Reinforcement Learning – Heuristic Planning

Qing Zhou

Mar, 2020

1 Introduction

The motivation of this assignment is to build a deeper insight into the heuristic search algorithm employed in game play, by implementing the Monte Carlo Tree Search (MCTS) algorithm on the game of Hex. The performance of this algorithm is analysed, by tested against the alpha-beta algorithm we are familiar with. And finally, we discuss the process of hyperparameter tuning and their implications. This report is laid out as follows: in section 2 we introduce the fundaments of MCTS and its applications, demonstrated with short examples; in section 3 we evaluate this algorithm by testing it against the alpha-beta algorithm, and present the results and their implications; in section 4 we discuss the way of hyperparameter tuning, and the implications and constraints; in the last section we conclude and present our final result for this assignment.

2 MCTS Hex

The corresponding python file for this section is 'MCtree_test.py', 'rltoy/mctree.py'.

The key to MCTS algorithm is to heuristic analyse and find the optimum move, by expanding the search tree based on random sampling in the search space. For each iteration in MCTS, the search is initialised at root node s_0, and the tree is constructed by adding subtrees. The playout is then performed at leave node, and the result backpropagated to the root node. To decide which node is better we assign a weight to each node, so that better nodes are more likely to be chosen in future playouts.

In overall, the MCTS algorithm consists of four phases: selection, expansion, simulation and backpropagation.

1. Selection: start from root node s_0 we select successive child nodes, until we reach a leaf node.
2. Expansion: unless on the leaf node the game is ended, we create more child nodes and choose node c from them.
3. Simulation: now proceed by one random playout at node c. The moves are self-played until the game is ended.
4. Backpropagation: in the end, we use the result of the playout backpropagated on the path of it from the root node, with two parameters updated: the visited count for all nodes and the win count.

The main challenge is the trade-off between exploration and exploitation, i.e. to find a high average win rate, with fewer simulations possible. And it is important to find the balance. The UCT formula

is given as follows[1]

$$UCT(j) = \frac{w_j}{n_j} + C_p\sqrt{\frac{\ln n}{n_j}} \tag{1}$$

where w_j is the number of wins in child node j, n_j the number of times child node j has been visited, n the number of times the parent node has been visited, and C_p is a tunable parameter for exploration/exploitation. The first term gives the level of exploitation whereas the second term can adjust the level of exploration. To test the MCTree we can do the following: create a board of size 2, and let the MCTree algorithm to be the 'Red' player. and set $C_p = 0$, itermax= INF. In this setup, the MCTree, if functioning, will always prefer a large winrate w_j/n_j. Now the board state is as follows: a red piece is placed at $(1, 0)$ and a blue piece at $(0, 1)$. According to the Hex rule, for the MCTree player one more piece at $(1, 1)$ is needed, to win the game. Hence given this situation, three simple assertions can be made

1. The number of visits at root node should be the number of wins at root, plus 1;
2. For the win node $(1, 1)$, the number of wins should be the number of wins of its root node, subtract by 1;
3. For the extra child node $(0, 0)$, the number of visits should be 1.

Below is the test results based on the above three assertions. "root.dump()" shows all the nodes visited, together with the number of wins and number of visits. And it can be seen the three assertions made passed without a problem.

```
board = HexBoard(2)
board.place((0, 1), board.BLUE)
board.place((1, 0), board.RED)
print(board)
mcs = MCTreeSearch(board, 'RED', itermax=-1, timedelta=1, Cp=0.)
root = mcs.search()
root.dump()
print(root.win == root.vis-1)
print(root.child['(1, 1)'].win == root.vis-1)
print(root.child['(0, 0)'].vis == 1)
```

```
   a b
 -----------------------
0 |- r |
1 | b - |
   -----------------------

Node root: w 24146, v 24147
 |- Node (0, 0): w 0, v 1
     |- Node (1, 1): w 0, v 0
 |- Node (1, 1): w 24146, v 24146
     |- Node (0, 0): w 24145, v 24145
True
True
True
```

Figure 1: A simple test for the MCTree algorithm. A board of size 2 is created, and for the "Red" player, one more piece at $(1, 1)$ is needed, to win the game.

Another test with minimax player was also performed.

```
   a b c d
 ------------------------
0 |- - - - |
1 | - - - r |
2 |  - - - - |
3 |   - - - - |
   ------------------------
```

```
   a b c d
 ------------------------
0 |b - - - |
1 | - - - r |
2 |  - - - - |
3 |   - - - - |
   ------------------------
```

```
   a b c d
 ------------------------
0 |b - - - |
1 | - - - r |
2 |  - - - - |
3 |   - r - - |
   ------------------------
```

```
   a b c d
 ------------------------
0 |b b - - |
1 | - - - r |
2 |  - - - - |
3 |   - r - - |
   ------------------------
```

(a) Board states in the game.

```
   a b c d
 ------------------------
0 |b b - r |
1 | - - - r |
2 |  - - - - |
3 |   - r - - |
   ------------------------
```

```
   a b c d
 ------------------------
0 |b b b r |
1 | - - - r |
2 |  - - - - |
3 |   - r - - |
   ------------------------
```

```
   a b c d
 ------------------------
0 |b b b r |
1 | - - - r |
2 |  - - r - |
3 |   - r - - |
   ------------------------

RED win
```

(b) Board state in the game.

It can be seen, the MCTree player can easily win the minimax player without a problem.

3 Experiment

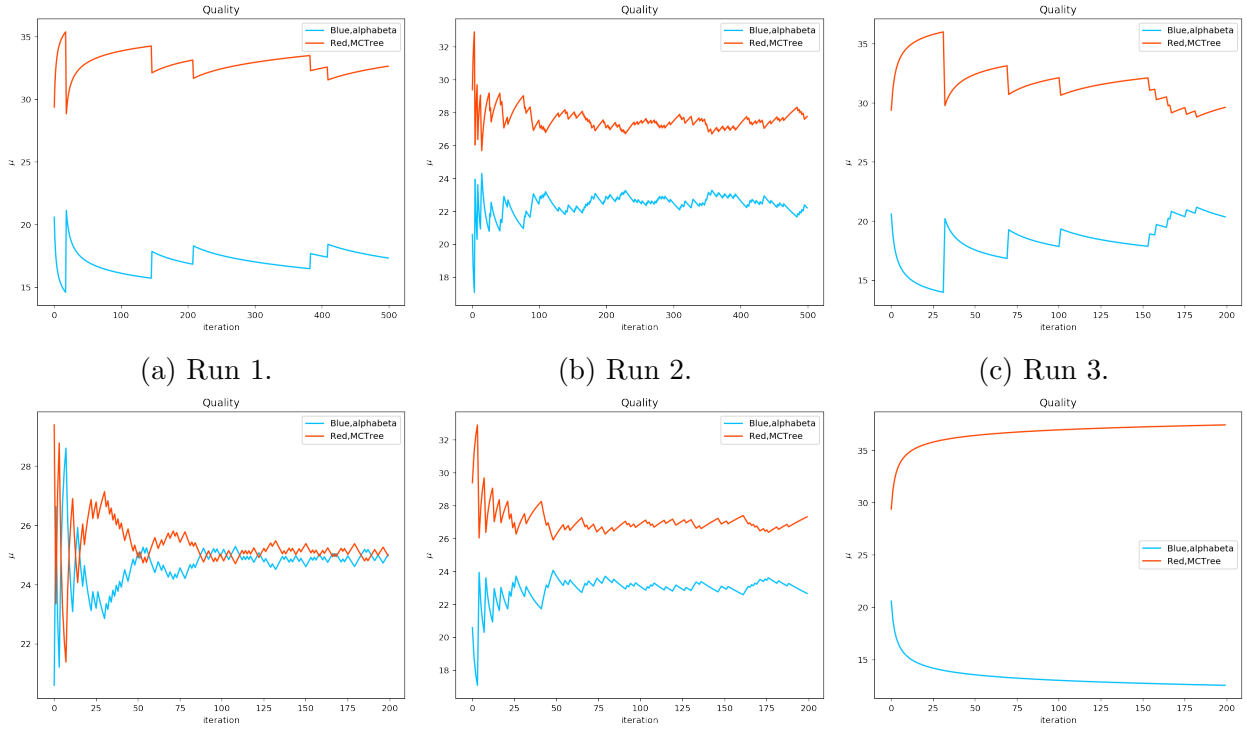
The corresponding python file for this section is 'experiment.py'.

Now to evaluate the strength of the search program we perform 6 different runs, the configuration for each run can be found in the table below.

Table 1: Simulation configurations for six different runs.

Board size = 4			
Run	depth	itermax	time limit
Run 1	20	10	-
Run 2	30	30	-
Run 3	30	100	-
Run 4	-	inf	1e-3
Run 5	-	inf	1e-2
Run 6	-	inf	1e-4
"BLUE": alphabeta,"RED": MCTree			

The Elo rating for the 6 runs are presented as follows. Overally, the MCTree player has a non-negnigible advantage. Comparing the results of Run 1, Run 2 and Run 3, it can be seen when increasing the number of iterations, the performance of the MCTree player is also improved, when increasing the search depth of the alphabeta player, the performance of him can be improved, however, it is not yet comparable to that of the MCtree player's. Comparing the results in the bottom panel,



(a) Run 1. (b) Run 2. (c) Run 3.

(d) Run 4. (e) Run 5. (f) Run 6.

it can be seen, when decreasing the time limit, the changes of winning for the alphabeta player is somewhat increased. In the case where time limit equals 0.01, he has a change of winning the MCTree player. In other cases he is always losing. The number of plays needed for a stabilised result for each case, can be found in the table below.

Table 2: Number of plays needed for stabilised result.

Run	Number of plays	Elapsed time [min]
Run 1	40	0.50
Run 2	50	1.08
Run 3	50	1.26
Run 4	100	0.06
Run 5	50	0.42
Run 6	50	0.06

4 Tune

The corresponding python file for this section is 'tune.py'.

The main challenge is the trade-off between exploration and exploitation, i.e. to find a high average win rate, with fewer simulations possible. There are two tunable hyperparameters which allows us to change the degree of exploration/exploitation desired. There are many ways which leads to the optimisation of the parameters, and a general way can be varying the parameters to calculate the number of operations needed, to obtain a converged result to optimize a polynomial. One of many formulas takes the following shape

$$\beta(n_j, \tilde{n_j}) = \frac{\tilde{n_j}}{n_j + \tilde{n_j} + 4b^2 n_j \tilde{n_j}} \tag{2}$$

So that for a fixed number of points, we can have the scatter plot of Number of operations vs. C_p finally at some C_p, converged. Then by varying N at the same time, we can find the optimum pair of C_p and N, to ultimately achieve the goal of obtaining converged results, yet at a lower cost, which is just another interpretation of the trade-off between exploration/exploitation. Now that understanding the goal, and with this simple setup based on the game of Hex, we can assess this in a much simpler way: since for each MCTree search, with varying values for itermax, it is understandable not every search can return a optimised node. Yet the node found may well be a sub optimum node. So when searching multiple times, with different choices of C_p, and N (itermax), we can expect the scatter in number of wins for the node found to be very different. But with careful choices of the pair of C_p and N, we can obtain a low scatter for number of wins, yet at a lower cost of N.

Knowing the above we can tune the hyperparameters as follows. First, choose a Hex board of size 5. Then, setup the board state as follows:

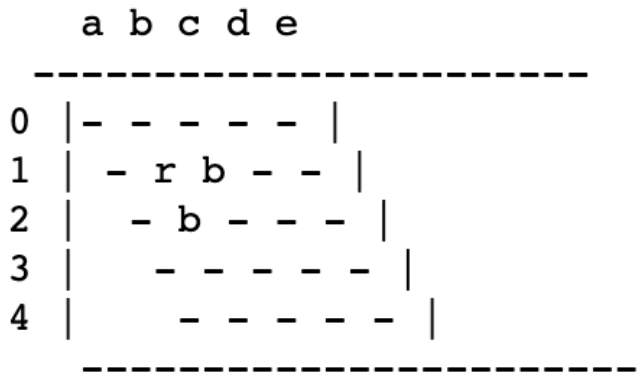


Figure 4: Board state for each search. Now there are three pieces already placed, and the MCTree is to perform given this state.

The reasons for this choice of board state are because

1. The state must be asymmetric for the optimum node to be able to converge;
2. The board size should be sufficiently large.

For each choice of C_P and N, we initialise the board state as above, and perform the MCTree search. Due to the randomness a scatter in the number of wins can be expected, as there is no guarantee to always find the optimum solution of the node.

We let C_p to be $1e-3, 1e-2, 1e-1, 1, 10, 100$ and let N to be $50, 100, 500, 1000, 2000$. Below shows the scatter in number of wins for different choices of C_p and N.

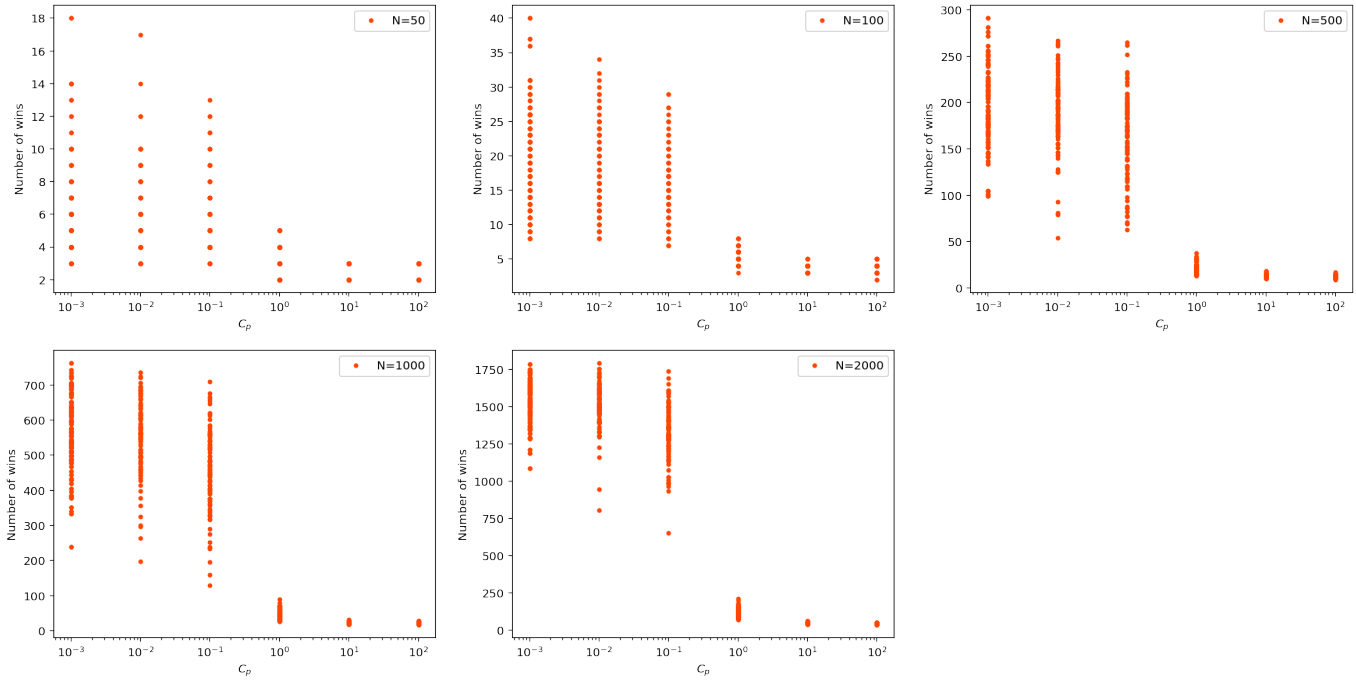


Figure 5: Number of wins calculated for various values of C_p and N (itermax).

It can be seen, for $C_p = 10$ and $N = 500$, we can have a sufficiently low scatter, with a lower cost in number of simulations. Hence the optimum choice of (C_p, N) is $(10, 500)$.

5 Conclusion

In this assignment, we implemented the MCTree algorithm on the game of Hex, to gain insights into the heuristic search algorithm which is key to reinforcement learning. The MCTree algorithm was evaluated by testing against the alphabeta algorithm. It was found, in general, the MCTS exhibits an advantage over the alphabeta algorithm, except for the case where the alphabeta player has unlimited search depth and time limit is 0.01 s – in this case the alphabeta player has a slight chance of winning. There are two hyperparameters which are tunable for the MCTS algorithm, C_p and N. To optimize them we plotted the scatter in number of wins for different choices of the parameters, and the optimum choice was found where a sufficiently low scatter can be achieved, with a lower cost of N. The best choice was found to be $(C_p, N) = (10, 500)$.

References

[1] Aske Plaat. Learning to play—reinforcement learning and games, 2020.