CS311 Reversed-Reversi Report

SID： 12012919

Name: 廖铭骞

Introduction

Reversed-Reversi is a simple board game played by two players representing Black and White. It is originated in England in the 19th century. In the game, both sides take turns placing disks in the chessboard, if any disks of the opponent's colour are in a straight line and bounded by the disk just placed and another disk of the current player's colour are turned over to the current player's colour. The size of the board is usually 8*8. When both sides have no pieces to play, the game is over. After that, the player with the least colour on the chessboard wins the game.

Purpose

The purpose of this project is to design and implement an AI algorithm that makes the winning rate against other players as high as possible. Also, the AI algorithm should be efficient, in order to find an optimal result within 5 seconds. The report focuses on the implementation of the AI algorithm that plays the reversed-reversi game. And our goal is to implement an intelligent agent to play the Reversed-Reversi game by using search methods.

Application

In game theory and economic theory, a zero-sum game is a mathematical representation of a situation in which an advantage that is won by one of two sides is lost by the other. If the total gains of the participants are added up, and the total losses are subtracted, they will sum to zero. Zero-sum game is very common in our life, such as poker, chess or Reversi in games, futures contracts and options in the market, etc., which are widely used.

Preliminaries

Formulation

Reversed-Reversi is a two-player deterministic zero-sum board game. In computer view, the chessboard can be characterised as a 8×8 two-dimensional array, and the colour can be represented by 1, 0 and -1, respectively representing the White, No colour and Black. Each step to place disk is to assign a colour to the array.

If the colour to be played is the same colour that has been on the chessboard before, and the chess pieces of different colours are surrounded by straight lines or diagonal lines, then the surrounded chess pieces will be converted into the colour to be played, that is, the value of the corresponding position on the chessboard will be changed to the other value.

Finally, when the game is over, the winner is the player that has the less discs of his colour than hisopponent. We can detect it by summing up all the values in the chessboard and see the result. If the result is positive, which means that the Black Player has less discs, then the winner is the Black Player, else if the result is negative, then the winner is White Player, otherwise, both sides draw.

Algorithm

In this project, I use the Minimax tree search with Alpha-beta pruning as the algorithm. And the most important part of the algorithm is the evaluation function. In order to better my evaluation function, I have tried many times on selecting and adjusting different values and weights with Genetic Algorithm.

Notation

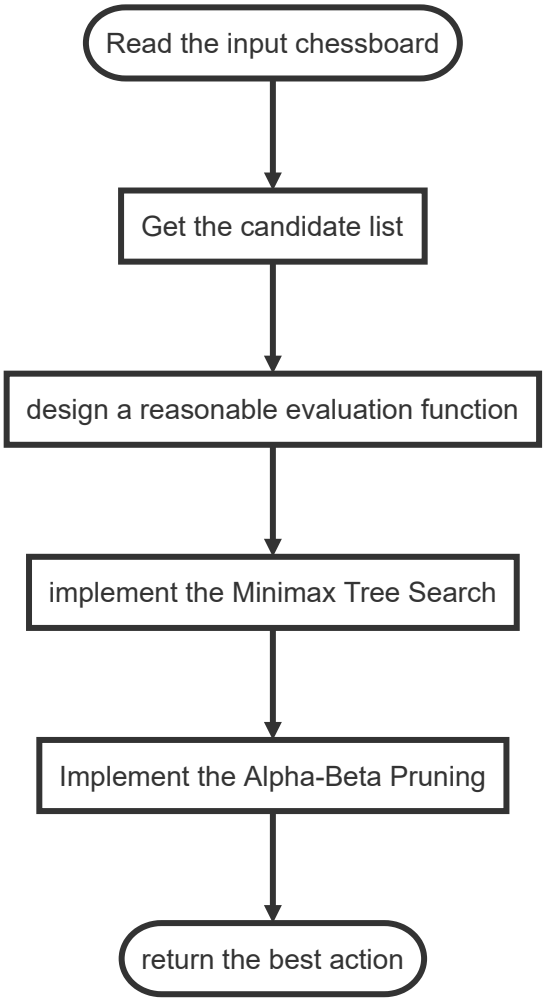
Symbols	**Description**
$time\_out$	the time limit
act_colour	the color that should take turn currently
$target\_colour$	the color that represents the side of player
$stage$	the current stage of the game
$chessboard$	the current chessboard
$V(b, c)$	the total value of the board according to chessboard b and colour c
$S(b, c)$	the number of discs which cannot be flipped according to chessboard b and colour c
$M(b, c)$	the number of possible moves available according to chessboard b and colour c

Symbols	Description
$F(b, c)$	the number of discs that have empty position around it according to chessboard b and colour c
$lw(b, c)$	the location weight of the chessboard b of colour c
cur_time	the current time
α	the upper bound of alpha-beta pruning
β	the lower bound of alpha-beta pruning
$depth$	the current searching depth
$w(i)$	the weight of evaluation function i
$value(i)$	the score of evaluation function i
$TOTAL(b)$	total score of chessboard b

Methodology

Work-flow

The project divides into four steps. The first step is to return an available list, which can be implemented by searching each directions for the acting player and find the available moves. The second step is to design an evaluation function, which includes the evaluation of stable discs, future mobility, frontier discs, etc. .The third step is to implement the Minimax Tree Search algorithm and the final step is to improve the Minimax Tree Search algorithm by using Alpha-Beta pruning. The flow chart is shown as belows.



Data structure

The primary data structure used in the project are listed in the following table.

Structure	Description
chessboard	a two-dimensional array representing the chessboard.
weighted-graph	a two-dimensional array representing the location weights.
candidates	a list of the available positions found
directions	a list representing 8 directions

Detail of Algorithm

Minimax Algorithm

The Minimax algorithm is a search-based algorithm that finds the minimum of the maximum likelihood of failure. Minimax is mainly used in zero-sum games, where one side wins and the other loses. The Minimax algorithm assumes that every move you make will move in the direction that is most beneficial to you, and that every move your opponent makes will take the move that is most unfavourable to you. Based on this idea, the Minimax algorithm can be represented by the following pseudo code.

```
function Minimax(self, chessboard, colour)

Input: chessboard: The current chessboard, which is assigned from {-1, 0, 1} in
each position.
       colour: The current player, 1 for self player, -1 for opponent player
Output: move, value: A best postion to place disc with evaluated value.

1: moves, next_chessboard <- getMoves(chessbaord, colour)
2: if len(moves) = 0, then
3:      return None, calValue(chessboard, colour)
4: end if
5: nextMove = null
6: value := 0
7: if colour = self.colour then
8:      tmp := -infinity
9:      for idx in range(moves) do
10:             curMove, curValue := Minimax(chessboard[idx], -colour)}
11:         if curValue > tmp then
12:             tmp := curValue
13:             nextMove := moves[idx]
14:         end if
15:     end for
16:     value := tmp
17: end if
18: else then
19:     tmp := infinity
20:     for idx in range(moves) do
21:         curMove, curValue := Minimax(chessboard[idx], colour)}
22:         if curValue < tmp then
23:             tmp := curValue
24:             nextMove := moves[idx]
25:         end if
27:     end for
28:     value := tmp
29: return nextMove, value
```

Alpha-Beta Pruning

The Alpha-Beta pruning algorithm is a search algorithm designed to reduce the number of nodes in its search tree that are evaluated by the minimax algorithm. This is a search algorithm commonly used in man-machine game confrontation. Its basic idea is to decide whether to continue the current search according to the current optimal result obtained by the previous layer. To be specific, the α is the maximum lower bound of possible value and the β is the minimum upper bound of possible value. During the searching process, if the possible value v makes $\alpha \geq \beta$, then the searching process along the path can stop, reducing the number of nodes that need to be searched.

Based on the idea, the Alpha-Beta pruning can be described as follows.

```
function Alpha-Beta(self, chessboard, depth, act_color, target_color)

Input:  chessboard: The current chessboard, which is assigned from {-1, 0, 1} in
each position.
        depth: The expected maximum depth the alogorithm will search.
        act_color: The current colour player who should place a disc in the
chessboard
        target_color: The self player's colour used to calculate the value of the
chessboard
Output: move, value: A best postion to place disc with evaluated value.

1: return maxvalue(chessboard, -infinity, infinity, self.colour, depth,
self.colour)
```

```
function maxvalue(chessboard, Alpha, beta, act_color, depth, target_color)

Input:  chessboard: The current chessboard, which is assigned from {-1, 0, 1} in
each position.
        Alpha: the maximum of the lower bound
        beta: the minimum of the upper bound
        depth: The expected maximum depth the alogorithm will search.
        act_color: The current colour player who should place a disc in the
chessboard
        target_color: The self player's colour used to calculate the value of the
chessboard
Output: move, value: A best postion for the self player to place disc with
evaluated value.

1: candidate_list, chessboard_list <- getMoves(chessboard, act_color)
2: if candidate_list.length = 0 or depth = 0 then
3:      return calValue(chessboard, target_color), None
4: end if
5: value := -infinity
6: next_move := None
7: for i in range(candidate_list.length) do
8: move, temp := minvalue(chessabord_list[i], Alpha, beta, -act_color,
target_color, depth-1)
9:     if temp > value then
```

```
10:       value := temp
11:       next_move := move[i]
12:       Alpha := max{Alpha, value}
13:       if Alpha >= beta then
14:          return next_move, value
15:       end if
16:    end if
17: end for
18: return next_move, value
```

```
function minvalue(chessboard, Alpha, beta, act_color, depth, target_color)

Input:  chessboard: The current chessboard, which is assigned from {-1, 0, 1} in
each position.
        Alpha: the maximum of the lower bound
        beta: the minimum of the upper bound
        depth: The expected maximum depth the alogorithm will search.
        act_color: The current colour player who should place a disc in the
chessboard
        target_color: The self player's colour used to calculate the value of the
chessboard
Output: move, value: A best postion for the opponent to place disc with evaluated
value.

1: candidate_list, chessboard_list <- getMoves(chessboard, act_color)
2: if candidate_list.length = 0 or depth = 0 then
3:      return calValue(chessboard, target_color), None
4: end if
5: value := infinity
6: next_move := None
7: for i in range(candidate_list.length) do
8: move, temp := maxvalue(chessabord_list[i], Alpha, beta, -act_color,
target_color, depth-1)
9:     if temp < value then
10:       value := temp
11:       next_move := move[i]
12:       beta := min{beta, value}
13:       if Alpha >= beta then
14:          return next_move, value
15:       end if
16:    end if
17: end for
18: return next_move, value
```

Evaluation Function

The evaluation function will evaluate the input board position, taking into account the various factors shown below. The value obtained by the evaluation function is used in the Alpha-beta pruning algorithm as a reflection of the state of both sides of the current situation. The total score of the current situation will be obtained according to the weighted sum of each factor, and the calculation formula is as follows.

$$Total = \sum_{i=1}^{4} w_i \times value(i)$$

Position Evaluation

For Reversed Reversi, the weights of different positions on the chessboard are different in the game. For example, in general, the weight of the four corners should be the lowest, because once a corner is occupied, it will no longer be flipped, and you'll be at a disadvantage. The positions near the corners have higher weights and should be preempted.

In the position evaluation function, I use the following weighted table to represent the importance of each position.

Index	A	B	C	D	E	F	G	H
1	500	-25	10	5	5	10	-25	500
2	-25	-45	1	1	1	1	-45	-25
3	10	1	3	2	2	3	1	10
4	5	1	2	1	1	2	1	5
5	5	1	2	1	1	2	1	5
6	10	1	3	2	2	3	1	10
7	-25	-45	1	1	1	1	-45	-25
8	500	-25	10	5	5	10	-25	500

When calculating the total position value of the chessboard, I will use the following formula.

$$value(1) = \sum_{position} chessboard[position] * lw(chessboard, position)$$

Mobility Evaluation

Mobility is also one of the important considerations in a chess game. On the one hand, when a player's mobility is better, he can predict more situations and choose a better position. On the other hand, when a player has no place to play, it will be the opponent's turn to play, which is very beneficial for the opponent to choose a better layout. Therefore, when assessing the situation, action also needs to be taken seriously.

In the mobility evaluation, I first get the candidate lists of both players, then I let `value(2)` to be the score of mobility.

$$value(2) = M(my\_color) - M(-my\_color)$$

Stability Evaluation

Stability is also a very important consideration. When a disc on the field can no longer change colour as the game progresses, it becomes a stability disc. When a corner becomes stable, the side pieces connected to it will also become stable. The number of stable pieces will directly determine the outcome of the final game, so the weight of stability is relatively large.

Let `value(3)` becomes the stability evaluation.

$$value(3) = S(my\_color) - S(-my\_color)$$

Frontier Evaluation

The frontier refers to a disc in which at least one of the surrounding nine lattices is empty. In the game, such pieces will increase the opportunity for the opponent to convert my disc to his colour. In the later stage, such pieces will make the opponent convert more of my own pieces to his colour, which is also of great significance to winning the game.

Let `value(4)` becomes the frontier evaluation.

$$value(4) = F(my\_color) - F(-my\_color)$$

Genetic Algorithm

The genetic algorithm will carry out a round robin battle against all individuals in each generation. Then, it will evaluate each individual's win rate, and save the individuals with the highest win rate in each generation. After recording a certain generation, I can round robin the optimal individuals of each generation again, so as to select the optimal individuals of these generations and achieve the goal of improving the performance of the algorithm.

Analysis

In the worst case, the time complexity of Alpha-beta pruning is equivalent to that of the Minimax algorithm, that is, no branch of the search will be pruned. At this time, assuming that the number of search layers is m and the branch coefficient is b, Then the time complexity of the Alpha-beta pruning algorithm is $O(b^m)$.In the ideal case, the best move of the algorithm are on the left of the game tree, the complexity is $O(b^{m/2})$. We can achieve it by remembering the best moves from the shallowest nodes, also we can keep the states. The deciding factor of the time complexity of Alpha-Beta pruning is from the move order during the search process. If we search the best result at first, then the latter branches will be largely pruned, which will

accelerate the process, otherwise, it will be as slow as Minimax algorithm. After a long time of manual adjustment and using Genetic Algorithm(GA) to adjust parameters, my parameters are becoming reasonable, along with the continuous optimization of the algorithm. In this project, within the limitation of 5 seconds, I can search for most 6 levels, which also guarantees my chance to win largely.

Experiment

Setup

Environment

As the software environment, the project is written in Python 3.9 with PyCharm IDE. The libraries I used include Time, Math and Copy. And the experiments are all done in Windows 11 Family Edition with Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz of 8 cores.

Dataset

The chessboard datasets used in this project are all manually generated by myself according to the target. In this way, I can more intuitively observe whether the algorithm for generating the set of feasible steps and selecting the optimal step is reasonable and efficient. In these chessboard datasets, there are many boundary cases, and it is suitable to detect whether the program considers the boundary conditions thoughtfully. The dataset have been instrumental in my debugging of Alpha-beta recursive programs.

What's more, I also implement a PK program to allow two AI class to battle, and save the battle log to a file. So, if an exception is caught, I can have a look at the log file, then correct my code, eventually construct some corner cases to test my code so as to increase the robustness of my program.

One of the dataset is as follows.

$$chessboard = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 1 & 1 & 1 & 1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 1 & -1 & 0 & -1 & 0 & 0 \\ -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To improve the performance of my program, I not only evaluated the performance by the AutoPlay and Playto button on the platform, but also implemented the GA algorithm to automatically tunning the parameters. Then I got a lot of dataset of the parameter of the location weighted graph. By choosing one of the parameter which can yield the highest win rate, I can optimize the parameters and improve the algorithm. I had trained for a thousand generations, and the training result is shown as belows. Due to space limitations, only a part is shown here.

{"args": [-0.36717457835217626, 7.094607770393502, 15.13807210567403,
7.772724468889448, 10.147659520545266, 36.93667941948853, 30.966910054515434,
20.864874957099797, -50.0, 12.953952699449248, 117.77821331183834,
-0.7279916979842015, 2.532234860819841, -7.154169263486011, 47.38103172570222,
-1.6579138714652588, 3.1741652607172277], "score": 28, "life": 4}, {"args":
[4.7612545006586995, 13.998628373602251, 18.574413089607923, 1.9364177188439555,
38.858354732415634, 50.0, 24.943664643916918, 45.0663090501306, -50.0,
47.022444075768824, 118.41376894348251, 4.560911673644438, 3.830840078761297,
37.240295165832634, 50.0, -1.8299154484022666, 3.6341568832472353], "score": 23,
"life": 3}

Results

performance measures

The performance measures taken in my project is the time and the optimality. For the time measurement, I calculated the average time used by different algorithms per step, and the maximum time limit is 5 seconds. For the win rate measurement, I used a round robin among different algorithm and logged their win rates.

Experimental results

Comparison of the time taken with and without Alpha-beta pruning

In this experiment, I used different searching levels to test the time.

Average Time \s	searching level	searching level	searching level	searching level	searching level	searching level
Type of algorithm	5	6	7	8	9	10
Minimax	0.522910118	1.566905022	5.396279812	16.77578855	56.47641087	180.821208
Alpha-Beta	0.135158539	0.300184488	0.714128971	1.661605597	4.141894102	7.625061274

Finding: From this comparison table, it can be seen that as the number of search layers increases, the speed ratio of the algorithm with Alpha-beta pruning compared to without it also increases similarly exponentially. And when the number of search layers is 10, the speed ratio is more than 20 times.

Comparison of the time spent in different stages

Firstly, let's define the early stage as a stage with no more than 10 discs in the chessboard, and the middle stage is a stage with number of disc between 10 and 45, and the late stage is the stage with number of disc more than 45.

Then, I recorded the time used by Minimax algorithm with Alpha-beta pruning. The comparison table is as follows.

Average Time \s	searching level	searching level	searching level
stage	**4**	**5**	**6**
Late	0.174554825	0.335508585	0.674799919
Middle	2.632534504	14.17163062	54.93694592
Early	2.519661427	14.78686857	61.44276762

Finding: It can be seen from the comparison table that in the search of the same number of levels, the search time in the early stage of the game will be much longer than the search time in the later stage. And when the number of search layers is 6, the time difference is nearly one hundred times.

Win rate

I have used the Genetic Algorithm to tune my parameters, and the final version was picked from the last generation. The final version has the highest win rate among all the individuals in the generation. The result can be seen in the Dataset part.

Finding: After selecting the parameters with higher scores in the genetic population, it can be found that a group of parameters with lower corner weights will have a higher winning rate in the actual game. And a set of parameters with higher stabilizer weights perform better in practice. In addition, this experiment also explores the role of some hyper-parameters, such as the size of the population in the genetic algorithm and the time limit of the search. It is found that when the population size is larger, the convergence speed is slower; The looser the search time limit, the higher the win rate.

Analysis

All the experiment results are good to reflect some thoughtful questions. And the results are also meet my expectation.

For the comparison of the time taken with and without Alpha-beta pruning, as the number of search layers deepens, the number of search branches pruned by Alpha-beta will also be greatly increased, so compared with the simple Minimax algorithm, using pruning will greatly reduce the number of nodes to be searched, thus bringing about a relatively fast speed. Just as the analysis of the worst and ideal time complexity of Alpha-beta pruning in the Methodology part, the relationship between them is exponential, just the same as the experiments in practice.

For the comparison of the time spent in different stages, when the number of search layers is the same, the search time used in the early stage of the game is far longer than the search time used in the later stage of the game. This is because in the early stage of the game, the set of optional moves is larger than in the later stage, which will lead to a much larger branching factor of the search, which further leads to a very slow search time in the early stage.

For the analysis of win rate, the low weight of the corner leads to an increase in the winning rate because the corner is the source of the stable piece. Once the corner is occupied, the pawn cannot be flipped again, and it will also make the nearby pawns of the same colour become stable pieces together, which brings great disadvantage to the player. The weight of the stability is the same.

Conclusion

Advantage and Disadvantage

In this project, I have used the Minimax algorithm with Alpha-beta pruning as the decision algorithm, and I also used the Genetic Algorithm for selecting and adjusting the parameters. And the Genetic Algorithm largely reduced the trouble involving the tuning the parameters. With the combination of the two algorithms, high efficiency and goo performance goal has been achieved, which matched my expectation.

There are also some disadvantages in the project. The searching depth is limited by the time limitation, so that the algorithm cannot search for a deeper level, which led to a partial loss of optimality of algorithm. Additionally, the performance of the algorithm is also limited by the implementation and the efficiency of the programming language. At the same time, the algorithm also lacks a excellent evaluation of the current chessboard. Due to the tight deadlines of this project, I am unable to conduct a more comprehensive evaluation of the chessboard according to the situation of the current chessboard.

Experience

In this project, I have learned a lot. First of all, I got a deeper understanding of the game algorithm of zero-sum games. By reading the relevant literature on the implementation of the Minimax algorithm and using Alpha-beta pruning to optimise it, I am more interested in AI and want to explore further. Besides, by adding features to better evaluate the current chessboard and by tuning parameters to optimise the evaluation function with Genetic Algorithm, I can improve the evaluation function.

What's more, in the Genetic Algorithm, I have also tried various evaluation function to assess the situation of the current chessboard such as location weights evaluation, frontier evaluation, stability evaluation and phase evaluation. Eventually, I made the performance of the algorithm significantly improved.

Further Thought

1. In the present version of code, I divides the process of game into three stages. In the future, I can evaluate the current chessboard more precisely and make more reasonable choice by dividing it into more stages and by using Genetic Algorithm to train the parameters.

2. In the present version of code, I just use the alpha-beta pruning to achieve the acceleration of Minimax algorithm. And for the alpha-beta pruning, there are also many methods to optimise it, for example, the Principal Variation Search(PVS) algorithm. In further step, I will also try to implement those algorithms to improve the efficiency of the Alpha-Beta pruning.
3. Further, the Monte Carlo Tree Search (MCTS) and neural network prove to work well in the zero-sum game. Given enough time, I will implement those algorithms to increase the win rate and elevate the quality of the project. Also, I will do more experiments to compare those algorithms with the Alpha-Beta pruning algorithm and get a deep understanding of the advantages and disadvantages of those algorithms.

Acknowledgement

In this project, I would like to express my gratitude to Computer Science and Technology department and my teacher Bo YUAN, and TA Yao ZHAO for they have imparted the knowledge of AI to me. Also, thanks to all the SAs who have kindly helped me overcome problems. Additionally, I want to thank all the resource in the internet that solve many problems for me.

Reference

[1] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” Artificial Intelligence, vol. 6, no. 4, pp. 293 – 326, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370275900193

[2] Orion Nebula, "黑白棋AI：局面评估+AlphaBeta剪枝预搜索" https://zhuanlan.zhihu.com/p/35121997

[3] Brian Rose, "Othello: A Minute to Learn... A Lifetime to Master", 2005

[4] Wikipedia contributors, “Zero-sum game — Wikipedia, the free encyclopedia,” 2021, [Online; accessed 4-November-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Zero-sum game&oldid=1050040981

[5] Reversi. (2022, June 6). Retrieved from Wikipedia, the free encyclopedia: https://zh.wikipedia.org/w/index.php?title=%E9%BB%91%E7%99%BD%E6%A3%8B&oldid=72022980