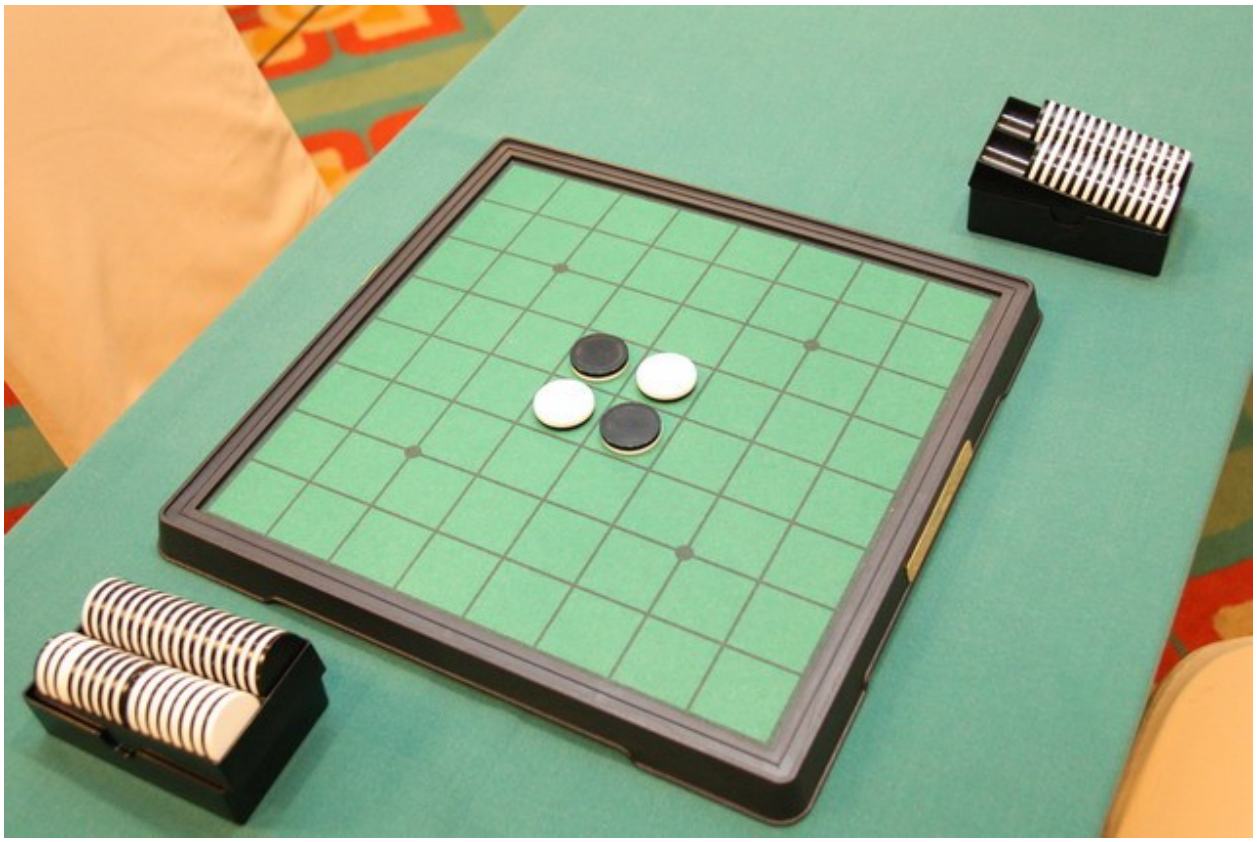


The picture comes from https://en.wikipedia.org/wiki/Reversi

Better AI for Reversi

1. **Introduction:**

In this project, we mainly focus on analyzing existed agents of a chess game called Reversi. When we were doing some background information search, we found there already existed more than 20 different agents, while they are taking various algorithms. This phenomenon aroused our interest. After one semester learning, we had learned many search algorithms in artificial intelligence field. Thus, our first goal is to compare those existing agents, what are the advantages and drawbacks of different algorithms. When some have a higher success rate while others do not.

What's more, we were trying to do some improvements based on current programs or even come up with new methods. For instance, it seems that most of the existing programs are using Markov Decision Process(MDP). Can we develop some new heuristic methods? Moreover, is it possible that we introduce Reinforcement Learning(RL) into this topic and develop an algorithm? How can we make it?

To be specific, the input of our programs will be an MDP with states S, actions A(s), transition model P(s'| s, a) and rewards R(s, a, s'). The output will be a policy guiding the agent on how to move.

2. **Background Information:**

Reversi, as known as Othello, is a classical chess game. Two players, one plays black and the other one plays white, take turn placing pieces on an 8 × 8 grid board. Every move must "reverse" at least one of the opponent's pieces. In order to reverse, player1 should place a piece adjacent to one of player2's pieces, so that there a straight line (horizontal, vertical or diagonal) ending with player1's pieces and connected continuously by player2's pieces. Then the player2' pieces on the line will change to player1's. One will win if he has reversed all his opponent's pieces. And if the chess board is full of players' pieces, then the one with larger number of pieces wins.

This chess game can be turned into a search problem. To be specific, it is a simple zero-sum game. Action space are those grids which players can place a piece into, and goal state is either all the opponent's pieces have been reversed or the board is full and we have a higher pieces' number than our opponent.

Currently Reversi with 4x4 board and 6x6 board problem have been "solved". As the board size is not very large, we can actually compute every single possible strategy. The problem size of 4x4 Reversi is 2^(4 x 4) = 65536, while problem size of 6x6 board is 2^(6 x 6) = 6.8719 x 10^(10). However, for 8x8 board problem, although human have "solved" it in practice, we still lack mathematical proof. After compared about 15 existing Reversi AI on GitHub and analyzed its learning algorithm, we basically conclude in 3 types.

1. **Comparison:**

2.1 "Dummy Ai"

As the final result is a comparison of the number of pieces, the easiest strategy we may think is "reverse as many pieces as we can". We called this agent "Dummy Ai", and this kind of agents can be beat by any experienced person. Actually according to David(2018), the more pieces you have in the early stage of the game, the higher chance you may lose.

2.2 MDPs

To develop a better agent, we decide to analyze Reversi itself. Is there any common methods to win the game? After analysis, we found two special impacts, "Stable Points" and "mobility".

“Stable Points” refers to a kind of special points which can’t be reversed. For example, all the corners of the board are stable points, as there’s no way they can be reversed. And if a boundary of the board is full of one player’s pieces, points in this whole line will become stable points. Thus, it’s easy to see that “Stable Points” are not fixed. One grid can become stable point under some condition. And after many experiments, we found that taking stable points always led us to a higher possibility to win this game. Not only because having “Stable Points” means that we have these several pieces as our own, but also because that “Stable Points” are usually boundary points, and it’s really easy for a player to reverse all the opponent’s pieces in the lines ending with these stable points. In practice, if someone gained one “Stable Point”, the increase in the number of pieces he gets is at least 8.

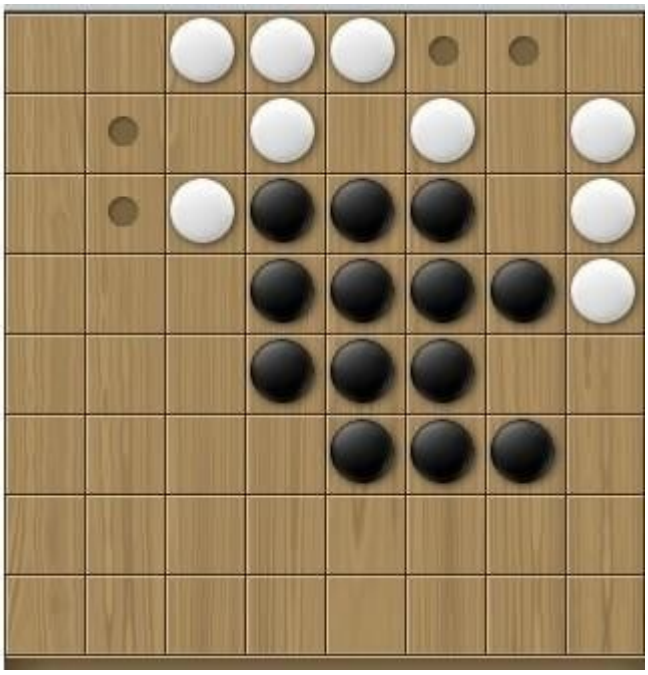
Based on this theory, a value table can be constructed for the 8x8 chess board.

	A	B	C	D	E	F	G	H
1	90	-60	10	10	10	10	-60	90
2	-60	-80	5	5	5	5	-80	-60
3	10	5	1	1	1	1	5	10
4	10	5	1	1	1	1	5	10
5	10	5	1	1	1	1	5	10
6	10	5	1	1	1	1	5	10
7	-60	-80	5	5	5	5	-80	-60
8	90	-60	10	10	10	10	-60	90

As mentioned earlier, the four corners, are the only fixed stable points in the whole board. Thus, they own the highest value. And those grids near the corners, however, has an extremely small value, in this table, they are labeled as -60 to -80. The reason why these grids are negative is that if an agent places a piece on that grid, the opponent will occupy the stable points in a highly chance. Thus, agents should avoid step on these grids.

Other boundary grids have a relatively high value, positive 10. This is because line grids are more likely to become stable points. And the middle 4 x 4 grids, have equal values of 1. As no matter where u place your pieces, it will not affect the stable points.

Just based on this single table is not enough. Another key concept is called “mobility”, which actually refers to action space. Different with many other chess games, action space in Reversi varies from time to time. Thus, in each move, maximum own’s future action space and minimize the opponents’ should also be considered as well.



In real programs, we will construct an evaluation function considering both the value table and the mobility. And as Reversi is a zero-sum game, with this evaluation function we can conduct Min-Max search on an agent.

$$V(S) = \sum_{i=1}^{8} \sum_{j=1}^{8} \omega[i,\ j] \cdot s[i,\ j] + \text{Mobility}(\ my\_color)$$

Moreover, as in the later period action space can be very large, we'll also introduce Alpha-Beta Pruning to reduce the computation size. Here's the code of this kind of agent.

2.3 "Open Book"

Though human still cannot list all the outcomes of an 8x8 Reversi, some programs are using partial of them to further enhance their agents. They called it an "Open Book", which contains strategies of looking-ahead of the first 24 moves. As the game is easy to lose control in the first few moves, agents can choose an opening strategy from this "Open Book"

Named Openings

Move Sequence	Opening Name
C4c3	Diagonal Opening
C4c3D3c5B2	X-square Opening
C4c3D3c5B3	Snake/Peasant
C4c3D3c5B3f3	Lysons
C4c3D3c5B3f4B5b4C6d6F5	Pyramid/Checkerboarding Peasant
C4c3D3c5B4	Heath/Tobidashi
C4c3D3c5B4d2C2f4D6c6F5e6F7	Mimura Variation II
C4c3D3c5B4d2D6	Heath-Bat
C4c3D3c5B4d2E2	Iwasaki Variation
C4c3D3c5B4e3	Heath-Chimney
C4c3D3c5B5	Raccoon Dog
C4c3D3c5B6	Rocket
C4c3D3c5B6c6B5	Hamilton
C4c3D3c5B6e3	Lollipop
C4c3D3c5D6	Cow
C4c3D3c5D6e3	Chimney
C4c3D3c5D6f4B4	Cow Bat/Bat/Cambridge
C4c3D3c5D6f4B4b6B5c6B3	Bat (Piau Continuation 2)
C4c3D3c5D6f4B4b6B5c6F5	Melnikov/Bat (Piau Continuation 1)
C4c3D3c5D6f4B4c6B5b3B6e3C2a4A5a6D2	Bat (Kling Continuation)
C4c3D3c5D6f4B4e3B3	Bat (Kling Alternative)
C4c3D3c5D6f4F5	Rose-v-Toth
C4c3D3c5D6f4F5d2	Tanida
C4c3D3c5D6f4F5d2B5	Aircraft/Feldborg
C4c3D3c5D6f4F5d2G4d7	Sailboat
C4c3D3c5D6f4F5e6C6d7	Maruoka
C4c3D3c5D6f4F5e6F6	Landau
C4c3D3c5F6	Buffalo/Kenichi Variation

2.4 Reinforced Learning

However, this computation is not cleaver. It will just consider the competition as a MDP model. Without advance learning algorithms, it does not have flexibility which can play different types of “tactics”. The agents with highest winning rates we find, is based on Reinforced Learning.

In order to use Reinforcement Learning, we need to give the agent a trainer: another implemented agent. When we treat the opponent’s choice as a negative reword from the environment, we can simply convert this to a model free RL. In this case, Temporal Difference Learning is a good choice.

Function: $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$

V(s) in this case represents the value of choosing this point. The value of choosing a point is considered base on a sequence of important cells and the probability of win. The training step can be divided into two parts: Opponent agent give a feed back, the next list of successors are generated. Random picking up next action to do the training and update V(s).

In this TD learning, we expect this agent can beat the training agent and human finally. The main idea is the flexibility of using different agent as trainers. This picture shows part of the code, and the right picture shows that after 5,000 times learning the remaining errors of the agent.

3. **Our Program:**

Based on these information, we also developed our own program. Our program allows agent vs. agent. We did not set this game towards players since we are focusing on how to improve existed AI. For some reasons, we only built a simple agent that use MDP to make decisions. We separate the Game into two parts: Agents and Game itself. So that it is convenient to edit and build more agents. These would allow other people try something new that can keep finding improvements. There are two parts in the Game itself: State class and Game class. State class has a simple reward method which only used in SimpleMDPAgent in agent.py. Users might want to implement new reward method base on their ideas. New Agents should be add in the agent.py. When Users want to test their new agents, they need to set their new agents into the Game constructor in the main method in board.py.

```
GameBoard:

    O X
      O   X X X
      X X   O
    X X X O O O
        X   O
    O X O O
    X   O
Black: 12 White: 11
GameBoard:
      O
    O X
      O   X X X
      X X   O
    X X X O O O
        X   O
    O X O O
    X   O
Black: 12 White: 12
```

There are five main method in State class in board.py. Method called display(self) print the game board into the terminal. Black color represented as "X" and white color as "O". Method check_win(self) allow the state update some information that will be

printed by display method and check whether the game is end. Method called change_color(self, color, position) is a method that will update the color of chess piece when there is a new position has been chosen with a color as inputs. update method combined methods above to update the game board one time after an agent chose an action(position). Method called get_successors(self, color) will return a set of positions that are valuable for current state of input color. Agents will use this method to get their successors.

Game class only has one method called start(self), it calls two agents to play as a sequence until one has won the match.

4. Conclusion**:**

From our observation and discussion, we thought that what AI's trainning do in Reversi is to find a general solution that has more probabilities to win the game at a specific position. As what we mentioned above, different positions has different importancty. In mdp, those difference are signed as values and saved as a table that represent the board. But this kind of value are human made and not flexible. What a clever AI should do is compute the value of all successors during the game. Since estimate or find out all possible choices in Reversi need a lot of time, AI need to update its value funciton impromptu. This require a trainning durring the game. So if game policy give AI a long time to "consider", it is a great idea to improve Reversi AI.

Reference:

1. Armanto, H., Santoso, J., Giovanni, D., Kurniawan, F., & Yudianto, R. (2012). Evolutionary Neural Network for Othello Game. *Procedia-Social and Behavioral Sciences*, *57*, 419-425.
2. Balduzzi, D., Garnelo, M., Bachrach, Y., Czarnecki, W. M., Perolat, J., Jaderberg, M., & Graepel, T. (2019). Open-ended learning in symmetric zero-sum games. *arXiv preprint arXiv:1901.08106*.
3. edax-reversi(2019); *abulmo*; Retrieved from: https://github.com/abulmo/edax-reversi
4. projectX(2019); *elzo-d*; Retrieved from: https://github.com/elzo-d/projectX
5. reversi-alpha-zero(2018); *mokemokechicken*; Retrieved from: https://github.com/mokemokechicken/reversi-alpha-zero
6. Reversi(2016); *laserwave*; Retrieved from: https://github.com/laserwave/reversi/
7. Reversi(2019); *im0qianqian*; Retrieved from: https://github.com/im0qianqian/Reversi
8. reversi-ai(2018); *Zolomon*; Retrieved from: https://github.com/Zolomon/reversi-ai

9. Reversi(2018); *Arminkz;* Retrieved from https://github.com/arminkz/Reversi
10. Zhang, Z., & Chen, Y. Searching Algorithms in Playing Othello.
Zwer H.(2016) AI Design of Reversi. Retrieved from http://hzwer.com/7865.html