COMP 424 ARTIFICIAL INTELLIGENCE FINAL PROJECT REPORT

Aleksas Murauskas, 260718389

I. Introduction

This report entails the efforts to create an Artificial Intelligence to play the game Pentago Twist. It describes my implementation of a minimax algorithm with alpha beta pruning, as well as a heuristic function to guide the algorithm. The Theoretical Basis section contains a brief overview of my implementation and discuss how I applied knowledge I gained from class and Pentago strategy guides I found online. The Code description section contains an in depth of the Artificial Intelligence I have written, and how each portion operates and interacts with each other. The Advantages and Disadvantages portion contains the features of the system and how they improve or hamper performance. The Future Improvements portion contains improvements I would make to my system given more time, resources, and feedback.

II. Theoretical Basis

As learned in class, I have implemented the Minimax algorithm with alpha beta pruning. This algorithm will traverse a tree of possibly board states, and at each depth, either maximize the perceived value of the player's move, or minimize the perceived value of the player's move in order to find the optimal play. The alpha and beta values assigned to each node will speed up the processing time by ignoring branches that are not perceived as valuable enough to travel along. To guide my Minimax algorithm implementation, I devised a heuristic function that I felt best represented a player's ability to win a game of Pentago Twist. I studied the Pentago Twist game though multiple online guidebooks and videos and found there were 32 possible ways to win the game: Twelve Rows, twelve Columns, and eight diagonal lines. My heuristic program first looks for nodes in which loss or victory are imminent and selects moves that deny or secure victory in the fewest number of moves. In the case where a victory is not visible, then a heuristic value is assigned for each of the 32 lines that where a victory can be earned, and the move's value is increased by the more possibilities the player has to win, and fewer that the opponent has.

III. Code Description

The Artificial Intelligence code written is entirely held within the studentplayer package. The package holds the given stub classes studentPlayer, MyTools, and the additional of the Tree Node class. StudentPlayer only enacts a simple call to find the minimax and heuristic methods in the MyTools class in in order to retrieve the best move. The MyTools class contains four methods: getMinMaxedMove(), minimax-Algo(), and heurisitc evaluation(). TreeNodes act as boardstates that the minimax algorithm will travel through.

A. The ChooseMove Method

StudentPlayer contains the chooseMove() method, which is called by the autoplay and game in order. Choose move contains commented out timer for testing how long it takes for my AI to return the best possible move. For the first two moves of my player's game, I call the method opening-move() which returns a random move that controls the center of the board, as running my minimax algorithm on an empty board is too costly. After the first two moves, the player will call my getMinMaxedMove() method from the MyTools class. This will return the best move possible given the boardstate and time and depth constraints, the . As the game goes on longer, there will be fewer possible moves and my minimax algorithm will traverse my tree faster, so I can increase the depth. At 8 player turns, or halfway through the possible game my depth increases by one. At 14 Player turns, the depth increases to four to see all possible remaining endgame scenarios.

B. TreeNode Class

The TreeNode class represents a node in the move possibilities tree that the Minimax algorithm will operate on. Each node holds parent-node field that points to the previous TreeNode in the minimax tree, so it is easy to recurse up the decision tree. The state and move field hold the PentagoBoardState instance that represents the board state and PentagoMove instance that represents the move that transitioned the board from the parent board state to the one held by the current tree node. The arraylist field children holds a list of tree nodes that represent the board states that the board can be transitioned to in one legal move. The heuristic-val field holds an integer that represents the evaluation score determined by my heuristic function, this field is initialized at 0. TreeNode only contains two functions, addChildNode(), which adds the passed TreeNode to the children array list, and getparent() which returns the pointer held in the parent-node field.

C. The OpeningMove Method

The openingMove() method is called for the first few moves of the game. I found the computational load of operating the minimax algorithm took too long on the first initial moves. This method controls the center of the board by selecting a random move that operates on the 4 center pieces of the board.

D. The GetMinMaxedMove Method

The goal of the method getMinMaxedMove() is to return the best possible move that the Student player can make given a board state and a maximum depth. This acts as the first iteration of the minimax algorithm, and since this will always be called by the student player, it will always be maximizing. When getMinMaxedMove() is called for the first time it updates two fields that hold the color of the student player's pieces and that of the opponents pieces for use in my heuristic function. The method will call all the available moves the player can make, and for each move, create a TreeNode containing a board state with the move played on it. Additionally an alpha and a beta value are created at the minimum and maximum integer values, for use in alpha beta pruning. The heuristic value is determined by calling the minimax-algo() method, which recursively iterates until a maximum depth is reached. Once the recursive calls completes, the value is compared to the best heuristic value found so far. If it is a better move, it is added to a list called best-moves-available right after the list is cleared and the best heuristic value is updated. If there are multiple moves with the same heuristic value, one will be chosen from the list at random and returned.

E. The MinimaxAlgo Method

The minimax-algo() method operates recursively calling itself to find the optimal move. The first thing it does when called is determining whether its tree node is a leaf, which occurs if the board state contains a game over or the final depth has been found. If the node is a leaf, the heuristic function is run on the boardstate held in the node and returns it. In the case the node is not a leaf, the method creates a node for each available move and prepares to recursively call minimax-algo() on the new node. In the case where this is a call maximizing the player's heuristic value, the method will update the node's heuristic value with the greatest result of the recursive minimax calls, and updates the alpha with the same value. In the case where the call's alpha has exceeded the call's beta, for loop breaks and prunes the branch. In the case where this is a call minimizing the player's heuristic value, the method will update the node's heuristic value with the lowest result of the recursive minimax calls, and updates the beta with the same value. In the case where the call's alpha has exceeded the call's beta, for loop breaks and prunes the branch. The method will return it's own heuristic value if the node is a leaf, and return the heuristic value of it's best child otherwise.

F. The Heuristic-Evaluation Method

The heuristic-evaluation() enacts a series of tests in order to determine the value of taking a move. The method first checks if the board state contains a victory. In the case it does, it will check if the winner is the current player. If not, the heuristic value will be a negative ten thousand, to disincentive the move as much as possible. If it is a draw, then it will return a set draw value, a negative. If it is the maximizing player's victory, It is a positive subtracted by the number of terms it would take the player to earn the victory. This subtraction ensures that the player will go for the victory it can earn in fewer turns. Otherwise, the function will attempt to judge the board by assessing the amount state of all the possible winning lines. The heuristic value begins at two thousand and that value will rise or fall based upon an number of eval-line() method returns. For each column, row, or diagonal of five consecutive board positions the eval-line() method returns a heuristic value to adjust the base value by. The more the maximizing player controls a line that they can still use to win, the greater the addition to the heuristic function. The more the minimizing player controls a line they can still use to win, the greater the subtraction from the heuristic value. The heuristic value for a loss or a victory is intended to be high enough that a victory or loss will always trump the heuristic value found from my line evaluations. Originally, I had this function evaluate all 32 possible lines of 5 that a player could win in, however, to streamline my heuristic function I reduced the number of lines that are evaluated to better fit within the time constraint. For example, the first row can form two winning lines, 1 to 5 and 2 to 6. Since these lines overlap for 4 pieces, I don't need to evaluate both lines. In doing this I achieve a heuristic value close to my full 32 line system, while cutting the eval-line() calls nearly in half. The heuristic evaluation value is then returned to the minimax algorithm.

G. The Eval-Line Method

The eval-line() is a helper method to the heuristic function. It returns a value that adjusts the heuristic value based upon the perceived value of that line in both player's games. It receives a list of piece values which can be white, black or empty. It first counts the number of blanks left in the line. If there are no pieces in the line or the line contains a mix of player pieces, it returns 0. In the case where the player controls the line, the method returns a positive value that increases the longer the line is and the closer to victory. In the case where the opponent controls the line, the method returns a negative value that decrease the longer the line is and the closer to victory. The values were determined through trial-and-error testing. To incentivize building a player's own line rather than block the opponent, the positive value of a move has a greater

magnitude than the negative value of the opponent making the same move.

IV. Perceived Advantages and Disadvantages

A. Advantages

I believe most students will be implementing a similar system in terms of use of a heuristic function and a minimax algorithm. I found that my algorithm would not be able to find the best move within two seconds giving my algorithmic structure and heuristic function on depths greater than two. Therefore, I improved my heuristic structure to operate minimax more accurately on smaller depths without significantly increasing computation time. In the early moves of the game, there are so many possibilities that operating on depths large than 2 or 3 is not possible within the time constraints. Compared to a to a greedy heuristic function that is binary on victory and loss on that board state, my heuristic method encourages moves that both lead the player to victory and block the opponents moves. Additionally, by subtracting the turn number from the value of victory, it encourages the Minimax algorithm to select a victory in the smallest number of turns. I have set the Minimax algorithm to increase the depth as the number of turns passed rises. This is because the number of nodes in the decision tree will decrease exponentially with turns, reducing the computational load on the algorithm. Increasing depth will allow my system to make better decisions, and since the game is nearly over the number of recursions is significantly reduced, there is little change to the overall processing time.

B. Disadvantages

As both an advantage and disadvantage is the inherent assumption of minimax, that the opponent is using the same heuristic function to judge their actions. This will impact all students who use this algorithm to guide their player, and because of this both players' will make decisions that seem irrational to their opponent if their heuristic functions are not the same. Unfortunately, through testing my algorithm would take too long to make the first two moves, so I developed a second method to set the first two pieces for my player. It selects a random move that enacts on the central four board placements. However, this means my first two moves are not the best move found in the given timeframe and therefore could set the player up for loss later in the game. To stay within the strict time limit of 2 seconds a move, I was forced to make a greedy decision once enough time had elapsed. Because of this, I implemented a timer that will terminate the GetMinMaxedMove() method to terminate early in the case the search was nearing the two second deadline. The method then greedily returns the best move found so far. While not a true greedy algorithm, having to do so under the time constraint no longer guarantees the best move given the current board state. To further reduce the computation time, I reduced the complexity of my heuristic function by reducing the lines evaluated. Originally, I had the function heuristic-evaluation evaluate all 32 possible lines of 5 that a player could win in, however, to streamline my heuristic function I reduced the number of lines that are evaluated to better fit within the time constraint.

V. Future Improvements

The primary issue I had while developing my minimax algorithm was not finding the best move but finding it under the time constraint. In order to do this, I would revise both my minimax and my heuristic function to reduce computation on each recursion of my minimax algorithm. Given time and feedback, I would improve I would spend more time fine tuning my heuristic function. My current heuristic function was improved by trial-and-error fine tuning and taking time to either further improve the static values or further refine the algorithm to find a more appropriate formula than the line completion evaluation I currently have. Improving my heuristic function for the early and midgame is the best way to improve my system's performance. My current heuristic method of checking lines is slow and finding a way to identify the best move more quickly and accurately would be ideal. If this improvement is efficient enough, I would no longer need to use my opening method, and can fully rely on my minimax method to operate on every move in the game. During my time studying strategies for Pentago Twist I found some boards that could be considered win conditions, but I was unable to implement these into my heuristic function. If I need to keep my minimax depth shallow, this would enable the algorithm to find a victory one iteration earlier than expected.