Project 3

Jordan White

May 14th 2021

1 Overview

In this project, I implemented two game tree search algorithms: Minimax search and α - β pruning search. Since the game tree was too large to fully traverse, I also created an evaluation function for the search algorithms to execute at non-terminal nodes in the game-state tree.

1.1 The Minimax Function

My Minimax algorithm consists of two mutually recursive functions, called Max-value' and 'Min-value.' These function share the same general structure.

First, the Max/Min value functions check if the current game-state object is terminal or if the depth-search limit has been reached. If the state is terminal, they return the victory/defeat reward; If depth limit has been reached, they return the reward based on the evaluation of the current state.

After these checks occur, the functions create a new game-state object for each possible action at the current state, then they pass each of these game-state objects to the other function (max-value passes to min-value and vice-versa).

The Max-value function returns the action with the highest corresponding evaluation that is returned by the Min-value function. The Min-value function returns the action with the lowest corresponding evaluation that is returned by the Max-value function.

After the game-tree has been fully searched up to a certain depth, the max-value function returns the optimal action for the current player. I used depth 4 for efficiency purposes, but the depth can be changed.

It is important to note that, when the state-tree cannot be fully searched, the correctness of the moves in Minimax search is limited to how accurate the evaluation function is.

1.2 α - β Pruning Search

My α - β Pruning Search algorithm maintains much the same structure as my Minimax algorithm. It has two mutually recursive functions that return the action with the maximum or minimum evaluation value. The difference between α - β Pruning search and Minimax search is that my α - β Pruning algorithm only evaluates moves that are absolutely necessary to evaluate. It does this by maintaining two values, α and β, throughout the tree search. α represents the minimum score that the maximizing agent (the current player) is assured of, β represents the maximum value that the minimizing agent (the opponent) is assured of. In this way, α - β Pruning search ceases evaluation of a certain move when there is guaranteed to be a better move somewhere else in the tree.

1.3 Evaluation Function

In a two player perfect information game such as this one, a good evaluation function is a zero sum utility measurement system, in which all utility not awarded to one player is awarded to the other. This is how my evaluation function works.

My evaluation function first takes various measurements of the game-state, such as which player (if any) has the ball, the distance of the players to their respective goals, the distance of the players from each other, whether the current player is in-between the opponent and his goal, and whether either player is in the scoring/penalty zone. It then awards points to each player according to the True/False value of these features in a given game-state.

For example, the utility of player 1 increases by 5 points when he has the ball, while the utility of the other player decreases by 5 points. Once he has the ball, player 1's utility increases gradually the closer he gets to his goal, and decreases the closer the opponent gets to him. The opponent's utility would rise or fall accordingly. This is one small example of how my evaluation function awards utility based on features of the game-state; I tried my best to balance the utility rewards for each feature.

Each player, while running Minimax or α - β Pruning search, is trying to maximize his own utility (returned by the evaluation function) and minimize the opponent's utility.

2 Findings

My Minimax function and the max-value function are shown in Figure 1. The min-value function is not shown, but has a similar structure to the max-value function.

My α - β Pruning search algorithm and the α - β max-value function are shown in Figure 2. The α - β min-value function is not shown, but has a similar structure to the α - β max-value function.

Upon execution of my program, the game renders and both players head for the ball. If one of the players grabs the ball while the opponent is within one square, the opponent will collide with the player and steal the ball. If one of the players grabs the ball while the opponent is still very far (over 3 rows) away, that player easily wins. The interesting case is when one player reaches the ball while the opponent is close by, but not close enough to easily steal it.

In this case, when one player reaches the ball only a few squares before the opponent, the opponent resorts to blocking the player's path to his goal by securing a spot in the same row as the player, one column away. The opponent then chases the player to the other end of the grid (to the opponent's goal). I find this tactic interesting because it is similar to the tactic called 'Opposition' in chess, in which one King (say, the white King) blocks the black King's path by occupying the space immediately adjacent to the black King. In chess, the black King cannot legally advance because he is not allowed to put himself into check; In the soccer game, the players can legally advance, but do not because that would allow the opponent to steal the ball.

Usually, this chase results in the the player conceding the ball to the opponent by kicking it. The player calculates that this is preferable to walking into the opponent's goal and scoring a point for him. Interestingly, however, the player being chased is sometimes still able to win by means of a tactic in which, after being chased to the wrong end of the grid, the player enters the penalty box and then side steps and passes the opponent. In the future, I would like to analyze this winning tactic more to determine why it is the player is sometimes able to perform it, but sometimes only able to concede the ball to the opponent.

I find it fascinating that a mere five steps of look-ahead (which is the breadth of the game-tree I allowed the algorithms to search to) allows the soccer agents to develop semi-complex tactics which display what many would consider to be rational behavior.

3 Reference

```
def minimax(self, state, player_id, depth):
    (a, e) = self.maxval(state, player_id, depth)
    print(a)
    return a

def maxval(self, state, player_id, depth):

        if(state.is_terminal or depth <= 0):
            return (None, self.evaluate(state, player_id))

        move = None
        v1 = -np.inf

        for action in state.actions:

            next_state = state.act(action)
            if(not next_state == None):
                next_tup = self.minval(next_state,
                player_id, depth - 1)

                if(next_tup[1] > v1):
                    v1 = next_tup[1]
                    move = action
            else: print('next state none (max)')


        return (move, v1)
```

Figure 1: My Minimax algorithm (shown at the top) and the associated max-value function.

```
def minimax_with_ab_pruning(self, state, player_id, depth,
alpha=-float('inf'), beta=float('inf')):

    (action, evaluation) = self.ab_max_val(state, player_id,
    depth, alpha, beta)
    print(action)
    return action

def ab_max_val(self, state, player_id, depth, alpha, beta):
    if(state.is_terminal or depth <= 0):
        return (None, self.evaluate(state, player_id))
    move = None
    v1 = -np.inf
    for action in state.actions:
        next_state = state.act(action)
        if(not next_state == None):
            next_tup = self.ab_min_val(next_state,
            player_id, depth - 1, alpha, beta)

            if(next_tup[1] > v1):
                v1 = next_tup[1]
                move = action
                alpha = max(alpha, v1)
            if(v1>=beta):
                return (move, v1)
        else: print('next state none (max ab)')
    return (move, v1)
```

Figure 2: My α - β pruning search algorithm (shown at the top) and the associated max-value function.