Sustech CS303 Artificial Intelligence Project1: Reversed-Reversi

Wu Jingfu, 12011437
Department of Computer Science and Engineering

***Abstract*—This project realizes an Reversed-Reversi game agent based on adversarial search algorithm and local search algorithm learnt from Sustech CS303, which finally reaches a stable search depth of 5 with a proper evaluation function, deepening the author's comprehension of AI.**

***Index Terms*—Artificial Intelligence, Reversed-Reversi, Alpha-Beta Search, Genetic Algorithm.**

I. Introduction

THIS project is based on the game *Revered-Reversi*,which is the reversed version of classical Othello game. The rule is that on a 8X8 chessboard, you turn the opponent's pieces into our pieces by clamping them with your own pieces in every step. When both of the players have no moves available, the player with less chess on the chessboard wins.

Since the game itself is algorithm-independent and has relatively simple rules, it provides an excellent platform for developing and verifying game search algorithm. However, the complexity in practical makes *Revered-Reversi* lack perfect mathematical solutions [1].

This project is intended to realize a *Revered-Reversi* AI based on game tree search algorithm, deepening the comprehension of search algorithm and artificial intelligence designing process.

II. Preliminary

The problem can be divided into two sub-problems:

1) **Alpha-Beta Search Algorithm**

   a) Initial State: the current chessboard(denoted by s)
   b) Player(s): the current player
   c) CandidateList(s): returns a set of legal actions
   d) Transition function:
   For an action a, *move_one_step(a,s)* returns the result state of the action.
   e) Terminal test: True if the game ends or the search is cut, False otherwise
   f) Utility: For a state s, *eval(s)* returns the evaluation value of the state.

2) **Genetic Algorithm**

   a) Population: a set of Revered-Reversi agents
   b) Fitness Funciton: the final score of each agent in competition(described in section 3.2.3)
   c) Reproduction: randomly exchange slice of weight matrices between agents
   d) Mutation: randomly change parts of some agents' weight matrices.
   e) Generation: decides the looping time.

The notations are elaborately described in table1.

TABLE I: Notations

Notations	Definition
Variables	
AI	class name of the chess agent
$color$	-1:black,0:empty,1:white (int8)
$chessboard$	chessboard with -1,0,1 (np.ndarray)
$move$	chess placement location (Tuple(int8,int8))
$\alpha\beta$ Search Algorithm	
r	the game round (uint8)
d_{cut}	the cut-off depth (uint8)
d	the current depth (uint8)
b	maximum branching factor of the searching tree
α	largest value for Max across seen children (current lower bound on Max's outcome).
β	lowest value for Min across seen children (current upper bound on Min' s outcome).
Genetic Algorithm	
I	weight scaling upper-bound ($i \in \{0.25, 0.5, ..., I\}$)
S	population size
n	number of generations
k	number of parents kept for next generation
W_m	weight matrix of mobility (np.ndarray)
W_s	weight matrix of stability (np.ndarray)
W_{pos}	weight matrix of positions (np.ndarray)

III. Methodology

A. Developing Procedure

As is presented below, this project is developed in three phases with methodologies accordingly:

1) Game rules and search algorithm realization
2) Calculation acceleration
3) Weight adjustment

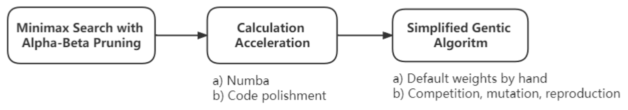
Minimax Search with Alpha-Beta Pruning
Calculation Acceleration
Simplified Gentic Algoritm
a) Numba
b) Code polishment
a) Default weights by hand
b) Competition, mutation, reproduction

Fig. 1: General Workflow

B. Detailed design

1) Alpha-Beta Search algorithm

This is the very basic of this project, enabling the agent to make decisions via adversarial search.

Algorithm 1 Alpha-Beta Search

```
1: function $\alpha\beta$SEARCH($chessboard, d_{cut}, color, r$)
2:     function MAXVALUE($board, \alpha, \beta, d$)
3:         $my, op \leftarrow candidateList(color, board)$
4:         if $d > d_{cut}$or not $my$ then
5:             return $eval(color, board, d{+}r, my, op), None$
6:         end if
7:         $v, move \leftarrow -inf, None$
8:         for $m$ in $my$ do
9:             $board1 \leftarrow board.copy()$
10:            move_one_step(m, color, board1)
11:            $v2, \_ \leftarrow MinValue(board1, \alpha, \beta, d+1)$
12:            if $v2 > v$ then
13:                $v, move \leftarrow v2, m$
14:            end if
15:            if $v2 \geq \beta$ then
16:                return $v, move$
17:            end if
18:            $\alpha \leftarrow max(v, \alpha)$
19:        end for
20:        return $v, move$
21:    end function
22:    function MINVALUE($board, \alpha, \beta, d$)
23:        $my, op \leftarrow candidateList(color, board)$
24:        if $d > d_{cut}$ or not $op$ then
25:            return $eval(color, board, d{+}r, my, op), None$
26:        end if
27:        $v, move \leftarrow -inf, None$
28:        for $m$ in $op$ do
29:            $board1 \leftarrow board.copy()$
30:            move_one_step($m$, $-color$, $board1$)
31:                                        ▷ $-color$: color of the opponent
32:            $v2, \_ \leftarrow MaxValue(board1, \alpha, \beta, d+1)$
33:            if $v2 < v$ then
34:                $v, move \leftarrow v2, m$
35:            end if
36:            if $v2 \leq \alpha$ then
37:                return $v, move$
38:            end if
39:            $\beta \leftarrow min(v, \alpha)$
40:        end for
41:        return $v, move$
42:    end function
43:    return MAXValue($chessboard, -inf, inf, 0$)
44: end function
```

In addition to the search algorithm itself, the evaluation function (denoted as eval() as is described in Preliminary) is designed based on trade-offs of accuracy and efficiency, as is illustrated below in Algorithm 2.

Algorithm 2 Evaluation function

```
1: function EVAL($color, chessboard, round, my, op$)
2:     if not $my$ and not $op$ then
3:         return $1e5$ if color wins else $-1e5$
4:     end if
5:     $my\_potential, op\_potential \leftarrow$ number of frontier chess
6:     $my\_stability, op\_stability \leftarrow$ number of stable chess
7:     $position \leftarrow sum(multiply(chessboard, W_{pos}[round]))*color$
8:     $mobility \leftarrow sum(multiply(Wm[round], array(len(my), len(op), my_potential, op_potential)))$
9:     $stability \leftarrow sum(multiply(Wm[round], array(len(my), len(op), my_stability, op_stability))$
10:    return $position + stability + mobility$
11: end function
```

The definition of frontier and stable chess are as following.

Frontier chess: the chess which border one or more empty squares and are not in edge squares [2]

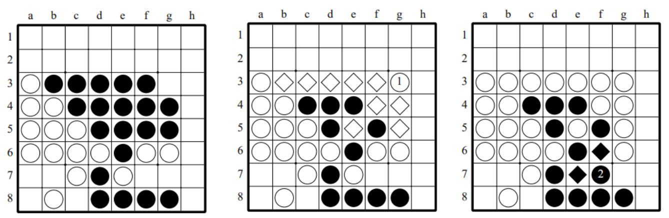

Fig. 2: frontier chess

Stable chess: the chess which are protected by the corner and can never be flipped [2]

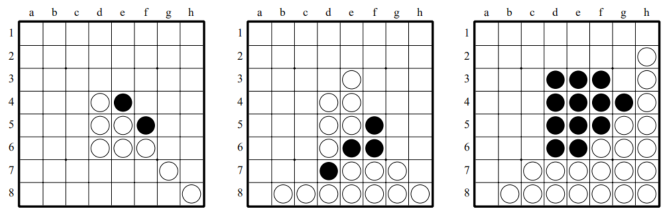

Fig. 3: stable chess

The detailed implementations of calculating frontier chess and stable chess are realized with references to articles [1] [3], where only stable chess on the edge squares are calculated, trading off for efficiency.

2) Calculation Acceleration

This project first tried to use two 8-bit-integers for chessboard state representation, whose **bit-operations** were thought to be of great help for acceleration.

However, the outcome is far from satisfaction. When trying to use numba.njit (add *@njit(cache=True)* as functions' decorator) ,exceptions are thrown by numba when it comes to bit-operations, which can't be fixed after long-term trying.

Then bit-operations are abandoned and all positional operations are adjusted to calculation **APIS from numpy**, which is consistent to **numba.njit** [4].

Additionally, many intelligent usages of numpy for matrix operations and weight calculating are implemented with reference to the previous work in github [5], which is significant for reaching the acceleration goal.

3) Genetic algorithm

The procedure of this algorithm is illustrated as below.

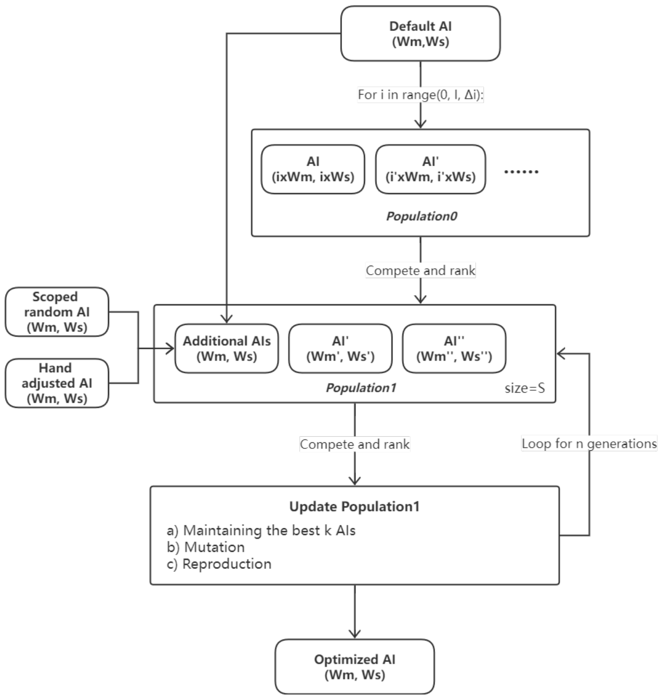

Fig. 4: Genetic algorithm

Default AI is initialized by concluding from experience and observing competitive performance on the platform, using W_{pos} from CS303 lab tutorial.

$$\begin{bmatrix} 1 & 8 & 3 & 7 & 7 & 3 & 8 & 1 \\ 8 & 3 & 2 & 5 & 5 & 2 & 3 & 8 \\ 3 & 2 & 6 & 6 & 6 & 6 & 2 & 3 \\ 7 & 5 & 6 & 4 & 4 & 6 & 5 & 7 \\ 7 & 5 & 6 & 4 & 4 & 6 & 5 & 7 \\ 3 & 2 & 6 & 6 & 6 & 6 & 2 & 3 \\ 8 & 3 & 2 & 5 & 5 & 2 & 3 & 8 \\ 1 & 8 & 3 & 7 & 7 & 3 & 8 & 1 \end{bmatrix} \tag{1}$$

Due to the limitation of hardware setting and time, the hyperparameters are set as below:

1) $S \leftarrow 10$
   (including one *Scoped random AI* and one *Hand adjusted AI*)
2) $n \leftarrow 10$
   (each generation stops for hand adjustment)
3) $k \leftarrow 2$
   (based on the fact that same scores may occur)

The competing procedure is implemented by mapping each AI to an integer(score) and play to each other twice(on the first move and not), where the winner gets 5 scores and the loser loses 5 points. The final rank is decided by the scores.

Population0 is generated for deciding the range of weight scaling, adapting to the fixed W_{pos}.

Population1 is the main population, where *Default AI* and one *Scoped random AI* are added each term, making sure that the *Optimized AI* is better than the original version.

Hand adjusted AI is adjusted from the champion of each term, which is obtained by analysing practical competitive performance on the course platform, together with scaling W_{pos} in different game phase decided by $round$.

C. Analysis

1) Alpha-Beta Search Algorithm
   • Completeness: Yes, it always reaches d_{cut}.
   • Time Complexity:
   1+b+b^2+b^3 +...+ $b^{d_{cut}}$= $O(b^{d_{cut}})$
   • Space Complexity: $O(b * d_{cut})$
   • Optimal: No. With ideal ordering, the time complexity reaches $O(b^{d_{cut}/2})$

2) Genetic Algorithm
   • Completeness: Yes, it always finishes the generation.
   • Time Complexity:
   $O(C_S^2 * 64 * O(b * d_{cut})) = O(b * d_{cut})$
   • Space Complexity: $O(S)$
   • Optimal: Yes

From the analysis above, it can be concluded that except for the hyperparameter d_{cut}, b(the maximum branching factor of the searching tree) is the deciding factor of the overall performance, which has been optimized by removing the repeated actions returned by candidateList(color,state)

IV. Experiments

The following are the testing environment and result of this project.

TABLE II: Testing environment

Software	Windows10
Hardware	-R7-5800H -16GB RAM -512GB
Data	Sakai-local_code_check random generator in Python
Tools	-PyCharm Community Edition 2021.3.1 -Anaconda Python 3.9 -Numpy 1.21.5 -Numba 0.55.1

TABLE III: Result

Factor	Result
$\alpha\beta$ Search Algorithm	
Running time	a maximum of 4.1877s in local judge with $d_{cut} = 5$ (The time-d_{cut} chart is shown as Fig.5)
Optimality	this project has tried to order the actions and store patterns in order to reach the ideal ordering of $\alpha\beta$pruning, but it turned out that the calculating cost outweighs the time saved in searching algorithm
Genetic Algorithm	
Running time	860.0523s in local judge with $d_{cut} = 5, S = 2, n = 1, k = 2$
Optimality	Due to the time limitation, the adjustment $d_{cut} \leftarrow 2$ decreases running time to 232.4140s in local judge with $S = 10, n = 1, k = 2$, resulting in a total experiment time of 1 hour for 10 generations.

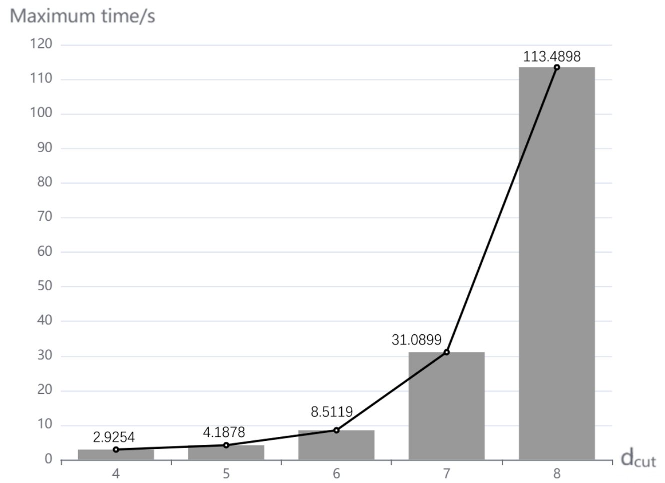

Fig. 5: $\alpha\beta$ Search Algorithm performance

Analysis

A stable searching depth of $d_{cut} = 5$ meets this project's expectation, resulting in relatively good performance on the course platform despite the instability of $numba.njit$.

As is presented in Fig.5, the time complexity $O(b^{d_{cut}})$ grows exponentially with d_{cut}. Additionally, decreasing d_{cut} significantly accelerates the running of Genetic Algorithm.

V. Conclusion

The Alpha-Beta Search Algorithm is relatively easy to implement, whose performance increases with its cutoff depth. However, its performance is sensitive to the design of the corresponding evaluation function. This project's experimental result meets the time complexity analysis.

This project taught me a significant lesson that only after assuring the correctness of the main algorithm itself can the project process on with efficiency. A great amount of time has been wasted when struggling with the evaluation function of a wrongly implemented search algorithm, which makes me bear it in mind that I should verify my project step by step other than urgently advance it.

References

[1] Li Xiaozhou, *Design and Implementation of Othello based on Improved Chess Game Tree Search*. South China University of Technology. CNKI, 2010.

[2] Othello and A Minute to Learn...A lifetime to Master are Registered Trademarks of Anjar Co., ©1973, 2004 Anjar Co., All Rights Reserved Copyright © 2005 by Brian Rose

[3] Eternal AddiCtion, *How to calculate stability in Reversi game*. Available:https://ask.csdn.net/questions/715847

[4] *Numba: A High Performance Python Compiler*.https://numba.pydata.org/

[5] https://github.com/Tokasumi/CS303-Fall2021-Reversed-Reversi