CS303 AI Project1: Reversed-reversi

Name: Shimin Luo
SID: 12012939

***Abstract*—AI for reversi has been a well researched topic for a long time. Reversed-reversi is an variation of Reversi, the goal of which is to minimize the player's chess pieces. This project applies alpha-beta algorithm for board searching, and dynamically changes the board weight matrix, searching depth and evaluation function to optimize the AI. The intelligent agent finally achieves relatively excellent performance in the Reversed-reversi game competition.**

I. Introduction

A. Problem Description

Reversed-reversi is a kind of zero-sum game played by two competitors with black and white pieces on an 8×8 chessboard. Each player takes turns to place his colored piece on the board, to flip the color of opponent's pieces between the piece he just played and any of another his piece. The constraint is: (1)A move of the player must flip at least one opponent's piece. (2)If the current player has not position to place his piece, skip his turn to his opponent. If both of the players are unable to place their pieces, the game terminates. The player owns less pieces wins.

Reversed-reversi's environment is fully observable, multi-agents, deterministic, static and discrete. It can be abstracted as adversarial search problem. Therefore, the strategies of adversarial search like minimax algorithm and alpha-beta pruning algorithm, can be applied to the game searching.

B. Problem Origin

Reversed-reversi is an variation of Reversi, which has the same rules as Reversi except for the way to determine the winner. As for the Reversi, it originated in 1883 century in England and immediately gained wide popularity in local area. Later, Reversi, which is full of interesting and recreational with simple but changeable rules, was introduced to China and soon attracted many people's attention.

Thesedays, artificial intelligence(AI) is one of the most active research fields. Particularly, chess is an important branch of AI research and with the development of game algorithms, more and more AI "chess players" have been created, such as AlphaGo. Reversi, as a classic board game, has also been researched over years. However, Reversed-reversi AI did not catch people's eyes until 2019, when a Japanese named JiTian released a "weakest Reversi AI" and claimed that nobody can lose to it. Indeed, its winning rate is only 0.5 percent after practicing, which is a landmark breakthrough for the Reversed-reversi AI.

C. Application

Reversed-reversi AI has some application in our real world. First of all, the evaluation model and corresponding weights design can help our human to realize the main factors we should consider, and the extent of their influence respectively during the game, which is particularly beneficial for the beginner. In addition, the evaluation function also shows the winning rate of the board quantitatively, which is convenient for the player to judge his moves and make progress. Last but not least, the algorithms and methods we use in this project can also be used in other searching problems, which may provide some novel ideas or solutions for other problems.

D. Project Purpose

The purpose of this project is to accomplish a Reversed-reversi AI. It needs to find out all valid actions, and the best move should be the last element of the candidate_list.

II. Preliminary

The problem can be formulated as an adversarial search problem, which is specified by a tuple (S, s_0, A, T, G, U).

1) States: $S = \{s_0, s_1, s_2, \dots s_n\}$ denotes a set of all possible states of the game, each state can be represented by (B, P), where B denotes current chessboard which is 2-dimensional 8×8 array with element $\{-1, 0, 1\}$, and P denotes current player.

2) Initial Status: s_0 denotes initial state which is represented by (B_0, P_0). Generally, B_0 is a empty chessboard and P_0 is the player with black piece in this game.

3) Actions: $A = \{a_0, a_1, a_2, \dots a_n\}$ denotes the set of all valid actions for a particular state s_i. It can be easily calculated by function $findActions(chessboard, player)$

4) Transition: T denotes the function mapping current state and particular move to the next state, which can be describe as $T(s_{prev}, a_i) \rightarrow s_{next}$

5) Terminal Test: G is a terminal test function returning a bool value, $True$ when game is over, $False$ otherwise. In the Reversed-reversi, G returns $True$ when both players' actions $A = \phi$.

6) Utility Function: U denotes utility function, or called evaluation function, which can evaluate a state's winning rate, and be represented by (s_i, p) where s_i denotes one of the chessboard state and p denotes AI's player itself. In this project:

$$U(s_i) = w_{1_{s_i}} \times \mu_{1_{s_i}} + w_{2_{s_i}} \times \mu_{2_{s_i}} + w_{3_{s_i}} \times \mu_{3_{s_i}} + w_{4_{s_i}} \times \mu_{4_{s_i}} \quad (1)$$

where $\mu_1, \mu_2, \mu_3, \mu_4$ are chessboard weight, opponent's mobility, piece difference and the force score (some extra points, positive value when opponent has a high probability of occupying the corner, negative when my AI has a high probability of occupying the corner) of state s_i. w_1, w_2, w_3, w_4 are the weights of the corresponding factors, which are dynamic during the game.

III. Methodology

A. General workflow

This section will introduce the core methods and algorithms of this project. The proposed method divides into:

1) Step1: Find valid actions, initialize $maxDepth$.

2) Step2: Do adversarial searching, including minimax algorithm with alpha-beta pruning and trigger evaluation function.

3) Step3: According to the consumed time, dynamically increase the $maxDepth$ and redo step2 until the time runs out.

B. Detailed algorithm

1) Find Valid Actions: AI must find valid actions A with given chessboard B and current player P. The following is the pseudocode for the algorithm:

Algorithm 1 : findActions

Input: chessboard, player
Output: Valid actions A

```
A ← [ ]
indexes ← chessboard without piece
for i in indexes do
    for dir in eight-directions do
        if consecutive pieces in dir != player and
        the end piece of dir line == player then
            append i into A
        else
            continue
        end if
    end for
end for
```

2) Adversarial Searching: Minimax algorithm with alpha-beta pruning is used as the searching method in this project to solve adversarial problem. In the minimax search tree, the states with depth more than $maxDepth$ will be evaluated by the utility function, aiming to estimate its winning rate, which will be further explained:

Weight Matrix:

This part of evaluation score is donated by μ_1, which represents the chessboard weight score.

The weight matrix is a two-dimension 8×8 matrix, each element represents the value of the grid with corresponding place in the chessboard. If the grid is placed by my color piece, then μ_1 adds that value. If opponent's piece is placed there, than μ_1 loses that value. Else, μ_1 remains unchanged. Finally, μ_1 is the weight score of the whole chessboard.

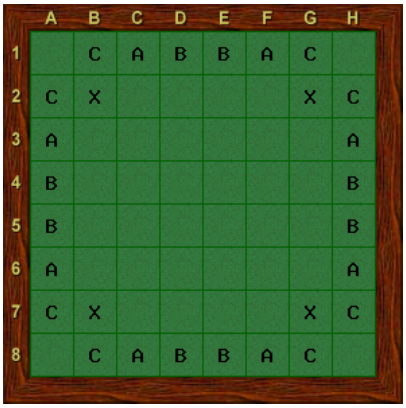

Fig. 1: The Reversi chessboard with some notations on critical positions

Generally, AI should avoid occupying four corners. So I assign an very small value to the corners. On the other hand, the places which is notated by X and C in $Figure1$ should be occupied as fast as possible to induce opponent to place pieces on corners. So I assign big values to them. Similarly, others positions are assigned according to the practice experience [1].

In addition, weight matrix is not static but dynamic. There are two situations will change the weight matrix. One is that, when my piece occupies a corner, the weight of pieces in edge row, edge column and diagonal, which contain the corner will decrease dramatically. This can help us not to develop our stable pieces. For example, if I occupy $(1, A)$ in the $figure1$, I will reduce the weight of all the pieces on the first row, first column and diagonal from $(1, A)$ to $(8, H)$. Another situation to change weight matrix is, if I occupy grids notated X, the weight of grid A will increase, which aims to make the my piece on X to be stable.

Opponent's Mobility:

This part of evaluation score is donated by μ_2, which represents how many valid actions my opponent has in current state.

Piece Different:

This part of evaluation score is donated by μ_3, which represents the difference of the opponent's pieces number and mine.

Force Score:

This part of evaluation score is donated by μ_4, which depends on both mobility of opponent and the location of his actions in current state. The less mobility his has, meanwhile, the more his valid actions are corners, this part of score higher.

Utility Function:

The utility function represents the winning rate of a state, taking four factors above into account, and combines them in a linear function with different weight parameters respectively, showing as $formula(1)$. Especially, the weight

parameters are dynamic during the game. When the game just starts, the chessboard weight and opponent's mobility play important roles. At the middle of the game, our AI is supposed to force opponent to occupy corner. At the end, the winning rate should be dominated by piece difference. After battling other AIs for many times, I finally assigned w_1, w_2, w_3, w_4 as following:

TABLE I: Weight Parameters During Game

	w_1	w_2	w_3	w_4
0-20 steps	35	30	20	25
20-40 steps	35	35	25	35
40-50 steps	45	20	35	20
50-60 steps	15	5	80	5

In addition, if the chessboard is a terminal state, the utility function return value $(\#opponentpieces - \#mypieces) \times 1000000$, which is big or small enough to be distinguished among normal evaluated value in the minimax function.

Searching Algorithm:

This project uses minimax algorithm with alpha-beta pruning to accelerate searching. The algorithm designed is shown in both flow-chart and pseudocode.

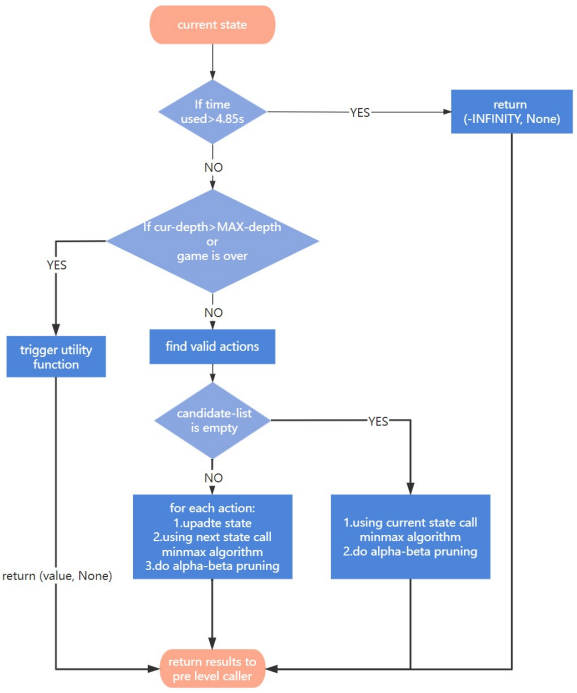

Fig. 2: Flow-chart of minimax algorithm with alpha-beta pruning

Some terminologies should be explained:
(1) $maxDepth$: A global variable which denotes the maximal searching depth of minimax algorithm.
(2) $utility(curChessboard, player)$: The function to get the result value of $formula(1)$.
(3) $result(chessboard, a_i, player)$: It is the transition function which maps current state and action to the next state. Board changes are made directly on the $chessboard$.

Algorithm 2 : Alpha-beta Pruning (including minimax)

Input: chessboard, player, startTime
Output: value, bestAction a_i

function MAX_VALUE($curChessboard, curDepth, \alpha, \beta$)
  if $time\ is\ over$ **then**
    stop searching immediately, return $-INF$ value
  end if
  if $curDepth > maxDepth$ **then**
    return $utility(curChessboard, player)$
  end if
  $v, move = -infinity, None$
  $A_i = findActions(curChessboard, player)$
  $temp \leftarrow curChessboard$
  if $len(A_i) == 0$ **then**
    $v2, \_ = min\_value(temp, curDepth + 1, \alpha, \beta)$
    if $v2 > v$ **then**
      $v, move = v2, None$
    end if
    if $v \geq \beta$ **then**
      return $v, move$
    end if
  end if
  for $action\ a_i$ **in** A_i **do**
    $result(temp, a_i, player)$
    $v2, \_ = min\_value(temp, curDepth + 1, \alpha, \beta)$
    if $v2 > v$ **then**
      $v, move = v2, a_i$
    end if
    if $v \geq \beta$ **then**
      $break$
    end if
    $\alpha = max(\alpha, v)$
  end for
  return $v, move$
end function

function MIN_VALUE($curChessboard, curDepth, \alpha, \beta$)
  .....
end function

returns $max\_value(chessboard, 0, -infinity, infinity)$

Since function min_value has the similar process of max_value with the opposite logic, I omit its pseudocode.

3) Dynamic Depth: At first, I designed different static $maxDepth$ according to the number of empty grids. Later, I assigned different $maxDepth$ according to the length of candidate_list. However, both methods could not perform very well. Sometimes the algorithm wastes a lot of time, but sometimes it even cannot finish searching the initial assigned depth. Finally, I used a $While$ to accomplish increasing the searching $maxDepth$ dynamically. When the searching lasts

for $4.85s$, it will immediately terminate and return the best action it has found before. Otherwise, after fishing searching the $maxDepth = i$, it will continue searching depth $i + 1$. The $AI.go()$ algorithm is shown below:

Algorithm 3 : AI.Go

Input: chessboard
Output: self.candidate_list
1: $startTime \leftarrow$ start time counting
2: $A \leftarrow findActions(Chessboard, self.color)$
3: Initialize original $maxDepth$
4: **if** empty grids on chessboard ≥ 11 **then**
5: **while** $True$ **do**
6: $value, move \leftarrow AlphabetaPruning(chessboard, self.color, startTime)$
7: **if** $value == -INF$ **then**
8: **if** $move$ is not $None$ **then**
9: $A.append(move)$
10: **end if**
11: $maxDepth+ = 1$
12: **else**
13: $break$
14: **end if**
15: **end while**
16: **else**
17: $value, move \leftarrow AlphbetaPruning(chessboard, self.color, startTime)$
18: **if** $move$ is not $None$ **then**
19: $A.append(move)$
20: **end if**
21: **end if**
22: $self.candidate\_list \leftarrow A$
23: return $self.candidate\_list$

Some explanations:
(1)When the searching time more than $4.85s$, the function $AlphbetaPruning$ will return $value = -INF$. Therefore, the $while$ breaks and algorithm returns the list.
(2) In the $line3$, I initialize the $maxDepth$ as following:

TABLE II: $maxDepth$ Initialization

Empty girds	**53-60**	**23-47**	**14-17**	**11-13**	**10**	**0-10**	**Else**
$maxDepth$	4	2	4	5	7	15	3

C. Analysis

The time complexity of MiniMax Algorithm is $O(b^d)$, where b is effective branching factor and d is the depth of the searching tree. Function $utility$ needs to traverse all chessboard locations to calculate the four factors discussed before. Denoting board size as n, then the time complexity of $utility$ is $O(n^2)$.

Therefore, the complexity of alpha-beta pruning with depth d is $O(b^d n^2)$. Because Alpha-beta pruning is inferior when the depth is small, I designed the dynamic $maxDepth$ to make the search deeper within the allotted time, which is a good optimization according to the experiments in next section. And in this project, n is 8. Obviously, the time complexity is mainly decided by the effective branching factor(actions A of the state) and depth.

IV. Experiments

A. Setup

1) Data set: In order to test my AI's performance, I simulated an adversarial environment by writing a script $vs\ platform$, where I can $import$ two different version AIs I wrote, and let them battle automatically. For each step, the console will print **initial** $maxDepth$, the **searching information about depth**, **time-consumed**, **player's move**, and **chessboard state**(triangle represents black piece and square represents white. This $vs\ platform$ is very convenient for me to find out whether the parameters I changed is useful or not. $Figure3$ shows a step in a game.

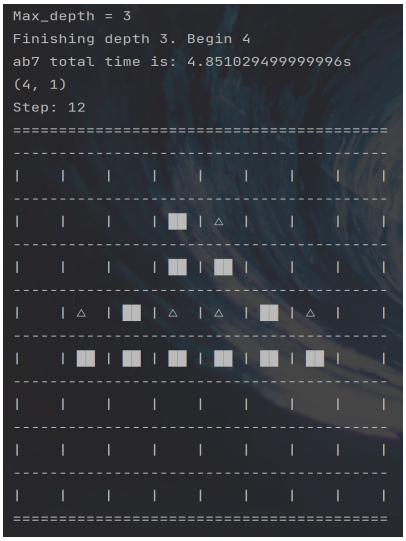

Fig. 3: Screenshot of one step printed by vs-platform

2) Environment:
software:

· python 3.9.13
· numpy 1.23.4
· online usability and round robin test: SUSTech Reversed Reversi AI platform

hardware:

· Processor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz
· RAM: 16GB(15.8GB is available)

B. Results & Analysis

I wrote the approximately 10 versions of AI during this project, which can be classified into about four version phases. All of the AIs use minimax algorithm and alpha-beta pruning:

TABLE III: AIs of different versions

	$phase_1$	$phase_2$	$phase_3$	$phase_4$
weight matrix	Y(s)	Y(s)	Y(s)	Y(d)
piece difference	Y	Y	Y	Y
opponent's mobility	Y	Y	Y	Y
force-corner score	Y	Y	Y	Y
dynamic weight parameters		Y	Y	Y
dynamic $maxDepth$			Y	Y

Where Y means the utility function takes into account that factor, and s, d means "static" and "dynamic" respectively.

In addition to my offline $vs\ platform$, I also wrote a script to let my AIs battle in $KUMITE\ MODE$ (any two AIs will play against each other), and calculate the wining rate of each AI. $TableIV$ shows the performance of four versions of AI on behalf of the highest level of four version phases above respectively.

TABLE IV: AIs of different versions

phases	local win-rate	average online score	worst time	characteristic
1	10%	40	$\leq$4s	weight matrix
2	18%	110	$\geq$5s	dynamic weight parameters
3	33%	165	$\approx 4.85s$	dynamic $maxDepth$
4	39%	180	$\approx 4.85s$	dynamic weight matrix

From the $TableIV$, we can see that the first phase version is worst and the final version phase is the best. It seems that taking into account more factors leads to better performance. However, it is not always true during the process I modified my code. For instance, when I first applied the method of changing $maxDepth$ dynamically, my AI lost to the previous version. I think the reason is that, after increasing the searching depth, some states used to be trigger the middle-game evaluation function is now trigger the end-game evaluation function. After the practice and changing the weight parameters appropriately, the AI's performance is as good as expected. Therefore, the conclusion is that, only when taking into account more factors with appropriate parameter weights, can indeed improve AI's level.

Meanwhile, the hyperparameters also play an important role in the performance. For example, the searching depth, before dynamically increase it, I found that the length of candidate_list in some cases can be up to 14, even when $depth = 3$, according our theoretical time complexity in $Methodology$ part. the algorithm need to search 14^3 states in allocated time, which is nearly impossible. For this reason, dynamic depth strategy is of great necessity.

In general, the AI's performance meets my expectation. During this project, I improved my AI step by step, meanwhile, we can also see the effect of different components and hyperparameters to AI's performance:

(1) Comparing phase version 1 and 2, we can know the importance of dynamic weight parameters, which I mentioned in the description of $Utility\ function$ in $Methodology$ part.

(2) From phase version 2 and 3, we can see that dynamically increase $maxDepth$ can more accurately calculate a state's win-rate. The phases 3 and 4 AIs' worst running time shown in the table also meets my expectation in $Methodology$ part, where I indicate the specified maximum search time is 4.85s.

(3) We can know that dynamically change weight matrix is also a good strategy from phase version 3 and 4, which is obvious and consistent with our theoretical analysis in $Methodology$ part.

V. Conclusion

A. Summery & Disadvantages

In this project, I implement a reversed-reversi AI using the alpha-beta pruning with corresponding evaluation function. I improve the evaluation function step by step: taking into account more factors, change some parameters dynamically and so on. After practicing different version AIs on my own $vs\ platform$ and online judge, I adjusted and selected the one with the best performance. However, it still has some limitations:

(1) I did not take the $action\ score$ into account, which is calculated by the information of exactly one action. For example, how many pieces I flip in this action.

(2) Logically, there should be a benefit for black piece to start the game. However, this advantage cannot be amplified in my algorithm and evaluation function.

(3) The parameters were not adjusted to the best comparing to some AIs using genetic algorithm [2]. In addition, the $stability$ is not considered.

B. Gains & Regrets

During the process, I am more familiar with package $numpy$, knowing how to use it to accelerate matrix operations. This project has trained my ability to think independently and solve problems in an all-round way. However, because of time limitation and academic pressure, I didn't try to write genetic algorithm, which I am desired to try. I am a little bit regret about it.

C. Future Work:

(1) Read the literature related to Reversi, learn the winning strategy, and introduce a variety of evaluation factors such as $action\ score$ and $stability$.

(2) Try to write genetic algorithm, which makes two AI play against each other automatically and adjusts the parameters automatically for better performance.

References

[1] Yasuhiro Osaki, Kazutomo Shibahara, Yasuhiro Tajima, and Yoshiyuki Kotani. An othello evaluation function based on temporal difference learning using probability of winning. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 205–211, 2008.

[2] Jean-Marc Alliot and Nicolas Durand. A genetic algorithm to improve an othello program. volume 1063, pages 307–319, 09 1995.