# Comp 424 Final project

Shanzid Shaiham (260913075) and Doruk Taktakoglu (260909243)

April 12, 2022

## 1 Motivation

In this project, our agent implements a simple Monte Carlo search to play against the adversary. During the testing phase of the agent, we experimented with ideas of implementing a minimax search (with and without alpha-beta pruning), Monte Carlo Tree Search (with various heuristics), and reinforcement learning techniques. However, given the time and resource limitations, a simple Monte Carlo search performed the best for us.

In contrast to a Monte Carlo Tree Search, our simple Monte Carlo search does not require searching through a tree. Instead, at every step of the game, our agent simply explores all possible moves, simulates the game forward from each move, and picks the action which has the best chance of a win.

Each move is evaluated by running multiple random simulations, and the scores from each of the simulations are averaged. So, our agent is not very affected by a few bad simulations.

A further advantage comes from the fact that the agent does not need to explore all possible moves or simulate each game to a terminal state, i.e. it can avoid running too many simulations in order to respect the time constraint at each move. The simulations themselves are executed with completely random moves, so computational overhead is very little for generating these simulations.

## 2 Theory

The algorithm for our agent at each step is as follows

---

**Require:** step(chess_board, my_pos, adv_pos, max_step)
  $moves \leftarrow get\_possible\_moves(chess\_board, my\_pos, adv\_pos, max\_step)$
  $best\_move \leftarrow get\_best\_move(moves)$
  **return** *best_move*

---

Continued..

```
def get_best_move(self, possible_moves, chess_board, adv_pos, board_size, max_time):
    # for each move, calculate the score
    # return the move with the highest score
    best_move = possible_moves[0]
    best_score = 0
    time_start = time.time()
    max_time_per_simulation = max(max_time / len(possible_moves), 0.01)
    for move in possible_moves:
        if time.time() - time_start > max_time:
            break
        my_pos = move[0]
        barrier_vector = move[1]
        new_board = chess_board.copy()
        new_board[my_pos[0], my_pos[1], barrier_vector] = True
        total_score = 0
        runs = 0
        for i in range(20):
            if time.time() - time_start > max_time:
                break
            runs = runs + 1
            p1score, p0score = self.simulate(new_board, my_pos, adv_pos, board_size, max_time_per_simulation)
            if p1score > p0score:
                total_score += 1
            if p1score < p0score:
                total_score -= 1
            else:
                total_score += 0.5
        p0_score = total_score / runs
        if p0_score > best_score:
            best_score = p0_score
            best_move = move

    return best_move

def simulate(self, board, p0_pos, p1_pos, max_step, max_time):
    chess_board = board.copy()
    # check while the game is not over
    board_size = len(chess_board[0])
    is_end, p0_score, p1_score = self.check_endgame(board_size, p0_pos, p1_pos, chess_board)
    time_start = time.time()
    i = 0
    while (not is_end) and (time.time() - time_start < max_time):
        moves = self.getPossibleMoves(chess_board, p1_pos, p0_pos, max_step)
        # get random move
        if len(moves) == 0:
            break
        random_move = moves[np.random.randint(len(moves))]
        # apply random move to chess_board
        chess_board[random_move[0][0], random_move[0][1], random_move[1]] = True
        # check while the game is not over
        if i % 2 == 0:
            is_end, p0_score, p1_score = self.check_endgame(board_size, p0_pos, p1_pos, chess_board)
        else:
            is_end, p0_score, p1_score = self.check_endgame(board_size, p1_pos, p0_pos, chess_board)
        p0_pos = p1_pos
        p1_pos = random_move[0]
        i = i + 1

    return p0_score, p1_score
```

As seen from the code, we're first calculating the possible moves and then evaluating each move (to the best possible extent allowed by the time restriction). The evaluation process consists of running each move through 20 simulations and then averaging the scores (+1 for win, -1 for loss, or +0.5 for tie) from the simulations.

For our implementation, we found that 20 simulations offered the best win rates compared to the

time taken to evaluate each move, and resulted in beating the random agent over 95% of the time.

In theory, our agent would be evaluating each possible future move to maximize the score that it can generate against the adversary. Since the simulations are random, this further allows the possibility of taking bad moves which may perform worse now, but may perform much better later. The stochastic nature also ensures that it is also very unlikely that we will be getting the same scores each time even if we run the same simulation again at a different step.

# 3 External resources applied

We extensively used Google to find resources discussing various formulations of the Monte Carlo search algorithm. This included everything from learning the very core ideas of the algorithm from several YouTube videos, to actual implementations of the algorithm in different games like Tic-tac-toe. Using that knowledge, we developed our own formulation of this algorithm which was best suited to the rules of this game.

# 4 Advantages and disadvantages

Before applying the pure Monte-Carlo search, we have implemented the Minimax Search with alpha-beta pruning and Monte-Carlo tree search. Compared to these two applications, our approach is much simpler to implement and also significantly faster. Also, because Colosseum Survival is a timed game, having a fast algorithm is very important because if you have the perfect algorithm that selects the most optimal move at each round but it takes longer than the time given to make each move, that algorithm will have a low win rate since time will be up before exploring every option. So for this game, we focused on creating a speedy algorithm. Alongside being simple to implement and fast, our approach is also semi-stochastic, so our student agent allows for bad moves in the beginning stages of the game which can turn out to be better towards the end.

The disadvantages of our approach are mostly based on repetitions. In our algorithm, the student agent plays 20 random games from each possible move that it can take. So, even though it is not statistically very high, we may have repeated simulations. Furthermore, at every state of the game, we need to calculate and validate possible moves from every step, so again it is possible to have some repeated checks. Lastly, if we were to play this game in a much bigger board, we may run out of time and not be able to make all the simulations, explore all the endgames, and thus not be able to find the most optimal move at each game state. But for the given board size that ranges between 6x6 to 12x12 our algorithm performed reasonably well.

# 5 Possible cases of failure

Our student agent seems to win against the random agent 95% of the time for the games that are played on a board that can be as small as 6x6 or as big as 12x12. However, if this game were to be played in a much bigger board, we will have a much bigger branching factor; thus it may cause our agent to run out of time before simulating the game from every possible move, which may lead our agent to not find the optimal move. Our agent may also lose against a very smart adversary that has implemented a couple of heuristics that makes its moves in a way that minimizes our moves and traps our agent.

# 6 Other methods tried

Throughout the process of designing our agent, we have implemented several different approaches. Firstly, we have decided to do a minimax search with alpha-beta pruning. Since the branching factor is too high for the minimax search with the alpha-beta we decided to look for an alternative approach: Monte-Carlo Tree Search. Apart from being complicated to implement, the Monte-Carlo Tree Search approach was not the best algorithm for our game due to the time constraints of the game. The student agent needs to make its move in 2 seconds, however, while implementing the MCTS in several cases we need to create new MCTS nodes describing different states of the game, roll-out from a

node's children to explore the future states, back-propagate to calculate the score of an action, and these actions require some time to be computed. After running the student agent using the MCTS algorithm, we have realized that similar to the minimax search with alpha-beta pruning, the MCTS algorithm also has a high branching factor especially for the cases that include a huge board size. So, to refrain from the time costs at creating these nodes in the MCTS algorithm, we decided to go with the pure Monte-Carlo search. To implement this algorithm, firstly we created a simulate function which plays the game randomly for both us and the opponent. We have also created a possible_moves function that returns the moves that an agent can make from an initial position given a chessboard. Then, for every element in the list that is returned by the possible_moves function, we run the simulate function. After running the simulate function for each of these possible moves, we have recorded the student agent's score which returns the number of cells in the user's zone when the game ends. We selected the move that yields the highest score. When we run our program for only 3 times, we have realized that there is a high variance in our win rate. To decrease this variance, we have decided to simulate the game from that state for 20 times. Also, rather than looking at the number of cells the user will have at the end of the game, we started to check whether the result is a win, tie or a loss for the student agent. If the endgame result is a win we add 1 point, if it is a tie we add 0.5 point, and if it is a loss we subtract 1 point. After simulating random games from each possible move, we select the move that has the highest total points, and make the move.

# 7 Further improvements

Based on this current approach, the next best step to improve this algorithm would be to use several, informed heuristics during the evaluation process of each possible move. A heuristic would be used to order the evaluation of the possible moves in such a way such that the moves which have the most likelihood of generating a win is evaluated first. This ensures that even if the application runs out of time to perform the evaluations of all the moves, it can still return a much more favorable move. These heuristics may include choosing moves which:

1. have a better possibility of trapping the adversary (playing offensive)
2. can stop the adversary from trapping our agent (playing defensive)
3. can block the adversary from placing walls at positions which may prevent our future moves

Additionally, looking into reinforcement learning techniques and genetic algorithms to train a model which can progressively learn how to play this game could also be explored. Such an approach would take considerably more thought and computational power to implement, however should perform much better than our agent if executed properly.