COMP 424 Final Project Game: *Colosseum Survival!*

Eric Pelletier eric.pelletier@mail.mcgill.ca
260895863

Matthew Wittman matthew.wittmann@mail.mcgill.ca
260911144

1. Approach

For our agent, we decided to implement a Monte-Carlo Search Tree algorithm. We decided to use this method for several reasons. First, the search space becomes exponentially large as we increase the size of the board. Second, we are limited by a time constraint during our decision process. Finally, there is not a clear deterministic decision of which action to take until we get closer the end game.

We first started by creating a *GameState* class in which we would be able to create an object to keep track of various simulations throughout our Monte-Carlo Search Tree algorithm. This class encapsulates information about the current state of the chess board, the position of both player as well as which player is making the next move. In addition, this class contains several method used to advance the state of the game. The first method defined is called *play* and it is used to execute a specified move for the current player. In addition, this class contains a method called *moves_for_curr_player* which generates all the valid moves that the current player is able to execute. To generate the valid moves, we started by creating a move for every possible combination of movements in the x and y coordinates that sum up to less than or equal the maximum step allowed along with each possible direction of placing a barrier. Then, we checked if the move was a valid move given the current state of the chess board ensuring that no move will lead to the player passing through a an existing barrier on the chessboard, no move will lead to the player trying to place a barrier at the location where one already exist and no move will lead to the player trying to pass over the opposition. The final method that this contains contains is *check_endgame*. This method was copied from the world.py file and was adapted to output whether the game has ended in addition to which player was the winner.

Next, we incorporated a *Node* class which is used to executed actions during our Monte-Carlo Tree Search algorithm. The node keeps track of its parent node, its children, as well as how many times it has been visited and its average reward. In addition, this class incorporates a useful method that is used to calculate the UCT value of the node relative to its parent. The value is calculated by first checking if it has already been visited. If it has not been visited, it will return a UCT value of 0 or else it will return a value equal to $Q/N + 0.5 * \sqrt{2 * log(parent.N)/N}$.

Having set up these 2 classes, we set up a final class called *MCTS* where the Monte-Carlo Tree Search algorithm will reside. This class first incorporates *heuristic_choice*. This method is used throughout our Monte-Carlo Tree Search algorithm to select a move that is a bit better than a pseudo-random move. We started by checking if there is a move that will lead us to ending the game, if so, we return that move. Next, we build 2 list that contain "good" moves. The first list contains moves that make the agent has a maximum of

2 barriers around itself after executing the move. By doing this, we can try to take out the possibility of surrounding itself with 3 barriers which may lead to the opposing agent closing the fourth wall around our agent and winning the game. The second list contains moves that stray away from the border of the chessboard by checking whether the position of the move is equal to 0 or the max board size - 1. This can help our agent from getting trap against the wall and possibly losing the game that way. Finally, we eliminate moves that lead to the agent ending the game and losing. If no moves are present in the 2 list mentioned, then we just return a random choice of move from the set of possible moves. Next, *MCTS* class, incorporate a *select_node* class in which the it will take in the root Node of the current game as well as the current GameState and select a children node to visited. The children nodes correspond to all possible reachable GameState from the current GameState by executing a single move. This method selects which node is its going to visit with the heuristic choice function as well as favoring nodes that have no been visited yet. This method continues to iterate over the children of the Nodes until it has reached a leaf that does not contain any children or a terminal node. If it reaches a leaf node, it will generate the children for that given node and return one of the based on the heuristic choice function. if it reaches a terminal node, it will just return that terminal node. Next, we incorporated a *rollout* function which is used to simulate a game play until it finished by generating move for the 2 players based on the heuristic choice function. Once it has fully simulated a game, it will return the winner for that game which then gets passed to the *backup* function. The backup function is used to calculate the reward that will get update at each parent node until it reaches the initial root node. The reward value is equal 1 if the player that has just played won the game, 0 if it has lost the game and 0.5 if the game ended in a tie. At each parent node, the reward value is switched to give an accurate reward to each node representing the move of the correct player. We execute the 3 functions, *select_node*, *rollout* and *backup* inside of a *search* function, until the specified time limit for the given player has been reached. With the *MCTS* class, we were finally able to complete our *step* function of our agent. In the *step* function, we first check whether it is the very first move that our agent is executing. If so, we will create a *MCTS* object with a GameState corresponding to the current chessboard, our position, the opponent's position as well as the maximum step allowed. In addition, we also create an empty root node. Since it is the first move for the agent and we are allowed to use a time limit of 30 seconds, we execute the MCTS search funcition with a time limit of 20 seconds. Next, we select the best move of the MCTS by selecting the move corresponding to the children node of the root with the highest UCT value found by our MCTS algorithm. If multiple nodes have the same UCT, we break the time by using our heuristic choice function. Finally, we update the GameState by playing our move locally and we return the move from the function. If its not the first move of our agent, we will determine what was the move executed by the opponent and play that move on our current GameState to ensure that we keep track of the game. By doing this, we are able to reused nodes that were excuted during the previous iteration until of restart with an empty node from the current GameState. Before, running our MCST search function and returning the best move found, we first check whether there is a possible move that will end the game leading to our agent's victory. If so, we return that move to win the game. When it is not the first move of the game, the MCST search function executes with a time limit of 1 second. We decided to use time limit of 20 seconds and 1 second to ensure that

we do not surpass the limits of 30 and 2 seconds respectively. We noticed that a time limit of 29 seconds and 1.7 seconds were sufficient while doing coming up with the best moves on board with a maximum size of 7. However, the algorithm seemed to struggle to stay under the time limit while operating on boards with sizes of 8 and 9. Therefore, we decided to use a time limit of 1 to be of the safer side since executing random moves is still much worse than any possible move that our agent can come up with within the time constraint.

2. Theoretical Basis

Each game can be represented with a directed graph such that nodes are game states and edges connecting the nodes represent a decision a player makes to achieve the successor game state. It's also important to note that outcomes depend solely on the decisions of players with equal amounts of information. Representing Colosseum Survival in game tree format allows for the analysis of the potential decision based outcomes which assists in finding the optimal sequence of decisions leading to a win. Since the board's shape is an mxm matrix, where m can take on a value ranging from 5 to 10, it would be unreasonable to build a complete game tree. With the use of a Monte Carlo Tree Search, a best-first search algorithm, an AI agent playing Colosseum Survival simulates random play-outs of the game generating insight regarding what the next best move is. The value placed on each of the potential actions is determined by the outcomes of the simulated games.

3. Evaluation of the Approach

3.1 Advantages

A major advantage of our approach is our heuristic choice function. Since the opposing agent is not selecting random moves during the simulation phase of the algorithm, our agent is able to gain more confidence. This is mainly from the fact that the opposing agent will most likely not select a random move that will terminate the game by boxing it self into a space that will lead to its loss. In addition, our agent is also able to select moves that might lead to it not being boxed in itself since it will stray away from the walls. Due to this, our agent will most likely be able to survive longer than the opposing agent. Furthermore, by selecting a move that will end the game and lead to the victory of our agent we able to *attack* the other agent. This becomes very useful since if we were to randomly select a move from the best moves available there is a possibility that our agent does not select the move that will end the game since it has not explore that path enough yet.

3.2 Disadvantages

However, a disadvantage of our approach is the time limit that we give to our MCTS search function. Since, we know that we could of had a time limit of around 1.7-1.9 seconds for smaller boards, we could of changed the time limit that we pass to the function based on the board size. This would lead to our agent performing better for smaller board sizes without has to sacrifice the possibility of exceeding the 2 second time limit for bigger boards. Since there is a significant different for an execution time of 1 second to 1.9 seconds, we believe that this could of increase our performance by a significant amount.

4. Previous Ideas

A previously considered implementation involved the use of a minimax algorithm in conjunction with alpha-beta pruning. Deciding against this implementation came down to the loss in time efficiency as the search space grows in complexity. With a large branching factor, the player has lots of options to evaluate before ultimately choosing one. With only two seconds to decide on the best move, the branching factor of the tree would eventually inhibit the agent's ability to reason about each potential option. To reduce the computation time, alpha-beta pruning was considered because of its utility in reducing the number of branches to search.

5. Improvements

In addition to the possible improvement mentioned in the disadvantages section, the significant improvement to our approach would be to incorporate a heuristic evaluation using a implicit minimax backups. We would perform this improvement by adding an addition value at each node which corresponds to the heuristic minimax value. We would update this value whenever we perform the backup function the Monte-Carlo Tree Search algorithm by looking the moves taken at each step. This would help in the decision process of our algorithm in which we would be able to evaluate the nodes with this addition value and determine which node to expand based on this minimax value, favouring nodes that have a higher value.

6. References

https://towardsdatascience.com/monte-carlo-tree-search-implementing-reinforcement-learning-

https://github.com/masouduut94/MCTS-agent-python/tree/2df128f87299ff3f93fc7ac7bba507ab52

https://www.cs.huji.ac.il/w~ai/projects/2012/Quoridor/files/report.pdf

https://upcommons.upc.edu/bitstream/handle/2117/127676/131875.pdf?sequence=1&isAllowed=y

https://project.dke.maastrichtuniversity.nl/games/files/bsc/Mertens_BSc-paper.pdf

https://ieeexplore.ieee.org/document/6633630

https://link.springer.com/chapter/10.1007/978-3-319-14923-3_4