Monte Carlo Search Tree with Killer Moves 🕺

Castro Obando, Sebastian
sebastian.castroobando@mail.mcgill.ca

Chaturvedi, Aryan
aryan.chaturvedi@mail.mcgill.ca
260976059

COMP 424 Fall 2023

1. Introduction

To win in "Colosseum Survival!", a player must anticipate future moves and predict control of areas. Implementing these intuitions is challenging, because it heavily depends on our grasp of the game and the time invested in creating winning strategies. Therefore, from the beginning, we wanted to adopt a strategy to outperform a random agent without relying on any sophisticated heuristics. This led us to choose the Monte Carlo Tree Search (MCTS) approach. The idea boils down to running as many inexpensive simulations as possible (within the two-second allowed) and let our agent compute which path seems the most promising.

We were impressed by the effectiveness of a basic MCTS implementation. Without any strategy beyond game simulations, we consistently outperformed the random agent. To make the agent better, we decided to implement the "killer moves" strategy, which consists in checking two parts. First, we check for immediate winning opportunities by confining the opponent to a single-cell square or preventing our agent from being trapped similarly. Second, limit the search tree by looking (in our case expanding) at the plays that result in more actions for our agents afterwards. With these killer moves approach we successfully defeated the random agent with a perfect win rate.

The simpleness of MCTS is a double-edged sword, and this becomes apparent playing against a human. Our agent can successfully survive, but it fails to look at the bigger picture which is area control. This will be discussed in a later section.

2. Design

2.1 Basic Structure

The main ideas behind the Monte Carlo Search Tree (MCTS) were discussed in class, but not its implementation. One of us had previously implemented MCTS for a tic-tac-toe game, so we decided to use the tutorial[1] for the tic-tac-toe agent as the base for our implementation. This tutorial gives the generic functions of MCTS and leaves the implementation logic to the reader.

From a high-level perspective, our agent's step function is called, which then creates a Monte Carlo Search Tree node. This node runs the algorithm and stops before reaching a time limit threshold. From our testing, one iteration of the algorithm can take up to 0.1 seconds, so this was chosen as the threshold. In addition to the MCTS algorithm, we implemented the concept of killer moves: before running the first iteration of MCTS, we check if we can win the game by trapping the opponent. This logic is also applied during simulations, basically returning 1 if the simulation results in our agent having a killer move, or -1 if the opponent does.

2.2 Monte Carlo Search Tree

We divided the MCTS agent into 2 objects (not counting the student agent object): *State* and *MonteCarloSearchTreeNode*. This separation is necessary because these objects encapsulate different functions. The *State* object tells us information about the state of the board, and the *MonteCarloSearchTreeNode* object performs the selection, expansion, simulation, and backpropagation.

We assume that the reader is familiar with the general ideas behind MCTS, but we want to give some basic ideas behind our implementation. Winning states return 1, losing states return -1, and tie states return 0. For the best child selection, we used the generic Upper Confidence Bound applied to Trees (UCT) formula:

$$UCT = \frac{w_i}{n_i} + C * \sqrt{\frac{\ln N_i}{n_i}}$$

When running the algorithm, we use a default c value of 1.4 as this was shown in class to be a reasonable value in theory. At the end of the 2 seconds (minus the 0.1 seconds threshold), the exploration phase is completely over and we want to return the best possible child. This is why we call the best action function with an exploration parameter of 0. So far, our code follows closely the basic implementation of the MCTS. However, we quickly noticed it was not enough to guarantee a 100% win rate against the random agent. So we will turn our attention to the expansion and simulation phases of the algorithm.

2.3 Designing Decisions for Expansion and Simulation.

The first challenges we faced while designing the agent were the implementation of the simulation and the tree expansion. From our understanding of the lectures, the simulation should be an inexpensive operation, and we

[1] Monte Carlo Tree Search (MCTS) algorithm tutorial and it's explanation with Python code.

want to run as many of them as possible within the allowed time. Our current implementation of the simulation involves picking random actions until the game ends, this makes the code simple and has the advantage of running fast. However, these simulations would be more helpful if we could guide them towards the promising paths. Thus, we needed to find a way to keep the simulation as cheap as possible while also implementing a way to prioritize certain moves.

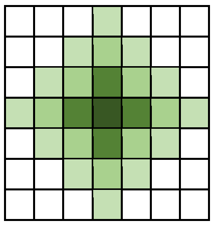
While doing research, we realized that one of the main strategies[2] for MCTS should be to avoid actions which should not be explored. What actions can we safely avoid in "*Colosseum Survival!*"? One obvious decision is to compute the number of moves our agent would have after a play. The intuition here is that expansions which lead to a lower number of possible moves for our agent will probably not yield into a winning state. This idea will make our agent run simulation from the expansion we judge to be more informative. To achieve this "informed" expansion while keeping a running time as low as possible, we couldn't use the function that gets possible moves since it performed a deep copy on every call. So we implemented a lightweight version of the function, which still uses BFS for the possible moves, but instead of performing the move it simulates the moves and goes back to the original state. Once we had a way to order the expansions, we just ordered the list of possible moves according to our strategy, including a flag so we don't run the ordering again if it was already computed.

This implementation did not perform as well as we thought. Even though we ordered the expansion list, this made little difference on the simulation because many of them were able to run and give us uninformative data. Thus, we decided to limit the expansion. This is basically telling our algorithm to go "deeper" into the tree rather than "wider". We decided to limit the tree expansion to the top 20 results by testing different numbers against the random agent. The limit of 20 seemed to perform the best against the random agent because it gave the win rate of 100% in the shortest time. This will be discussed further in the quantitative analysis.

3. Quantitative Analysis

Theoretical Breadth and Depth of Game State Tree

Breadth: The breadth of our search tree can be modelled by the number of actions we can take in a turn. Of course, this number varies with the current state of the game, but it's largely affected by the maximum number of steps we can take, or k. By visualizing the range limited to k steps, we get this figure:



[2] Pitfalls and Solutions When Using Monte Carlo Tree Search for Strategy and Tactical Games [page 349]

Commented [1]: Analyze your agent's quantitative performance using criteria we mentioned in class. For each, state a numerical quantity or formula and give a 3-4 line text explanation (overall 1 page):

1. What depth (a.k.a. look-ahead) level does your agent achieve in the size 12 board? Is it the same for all branches, or deeper in some cases than others?

2. What breadth does your agent achieve in the size 12 board? That is, how many moves do you consider at each level of play? Is this the same or different for the max and min player?

3. How do the above factors (depth and breadth) scale with board size. Attempt to give a rough big-Oh notation and explain.

4. What were the impact of any heuristics, such as evaluation functions and move ordering? Compare the breadth and depth achievable with and without these factors.

5. Predict your win-rates in the evaluation, against: (i) The random agent, (ii) Dave (an average human player), and (iii) your classmates' agents.

Commented [2]: part 5 done : ☑ @aryanchaturvedi075@gmail.com let me know if you think this is good. :)

For the illustration above, $k = 3$ and $m = 7$, which creates a diagonal square perimeter that gets larger with each k, so the number of blocks can be simplified to $1 + 4(1 + 2 + \dots. + k)$, which gives an arithmetic series given by $1 + 4\left(\frac{k(k+1)}{2}\right) = 2k^2 + 2k + 1$. For each block, there's 4 possible walls that can be placed, therefore given k, the breadth of the tree without any pruning would be $8k^2 + 8k + 4$. The branching factors on a 6 x 6 board and a 12 x 12 board would be 100 and 340 respectively as an upper bound.

Depth: The depth of a single branch from the starting point can loosely modelled depending on the number of available walls, which depends on the size of the board, or m. Since each block on the board would have 4 sides, but also almost every wall is share by 2 blocks, and there's m^2 blocks, we could say there's $2m^2$ number of walls. A more accurate measurement can be derived from the patterns in visually counting the number of walls, excluding the edges on the border of the board itself, which can simply be written as $2m(m-1)$. The depth of the tree on a 6 x 6 board and a 12 x 12 board would be 60 and 264 respectively as an upper bound.

Simply put, B = $O(k^2)$ and D = $O(m^2)$

3.1 Look Ahead in size 12 board.

Since we decided to implement a Monte Carlo Tree Structure, it is difficult to measure the depth of computation our agent went to during the light rollout stages, and this is because it is a random move simulation, which is bound to vary drastically, but in the worst case would have the previously mentioned maximum depth of 264 on a 12 x 12 board assuming a wall was built on every available location, but only with a very small probability.

Outside of the rollout, and looking just at the Fully expanded tree, our agent was able to accurately model until 2 moves ahead. Since we implemented a UCT expansion procedure, given in the formula $\text{UCT} = \frac{w_i}{n_i} + 1.4\sqrt{\frac{lnN_i}{n_i}}$, our exploitation component has a range of [-1, 1], since w_i only taken a win as + 1 and loss as -1, but our exploitation component can easily go beyond 1. This would mean that after every grandchild explored, the exploration components of the nodes 1 level prior will have much more weightage and thus also be explored. Since our agent never went past a depth of 2, we can conclude that the depth of the Monte Carlo Tree was the same one each branch. This can be verified even on the 6 x 6 board where our agent yet again achieves a maximum look ahead of 2 moves. We also had an alternate strategy for the value of w_i, which is further discussed in **Section 6: Future Improvements**

3.2 Breadth achieved in size 12 board.

Knowing that the board's branching factor was between 100 and 340, we knew we had to largely prune down our breadth. We decided to cut down the list of children down to 20 of the most promising moves. We defined "promising" based on how many moves could be played from that child state, which involved a Breadth-First Search from the child state. This was a greedy heuristic which prioritized moves that lead us to more open space and accounted for the adversary having the same priority. Thus, we decided our breadth would be constant at each level, at 20 of the most promising moves.

3.3 Theoretical Performance of Agent

This section is combined with the impact of heuristic. As mentioned earlier, our branching factor was reduced from $O(k^2)$ down to a just 20 due to the implementation of the Move ordering and selection heuristic. This helped reduced the number of simulations run on every child node, because on a 12 x 12 board, with purely random simulations, our agent was not even able to explore all the possible moves, but a fraction of it, at 60 nodes. We selected 20 as a hard limit since we wanted to explore at least a couple of the children of the most promising nodes to have more accurate data. Perhaps another improvement we could have done was to alter the n_i within the exploration component to be weighted based on ordering in the queue of children. So our agent wouldn't try to explore all the children, but rather keep a frontier

The number of 60 nodes was measured by counting the number of times a MCTS Node was instantiated during the 2 second run. We realised this number varies drastically based on the size of the board, i.e., our agent instantiated around 600 MCTS Nodes on a 6 x 6 board. The reason for this became obvious as we had another issue glaring at us, which was the depth of each of these simulations. On a 6 x 6 board, thanks to our killer move heuristic our light rollouts ran for a short period of time. However, this was not the case on a 12 x 12 board. Given that the maximum achievable depth is 264 in contrast to just 60, and that killer moves heuristic were a lot less frequently applied given that there was plenty of time before opponents start running into region with multiple times, each light rollout ran for a lot longer. Our killer move heuristic was a lot more effective on a 6 x 6 board than a 12 x 12 board, and thus our depth factor was mostly unaffected.

Overall, thanks to our heuristics, our *best_child()* function was improved from $O(k^2)$ down to constant time.
Other functions had the following runtimes:
Expand () = $O(k^4)$
Tree_policy () = (m^2) but with a call to the *Expand ()* function on creating a new node
Rollout = $O(m^2)$ but with *is_game_over()* function that is an inverse Ackermann function that grows very slowly
Backpropagation () = = $O(m^2)$
Each single iteration within the 2 seconds can be thought to run in $O(m^2k^4)$

3.4. Win rate prediction

In this section we will discuss our prediction on win rates against :

i. <u>Random agent</u> : After running auto play four times against the random agent, both starting and being second player, we are confident that we have a win rate of 100% against the random agent.

ii. <u>Human agent</u>: it depends on the strategy of the human player. Our agent struggles at "sensing" possible areas of controls. This becomes obvious when the winning strategy involves maneuvering towards corridors to close off large areas, which humans can usually anticipate. Based on our tests with friends and family, where we won 11 out of the 60 games, we estimate our win rate against humans to be between 10 and 20%. We think this win rate will drop to 0% against Professor David since he has some level of experience with this game.

iii. Classmates' agents: it's difficult to give a number because we did not have the opportunity to play against our classmates during the development of our agent. Given that we only implemented two of the five suggested strategies in the report, we expect to be at the lower end of the average. We estimate to win against 30% to 40% of the agents.

4 Strengths and Weaknesses of our Agent

4.1 Strengths

Our agent is by nature, quite defensive. In this order, it is always prioritized escaping when an opponent tries to closely trap our agent. Even against a human player, our opponent can appear to be quite slippery, especially if the opponent comes in close territory and then tries to trap our agent, it would very struggle to do so, so long as the agent places a wall within the range of maximum steps of our agent, it can sense areas that it would need to prioritise and escape to.

Another strength of our agent is its simplicity, and this simplicity is tremendous against a random agent, because it has minimal chances of loosing or a random Agent, close to 0. With such simplicity, our computation times are cheap, and thus we can map a lot of favourable states for ourselves. In another one of agents was able to look up to 5 moves ahead, specifically on a 6 x 6 board.

With the perception of walls, our agent is more effective on a smaller board than a larger one, because it can more easily sense traps which our being laid out close by, and thus going near that trap would automatically be pruned out in the breadth of children to explore, rather it would consider escape blocks to be of more priority and focus computations towards escaping states.

4.2 Weaknesses

One of the major problems that we were not able to solve was the fact that our killed move heuristic did not significantly reduce the depth factor of our random simulations. This became a big drawback in a 12 x 12 board where our tree itself was quite small and thus we did not explore enough points to make a sound judgement of what's a best move to play. The random simulations were meant to be cheap computations to replace the need for a depth limiting heuristic function, however the runtime became too expensive for the value of the result we obtained from our simulation.

A possible implementation of a state evaluation heuristic at the defined depth of the simulations would have been a very valuable reduction in computation time. For example, a hard limit of a depth of 20 moves during rollout and then returning a state evaluation prediction based on which player has the most reachable space as well as number of walls in the player's neighbourhood determines who would win or lose at the end of the simulation. Doing this would tremendously speed up our exploration rate since our branching factor is constant for all the board sizes, and our agent could look upwards of 5 moves ahead.

Another drawback of our implementation was that our agent tends to prioritize playing more defensively all the time and never makes an aggressive move unless an obvious killer move situation arises. It loses the ability to capitalise on the disadvantageous moves played by the opponent, for example, the opponent would be in a tunnel with a dead-end, but our agent wouldn't be able to capitalise on that and switch to a more aggressive play. Lastly, our agent also fails to recognise at times when there's an obvious, but larger scale trap being laid out. For example, with an agent that is trying to build a wall around then center and trap our agent to the smaller half, our agent cannot perceive that it is being trapped into a smaller half, simply because of the moves within its range, there's a lot of cells that are away from the center line, which appear to give more playable moves later.
Given the lack of aggression when needed, our player can lose quite easily to an agent with a simple Alpha-Beta implementation, that would prioritise aggressive play through its move ordering heuristic.

5 Previous Approaches

5.1 Alpha -Beta Pruning

Our initial approach considered Alpha Beta Pruning method with a state evaluation and move ordering heuristic. This approach would have been optimal but at the time we couldn't figure out a good state evaluation heuristic for a defensive style of play. In retrospect, an iterative deepening search along with the previously mentioned possible evaluation of which player has more reach moves and walls near them to determine a win or a lose could have just as fairly been implemented there. Although we are unsure how effective that would have been in pruning our tree and instead, we would have had to rely more on move ordering Heuristics, like the one's we ended up implementing, just without the need for shortening the list of children, since this would be similar to running a depth first search on the more promising nodes on the left. If done successfully, we would have been able to emulate more accurate results with an Alpha Beta pruning approach than we did with our Monte Carlo Approach. We never fully tested out this Approach.

5.2 Monte Carlo Tree with a Potential Field Planner

This idea was taken from the field of Robotics where potential field planners are used to avoid obstacles and get to a desired location. This would have required another grid, but which was only 2 dimensional, to model the "danger" level of each cell. The idea of this approach was also to prioritize being in the center of the grid towards the beginning of the game. Essentially, when the game would first start, we would iterate through the chess board, and for every wall we found, we would increment the "danger" level of cells in the surrounding radius based on the distance from where the most recently added wall was. So the immediate spot where the wall was added would have the highest increment of "danger" level an areas with the fewest walls would have the least level of danger level. Running this initially when the game would start would indicate the there was a higher associated threat level close to the edges, and the center was "safer" pushing the agent to build more walls around the center, and as more and more walls would be built, the potential field would start evening out and the agent would then start moving to other less explored areas of the board.
This seemed like a promising approach, simply because the update process would run in constant time, updating a 5 x 5 region near it, so at most 25 blocks would have been iterated over. Once that was done, we would once again choose the actions within 5 of the most promising blocks to run simulations on. While testing out this agent

on a 6 x 6 grid, we were able to achieve a depth of 5 moves within the tree itself, implying that this agent would have been more effective for the exploitation component. However, the downside, was that it considered the number of walls close to every cell and played extremely defensively, which wasn't always a good thing. This was evident when we ran this agent against our main agent, and with 100 simulations run, it had a win rate of about 13% on a 6 x 6 board and 4% on a 12 x 12 board.

6 Potential Improvements

There were many aspects of our agent that we could have vastly improved, mentioned throughout the report previously, and we'll try to incorporate those changes time permitting.

6.1 Developing reliable state evaluation heuristic to be used at a defined depth during Rollout

We would like to develop a good state evaluation heuristic to limit our rollout time to a certain depth and get a cheaply computed result. This would be crucial since that is the whole point of a rollout, it's not certain and it's not representative of how good the child state is. A state evaluation function could do the same task with roughly the same level of accuracy, but only with fasted overall computation times, and thus a larger expanded tree to give a better frame of reference to decide the best action.

6.2 Modifying UCT to perform more Exploitation than Exploration

We reduced our child states down to 20 of the most promising moves, we could have modified the UCT function, or implemented an altogether different function to evaluate which child node to explore before expanding, so that we don't have to run simulations on all the grandchildren of the child nodes. Moreover, within the UCT function itself, the weightage of exploration would easily be more than the weightage of the exploitation, and thus our agent was constantly exploring more moves than exploiting any sequence of moves that could lead to a winning state.

A great way to provide more weightage to our agent would be to make use of running lengthy simulations. As in, instead of just returning whether our agent won or lost, we could really use to our advantage by how much or how badly our agent lost. This could be done by simply returning the difference of scores between the 2 agents at the end of the simulations. A draw would lead to an increment of 0, while a win would increase it to a max of m^2 – 1, and vice versa for a loss. This way the simulation might also relay domain specific knowledge based on the intuition, that a state that resulted in a massive victory is likely to yield more promising states, and to completely avoid states where we lost poorly, since nearby states would also likely be bad for us. In the initial states $\frac{\pm m^2 - 1}{n_i}$ is more likely to be greater than [-1, 1] and would thus give more weightage to the exploitation component of UCT.

6.3 Developing an Aggression Heuristic

We could assignment a weightage to the number of moves each state opens up for our agent, as well as add an aggression heuristic that uses Manhattan distances and A* Search to assignment "aggression" value to a state/

cell and try to balance between to the 2 of them, rather than entirely depending upon the which blocks give us more room within our move ordering Heuristic.