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

ECS7002P: AI in Games

Monte Carlo Tree Search with Progressive Bias and Decaying Reward

Group-AS

Azar Park	180047723
Shrabana Biswas Shruti	210889846
Mughees Asif	180288337

Due: 08 November 2021

Abstract

This research focuses on *Pommerman*, which is a baseline multi-agent game with partial and full-observability options. The game can be played either in team (TEAM) or Free-For-All (FFA) mode and the goal is to be the last agent/team standing whilst progressing through the game equipped with player-health damaging *bombs*. The main aim of the study was to improve upon the classical Statistical Forward Planning algorithm: Monte Carlo Tree Search (MCTS). MCTS is a highly selective best-first search method used for determining optimal outcomes in a given domain by obtaining random samples from the decision space and constructing a search tree based on the results. The new agent, `MCTSBias`, was modified with progressive bias and the decaying reward strategy. The results highlight a decisive improvement in the overall performance of the new agent as compared to the vanilla `MCTS` which was outperformed during experimentation. In addition, other players with different technical architectures were also explored to validate the performance of `MCTSBias`.

Keywords—Pommerman, Statistical Forward Planning, Monte Carlo Tree Search, Progressive Bias, Decaying Reward

Contents

1 Introduction

The *Pommerman* game is a variant of the classical Nintendo game called *Bomberman* and was created to discover the behaviour of multi-agent learning [1]. Subsequently, the game has attracted many researchers to investigate the exciting challenges that are a a by-product of different Artificial Intelligence (AI) techniques. In particular, Monte Carlo Tree Search (MCTS) due to the technique's dominance in the ancient game, Go, which was previously considered extremely difficult for AI methods [2]. The study from [3] showed that the MCTS algorithm is used for optimization and to improve the results on various benchmark problems. Pommerman is an excellent guide for learning opponent modelling and implements Forward models "...to simulate possible future states when given state-action pairs" [4].

This research focuses on the modification of the vanilla MCTS agent to use progressive bias, in addition to employing the decaying reward strategy to maximise the performance of the player. The improved agent, `MCTSBias`, is then compared to other players and tested with different parameters to determine the reliability of the research.

The structure of the paper is as follows:

- Section 2 outlines the *Pommerman* game.
- Section 3 provides an elucidation of the technical framework of MCTS.
- Section 4 presents the baseline parameters and highlights the experimental results.
- Section 5 evaluates the results of the study with further recommendations.
- Section 6 concludes the paper.

2 Pommerman

The environment of Pommerman is a randomly generated 11×11 grid with each player placed in one of the 4 corners of the grid at at the start. The game consists of two types of obstacles: wooden and rigid boxes [4]. The players are equipped with *bombs* which are only allowed to be dropped one at a time with a blast radius of 1 cell around the bomb. The bomb explosion destroys players, wooden objects, and other items or bombs after 10 ticks [4]. After every tick, the agents are supplied with the knowledge of the state and must return one of the following actions: `STOP`, `RIGHT`, `LEFT`, `UP`, `DOWN`, and `BOMB` [4]. The game happens in two modes: Free-For-All (FFA), where all 4 players eliminate one another, or team (TEAM) mode, where agents compete in predefined pairs.

The objective of the game is to be the last player/team standing, and the game can also be played with full or partial observability. Fully observable environments provide a complete snapshot of the state of the environment, with all *relevant* information associated with the choice of action, to the agent [5].

3 Background

3.1 Monte Carlo Tree Search (MCTS)[1]

Monte Carlo Tree Search (MCTS) is a highly selective best-first search method used for determining optimal outcomes in a given domain by obtaining random samples from the decision space and constructing a search tree based on the results [6]. MCTS uses randomised search space exploration to build an asymmetric tree that is skewed towards only the most promising results from the search space. The algorithm is thus able to forecast the most promising steps accurately using the outcomes of prior explorations.

$$a = \underset{a \in A(s)}{\arg\max} \left(Q(s,\ a) + C\sqrt{\frac{\ln \mathrm{N(s)}}{N(s,\ a)}} \right) \tag{1}$$

where:

$Q(s,\ a)$ = the value of a action a in state s (exploitation term)
$N(s,\ a)$ = number of times the action a has been played from state s
$A(s,\ a)$ = set of available actions
C = exploration constant (equal to $\sqrt{2}$)

Algorithm 1 General MCTS algorithm

```
function MCTSSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_i ← TreePolicy(v_0)            ▷ selection & expansion
        Δ ← DefaultPolicy(s(v_i))        ▷ simulation
        BackUp(v_i, Δ)                   ▷ backpropagation
        return a(BestChild(v_0))
    end while
end function
```

3.1.1 Selection

The MCTS method uses a recursive technique to explore the depth of the tree by starting from the root node. To choose the nodes with the highest estimated value, the technique employs an evaluation function.

To offset the exploration-exploitation dilemma [6] in MCTS, Upper Confidence Bound for Trees (UCT) (Section 3.2) is a technique which strikes the right balance between the two during the selection step. A node is chosen during the tree traversal based on some parameters that return the highest value. The parameters are defined by Eq. 2:

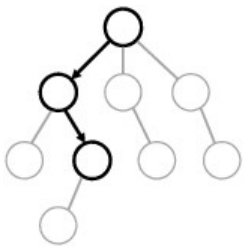

Figure 1: Selection

[1] Algorithm 1, 2, 3, 4 & 5 have been referenced from [6], while Fig. 1, 2, 3 & 4 are from [7].

$$S_i = \frac{w_i}{n_i} + C\frac{\sqrt{t}}{n_i} \tag{2}$$

where:

S_i = value of the node j
w_i = number of wins after the i^{th} move
n_i = number of simulations after the i^{th} move
C = exploration parameter (equal to $\sqrt{2}$)
t = total number of simulations for the parent node

The selected child node that is also a leaf node returns the highest value from Eq. 2 when traversing a tree during the selection phase.

Algorithm 2 Selection algorithm

```
function TreePolicy(v)
    while v is not non-terminal do
        if v is not fully expanded then
            return Expand(v)
        else
            v ← BestPolicy(v, C_p)
        end if
    end while
    return v
end function
```

3.1.2 Expansion

A new or several new child nodes are added to the tree according to the available actions [6].

Algorithm 3 Expansion algorithm

```
function Expand(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
end function
```

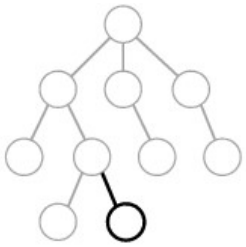

Figure 2: Expansion

3.1.3 Simulation

To obtain an outcome, a simulation is run from the new node(s) using the `DefaultPolicy`, which chooses a child node at random and simulates the entire game from that node until the game reaches its terminal state.

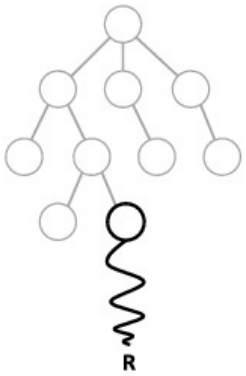

Figure 3: Simulation

Algorithm 4 Simulation algorithm

function BESTCHILD($v,\ c$)
return

$$\text{argmax}\ \frac{Q(v')}{N(v')} + c\left(\sqrt{\frac{2\ln N(v)}{N(v')}}\right)$$

end function

function DEFAULTPOLICY(s)
while s is non-terminal **do**
choose $a \subset A(s)$ uniformly at random
$s \leftarrow f(s,\ a)$
end while
return reward for state s
end function

3.1.4 Backpropagation

When the algorithm approaches the end of the game, it assesses the present state to identify the winner. It backtracks through the tree to the root node, increasing the visit score for all nodes visited. If the player for that position wins the play-out, the winning score for that node is also incremented.

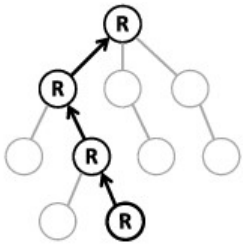

Figure 4: Backpropagation

Algorithm 5 Backpropagation algorithm

function BACKUP($v,\ \Delta$)
while s is not null **do**
$N(v) \leftarrow N(v) + 1$
$Q(v) \leftarrow Q(v) + \Delta(v,\ p)$
$v \leftarrow$ parent of v
end while
end function

3.2 Upper Confidence Bounds for Trees (UCT)

It is frequently more efficient in terms of memory to recalculate $s(v)$ as TreePolicy descends the tree rather than saving it for each node. The component of the reward vector associated with the current player p at node v is denoted by $\Delta(v,\ p)$. Because the exploration constant C is set to 0 for this final call on the root node v_0, the return value of the overall search, in this case, will be $a(\text{BestChild}(v_0))$, which will offer the action a that goes to the child node with the highest reward [6].

UCT is an extension of the MCTS algorithm combined with Upper Confidence Bound (UCB) which "... encourages the exploitation of higher-reward choices [and] encourages the exploration of less visited choices" [6].

3.3 Progressive Bias

The selection approach is more accurate when a node has been visited frequently. Progressive bias is an intuitive technique that adds domain-specific knowledge to MCTS. The goal of the technique is to lead the search based on heuristic knowledge that "... approximate the true utility of a state without doing a complete search" [5]. To that end, the selection approach is tweaked based on the new information. When only a few games have been played, the impact of this alteration is significant, but it fades quickly (as more games are played) "... to ensure that the strategy converges to a selection strategy" [8]. Therefore, by using Eq. 1:

$$a = \underset{a \in A(s)}{\arg\max} \left(Q(s,\ a) + C\sqrt{\frac{\ln \mathrm{N(s)}}{N(s,\ a)}} + \boxed{\frac{\boldsymbol{h(s,\ a)}}{\boldsymbol{1 + N(s,\ a)}}} \right) \tag{3}$$

where:

$N(s,\ a)$ = number of times the action a has been played from state s
$h(s,\ a)$ = heuristic function; game dependent

3.4 Decaying Reward

To further limit the computational cost and unnecessary tree exploration, a decaying reward can be implemented that multiplies with the reward at every step. This enables a quicker win as states closer to the root are accessed and used more often.

$$R = R \times \boxed{\boldsymbol{\gamma^k}} \tag{4}$$

where:

R = original reward
γ = discount factor bounded between $[0, 1]$
k = distance of the current node from the root

4 Experimental Study

4.1 Implementation of `MCTSBias`

4.1.1 Baseline parameters

Table 1: Baseline parameters for experimentation

Parameter	Value
Levels	10
Repetitions	5
Iterations	200
Roll-out depth	12
Exploration constant C	$\sqrt{2}$
`CUSTOM_HEURISTIC`	0
`ADVANCED_HEURISTIC`	1
Discount factor γ	0.8

4.1.2 Progressive bias

Progressive bias was added to the `uct` method within the `SingleTreeNodeBias` class using two approaches:

1. `customBias`: The `CUSTOM_HEURISTIC` with a value of 0 was added to the progressive bias function resulting in the effective cancellation of the progressive bias term. This might seem counter-intuitive but allowed the comparison of vanilla `MCTS` and `MCTSBias` with *only* decaying rewards.

$$a = \arg\max_{a \in A(s)} \left(Q(s,\ a) + C\sqrt{\frac{\ln \mathrm{N(s)}}{N(s,\ a)}} + \cancel{\frac{\mathbf{0}}{\mathbf{1 + N(s,\ a)}}} \right) \tag{5}$$

2. `advancedBias`: The `ADVANCED_HEURISTIC` with a value of 1 was also added to the progressive bias function for some tests and as the value of $N(s,\ a)$ increases, the value of the added bias *reduces*.

$$a = \arg\max_{a \in A(s)} \left(Q(s,\ a) + C\sqrt{\frac{\ln \mathrm{N(s)}}{N(s,\ a)}} + \frac{\mathbf{1}}{\mathbf{1 + N(s,\ a)}} \right) \tag{6}$$

4.1.3 Decaying reward

The decaying reward was added to the `rollOut` method within the `SingleTreeNodeBias` class as the discount factor is multiplied with the reward at the end of each roll-out simulation. A value of 0.8 for γ was chosen to enable a conservative long-term strategy, where the unnecessary exploration of the game tree was reduced by 20% as per the depth of the current node.

4.2 Results

4.2.1 Team mode[1]

Table 2: `customBias` comparison

Player	Wins (%)	Ties (%)	Loss (%)
MCTS	26	24	50
MCTSBias	50		26
MCTS	26		50
MCTSBias	50		26

Table 3: `advancedBias` comparison

Player	Wins (%)	Ties (%)	Loss (%)
MCTS	36	24	40
MCTSBias	40		36
MCTS	36		40
MCTSBias	40		36

Table 4: Roll-out depth increased from 12 to 20 with partial observability and `customBias`

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	30	8	62
RHEA	62		30
MCTS	30		62
MCTSBias	62		30

Table 5: Roll-out depth increased from 12 to 20 with partial observability and `advancedBias`

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	32	6	62
RHEA	62		32
MCTS	32		62
MCTSBias	62		32

[1] Corresponding colours denote team match-up.

4.2.2 Free-For-All (FFA)

Table 6: `customBias` comparison

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	0	0	100
RHEA	18	6	76
MCTS	18	18	64
MCTSBias	44	20	36

Table 7: `advancedBias` comparison

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	0	0	100
RHEA	30	10	60
MCTS	22	8	70
MCTSBias	32	14	54

Table 8: Roll-out depth increased from 12 to 20 with partial observability and `customBias`

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	0	0	100
RHEA	32	2	66
MCTS	28	10	62
MCTSBias	28	12	60

Table 9: Roll-out depth increased from 12 to 20 with partial observability and `advancedBias`

Player	Wins (%)	Ties (%)	Loss (%)
OSLA	0	0	100
RHEA	24	6	70
MCTS	28	16	56
MCTSBias	30	16	54

5 Discussion

5.1 Analysis

The aim of the study was to explore the utilisation and applicability of the Monte Carlo Tree Search (MCTS) algorithm within the *Pommerman* game environment. Two main modifications were instituted to improve on the vanilla `MCTS` agent: progressive bias, and decaying rewards.

Table 1 from Section 4.1.1 highlights the baseline parameters of the experiment that were modified throughout the experimentation phase with various outcomes as displayed in Section 4.2. The pertinent factors are the heuristic values and the discount factor. The heuristic values were left unmodified to keep a measure of improvement on MCTS by using Upper Confidence Bounds for Trees (UCT), whilst a discount factor of 0.8 (nearer to 1) was chosen to enable the agent to take a long-term strategy as the game objective requires a *delayed* reward (winning at the end of game) instead of instant gratification.

Section 4.2.1 displays the results for the TEAM mode of the game. Majority of the experiments were run using teams of alternating players of `MCTS` and `MCTSBias`. The `MCTSBias` agents integrated with `customBias` (Table 2) and `advancedBias` (Table 3) outperform the vanilla `MCTS` by winning 63% and 11% of the games, respectively within the same teams. Evidently, `MCTSBias` agents with `customBias` outperform the same agents with the `advancedBias` option. As highlighted in Section 4.1.2, the `customBias` agent has effectively no progressive bias as the heuristic value is 0, thereby, cancelling the effect of the bias function. However, the decaying reward implementation still holds. This provides a valuable insight into `MCTSBias`, where the agent with no bias function is more performant due to the selection strategy for *Pommerman*, where the nodes closest to the root are not necessarily the optimum choices. Additionally, as the "... [heuristic function $h(s,\ a)$] is computed by using pattern matching ... in combination with the proximity to the last move. The exact formula uses coefficients tuned with simulated annealing. The time consumed to compute $[h(s,\ a)]$ is in the order of a millisecond, which is around 1000 times slower than playing a move in a simulated game" [8]. Changing the roll-out depth from 12 to 20 with additional players (Table 4–5) in partial and fully observable modes resulted in further improvement as the `MCTSBias` agents again outperform the vanilla `MCTS` by winning 70% and 64% of the games, respectively within the same teams. The architecture of `RHEA` is beyond the scope of this research, but it is worth noting that the `RHEA` and `MCTSBias` agents perform comparably, when placed in opposing teams.

Section 4.2.2 highlights the results for the Free-For-All (FFA) mode of the game. The experiments were run using 4 players: `OSLA`, `RHEA`, `MCTS` and `MCTSBias`. The `MCTSBias` agents integrated with `customBias` (Table 6) and `advancedBias` (Table 7) outperform the other agents `MCTS` by winning each game. Furthermore, the `MCTSBias` agents with `customBias` outperform the same agents with the `advancedBias` option. The roll-out depth was also changed for the FFA mode (Table 8–9) resulting in similar performance to `RHEA`, whilst drawing and outperforming the vanilla `MCTS`. Since, from [4], "... in full observable mode, MCTS is able to beat simpler and hand-crafted solutions", the `MCTSBias` agent shows definitive improvement on vanilla `MCTS`.

5.2 Future work

Further improvements to the experiment for future work include:

- The game can be run with an increased number of levels, repetitions, iterations, and roll-out depth to fine-tune the performance of the `MCTSBias` agent.
- Experimenting with different heuristic values instead of using 0 and 1 would further highlight the applicability of progressive bias in a game such as *Pommerman*.
- The discount factor can be varied to see how the agent performs with myopic or long-term strategies.
- Exclusive experimentation to test the performance of the new agent (`MCTSBias`) as compared to the strongest opponent (`RHEA`) faced during this research.

6 Conclusion

The study was conducted to test the performance of a new agent capable of successfully participating in the *Pommerman* game. The study was executed by improving the vanilla MCTS algorithm by adding a progressive bias function and decaying reward strategy. The results shows marked improvement in the new agent `MCTSBias` that have been consequently analysed in detail. Further improvements have also been suggested that would enhance the researchers understanding of the new agent and in what game domain, it would be suitably applicable.

References

[1] C. Gao, P. Hernandez-Leal, B. Kartal, and M. E. Taylor. *The Pommerman team competition or: How we learned to stop worrying and love the battle*. Dec. 2018. URL: https://www.borealisai.com/en/blog/pommerman-team-competition-or-how-we-learned-stop-worrying-and-love-battle/ (page 1).

[2] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2011.03.007. URL: https://www.sciencedirect.com/science/article/pii/S000437021100052X (page 1).

[3] Edgar Galván-López, Ruohua Li, Constantinos Patsakis, Siobhán Clarke, and Vinny Cahill. "Heuristic-Based Multi-Agent Monte Carlo Tree Search". In: *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*. 2014, pp. 177–182. DOI: 10.1109/IISA.2014.6878747 (page 1).

[4] Diego Perez-Liebana, Raluca D. Gaina, Olve Drageset, Ercüment İlhan, Martin Balla, and Simon M. Lucas. "Analysis of Statistical Forward Planning Methods in Pommerman". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 15.1 (Oct. 2019), pp. 66–72. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/5226 (pages 1, 9).

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson, 2014 (pages 1, 5).

[6] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810 (pages 2, 3, 5).

[7] Steven James, George Konidaris, and Benjamin Rosman. "An Analysis of Monte Carlo Tree Search". In: Feb. 2017 (page 2).

[8] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. "Progressive Strategies for Monte-Carlo Tree Search". In: *New Mathematics and Natural Computation* 04 (Nov. 2008), pp. 343–357. DOI: 10.1142/S1793005708001094 (pages 5, 9).