Outline

1. Introduction
2. Preparation
3. Markov Decision Process
   1. State
   2. Action
   3. Reward
      1. -1 reward for each step
      2. +100 if won (terminal state)
      3. -100 if lost (terminal state)
      4. -10 if draw (terminal state)
   4. Equations
   5. Common mistakes
4. Development process
   1. Planning and designing
   2. User interface
   3. Programming language
   4. Testing
      1. Testing all method functions, accessor functions, mutator functions
5. Challenges and solutions
   1. Unforeseen obstacles
6. Figures and tables
7. Conclusion
   1. Functionality
   2. Future improvements
8. References

Markov Decision Process

MDP

The initial starting state of the MDP is an empty board with dimensions 9x9. The first move is decided by the agent using their policy. The state space is 81 possible state space for the first turn. The second turn has 80 state spaces. Continuing until terminal state has been reached. Terminal states result from either player pieces or enemy pieces consuming 5 consecutive squares in the grid, or all possible positions in the grid have been taken equivalently no possible actions are available. Therefore, the theoretical max limit of the state space is 81! = 5.7971e120. An empty space is denoted by integer 0. Player piece is int 1 and enemy is int 2.

All possible actions are the row and column positions of the board. The board is ordered row major and column minor. The action at row 0, column 0 is positioned at the top left corner of the board. Action 8, 8 corresponds to the bottom right position. The action space therefore is [(0,0), (0,1), … (8,7), (8,8)] with total 81 actions. If a position in the board is taken, the corresponding action will raise an error.

The reward for taking an action that does not lead to a terminal state is -1. The reward for taking an action that leads to a win is 100. The reward for a loss is -100. The reward for a draw is -10.

Equations

Minimax algorithm

Alpha beta algorithm

Planning design

Board class holds the game state, all available actions, whose current turn, number of pieces and board dimensions. All board variables and functions are public. Functions include placing a piece, switching turns, running the minimax and alpha beta algorithms. As well as checking terminal conditions such as win or draw. There are game functions specifically for rendering player versus player, player versus enemy, and enemy versus enemy games using pygame.

Interface

Interface is rendered using pygame library. Custom assets for background board and black and white pieces. The interface is enabled only for player versus player and player versus enemy game modes. The board state is rendered to the screen for player feedback. Place pieces by using mouse button left click on top of available empty squares.

Challenges and solutions

Checking the win condition was a matter of brute force and could’ve been implemented in a smarter and more efficient way. However, checking all possible positions still achieved the same effect. To start, the win condition can only happen after a player or enemy has placed a piece. The goal is to create a string of 5 of your pieces in a row, then the win condition only needed to check after 11 pieces have been placed on the board. After that, we need to check 5 squares on the board at a time and count the number of pieces summing to 5. Not only that, but we also need to check if all pieces are all from the player or all from the enemy. The permutations of a win condition can manifest itself horizontally, vertically, positive slope diagonally, and negative slope diagonally. If found, then we can return true. If not, then the win condition is false, therefore continue playing.

Implementing minimax algorithm took longer than expected. The pseudocode looks deceivingly easy until we are tasked with checking all conditions and recursively calling ourselves. The main control loop checks if the exit conditions have been met. The exit conditions include reaching maximum depth, if win condition is met or draw condition is met. If so, we evaluate the board position and return the heuristic score of the board according to whomever called the minimax algorithm. The tricky part is handling if player called the minimax or enemy called the minimax. Implementing the alpha beta pruning was unexpectedly easy. Simply pass alpha and beta into the minimax and check if alpha is greater or equal to beta, if so then prune the branch.

Scoring the board position or heuristic evaluation of the board position needed to be implemented. The basic structure of the heuristic evaluation is to score points based on the number of 4 in a row, 3 in a row, and 2 in a row. Not only do we need to evaluate our position, but also the enemy’s position. The sum of these two scores will define the heuristic evaluation of the board. Point distribution is not equally distributed on both sides. The enemy position score is more heavily weighted against the player position. Depending on whose turn is next and the current position of both sides, the highest priority is to block a move resulting in the enemy winning. Favoring defense more than offense to hopefully sustain the game and prolong the chances of winning.

User testing of score evaluation followed the idea of playing higher defense than offense. In testing, blocking an enemy’s pen ultimate move allowed for more favorable chances of an endgame. A pen ultimate move can be defined as placing a piece that results in 3 in a row with empty spaces on both sides. In this case, no matter what the player does, even if playing optimally or non-optimally, will result in a guaranteed win for the enemy in two moves. Preventing this from happening stops the enemy from winning, and allows the player to better improve the current position.