CS303 Reversed Reversi Project Report

ManipEgo

I. Introduction

This project is called reversed reversi. Intuitively, this indicates that this project is about a special kind of reversi whose rule for deciding a winner is reversed. To be specific, player who has the least chess on board wins the game when no player is able to settle another legal chess. Our aim on this project is to derive an AI for playing this game, and beat as many other AIs as possible.

II. Preliminary

A. Problem Formulation

The game consists of one 8×8 board and two kinds of chess with different colors white and black. Represent the position on the chessboard with tuples in the form of (i, j) where i represents the row number and j represents the column number, $i, j \in \{1, 2, 3, 4, 5, 6, 7, 8\}$. At the initialization of the game, two pieces of white chess are placed at $(4, 4)$ and $(5, 5)$, and two pieces of black chess are placed at $(4, 5)$ and $(5, 4)$. After the game begins, two players will take turns to place chess in their own color on the board, when there are legal positions. When there is none, the player skips his turn. When both players have no legal position to place any chess, the game ends, and the player with less chess in his color on the board is regarded as the winner. A legal position for a player in color c is defined as (i, j),
$i, j \in \{1, 2, 3, 4, 5, 6, 7, 8\}$,
$\exists x, y \in \{1, 2, 3, 4, 5, 6, 7, 8\} \wedge$
$(i, j) \neq (x, y) \wedge C\{(x, y)\} = c \wedge$
$\{(i = x \wedge \forall (m, n), m = x \wedge min(i, x) < m < max(i, x) \wedge$
$C\{(m, n)\} = -c) \vee$
$(j = y \wedge \forall (m, n) \wedge n = y \wedge min(j, y) < n < max(j, y) \wedge$
$C\{(m, n)\} = -c) \vee$
$(i + j = x + y \wedge \forall (m, n) \wedge m + n = x + y \wedge min(i, x) < m <$
$max(i, x) \wedge C\{(m, n)\} = -c) \vee$
$(i - j = x - y \wedge \forall (m, n) \wedge m - n = x - y \wedge min(i, x) < m <$
$max(i, x) \wedge C\{(m, n)\} = -c)\}$,
where c and $-c$ denotes the two different colors, $c \neq 0$, and $C\{a\}$ denotes the color of the chess on position a if there is a piece, and 0 if there is none.

The object of an agent is to win the game with its best effort, namely to maximize $Count(-c) - Count(c)$ at the end of the game, where $Count(x)$ denotes the number of chess in color x on the board, and c denotes the agent's color, $-c$ denotes the other color.

B. Notations

All notations defined in the *Problem Formulation* section will be used. Other notations besides those are listed below.

- $white\{a, b...\}$ represents position tuples $a, b...$ are occupied by white chess. $black\{a, b...\}$ represents the same for black chess.
- Numbers with bracketed numbers after them are base bracketed number. For example, $10010_{(2)}$ means that 10010 is binary. Numbers without bracketed numbers after them are regarded as decimal.
- c when not specially introduced, represents the color of the player in the context, and $-c$ represents the color of another player.
- $Count(c, s)$ denotes the number of chess in color c on the board at status s. If s is not specified, it denotes the number of chess in color c at the end of the game in general, without certain end status.
- $V(c, s)$ denotes the sum of chessboard weights for a player in color c at board status s. Chessboard weight is defined in $Methodology$ section.
- $M(c, s)$ denotes the mobility of a player in color c at board status s. Mobility is defined in $Methodology$ section.

III. Methodology

A. General Workflow

The proposed method can be generally divided into two phases, both of which are based on the *minimax algorithm* and some combinations of evaluations.

At the first phase, two factors are involved in the evaluation. One is the sum of the position weights where chess are placed, the other is the mobility of the two players, namely the number of possible moves they can take. These two factors are linearly combined to form an overall value formula. The algorithm uses values returned by it to make $minimax$ decisions.

At the second phase, the algorithm can expand the search depth to the end of the game within the given time limit. Hence the algorithm can directly count the number of chess at the end. The algorithm at this stage uses $Count(-c) - Count(c)$ to make decisions.

Additionally, at each phase of the game, the states are further divided into more sub-phases, between which other more hyper-parameters (e.g. search depth) vary. But the decision-making of different sub-phases belonging to the same phase is the same.

B. Detailed Designs

- Binary Representation of the Game Status:
  The status of the chessboard, in this context, refers to different possible sets of positions on which two colors of chess are placed. Intuitively, the status can be represented in two lists of tuples, one contains all the positions on which white chess are placed, another contains all the

positions on which black chess are placed. For example, as introduced in the *Problem Formulation* section, the initial status of the board can be represented as $white\{(4,4),(5,5)\}, black\{(4,5),(5,4)\}$. This can be directly translated into program tuple list data structure. But to accelerate processing, binary representation of the status can be derived. A board can be listed first by row then by column. Hence $8 \times 8 = 64$ digits are enough to represent a board status. If representing only one colored chess, with 1 for placed a chess and 0 for no chess, a certain status can then be represented by a 64 bit integer. To include another color, one more 64 bit integer is needed. For example, if start from the least significant digit, the initial status can be represented as $white : 1000000001000000000000000000000000000000_{(2)}$, $black : 0000000100000000000000000000000000000000_{(2)}$. Hence two 64 bit integers can represent a board status.

- Binary Moves and Position Checks:
  All binary moves can be done with binary shifts. For instance, move one grid downward can be done with an 8 bits left shift. All binary position checks can be accomplished with bit-wise *and* and *or* operations between the status number to be checked and the position check number. For example, to check whether a status s contains a piece of chess at position $(1,2)$, conduct bit-wise and between $10_{(2)}$ and the status number s. If the outcome is greater than 0, that means there is a chess on $(1,2)$, according to the definition of bit-wise and. Similarly, bit-wise *or* helps to place a chess on the corresponding position.
- Minimax Algorithm:
  Minimax is an algorithm based on the Game Theory. It assumes that all moves by a player maximize his own value, all moves by another player minimize his rival's value. Based on the assumption, it searches through all possible moves to find the one with the largest value. A pseudo code for the algorithm is as shown in *Algorithm* 1.
  The MOVES function returns all possible moves according to the state passed in. The GAME-OVER function decides whether the game is at an end. The EVALUATE function evaluates the value of the chessboard state with certain methods. The NEXT-STATE function returns the next state caused by taking the passed in move on given state.
- α-β Pruning:
  On the basis of the *Minimax Algorithm*, it is obvious that the max-value greater than the already-found min-value one step above will not be adopted by the min-value layer above. The same another way round for the min-values. Hence branches with such values can be pruned.
- Genetic Algorithm:
  The genetic algorithm conducts round robin games between every pair of agents with different hyper parameters in each generation. Then, it evaluates each agent's win rate, and saves the agents with the highest win rates. It crossbreeds winners in the last generation at the beginning of each generation after the first one, and round robin the new agents with other randomly generated ones again. After some certain generations, the rather good hyper parameters accumulates within best agents, which help *Minimax Algorithm* decide hyper parameters.
- Weighted Chessboard:
  As part of the evaluation, a weighted chessboard gives different values to every single position on the chessboard. But according to the symmetry of the chessboard, only 10 sets of positions are different in value. After the values and the status of the chessboard are given, an evaluation function $V(c,s)$ sums all positions' values where there are placed chess in color c at board status s. For a player, maximizing the value of $V(c,s) - V(-c,s)$ is considered a good decision.
- Mobility:
  Another evaluation method. The mobility of a player in c at a given board status is defined as the number of possible moves he can take. To maximize $W(-c,s) - W(c,s)$ is considered a good decision.
- Evaluation Functions: The final evaluation values used for decision making in the phase one of *Minimax Algorithm* is given by a linear combination of $V(c,s) - V(-c,s)$ and $W(-c,s) - W(c,s)$. For example, $1 \times (V(c,s) - V(-c,s)) + 2 \times (W(-c,s) - W(c,s))$. The final evaluation values for decision making in the phase two is given by $Count(-c) - Count(c)$.

Algorithm 1 Minimax Algorithm

```
1:  procedure MINIMAX(state, color, depth_max)
2:      v ← MAX-VALUE(state, color, depth)
3:      return the move MOVES(state, color) with value v
4:  end procedure
5:
6:  procedure MAX-VALUE(state, color, depth)
7:      if GAME-OVER(state) or depth = 0 then
8:          return EVALUATE(state, color)
9:      end if
10:     v ← −∞
11:     for each move in MOVES(state, color) do
12:         v ← max(v, MIN-VALUE(NEXT-STATE(state, color, move), −color, depth − 1))
13:     end for
14:     return v
15: end procedure
16:
17: procedure MIN-VALUE(state, color, depth)
18:     if GAME-OVER(state) or depth = 0 then
19:         return EVALUATE(state, color)
20:     end if
21:     v ← ∞
22:     for each move in MOVES(state, color) do
23:         v ← min(v, MAX-VALUE(NEXT-STATE(state, color, move), −color, depth − 1))
24:     end for
25:     return v
26: end procedure
```

C. Analysis

An α-β *Pruning Minimax Algorithm* degenerates to simple *Minimax Algorithm* in the worst case, namely it prunes no branch. In that case, assume that the number of search layers is d and the branch coefficient is b, then the time complexity of the algorithm is $O(b^d)$. In the ideal cases, the algorithm picks best moves on the left sub-tree. The time complexity is then $O(b^{d/2})$.

The deciding factor of this algorithm is the order of search. Since that the algorithm skips only inferior moves compared with current best move, if the global best move is searched first, then all other moves will be skipped. On the opposite, if the algorithm searches moves from the worst to the best, then it degenerates to the plain $MinimaxAlgorithm$.

IV. Experiments

A. Setup

- Environments:
  Hardware Environment:
  CPU: AMD Ryzen 7 4800U with Radeon 8 Cores
  RAM: 16GB DDR4 3200MHZ
  Software Environment:
  Python: 3.9.12 with Anaconda
  Numpy: 1.21.5
  No other package used
- Data-sets:
  No prepared data-set is used in testing. All data are generated by a newly implemented game controller. The controller instantiates two agents to fight each other, and collect and monitor wanted data during the game. Different hyper parameters are accepted and number of rounds can be set so that the performance of any two sets of parameters can be tested. The controller visualize game process with "+", "O" and "X" characters and save step information according to the settings. It also time the game and terminate agents that exceeds the time limit. The *Genetic Algorithm* also reuses this controller to initiate a round robin.

B. Results

- Measurements:
  The performance of agent is first measured with time cost. All agents that should be further measured must not exceed 5 seconds run time per step through out the game. Then the performance is measured with the number of chess at the end of the game, the less the better. Finally, the equally valued agents are put into a round robin, where their win rates decides the best agent.
- Experimental Results:
  Many sets of experiments are conducted, and all details cannot be listed. An example is shown here:
  The adaptability here refers to the counts of winning rounds. By comparing the adaptabilities, it can be found out that chessboard weights improve performance largely, since that the agent with all 0 weights merely wins. It is also obvious that the deeper one agent searches within the time limit, the better it performs.

TABLE I
Experiment Example

Parameter sets	1	2	3	4
adaptability	22	19	19	1
depth lists	[5,3,6]	[5,2,6]	[4,3,7]	[5,3,7]
weight lists	[-500,20...]	[-500,25...]	[-500,15...]	[0,0...]

- Analysis:
  All experiment results are good and conform to the expectation.
  The chessboard weights improve performance largely, for that different positions are statically unequal in terms of significance. Possessing certain positions at certain stages of the game more likely leads to better situations. Hence making the agent have preferences on positions is beneficial.
  On that basis, the importance of search depth is also explainable. The deeper it goes, the more situations it consider, the better chance of possessing good positions it can find. A noticeable fact here is that the depth limits are set to different values according to different stages of the game, defined by the total count of chess on the chessboard, so that the agent can stay in the time limit through the whole game without wasting time when the search space is narrow.
  The evaluation performance here also conforms to the analysis in the *Methodology* section, where why an agent uses such evaluation functions involving weighted chessboard is explained.

V. Conclusion

A. Advantages & Disadvantages

The *Minimax Algorithm* makes it clear to decide the moves step by step, thanks to the assumption of the Game Theory. But the problem is, in real games, the opposite player probably has a totally different set of evaluations, hence assuming he play the best under the agent's own circumstance likely does not work. A just suitable set of parameters is fatal to the real game performance of this algorithm.
The *Genetic Algorithm* makes it easy to tune the hyper parameters, only at the cost of super large amount of time. In practice, the algorithm took about 72 hours to evolve 168 generations with each at a size of 64. Another difficulty is that it is hard to decide the variation parameters of the algorithm, to avoid both drifting too far away from the better answers and taking too long to find a good enough answer.

B. A Third Look at the Experiment Result

After all that analysis, the agent is also tested online with other agents derived by other students in this course. It performed just as expected. The performances are largely influenced by the suitability for the rival agents' parameters, especially when they also use *Minimax Algorithm*.

C. Lessons Learnt

When working on a time consuming learning algorithm, it is better to start early and have access to better hardware, since every adjustment to it takes long time to be verified.

D. Further Thoughts

- First, very intuitively, the hyper parameters can be further improved with a larger and deeper evolution of the *Genetic Algorithm*.
- Second, the stages and phases of the game can be further divided, until every step of move has its own set of parameters.
- Third, the efficiency of the code can be further improved. For example, use $numba$ to enhance the speed.
- Last, other algorithms like the *Monte Carlo Tree Search* and some Neural Networks can also be implemented as alternatives.

REFERENCES

[1] Brian Rose, "Othello: A Minute to Learn... A Lifetime to Master", 2005
[2] 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
[3] Orion Nebula, "黑白棋AI：局面评估+AlphaBeta剪枝预搜索" https://zhuanlan.zhihu.com/p/35121997