Reversi Report for Project 1

Course: Artificial Intelligence

Name: 卢斓

ID: 11810935

1. Preliminaries

1.1 Problem Description:

The content of this project is to design a Reversi game using techniques of artificial intelligence. Furthermore, students should use the knowledge learned in classes to improve the intelligence of the game service and defeat as many other AI agents as possible.

Through the designing and modifying process, we students are expected to exercise and master the Minimax Algorithm and Alpha-Beta Pruning Algorithm, and understand and try Monte Carlo Tree Search Algorithm with time being sufficient.

In the project, PyCharm is used as my python editor and some python libraries like numpy, random and datatime play an important role in designing and testing my own codes. In addition, a specialized reversi play website, http://10.20.26.45:8080/, is provided by the teacher for further testing and examining. When a game is started, the process of the game can be shown to the two players and the game data can be downloaded from this website.

What's more, code samples for showing the structure from the botzone wiki and course materials provided by our lab teacher are referenced.

1.2 Problem Applications:

The Minimax Algorithm plays a very important role in a variety of board and

game games while Alpha-Beta Pruning algorithm do help the programmers improve the code implementation efficiency greatly in some occasions. These knowledge and techniques can help us to evaluate different game and competition situations in our life with limited or other resources especially when there are exactly two agents in this environment and then make the best decisions.

2. Methodology

2.1 Notation:

2.1.1 class AI(object): This is the class of the reversi game. With this class we can create a object for a game with many provided functions which are related with situation judgement, data calculations and decision making. It has 4 attributes which are chessboard_size, color, time_out and candidate_list. They represent the size of the chessboard used for testing, the current player who will make a movement, the maximum time limit for making this decision and all valid positions for the chosen user under the current board situation with repeated elements allowed respectively. In addition, the last element of the candidate_list will be chosen as the final choice for the next movement.

2.1.2 go(self, chessboard_go): This is the only function which will be called by the class AI object. All steps and functions needed for picking out an expected position should be realized or called in this function. “Chessboard_go” represents the current board.

2.1.3 check(self, chessboard_check): This is the function for modifying the “self.candidate_list” to get required valid candidate_list for the next movement. “Chessboard_check” represents the current board.

2.1.4 get_list(self, chessboard_list, color_ge): This is the function for finding out all valid positions for the player “color_ge” under the provided situation “chessboard_list” and then append the results to a record list. What is returned is the record list.

2.1.5 minimax_alphabeta(self, chessboard_ab): This is the function for finding out the best movement position for the current user “self.color” where

“chessboard_ab” represents the current board. Attribute “self.candidate_list” is modified to hold all valid movements and the only choice for the next movement should be in the end of the “self.candidate_list”.

2.1.5.1 max_value(chessboard_max, alpha, beta, depth): This is the function for finding out the required position which provide a max score for the current user “self.color” under the given chessboard “chessboard_max”. In this function, “alpha” and “beta” are two parameters used for alpha-beta pruning and “depth” is used for search termination judgement. What are returned are “v” and “node” representing the score and the chosen position for a max score movement.

2.1.5.2 min_value(chessboard_min, alpha, beta, depth): This is the function for finding out the required position which provide a min score for the current user “self.color” under the given chessboard “chessboard_min”. In this function, “alpha” and “beta” are two parameters used for alpha-beta pruning and “depth” is used for search termination judgement. What are returned are “v” and “node” representing the score and the chosen position for a min score movement.

2.1.6 terminal(self, depth, a_list): This is the function for Minimax Algorithm termination judgement. What is returned is the truth value for termination judgement. When “depth” equals 0 or “a_list” which contains valid movements has 0 element, it should return false.

2.1.7 result(self, chessboard_re, dot, color_re): This is the function for simulating a movement where “chessboard_re” is the chessboard before this movement, “dot” is the position for this movement and “color_re” is the player for this movement. The return is the chessboard after the movement.

2.1.8 cal_stable(self, chessboard): This is the function for approximately calculate the number of stable discs for both sides under the given situation “chessboard”. The return is the number of stable discs of the current player “self.color” minus number of stable discs of his opponent “-self.color”, which is the difference value.

2.1.9 utility(self, chessboard_ul): This is the function for evaluating and calculating the score of the given chessboard situation “chessboard_ul”. What is returned is the score considering “self.color” as the positive player.

2.2 Data Structure and key values:

2.2.1 self.chessboard_size: A value of the size of the chessboard used for testing.

2.2.2 self.color: A value of the current player who will make a movement.

2.2.3 self.time_out: A value of the maximum time limit for making this decision.

2.2.4 self.candidate_list: A list of all valid positions for the chosen user "self.color" under the current board situation with repeated elements allowed. In addition, the last element of the candidate_list will be chosen as the final choice for the next movement.

2.2.5 chessboard(including "chessboard" with suffixes like "_go", "_check", "_list" and so on) and new_board(in result(self, chessboard_re, dot, color_re)): A numpy array of size 8*8 for presenting a chessboard situation with 1 in a position representing this position is located by a white piece, -1 implying a black piece and 0 meaning a empty free position.

2.2.6 eval1: A numpy array of size 8*8 for showing the location weight values of the 64 different positions in a 8*8 chessboard.

2.2.7 eval2: A numpy array of size 8*8 with 1 in every position. It is used for calculating the number of pieces of those belong to the white player minus those belong to the black player.

2.2.8 a_list(in get_list(self, chessboard_list, color_ge)): A list of all valid positions for a chosen user "color_ge" under a given board situation "chessboard_list".

2.2.9 direction: A set of 8 valid moving directions.

2.2.10 visit(in cal_stable(self, chessboard)): A numpy array of size 8*8 which gives information on whether a position in a given chessboard has already been considered and added into result during the process of calculating the number of stable pieces.

2.2.11 ai(in main): A object for class AI representing one chess game with a group of methods mentioned in part 2.1 provided. This is used for self test and simulate.

2.3 Model Design:

2.3.1 Problem Formulation:

For a 8*8 chessboard, the problem can be seen as picking out the position which can provide the highest score for the player who is making this decision under a chessboard given if there is still any empty locations for a valid movement, which should be repeated for 60 times with the decision maker changing between the white player and the black player in turn. In every turn, the chessboard provided should be the result status of all the previous movements. Initially, the chessboard will be set to have 4 pieces in the center as required and the two players can choose their playing orders.

2.3.2 Solution Description:

For one movement, the implement process can be shown as below which is invoked by the method “go(self, chessboard_go)”.

Method “go()” calls method “check()”.

Method “check()” calls method “minimax_alphabata()” during which the search methods “max_value()” and “min_value” are called in turn until the terminal condition id true. Functions “utility()”, “cal_stable()” and “result()” are called in this process for evaluating the scores and generate new chessboard situation. The result of the “minimax_alphabata()” will be append to the self.candidate_list.

Finally, the end element of the self.candidate_list being chosen as the move.

2.4 Algorithm Details:

A brief description for all functions has been given in part 2.1. Therefore, only important and complex functions will be discussed in this part with additional information shown.

2.4.1 Minimax Algorithm along with Alpha-Beta Pruning:

```
def minimax_alphabeta(self, chessboard_ab):
 def max_value(chessboard_max, alpha, beta, depth):
        a_list = self.get_list(chessboard_max, self.color)
        node = (-1, -1)
        if self.terminal(depth, a_list) is true:
            return self.utility(chessboard_max), node
```

```
            v = -1e10
            for every a in a_list:
                val, ns = min_value(self.result(chessboard_max, a, self.color),
alpha, beta, depth - 1)
                v = max(v, val)
                if v >= beta:
                    return v, node
                if v > alpha:
                    alpha = v
                    node = a
            return v, node

        def min_value(chessboard_min, alpha, beta, depth):
            a_list = self.get_list(chessboard_min, -self.color)
            node = (-1, -1)
            if self.terminal(depth, a_list) is true:
                return self.utility(chessboard_min), node
            v = 1e10
            for every a in a_list:
                val, ns = max_value(self.result(chessboard_min, a, -self.color),
alpha, beta, depth - 1)
                v = min(v, val)
                if v <= alpha:
                    return v, node
                if v < beta:
                    beta = v
                    node = a
            return v, node

        val, ns = max_value(chessboard_ab, -1e10, 1e10, 3)
        if ns is not (-1, -1):
            Update self.candidate_list by appending ns to its end
```

2.4.2 Evaluating Function:

```
def utility(self, chessboard_ul):
    val1 = position weights
    val2 = discs number
    count = number of moves being implemented
    val3 = mobility
    num = number of stable discs

    if count < 15:
        val = (val1 + val2 * 0.5) * self.color + val3 * 15 + num * 1000
    elif count < 35:
        val = (val1 * 0.8 + val2) * self.color + val3 * 60 + num * 1000
    elif count < 45:
        val = (val1 * 0.5 + val2 * 2) * self.color + val3 * 50 + num *
1000
    else:
        val = (val1 * 0.3 + val2 * 10) * self.color + val3 * 50 + num *
1000
    return val
```

2.4.3 Stable Disc Calculation:

```
def cal_stable(self, chessboard):
    num = 0
    For every corner of the chessboard:
        If corner is occupied:
            If corner's color is self.color:
                num += number of discs connected with the corner and
            with the corlor
            Else:
                num -= number of discs connected with the corner and with
            the corlor
    Return num
```

3. Empirical Verification

3.1 Dataset:

For this project, I did not use any datasets since no standard test dataset is suitable for this Reversi game.

For testing, 10 cases were provided by the teacher in the usability test. These test cases help me to generate a valid “candidate_list” which contains all valid positions for a next movement. If not passing all tests, a self-realized code would be implemented to simulate a decision making process and print out the content of the candidate_list in the console which will be compared with all valid moves found by myself manually for finding and correcting the bugs.

In addition, during the points race and the round robin, I also used the Reversi platform to play with other opponents. With this platform, every step of the play could be explicitly shown to me and some process information could be downloaded for further analysis. With this information, the real move chosen by the AI could be shown and compared with a more optimized move choice based on my personal analysis of a specialized chessboard situation considering the knowledge I hold for the Reversi game. And then, modify the evaluating function “utility()” to try to get a better move in a specialized step.

What’s more, I have also written code for simulating a whole Reversi game process which is in the main method of my python file for further testing when I want to get more some process information which can not be downloaded from the testing platform or during the period the playto function is disabled. Two testing modes are provided. The first is to simulate a whole Reversi game. Initially, create a chessboard and an AI object and set its four attributes as what we need . Use a counter which starts at 1, ends at 60 and increments by 1 every time to control the times of searching. For one search process, run ai.go() method under this circumstances to get the best movement position and run ai.result() to update the current chessboard. Then exchange the current player “self.color” with his opponent. Print all information needed during this process. The second is to implement search for only one time with an input chessboard. Print the calculated “candidate_list” and its last element which is chosen to be the next movement position. Compared the output result with your expected location and modify the evaluation function “utility()” according to this.

3.2 Performance Measure:

3.2.1 Measure Metrics:

3.2.1.1 worst case time needed:

Since the testing platform set a time limit for searching on one movement, the search will be automatically terminated and the end element of the candidate_list is chosen to be the return position for this move if time exceeds. To prevent an empty candidate_list from being returned, we should run the get_list() method first to get all valid positions and then do the Minimax Search and Alpha-Beta Pruning to find a better position. In addition, if the search depth is set to be too deep, the search will also be terminated without a approximate best result being returned which leads to the last element from the candidate_list being chosen without any evaluation and supporting which is exactly a random choice. To prevent this from happening, a compromise of reducing the search levels is adopted. And according to this, the initial value of depth is set to be 3.

3.2.1.2 Usability Test:

10 test cases were provided by the teacher to help me find out all valid moves for a given player and chessboard.

3.2.1.3 Rank of the Points Race and Round Robin:

The data and information of a play can be seen and downloaded from the testing platform. According to the current rank and the play data, code can be adjusted to get better performance.

3.2.1.4 Self-test Data:

According to the self-test data, code can be adjusted to get better performance. Detailed description of the self-test is given in part 3.1.

3.2.2 Test Environment:

All tests are conducted through either the testing platform provided by the course or the localhost tests. Detailed information is provided in part 3.1.

3.3 Hyperparameters:

3.3.1 Possible Influencing Parameters:

3.3.1.1 Positions:

The importance for different positions in a chessboard are different.

Therefore, different position importance level-board weights should be assigned to different positions. However, the symmetric positions in a chessboard should share the same score.

As for the four corners, they should have a higher score since once they are occupied, there is no possibility for them to be turned into another color, which means they become the stable discs. Furthermore, discs next to them with the same color also become stable discs.

As for C and X positions in a chessboard, they should be assigned with extremely low scores since they will possibly make the corner occupied by the opponent.

3.3.1.2 Stable Discs:

Compared with flipped discs, stable discs will not be possible to be turned into another color once they are occupied, so additional scores should be given to stable discs.

3.3.1.3 Mobility:

Mobility refers to the number of positions which are valid for a player during this turn and it is important for the Reversi game especially during a middle stage. So additional scores should also be given according to the value of the mobility.

3.3.1.4 Disc Number:

Since the goal of the Reversi game is to have more discs in the end, the disc number should also be an important parameter to be considered especially near the end of the stage.

3.3.1.5 Stage:

During the beginning of the game which is set to be less than 15 moves in my file, stability and position weights are more important. However, at the beginning, it is not possible to have much stable discs, so the really important role should be the position weights.

During the middle stage 1 of the game which is set to be less than 35 moves in my file, the mobility becomes more important.

During the middle stage 2 of the game which is set to be less than 45 moves in my file, it is just a little different from the middle stage 1. The position weights and mobility become less important while the number of

discs becomes more important.

Near the end of the game, disc number should play a more important role than the previous stages.

3.3.2 Parameter Adjustment:

For a complete description of the parameter adjustment process, please turn to part 3.1 where two ways of parameter adjustment through the platform and the self-written code respectively are discussed.

3.4 Experimental Results:

For the usability test, the code uploaded by me has passes all test cases. And my final ranks for the points race and round robin are 45 and 61 respectively.

3.5 Conclusion:

3.5.1 Advantages and Disadvantages:

For this project, algorithms of Minimax Search and Alpha-Beta Pruning are used for approximate a best move during a limited time.

In addition, the code designed considers a lot of parameters which do make a difference to the move according to the Reversi theory which is discussed in part 3.3.1. And through the testing process, the evaluating function "utility()" has been continually modified according to the testing data and results which makes the code become more competitive.

However, the calculation of parameters like the mobility is not complete and thus the evaluating function is not that competitive. For simplicity, I only considered the stable discs which are connected with the corners. Besides, the search depth is limited to 3 due to the time limit.

3.5.2 Summary:

Through the project, I do learned a lot including the idea and implementation of the Minimax Algorithm and Alpha-Beta Pruning. What's more, practical skills and experiences are gained for adjusting parameters towards a better performance. Howerver, the final evaluating function is still not that complete or competitive. In the future, efforts can be made to better adjust the parameters accurately especially taking into consideration their weights changing trend design as the stage changes.

4. References

https://wiki.botzone.org.cn/index.php?title=Reversi