COMP 424 - Final Project Report

Nicolas Fertout - 260826282

April 13, 2021

Overview

Throughout this report, I will provide an analysis of my code and ideas during the course of the project. We will first go over the motivation behind my implementation. Secondly, some of the main components and ideas will be detailed, both conceptually and technically. Furthermore, the key advantages and disadvantages will be described. Finally, we will go through the major possible improvements and further ideas that could enhance the performance of the algorithm.

1 Motivation

Since Pentago-Twist is a 2-player game, fully observable and deterministic, the first (obvious) idea is an implementation of the MiniMax Search algorithm. MiniMax Search uses an evaluation function to compute the utility of a given board state, and then chooses the move that can result in the highest possible utility value, assuming that the opponent is choosing the moves that can result in the lowest possible utility values (using the same evaluation function; more on this in the Advantages and Disadvantages section).

However, as seen many times in class (see Lecture 9 - Games), and as emphasized by the Project Details, the large number of possible moves reflects a high branching complexity, which impacts the time complexity of $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree (in this case, $m = 36$ on turn 1, since there are at most 36 turns in a game). I thus decided to implement an α-β pruning algorithm, which is an improved MiniMax Search, where the main difference is the pruning of paths (in the search tree) that appear to be worse than what we can already achieve. We will now go over this approach in more details.

2 Technical Approach

2.1 Theoretical basis of the approach:

The key idea of the α-β pruning algorithm is the assignment of an α-value and a β-value to every node, from bottom to top, until we reach the root node (i.e., the board state we are currently at). As for the regular MiniMax, we alternate between Max nodes (that represent our player), and Min nodes (that represent the opponent). At Max nodes, we only update the value of α, which serves as a 'lower bound' for the utility, and at Min nodes we only update the value of β, which serves as an 'upper bound' for the utility. Pruning happens whenever an α-value is greater than or equal to a β-value for a given node. By reducing the number of nodes visited, we can lower the time complexity from $O(b^m)$ down to $O(b^{m/2})$ in the optimal case, allowing the agent to double the search depth (See Lecture 9 - Games).

However, since we cannot afford to only evaluate leaves (i.e., states where there is a winner or a draw), we need to design a function to help us evaluate any intermediate state. Due to the complexity of the game, this was one of my main focuses during this project.

2.2 How the program works:

The program consists of 3 main components: the caller function, the α-β pruning algorithm and the evaluation function.

2.2.1 The Caller Function: `chooseMove(PentagoBoardState)`

This function is in charge of calling the α-β pruning algorithm, as well as some basic tests and variable declarations. Some of the latter are `startTime`, which will help to keep track of how much time we have left, and `myColor`, which indicates the color of our player ($0 = White$; $1 = Black$).

Before calling the α-β pruning, it checks if there is a legal move that can win this turn, and if so, it returns this move. This prevents the algorithm for running for 2 additional seconds.

It is also within this function that the depth limit is set. From many tests, it seems that a depth 3 in reasonable for a 2-seconds move. Note that depth limit is set to 2 for the very first moves (where the depth won't affect the game in many aspects) and last moves (where there are only 1 or 2 available slot(s) left).

We then call the α-β pruning function, and return the obtained move. If the move returned is null (which only happens when every move result in a loss on the very next turn), we then use a last-second procedure that returns the move that leads to the highest utility at depth 1.

2.2.2 The α-β Pruning Algorithm: `getMoveWithabPruning(PentagoBoardState, int, int, long)`

It is within this function that the structure of the program is implemented. As explained in the theoretical approach, we alternate between `maxValue` and `minValue`, until we either run out of time, or reach the depth limit. These sub-functions are responsible for updating the α and β at every node (using a data structure called `AlphaBetaPBS`, which contains a `PentagoBoardState`, as well as 2 `int` fields for α and β).

Those values are accessed by `getMoveWithabPruning`. The latter will then iterate through the list of all children board states, until one of the child's β is equal to the root node's α (implying that this node corresponds to the obtained value of α).

As an insurance check (and mostly because my evaluation function is not perfect), before choosing a move, we check if choosing this move could result in a loss, and if so, we eliminate it and choose another one. If every single move can lead to the opponent winning on the very next turn, we then return null back to the caller (see The Caller Function).

2.2.3 The Evaluation Function: `evaluationFunction(PentagoBoardState, int)`

This function is where most of the design choices were made. While in chess a given board state can (most of the time) be roughly evaluated from the number of pieces remaining for each player, in Pentago-Twist, even the simplest evaluation involves many different factors!

I started off with a basic function that returns 1 if our player won, -1 if the opponent won, and 0 otherwise. However, this choice was not accurate enough, since the algorithm will end up making an actual decision only when it is losing or winning in the next few turns. The rest of the time, it will behave in a quasi-random manner.

From the latter observation, I then wrote a second evaluation function, that was iterating through the board, and returned the maximum number of pieces aligned by our player. After realizing that it was not taking into account my opponent's performance at all, the next improvement was to evaluate my utility, and then subtract the utility of my opponent. A neutral board would then be evaluated to 0, and a favorable board should be given a positive value.

Though the above function did perform relatively well, my algorithm was only considering its longest alignment of pieces, omitting that once the opponent blocks it, the next few moves will be chosen almost randomly (since no more improvement can be made). Therefore, I decided to add some complexity to the evaluation function. Here is a complete overview of how the final version operates:

1. Consider a sub-function called `evaluateMe1(PentagoBoardState boardState, int Color)`. For a board state, this function computes the maximum number of aligned pieces of the given color horizontally, vertically, and on the two big diagonals.

2. If any of those values is greater than or equal to 5 (implying that the given color won), the function returns 1000, without any further computations.

3. Moreover, if some very favorable patterns (specified by a helper function called `additionalChecks(PentagoBoardState, int)`) are recognized, the function returns 70 (or -70, depending on the color). Those patterns facilitate the set up of 'mate in 2' moves, which often result in a win within the next 2-3 turns (See examples of favorable patterns in Figure 1 and Figure 2).

4. In any other cases, the sub-function returns the sum of the factorial of all 4 values computed in step 1 (so that an alignment of 4 has a higher weight that 3 alignments of 2 for instance). However, since the main diagonals seem to give a lesser advantage than the horizontal and vertical alignments, their factorial values are divided by 2. This precision will usually make our player choose a move that increases its horizontal/vertical performance over its diagonal.

5. Finally, the evaluation function will call the above helper function on our color, and then subtract it by the result of the helper called on the opponent's color.

Note that calling the helper evaluation function on both the opponent and our player allows us to create different behaviors for our bot! For instance, adding a smaller (i.e., more negative) coefficient on the opponent's values will result in a more defensive gameplay, whereas increasing the coefficient on our player's values will result in a more aggressive gameplay. In my final submission, I chose equal weights, so that the bot is balanced, trying to improve its performance as much as to prevent the opponent from doing so.

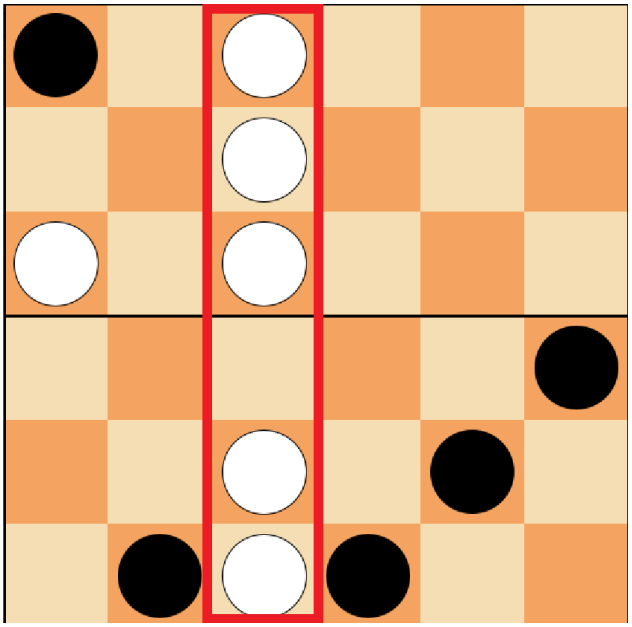

Figure 1: A favorable pattern for White

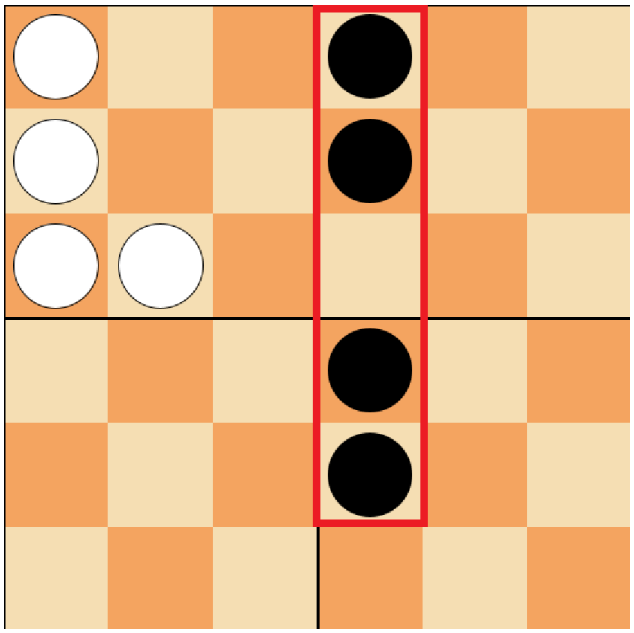

Figure 2: A favorable pattern for Black

2.3 Other approaches considered:

During the course of the project, I also implemented a simple version of the Monte Carlo Tree Search (MCTS), which had the advantages of not requiring a heuristic, while being unaffected by the branching factor (See Lecture 10 - MonteCarloTreeSearch). However, my implementation of MCTS was not performing as well as my α-β implementation.

From a theoretical point of view, this is understandable, since my α-β pruning algorithm will always focus on adding alignments, whereas my MCTS only chooses moves that have ways of resulting in a win at some point later in the game. Besides, given the same amount of time, MCTS will tend to visit less nodes at smaller depth, which could make the algorithm miss a winning or favorable move that α-β would not miss. In order to improve the MCTS agent, we would need to improve the algorithm, by pruning some uninteresting moves for instance.

3 Advantages and Disadvantages

While the α-β pruning agent can perform well and win agaisnt most novice human players, it is far from perfect. Here is a list of its various advantages and weaknesses:

3.1 Advantages:

- Every move the agent chooses will appear logical, and will often increase the number of aligned pieces while reducing the opponent's one.
- The algorithm conducts a (quasi-)exhaustive depth-3 search at every move, allowing it to never miss moves that result in a 'mate in 2' (i.e., an unstoppabable win in 2 turns).
- The bot is fully deterministic (i.e., it will always choose the same move at a given board state). So, while a MCTS bot could have a positive probability of missing an 'obvious' favorable move, the α-β pruning agent will always choose it (provided that its benefits can be seen at depth 3).
- The implementation of the simple pattern recognition function will often allow our agent to win against an otherwise identical opponent. Those patterns play the role of 'traps', in which an opponent with a rudimentary evaluation function might fall. Moreover, those patterns are meant to be hard to detect, since they use several alignments of 2-3 pieces, for which other students might give a smaller weight. Indeed, two alignments of 2 pieces (recall Figure 2) can actually be more dangerous for the opponent than a single alignment of 3.

3.2 Disadvantages:

- α-β pruning is only (theoretically) optimal against an opponent that uses the same heuristic as our player. Therefore, due to the complexity of the game and to the difficulty of evaluating a given state, it is very unlikely that any student in this class will use the same heuristic as me. However, if their heuristic is close to the evaluation function I used, the number of details (e.g., pattern recognition, weights given to specific alignments, etc...) could play a huge role in determining who the winner will be.

- On a similar note, the agent will tend to perform worse than expected against opponent that have a viable strategy that is not taken into account in the evaluation function. One of the main weaknesses of the evaluation function is the oversight of opponent's pieces that block an alignment (for instance on the big diagonals).
- Another weakness can be derived from one of the advantages cited above: since the algorithm has only an exhaustive depth-3 search, it will almost always miss 'mate in 2' opportunities for the opponent. Those moves can only be detected with a depth of 4 (depth-2 being the opponent's very next move).
- The above disadvantage (among other reasons) brings to light one of the easiest way to win against my bot 100% of the time (recall: the algorithm is fully deterministic). We can simply place pieces vertically or horizontally on the middle row/column of two adjacent quadrants, and keep adding pieces on the row/column until we can find a winning move. The reason for this huge drawback is that we would need a depth 4 to detect it and prevent this from happening (See 4. Possible Future Improvements).

4 Possible Future Improvements

4.1 Opening moves: optimizing the first 30-seconds long move

From the code analysis, we saw that the first few moves were chosen almost instantly by the α-β pruning agent. While it is obvious that spending 30 seconds choosing the first move seem useless, there are ways to make the best out of this given time.

An interesting idea could be applied to openning moves. As for chess, it seems that the first few moves could be known and learnt by the algorithm. Indeed, most of the empty slots are not worth considering during the first moves. So, an improvement could be done by implementing a function that, much like a machine learning algorithm, runs thousands of games (before the submission of the project), and keep track of moves that often lead to a favorable 'midgame' position. Those moves could then be written in a csv file, next to the associated board state (using the `toString()` provided function).

Once this library of openning moves is large enough, we code another function that would, during the 30 given seconds, load this csv into a hash map, where the key is given by the board state, and the value is the most favorable move discovered. We could then use this hash map on the subsequent moves to see if we know this position, and if so, to return the corresponding move.

This would guarantee a favorable entry into the 'midgame', that would hopefully result in a win.

4.2 Other improvements

Other improvements may include:

- Pruning identical moves (i.e., moves that lead to the exact same board state), which would result in fewer branches in the search tree, and a larger number of nodes visited;
- As seen theoretically (See Lecture 9 - Games), randomizing the move ordering would ensure a better efficiency, with a time complexity averaging $O(b^{\frac{3m}{4}})$;
- Depth-4 is key. Achieving an exhaustive depth-4 search (via additional pruning for instance) would prevent our agent to fall into the opponent's 'mate-in-2 traps'.

Conclusion

To conclude, this project allowed me to understand the course content even more, by applying the theoretical knowledge I have learnt, and manipulating algorithms within a game with high branching complexity.

The competition aspect of the grading scheme motivated me to do my best, and pushed me forward. In order to obtain a agent that would match my performance expectations, I had to face many challenges, and to develop an efficient program, as well as ideas that could make me stand out during the tournament.

As has been noted, many different approaches might be considered, so it would be worthwhile to compare and analyze them, both conceptually and practically, to determine the most efficient way to apply Artificial Intelligence to the Pentago-Twist game.