COMP 424 Final Project Report: *Colosseum Survival!*

Authors: Andre Tandoc and Eric Gong
Course Instructor: David Meger
Due Date: Dec 5th, 2022, 8:59PM EST

1. Technical Approach

1.1 General Explanation of Our Agent

Our agent uses the MCTS (Monte Carlo tree search) algorithm as a gameplaying strategy. We set a maximum computing time(under tournament constraints) to run our simulations and a flag that determines whether or not we are setting up, in which case we give more time. During the alloted time, we run as many simulations as we can from our initial state that we want to find the best move for, and return the total amount of wins/runs recorded for each play from our current position. We then calculate the best possible play for our agent at that time by choosing the play with the best win rate and return it.

1.2 Monte Carlo Tree Search

Our simulation function run_sim() first instantiates a deepcopy of the current board state so as to not modify the original board. This would be our tree. We also keep a set of visited_states to keep track of nodes we have already visited in the tree and the play counts locally for optimization. We then start running simulations until the maximum amount of actions is reached.

We sample a list of possible random actions from our current state and then store both the action and the state into another iterable. From here, we begin the selection phase and check: if the current state is not a leaf node, we move to the next state that maximizes the UCT formula(here the constant is set to 1.4). If the current state is a leaf node that has already been visited, we add a new state(node) for each action from our current state(expansion phase), set the current state to the first new child node, then perform a rollout. Otherwise, if the current state is a leaf node that has not been visited yet, we immediately rollout. Our rollout is just a random simulation of legal moves.

After choosing a node, we simulate the move, record the state of the chessboard and mark the node as visited. check_endgame() checks if the action is a winning move, and sets our player as a winner if True. The algorithm is configured to be limited to a maximum of 30 moves, and a time limit of 1 second, or 15 seconds for setup.

After tree traversal is finished, we back-propagate through the tree, sum the total amount of plays and wins and update their values.

2. Motivation for Technical Approach

Our inspiration for this approach came from analyzing AI strategies used for *perfect information* games. That is, games where no information is hidden from the other player and games where no luck, chance or randomness is involved. Due to the deterministic nature of the game, a tree, given unlimited memory and time, can be constructed that contains all possibilities. Finding the absolute best play for any given situation is only a matter of searching through the tree, as it will grow in memory and converge towards the actual optimal play. However, this is unrealistic in the real world. We needed to find a strategy that can run in reasonable time and compute the best possible play given the constraints. In this class, we covered many ideas that allow us to figure out what to do when the full tree could not be computed.

This led us to choosing MCTS as our game AI technique. This is primarily due to its' performance in games with a high branching factor(high number of possible moves each turn). Compared to other strategies such as minimax or alpha-beta pruning, we can limit how much/how far we want to search our tree. Minimax is too impractical, and alpha-beta pruning requires prior information. Furthermore, the basic implementation of MCTS is easy to build and improve upon as the branching factor gets higher. It is also very adaptable to constraints(such as time limits in tournaments).

2.1 Theoretical Basis: UCT (Upper Confidence Bound applied to Trees)

The tree policy for our MCTS algorithm is UCT. This strategy is ideal for a game where we must try many different options, but weigh the best option more heavily. UCT constructs statistical confidence intervals that try to balance exploration vs. exploitation. That is, we choose the action with the highest upper bound, and the more we pick it, the narrower the interval becomes. This means that we can then explore the other actions that have not been visited as much in case there lies a better option. Our regret(difference between theoretical best vs. expected) will only grow as O(ln n). Here, we set the constant to 1.3 to favor exploitation over exploration. This is due to not having much computation time in between turns, so we play it safe and stick to known good moves we've already discovered.

3. Advantages and Disadvantages

The advantages of our approach with Monte Carlo tree search are mostly on time constraints as the search is limited in a better way than the other two which have to either rely on evaluation functions to limit the depth of search or have to calculate the whole search space and then search in it. Having a sort of random element also helps in exploration when the game isn't "solved" like in the mini-max and alpha-beta pruning searches as the game can have many situations with local maxima and minimums (i.e. there's a good move that is far search wise that is more optimal but not easily representable by an evaluation function).

The disadvantages of this approach are obviously that if the random process cannot be completed adequately or is not run for long enough given the time constraints, our moves are not optimal. This is a worry in the beginning part of the game where search spaces are at their highest. There is also the part where we have to tune the method a bit with some variables such as search depth and how we value exploration versus exploitation. This is highly variable for different board sizes and random start positions which can be detrimental given our time constraints.

4. Other Approaches

For other approaches, we first tried mini-max with no evaluation function. This was done by creating a few helper functions to help the main step() function in student_agent. The first was a function to search for valid available moves called get_moves(). Then two functions max() and min(), one to calculate the move with the minimum minimax value and another for the maximum minimax value. And finally, we had a function to determine if the game ended. Inside step(), we would recursively call max() and min() for every valid move found by get_moves() to search through the search space. Once at leaf nodes, max() or min() would return the winner and compare recursively with the appropriate values. This approach was not very good given the project requirements as it took very long to calculate the optimal move at each step. This is because the branching factor and search space are pretty huge. Every turn, we have to consider all the tiles on the board reachable within the max step number times the number of possible barriers that can still be placed. Doing this for boards sizes bigger than 5 took way too long so we tried some other approach.

This time, we created some evaluation functions so that we could limit the search depth. This was hard as the game is pretty random and thus heuristics are hard to come up with. So one such function we tried was to have the number of tiles reachable at any turn be our evaluation function. We would have an extra helper function that searches for every reachable tile and return this number when max() and min() alternated 4 times (arbitrary number taken from slide 13 of L5-Games2.pdf). This makes it so that we "look" 4 moves into the future and then evaluate the position. This however didn't work really well with large board sizes and still took an unacceptable amount of time.

Another evaluation function we tried was to calculate how close each move would bring our agent to the opponent. The intuition here being that we want to maximize the amount of space "behind" our agent after every move and so we should be moving "towards" the adversary. This would be implemented with a simple helper function that calculated the distance in tiles between the indexes of the position of both agents (which is simpler and quicker to calculate than the previous evaluation function). This seemed to work faster than our previous evaluation function but still didn't calculate moves fast enough. One thing this method also tends to do is box itself in for some situations and so not optimal all the time.

5. Future Improvements

As for future improvements, we considered having more and better-tuned evaluation functions so that we don't rely on calculating deeper in the search tree as much to solve the time requirements. We also considered analyzing the game more so that we could maybe come up with optimal opening moves so that we can get to the latter stages of the game quicker where calculations and search space are smaller due to the board being filled up leading to less amount of possible moves. If the requirements were relaxed a bit, we could introduce multi-threading, memory to do some learning and more developed heuristics on game positions such as common patterns that happen (for example, recognize when it is possible to box in the opponent and when it is beneficial to do so). As an extra, we also considered doing alpha-beta pruning with our previous mini-max to see how the run-time would be affected as it was one of the main points of contention and the reason we did not use that method.