COMP 424 Final Project Report

Zhiming Zhang ZHIMING.ZHANG@MAIL.MCGILL.CA
School of Computer Science, Department of Mathematics and Statistics
McGill University
Montreal, Quebec H3A 0G4, Canada

Minh Anh Trinh MINH.TRINH@MAIL.MCGILL.CA
School of Computer Science
McGill University
Montreal, Quebec H3A 0G4, Canada

1. Abstract

The goal of this project is to build an AI agent that plays in the ***Colosseum Survival!*** (https://github.com/Ctree35/ Project-COMP424-2022-Winter) game, where the AI agent tries to put barriers around itself in a chessboard such that when it is separated from its opponent in two different closed zones, the number of blocks in its zone is maximized to win against the other opponent.

2. Technical Approach

We used the Negamax algorithm to build our AI agent. In our *negmax* function (line 180-237), we first check for whether the game ends using the *check_endgame* function, which return the boolean value of whether the game is over and compute the current score of the agents. If the game is over, we return the game score by using the *get_score_game_over* function to compute the difference in scores of the two agents, and return *None* for *best_move*, since it is unnecessary to know about the best move at this point. Moreover, because decision about a move has to be made within 30 seconds for the first move and within 2 seconds for each subsequent move, in lines 188 and 189, we make sure that our AI agent does not exceed the time limit. We set the global variable *time_out_value* to 25 for the first move (line 297) and to 1.5 for subsequent moves (line 301). The reason as to why we set *time_out_value* to be smaller than the allowed time constraints is due to the possible random errors of computers when computing the time. When we tested our code, our AI agent are still able to make accurate decisions within the time constraints that we set out. Hence, we believe that our set time limits are reasonable.

Similar to the Minimax search algorithm introduced in class, we compute utilities when game is over or when $depth = 0$ (lines 192-196). Our evaluation function is the *get_score* function in lines 55-88, which compute the score at the leaf states from our AI agent's point of view. It computes the score for the player as follows:

$$\sum_{c \in S} (maxStep - step(c))$$

Where S is the set of blocks reachable by the player and step(c) is the minimum number of steps needed by the player to reach block c. The heuristic score is computed as score(player1) - score(player2). The logic behind of this heuristic score is that we want our player to have more blocks reachable and fewer number of steps to reach them. The *get_score* function enumerates the reachable blocks using BFS search algorithm.

When the game is over, that is, the two players are separated into two closed zones. The heuristic score is simpler to compute. The player score is computed by the number of blocks reached by the player multiplied by 10000 and the heuristic score is also the difference of the two player score. By multiplying 10000, we want the terminal game state score dominates non-terminal game state score.

We then generate all possible valid moves from the current position of our AI agent and store these successor moves in the variable *moves* (line 201). This is done using BFS search algorithm. Since the goals of our AI agent is to maximize the number of blocks in its zone when two agents are separated in two different closed zones, our strategy to win the game is to advance as far as possible from the current position and come as nearest as possible to the adversarial opponent. Therefore, before searching over the successor moves, we sort them based on the distances with the opponent (line 203). The moves result in positions that have the closest distances with the opponent are prioritized. With this ordering, the best nodes are going to be checked first by alpha-beta pruning. This will optimize alpha beta pruning as more frequent alpha/beta cut-offs will occur. Hence, more branches are pruned which lead to faster game tree search.

Before beginning the foreach loop search, we preset the *value* variable to be the lowest value possible. We then use iterative deepening depth-first search to search over the successor moves. In the foreach loop search, we first make a deep copy of the chessboard and execute the current move. We then compute the return for the other agent's move and save it in *subscore*. Since the utility is being computed from our AI agent's point of view, we compare *value* with the negated value of *subscore*, and note new best score if the negated *subscore* is bigger than *value* and set this as the best move. As we use alpha-beta pruning to improve tree search performance, we adjust the search window by updating the *alpha* varaible by choosing the bigger value between the current *alpha* and *value*. After that, we make sure that if the *alpha* value is bigger than or equal to the current *beta* value, we know that the successor nodes contain worse utilities than the one we currently have. Therefore, we cut off the search at that point to avoid wasting time searching deeper.

2.1 Motivation

There are many factors that motivated us to use Negamax algorithm. First of all, Colosseum Survival is a deterministic game with perfect information. All the agents in the game are able to observe their opponent's move and the exact state of the game. It also has a set of rules by which the agents abide to make their moves, which fully determine the state changes of the game. Moreover, the agents in this game play in an $M \times M$ chessboard where M is between 6 and 12, such that that the maximum number of steps for each move is restricted to $K = \lfloor \frac{M+1}{2} \rfloor$. Due to these restrictions, we knew that the branching factor for the search state would not be a huge number, and algorithms like Minimax could be used. Most importantly, Colosseum Survival is a two-player game that has zero-sum property. When the two players are separated in two different zones, the number of blocks that one player obtains in its zone are the exact number of blocks the other player loses to its adversarial opponent. Thus, the two agent has conflicting goals such that the utility of one agent is the negation of the utility for the other agent, $\max(agent_1, agent_2) = -\min(-agent_1, -agent_2)$[1]. For this reason, we used Negamax which is a simplified variant of the Minimax algorithm learned in class. However, instead of having to maximizing or minimizing utility depending on whichever agent's turn it is like in Minimax, with Negamax we will always maximize utility as the value of utility is calculated in one agent's point of view only. These reasons motivated us to use the Negamax algorithm, which we believed was efficient.

2.2 Theoretical Basis

[1] Similar to the Minimax algorithm learned in class, Negamax uses the same search tree in which each node represents a game state resulted from a move of an agent. Negamax is a recursive algorithm implemented as an iterative deepening depth-first search. Negamax expands the search tree to the terminal states and compute the utilities in these leaf nodes. Then the algorithm backs up from the leaves towards the current game state. However, the only difference Negamax has compared to Minimax is that instead of using two separated subroutines for the Min player and Max player, Negamax searches for the maximum value for all its nodes as it negates the utility for the adversarial opponent such that $\max(agent_1, agent_2) = -\min(-agent_1, -agent_2)$. Moreover, alpha-beta pruning is also implemented in our Negamax algorithm. With the use of alpha-beta pruning, the number of nodes that are evaluated by the Negamax algorithm is reduced as it prunes away branches that seem to be worse than the current utility (when $alpha >= beta$, where $alpha$ is the highest leaf value found for our AI agent and $beta$ is the lowest leaf value found for the adversarial opponent), hence cannot possible affect the final optimal path.

2.3 Advantages and Disadvantages

As Negamax is very similar with the Minimax algorithm learned in class, it also has the same properties as Minimax. Since the game is deterministic and there is a limit to the size of the chessboard as well as the number of maximum steps for each move (as mentioned in the previous subsection), we know that the search tree is finite. Hence, the first advantage of our algorithm is completeness. On the other hand, we ordered moves based on our strategy as explained above and used alpha-beta pruning, which improve the search tree performance for the AI agent to make decision within the allowed time constraints.

However, there are also disadvantages to our algorithm. Similar to Minimax, Negamax has to search every node in the search tree in order to take actions that maximize its utility. For bigger board sizes, our algorithm may require longer time to make a decision. Within the time constraint, our AI agent may not be able to search deeper nodes, which results in the possibility that our AI agent may not be effective in selecting the optimal path.

3. Discussion

Prior to using Negamax algorithm to build our AI agent, we used Monte Carlo tree search (MCTS) algorithm. However, repeated testing of our code yielded lower winning rate against random agent. This is due to the fact that MCTS does not use an explicit score-estimating evaluation function to decide the best move, but rather uses the statistics from playing out stimulated games itself. Hence, the longer the algorithm runs, the more the AI agent can effectively select the most efficient path. However, we found that the given time constraint for each move did not allow our AI agent implemented with the MCTS algorithm enough time to run through a huge number of iterations, especially with large board sizes. Hence, it seemed that with large board sizes, our AI agent implemented with the MCTS algorithm was less likely to choose the optimal path, which resulted in lower winning percentage against random agent than the AI agent implemented with the Negamax algorithm.

Moreover, we are given the maximum memory usage of 500 mb of RAM. However, MCTS requires large amount of memory. Therefore, with large board sizes, the AI agent implemented with the MCTS algorithm can rapidly exceeds the memory requirements after a few iterations.

3.1 Improvements

There are two improvements that we think will make our AI agent faster and more effective. First of all, we could improve the tree search performance by initializing *alpha* and *beta* with alternate values, instead of setting them to negative infinity and positive infinity respectively. We believe doing so could reduce the search window, hence pruning more possible uninteresting branches. Secondly, some of the game states may repeat, we therefore could use a hash or lookup table to store the game states in order to retrieve the moves quickly, which will speed up our AI agent's decision making process.

References

[1] Wikipedia contributors. *Negamax — Wikipedia, The Free Encyclopedia.* [Online; accessed 13-April-2022]. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Negamax&oldid=1067237200`.