COMP 424 Final Project Report –Team Joystick Fighters

Team Joystick Fighters
Chang He (email: chang.he2@mail.mcgill.ca ID:260800634)
Zongxuan Lin (email: zongxuan.lin@mail.mcgill.ca ID:260780979)

1. Introduction

This is a coding project to implement an AI agent to play the game Colosseum Survival. During the development of the agent, team joystick fighters continuously develop agents implementing different AI algorithms. Among those agents, the best-performed one is submitted as the final student agent. The main content of the report will elaborate on the algorithm, theoretical basis, assessment of the current agent, development process and other agents developed, and potential improvements.

2. Current Agent Algorithm Explanation

To effectively play the game in different scenarios, the current agent implements multiple algorithms, including the greedy algorithm, and the Monte Carlo tree search algorithm.

The agent begins choosing the next step by evaluation of available choices, with a function called all-steps-possible. Based on its current location, and the maximum step size, it searches a square area with a width of 2 times the maximum step size, and current location in the center. The all-steps-possible function then narrows the range down by checking if the coordinates in the range are out of the border of the game board (marked as the blue line in the diagram below), and if any points in the range have the horizontal and vertical distance from the current position combined to exceed the maximum allowed step size. After excluding points violating the 2 principles, the function has successfully narrowed the search range down to the effective search area, highlighted with red bold borders. the all-steps-possible function evaluates the effective search area and all possible directions with the check-valid-step function copied from the world file. All valid steps are added to an array list to be evaluated later.
Once the agent has determined all possible steps available, it passes the list into the all-next-states function where, a list of sub-sequential game boards, a list of locations, and a list of barrier directions is generated. With all the possible next states available (games boards, possible locations of the player, and adversary's location, the choice function can evaluate the utility of the choices by resorting to the check-endgame function. The greedy algorithm is utilized in this round of evaluation. If the agent found a step to be leading to its victory, it will return that step. Otherwise, it will loop through all the possible steps, marking the losing states with a utility of -1, and other states with a utility of 0.

During the second round of iteration, the agent continues to evaluate all open states, where the utility is 0, and the result is False. The goal of this iteration is to narrow down the range of the Monte Carlo tree search and build the skeleton for it. In this step, it loops finds and

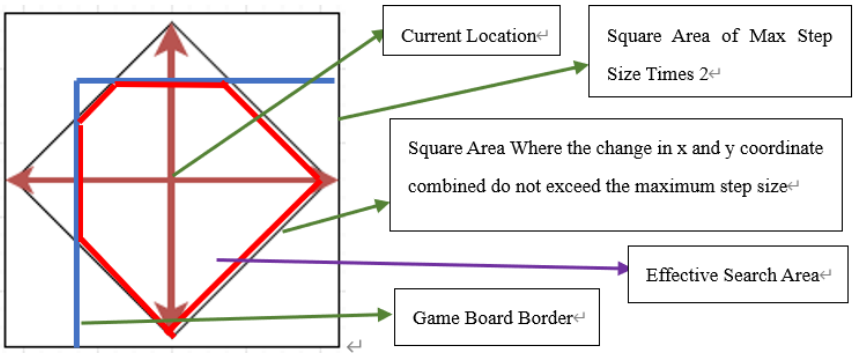

Figure 1: Illustration of Find All Possible Steps Function

loops through all possible states after the adversary's choice for each open state. If it finds a state leading to its failure, the corresponding branch is marked as closed, with a utility of -1. If all the possible states of a branch would lead to its victory, the corresponding branch will be marked as a choice leading to its victory. The agent would then return its step to that winning branch. In the meanwhile, if all consequential states of the adversaries choice do not lead to failure of the agent, and not all adversary's choice leads to the victory of the current agent, the location of the branch, all the possible states, and adversary locations are saved to be used later in the Monte Carlo Tree Search.

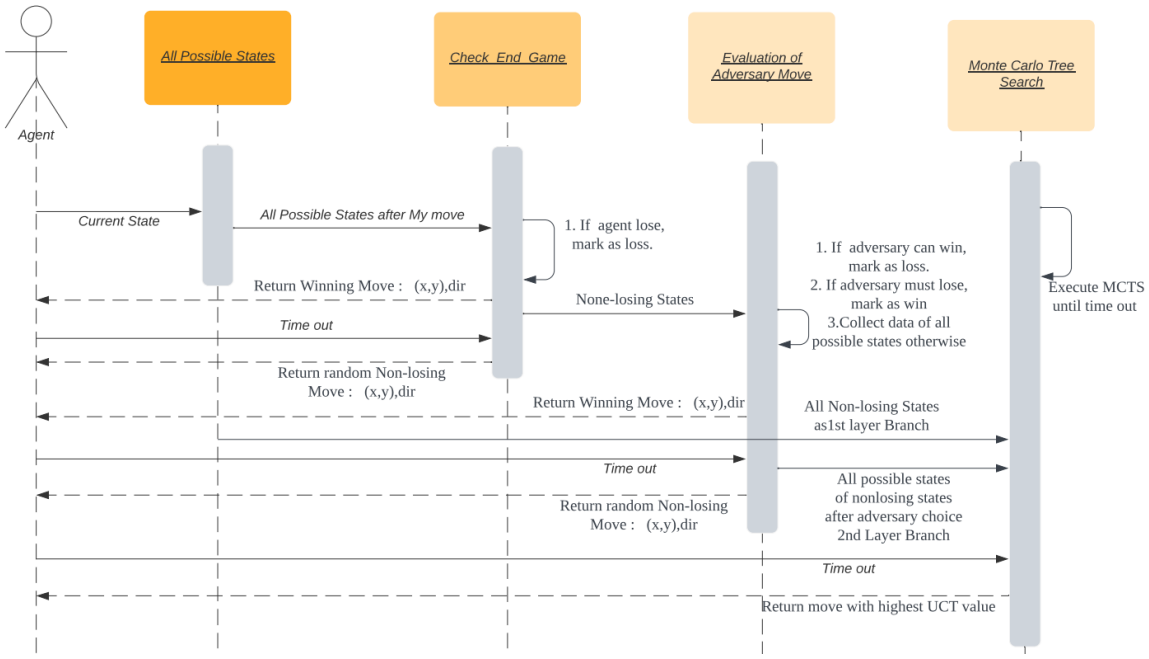

Figure 2: Agent Decision Making Process Sequential Diagram

If there is no winning move for the agent, and there are some moves with open states other than the moves causing the agent to fail. It would start doing a Monte Carlo tree search with the potential moves. By looping through the tree of states constructed in the previous

step, with a random simulation function. After several iterations before the decision-making timeout, the agent would evaluate all the choices according to their Q value, and return the move with the highest Q value.

Figure.2 above summarizes the logic of the agent. Besides the ideal decision-making process elaborated above, the agent constantly checks the time it used to make sure it does not exceed the decision-making time (30 seconds for the first move, and 2 seconds for the other moves). If the time is up, and the agent has not yet entered the MCTS, it would randomly return a step that will not lead to its own failure. If it enters the Monte Carlo Tree Search before the time is up, it will run a random simulation as many times as possible, and return the step which MCTS calculates to be the most promising step when reached the time limit.

3. Theoretical Basis

In this section, we will focus on the theoretical background of our chosen approach. Our program is implemented based on the Monte Carlo tree search (MCTS) algorithm, which is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games [1]. MCTS uses random simulations of the game to determine the best move to play. Although it is not guaranteed that the selected move is optimal, it performs much better and it is also more practical to implement than other optimal algorithms such as Minimax under strict time constraints. It adopts the Monte Carlo method, which uses repeated random sampling to estimate the best move to play given the status of the board. It relies on the Law of Large Numbers, such that an observed sample average from a large sample will be close to the true population average and that it will get closer the larger the sample. Therefore, this algorithm is not optimal but its performance will converge to the optimal algorithm given enough time. This allows the algorithm to stop and return a good enough solution under a limited time, rather than searching indefinitely and stopping only when a solution is found.

MCTS has four stages: selection, expansion, simulation, and back-propagation. This is repeated until the time requirement is met and the move that leads to the highest random-play win rate is selected. The exploration-exploitation trade-off is balanced by the Upper Confidence Bounds applied to Tree (UCT) and it is implemented in the expansion phase. This algorithm allows each iteration to pick the nodes that are promising and at the same time explore nodes that are not visited. Starting at the root node, a UCT value of the node is calculated and the node with the greatest UCT value is selected. This is repeated until a leaf node is reached which is the node that will be expanded. The equation to calculate its UCT value is shown below, where $Q(s, a)$ is the win rate of the node, $n(s)$ is the number of times the parent of the node has been visited, and $n(s, a)$ is the number of times the node has been visited. The parameter c is a hyper-parameter that can be used to adjust the balance between exploration and exploitation and commonly $\sqrt{2}$ is used.

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

4. Advantage and Disadvantage of the Current Agent

This version of the agent is the most competitive agent our team has developed and tested. However, despite the fact that the current agent is optimized for performance, it still has some shortcomings. This section elaborates on its advantages and disadvantages.

4.1 Advantages

The current agent has both the advantage of the greedy algorithm as well as the Monte Carlo Tree Search Algorithm. Combining the 2 algorithms in a logical way managed to avoid some disadvantages of both algorithms.

- With the implementation of the greedy algorithm before the Monte Carlo Tree Search, the current agent can avoid errors caused by random simulation of the Monte Carlo tree search. For the Monte Carlo tree search, in order to run enough simulation to collect enough data, the random simulation is not checked. In other words, the random simulation of the Monte Carlo Tree Search is not perfectly precise. Our previously implemented MCTS agent sometimes can choose to suicide because of such simulation precision issues.

  Thereby, using a greedy algorithm to check the available steps before the Monte Carlo Tree Search process effectively avoids the agents from making serious mistakes, like committing suicide, during the game.

- The greedy algorithm also allows the agent to choose a winning step more quickly if the agent can win in a round. In the first and second evaluation process, the agent would carefully check all possibilities, and if there is a choice that could lead to its victory, it would return it. In such a case, the agent is certain about how to win the game, and it saved a significant amount of time in decision-making process.

- In the Monte Carlo Tree Search, the agent is looping through a tree with only steps that would not cause it to lose, because those steps are all labeled and filtered out in the previous greedy search process. Besides, all starting states are saved in the previous greedy algorithm evaluation process, the MCTS would only need to input those states as parameters in the random simulation process. In this way, the search tree is smaller, and the Monte Carlo Tree Search can run more efficiently

- The current agent utilizes the maximum amount of time within the limit to make a decision. According to our team's simulation result, 0.4 seconds is the minimum amount of extra time for the agent to return its choice within the time limit, even in the most complex situation. The decision-making process takes at most 29.6s in the first round and 1.6s in the other rounds. In this way, the decision-making time in the most complex situation (12*12 game board) is controlled to be within 1.99s.

  With cautious tuning of the decision-making time, the current agent manages to use all the time it can safely use without violating the decision-making time limit.

- The current agent also features a smart function to find all the possible steps. It narrows down the possible positions before using the check valid step function (fig.1). Check valid step function is a breadth first search algorithm, which takes a significant amount of computation, especially when it is applied multiple times.

  Such improvement could optimize the speed significantly. One previously developed agent was checking all the boards to find possible steps, and could only run on a 5*5 game board, with a decision-making time of 5 seconds. After implementation of this improved function, it could run with an 8*8 game board.

4.2 Disadvantages and weaknesses

The current version of the agent has made several compromises and there are still some weaknesses that can be mitigated with optimization and adjustment of key parameters.

- Implementation of the greedy algorithm before the Monte Carlo Tree search compromises the precision of the MCTS process because less time would be allocated to the Monte Carlo Tree Search. The less time it has, the less data it collects, and its decision is more likely to be not optimal.

- The RAM usage of the current agent is not optimized. The Monte Carlo Tree Search Process has its nodes discovered before starting the algorithm leads to using an extra amount of RAM than normal. As a result, the current agent takes as much as 100MB of RAM when playing games on a 12*12 game board. This space could actually be saved as some of the branches might never need to be expanded.

- The current agent's decision-making time is optimized for 12*12 game boards but not other smaller ones. During competition with other agents on smaller game boards, its shorter decision-making time could lead to the wrong choice of steps.

- During the start of the game, since there is a huge selection of possible steps, it is highly possible that our agent could take a random step since it may not have enough time to do enough Monte Carlo Tree Search.

4.3 Expected Failure Mode

The expected failure mode of the agent is to be defeated by another AI agent that more effectively evaluates its move. Since this agent does not commit suicide, it is unlikely to be defeated by a random agent. this agent can secure its victory by just patiently waiting for the random agent to commit suicide.

Thereby, the most probable failure mode of this agent is to be defeated by a more efficient and intelligent AI. If another agent evaluates its possible moves more efficiently, that agent could take advantage of this agent's possible random behavior during the start of the game. As the more effective AI builds up its advantage, it is highly likely for it to eventually occupy more space and win the game against the AI agent of our team.

5. Other Agents Developed

Many other agents are developed as we actively look for ways to implement an effective way to improve the smartness, and efficiency of our agent.

- The first agent developed is the greedy agent, which implements the basic logic of a human player. It would check all possible states for the next round of the game, if it finds that a move would lead to its victory, it would choose that move. If some moves would lead to its failure, it would never choose these moves unless there are no other moves to choose from.

  This agent features a fast response and a more than 95% high winning rate against the random agent. It is mainly used to test the effectiveness of other agents implementing more complex AI algorithms.

- The second agent is the greedy agent combined with a simplified Monte Carlo Tree Search. This agent doesn't have the node expansion feature of Monte Carlo Tree Search. Its random simulation function is also not effective enough, since it constantly calls the validate step function. As a result, it cannot run on larger than 8*8 game boards. It is also frequently defeated by the greedy agent, indicating that its way of evaluating the next step is not effective.

- The third agent uses Monte Carlo Tree Search Only. During the development of this agent, our team managed to speed up the random simulation process, such that each round of random simulation takes only 0.01 seconds, this agent effectively implements the MCTS algorithm, and its wining rate against the greedy agent is more than 60%. Thereby, this agent is also a competitive alternative for the tournament.

- The fourth agent is a mini-max agent searching all possible states over 2 rounds. This agent evaluates the next step more precisely than the MCTS agent, and often wins the MCTS agent on small game boards, but it cannot operate on larger than 7*7 boards.

As a result, only the greedy agent, MCTS agent, and the current agent can play the game on 12*12 boards. Our team let the agents play against each other, and the result is summarized in the table below. As the table demonstrated, the currently selected agent is testified to be the most competitive among all the available agents.

Figure 3: Selection Process of the More Competitive Agent

Agent Type	Wining Rate Against Random Agent (100 Simulation)	Wining Rate Against greedy agent (50 Simulation)	Simulation Result of the 2 most competitive agent (100 Simulation)
Greedy	96%	N/A	N/A
MCTS	99%	67%	31%
Greedy + MCTS	100%	61%	69%

6. Potential Improvement

There are many ways to improve the current agent's performance.

- Firstly, the time limit for choosing a step can vary dynamically with the board size. With the larger board, it takes a longer time for the agent to evaluate and return a step. The decision-making process is shorter on the smaller boards. Adjusting the agent's decision-making time according to board size would thus allow it to run more simulations and make smarter decisions on small game boards.
- Secondly, during the evaluation of the adversary's move, the agent could first evaluate its adversary's next move that connects with an existing barrier. If these moves do not cause the failure of the agent, it can then save the adversary's other moves that do not touch the existing barrier to the potential lists.

  The logic behind this is that a move deciding the result of the game has to be connected with existing barriers. By evaluating these moves first, the agent can evaluate the danger of its move more efficiently.
- Another substantial improvement could be making the random simulation time to be shorter. In such a way, that agent can have more data to analyze within the time limit.
- Moreover, other more advanced techniques could be used to improve the smartness of the current agent, such as deep learning.

7. Conclusion

In this report we introduced the mechanics of the current agent algorithm, including the method of generating all possible next moves, using a greedy algorithm to check for an automatic win or lose to narrow the search space of MCTS, and finally using MCTS to determine the most promising move. Then the advantage of the current agent is discussed: using greedy before MCTS search prevents the agent to choose a losing move and shortens the amount of time to find a winning move when it is available; and a smart function to find all steps which further saves time before calling the function to check valid step. The disadvantages of the agent are reduced time spent on MCTS due to the greedy algorithm, RAM usage problems, sub-optimal decision making on smaller boards, and possible random decisions when the time is not enough. Other agents have been carried out such as a greedy agent, a simplified model of our current agent, and a minimax agent and by experimental results, the best agent is the greedy and MCTS agent which is our current agent. There are still methods to improve the current agent's performance by varying strategies against different board sizes and board status, or simply shortening the amount of time required for the random simulation.

References

[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Pearson, 2022.