COMP 424 Final Project Report

Course Instructor: Jackie Cheung and Bogdan Mazoure
Students: Michael Lu (michael.lu2@mail.mcgill.ca, ID: 260915796) and
Nuri Ege Zararsiz (nuri.zararsiz@mail.mcgill.ca, ID: 260922586)
Due Date: April 12th, 2022, 11:59PM EST

1. Solving the Problem

1.1 Introduction

Throughout the term, we learned a variety of strategies that agents can use to mimic the cognitive functions of human intelligence. For example, knowledge representation as memory, search and inference as reasoning, planning and decision making as executive control, and model learning as learning. To better understand what strategy to use, we must first analyse the type of problem we are trying to solve.

1.2 Analysis of the Problem Domain

The game, ***Colosseum Survival!*** (https://github.com/Ctree35/ Project-COMP424-2022-Winter), is a 2-player turn-based strategy game in which the 2 players move in an MxM chessboard and put barriers around them until they are separated in 2 closed zones. M can range from any value between 6 and 12 inclusively. The goal of each player is to maximize the number of blocks in their zone by the end of the game in order to win.

There are many aspects of human intelligence that require coordination in order to play this game such as search and inference (being able to reason what is a "good" move and what is a "bad" move) and model learning (learning from games played in order to develop a winning strategy). In addition to this, the agent must also abide by the strict bounds of speed and efficiency. The agent is limited in the number of cores it can access (CPUs) and the time it can take to play its moves (30s on the first move, 2s for every move after).

1.3 Motivation for Approach

To better understand the mechanics of Colosseum Survival and develop a winning strategy, we decided to research games that exhibit similar characteristics. We first started out with board games we already knew and covered in class such as Chess and Go. After searching around, we discovered another strategy game called "Quoridor" which implements similar game mechanics as Colosseum Survival such as wall placements, blocking opponents, and controlling squares. One common thread between all these games was that the AI agents designed to master them used MCTS. Satisfied, we decided to implement this search algorithm in our agent following the approach covered in Lecture 9.

1.4 Theoretical Basis for Approach

To make good use of the allotted 30s on the first move, we decided to use a Vanilla MCTS combined with Alpha-Beta Pruning. However, because we did not implement a database in our approach, we found this method to be too time consuming and inefficient to use in subsequent moves. This is because we could not simulate enough games within the 2s for an accurate representation of the "goodness" of a move. We then decided to focus on the simulation aspect of MCTS for the 2s moves. Using Alpha-Beta Pruning along with additional heuristics to trim the list of moves, we were able to run many simulations within the 2s time limit which increased the certainty that the move our agent selected was good.

Due to the random nature of MCTS, one caveat we thought of was the possibility of our algorithm to miss game winning moves. To address this, we decided to implement one-step forward checking to test every move the agent can make and see if it results in a win. This ensures the agent can always find the winning move if it exists.

The Vanilla MCTS and the Alpha-Beta Pruning algorithms we used for our agent were implemented based on the lecture notes while adjusting certain aspects to better fit the game. The modified version of the MCTS algorithm we used during the 2s move turns and the one-step forward checking were implemented by ourselves based on our understanding of the game and the advantages and disadvantages of Vanilla MCTS and Alpha-Beta Pruning.

1.5 Explanation of How the Program Works

The program we came up with draws inspiration from Alpha-Beta Pruning, Bayesian Networks, and Monte Carlo Tree Search (MCTS). The beginning of our step function runs our posMoves() function which checks for all valid moves the agent can make from its current position by looping through a K*K region and checking whether the move is valid or not. The return of this function is a list. From this list, we apply each of the moves to the chessboard and check whether the game has ended and that our agent has won. This portion of our code ensures the agent can find the winning move (if it exists) deterministically without having to rely on the random simulation from MCTS. If no winning move is discovered, we pass the input data to our MCTS portion.

From here, we first check whether this is the first move or not. If it is the first move, we expand the node and run our Alpha-Beta Pruning algorithm to eliminate poor moves using several heuristics such as detecting shallow traps (eliminating moves that create 3 walls surrounding our agent), controlling the distance between our agent and the adversary (this was implemented conditionally for the larger boards to limit the branching factor), and removing moves that end in an immediate loss (whether played by our agent or by the adversary). Although our usage of Alpha-Beta Pruning is shallow, because we used it at the beginning of the move (at the root), it is effectively used as our Tree Policy to generate a list of high-quality and reasonable moves. From here, we follow the steps in MCTS and run for 25s:

1. Selection (usage of Alpha-Beta Pruning as a Tree Policy to discover moves - potential nodes)
2. Expansion (random selection of [at most] 10 nodes from the list returned by our Alpha-Beta Pruning)
3. Simulation (expand a node and run simulations by using a Default Policy [random walks] for both the agent and adversary until the game is over)
4. Backpropagation (return the result of the game [win(1), lose(0), or tie(0.5)] and update the value at the expanded node)

To aid in the selection of a node to expand upon, we implemented Upper Confidence Trees (UCT) to help create a balance between exploitation and exploration. After running many simulations on the possible moves, we select and return the move with the highest win percentage and set the bool firstrun to false so that the future moves would only run for 1s. The reason why we selected 25s and 1s for the first and future moves is to eliminate the possibility of timing out due to the randomness of the board size and wall placement.

2. Analysis of Agent Performance

2.1 Advantages of Our Approach

The primary advantage to using Monte Carlo Tree Search as the main search algorithm is that it can effectively operate without any knowledge of the domain apart from the rules and end condition. Other advantages include its ability to find its own moves and learn as it plays. This is a particularly important aspect of the agent in terms of adaptability since it can still play (relatively well) given any board configuration.

The advantage of combining both MCTS and Alpha-Beta Pruning can be seen by looking at the downsides of both. Alpha-Beta Pruning fails when there is not a strong evaluation function, but MCTS does not rely on any evaluation metric. MCTS fails in short-term visibility due to the random selection of moves for each player, but Alpha-Beta Pruning considers all moves by both the agent and the adversary and can eliminate moves that end in a loss whether by the agent trapping themselves or getting trapped by the adversary. As a result, MCTS can spend more time running simulations on fewer moves to get a more accurate result about the quality of the moves.

2.2 Disadvantages of Our Approach

According to tournament rules, our agent is given 30 seconds for the first move to perform any required setup and 2 seconds for each move thereafter. Although our agent utilizes a majority of the initial 30 seconds (we set a threshold of 25s to avoid the possibility of timeouts) to run as many simulations as possible before selecting the best move for itself based on the adversary's possible first moves, the results are unfortunately not saved in a database for future reference. This is a critical flaw because our agent does not make good use of the allotted 500 MB of RAM. As a result, the agent cannot utilize its past simulations

and build upon them which ultimately limits the depth of the search.

Like a double-edged sword, the way we handle node selection for expansion is both an advantage and a disadvantage. By greedily selecting up to 10 moves, we allow more simulations to be performed on a smaller set of nodes which increases the certainty of our agent's decision that it is making the best possible choice from its repertoire. However, the downside of this is that by omitting moves, we introduce the possibility of eliminating the absolute best possible move the agent could make in its current position.

2.3 Expected Failure Modes/Weaknesses of Our Approach

Our agent struggles to make good strategic moves when the game is played on a larger board ($M > 10$). As the board size gets larger, there are more moves to consider due to the large branching factor. When dealing with a large branching factor, the Vanilla MCTS is rendered ineffective because the games take longer to complete which results in fewer games simulated. This, combined with the huge number of moves to consider limits the accuracy representation of the "goodness" of any particular move.

To account for the limitations MCTS experiences in larger boards, we implemented several heuristics within our Alpha-Beta Pruning part of the algorithm to try and reduce the number of options. However, as the game progresses and the number of valid moves decreases, these heuristics may lead the algorithm to prune a majority of moves from its list, limiting the movement of our agent and the breadth of the search. The consequence being that after simulation, none of the moves may be "good".

3. Future Improvement

3.1 Memory Management and Utilization

To address our first disadvantage we could implement a dynamic programming (DP) table to store the results from each expansion. Due to the Default Policy applied during the simulation step in MCTS, there exists the possibility of reaching the same board configuration through a different sequence of moves. In chess, this is known as transposition. By sharing the knowledge between transposable states, we can apply a Rapid Action-Value Estimation (RAVE) heuristic to return a rapid (but biased) value estimate for the move in consideration. This combined knowledge may allow us to prune moves that are below a certain threshold based on previous simulations. As a result, we can eliminate moves from consideration faster and focus more resources on either exploration of other options or exploitation of the current best move depending on the value returned by the UCT.

3.2 Decision Based Model Move Selection

To address our second disadvantage we could integrate a Bayesian/Neural Network to help make decisions on what nodes to keep for expansion based on previous experience.

Limited by both time spent working on the project and tournament rules, we were not able to implement a decision based model to help the agent choose what moves to keep and what moves to discard. By adding a Bayesian/Neural Network, the agent can factor in its current position, the adversary's position, and the current state of the board and receive a weighted list of moves. This ensures that the agent has selected 10 locally optimal moves to keep and run simulations on rather than 10 random moves. In addition, over the course of thousands of games, our agent can learn to recognize hidden patterns and correlations from the game data and continuously learn and improve its strategy.

4. Other Approaches Considered

4.1 Bayesian Networks

In the beginning, we initially started with a Bayesian Network approach. The idea behind this was that once the network was constructed, we would utilize automation to train the agent. Ideally, we would take 2 version of our agent (the current best version and a new version with one or more weights updated) and pit them against each other. Similar to hill-climbing, whichever agent wins would become the current best and we would repeat this process thousands of times to converge on the local optimal agent. However, we found the implementation to be difficult and slow and abandoned the idea. Compared to our final approach, we did not implement model learning, but we did use a few concepts from our Bayesian Network design to help create the heuristics that we used with our Alpha-Beta-MCTS hybrid algorithm.

4.2 Alpha-Beta Pruning

The next idea we considered was using Alpha-Beta Pruning. Similar to chess AI Stockfish, we planned to search through the different states in the tree from our current position and prune unnecessary branches (moves) using greedy heuristics in order to find better moves faster. One problem we encountered early on was deciding on the evaluation function to use. Unlike in chess where the pieces have material value, there was no clear evaluation metric we found to be sustainable in evaluating the board position. We tried to implement an area evaluation function by completing the wall or "filling in the blanks" to identify who currently controls the most area. However, this proved to be difficult because of the different ways to complete the wall and other factors to consider. Like in chess, material value is not the only metric used to evaluate one's position. Other factors to consider in this game include potential traps, strategic boundaries, and forced/useless moves. Without a strong evaluation function, this made Alpha-Beta Pruning difficult to implement and due to the large branching factor, very costly in time to search for a good move. Compared with our final approach, we decided to implement a shallow search using Alpha-Beta Pruning with the evaluation of win, loss, and neither.