Connect Four computer agents

Erik Paskalev

October 2024

1 Introduction

Games have long been at the forefront of Artificial Intelligence research as a domain well-suited for pushing the limits of the field. In this paper, we explore and implement the some of basic algorithms used for creating computer-based game-playing systems, that serve as a basis for the current state-of-the-art algorithms, namely the Minimax algorithm and alpha-beta pruning. For our game of choice, we will use Connect Four.
This choice is due to the significantly lower computational complexity compared to games like Chess, Go, Dota 2, etc. Connect Four is much simpler - having a maximum of seven moves per turn and a maximum game length of 42 moves by default. This simplicity allows for the use of inexpensive hardware and allows even simple algorithms to perform well.

2 Problem statement

The problem of creating an AI play that can play a game intelligently can be defined as two problems that build on each other.

1. An optimization problem - Given a game state find whether the position is winning, losing, or a draw assuming optimal play.

2. A search problem - Given a game state find the optimal move.

Both of these definitions are nearly identical in terms of execution and are usually used in tandem.

3 Minimax

The minimax algorithm is conceptually one of the simplest approaches to solving decision problems. The base idea of this approach is that to evaluate the

current position we will brute force all possible games that could be played from it. Player 1 is trying to reach a winning game state while player 2 is trying to reach a losing state for player 1 and we can denote this numerically as player 1 trying to maximize evaluation of the position, with player 2 minimizing it. Using this idea evaluation works by recursively playing out each game using a depth-first-search approach. The evaluation of each state is calculated once the entire subtree of states under it has been calculated and it's determined by the current player that would be playing if this state is reached and taking the minimum or maximum evaluation from child states.

This version of minimax as stated above is as a matter of fact a provably optimal algorithm. It will find the best move every time, however, that comes with a massive problem that sits at the core of most game algorithms and that is computational complexity. Complexity is the number of operations the computer would need to make to execute the algorithm even for a relatively simple game like Connect Four. trying to brute force all positions would be impossible even for a supercomputer. A basic pessimistic upper-bound for the number of positions that the minimax algorithm will evaluate for a NxM board would be the number of actions available per move to the power of the total number of moves that can be played which would be $O(N^{N*M})$. For a standard 7x6 board that is on the order of 10^{35}. Now in practice, the real number of positions is much lower, but regardless this illustrates that the base minimax will not work.

The modification we make to fix this is to add a depth limiter that tracks how many moves deep we are in the calculation and when we reach a specified depth we stop and treat the current state as a terminal one. The problem now that we have is the evaluation of this new terminal node since the game has yet to end and we cannot definitively say whether the position is winning or losing.

The solution is implementing a heuristic, an evaluation function that directly outputs an evaluation of a game state without any searching and usually, these are usually hard-coded. There is much research on Heuristics as the difference between a good heuristic and a bad one can be massive. For all testing conducted, we will use a completely uninformative heuristic (unless the state is terminal we just return the evaluation as 0).

This implementation using a heuristic is the basic minimax. The implementation for it in Python can be seen in Figure 1. The function is defined in a class MinimaxPlayer that contains some of the data regarding the game i.e board size, max depth, etc.

The minimax algorithm's complexity is relatively simple to calculate as it runs for a fixed number of iterations D with N actions at each step, so the total worst-case complexity is $O(N^D)$. This complexity can lower dramatically as you approach the endgame.

```
def explore_pos(self, board: board, curr_depth: int, curr_player_id: int) -> [int, float]:
        self.evaluation_count += 1
        # set the best position to initially be the worst possible for whichever players turn it is
        best_eval: [int, float] = [-1, -np.inf]
        if curr_player_id == 2:
            best_eval[1] = np.inf

        # checks weather the state is terminal.
        is_terminal = self.heuristic.winning(board.board_state, self.game_n)
        # return heuristic evaluation if maximum depth has been reached or current position is terminal
        if curr_depth == self.depth or is_terminal != 0:
            # return the outcome of the current position
            heuristic_eval = self.heuristic.get_best_action(curr_player_id, board)
            return heuristic_eval

        # checking if we are player 1 or 2
        if curr_player_id == 1:
            # looping over each possible action
            for col in range(board.width):
                if board.is_valid(col):
                    # generating new board state
                    new_board: board = board.get_new_board(col, curr_player_id)
                    # Recursively evaluate the position resulting from the current move
                    curr_eval: [int, float] = [col, self.explore_pos(new_board, curr_depth + 1, 2)[1]]
                    # comparing to the position with maximal evaluation found so far
                    if best_eval[1] <= curr_eval[1] :
                        best_eval = curr_eval
        else:
            # looping over each possible action
            for col in range(board.width):
                if board.is_valid(col):
                    # generating new board state
                    new_board: board = board.get_new_board(col, curr_player_id)
                    # Recursively evaluate the position resulting from the current move
                    curr_eval: [int, float] = [col, self.explore_pos(new_board, curr_depth + 1, 1)[1]]
                    # comparing to the position with minimal evaluation found so far
                    if best_eval[1] >= curr_eval[1] :
                        best_eval = curr_eval
        # backpropagating the evaluation of the current position and the action that lead to it
        return best_eval
```

Figure 1: An implementation of the basic Minimax algorithm

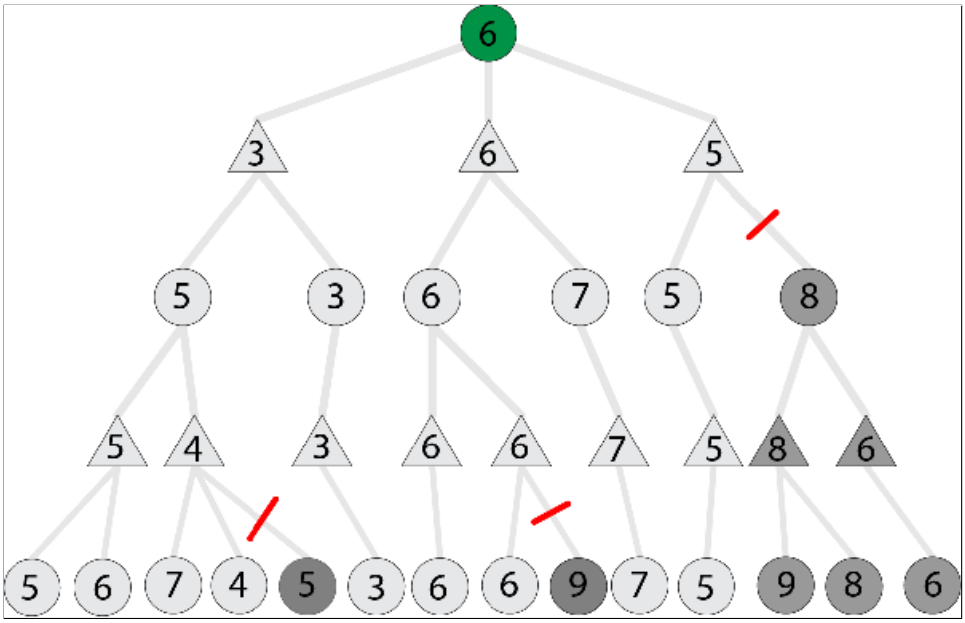

Figure 2: An example of alpha-beta pruning in practice [1]

4 Alpha-beta pruning

Alpha-beta pruning is an optimization on the base minimax that gives the same answer only more efficiently. The main idea hinges on noticing that to evaluate a position we only need to know the evaluation of either the highest or lowest evaluation depending on the current player. If in the process of computing one of the states on a layer below the best possible evaluation of that position becomes worse than the best one we have found so far then we can stop the computation. For an example see Figure 2 [1]. In practice, the implementation has only two additions over the minimax - the alpha and beta variables. These are locally defined for each game state as the best position calculated by the current point of calculation for player 1 (alpha) and the best position for player 2 (beta). If we are currently in a specific position then we check if alpha is greater than beta. If this holds then it means that in the current branch the best possible evaluation we could get is worse than an evaluation we already have so searching this branch is unnecessary and can be pruned. Otherwise, we update the values of alpha and beta with the newly calculated positions. see Figure 3 for detailed implementation.

Determining the theoretical complexity of alpha-beta pruning is more challenging than for the basic minimax as it depends on a somewhat arbitrary decision that being what order the depth-first-search takes. In the worst case,

```
def explore_pos(self, board: board, curr_depth: int, alpha: float, beta: float, curr_player_id: int) -> [int, float]:
        self.evaluation_count += 1
        # checks weather the state is terminal.
        is_terminal = self.heuristic.winning(board.board_state, self.game_n)
        # return heuristic evaluation if maximum depth has been reached or current position is terminal
        if curr_depth == self.depth or is_terminal != 0:
            # return the outcome of the current position
            heuristic_eval = self.heuristic.get_best_action(curr_player_id, board)
            return heuristic_eval

        # checking if we are player 1 or 2
        if curr_player_id == 1:
            best_eval: [int, float] = [-1, -np.inf]
            # looping over each possible action
            for col in range(board.width):
                if board.is_valid(col):
                    # generating new board state
                    new_board: board = board.get_new_board(col, curr_player_id)
                    # Recursively evaluate the position resulting from the current move
                    curr_eval: [int, float] = [col, self.explore_pos(new_board, curr_depth + 1, alpha, beta, 2)[1]]
                    # comparing to the position with maximal evaluation found so far
                    if best_eval[1] <= curr_eval[1] :
                        best_eval = curr_eval
                    # pruning the current branch if it is worse for player 2 compared to an already searched branch.
                    if best_eval[1] > beta:
                        return best_eval
                    # updating alpha aka. the best possible position found for player 1 so far
                    alpha = max(alpha, best_eval[1])
            return best_eval
        else:
            best_eval: [int, float] = [-1, np.inf]
            # looping over each possible action
            for col in range(board.width):
                if board.is_valid(col):
                    # generating new board state
                    new_board: board = board.get_new_board(col, curr_player_id)
                    # Recursively evaluate the position resulting from the current move
                    curr_eval: [int, float] = [col, self.explore_pos(new_board, curr_depth + 1, alpha, beta, 1)[1]]
                    # comparing to the position with minimal evaluation found so far
                    if best_eval[1] >= curr_eval[1] :
                        best_eval = curr_eval
                    # pruning the current branch if it is worse for player 1 compared to an already searched branch.
                    if best_eval[1] < alpha:
                        return best_eval
                    # updating beta aka. the best possible position found for player 2 so far
                    beta = min(beta, best_eval[1])
            return best_eval
```

Figure 3: Implementation of alpha-beta pruning

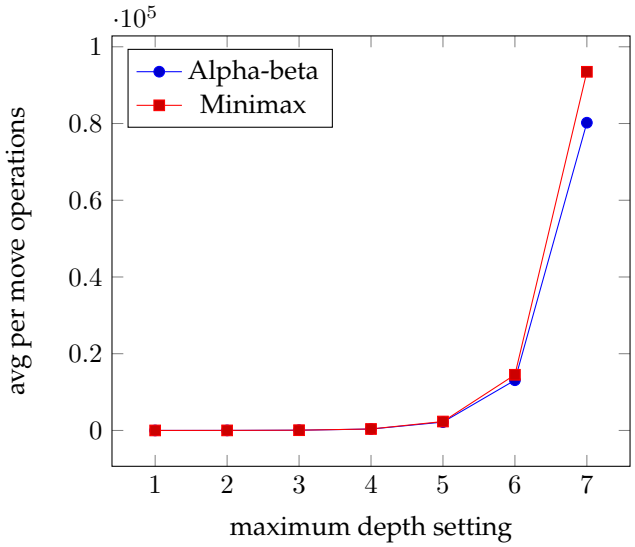

Figure 4: The graph illustrates the difference in efficiency between the two algorithms. This data was generated using the standard version of Connect Four - 7x6 board with 4 in a row to win and most importantly using no heuristic. This illustrates how important the heuristic is for alpha-beta as since the no heuristic is gives evaluation 0 if the state is not terminal then for most of the game alpha-beta does nothing as non of the early states will be terminal that's why the difference in performance is low.

alpha-beta pruning finds the optimal path last and doesn't improve on minimax having the same complexity of $O(N^D)$, but in the best case the best path is found first, and essentially half of the layers get skipped leading to a complexity of $O(N^{\frac{D}{2}}) = O(\sqrt{N^D})$

5 Results

The main purpose of this section is to highlight what kind of practical improvement alpha-beta has over the base. For the comparisons see Figures 4 and 5.

6 Discussion

In the process of implementation, there were some issues regarding implementation and testing. Mainly corner cases where the bot would randomly blunder a one-move loss, this was caused by a conflict in implementation with some of the pre-coded template functions. Other than that the underwhelming result

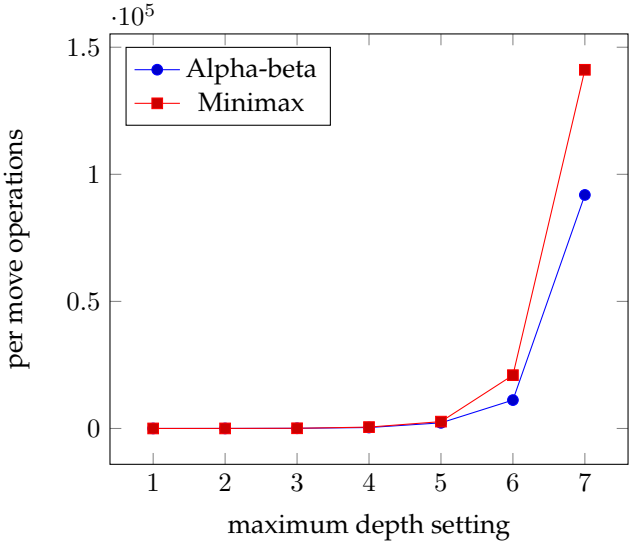

Figure 5: A similar analysis of minimax and alpha-beta this time using the simple heuristic from the github repository. Here the difference is much more noticeable at the highest tested value.

regarding the performance of alpha-beta pruning potentially caused by poor implementation.

7 Conclusion

From the analysis of both minimax and alpha-beta pruning we can conclude that combined they serve as a great baseline for a more complex algorithm and allow for easy testing of different heuristic functions allowing for a high ceiling for the performance of agents based on these models.

8 References

https://www.javatpoint.com/ai-alpha-beta-pruning [1]