AI algorithm in Reversed Reversi

He Zhu, Southern University of Science and Technology

***Abstract*—This document is the CS303 project report from SUSTech.**

***Index Terms*—**$Reversed\ Reversi, \alpha - \beta\ pruning\ algorithm$

I. Introduction

REVERSI is a strategy board game for two players, played on an 8×8 uncheckered board. It was invented in 1883. Othello. Reversed Reversi is another similar game, which aims to keep a minimum number of pieces on the board. Although its rules are simple, but strategies are complex and varied, like Stable Discs, Mobility, Frontiers, Parity etc, which make great contribution to the final result. Due to some characters:
(1)Only two players played the adversarial game, and everyone aims to win and choose the best move to get the most benefits, also hope the opposite to get the worst situation.
(2)This game exist some strategies to evaluate the situation.
(3)This game have the limit situations and can be exhaustive enumeration.
Some Adversarial Search Algorithm will have a good performance on this kind of game. This project can help us cultivate thinking ability and the experience that makes idea to real code by python. This project can definitely train the coding ability of python and help us review the data structure and algorithm learning in previous class. We will also learn a lot about code optimization and game theory, which will help us a lot in our future study and research life.

II. Preliminary

The problem can be formulated as a the zero-sum game problem, in terminology, this means that two Agents in a defined, fully observable environment must take turns acting in such a way that at the end of the game utility values are always equal and of opposite sign. Adversarial search algorithm (Min-Max Search , $\alpha - \beta\ pruning\ algorithm$) was used to find the optimal policy generally. Firstly, some terminology and notation using throughout this report are as above:

TABLE I
The symbol table.

symbol	definition
δ	the current grid
δ'	the next grid
$A(\delta)$	the action that current player chooses
$V(\delta, color)$	the value that represent the situation of current grid for certain color
$color_{my}$	the color of the programming player
$color_{op}$	the color of the opponent
$\theta_{\delta,color}$	the candidate list of certain grid and color

The way to choose a strategy for adversarial search is an optimization problem, i.e., optimizing the utility function of the current situation:

$$A(\delta) = arg\ maxV(\delta'). \tag{1}$$

This equation means that you need to maximize next state value by current action.

However, due to limited arithmetic power, we usually cannot search to the endgame of the board and need to use $\alpha - \beta\ pruning\ algorithm$ with a finite number of search actions to improve the performance. So the problem is transformed into optimizing the utility function for the situation n steps after the current situation δ^n

$$A(\delta) = arg\ maxV(\delta^n). \tag{2}$$

So I mainly use the searching method based on evaluating current δ^n to solve this problem. Also I try some Reinforcement learning method like Monte Carlo Search Tree.

III. Methodology

A. evaluating function of δ

Considering the time, I use five features: stable pins number, Mobility, Frontier, Pieces difference and position value to construct the evaluating function.Then make a linear combination of these five features based on some constant c_i

$$\begin{aligned} V(\delta, color_{my}) = & c_1 \cdot Position(\delta, color_{my}) + \\ & c_2 \cdot Mobility(\delta, color_{my}) + \\ & c_3 \cdot Stable(\delta, color_{my}) + \\ & c_4 \cdot Frontier(\delta, color_{my}) + \\ & c_5 \cdot Diff(\delta, color_{my}) + \\ & c_6 \cdot Greedy(\delta, color_{my}) \end{aligned} \tag{3}$$

where

$$s[i,j] = \begin{cases} 1 & color[i,j] = color_{my} \\ -1 & color[i,j] = color_{op} \\ 0 & others \end{cases} \tag{4}$$

$$Position(\delta, color_{my}) = \sum_{i=0}^{8}\sum_{j=0}^{8} w[i,j] \cdot s[i,j] \tag{5}$$

$$Diff(\delta, color_{my}) = \sum(color_{op}) - \sum(color_{my}) \tag{6}$$

$$Mobility(color_{my}) = len(\theta_{\delta,color_{my}}) \cdot constant \tag{7}$$

$$C = \sum_{j=1}^{5} c_i \tag{8}$$

$w[i,j]$ means the value of certain position at position weight array, which is the most important features in evaluating function. Mobility reflects the selection steps number of current stage and a high mobility means a good stage for the player, however, $Mobility(color_{my}) == 0$ is also a good selection , which means the opposite will continue to go. Considering the high time complexity , we just count the Stable Pieces $Stable(\delta, color_{my})$ generated by the corner pieces. In the reverse Othello, we hope the Stable Pieces as little as possible. Frontier discs $Frontier(\delta, color_{my})$ are defined as discs that border one or more empty squares. More frontier discs have more advantages for the players. $w[i,j]$ and constant c_i is experienced arguments.

Noted that the $w[i,j]$ will change as the game progresses. The Simulated Annealing search and genetic algorithm [Gerges et al.(2018)] will be used to fine-tune position weight array $w[i,j]$ and constant c_i. The depth of the search still needs to be considered, for example, when the stages approaches to end (after 50^{th} stage), the evaluating function will focus on the $Diff(\delta, color_{my})$ to perform a final search with the 10 searching depth.

Another things I considered is about the balance of predict future and greedy selection for the current stage. As we all know the $n\ depth\ \alpha-\beta\ pruning\ algorithm$ focus on the n^{th} stage after the current stage but ignore the situation of the current stage. So the $Greedy(\delta, color_{my})$ was defined to add some greedy things to the evaluating function.

$$Greedy(\delta, color_{my}) = \frac{\sum_{j=0}^{n-1}(n-j)\cdot Reserve_j}{\sum_{j=0}^{n-1} j} \tag{9}$$

where the $Reserve_j$ means the number of the opposite's pieces that the player reserves in the j^{th} stage after the current stage.

B. Alpha-Beta pruning search

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the mini-max algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. The pseudo-code are as follows. The time complexity of this searching algorithm is $O(|\theta|^{depth})$, but actual time complexity is much smaller than it. The actual time complexity is largely influenced by the order of candidate. So we can use a pre-search with 1 depth to sort the candidate by the value. In theory, the effect of pruning will be much better. This optimization for pruning is very strong. When depth=7, it is generally 24s for one step to not sort the candidate list first, and the time to sort the candidate list first is generally 9s for one step, and 17s when the worst case. The depth will undoubtedly influence the effect of the algorithm especially near the end of the game. I tested the time taken to search for different depths using the Mac-Book Pro 2021 with M1 Max, noting that the next layer have 17 mobility. When depth=5, it is generally 1.2s for one step, however when depth=7, it is generally 24s for one step. This roughly matches the time complexity of $\alpha - \beta\ pruning\ algorithm$.

Algorithm 1 Alpha-Beta pruning search

```
abCut($\delta$,color,depth,alpha,beta)
    if $depth \le 0$ or $time\ end$
        $return\ V(\delta, color_{my})$
    if $mobility(\delta, color_{my}) \le 0$ :
        if $mobility(\delta, color_{op}) \le 0$ :
            $return\ V(\delta, color_{my})$
        else:
            $return\ abCut(\delta', -color, depth, alpha, beta)$
    if $color\ is\ color_{my}$ :
        for $candidate\ in\ \theta(\delta, color)$ :
            $\delta' = update(\delta, candidate)$
            value = abCut($\delta', -color, depth-1, alpha, beta$)

            if $value \ge alpha$ :
                $alpha = value$
            if $alpha \ge beta$ :
                break
        $return\ alpha$
    else
        for $candidate\ in\ \theta(\delta, color)$ :
            $\delta' = update(\delta, candidate)$
            value = abCut($\delta', -color, depth-1, alpha, beta$)

            if $value \le beta$ :
                $alpha = value$
            if $alpha \ge beta$ :
                break
        $return\ beta$
```

C. Monte Carlo search tree

Algorithm 2 Monte Carlo

```
mct($\delta$,color)
    choose $candidate\ in\theta_{\delta,color}$
        $\delta' = update(\delta, candidate)$
        if $game\ is\ over$ :
            back-Propagation(utility($\delta'$))
        $mct(\delta', -color)$
```

In computer science, 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. In that context MCTS is used to solve the game tree. This algorithm don't rely on the evaluating function of current grid. It just make random choice or make choice by the certain function until the end and make back propagation to the root. This algorithm theoretically will make a make a satisfied performance with enough repeat. In practice, I can only repeat roughly 10000 times when the middle of the game. It's not enough to utilize the advantage of the algorithm.

IV. Experiments

A. Setup

Two kinds of data-sets were used. One is a collection of downloaded data from platform matchmaking. the other is residual game data generated using AI self-gaming.The majority of the generated data set is concentrated in the 30 to 50 stage, with only 8% of the generated boards occupying at least 1 Corners and 73% occupying 4 X-squares. An example board generated from the AI self-gaming is above.

0	1	1	0	0	0	1	0
0	-1	1	1	1	-1	1	1
0	1	1	-1	1	-1	1	1
0	1	-1	-1	1	1	1	0
0	0	-1	-1	1	1	0	0
0	0	-1	-1	0	1	0	0
0	-1	-1	-1	-1	-1	-1	0
-1	0	-1	-1	-1	-1	0	-1

Fig. 1. An example board generated from the AI self-gaming

We use the production of these opening banks for initialization in the genetic algorithm.
Test Environment: Testing should make basic visualization. It's important to build the test platform. And the Visualization simulation matchmaking platform are as follows.

Environment: Python 3.10, numpy 1.44, numba 0.55.1

B. Results

TABLE II
The time table

searching method	depth	average time per game
Min-max	3	10.14s
Min-max	4	83.77s
Min-max with numba	4	97.42s
Alpha-Beta pruning Search	6	132.17s
Alpha-Beta pruning Search with pre-searching	6	89.94s
Alpha-Beta pruning Search with pre-searching and numba	7	27.25s

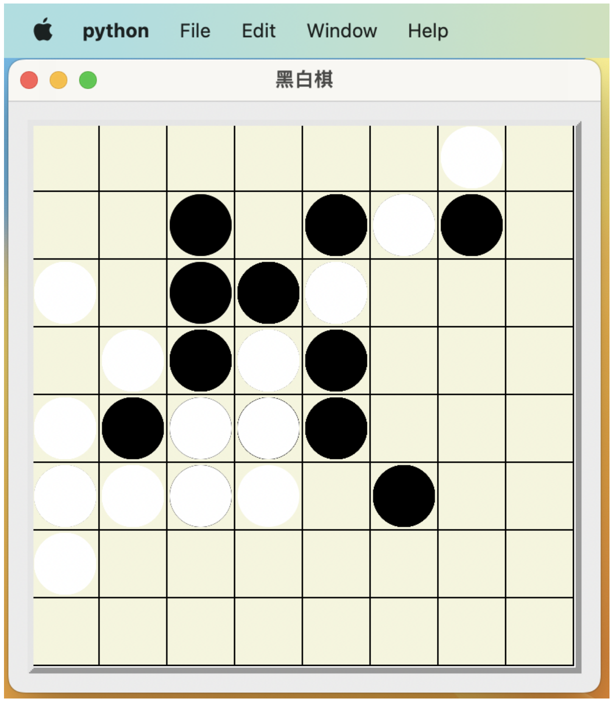


Fig. 2. Visualization simulation matchmaking platform

TABLE III
2-player game result

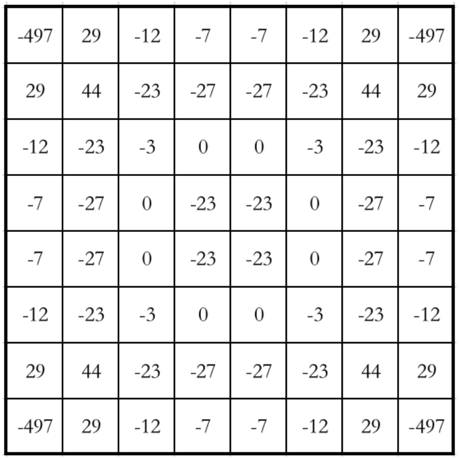
Game between different model	winning times of 10 games
4 depth A-B sorted pruning search vs 4 depth A-B sorted pruning search	5:5
4 depth A-B pruning search vs 6 depth A-B pruning search	8:2
4 depth A-B pruning search vs mct	10:0
4-depth A-B focus on Mobility vs 4-depth A-B	6:4
4-depth A-B focus on Greedy vs 4-depth A-B	4:6
4-depth A-B focus on Stable vs 4-depth A-B	7:3
4-depth A-B focus on Frontier vs 4-depth A-B	7:3
4-depth A-B focus on Position vs 4-depth A-B	5:5

The Experiments Environment is based on the basic parameter settings. Every Experiments we just modify only one kinds of parameter's constant c_i.

From the above experiments we can obtain the following results. (a) In terms of time consumption, the time consumed increases exponentially as the number of search layers increases, which is consistent with our expectations. The Numba library has a speedup for larger search levels, but it is not good for algorithms with smaller search levels, because Numba is very time consuming to compile, which slows down the speedup. Using Alpha-Beta pruning Search with pre-sorting will increase the speed significantly. In terms of performance: Since we set the number of layers in the evolutionary computation to 4, the parameters are adapted to a search with 4 layers, so 4 layers perform better than a deeper search. Regarding the contribution of various features to the evaluation function, we found that focusing on the $Greedy$ parameter is detrimental to the algorithm, focusing more on action in the middle stage, more on stabilizer in the late stage, and more on $Difference$ in the final search will improve the algorithm's fighting power more. Monte Carlo Tree Search require high performance, but my implementation of mct has low performance, so it is much less powerful than a-b pruning search with tuned parameters.

C. analysis

In summary, I chose the basic 4-depth a-b pruning as the final search function, noting that the 4-depth refers specifically to the stage between 10 and 50. First of all, the search depth is not the greater the better, the parameters settings need to match the search depth. Considering the balance between greedy and prediction, I deliberately set a smaller layers and it proved to be a good choice. In the Methodology part above, we hypothesized that a higher $Mobility$ would lead to more choices and the higher win rates, and $Stable$ would work later to avoid choosing the Corners position, meanwhile, excessive $Greedy$ and excessive future prediction would have an impact on the results. All of which are consistent with our expectations.
I use a genetic algorithm to determine the matrix of positions in each stage. Larger populations mean more possibilities but higher computational complexity. The parameter matrix initialized according to the genetic algorithm performs better and as my expected, for example, the weights of the edges and corners are small and the weights of the X-squares are large, the result are as follows.



-497	29	-12	-7	-7	-12	29	-497
29	44	-23	-27	-27	-23	44	29
-12	-23	-3	0	0	-3	-23	-12
-7	-27	0	-23	-23	0	-27	-7
-7	-27	0	-23	-23	0	-27	-7
-12	-23	-3	0	0	-3	-23	-12
29	44	-23	-27	-27	-23	44	29
-497	29	-12	-7	-7	-12	29	-497

Fig. 3. Position Matrix.

As the analysis in the Methodology part: The time complexity of this searching algorithm is $O(|\theta|^{depth})$, but actual time complexity is much smaller than it. The conclusion is fully consistent with the results of our tests. Mct must be complementary with speed optimization, but I encounter some problems to add numba. The algorithm can't make well in the searching times less than 10000 and performed very poorly.

V. Conclusion

The effect of Alpha-Beta pruning search is mostly determined by the evaluating function and the searching depth. As the analysis in the METHODOLOGY part and EXPERIMENT Part:
1.The depth with certain evaluating function can make best result.
2.The $Stable$, $Greedy$, $Mobility$, and the position matrix make a great contribution to the Game.
3.Use Genetic algorithm to obtain better position matrix compare to Manual adjustment.

The best method to measure the current grid is by CNN network. Only measured by weighted matrix and mobility, it can't utilize the relative position of piece. CNN can utilize the position information of the pieces.
A fast algorithm will influence on optimization and testingespecially for the MCT. Some trick use to speed up my code is using yield instead of list, using C-style coding your python, then use numba.
My best ranking was 20, but gradually slipped at the end. As long as you upload your algorithm to the website, other people may continuously play with you to find a better code aiming at your code. The radical problem is that weight matrix and mobility can't really determine the value of the δ. The result is that many people are stilling modifying the argument until end.

I will try other method in the future, like Alpha-Zero. And I will continue to optimize my code to speed up the the mct .

References

[Gerges et al.(2018)] Firas Gerges, Germain Zouein, and Danielle Azar. 2018. Genetic algorithms with local optima handling to solve sudoku puzzles. In *Proceedings of the 2018 international conference on computing and artificial intelligence*. 19–22.