COMP 424 Final Project Game: *Colosseum Survival!*

Luis Yoon, 260773791 & Maxim Boucher 260902501

1. Program Description

For this project, we have implemented a fairly simple version of **Monte Carlo Tree Search (MCTS)** based on the the slides, with some added adjustments used to tailor it to the specific needs of this project. We will begin by going over our motivation for this particular approach.

Prior to the programming portion of the project, we first narrowed down all the viable options to iterative-deepening Alpha-Beta Minimax and MCTS, both of which we have seen in lectures. We started off with AB-pruning Minimax, but due to unforeseen circumstances our first day of the actual implementation was delayed. It was not long until we realized that creating an evaluation function for this game would be a significant challenge. This was due to the fact that simple-but-effective evaluation methods common in other games, such as taking the ratio of player pieces vs adversary pieces, number of points obtained so far, or distance to a goal, etc... were not applicable for this game. The fact that MCTS does not require a complicated, hand-crafted evaluation function made it a more desirable choice, so we decided to implement that instead. Nevertheless, we still had our doubts as to whether or not MCTS would be suitable for the 2 second time limit and 500MB memory constraint. Fortunately, after getting a sense for the computation speed using auto-play and after running tests using tracemalloc() to track the memory cost of copying and storing 10000 passed-in board states, our concerns were alleviated; plenty of fully-random games could be played in a short amount of time, and the memory use never exceeded 187.5MB.

Now that our doubts had been cleared, it seemed to us that MCTS was the most effective and safest to implement within the short time span. Additionally, there were several techniques covered during lecture such as RAVE, alternate tree policies and more that we could use to further improve our algorithm if results were inadequate or if we had extra time before the due date. Ultimately, the only such technique we implemented was the Upper-Confidence-Tree policy (from the lecture 9 slides 42-49, referred to as UCT from now on) due to its effectiveness and straightforward implementation.

In essence, our algorithm follows a simple MCTS structure, using the UCT policy for tree nodes and a random-selection default policy. Here are the additional helper classes constructed for use within the step function:

- Board State:
  Contains the state of the game at a given step: the board as well as the opponent and player positions. This information is used in one of its own functions to simulate random actions. A function to verify the terminal state of the game is also included.

- MCTSNode:
  Represents the nodes of the MCTS tree, which contains a BoardState, the (win count)

and the total number of visits to be used for UCT calculations. It also has a function used to expand the nodes, which adds a **subset** of valid moves to its array of children (we will go in-depth on this decision later).

- UCT:
  A simple, 'static' class that is used for the tree-policy calculations.

The step function begins by getting the current time and defining a time limit which we've set to 1.9 seconds; this leaves 0.1 seconds for the main loop to finish its computation, thus ensuring the overall 2 second time limit is respected. Then, we proceed by copying the passed-in state (the board and positions) into a new BoardState, which is then assigned to a new MCTSNode to serve as the root node of the MCTS tree. The main loop then begins, which will go on for as long as it does not exceed the 1.9 second time limit.

The main while loop encompasses the MCTS algorithm, which operates in 4 basic steps as seen in the slides. First, "Selection", where the UCT policy is used to select the target node. Next, "Expansion", which is the step in which we make our most significant change from 'pure' MCTS. In lieu of expanding **all** of a node's valid states, we instaed only include a random assortment of nodes. The number of nodes expanded is proportional to the max_step value, which we know to be proportional to the size of the board. This was done to simulate the fact that larger boards should have more valid moves. If we expanded every single possible valid move, our algorithm would create excessively wide trees due to this game's high branching factor. Wide trees should be avoided because they reduce accuracy by heavily favoring exploration over exploitation.

Then, "Simulation" occurs, where BoardState's action-simulating functions are used on a node to execute random moves until a terminal state is reached. This function returns a score based on the result of the simulation, where 1 is for a win, 0.5 is for a draw, and 0 for a loss (returning the size of captured territory was also considered, but a better result was obtained using the simple 1-0.5-0 schema). Finally, the returned result of the random-rollout is used during "Backpropagation" to update all parent nodes up to the root of the tree.

Once the time limit of the loop is exceeded, the node which is an immediate child of the root node and which has the best UCT score among its siblings will be selected and returned by the step function to be played in the actual game.

2. Theoretical Basis

The main structure of the algorithm was based on the MCTS as discussed in class (lecture 9). This algorithm is effective when the evaluation function is challenging to create as there is no need for one; it simply repeats these four steps until the time limit is reached:

1. Selection:
   Use the UCT policy on nodes that are already stored in the tree to select the best path. UCT ensures the selected path balances exploitation and exploration.

2. Expansion: Once a leaf node is reached, expand the node. Here, we proceed with the strategy mentioned in slide 65: we reduce the amount of nodes to be expanded

due to the large branching factor. When a tree has a large branching factor, the algorithm would spend far more time exploring than it would exploiting, even with UCT. Thus, this reduction was necessary to ensure that this balance is maintained and that computational time is spent efficiently.

3. Simulation:
   From one of the expanded nodes, perform a random-move simulation (also called a random roll-out) of an entire game until the terminal state is reached.

4. Backpropagation:
   Once the terminal state is reached, update the value and visit count of the current node and all of its predecessors.

While our main guide was the class slides, we also consulted code from Baeldung (`https://www.baeldung.com/java-monte-carlo-tree-search`) to clear any confusions.

3. Advantages & Disadvantages

- Advantages:
  - As briefly mentioned before, MCTS is simple, effective, and adjustable. Thus, we were swiftly able to get the plain version working, and tailoring the algorithm to fit this specific project was also fairly simple. Further modifications would not have been extremely difficult to implement either, if time had permitted.
  - Acceptable results were still reachable without having to formulate a sophisticated and complicated evaluation function.
  - Accommodates any board size due to our node-expansion function. Being able to decide the subset-size for the expansion prevented the MCTS tree from growing excessively large, ensuring that it conforms to the memory limit while retrieving desirable results rapidly.
  - MCTS can perform well against Minimax algorithm opponents, since these rely on the assumption that the opponent player is playing optimally according to the same evaluation function. In our case, close-to-random moves can occasionally be selected (although these are never immediately fatal, such as trapping oneself into a box), which can confuse such opponents, especially those with sub optimal evaluation functions.

- Disadvantages:
  - By far the biggest disadvantage of our algorithm is that we expand nodes only with a random subset of all possible children. This does have its own benefits (as mentioned above) and the algorithm still yields consistently good results, but it may also lead to failure; It is possible that the algorithm misses instant-victory states and it may collect only sub-optimal or identical moves, wasting the limited computation time. This is what causes the roughly 5% failure rate against the random agent, and will likely cause student-agent losses.

- As discussed in class, many effective MCTS algorithms greatly benefit from a common-move database to improve run-times. Unfortunately, this was not possible for this project, which comes at our disadvantage.
- The algorithm does not capitalize on the 30-second grace period for the first move of the game. This time could have been used to get a hyper-specific, accurate estimation for the first move (although we figured this was not crucial as the first move in this game is not as important as it is in other games such as chess).

4. Other Approaches Used

This was briefly mentioned above, and although we had not gotten very far with the actual implementation, there was an attempt to approach this project through iterative deepening Minimax with Alpha-Beta pruning. With a proper, highly accurate evaluation function it would have outperformed under the specific circumstances of the project, as MCTS needs a lot of time and space to perform optimal or close to optimal.

5. Potential Improvements

Currently, our algorithm only expands a subset of states for each node to save on time and memory. However, since this subset is chosen completely randomly, an evaluation function used to refine the selection of the random moves would be a worthwhile upgrade; we could select only the best-looking random nodes, whose evaluated scores lie above a certain threshold. Moreover, this evaluation function could also speed up the random roll-out process by introducing cutoffs, where simulations or expansions do not go all the way to the end and instead return an evaluation function defined estimate. Obviously, the effectiveness of these upgrades greatly depends on the evaluation function's accuracy.

If we were allowed to read from a file, it may also be very helpful to create a database of the advantageous initial moves and commonly-used states over multiple run times (much like how AlphaGo operates, as seen in class). This way, we would theoretically perform better with less computation. However, it is worth mentioning that this would not be as effective as in games like chess, due to the fact that the early moves in this game are not as crucial (aside from an instant failure/victory like locking oneself / the opponent into a 1x1 box, which is seldom possible at the very beginning of a game).

Finally, there are other MCTS-specific techniques seen in class we could have implemented. Chief among these is RAVE, where the ability to store and share already-computed states across multiple simulations would progressively speed up the AI as it continues to run.