Reversed-Reversi: Project 1 of CS303 Artificial Intelligence

11912039 Xinying Zheng : Department of Computer Science and Engineering

November 5, 2021

1 Preliminaries

1.1 Problem Description

1.1.1 Goal

Reversed Reversi is a board game that asks Players to place disks on the board in turn. Each turn, any disks of the opposite color that are nipped by the disk of current player's color will be turned over to the current player's color. The object of the game is to have the fewest discs in the board that display your color when the last playable empty square is filled. The goal of this project is to implement an algorithm to play this game that is as intelligent as possible.

1.1.2 Software and Hardware

The project is written with Python 3.9.7, with pycharm as code editor. The configuration of the test platform is Intel Xeon Gold 6240 @ 64x 2.6GHz.

1.1.3 Algorithm

In my implementation, I use alpha-beta search as basic algorithm frame together with smart evaluation function. The evaluation function consist of several contributing factors such as chess difference between the two sides and the weight matrix.

1.2 Problem Application

This kind of algorithm is of wide use in our life. For example, Now many games developers use alpha-beta pruning algorithm to achieve its robot model especially board games.

2 Methodology

2.1 Notation

- num(self/oppo): number of discs of us or opponent.
- ChessDiff(state)=num(self)-num(oppo).
- Weight[x][y]: denote the weight scores of point [x][y].
- Stable(state): denote the number of stable points of our color.
- board[x][y]: denote the point in row x and column y, where $0 <= x <= 7, 0 <= y <= 7$.

2.2 Data Structure

- class **move**: to store the action and the affected opponent's discs.
- field **candidate-list** in class AI: a list of the valid positions which have found.
- field **candidate-move** in class AI: a list of the corresponding move.
- **weight**: a weight matrix to measure the importance of each position.
- **High score** in the matrix means a relatively disadvantage place while **low score** means a relatively good place.
- **[1,0],[-1,0],[0,1],[0,-1],[1,1],[1,-1],[-1,1],[-1,-1]**: eight directions of a point in the board.

2.3 Model design

I use the **alpha-beta pruning** algorithm as the basic frame with a smart **evaluation function**, the steps are as follows.

- Decide maximum search length
- Max-value-search for next action
  - find all valid action
  - Min-value-search
- Take action

The main ideas are as follows:

2.3.1 Search depth

The search depth dynamically changes according to current situation. To get the action within time, I make the search depth smaller in the middle and larger at the beginning and end.

2.3.2 Get Action

Consider all blank points, search its eight directions. If there is direction on which opponent's discs are nipped by ours, then this point is a valid place.

2.3.3 Min-Max search

During the process of the game, we are always trying to strives for the maximum return in every step, which is considered as the **Max process**. If we construct the search tree, the corresponding nodes in it are called **Max nodes**. We don't know what our opponent will do, the only thing we know is that he want to minimize our gain as much as possible,so we just consider it to take the action to achieve this goal, which is called **Min process**. The corresponding nodes in the game tree are called **Min nodes**. So playing chess is all about alternating Max and Min. To behave intelligently, we should searches as many steps forward as possible. As the search deepens, the choices get smarter and smarter.

2.3.4 alpha and beta pruning

The only limit is the **time** and **space**. To save the searching time so that we can search more deeply, I adopt **alpha-beta pruning** to cut off some unnecessary node. Two values are introduced for each node: **alpha and beta**, which respectively represent the lower limit and upper limit allowed by the node's valuation. At the beginning, the root alpha = -infinity, beta = infinity. For the Max node, if the return value of the child node is greater than alpha, alpha is updated to that return value. For the Min node, the process is reversed, and beta is updated to the return value of its smaller child node. When *beta is less than or equal to alpha*, pruning occurs.

	0	1	2	3	4	5	6	7
0	500	-25	10	5	5	10	-25	500
1	-25	-45	1	1	1	1	-45	-25
2	10	1	3	2	2	3	1	10
3	5	1	2	1	1	2	1	5
4	5	1	2	1	1	2	1	5
5	10	1	3	2	2	3	1	10
6	-25	-45	1	1	1	1	-45	-25
7	500	-25	10	5	5	10	-25	500

Table 1: The Weight Matrix.

2.3.5 Evaluation function

I implement an evaluation function to measure each player's income. The contributing factors are as follows. [1]

$$Score = \sum_{0}^{7}\sum_{0}^{7} weight[i,j] * state[i][j] - a * stable\_cnt - b * chess\_diff$$

where a,b is coefficient.

- **weight matrix(Table 1)**:
  1. **Each point in the board has different importance for a player's condition.** For example, the four corner points can bring vital disadvantages since when we occupy them, the number of discs with our color permanently increase as they can never be turned again. So we should avoid to occupy it. Generally, low score in the middle, high around.
  2. **The weight matrix is dynamic.** For example, initially, the point next to the corner is a relatively good place, but when the corner has been occupied by us, it will immediate become a bad place as well. So we should dynamically change the weight.
- **stable disc**:
  1. Stable discs refer to those discs that cannot be turned by the other side anymore.
  2. **A disc is stable in one direction if and only if it is connected with a stable point in that direction or is connected with border.**
  3. **A disc is stable if and only if it is stable in four directions and these four directions are not opposite to each other**((1,0),(-1,0) contribute to only one direction's stable).
  4. To accurately calculate the number of stable discs spend much time so I choose to only calculate those points at which **both rows and columns and both diagonals are filled**.
  5. The stable discs don't appear in the beginning steps, so we add it into evaluation function after some steps.
- **chess diff**:
  1. Scan the board to get the number of discs of our color and opponent's.

2.4 Detains of Algorithm

2.4.1 Get-MAX-DEPTH()

According to current round, get the search depth. Pseudo-code refers to Algorithm 1.

2.4.2 Get-Action()

Get all valid actions of current player.Pseudo-code refers to Algorithm 2.

Algorithm 1 Get-MAX-DEPTH()

Input: Current State: $state$.
Output: Maximun search depth in this turn: $depth$.

```
depth ← 4;
round ← state.round;
if round <= 3orround >= 50 then
    depth ← 6
end if
if round <= 5orround >= 45 then
    depth ← 5
end if
```

Algorithm 2 Get-Action()

Input: Current State: $state$,current color:$color$.
Output: A list contains all valid point in this turn : $candidate - list$.

```
candidate − list ← empty list;
emptyPoints ← all blank points;
for point in emptyPoints do
    flag ← False;
    for direction in eight drection do
        next ← next point in this direction;
        if next is opposite color then
            next ← next point in this direction;
            while next is opposite color do continue;
            end while
            if next is empty then
                flag ← True;
            end if
        end if
    end for
    if flag is true then candidate-list.add(point)
    end if
end for
```

2.4.3 Get-Weight()

Get Weight Score of current board. Note that a high weight means a good situation for us, so we should add it to total_score. Pseudo-code refers to Algorithm 3.

Algorithm 3 Get-Weight()

Input: Current State: *state*.
Output: weight score of the state: $Value$.

```
Value ← 0;
if the corner has been occupied by us then
    the weight of point next to the corner ← 250
end if
if the corner points is empty and is nipped by the opposite point then
    the weight of corner ← 1000
end if
for every point in the board do
    if point is occupied with our color then
        Value− = Weight of the point
    end if
    if point is occupied with opponent's color then
        Value+ = Weight of the point
    end if
end for
```

2.4.4 Get-Stable()

Get the stable discs of our color. Note that more stable discs, worse situation for us, so we should subtract it from total_score. Pseudo-code refers to Algorithm 4.

2.4.5 Get-Chess-Diff()

Get the chess Difference of current state. Note that the bigger the chess difference, the worse our situation, so we should subtract it from total_score. Pseudo-code refers to Algorithm 5.

2.4.6 Calculate()

Here we get a state, and calculate its score based on my evaluation function. Pseudo-code refers to Algorithm 6.

2.4.7 Max-value()

Max node in the search tree. Pseudo-code refers to Algorithm 7.

2.4.8 Min-value()

Min node in the search tree. Pseudo-code refers to Algorithm 8.

2.4.9 Go()

Act as a main function. All logic begins here. Pseudo-code refers to Algorithm 9.

3 Empirical Verification

3.1 Dataset

There are several source of my test code. I write a script and take state matrix as input to test whether my code will generate all valid action.

Algorithm 4 Get-Stable()

Input: Current State: $state$.
Output: Number of stable discs of our color: num.

```
num ← 0;
if the corner has been occupied by us then
    num+ = 1;
end if
for four corner points do
    Expand the point clockwise, increase the num if it is occupied by us;
    Expand the point counterclockwise, increase the num if it is occupied by us;
end for
for All rows do
    if all points in this row is occupied then
        Sign this row as full
    end if
end for
for All columns do
    if all points in this column is occupied then
        Sign this column as full
    end if
end for
for All 15 the diagonals with slope 1 do
    if all points in diagonal is occupied then
        Sign this diagonal as full
    end if
end for
for All 15 the diagonals with slope -1 do
    if all points in diagonal is occupied then
        Sign this diagonal as full
    end if
end for
for all point in the board with our color do
    if this point in a full row, full column and two diagonals are full then
        num+=1;
    end if
end for
```

Algorithm 5 Get-Chess-Diff()

Input: Current State: $state$.
Output: Chess-Diff: $Diff$.

```
diff ← 0;
for every point in the board do
    if point is occupied with our color then
        diff+ = 1 of the point
    end if
    if point is occupied with opponent's color then
        diff− = 1 of the point
    end if
end for
```

Algorithm 6 Calculate

Input: Current State: *state*.
Output: Score of current state: *Score*.

$Score \leftarrow 0$;
$Stable \leftarrow 0$
$Weight \leftarrow Weight(state)$;
$Chess - Diff \leftarrow Get - Chess - Diff(state)$
if $round >= 30$ **then**
  $Stable \leftarrow Get - Stable(state)$
end if
$Score \leftarrow Weight - 10 * Stable - 15 * Chess - Diff$;

Algorithm 7 Max-value

Input: $state, alpha, beta, depth, MAX\_DEPTH$.
Output: v,$action$.

$depth >= MAX\_DEPTH$ return Calculate(state),None;
$Valid\_action \leftarrow Get - Action(state)$
if Valid_action is empty **then**
  if opponent have valid action **then**
    Min-value(state,alpha,beta,depth+1,MAX_DEPTH)
  else
    return Calculate(state),None;
  end if
end if
$v \leftarrow -infinity$
$action \leftarrow None$
for every valid action **do**
  Change state due to the action
  v,tmp = Min-value(state,alpha,beta,depth+1,MAX_DEPTH)
  rollback the state
  update v,alpha,action
  if $v >= beta$ **then**
    return v,action
  end if
end for
return v,action

Algorithm 8 Min-value

Input: $state, alpha, beta, depth, MAX\_DEPTH.$
Output: v,$action.$

$depth >= MAX\_DEPTH$ return Calculate(state),None;
$Valid\_action \leftarrow Get - Action(state)$
if Valid_action is empty **then**
  if opponent have valid action **then**
    Max-value(state,alpha,beta,depth+1,MAX_DEPTH)
  else
    return Calculate(state),None;
  end if
end if
$v \leftarrow -infinity$
$action \leftarrow None$
for every valid action **do**
  Change state due to the action
  v,tmp = Max-value(state,alpha,beta,depth+1,MAX_DEPTH)
  rollback the state
  update v,beta,action
  if $alpha >= v$ **then**
    return v,action
  end if
end for
return v,action

Algorithm 9 Go()

Input: Current State: $state.$

$MAX\_DEPTH \leftarrow Get - MAX - DEPTH(state);$
$Value, action \leftarrow Max - value(state, -infinity, infinity, 0, MAX\_DEPTH)$
$round+ = 1$
if action is not empty **then**
  candidate_list.add(action);
end if

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	-1	0	0	0	0	0
3	0	0	-1	-1	0	0	0	0
4	0	0	-1	1	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Table 2: Test case example

- I ask some of my classmates for their logs and get the state matrix which consist of -1 and 1 like Table 2. Manually judge whether my code will generate all valid action.

- I design some common and extreme test cases by playing the game and choose some states and transfer them to state matrixes.
  1. The first round, last round, middle round with a relatively large number of valid actions.
  2. The state when current player don't have any place to put disc.
- For the round robin
  1. I choose several state which have many valid actions and find its best action manually and compare with what my algorithm choose.

3.2 Performance measure

The performance can be measured in following factor.

- **Max_Depth within time limit**
- **Longest time on making a decision**
- **Average time on making a decision**
- **Test cases passed in the usability test**
- **The winning percentage on the platform**
- **Rank**

3.3 Hyperparameters

There several parameters I used in my algorithm, I list them and their values as following.

- Weight at each position
  1. Initially the weight of every position was set as Table 1;
  2. I choose 250 as weight for the point next to an occupied corner points;
  3. I choose 1000 as weight for the corner points which nipped by opposite discs;
- The coefficient before stable_cnt and chess_diff in the evaluation equation;
  1. a=10 and b= 15
- Max_depth each round
  1. The only way I judge it is that it cannot exceed the time limit of the platform;

2. I choose to search 6 layers and 5 layers in the very begging and last phase and 4 layers in the middle;

- How precious and at which round should I calculate the number of my stable discs

  1. By observation, I notice that the stable discs don't appear until around 30 round of the game;
  2. And it is a time-consuming step, we don't need to find the exact number of stable discs;
  3. So I choose to count only discs which in a full row, full column and dull diagonals after 30 rounds;

In this project, I don't have local methods to improve my Hyperparameters. I just choose randomly at first and rely on round robin to improve it, which is finally proved to be a bad idea. But now, I have some ideas to adjust my Hyperparameters. [2]

- Figure out about 10 extremely good states and extremely bad states and some normal states, calculate it's score using my evaluation function, judge whether its value satisfy our expectation.
- For every lost game, trace it and see which step directly leads to its failure, adjust the hyperparameters carefully to make a different decision.
- Using the way of machine learning to train the model and get a smart hyperparameters(long way to go).

3.4 Experimental results

I think I should choose my hyperparameters more smartly and do more experiments so that it can behave more intelligently. The Experimental results are as following

- **Test cases passed in the usability test :** 10;
- **The winning percentage on the platform:** 30% with win_count:118; lose_count:287.
- **Rank:** 160.
- **Max_Depth within time limit** : 6 or 7 in the first and last three round, 4 in the middle of the game.
- **Longest time making a decision** : 4.78s.

3.5 Conclusion

3.5.1 Advantages

- A relatively clear code structure.
- By observation, I can always get a good situation in the beginning.
- Use alpha-beta pruning to save the search time.
- Dynamically change my hyperparameters.

3.5.2 Disadvantages

- Bad hyperparameters choice in the evaluation function.
- No specific techniques to choose hyperparameters.
- No adequate local testing for my code.
- Can include more factor in my evaluation function, and it's weight in the whole score should also dynamically changed during the game.
- Unnecessary copy of the list in my code.

3.5.3 improvements

- Dynamically change the weight of every contributing factor in the evaluation equation.
- In searching steps, for every child node, we can first search shallowly and sort them according to their value. And choose the first several node to do a more deep search.
- Although alpha-beta is an improvement over MinMax search, it is only a minor improvement because the order of nodes at each level is not handled in a special way, wasting more pruning opportunities. So we can adopt some special techniques to increase pruning.
  1. Divide nodes into 3 categories.
     - **Very good nodes**: nodes where alpha_beta pruning happens.
     - **Contributing nodes**: nodes that update alpha/beta value.
     - **Very bad nodes**: none of above cases happens.
  2. This kind of dividing is very stable, which means that it's nearly impossible that we have a node that's very bad this turn of search, but the next search we have the same node very good. So we can introduce a **history table mechanism** to record this partition, which ensures that every search starts with the very good nodes, then the contributing nodes, and finally the very bad nodes, thus greatly increasing the number of pruning occurrences.
  3. When we meet **Very good nodes**, we insert it to the first position of all child nodes.
  4. When we meet **Contributing nodes**, we exchange it with its front node.
  5. When we meet **Very bad nodes**, we do nothing.
- Besides, there is another information that has not been well utilized. There are many same nodes that have appeared before. So it can save a lot of time to make good use of the information retained by these nodes. Therefore, a **Transposition table** is introduced, it records the relevant information of the node encountered. When a node is searched, if the record of the node is found in the replacement table and is completely matched with the current node, we can just return the evaluation value saved in the table without further search. Otherwise, if it is partial matching, the best action in the history of the node is really valuable information.

References

[1] C. Rose, *Othello*. Macmillan Education, 2015.

[2] M. Buro, "Statistical feature combination for the evaluation of game positions," *Journal of Artificial Intelligence Research*, vol. 3, pp. 373–382, 1995.