Reserved Reversi: Project Report

Yiqian Huang

I. Introduction

THE course project is based on the well-known board game ***Reversi*** [3] and defines a new game ***Reserved Reversi***: it has the same board and rules as *Reversi*, but with the difference that the winner is determined in reverse. *Reversi* requires the side with the higher number of final pieces to win, while *Reserved Reversi* requires the side with fewer pieces to win. The goal of the course project is to design an AI algorithm for *Reserved Reversi* that can play either side of black and white against other humans or agents in order to win as much as possible. In this project, I implemented an algorithm based on alpha-beta pruning search and a static evaluation function, and used a genetic algorithm to improve the accuracy of the evaluation function. For any legal position, it can return a drop point to get closer to the winning position. The code of the algorithm is implemented in `python` language and uses `numba` optimization, which greatly improves its efficiency in real-world applications. In the Online Judge server of this course, my code achieved the second place in the ranking. Furthermore, my algorithm can be easily extended to the classic *Reversi*.

II. Preliminaries

A. Reserved Reversi

Reserved Reversi is a strategy game played by both white and black players on an 8×8 board. To make it easier to understand, we will first describe the rules of the game in natural language. We denote (a, b) as the position of row a and column b on the board. Each piece is divided into black and white, and the game starts with two black discs on $(4, 3)$, $(3, 4)$ and two white discs on $(3, 3)$, $(4, 4)$ (see Figure I). Each game is always played by black first, and then both players take turns playing.

	0	1	2	3	4	5	6	7
0								
1								
2								
3				○	●			
4				●	○			
5								
6								
7								

TABLE I: Initial Board

When it is a player's turn to play, he must choose a **legal position** on the board and place one of his discs there. A position is legal if and only if:

- There are no discs on this position.
- One or more opponent's discs are sandwiched between this position and the position with his own discs in any of the eight directions (horizontal, vertical and diagonal) around this position.

When the player makes a move, all opponent's pieces that are sandwiched in all eight directions must be flipped to the player's side. If the player does not have a legal position, he must skip this move, allowing his opponent to make consecutive moves. If there is one or many legal positions, the player must also make a move on this turn. The game ends when both players have no legal position. At the end of the game, the player who has **fewer** pieces of his color than his opponent is the winner. Note that the discs can neither be removed from the board nor moved from one position to another.

B. Formulation

In order to precisely describe the rules and current state of the game in the algorithm, we need to do mathematical formulation of the rules. First, we represent a **board** as a sequence $\Omega = \{p_{0,0}, p_{0,1}, \ldots, p_{7,7}\}$, where $p_{i,j} \in \{-1, 1, 0\}, \forall i \in \{0, \ldots, 7\}, j \in \{0, \ldots, 7\}$. The constants $-1, 1, 0$ indicate that a position is black, white or no discs respectively. When the context is clear, we denote $p_{\Omega,i,j}$ as the element $p_{i,j}$ that is in Ω. Further, we denote a **state** as $s = (\kappa, \Omega)$, where $\kappa \in \{-1, 1\}$ represents the current turn is black or white respectively, and Ω is a sequence representing the board. The board represented in Figure I is denoted as Ω_0, hence the game starts with the initial state $s_0 = (-1, \Omega_0)$. When the player κ place a piece at board Ω, then we change the state $s = (\kappa, \Omega)$ to a **successor state** $s_{next} = (-\kappa, \Omega_{next})$. We denote the set of successor states of s as $next(s)$. Specifically, if the player κ has no legal position, then the successor state of s is only $(-\kappa, \Omega)$. If neither player has a legal position, which represents that s has no successor state, then s is said to be a **terminal state**. For a terminal state $s_t = (\kappa_t, \Omega_t)$, we call s a **win state** (resp. **lose state**) for player κ if $count(p_{\Omega_t,i,j} = \kappa) < count(p_{\Omega_t,i,j} = -\kappa)$ (resp. $count(p_{\Omega_t,i,j} = \kappa) > count(p_{\Omega_t,i,j} = -\kappa)$), and a **draw state** if $count(p_{\Omega_t,i,j} = \kappa) = count(p_{\Omega_t,i,j} = -\kappa)$, where $count(x)$ is the number of elements that satisfy condition x. If a state s_i can be changed to s_j by iterative steps, it is called that s_i can **reach** s_j.

With the above formulation, we are now ready to define our problem which our algorithm focus on.

Problem 1 (Reserved Reversi): Given a state $s = (\kappa, \Omega)$, we aim to find a successor state $s_{next} \in next(s)$, which can reach a winner state for player κ. If no winner state is reachable, find a successor state which can reach a draw state.

One way to solve this problem is to enumerate all possible state changing processes. However, this is unrealistic - there are 3^{64} possible situations of board, and enumerating them is

Algorithm 1: *Alpha-Beta Search* (Ω, κ, κ_0, d, α, β)

1 **if** $d = 0$ **then return** *Evaluation*(Ω, κ_0),$\emptyset$;
2 $\alpha_0, \beta_0 \leftarrow \alpha, \beta$;
3 $s_t \leftarrow \emptyset$;
4 **foreach** $(\kappa_i, \Omega_i) \in next((\kappa, \Omega))$ **do**
5 $v \leftarrow$ *Alpha-Beta Search*$(\Omega_i, \kappa_i, \kappa_0, d-1, \alpha_0, \beta_0)$;
6 **if** $\kappa = \kappa_0$ **then**
7 **if** $\alpha_0 < v$ **then**
8 $\alpha_0 \leftarrow v$;
9 $s_t \leftarrow (\kappa_i, \Omega_i)$;
10 **else**
11 **if** $\beta_0 > v$ **then**
12 $\beta_0 \leftarrow v$;
13 $s_t \leftarrow (\kappa_i, \Omega_i)$;
14 **if** $\alpha_0 >= \beta_0$ **then return** v, s_t;
15 **if** $\kappa = \kappa_0$ **then return** α_0, s_t;
16 **else return** β_0, s_t;

an unachievable task. Hence the goal we can try is to find a successor state that can have a high expectation of reaching a winning state.

III. Methodology

A. Basic Framework

The algorithm I designed consists of two basic parts: *Alpha-Beta Pruning Search* [4] and *Static Evaluation Function*.

Alpha-Beta Pruning Search is an adversarial search algorithm that is mainly applied to two-player games that can be played by machines (*e.g.*, tic-tac-toe, chess, go). When the algorithm evaluates that the subsequent moves of a strategy are worse than those of the previous strategy, it stops computing the subsequent development of that strategy. The algorithm reaches the same correctness guarantee as the *Minimax Algorithm* [2], but prunes out branches that do not affect the final decision. The pseudo code of this algorithm is illustrated in Algorithm 1. For a current state $s_i = (\kappa_i, \Omega_i)$, suppose that we are player κ_i, and we run *Alpha-Beta Search*(Ω_i, κ_i, κ_i, d_{max}, $-\infty$, ∞), where d_{max} is the maximum number of recursive layers we want the algorithm to search for, then s_t in the output is the next state found by the algorithm. If *Alpha-Beta Search* can search until the terminal state appears, then it can return a definite solution. However, in practice the algorithm can only recurse 3-8 layers. Thus, when the final layer is not the terminal state, we need a heuristic evaluation function to determine whether the state is favorable to κ_i.

As a static position evaluation function (input is only the current board, output is only a real number) which is shown in Algorithm 8, I first divide the board into three phases (opening, midgame, endgame) and use different parameters in each phase(Lines 1-2). Then in each phase, I consider the following aspects to evaluate the position: difference of pieces, positional weights, stable discs, frontiers, and mobility weights(Line 3-7).

Algorithm 2: *Evaluation* (Ω, κ_0)

1 calculate sum of discs as sum;
2 $k \leftarrow (sum - 5)/20$;
3 calculate **difference of pieces** as dif_{κ_0};
4 calculate **positional weights** as pos_{κ_0};
5 calculate **stable discs** as $stable_{\kappa_0}$;
6 calculate **frontiers** as $front_{\kappa_0}$;
7 calculate **mobility weights** as arb_{κ_0};
8 **return** $arb_{\kappa_0} + pos_{\kappa_0} + c2_k \cdot stable_{\kappa_0} + c3_k \cdot dif_{\kappa_0} + c4_k \cdot front_{\kappa_0}$;

- **difference of pieces**: Intuitively, the fewer the number of pieces, the better for you. However, if we only consider the number of pieces, we will often be biased in your judgment of the position. Here, we simply calculate the dif value in this way:

$$dif_{\kappa_0} = count(p_{\Omega_t,i,j} = \kappa_0) - count(p_{\Omega_t,i,j} = -\kappa_0).$$

- **positional weights**: Carve out the relative importance of edges, corners, and centers. The weights are represented by a matrix A of size 8×8. At each position on the board, if there is a piece of our own color, add the weight of that position to the total weight; if it is the opponent's piece, subtract the total weight from the weight of that position. The final total weight is the pos value. Mathematically, pos

$$\sum_{i\in[8],j\in[8],p_{\Omega_t,i,j}=\kappa_0} A_{i,j} - \sum_{i\in[8],j\in[8],p_{\Omega_t,i,j}=-\kappa_0} A_{i,j}.$$

- **stable discs**: A piece that absolutely will not be flipped in color. A corner is an example of a stable disc. Considering the complexity of calculating exactly the stable discs and the fact that it is basically impossible to determine the stable discs before the endgame, only some of the stable discs are judged in the code: the corner and the discs of the same color near the corner. Suppose that the number of stable discs for player κ is ss_κ, then

$$stable_{\kappa_0} = ss_{\kappa 0} - ss_{-\kappa 0}.$$

- **frontiers**: Pieces with at least one empty position around them. In my design, the more empty positions there are, the higher the weight of the frontiers. The fewer frontiers you have, means that your opponent's discs will "wrap" your own discs, which is more advantageous to you in the endgame. Similar to the calculation of stable discs, $front_{\kappa_0}$ is the difference of κ_0's frontier number and $-\kappa_0$'s frontier number.
- **mobility weights**: Consider one player's all legal positions, each with a mobility value. The higher the mobility, the more you can "dictate" the course of the game. Similar to the calculation of positional weights, the mobility weights are represented by a matrix B of size 8×8.

Finally, in the evaluation function, these factors are summed up with different weights, which is the evaluation value of the situation. It can be seen that there are 64 parameters in each of matrices A and B. Considering the symmetry,

only 8 parameters are considered for each matrix. Adding $c2_k, c3_k, c4_k$, where $k = 0, 1, 2$, these parameters are the ones we need to adjust manually. The more accurate the parameters, the more accurate the evaluation of the game. However, these aspects are derived from classic *Reversi*, so we have no experience with how these parameters are set in *Reverse Reversi*. A simple idea would be to take the parameters that apply to *Reversi* and reverse them, however this does not accurately describe the game in *Reverse Reversi*. For example, in both games, mobilitys should take the major part of the game and should not be reversed. In order to adjust these parameters adaptively, we introduce a completely new technique: the Genetic Algorithm.

B. Genetic Algorithm

Genetic Algorithm (GA) [7] is a stochastic search and optimization method that simulates the replication, crossover and mutation that occur in natural selection and genetics. It starts from any initial population and generates a group of individuals more suitable for the environment through random selection, crossover and mutation operations, so that the population evolves to a better and better region in the search space, and thus keeps reproducing and evolving from generation to generation, and finally converges to a group of individuals most suitable for the environment, thus finding a high-quality solution to the problem.

My design basically follows the framework of *Simple Genetic Algorithm*, with a mutation rate of 0.05 and a population size of 56, and each generation pitted all individuals against each other, keeping the higher half of the population and hybridizing to bring the population size back to 56, and keeping the six strongest historical versions, Every individual after generation 6 will play against the historical version. The algorithm is illustrated in Algorithm 3. It is worth describing in detail that the genes (independent variables) of the individuals added in the initialization contain the parameters of the evaluation function described in section III-A. When individuals are playing against each other, they greedily choose the movement that maximizes the value of the evaluation function instead of performing an alpha-beta pruning search. This is due to save the running time of the genetic algorithm.

This algorithm does not set a termination condition, so when we want to take out the strongest version so far, we can find the current result from the log. Theoretically, as the generation increases, the obtained version will be more and more accurate in estimating the board. On the online judge server of the course, the algorithm using the parameters of the 500th generation of genetics algorithm has been able to obtain the **top 5**.

C. Further enhancements

In addition to the above basic framework, the algorithm uses many other optimization techniques in order to be able to compete with a branch of algorithms using *neural networks*, *alphaZero* [6] and other deep learning algorithms.

Algorithm 3: *GA* ()

1 randomly initialize 56 individuals as $waiter_0, \ldots, waiter_{55}$;
2 **for** $generation = 1, 2, \ldots, \infty$ **do**
3 All waiters (including the 6 strongest historical versions) play against each other, and count the times each waiter $waiter_i$ wins in black and white, denoted as $black_i$ and $white_i$;
4 Sort all waiters by $min(black_i, white_i)$ in decreasing order;
5 Push $waiter_0$ into historical versions;
6 Delete the last half of waiters;
7 **foreach** $waiter_i, i = 0, \ldots, 27$ **do**
8 Randomly choose one waiter to product with $waiter_i$, denoting the production as son;
9 $waiter_{i+28} \leftarrow son$;

1) Iterative Deepening Search: One of the difficulties of using *alpha-beta pruning search* is that it is not easy to determine its search depth. If the search depth is too large, the search may waste too much time in a certain search branch; if the search depth is too small, the search will stop quickly. In this project, the time limit for each step is 5 seconds, and if the search stops quickly, then the time limit is not fully utilized. A simple idea is to deepen the search layers step by step: within 5 seconds, if the search ends in 3 layers, search 4 layers, and so on. This simple idea is called *iterative deepening search* (IDS) [5].

Furthermore, IDS has a more profound application to *alpha-beta pruning search*: if the search order of each *alpha-beta pruning search* is optimal, then the number of branches per expansion is reduced from $O(b)$ to $O(\sqrt{b})$ [4]. And the search order can be determined by the IDS of smaller depth. Motivated by this, we save the current optimal route for each *alpha-beta pruning search*, and search this route in the next search in priority. This not only reduces the number of branches to be expanded, but also ensures that the result of each search is at least no worse than the previous one.

2) Final Search: When the game is nearing its end, a simple *Min-Max search* can be used to quickly find out if there is a sure win move. In practice, the Min-Max search is even faster than the static evaluation function when the number of pieces is not less than 56. Recall that the evaluation function has a time complexity of $O(n)$, but has a non-negligible constant, since it has to scan the board many times to obtain evaluation values for all aspects. Therefore, when the number of pieces of Ω is not less than 56, we replace $evaluation(\Omega, \kappa_0)$ in line 1 of algorithm 1 with $finalSearch(\Omega, \kappa_0)$.

3) Evaluation of edge weights: From a large number of actual games I have found that high level games pay a lot of attention to the edges of the board. So I consider adding the evaluation of the four edges of the board to the static evaluation function.

For the edge where all pieces are played, I simply calculate its weights: 2 for the two corners and 1 for the other positions.

For the other edges, consider using **dynamic programming** to calculate its weights recursively. The basic idea is that both players in each position have an equal probability of placing a piece, and when the position has the possibility of reversing the opponent's disc, the probability of placing a piece decreases. Finally, this weight is added to the static evaluation function as an aspect of chess evaluation.

4) `numba`: `numba` [1] is a `JIT` compiler that compiles *python* functions into machine code. Numba-compiled *python* code (array operations only) can run as fast as *C* or *FORTRAN*. Not only in practice, `numba` can greatly increase the search depth of the algorithm, but in genetic algorithms, `numba` also makes it possible to use $finalSearch(\cdot)$ to directly determine the winner in the endgame, making the evolved parameters more accurate. On the online judge server of the course, the algorithm using `numba` has been able to obtain the **2nd rank**, and maintain a 50 percent win rate against the algorithm of 1st rank.

D. Extension to the classic Reversi

Recall that the only difference between *Reverse Reversi* and classical *Reversi* is the determination of the winning and losing situation: the winner of the former is equivalent to the loser of the latter. So due to extend to classical *Reversi*, we need to adjust the parameters of the evaluation function. Fortunately, we do not need to adjust them manually: the genetic algorithm can do most of the work for us. Just reverse the win-lose condition and restart the genetic algorithm to generate the parameters of the evaluation function for classical *Reversi*. Since the rules of the two games are identical, no other part of the algorithm needs to be modified.

IV. Experiments

A. Setup

All experiments on the algorithm were run on my own computer. The environment configuration of my computer is described in Table II. I built a simple self-matching platform that can support two different algorithm implementations playing against each other as black and white players. In all versions, the population size, elimination rate, and mutation rate of the genetic algorithm are given in section III-B; in the following, we denote **Baseline** as the algorithm that implements the basic framework (*i.e.*, *Alpha-Beta Pruning Search* and *Static Evaluation Function* described in section III-A), and **Final** as the final algorithm that implements all optimization methods in section III-C.

Operating System	Windows 11
CPU	AMD Ryzen 7 5800H
GPU	NVIDIA GeForce RTX 3060 Laptop GPU
kernel number	8
Python Version	3.8.3
Numpy Version	1.18.5
numba Version	0.50.1

TABLE II: Environment

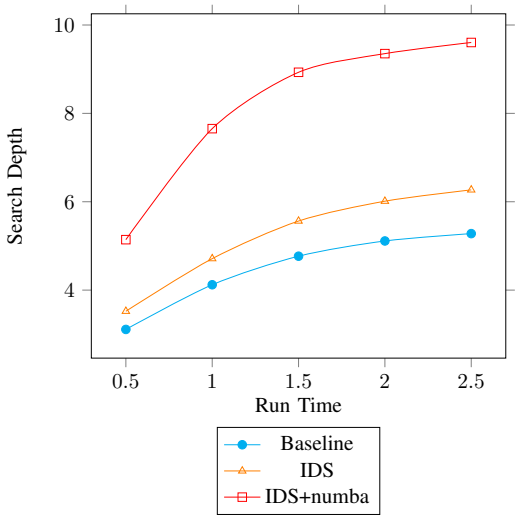

Fig. 1: Search Depth of Optimization Techniques

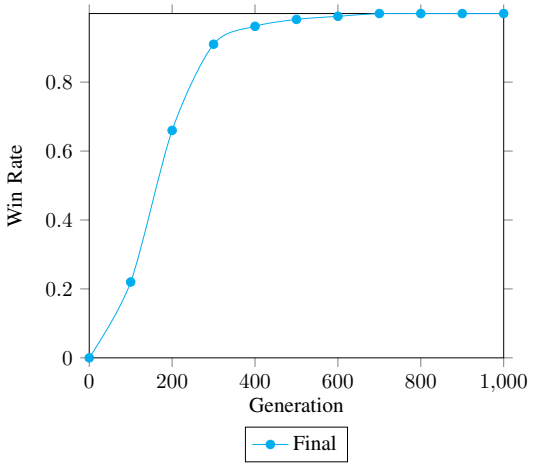

Fig. 2: Search Depth of Optimization Techniques

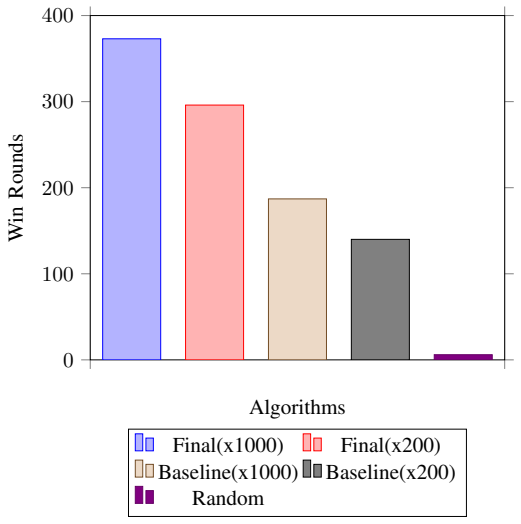

Fig. 3: Win Rounds of algorithms

B. Results

First we verified the speedup effect of *IDS* and `numba` on the algorithm. As described in Figure 1, *IDS* has increased the number of search layers in the baseline by about 30%. Furthermore, `numba` significantly accelerates the code, doubling the number of search layers on average, which means that the overall speedup is nearly 1000 times.

Next, using **Baseline** with 1000 genetic generations as a benchmark, we let **Final** with different genetic generations play against it and record the win rate, which is illustrated in Figure 2. As can be seen, the accuracy of the parameters does become higher as the number of genetic generations increases, and adding the evaluation of edges improves the accuracy even more.

Finally, we pitted the following five algorithms two against two 100 times: **Final** with 1000 genetic generations, **Final** with 2000 genetic generations, **Baseline** with 1000 genetic generations, **Baseline** with 200 genetic generations and randomly move. Each algorithm was played against each other 400 times in total. The results of the matchmaking are shown in Figure 3.

V. Conclusion

I implemented an algorithm for *Reverse Reversi*, which is based on *Alpha-Beta Pruning Search*, *Static Evaluation Function* function and *Genetic Algorithm*, and uses techniques such as *IDS*, `numba` to improve the running efficiency, and achieved excellent results in real-world matchmaking.

References

[1] Numba: A high performance python complier. https://numba.pydata.org/.
[2] Wikipedia: Minimax algorithm. https://en.wikipedia.org/wiki/Minimax/.
[3] Wikipedia: Reversi. https://en.wikipedia.org/wiki/Reversi.
[4] Knuth, D. E., and Moore, R. W. An analysis of alpha-beta pruning. *Artificial intelligence 6*, 4 (1975), 293–326.
[5] Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence 27*, 1 (1985), 97–109.
[6] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).
[7] Whitley, D. A genetic algorithm tutorial. *Statistics and computing 4*, 2 (1994), 65–85.