Reverse-Reversi AI Report

Zinan Li
12011517

***Abstract*—This article describes the principles, implementation and experimental results of an Reverse-Reversi AI that combines Monte Carlo tree search algorithm and Neural network.**

I. Introduction

Reversi is a strategy board game for two players, played on an 8×8 game board. It was invented in 1883. Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color. The objective of the game is to have the majority of disks turned to display one's color when the last playable empty square is filled.

In this project, we are required to make a Reverse-Reversi agent. The winning condition of Reverse-Reversi is opposite to the Reversi, the player who own fewer disks at the end of the game win. In simple terms, we need to design the "weakest" Reversi Agent.

Monte Carlo tree search algorithm and Minimax search with alpha-beta pruning are two most command algorithms using in board game agent. Minimax is a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. Minimax algorithm with alpha-beta pruning have great performance in game with fewer possible cases include Reversi. However, the performance of the Minimax algorithm greatly depends on its heuristic function. Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games. Different with Minimax, pure Monte Carlo tree search does not need an explicit evaluation function. In 2016, MCTS was combined with neural networks and achieve great success in various board games. [1]

In this project, I implement a Reverse-Reversi agent using both **pure Monte Carlo Tree search algorithm** and **Monte Carlo Tree search algorithm with neural network.** The submitted version is pure Monte Carlo Tree search algorithm because I use PyTorch build the neural network version, which is not allowed. The code is available at:

https://github.com/Buzzy0423/SUSTECH_UG_Resource/tree/master/CS303_AI/Project/project1

II. Preliminary

This section mainly explain the formulate description and notation of the problem.

A. Game state

Set the $S = \{s_0, s_1, s_2, ..., s_n\}$ be a set of game state, every $s_i \in S$ is the state of game board. Particularly, s_0 is the initial state of the board, which has four disks on the center of the game board. Every game state s can be represented as a 8×8 two-dimensional array, where $P_b = -1, P_w = 1, P_e = 0$ denotes respectively black disk, white disk and empty. For ease of coding, I define **canonical state** $\overline{s} = P_{currentplayer} \times s$ as a special game state. When the current player is black, the canonical board is the reverse board, and when the current player is white, it is the same board. **Via using the canonical board, the next step is always play by white**. All of the following game states s default to the canonical state unless otherwise noted.

B. Movement

The movement $a = (x, y)$ is an action of placing one disk at a specific position (x, y) on the game board. For computation convenience, a can be converted into an integer $a = 8x + y$, which is uniquely defined by the movement coordinate. $play(s, a)$ is a function returns the next state s' and next player p' after taking move a on state s.

C. Game result

For each game state s, there exist a corresponding game result $e \in E = \{-1, 0, 1\}$. $get\_game\_result()$ is a function mapping game states S to game result E, defined as below:

$$get\_game\_result(s) = \begin{cases} 0, & game\ not\ end\ in\ state\ s \\ 1, & white\ win \\ -1, & game\ draw\ or\ black\ win \end{cases}$$

Mention that the input of $get\_game\_result()$ is canonical board by default.

D. State count

For each state s, there exist a corresponding $c_s \in N$ represent how many times state s have been visited. $NS()$ is a function mapping from game states S to state counts.

E. Valid movements

For each state s, there exist a corresponding set v_s represent all the valid moves for state s. $VS()$ is a function mapping from game states S to valid moves sets.

F. Movement count

For each unique pair (s,a) of state s and movement a, there exist a corresponding $n_{(s,a)} \in N$ represent how many times movement a has been taken from state s. $NSA()$ is a function mapping from pairs of game state and movement to movement counts.

G. Movement reward

For each unique pair (s,a) of state s and movement a, there exist a corresponding $q_{(s,a)} \in [-1,1]$ represent the expected reward for taking action a from state s. $QSA()$ is a function mapping from pairs of game state and movement to expected reward, defined as below:

$$QSA(s,a) = \frac{\sum simulation\ result\ for\ taking\ (s,a)}{NSA(s,a)}$$

III. Methodology

In this section, I will focus on the Monte Carlo search tree algorithm, its workflow and detailed implementation.

A. Monte Carlo tree search algorithm

Monte Carlo tree search(MCTS) algorithm is base on random sampling for deterministic problems which are difficult or impossible to solve using other approaches. MCTS starts with the current game state s_0 and selects the most promising moves based on their UCB scores until reaches an unexplored or game end state s'. The UCB score of (s,a) is defined as:

$$UCB(s,a) = Q(s,a) + C\frac{\sqrt{2ln(NS(s))}}{NSA(s,a)}$$

C is a hyper parameter that controls the degree of exploration; by default, $C = 1$.

The algorithm then performs **roll-out** on s', which means it plays the game start from state s' randomly until it reaches the game end state s_e. Roll-out return the simulation result as $get\_game\_result(s_e)$. The result of the roll-out is used to update $Q(s,a)$ in the states on the path from s_0 to s'; this process is known as **back propagation**. The whole process is shown as Fig. 1.

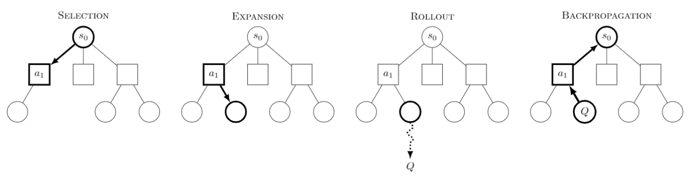

Fig. 1. MCTS work flow

B. Neural network

The neural network part of the MCTS is inspired by AlphaZero developed by DeepMind. Given the small size of the Othello game, I simplified the entire neural network in order for it to converge quickly.

The network is composed of four convolution layers, each with 32 channels and the kernel shape $(3,3)$, as well as four fully connected layers. The network's input is the game state s, which represents a tensor with the shape $(8,8)$. The first two convolution layers are zero-padding with padding size 1, making the shape of the tensors the same size as the input. Because the third and fourth convolution layers have no padding, the shape of the tensor after the fourth convolution is $(4,4)$. The tensor of all 32 channels are reshaped into a (512,1) shaped tensor and passed through two fully connected layers. After that, the tensor go through two different fully connected layers to get two outputs, **one for the expected reward** $v(s)$ **for input state** s **and the other for the probability vector** $\vec{p}(s)$ **over all possible place on the board**.

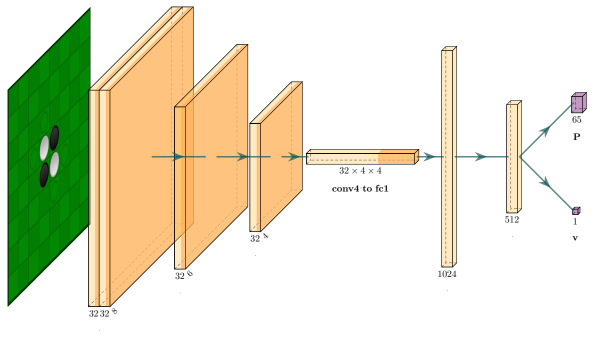

Fig. 2. Neural network sketch

The neural network is trained by playing against itself. At the end of each game of self-play, the neural network is provided training examples of the form $(s_t, \vec{\pi}_t, z_t)$. $z_t \in \{-1,1\}$ is the actual result of the game from the perspective of player at s_t and $\vec{\pi_t}$ is defined as:

$$\vec{\pi}(s) = \frac{N(s)}{\sum_a N(s,a)}$$

The neural network is then trained to minimise the following loss function:

$$l = \sum_t (v(s_t) - z_t)^2 - \vec{\pi}_t \cdot log(\vec{p}(s_t))$$

C. Implementation of pure MCTS

In the process of implementing the algorithm, I used the following optimization methods:

1) numba: I use numba to accelerate the program, I refactor the code. I separated out some heavily repetitive loops to ensure that they could be accelerated by the numba library's njit method.

2) space–time trade-off: The first version of my MCTS algorithm constructs a new search tree every time the function search is called, which is a massive waste of time. After optimization, the data of states is saved in the class variable, and function $search()$ can use previous simulation results. The comparative data will be presented in the experimental section.

3) canonical board: Via using the canonical board, the next step is always play by white. The number of all possible game state reduce by up to half, greatly improve the searching efficiency.

The pseudocode of MCTS is as below:

Algorithm 1 MCTS

```
Input: State $s$, Player $p$
Output: Best move $a$
 1: function $mcts$(State $s$, Player $p$)
 2:     $s = s \cdot p$ //canonical board
 3:     while not time out do
 4:         $search(s)$
 5:     end while
 6:     $s$, $a$ = $argmax(QSA(s, a))$
 7:     return $a$
 8: end function
 9:
10: function $search$(State $s$)
11:     if $s$ is game end state then
12:         return $-E(s)$
13:     end if
14:     if $s$ has not been visited then
15:         $E(s) \leftarrow get\_result(s)$ //already defined above
16:         $V(s) \leftarrow$ all valid moves $a$ of $s$
17:         $N(s) \leftarrow 0$
18:     end if
19:     $N(s) \leftarrow N(s) + 1$
20:     if $s$ have a move $a$ not been taken,  then
21:         $s', p' \leftarrow play(s, a)$
22:         $s' = s' \cdot p'$
23:         reward $\leftarrow roll\_out(s')$ //already defined above
24:         return reward
25:     end if
26:     $s, a \leftarrow argmax(s, a)$
27:     $s', p' \leftarrow play(s, a)$ //already defined above
28:     $s' \leftarrow s' \cdot p'$
29:     $v \leftarrow search(s')$
30:     update $QSA(s, a)$
31:     return -v
32: end function
```

D. Implementation of MCTS with neural network

Due to the length of the article, the pseudocode of MCTS with neural network will not be listed here. The detailed implementation can be found in my github repository.

The mainly difference between pure MCTS and MCTS with neural network is the result of roll-out is replaced by the output of the neural network.

E. Algorithm analysis

Because of the randomness of the algorithm, it is difficult to find the exact time complexity of MCTS. But what is certain is that the most time-consuming part of the algorithm is some of the methods that involve the rules of the game, such as find all valid move for state s or $play(s, a)$. These function involve a lot of loops, which can be speed up by numba. Detailed data will be shown in following part.

IV. Experiments

A. Experiments setups

1) Training setups: In the process of training the neural network, I run the training in the remote server. All the training data is from self-play game.

Remote server's environment is Python3.8(ubuntu20.04) and PyTorch 1.11.0. GPU of the server is Nvidia RTX3080 with Cuda version 11.3. CPU of the server is 12core Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz.

2) Test setups: The test was conducted using my own macbookpro with Python3.8.7 and Numpy1.22.2.

To assess the algorithm's efficiency, define R as the number of search rounds per second. Let the different versions of the algorithm(time limit for each move is 1s) play a total of three rounds with white and black pieces respectively, and then compute their R.

B. Experiments results

Total time spent on neural network training is 12 hours. The final loss is 1.8 for $\vec{p}(s)$ and 0.6 for $v(s)$

The final game result of MCTS vs optimized MCTS is 0:6.

The final game result of optimized MCTS vs MCTS with NN is 0:6.

The test result is shown as below:

TABLE I
Algorithm efficiency with numba

Algorithm	MCTS	Optimized MCTS	MCTS with NN
R	1005round/s	5297round/s	2657rounds/s

TABLE II
Algorithm efficiency without numba

Algorithm	MCTS	Optimized MCTS	MCTS with NN
R	503round/s	1174round/s	1280rounds/s

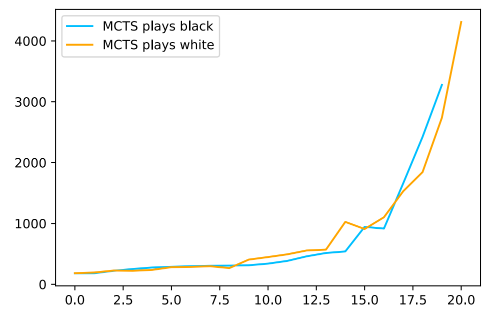

Fig. 3. R of MCTS

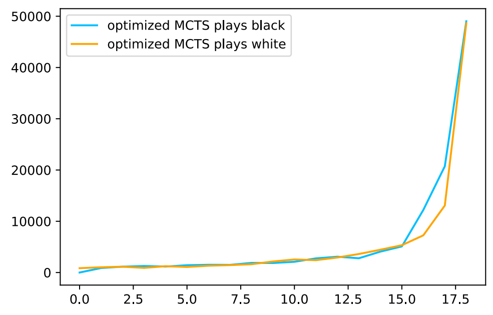

Fig. 4. R of optimized MCTS

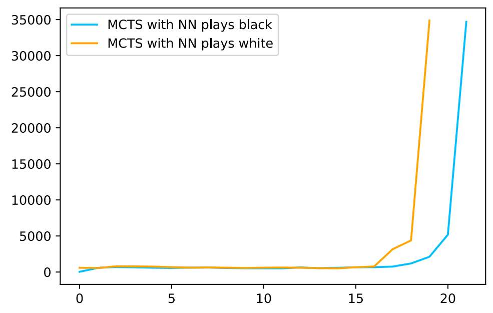

Fig. 5. R of MCTS with NN

C. Analysis

Comparing the efficiency of the algorithm with and without numba acceleration shows that **the improvement of numba on the execution efficiency of the algorithm is huge**. Especially in optimized MCTS, the efficiency gains of nearly five times.This result shows that the operational efficiency of the programming language has a significant impact on the effectiveness of the algorithm.

The comparison of the R of the algorithm shows that the **optimization methods mentioned in the methodology part above have a good effect on the efficiency of the algorithm.** For pure MCTS, the more efficient the algorithm, the better the judgment made.

The efficiency of MCTS with NN is lower than optimized MCTS when using numba. I presume it is because **neural network prediction takes up more time**, so the efficiency improvement is not as good as MCTS.

Because of the significant decrease in possible states, R **increases explosively at the end of the game for all algorithms.**

MCTS with NN is doing well in the test. It makes a better strategy than pure MCTS.

V. Conclusion

In this project, I implement pure MCTS and MCTS with neural network. Though I am disappointed that I cannot upload the neural network version of my MCTS, I learned a lot in the coding process.

Although MCTS is a good algorithm, **it lacks the use of priori knowledge**. This also causes MCTS to perform worse than Minimax when the number of search rounds is low. Neural networks compensate for this shortcoming of MCTS, it provide a relatively precise result for the state. Therefore the performance of MCTS with NN is the best.

The optimizations I made to the algorithm were very successful and even exceeded my expectations. During the optimization process, I refactor nearly half of my code, which takes a long time. **I assume that python works better with built-in data structure than customized data structures**.

The execution efficiency of Pyhton limits the effectiveness of the algorithm. Numba performs a pre-compilation-like process on the code, which greatly increase the execution efficiency.

The pure MCTS can be further optimized by store some expected reward of game states(game note). This optimization can reduce the impact of randomness to a certain extent and further improve the efficiency of the search.

References

[1] Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; Driessche, George van den; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda; Lanctot, Marc; Dieleman, Sander; Grewe, Dominik; Nham, John; Kalchbrenner, Nal; Sutskever, Ilya; Lillicrap, Timothy; Leach, Madeleine; Kavukcuoglu, Koray; Graepel, Thore; Hassabis, Demis (28 January 2016). "Mastering the game of Go with deep neural networks and tree search". Nature. 529 (7587): 484–489.