AI in Games Assignment 1

Animesh Devendra Chourey 210765551

Paul Sasha Mutawe 210715187

Ayesha Jaleel 210774047

School of Engineering and Computer Science, Queen Mary University of London, UK

ec21949@qmul.ac.uk
ec21935@qmul.ac.uk
ec21288@qmul.ac.uk

Abstract. The Pommerman is a multi-player game played among a set of agents or human players, where the last person to stand wins. The objective of this report is to focus on and elucidate one of the advanced algorithms making several breakthroughs in game development, the Monte Carlo tree search. We have made modifications to the existing MCTS algorithm by introducing a progressive strategy known as progressive bias, where a heuristic knowledge is added to the selection step of the algorithm, which helps in the traversal of nodes that are less often visited. We gauge the outcome of the enhanced MCTS algorithm by pitting it against other agents that execute disparate advanced and simple search algorithms (RHEA, OSLA etc.) in partial and full observability and executing in two game modes. The results of our experimentation are studied comprehensively and detailed in the subsequent sections of this report.

1 Introduction

Whenever playing a game, one is often pitted against single or multiple enemies. Developing games in Artificial Intelligence is arduous task as there is no one clear solution to go about the problem. Pommerman is a video game founded on the identical working of the classic console game called Bomberman. The main motive behind the game and this module is to test the behavior of our player's predictive capability against multi-agent game setup. Since it is a board game, to compete against enemies our player must be significantly intelligent enough to outlast them till the end. For the agent to be quick and sharp, it must consider multiple factors beforehand and opt for the best possible route while deciding. Every player looks out for itself and plays for a zero-sum game so it's important for our player to behave accordingly and gain advantage by making prompt and accurate decisions. In order to follow and act duly, we will be going to implement Monte Carlo Tree Search (MCTS) algorithm whilst making some variations to it as our agent. MCTS proved to be substantially better than other algorithms in this setup.

This report is going to outline Pommerman game setup along with the literature review describing previous work. It will then define the algorithm adopted as our agent in detail to give a better understanding to novel readers explaining the approach taken to overcome the problem at hand. Afterwards, experimental study will provide with the analysis of how experiments with different setup were conducted to obtain the results. Finally, the report will conclude along with the scope of future work.

(Give an introduction to the problem tackled. Why it is important? Outline the rest of the document.)

Initial introduction to the problem, motivation and outline of the rest of the report.

2 Pommerman

Pommerman is based on a classic game called bomber man which has 4 agents in different game modes with the goal to eliminate other players using perks like bombs etc. the goal is to be the last man standing or the game end by the time the game makes its last tick. The P from Pommerman comes from the partial observability setting that we can use for further experimental study.

The focus of pommerman is to provide a framework on statistical forward planning. Statistical forward planning frameworks are approaches that use froward models to simulate future state when they observe a state and action pair.

The Pommerman game has three game modes: Namely FFA (Free for All, 4 Players, Full observability or Partial Observability), Team (2vs2, Partial Observability), Team Radio (2vs2, PO, Radio Comm). The game is played on a randomly made 11x11 grid with soft wooden obstacles that you can destroy with bombs and hard obstacles that cannot be destroyed. The Pommerman game has items that can be picked up by agent: items in the breakable wooden obstacles, Kick perk that allows to kick away a bomb before it explodes, range perk that adds range of bomb explosion and an extra bomb that can be picked up.

The challenges that come with Pommerman include dealing with more than one agent, Observability, Duration of certain actions, the layout of the soft and hard balls and the line-of-sight considerations.

3 Background

Pommerman is a game that uses statistical forward planning methods. Statistical forward models are methods that simulate future moves by observing the current state of a node plus the action associated with that specific node. The statistical forward method that we used in this assignment is the monte Carlo tree search method along with an advanced heuristic called a progressive bias which will be explained later.

Tree search algorithms are a method used to search all possible moves that may exist after every game turn. Using a brute force to in a possibly exponential tree search

to find a solution considering every state and action requires a lot of computational power.

MCTS is a heuristic driven search algorithm that is a combination of classic tree search implementations alongside machine learning principles of reinforcement learning. MCTS finds the best move out of a set of moves with four steps namely, Selecting, Expanding, Simulating, updating all the nodes till it finds the optimal solution to the assigned problem.

Selecting. This process selects the node with the highest possibility of winning from the search space of all nodes. This process happens at all levels of the search tree until the final node is reached.

Expanding. After selecting the right node, we expand the options by creating children nodes which are future moves that can be played in the game.

Simulation. We use Reinforcement Learning to make random decisions in the game further down from every children node. Then, reward is given to every children node — by calculating how close the output of their random decision was from the final output that we need to win the game.

Backpropagation. The positive or negative scores reached by the final node is backpropagated back up the tree updating all the nodes used to reach that conclusion.

After updating all the nodes, the loop again begins by selection the best node in the tree→ expanding of the selected node → using RL for simulating exploration → backpropagating the updated scores → then finally selecting a new node further down the tree that is actual the required final winning result alongside machine learning principles of reinforcement learning. MCTS finds the best move out of a set of moves with four steps namely, Selecting, Expanding, Simulating, updating all the nodes till it finds the optimal solution to the assigned problem.

4 Method

The Pommerman is an exemplar of a multi-player game which exhibits many distinct features of AI such as learning, pathfinding and search trees. It utilizes several heuristic based algorithms for its agents to help make a move that guarantees the best outcome and rewards, such as Monte Carlo Tree Search (MCTS), Rolling Horizon Evolutionary Algorithm (RHEA), One Step Look Ahead (OSLA) etc. The approach applied in this assignment is improvising the selection phase of MCTS algorithm through the process of adding a progressive strategy called progressive bias.

MCTS for PommerMan

The MCTS implementation in Pommerman has 3 parameters to specify its configurations for the algorithm (MCTSParams.java)

- K = Math. sqrt (2) - constant value for UCB calculations
- rollout_depth – number of steps the tree can grow from root node
- heuristic_method = - the heuristic used to evaluate the state

Custom Heuristic: - The custom heuristic method, which is the heuristic function for MCTS implementation in the Pommerman, (CustomHeuristic.java) uses several parameters to help find the optimal solution from a certain state. The parameters based on which the algorithm works are:

- Number of teammates who are alive
- Number of enemies who are alive
- Number of wooden blocks that remain
- Blast strength
- Ability to kick bombs

The difference between the parameters mentioned above is calculated for a specific state with respect to the root state. The function returns –1 if the player loses the game and 1 if the player has won.

Progressive bias

The progressive bias is a progressive strategy technique, wherein the selection process of MCTS is modified by incorporating additional heuristic knowledge. This approach especially helps those nodes that haven't been traversed enough with the traditional UCB computations and a heuristic function that is dependent on the game, may help in the selection process of such nodes. We implement this by adding a heuristic function to the existing calculation of Upper Confidential Bound (UCB) to bias the selection procedure.

UCB = V_i + C * sqrt (Ln N/n_i) + h_i/n_i,

Where h_i is the heuristic value of the i^{th} state.

Again, the UCB value for each of the child states is calculated as per the modified formula for UCB and the node that maximizes this value is chosen.

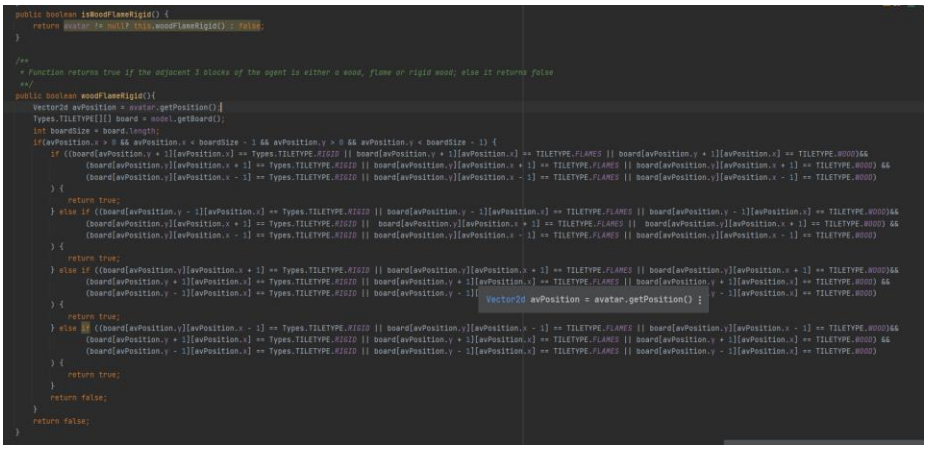
```
//roll(gsCopy, child.actions[child.childIdx]);
double rollval = rootStateHeuristic.evaluateState(gsCopy);
double hvVal = child.totValue;
double childValue =  hvVal / (child.nVisits + params.epsilon);

childValue = Utils.normalise(childValue, bounds[0], bounds[1]);

double uctValue = childValue +
        params.K * Math.sqrt(Math.log(this.nVisits + 1) / (child.nVisits + params.epsilon)) + (rollval / (1 + child.nVisits + params.epsilon));
```

Here "rollval" variable is performing the task of h(s) and "1+child.nvisits.params.epsilon" works as 1+N(s,a).

For the implementation of progressive bias, we have added another parameter to the existing heuristic used for MCTS, the 'CustomHeuristic'. The attribute that was introduced is called 'isWoodFlameRigid, where the method checks if a breakable wood, rigid wood or flame is present in the three adjacent blocks of the agent. If any of the objects mentioned are present, the function returns true, otherwise it returns false. Based on the boolean value returned by the method, an integer value of 1 or 0 is assigned to the attribute 'isWoodFlameRigid. This attribute, multiplied with a factor is estimated into final heuristic calculation for that state.



```
public boolean isWoodFlameRigid() {
    return avatar != null? this.woodFlameRigid() : false;
}

/**
 * Function returns true if the adjacent 3 blocks of the agent is either a wood, flame or rigid wood; else it returns false
 **/
public boolean woodFlameRigid(){
    Vector2d avPosition = avatar.getPosition();
    Types.TILETYPE[][] board = model.getBoard();
    int boardSize = board.length;
    if(avPosition.x > 0 && avPosition.x < boardSize - 1 && avPosition.y > 0 && avPosition.y < boardSize - 1) {
        if ((board[avPosition.y + 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y + 1][avPosition.x] == TILETYPE.FLAMES || board[avPosition.y + 1][avPosition.x] == TILETYPE.WOOD)&&
                (board[avPosition.y][avPosition.x + 1] == Types.TILETYPE.RIGID || board[avPosition.y][avPosition.x + 1] == TILETYPE.FLAMES || board[avPosition.y][avPosition.x + 1] == TILETYPE.WOOD) &&
                (board[avPosition.y][avPosition.x - 1] == Types.TILETYPE.RIGID || board[avPosition.y][avPosition.x - 1] == TILETYPE.FLAMES || board[avPosition.y][avPosition.x - 1] == TILETYPE.WOOD)
        ) {
            return true;
        } else if ((board[avPosition.y - 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y - 1][avPosition.x] == TILETYPE.FLAMES || board[avPosition.y - 1][avPosition.x] == TILETYPE.WOOD)&&
                (board[avPosition.y][avPosition.x + 1] == Types.TILETYPE.RIGID ||  board[avPosition.y][avPosition.x + 1] == TILETYPE.FLAMES ||  board[avPosition.y][avPosition.x + 1] == TILETYPE.WOOD) &&
                (board[avPosition.y][avPosition.x - 1] == Types.TILETYPE.RIGID || board[avPosition.y][avPosition.x - 1] == TILETYPE.FLAMES || board[avPosition.y][avPosition.x - 1] == TILETYPE.WOOD)
        ) {
            return true;
        } else if ((board[avPosition.y][avPosition.x + 1] == Types.TILETYPE.RIGID || board[avPosition.y][avPosition.x + 1] == TILETYPE.FLAMES || board[avPosition.y][avPosition.x + 1] == TILETYPE.WOOD)&&
                (board[avPosition.y + 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y + 1][avPosition.x] == TILETYPE.FLAMES || board[avPosition.y + 1][avPosition.x] == TILETYPE.WOOD) &&
                (board[avPosition.y - 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y - 1][
                    Vector2d avPosition = avatar.getPosition() ;                                          y - 1][avPosition.x] == TILETYPE.WOOD)
        ) {
            return true;
        } else if ((board[avPosition.y][avPosition.x - 1] == Types.TILETYPE.RIGID || board[avPosition.y][avPosition.x - 1] == TILETYPE.FLAMES || board[avPosition.y][avPosition.x - 1] == TILETYPE.WOOD)&&
                (board[avPosition.y + 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y + 1][avPosition.x] == TILETYPE.FLAMES || board[avPosition.y + 1][avPosition.x] == TILETYPE.WOOD) &&
                (board[avPosition.y - 1][avPosition.x] == Types.TILETYPE.RIGID || board[avPosition.y - 1][avPosition.x] == TILETYPE.FLAMES || board[avPosition.y - 1][avPosition.x] == TILETYPE.WOOD)
        ) {
            return true;
        }
        return false;
    }
    return false;
}
```

The above figure shows the Boolean method we created in which the agent observes whether there is a rigid block, flame block or wooden block. The Boolean value will be set to 1 or 0 and there will be an int value made from the Boolean value which in turn is multiplied by a factor that is used in the progressive bias function.

The difference between these attributes is computed for a certain state with respect to its root state. The function returns –1 if the player loses the game and 1 if the player has won. By taking into consideration the above attributes, the selection process could be redirected to different nodes as opposed to the traditional MCTS approach.

5 Experimental Study

The experimental study undertaken consisted of first collecting the results of how the MCTS algorithm performed without the tuning (Figure 1 & Figure 2). After that we observed how the MCTS performed with tuning we tuned the MCTS algorithm by adding the progressive bias to the MCTS algorithm. In the agents below the basic algorithmic differences are that SimplePlayer only uses heuristics to make judgments e.g., where & when should the bomb be placed, OSLA uses one rollout, MCTS is like OSLA, but it uses multiple rollouts, RHEA uses evolutionary mutations to make its actions. Overtime average is the amount of time after the time limit that the agent still uses to make a decision. In this specific case were not told to experiment with the overtime average thus why our MCTS2 agent makes decisions in overtime, we did not edit the algorithm to make a decision once the time limit has been reached.

Figure 1 below consists of a free for all game played for 10 iterations and five levels. The observability here is set to minus one which is full observability which means that every agent in the game can see all spaces (there is no fog of war). The players involved in this game are MCTS, RHEA, OSLA, Rule based. As we can see the MCTS player whoch is the most advanced agent has the highest percentage wins at 54% and the OSLA player which is the least competent agent has the worst score 0%.

0-10-5--1-5-4-2-3 [MCTS,RHEA,OSLA,RuleBased]

N	Win	Tie	Loss	Player (overtime average)
50	54.0%	6.0%	40.0%	players.mcts.MCTSPlayer (1.0)
50	26.0%	6.0%	68.0%	players.rhea.RHEAPlayer (0.0)
50	0.0%	0.0%	100.0%	players.OSLAPlayer (0.0)
50	12.0%	4.0%	84.0%	players.SimplePlayer (0.0)

Figure 1

Figure 2 shows the same exact game space with only one change; observability was changed to partial observability. Which means that all agents could only see up to five block spaces. This alters the game space for the agents in which the more advanced agents now have the upper hand as they can alter their actions to the new search space. As we can see here, MCTS wins by 24% with the least losses.

0-10-5-5-5-4-2-3 [MCTS,RHEA,OSLA,RuleBased]

N	Win	Tie	Loss	Player (overtime average)
50	24.0%	64.0%	12.0%	players.mcts.MCTSPlayer (1.0)
50	6.0%	60.0%	34.0%	players.rhea.RHEAPlayer (0.0)
50	0.0%	2.0%	98.0%	players.OSLAPlayer (0.0)
50	6.0%	6.0%	88.0%	players.SimplePlayer (0.0)

Figure 1

The next figures involves the progressive bias added to the monte carlo tree search and this leads to an improved tree search algorithm whch in turn makes the agent more advanced. This first game was a free for all with full observability with the same amount of levels and iterations (amount of levels & iterations doesn't change throughout all experiments). Agents involved in this figure are MCTS, RHEA, OSLA and MCTS2 (Our player). As we can see our agent wins with a percenage of 18% and only loses a percentage of 20%. The progressive bias has made the MCTS agent more effective.

0-10-5--1-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2] FFA mode Full Observability

N	**Win**	**Tie**	**Loss**	**Player (overtime average)**
50	**4.0%**	**72.0%**	**24.0%**	**players.mcts.MCTSPlayer (1.0)**
50	**2.0%**	**56.0%**	**42.0%**	**players.rhea.RHEAPlayer (3.0)**
50	**0.0%**	**0.0%**	**100.0%**	**players.OSLAPlayer (0.0)**
50	**18.0%**	**62.0%**	**20.0%**	**players.mcts.MCTSPlayer2 (1.0)**

Figure 3

Figure 4 shows the exact same configuration as above with the observability set to up to 5 blocks meaning there is a fog of war setting. With this single change the percentage winning of every agent decreases which shows that observability makes the game harder to win for all agents but our agent with the progressive bias still wins buy 12%.

0-10-5-5-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2] Partial Observability

N	Win	Tie	Loss	Player (overtime average)
50	10.0%	52.0%	38.0%	players.mcts.MCTSPlayer (0.0)
50	6.0%	60.0%	34.0%	players.rhea.RHEAPlayer (0.0)
50	0.0%	4.0%	96.0%	players.OSLAPlayer (0.0)
50	12.0%	66.0%	22.0%	players.mcts.MCTSPlayer2 (0.0)

Figure 2

The last part of the experimentation is the team mode with the advanced heuristic because as we have seen the advanced heuristic method of the progressive bias out performed the custom heuristic. The first experiment in figure 6 was the team of MCTS & OSLA vs RHEA & MCTS2 (Our player) with full observability. In this figure the MCTS2 team wins with a percentage of 56%.

1-10-5--1-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2] Full Observability Team Mode

N	Win	Tie	Loss	Player (overtime average)
50	32.0%	12.0%	56.0%	players.mcts.MCTSPlayer (0.0)
50	56.0%	12.0%	32.0%	players.rhea.RHEAPlayer (0.0)
50	32.0%	12.0%	56.0%	players.OSLAPlayer (0.0)
50	56.0%	12.0%	32.0%	players.mcts.MCTSPlayer2 (0.0)

Figure 5

The last result is the MCTS & OSLA vs RHEA & MCTS2 with partial observability. As we have observed before partial observability decreases the win rate of all the agents including our player. RHEA & MCTS2 won with a win percentagae of 64% which is significantly higher than the percentage win they had with full observability.

1-10-5-5-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2] Team Mode Partial observability

N	Win	Tie	Loss	Player (overtime average)
50	24.0%	12.0%	64.0%	players.mcts.MCTSPlayer (0.0)
50	64.0%	12.0%	24.0%	players.rhea.RHEAPlayer (9.98)
50	24.0%	12.0%	64.0%	players.OSLAPlayer (0.0)
50	64.0%	12.0%	24.0%	players.mcts.MCTSPlayer2 (0.86)

Figure 6

In figure 7 when rollout depth was decreased to 6, our MCTS2 player came dead last. Rollout depth is the amount of levels the tree exapnds down to find the most optimal state. When rollout depth is reduced to 6 the opitmal value is not found thus win rate is low because the progressive bias cannot be used along the nodes thus making our agent inept to perform. Although its very interesting to see that MCTS without the progressive bias happens to have 28% more wins than our agent.

Roll depth=6

0-10-5--1-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2]

N	Win	Tie	Loss	Player (overtime average)
50	28.0%	20.0%	52.0%	players.mcts.MCTSPlayer (0.0)
50	38.0%	18.0%	44.0%	players.rhea.RHEAPlayer (11.44)
50	0.0%	0.0%	100.0%	players.OSLAPlayer (0.0)
50	4.0%	30.0%	66.0%	players.mcts.MCTSPlayer2 (0.88)

Figure 7

When rollout depth is increased to 10 the win percentage for our agent MCTS2 is lower than usual. When rollout is increased it decreases the amount of breadth that the tree search has and what that does is decreases the amount of other optimal soluions that the algorthm could find thus decreasing the percentage win rate.

0-10-5--1-5-4-2-7 [MCTS,RHEA,OSLA,MCTS2] Roll Depth=10

N	Win	Tie	Loss	Player (overtime average)
50	6.0%	70.0%	24.0%	players.mcts.MCTSPlayer (0.0)
50	6.0%	60.0%	34.0%	players.rhea.RHEAPlayer (0.0)
50	0.0%	4.0%	96.0%	players.OSLAPlayer (0.0)
50	10.0%	66.0%	24.0%	players.mcts.MCTSPlayer2 (0.0)

Figure 8

6 Discussion

This section encapsulates and sheds lights on the evidence procured. The expectations with the settings applied in accordance with the results give positive fruition. OSLA player always comes shorthand while performing is solely due to its inability to look past more than a step ahead in comparisons to other players several steps look ahead advantage. OSLA is not recommended to be applied as either it is always losing, or its win ratio does not really make any significant case to make any difference. RHEA is also underperforming when in comparison with the MCTS, because there is a low overhead time for the moves as it is vital to make quick decisions, it is not able to beat the overall average win percentage of MCTS. Overhead time served as a detriment for the RHEA performance. Another factor that contributed to such finding which can also be backed by the research done on RHEA is that it always underperforms MCTS when there is low population size for it to evolve over. One way to interpret it is that, here in Pommerman specific setup because of the low branching factor there are not sufficient variations available for RHEA to try for the better evolution.

Introducing Progressive Bias to the MCTS improved the MCTS performance even more but not to the expectations. It can iterate more effectively because it enhances the selection process with little domain knowledge. Although the win percent is comparatively higher when compared to other agents, our enhanced agent is not winning a large proportion of overall matches played. One thing that could be noted here is that as we tweaked only the selection process here. Therefore, this points towards the gradual increase in win rate. Surprisingly enough changing the heuristic for the estimated evaluation also did not affect the results drastically. Altering the heuristic's decision ability to make actions based on the next step reward constructed on trap situation only made slight raise in the win percentage. This is the step which we did not anticipated. Unexpectedly, altering the heuristic did not improve much. This only

further supports the evidence we got as MCTS efficiently and quickly adapts when chances of winning recedes as there are a lot of draw percentage.
The performance of our agent could have been improved to a greater extent but because of limited time constraint no further variations could be added to MCTS. Biasing rollout and average sampling techniques supposedly could have enhanced the win average. Furthermore, minimax alpha-beta pruning implementation can also be used to reduce the overhead drastically. Making more hybridization in MCTS even after implementing progressive bias will only further boost the probability of winning.

7 Conclusions and Future Work

As part of this assignment, we had executed one of the progressive strategies, namely progressive biasing to refine the selection phase of the MCTS algorithm in the Pommerman. We had run tournaments in the game between the agent that performed our enhanced MCTS and other agents that carried out search tree algorithms like RHEA (Rolling Horizon Evolutionary Algorithm), OSLA (One Step Look Ahead), and the existing MCTS, with partial and full observabilities in Free for All (FFA) mode. From the results that were obtained, it was evident that the modified MCTS algorithm could outplay the other agents in both full and partial observabilities. The agents were also pitted against each other in team mode, and the team that played our enhanced version of MCTS has shown promising results.

In the future, we will study and experiment with some techniques of selection enhancements such as All Moves as First (AMAF) and Rapid Action Value Estimate (RAVE). We are also interested in implementing another progressive strategy called progressive uprunning to study the effects of restricting branching factors and progressively increasing it over time.

References

1. C. Browne, E. J. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, 2012.
2. Brilliant.org. 2021. Tree search | Brilliant Math & Science Wiki. [online] Available at: <https://brilliant.org/wiki/tree-search/#:~:text=A%20tree%20search%20starts%20at,traverse%2Fpass%20through%20a%20node.> [Accessed 1 November 2021].
3. En.wikipedia.org. 2021. Monte Carlo tree search - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Monte_Carlo_tree_search> [Accessed 1 November 2021].
4. (R. D. Gaina, J. Liu, S. M. Lucas, and D. P´erez-Li´ebana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in European Conference on the Applications of Evolutionary Computation. Springer, 2017, pp. 418–434.)

5. G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk and H.J. van den Herik: "Progressive Strategies for Monte-Carlo Tree Search"
6. A Survey of Monte Carlo Tree Search Methods
7. Medium. 2021. The Practical Value of Game AI. [online] Available at: <https://towardsdatascience.com/ai-research-and-the-video-game-fetish-71cb62ffd6b3> [Accessed 29 October 2021].
8. Worldscientific.com. 2021. PROGRESSIVE STRATEGIES FOR MONTE-CARLO TREE SEARCH | New Mathematics and Natural Computation. [online] Available at: <https://www.worldscientific.com/doi/abs/10.1142/S1793005708001094> [Accessed 20 October 2021].