AI Final Project Report Winter 2022

Ling Fei Zhang, 260985358
Brandon Ma, 260983550

April 8, 2022

1 Motivation

Considering that Colosseum Survival is a game very similar to Chess and that we're not allowed to use File IO, we considered using either Minimax with Alpha-Beta Pruning or Monte Carlo Tree Search. At lower board sizes, both of these approaches are solid algorithms that can be used for our agent. However, higher board sizes will make MCTS choose moves that are almost at random since the search space is too large. Minimax, on the other hand, may not reach a very high depth, but a good heuristic can compensate for this. This is why we have chosen to implement Minimax with Alpha-Beta Pruning: at lower board sizes, it will see high depths and at higher board sizes, it can follow a general strategy.

The approach that we have chosen to do is trying to reach the highest depth possible within the alloted time. The strategy that we have chosen to do is taking moves that favor a centralized position, limiting opponent's moves and avoiding being boxed in. Details of this strategy is described on the following page.

Note: In this document, the terms "checkmate" and "mate" refer to boxing an opponent.

2 Theoretical Approach

First, we need to implement some basic strategies. We implemented a Depth First Search in order to enumerate all the possible moves given a specific position, a chess board and a maximum number of steps. We are using a heuristic that is considering the following points at every turn:

1. Check how close we are to the center. The closer we are, the more points we award to our agent. In parallel, we also evaluate how close the opponent is to the center. The heuristic score is given by our agent's position relative to the opponent's position towards the center.

2. Check the number of walls surrounding both our agent and the opponent. The distance to consider is $n = \lfloor \frac{\text{Board Size}}{4} \rfloor$. The heuristic score is given by the number of walls around our agent relative to the number of walls around the opponent.

3. Check how many moves our agent and the opponent can make. The heuristic score is given by the number of moves our agent has relative to the number of moves the opponent has.

4. Check if we can checkmate the opponent. If so, the heuristic will give a score of ∞ to the position. However, if the opponent can checkmate our agent, then it will instead have a heuristic score of $-\infty$.

The algorithm that we have chosen to implement is Minimax with Alpha-Beta Pruning. The goal is to use a heuristic function to give a score to each move. This implies that the agent can decide which move has better winning chances relative to others. The leaf nodes that we get from the Alpha-Beta algorithm are positions that are evaluated according to the four points above, each having a different weight. We have decided to make our agent more aggressive, thus we are giving a higher scaling factor to limiting the opponent's moves. To help with Alpha-Beta Pruning, we sort the first set of moves in order of closeness to the center.

The way we have chosen to perform our search is using Iterative Deepening Search. The algorithm will try to reach the highest depth possible within two seconds. If it times out, it will take the best move of the previous depth search.

3 Advantages and Disadvantages

3.1 Advantages

Since we let the algorithm run for the entire two seconds, we are able to reach high depth with small size boards. Our program is always able to find mate in 1's, or a forced mate and we'll avoid moves that get ourselves checkmated whenever possible. It also favors centralized play, which reduces chances of getting cornered.

Another advantage of our implementation is that we can easily tweak the weights of the heuristic functions such that the agent can adapt to more aggressive/passive plays.

3.2 Disadvantages And Weaknesses

The algorithm doesn't prune as much as we expected. Hence, at higher board sizes, it is only able to reach a maximum depth of 1 in most cases. Furthermore, when the opponent is out of the range of our agent's moves, it will play extremely passively, as it does not know what to do. In some cases, it even plays a move that disadvantages itself. Due to the nature of Minimax, our agent assumes that the opponent will play optimally. Thus, if the program sees that the opponent can force us into a mate, it will view all moves as a loss and boxes itself.

4 Other Approaches

We've tried other approaches at the beginning such as using a Breadth First Search in order to generate the tree of possible moves. We realized that at each iteration of a Breadth First Search, we would also have to keep track of all the nodes up to the depth that we are searching for. On the other hand, an Iterative Deepening Search will not have to keep all of them in memory since either some nodes are already visited or they are nodes that we will visit later. Not to mention, Iterative Deepening Search also does better in our situation since we have a time constraint. In the event that we reach the time limit, the algorithm will still have some sense of what a good score is even without a full traversal of the desired depth.

After discovering our flaws with our Minimax algorithm, we attempted at removing moves that were futile, such as moving towards the edge of the

board during the start of the game. When we exceeded a certain amount of possible moves, we were hoping that this would reduce the search space of our program. However, it led to a few errors, which resulted in the agent performing random walks at times. This was likely due to removing moves that could've been the only escape of the agent. It was also difficult to determine an appropriate cutoff value to determine when we would perform this elimination of moves.

Another approach that we have considered is to add a distance heuristic function. If the chessboard is one wall away from ending the game (i.e. either player1 or player2 wins if the wall is placed), then the function would calculate the distance of each possible move that our agent can make towards that game ending wall. The closer the moves are to the wall, the higher score they are given. The issue with this approach is that we cannot use an absolute distance since walls might be in the way. We would have to implement a shortest path algorithm which would take way too much time to compute within a two second timeframe. Hence, we have decided to not follow this approach to allow for potentially higher depth search.

5 Further Improvements

Considering that we are running a Minimax Algorithm with Alpha-Beta Pruning, a good improvement to our program is to sort the list of possible moves from best to worst before giving it to the algorithm. This way, the pruning can be more efficient, and the bad moves can be eliminated quicker.

Another improvement is to use Quiescence Search. This consists of not setting an absolute depth for each branch of possible moves. Since the algorithm is limited in how far it can look ahead, it is very possible that it misses a sequence of moves that can occur at leaf nodes. For example, the agent might want to avoid moving closer to the opponent because it forces the agent to put a wall "behind", and closing off an area of squares. If the agent only realizes that he loses a good portion of the squares without knowing that this move wins the game or puts the opponent at a great disadvantage, then it will likely not play it. A way to get around this is to perform a second round of limited search on the leaf nodes that are "unstable" (like the example above, or any move that puts the agent at an uncomfortable position, but that disadvantages the opponent even more). This way, it resolves the issue in a more dynamic way, and also makes the heuristic more accurate.