COMP 424 - Artificial Intelligence
Final Project Writeup

Matthew Lesko-Krleza
260692352

April 11, 2019

Contents

1 Design and Motivation **1**

1.1 Depth and Timeout . . . 1

1.2 Continuous-Piece Utility Function . . . 1

1.3 Win-in-One Heuristic . . . 2

2 Theoretical Basis of Alpha-Beta Pruning **2**

3 Advantages and Disadvantages **3**

3.1 Advantages . . . 3

3.2 Disadvantages . . . 3

4 Alternatives and Results **3**

5 Future Improvements **4**

1 Design and Motivation

The core game is entirely implemented by the head TA. This report assumes that the reader is familiar with the Pentago Swap game and with its implementation by the head TA. The design implemented for the AI StudentPlayer is an Alpha-Beta-Pruning Search technique that uses a Continuous-Piece scoring function, and a simple Win-in-One heuristic function. Alpha-Beta Pruning is a search algorithm that prunes off nodes evaluated by the Minimax algorithm in a search tree [1]. The search algorithm recursively processes each legal move for each respective game state, computes a score for each game state, and terminates when either the maximum depth or timeout is reached. My alternative implementation is that of a Monte-Carlo-Tree Search technique that also uses the same Win-in-One heuristic function. I'll focus on the design of the AB-Pruning technique.

1.1 Depth and Timeout

My implementation of the AB-Pruning algorithm searches future game states up to a depth of 3 or until a two second timeout is reached. In either case upon terminating the search, the best move discovered so far is chosen to be played for the current game state. The best move is determined either by having a largest Alpha or lowest Beta value, depending on whether the current player is the White or Black player respectively. I've selected a maximum depth of 3 because I've noticed through testing that anything larger than that would surpass the maximum time limit allocated for a move (2 seconds) other than the starting move. The threshold of a two second timeout is also added so that the agent can reliably make a move within 2 seconds and not get penalized or disqualified.

When the Alpha value is greater than the Beta value within a given branch, the algorithm doesn't consider the other possible values within that branch, because it would guarantee a minimum score of Alpha. This saves on computation time.

1.2 Continuous-Piece Utility Function

The utility function determines the score of a state by the existence of a piece in a certain coordinate, and by the number of continuous pieces. A code snippet is shown to better detail the function:

```
if (piece == PentagoBoardState.Piece.WHITE && neighbor ==
   PentagoBoardState.Piece.WHITE) {
                         stateScore += Math.pow(2, horizontalPieceCount);
                         horizontalPieceCount++;
```

If two neighboring pieces are of the same color, then the score increases (decreases if it's an opponent piece). Depending on how many pieces are in a continuous sequence, the score added (or removed) may be larger. A line with 3 pieces would add a score of 2^3, since it's more valuable than a line with 2 pieces with score 2^2. I chose this approach because then my AI agent would rather attempt to make long continuous lines instead of multiple disconnected and short lines. I compute this function for horizontal, vertical, forward-diagonal, and reverse-diagonal lines. Furthermore, if the player has a winning state it receives a maximum score, inversely, if the state is a winning state for the opponent it receives a minimum score.

```
if (currentPentagoBoardState.getWinner() == currentPlayer) {
                stateScore = Integer.MAX_VALUE;
            }
            else {
                stateScore = Integer.MIN_VALUE;
            }
```

I've done this because then there are max and min bounds for winning and losing. Also, if a state has an incredibly low score, the agent will attempt to avoid it or block it.

1.3 Win-in-One Heuristic

The Win-in-One heuristic simply loops through all possible moves for the given state, and plays the move that would yield the current player a win. This heuristic is computed before the search algorithm is run. It is also used for the Monte Carlo Tree Search. I use it because it would guarantee me a win if there currently exists one.

2 Theoretical Basis of Alpha-Beta Pruning

To first understand Alpha-Beta Pruning, we need to understand Minimax. The Minimax algorithm evaluates the score of leaf nodes by the use of a utility function. When leaf scores are evaluated, the parent node takes on the score of either the largest or smallest score from its leaves, depending on whether it's a Maximizing or Minimizing play respectively. Assuming I'm the White player, I would like to Maximize my score, and I can assume that my opponent attempts to Minimize my score, therefore I would be the Maximizing player, and the opponent would be the Minimizing player. When leaf scores are evaluated, the parent node takes on the score of either the largest or smallest score from its leaves, depending on whether it's a Maximizing or Minimizing play respectively. This can be extremely computationally expensive. Given a branching factor b and a depth d, a full tree search would require a time complexity of $O(b^d)$.

The benefit of AlphaBeta Pruning over Minimax is that it prunes off branches that are deemed unreachable, hence saving on time. This saving could help my agent explore the game state tree of a larger depth, and effectively more moves ahead. The reasoning behind this is as follows [1]:

- If the Maximizing-Player finds a move whose value is greater than or equal to the value of an alternate Minimizing-Player move found higher in the tree, the Max-Player should not look further because the Min-Player would certainly take the alternate move;
- Conversely, if the Minimizing-Player finds a move whose value is less than or equal to the value of an alternate Max-Player move found higher in the tree, the Min-Player should not look further because the Max-Player would certainly take that alternate move;

The algorithm keeps track of the values: Alpha and Beta, which constitute the minimum score that the maximizing participant is assured of and the maximum score that the minimizing participant is confident in making respectively. Alpha and Beta are each initialized with their worst possible values, negative infinity and positive infinity (Integer.MIN, Integer.MAX in Java). The algorithm starts by evaluating the score of leaf nodes. A parent node's score takes on either the

Maximum/Minimum score from its leaf nodes depending on whether the Maximizing/Minimizing Player is playing that turn. Alpha and Beta values are updated at each parent node score update.

$$Alpha = max(Alpha, current\_parent\_score), Beta = min(Beta, current\_parent\_score)$$

. When the Maximizing-Player is assured of making better moves than the Minimizing-Player along one branch (Alpha ¿= Beta), then the children nodes within that branch can be pruned.

If searching is done optimally, where the best Maximizing-Player moves are always searched first, then the number of leaves evaluated is roughly $O(b * 1 * b * 1 * ... * b)$ for odd depth and $O(b * 1 * b * 1 * ... * 1)$ for even depth. In other words, in an optimal scenario, the time complexity would be $O(b^{d/2}) = O(\sqrt{b^d})$. On average, this is faster than Minimax by an exponential factor $d/2$.

3 Advantages and Disadvantages

3.1 Advantages

The Advantage of Alpha-Beta Pruning over Monte Carlo Tree Search (my alternative approach) is that I can develop an expert system and heuristic utility function that could decide which moves are best to make in the future. The improvement over naive Minimax is that I can prune off branches, hence saving on complexity time. I could either use this saving to be able to work within a design constraint (low memory and time resources) or use it to lengthen the depth of my game tree and compute more moves ahead of time.

3.2 Disadvantages

The major disadvantage of the algorithm is that I'm assuming that my opponent is also using the same Alpha-Beta-Pruning strategy as me. There may be a promising state I'd like to reach if the other player is also using Alpha-Beta-Pruning but I can't guarantee that I'll reach that state if the other player is using a different strategy. A weakness due to my implementation is that since my heuristic first searches for horizontal lines, it has a bias towards making them (as I've seen through testing). So if my opponent is strong at blocking horizontal lines, I can be defeated easily. Another weakness is that I recompute a lot of the same game states throughout different matches. I could store data from previous matches and use it to help compute scores for game matches much quicker. Another weakness I've discovered in my implementation is that I store and use a *PentagoBoardState* instance at each node within the Alpha-Beta-Pruning tree. I've seen that this class is extremely inefficient. I'll go further in detail for this in the improvements section.

4 Alternatives and Results

I also implemented a Monte Carlo Tree Search strategy. This strategy selects the most promising node from a game tree using an Upper Confidence Tree Formula [2], expands on that promising node, runs random simulations from the new children and back-propagates newly computed scores from the simulations. Finally, it picks a root node with the highest score. MCTS evaluates the score of a node using the UCT formula [2], which depends on a utility function (whether the current player or opponent wins) and the number of times a state is visited during algorithm run.

I implemented this strategy and ran it against my AB-Pruning strategy, both of them without the Win-in-One heuristic. I wanted to see how the base algorithms would compete against one another. I ran one experiment three times: run Autoplay for 10 games with both agents. The results can be seen within 1. They both win a similar number of games, and draws happened frequently (3/10 times in this case). I ran the experiment again with both agents using the Win-in-One heuristic. In that scenario, each agent won 5 times. When playing against the *RandomPentagoPlayer*, each agent won 10/10 games.

Experiment	MCTS Wins	AB-Pruning Wins	Draws
1	3	4	3
2	4	3	3
3	3	4	3

Table 1: MCTS vs AB-Pruning Over 10 Games

I decided to use AB-Pruning instead of MCTS because I belies it has more potential to perform better than MCTS. Given a better utility function (block opponent moves, or create traps), I believe AB-Pruning would be able to beat MCTS. AB-Pruning is also more humanly interpretable, MCTS runs random simulations and decides a move based on how well the simulations perform, whereas my AB-Pruning implementation directly attempts to make continuous lines.

5 Future Improvements

One improvement that I would like to consider is to reimplement the *PentagoBoardState* class. Instead of keeping a Piece (WHITE, BLACK, NOBODY) at each position, either null, 0 or 1 can be stored at a location on the board. I would remove the UnaryOperators as well. That way we save on memory. Another optimization strategy I could implement for my AB-Pruning and MCTS implementations is to only store the Pentago Move in a node within the search tree. The search tree would contain an initial board state. Any time I need information about the board state at a given node, I recursively recreate the board from the moves within the path to my current node. That way I store a move instead of the entire board state at every node, hence I would save on memory and be able to perform more computation. This would allow me to consider more future states and make better plays. Another improvement on memory could be to reimplement my AB-Pruning search in a lower level or faster language instead of Java, and have the *chooseMove* method interface with my algorithm implemented in the faster language. The possible languages included, and not limited to are: C, C++, Rust, and GoLang. since these languages would consume less memory and/or search future states quicker within the two second allocation for a move, then my agent could discover and consider more future states, and make better informed choices.

I would also like to improve my utility function. As of now, there's a bias towards making horizontal lines. I could implement a utility function that passes a 2x2 kernel over the board. The algorithm would attempt to discover any continuous lines (horizontal, vertical, or diagonal) within the 2x2 kernel. That way there wouldn't be any bias towards any specific line direction, enabling my agent to make more generalizable plays. Finally I could store data from previous matches and implement a data-enabled utility function that would make better informed score predictions on states. These improvements consider all the disadvantages previously described except the one that assumes that the other player is also using AB-Pruning, which is by design of the algorithm.

I would also implement a better Win-in-One heuristic. As opposed to only considering the next move that guarantees me a win, I could implement an algorithm that would make it consider moves that would allow my opponent to win. If it hasn't discovered a state that guarantees me a win within one move but it discovers a next state that would allow my opponent to win, it would block that move right away.

References

[1] D. Edwards and T. Hart, "The alpha-beta heuristic," 1961.

[2] "The upper confidence bound algorithm," 2016.