COMP 424 - Artificial Intelligence
Final Project

Manuel Salomon
260851102

April 13, 2021

1 Approach and motivation

Pentago-Twist falls into the Moku family of games. Other well known games in this category include the classical Tic-Tac-Toe and others like Connect-4. These are all played in a board and share one thing, they are perfect information games. Agents act sequentially and get to fully observe the state of the world before deciding what to do. With each decision, agents seek to maximize their utility.

Knowing this, I decided to take a deterministic approach. For every turn, the agent approximates the state space and the game-tree, from there, it draws assumptions and decides what to play next.

Notice how the term 'approximate' was used. This is because it is impossible to consider the whole space state as well as the game-tree and pick a move in less than two seconds, which is exactly the time our agent gets to operate. A little bit of math shows that:

The naive way of computing the approximate state space is to realize that the board is 6×6 cells. Each cell has three options: empty, white or black

$$3^{36} = 1.5 \cdot 10^{17} \text{ states}$$

But this is not the accurate figure, since it counts things like a board full of white pieces as a valid state, even though such state is clearly impossible. The actual number is approximated in the paper On Solving Pentago (click to visit), published on 2011:

$$2.4 \cdot 10^{16} \text{ states}$$

This is obviously still very large.

As mentioned before, with each decision agents seek to maximize their utility. Thus the Minimax algorithm, further optimized with Alpha-Beta pruning was chosen for this agent.

2 Technical analysis

Minimax is a recursive or backtracking algorithm used in decision making. It returns the optimal move to play assuming the opponent plays optimally too.

To find the optimal move, it performs a depth-first search on the game tree and evaluates its leaf nodes with a utility function. The parent nodes then take on either the largest or the smallest score from its children, depending on whether we are trying to maximize or minimize the score. The current implementation maximizes the score of our agent and minimizes the opponent's score.

This can be very computationally expensive given a big branching factor, in other words, if there are a lot of different options to play. Consider a game with a branching factor of b and a depth of d, a full tree search requires a time complexity of $O(b^d)$.

This is very large, that is why we have to carefully limit the depth. To respect the two second constraint, it was determined through trial and error, that a depth of three is the maximum our agent could afford in the early stages of the game. As the game progresses, the number of valid moves left to play decreases and so does the size of the game tree. This gives us the opportunity to increase the depth a little bit and thus increase the forward-planning capacity of our agent. Again, through trial and error, it was determined that after the 10^{th} turn depth could be increased to four, and after the 15^{th} turn, to five.

2.1 Utility function

In order to evaluate the leaf nodes, a utility function has to be specified. The goal of Pentago is in fact very simple, we just want to align five pieces. Naturally, a utility function that rewards the agent for aligning pieces comes to mind. But remember, the same way this function rewards our agent, it must penalize it for allowing the opponent to align pieces too.

There are three different ways to align pieces: horizontally, vertically and diagonally. Our agent takes all three into account following this logic:

```
if (currentPiece == neighbourPiece == WHITE)
        score += (numberOfPiecesAligned << 2) * w
else if (currentPiece == neighbourPiece == BLACK)
        score += (numberOfPiecesAligned << 2) * b
```

Where w and b are simply integers thaht take on the values 1 or -1 depending on whether our agent places white or black pieces.

The implementation loops through the entire board two times. The first time it counts the pieces aligned horizontally and vertically, and the second time the pieces aligned diagonally.

It is clear that this function rewards and penalizes our agent equally depending on the overall number of pieces aligned. Naturally, as the number of pieces

aligned grows, the reward and the penalty grow too. For example, two pieces aligned add a value of 2^3, whereas three pieces aligned add in 2^4 to the score. Four aligned pieces contribute 2^5 to the value.

Five pieces aligned directly translate to victory or defeat, hence a score of Integer.MAX_VALUE is returned if our agent aligns such number of pieces. Integer.MIN_VALUE is returned if the opponent does it.

In case of draw, 0 is returned. This implies that if we are currently 'winning' (winning defined here as having more pieces aligned than our opponent, even though this may not translate directly into victory), the agent is given a chance to continue the game and try to win. Same logic applies conversely to the opponent.

2.2 Optimization

As mentioned before, the Minimax algorithm is optimized using Alpha-Beta pruning. What pruning does is, it discards branches of the game tree that are considered unrecheable, hence saving computing time and spending it in evaluating more and deeper moves.

The reasoning behind pruning is as follows:

- If for the Maximimzing player, a value greater than or equal to the value of an alternate Minimizing move is found higher in the tree, do not look further in that branch since it is assumed that the Minimizing player will take the alternate move.

- Same logic applied conversely to the Minimizing player.

Here is the pseudocode from Wikipedia that was implemented:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value
```

By the same Wikipedia article, this effectively reduces the complexity from $O(b^d)$ to $O(\sqrt{b^d})$, on average, an exponential factor of $d/2$.

3 Advantages and disadvantages

3.1 Advantages

The biggest advantage of Alpha - Beta Pruning over other possible implementations such as Monte Carlo Tree Search is that a heuristic utility function can be developed based on knowledge of the game, thus deciding which moves are worth making and which are not. Furthermore, this utility function can be upgraded and refined in the future as more knowledge is acquired, thus making it more and more sophisticated.

The advantages over standard Minimax have already been mentioned: entire branches get pruned off, which leads to less memory and time being used, or more and depper moves being evaluated, or a mix of both.

3.2 Disadvantages

The biggest disadvantage with this implementation is that since it only considers a move superior to another move if its value is higher, aligning pieces horizontally clearly dominates the other startegies such as aligning vertically or diagonally. This is because horizontal moves are computed and evaluated before the rest.

Moreover, this applies to all Minimax implementations, it is assumed that the opponent plays optimally with respect to my logic. If he does not, my agent is not guaranteed to reach the state he planned on reaching a couple of moves before.

4 Future improvements

In my utility function I used bit shifting to compute the powers of 2. It certainly makes little to no difference when evaluating the game tree (compared to using the function Math.pow()), but it is there to represent the power of bitwise operations, which could be implement in the *PentagoBoardState* class. This class uses a 2D array of Pieces to represent the board and quadrants. It could instead use two 2D arrays, one for white pieces and the other for black pieces, made of 0s and 1s to represent occupied and free respectively. Moves would then be processed with bitmasks and some bitwise operations, thus saving a lot of memory and time. As a result, more and deeper moves could be evaluted and performance would considerably increase.

Another possible way of saving time could be using low level language implementations such as C or C++.

Finally, unbiasing the agent towards horizontal lines and making it diversify its moves could also help increase performance.

References

- "On Solving Pentago", Niklas Büscher, 2011 (*link*)
- "Alpha–beta pruning", Wikipedia (*link*)