Pentago-Swap: Alpha-Beta Pruning Agent

Sadhvi Mehta
260747607

April 11, 2019

1 INTRODUCTION

Two-person, perfect information, deterministic games are a well-researched topic in Artificial Intelligence. This paper aims to explore game theory as a search problem for the game 'Pentago-Swap'. This is a two-player game with the objective of placing 5 pieces in a row on a 6x6 board. The algorithm of focus is Alpha-Beta Pruning, an optimization of the Minimax Algorithm. The paper discusses the motivation behind the chosen algorithm, its implementation, including the formulation of the utility function and chosen depth, as well as future improvements. The end goal of this project was to achieve a 100% win rate against randomly generated moves.

2 EXPLANATION OF PROGRAM AND MOTIVATION

The implemented AI agent performs the Alpha-Beta Pruning algorithm to determine the next best move. Naturally, it assumes that it is the max player, and its opponent is the min player.

Upon the agent's turn, the program calls the *pickMove()* method. This method encompasses the setup of the algorithm as well as the actual call to the method implementing the alpha-beta pruning. The setup for the algorithm includes defining the max depth to be explored and defining the player ID of the bot. Following this, a set of four preliminary moves are attempted: placing a piece at the center of any of the 4 quadrants. The reasoning for this is given in the 'Theoretical Basis' section.

The code implementing Alpha-Beta Pruning consists of two methods: *maxValue()* and *minValue()*. Both methods iterate through all legal moves, processing each move on a 'cloned' board, and recursively passing this cloned board to each of their counterparts for further processing. There are two breaking conditions for the recursive calls:

- Max depth is reached.
- Alpha is greater than Beta.

If maximum depth is reached, a utility function is called to evaluate the value of move. This value is then returned to the recursive caller and compared against the current best move. Similarly, if the updating of the best move leads to the value of alpha being greater than beta, no further moves are evaluated for the current recursive call and the best move is returned to the caller function. It is also important to note that a nested Java class was implemented to keep track of the best move. It consisted of the move itself as well as its generated value.

The motivation behind choosing Alpha-Beta Pruning was primarily because this algorithm is known to play well in deterministic, perfect information, two-player games. In addition, due to the assigned timeout, the pruning in Alpha-Beta would help meet the timeout constraint better than Minimax. It was also favored over Monte-Carlo Tree Search due to lack of time for learning and performing rollouts required for MCTS.

3 Theoretical Basis

3.1 Heuristic

In Alpha-Beta Pruning, the heuristic function is the critical decider in the performance of the agent. The implemented heuristic decided on how many pieces one had in a line as well as the possibility of this line to grow into '5-in-a-row'. Grow into '5-in-a-row' is explained using the example below.

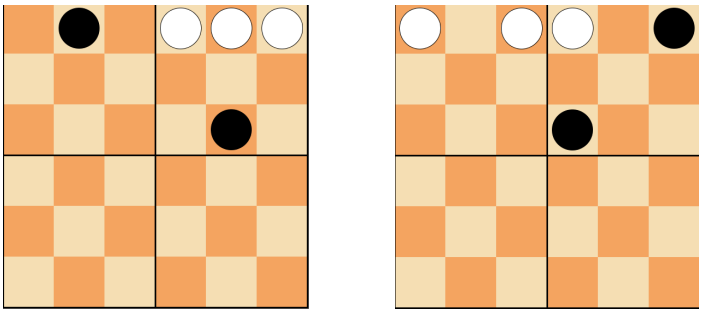

Figure 3.1: Heuristic Function Evaluation Example

In Figure 3.1, the grid on the left would receive a score of zero for Player 0. This is because although there are 3 white pieces in a row, the black piece is prohibiting those 3 in a row to

grow into 5 in a row with the current board arrangement. On the other hand, the grid on the right has 3 white pieces that are aligned with 2 empty cells that could potentially have white pieces in the future. Thus, the possibility of having 5 pieces in a row is still there, causing for the utility function to assign a positive value for this move.

This 5-in-a-row check is repeated for all rows, columns, and diagonals consisting of 3 and 4 pieces. In the case of 4 pieces, a value of 8 is assigned. In the case of 3 pieces, a value of 5 is assigned. In the case of 5 pieces, the greatest/lowest possible value of a double in java is assigned, depending on if it is the max or min player. Else, a value of zero is assigned. Moreover, these assigned values for each case were obtained by trial and error. Too large of a difference between each case would lead to a possible timeout. Thus, it was important to strike a balance such that the difference in weights is large enough to choose the optimal move without causing a timeout.

In this way, the heuristic would calculate the difference between the cost of the max player and the min player for every move and bubbles up this value. This is useful because it takes into account both the player and the opponent for each move.

3.2 Initial Moves

Another modification made was to always pick the middle cell of a quadrant if it is available. This move is known to be a good strategy in Pentago-Swap and allows your piece to be open to as many possible winning combinations versus any other cell. Thus, instead of the agent doing unneccessary computation, if a middle cell is available, it should place the piece there.

3.3 Max Depth

The max depth was also a crucial variable for the performance of the agent. It was limited by the timeout of the game as well as the complexity of the utility function. Through testing, a max depth of 2 was decided on.

3.4 Other Methods

Two variations of the player were developed before the final one. The first implemented agent followed the Minimax algorithm. It did not do any pruning and thus often timed out. This then led to the implementation of Alpha-Beta Pruning. The initial implementation of this agent had a slightly different heuristic. Instead of assigning the first four moves to the middle of quadrants, it relied solely on the algorithm itself to find the best move. In addition, within the heuristic, it gave additional points to board arrangements that resulted in a piece being placed in the middle of a quadrant. However, this heuristic often led to a set of poor initial moves as the random player would at times defeat the agent within the first 7 moves of the game. Consequently, this led to the assignment of the first initial moves to the quarant center if possible and the modified heuristic.

4 Pros/Cons

This approach has a strong utility function. The heuristic takes into account not only the current arrangement but foresees if this arrangement can grow into a future win. In this way, it does not trivially assign points to moves that will hold no value one level deeper. However, a significant con in this implementation is the high time complexity when calculating values of moves. Currently, the depth explored is only two. This might be due to inefficient coding or just a computation heavy heuristic. In this way, performance is significantly lower than it could be if the calculation time was not so large.

5 Future Improvements

A future improvement would definitely be to try and improve the time complexity in order to search a larger depth. This was attempted by setting a larger depth of 3 once a certain number of turns had occurred. However, this still led to the occasional timeout. It is still important to note that for the games where it did not timeout, the agent was able to beat agents of other students to who it was previously losing against. Hence, the next suggestion would be to implement a timer that causes for the agent to return the current best move if the time runs out before the algorithm is complete. Another possibility for exploring a larger depth could be to find a more compact way of passing the state of a board to recursive call. This would save time, thus allowing for a deeper search.

6 Conclusion

In conclusion, the Alpha-Beta Pruning agent beat the random player with a 100% win rate. However, the agent could be further improved to beat smarter opponents by optimizing the code to accommodate a larger depth.