the simulation ends

in a win for a specific player, that player's moves leading up to the win are considered favorable and are assigned a higher value. Similarly, if the simulation ends in a loss, those moves are assigned a lower value.

- **Monte Carlo Simulations**: The term **Monte Carlo** refers to the statistical method of using random sampling to estimate results. In MCTS, Monte Carlo simulations involve performing a large number of random rollouts to approximate the probabilities of different outcomes based on the actions taken.
- **Self-Play and Win Percentage**: MCTS often employs self-play, where it plays against itself during the simulations. By doing so, it explores a wide range of possible moves and counter-moves. During these self-play simulations, the algorithm records the win percentage of each move. For instance, if a certain move leads to a win in 70% of the simulations, it's assigned a higher win percentage.