Monte-Carlo Tree Search for Colosseum Survival

Submitted by: Kevin Khouri (260761494) and Kenzy Abdel Malek (260637567)
Submitted on: April 12th, 2022

1. Introduction

Colosseum Survival is a two player, fully-observable, deterministic, zero-sum game in which two players on an $M \times M$ chess board take turns trying to move a certain amount of steps per turn and place a wall along the borders of tiles in order to partition the board. The game ends when both players are separated by a continuous wall spanning two opposite sides of the board. The player who finds themselves inside the partition with the most tiles wins the game. Given Colosseum Survival's characteristics, we explore possible AI strategies for our agent before settling on a Monte-Carlo Tree Search (MCTS) approach. We first justify our choice of MCTS by studying theoretical foundations of the algorithm. We then describe our initial agent and improvements which were made to it, resulting in our final MCTS agent. Finally, we propose future improvements that could be made in order to improve efficiency and hope that our agent performs well in the tournament!

2. Choosing an approach

In deciding which approach between Minimax and MCTS to choose, we began by considering the branching factor of the game along with the memory and time restrictions imposed by the assignment. For chess boards of size 6×6 to 12×12, if an agent is placed in the middle with no other wall or opponent in the board (this situation leads to the highest number of possible moves an agent would have to consider), then the number of possible wall placements (i.e. moves, which consist of moving to a tile and placing a wall at one of its borders) ranges from 84 (in the 6×6 case), up to 324 (in the 12×12 case). If we were to implement a basic version of Minimax, then a node in the game tree would need to at least store the chess board representation, pointers to children nodes, and utility values. Assuming the chess board is of the largest allowable board size, it would have to be represented by a $12 \times 12 \times 4$ array of boolean values, therefore requiring a total of 576 bits. Moreover, each pointer in Python requires 32 bits (or possibly 64) so storing all children of a node would require at least $(324)(32) = 10'368$ bits. Finally, the utility value could be stored as a single boolean. Therefore, a single node would require a total of $10'945$ bits. From this, it follows that the number of bits required for the first few depths of a Minimax game tree would be:

Depth 0 : 10945 bits (for the root node)
Depth 1 : $(324)(10'945) = 3'546'180$ bits (for all the root node's children)
Depth 2 : $(324)(324)(10'945) = 1.15 \times 10^9$ bits

Thus, we can see that storing a game tree of depth as little as 2 would already exceed the memory limit, so a Minimax agent would only be able to think up to one move ahead and the problem lies in storing all children of a node. Since MCTS can more easily avoid such explicit generation of all possible moves per state of the game, doesn't require an evaluation function, and has shown success recently in games with high branching factor such as Go [1], we decided a basic MCTS implementation would be a good base to start with.

3. Theoretical foundations of Monte-Carlo Tree Search

The most basic version of Monte-Carlo Tree Search is an adversarial search algorithm which relies on random simulations of the game until termination, given a certain state, in order to decide on the most promising action to take next. A key aspect to note about this algorithm is that each node in the search tree is given a value, which is based on its win rate (the number of times taking an action at a given state lead to a win, taking into account simulation results). Monte-Carlo Tree Search consists of four phases, which are repeated for a fixed number of iterations (or until time runs out).

i) Selection: starting from the root of the game tree, keep selecting the next successor state greedily with respect to some selection policy until a leaf node is reached. A good policy allows for a balance between exploration and exploitation, so as not to miss out on seemingly less promising nodes in the beginning, which might actually lead to a better result once played out. A standard tree policy is the UCB value:

$$Q^{\star}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}. \tag{1}$$

Where s is a state, a is an action that can be taken at that sate, c is some constant, $n(s)$ is the number of times state s has been visited and $n(s,a)$ is the number of times action a was taken in state s.

ii) Expansion: at this point, a leaf node has been reached. This phase therefore expands the leaf node by generating a child for each available action. Children can be generated with respect to any order.

iii) Simulation: after a node has been expanded and one of its child nodes has been visited, the algorithm now wants to rate that action and it does so by simulating the rest of the game. Various approaches (playout/default policies) exist for simulating the rest of the game, but random actions are often the best way to go. Moreover, one does not have to equally distribute the number of simulations amongst the child nodes. In other words, there can be more simulations allocated to more promising moves.

iv) Back-propagation: At this point, simulations are done and the algorithm must update the values for nodes visited during Selection and Expansion so that they reflect each action's win rate.

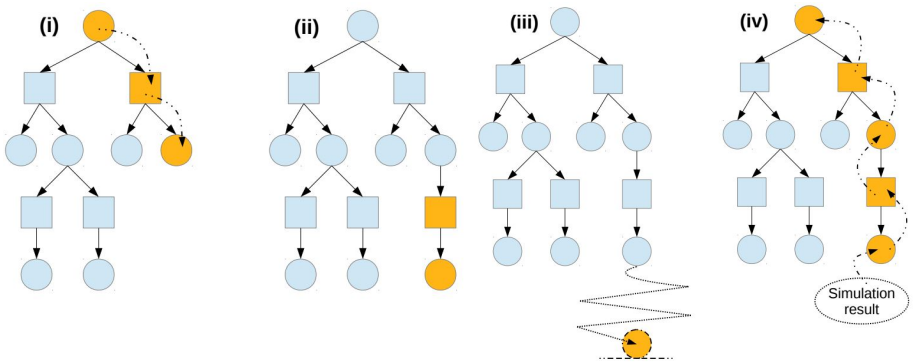

Figure 1: The four stages of MCTS illustrated. Orange is used to represent parts of the tree that are modified or visited at each stage. Image source: [2]

4. Our Agent: Technical Approach

In order to implement a successful agent to play Colosseum Survival, we started by implementing a basic Monte-Carlo Tree Search approach as an initial baseline.

4.1 Implementation of our Initial Baseline MCTS Agent (Agent A)

Implementing the first version of our MCTS agent required two data structures:

1) A class `Node` in order to represent a node in the search tree. Each node has the following attributes:

   - A copy of the chess board for the state the node represents (in order to keep track of the wall positions).
   - Our agent's position in this state, and the adversary's position.
   - A boolean representing whose turn it is to move in this state.
   - A boolean representing if this node represents a terminal state.
   - A double representing the ratio of wins/playouts.
   - A pointer to its parent node.
   - Pointers to each of its children nodes.

   Methods which can be called by a `Node` are:

   - `createChildrenNodes()`, which creates a child node for all possible moves from the current state this node represents. Note that this method ended up being replaced in order to save on memory (this will be discussed shortly).
   - `isTerminal()`, which returns a true or false depending on if the node which called it is terminal or not and returns the score for both players.
   - `UCB1()`, which returns the UCB value for the node, calculated according to equation 1.

2) A class `SearchTree` in order to represent the search tree. A search tree has the following attribute:

   - A `Node` object representing the root of the tree.

   Methods which can be called by a `SearchTree` are:

   - `select()`, which selects and returns a node in the search tree to be expanded.
   - `expand(leaf)`, which expands the node returned by the select() method by generating all possible children nodes using a call to createChildrenNodes(), then returns a random child node.
   - `simulate(node)`, which simulates a full game between two random agents from the current state represented by `node` and returns the result (1 if our agent wins, 0 otherwise)
   - `backPropagate(result, node)`, which, given the result returned by the simulation, will back-propagate it up the search tree while updating the win/playout ratio for all nodes.

3) A class `StudentAgent` in order to actually play a game. This class is initialized with a SearchTree, which is created when StudentAgent is first initialized. When our agent's step() function is called for the first time, a Node object which represents the current state of the game is created and assigned as the root of the SearchTree. In order to avoid losing the information contained in the SearchTree between subsequent calls to the step() method, but to minimize memory usage, the SearchTree's root is updated by assigning it to one of the children of the current root (i.e. the current state the game is in). The parent pointer of this new root (which points to the old root) is set to None, and the remaining sub-trees are deleted by the garbage collector as illustrated in Figure 2

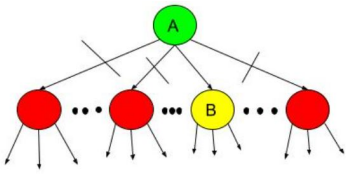

Figure 2: Subsequent calls to step() changes the SearchTree root to B, a node that matches the current state the agent is in when step() is called. The old root, A, and all other sub-trees (red nodes) are deleted.

With these data structures, our agent's `step()` function simply calls the MCTS algorithm, and then returns the result. Below is the pseudo-code of our MCTS agent:

```
def Monte_Carlo_Tree_Search():
    while(time is remaining):
        leaf = searchTree.select()
        child = searchTree.expand(leaf)
        result = searchTree.simulate(child)
        searchTree.backPropagate(result, child)
    return child of root with the maximum UCB1 value
```

An initial problem with the base implementation is that when a node was expanded, a new child node was created for every possible legal move that could be made from the current node, which took too much memory. As a compromise, we implemented a `MoveSet` data structure so that each node could store its set of unused legal moves in its own MoveSet. When a node is expanded, a call to `createChildNode()` replaces the call to `createChildrenNodes()`. A random move from the MoveSet is removed and a new node representing this move is generated. This implementation was more memory efficient than creating a child node for every possible move immediately during expansion because a single move in the MoveSet contained only the position and wall direction placement information for each move, which required less memory than a Node object. As such, a MoveSet was a simple wrapper for a Numpy array that contained the functions to append a move to the MoveSet, pop a random move and check if the MoveSet is empty. Finally, we experimented more with various data structures and found that a Numpy array was the most efficient way to store the (x, y) coordinates and wall direction of a move in the MoveSet. In other words, using a list, set or array in Python were not sufficient, as can be seen in Figure 3.

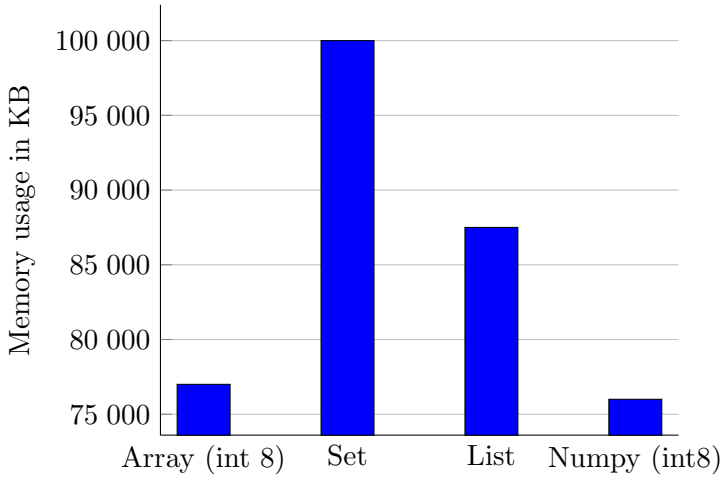

Figure 3: Peak memory used in the entire process during an autoplay of 10 games with the different built-in or library data structures used in Python to represent a MoveSet for each Node.

This finalizes our initial MCTS agent for Colosseum Survival! We refer to this agent as **Agent A**.

4.2 Experimentation to Improve Agent A and Results That Lead to Our Final Agent (Agent B)

After creating our baseline implementation, Agent A, we conducted experiments to test the performance of different heuristics and techniques against Agent A. We call **Agent B** the agent which is identical to Agent A but with these heuristics and new techniques applied.

4.2.1 Experiment 1: checking if a state is terminal during simulation

During the simulation phase with Agent A, a call to a function `isTerminalState()`, which uses a union-find algorithm, is made in order to determine if the game is over. Because of the union-find, testing if a state is terminal is very time consuming. We therefore tried to optimize this step by checking if a game is over only after **both** players move in the simulation, instead of after every player's turn. We were able to see this with the following reasoning: if the game ended and say player X won, then if player Y played an additional legal move, it wouldn't affect the tiles that X owns, nor would it increase the number of tiles Y owns; Y would still lose. Now if player X played an additional move, it is unlikely (but possible) that they would lower the number of tiles they own to be below the number of tiles Y owns. Thus, during a simulation, it is only ever necessary to check the state of the game after the adversary has played. Applying this to Agent B, and playing 1000 games between Agent A and Agent B, resulted in a win rate of 54.4% for Agent B. Moreover, Agent B performed on average 38.6% more simulations per turn over 50 turns. Hence, although reducing the stages at which the agent checks if a node is terminal increased our agent's win rate in small amounts, we incorporated this technique into our Agent B.

4.2.2 Experiment 2: Early Playout Termination (EPT) during simulation

A noticeable bottleneck in our agent's implementation was running the simulations. Simulated games were observed to take up to 40 turns in some cases. Because of this, we experimented with implementing MCTS with Early Playout Termination (EPT) and an evaluation function. The evaluation function takes input an incomplete game state and returns the ratio of the number of reachable tiles from our agent over those reachable from the adversary, normalized between 0 and 1. Thus, the more restricted the adversary's agent is compared to our agent, the more likely the current state in the simulation is to lead to a win for our agent, and the higher the return value. Figure 4 shows results of 1000 simulations between Agent A and Agent B, where Agent B now also uses EPT defined by $N =$ maximum number of moves per simulation, with the corresponding simulation cutoff limit used. No significant benefit was observed, so we did not incorporate this feature into Agent B.

4.2.3 Experiment 3: bidirectional search instead of union-find

From the results of Experiment 1, the increase in average number of simulations confirmed that checking if a state is terminal is an expensive operation. Our agent does such a check not only during simulation (after both players have moved, as implemented after Experiment 1), but also every time a node in the search tree is generated. The original algorithm for this check used a union find. In this experiment, we re-implemented the check to instead use a

N	Win rate
4	48.1%
5	49.7%
6	49.0%
7	50.7%

Figure 4: Win rates corresponding to N, the maximum number of moves per simulation.

version of bidirectional search. Applying this to Agent B and simulating 1000 games against Agent A resulted in a win rate of 53.4% for Agent B. Agent B also performed on average 190.6% more simulations per turn, over 50 turns. Although the win rate is insignificant in comparison to the heuristic used in Section 4.2.1, the increase in the number of simulations was enough to incorporate this into Agent B.

As a final result, **our official agent to play in the tournament is Agent B**, which uses bidirectional search instead of union-find to check if a state is terminal, and during a playout/simulation phase in MCTS only checks if the playout is over after the random agent representing the adversary has played.

5. Our Final Agent: Strengths, Weaknesses and Future Improvements

Our final Monte-Carlo Tree Search agent for playing Colosseum Survival performs well overall. According to the changes and experiments made throughout the design process, it wins against a random agent almost every time and performs adequately against a standard MCTS agent such as Agent A. One weakness however is that it is susceptible to missing moves that would be "obviously" good to a human or agent with a good evaluation function because of the non-deterministic nature of expanding nodes and simulating games. Moreover, a high number of simulations is required to converge to the optimal play, a key restriction. An advantage of our agent is that it does not rely on any heuristic evaluation function which would require use of expert knowledge of the game. A better heuristic than what was used for EPT during our experiments could potentially lead to future improvements and would be worthwhile exploring to make EPT show improvement in our agent. Another improvement could stem from biasing moves that put the agent in the centre of the board, or push the adversary into a corner. This bias could be applied by modifying the UCB1 function used during the selection phase of MCTS to also incorporate an additional term that ranks the relative value of states according to each player's current position. That is, the selection phase would consider exploration, exploitation, and a third term for positional advantage when selecting nodes. Finally, one could also consider incorporating Rapid Action Value Estimate (RAVE) into our agent. Most importantly, a good scientific approach and experimentation will be needed in order to see if this is worth incorporating, as RAVE heuristics have been shown to lead to a lower win rates when compared to MCTS agents.

References

[1] Guillaume Chaslot, Sander Bakkes, Istvan Szita and Pieter Spronck. "Monte-Carlo Tree Search: A New Framework for Game AI". *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. Universiteit Maastricht, 2008. https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf

[2] Adrien Couetoux. "Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems". *Data Structures and Algorithms*. Université Paris Sud - Paris XI, 2013. https://tel.archives-ouvertes.fr/tel-00927252/document