Artificial Intelligence Project Report: Implementing and Tuning Reversed Reversi Agent

Yuwei Xiao
12012902
Department of Computer science and Engineering

***Abstract*—This article is a project report of CS303 Artificial Intelligence. The topic of the project is a Reversed Reversi AI. In this report, I will introduce the Reversed Reversi game, the agent I developed and the tuning process I made. The report will also cover the analysis and experiments of the agent.**

***Index Terms*—Othello, Mini-max, alpha-beta pruning**

I. Introduction

Othello, as known as Reversi, is a strategy board game with history that can date back to 19th Century in England. It has enjoyed a wide popularity around the world for its simple rules and intriguing strategies.

Game playing, as one of the most challenging fields of artificial intelligence has received a lot of attention. Games like Othello, which have proven to fit in well with computer game playing strategies, have spawned a lot of research in this direction.

The primary reason being the small branching factor, allowing the computer to look ahead in abundance, thrashing human intuition and reasoning. As processors increase in speed and complexity, the ability of computers to reason beyond the current state increases, while human intelligence maintains a fairly static average performance over generations. This leads to an extending gap in the Othello-playing ability of humans and computers.

This report will primarily show an implementation of a mini-max algorithm and alpha-beta pruning based Reverse Othello AI, while analysing its performance. Meanwhile exploring how the various Othello heuristics interact.

II. Preliminary

Othello is a turn-based game where the black and white player try to overcome each other in the final domination of an 8x8 board. Players move alternately by placing a new disk in an empty square in order to bracket one or more opponent's disks between the played disk and another of its own color already on the board. The game finished when the board is full or neither of the two players has valid move. And the winner is the one with more disks on board. However, the winner is the one with less disks in our project - Reversed Othello.

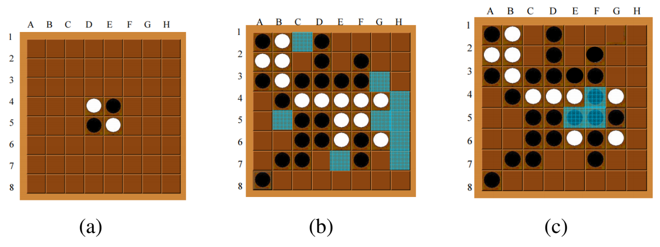

Fig. 1. **Different states in Othello [5].** (a) Board showing the starting game configuration of Othello. (b) Blue highlighted squares show valid moves for black. (c) Blue highlighted squares show the flanked coins once the black has been placed in (G, 5).

A. Problem Formulation

Table 1 list the notations used in formulating the problem and through out the article. Meanwhile, I will use these notations to formulate the goal of Othello agent in algorithm description.

The formulation of the problem: given an chessboard *board*, determine the best move *m*, in order to maximize the possibility of winning the game, with the help of heuristic analysis of the board game, such as *diff*, $board_w$, w_s, w_m and w_f.

That is to say, the problem can be formulated as a tuple $(board, d)$, and the objective of the agent is to maximize v of m_b.

TABLE I
Notations

Notation	Name	Explanation
diff	disc difference	Difference of disk number
d	search depth	Max search depth of alpha-beta search
b	branching factor	Branching factor of alpha-beta search
m	move	Position on board to drop a disc
m_b	best move	Best position on board to drop a disc
board	chessboard	Current state of chessboard
$board_w$	weighted board	Chessboard with weights
v	utility value	The utility value of a move
v_t	temporary value	Temporary value during calculation
v_w	overall weight value	Value of chessboard's overall weight
w_m	mobility weight	Importance of mobility in total utility
w_s	stability weight	Importance of stability in total utility
w_f	frontier weight	Importance of frontier in total utility

III. METHODOLOGY

The design of the agent focus on the implementation of alpha-beta pruning based mini-max search algorithm, and combining several heuristics and optimizations to improve the agent's performance. The workflow of the purposed method will be shown in subsection A. Followed by the details of the algorithm in subsection B. The heuristic functions will be introduced in subsection C. Then, I will introduce my method of utility calculation. And Complexity and Optimality will be discussed in subsection D and E.

A. *General Work Flow*

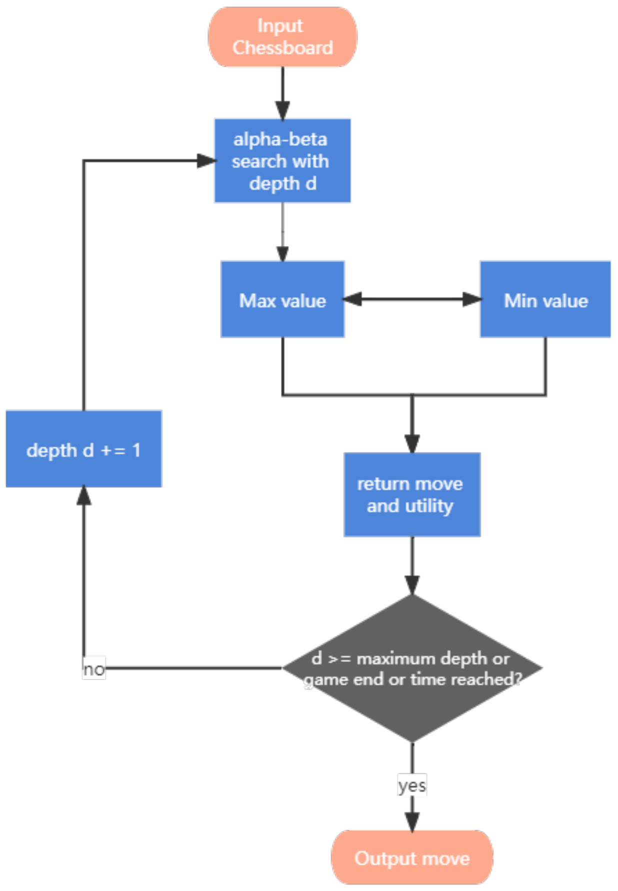

Fig. 2. **General Work Flow of my agent.** The algorithm can be generally divided into 3 parts: a. use iterative depth search to explore the chessboard; b. in each search, use Mini-max algorithm and(or) alpha-beta pruning to explore the actions and find the best move; c. if the depth limit is reached, game ended or time limit is reached, return the utility of current chessboard

B. *Alpha-beta Pruning*

Alpha-beta search [4] is similar to mini-max, except that efficient pruning is done when a branch is rendered useless. Such pruning tends to be rather effective and the search can proceed to great depths, allowing the computer to implement a relatively more powerful look ahead. Pruning is done when it becomes evident that exploring a branch any further will not have an impact on its ancestors.

Algorithm 1 alpha_beta_search(board)

```
Input: The current chessboard, board
Output: The best move according to the search result, $m$
1: $v,\ m = max\_value(board, \alpha, \beta)$;
2: return $m$;
```

Algorithm 2 max_value($board, \alpha, \beta$)

```
Input: The current chessboard, board; The value of the best
    alternative for MAX, $\alpha$; The value of the best alternative
    for MIN, $\beta$;
Output: The best move according to the search result, $m_b$;
    The utility of the move, $v$.
1: if game end or depth limit is reached then
2:     return $None,\ utility(board)$
3: end if
4: $v,\ m_b = -\infty,\ None$
5: for move in valid_moves do
6:     $v_t,\ m = min\_value(board, \alpha, \beta)$
7:     if $v_t > v$ then
8:         $v = v_t,\ m_b = m$
9:     end if
10:    if $v \geq \beta$ then
11:        return $v,\ m_b$
12:    end if
13:    $\alpha = max(\alpha, v)$
14: end for
15: return $v,\ m_b$;
```

Algorithm 3 min_value($board, \alpha, \beta$)

```
Input: The current chessboard, board; The value of the best
    alternative for MAX, $\alpha$; The value of the best alternative
    for MIN, $\beta$;
Output: The best move according to the search result, $m_b$;
    The utility of the move, $v$.
1: if game end or depth limit is reached then
2:     return $None,\ utility(board)$
3: end if
4: $v,\ m_b = +\infty,\ None$
5: for move in valid_moves do
6:     $v_t,\ m = max\_value(board, \alpha, \beta)$
7:     if $v_t < v$ then
8:         $v = v_t,\ m_b = m$
9:     end if
10:    if $v \leq \alpha$ then
11:        return $v,\ m_b$
12:    end if
13:    $\beta = min(\beta, v)$
14: end for
15: return $v,\ m_b$;
```

C. *Heuristics*

In adversarial search, heuristic is an important factor for improving the AI's performance. And the most intuitive way to calculate a heuristic value is to create a linear combination of the quantitative representation of the various important factors. In this Reversed Reversi project, I implemented 4 kinds of heuristic [3] function, namely, **(a)mobility**, **(b)stability**, **(c)frontier discs** and **(d)overall weight**. These heuristics will be introduced in detail and a pseudo-code for utility calculation will be given. Moreover, the evaluation on different heuristics will be discussed as well.

- **Mobility:** Measures how many moves a player has. Restricting opponent's mobility contributes to forcing them choose the bad moves. I consider mobility equals to the number of legal moves. Since it's easy to understand and calculate mobility, its algorithm is omitted for simplicity and neatness.
- **Stability:** Stability is a key factor in Othello. The stability measure of a disk is a quantitative representation of how vulnerable it is to being flanked. Since corners are always stable in nature, and as you build upon corners, more discs become stable in the region. So I consider a disc is stable if it is connected to corners.(Connected means there exists a horizontal/vertical path to edge where all positions on that path are filled with discs of same color.)

Algorithm 4 stability(board, color)

```
Input: The current chessboard, board; The color of the stable discs that you want to calculate, color.
Output: The number of stable discs of the color, stab
1: $stab = 0$
2: for all discs $disc$ with color color on board do
3:     if $disc$ is connected with corner then
4:         $stab = stab + 1$
5:     end if
6: end for
7: return $stab$;
```

- **Frontier** Discs adjacent to empty squares have a greater chance of being flipped by opponent. So a disc is considered at frontier if there is vacant spot around its neighborhood. Therefore, we would like to maximize the number of frontier discs we have.

Algorithm 5 frontier(board, color)

```
Input: The current chessboard, board; The color of the frontier discs that you want to calculate, color.
Output: The number of frontier discs of the color, $f$
1: $f = 0$
2: for all discs $disc$ with color color on board do
3:     if there is an empty spot around $disc$ then
4:         $f = f + 1$
5:     end if
6: end for
7: return $f$;
```

- **Overall weight** An overall weight of different positions on chessboard. Measure the importance of different positions, such as corners and edges.

Algorithm 6 overall_weight(board, color)

```
Input: The current chessboard, board; The chessboard with weights, $board_w$.
Output: The overall weight of the chessboard, $v_w$
1: $v_w = 0$
2: for all discs $disc$ on board do
3:     $v_t$ = the weight of $disc$ on $board_w$
4:     if $disc$ is player's disc then
5:         $v_w = v_w + v_t$
6:     else if $disc$ is opponent's disc then
7:         $v_w = v_w - v_t$
8:     end if
9: end for
10: return $v_w$;
```

D. *Utility Calculation*

Since it's hard to balance the importance of various heuristics, I set up several experiments and study existing work that related to heuristics in Othello. The specific result of these experiments and work will be shown in the next section.

Based on my research, I decide to specify the game process into 4 stages: **(1)Beginning Stage**, **(2)Mid Stage**, **(3)Ending Stage** and **(4)Game End**.

- **Beginning Stage:** Initial stage of the game
- **Mid Stage:** After 1 disc is present on the edges
- **Ending Stage:** At least 2 disc of the same color are present in the corners
- **Game End:** Neither player nor opponent has valid move

In the *Beginning Stage*, it's good to have higher mobility, which gives us more choices of the strategy and to avoid the bad moves that the opponent want us to make.

In the *Mid Stage*, it's better to make the moves with more frontier discs, so that opponent will be more likely to flip our discs. And in this stage, it's not recommended to put discs on edges and corners. In addition, it's essential to capture the *x positions* and *c positions*. *x positions* are positions like (B, 2) in Fig.1.(a). And *c positions* are positions like (B, 1) and (A, 2) in Fig.1.(a). By capturing these positions, we are more likely to force opponent go to the corners.

In the *Ending Stage*, the *diff* and *stable discs* are critical. With fewer *stable discs*, we are more possible to have fewer discs on the chessboard, which determines the result of the game.

In the *Game End*, we can simply return *diff* to show the winner of the game.

Based on these 4 stages, I implement the utility calculation method, which count in different heuristics corresponding to current game stage.

Algorithm 7 utility_calculation(board)

```
Input: The current chessboard, board;
Output: The utility of the game, $v$
1: if game end then
2:     return diff
3: end if
4: $v = overall\_weight(board, player'scolor)$
5: if current stage is Beginning Stage then
6:     $v = v + w_m \times mobility$
7: else if current stage is Mid Stage then
8:     $v = v + w_f \times frontier$
9: else
10:    $v = v + w_s \times stability$
11: end if
12: return $v$;
```

The weights of different heuristics are extremely difficult to decide. After a lot of experimentation against others AI on the platform and a painstaking research of the Othello strategies, I determine the weights of heuristics and $board_w$ as follow:

$$w_m = 5, w_s = 25, w_f = 15$$

TABLE II
CHESSBOARD WITH WEIGHT

-800	260	-50	-30	-30	-50	260	-800
260	100	-10	-5	-5	-10	100	260
-50	-10	-5	3	3	-5	-10	-50
-30	-5	3	1	1	3	-5	-30
-50	-10	-5	3	3	-5	-10	-50
260	100	-10	-5	-5	-10	100	260
-800	260	-50	-30	-30	-50	260	-800

E. Complexity

The time and spacial complexity of alpha-beta search algorithm follows closely to that of **Depth First Search** algorithm. Despite designated for improving the searching efficiency, the pruning method itself does not guarantee any improvement in worst case analysis, because a bad ordering of movement (we will see later, as an speed-up method) will cause a degeneration to MinI-Max algorithm, which is of the same complexity with that of DFS. Hence will prove the complexity of the worst case with average branching factor:

- **Time Complexity Analysis** The average branching factor of the Othello game [1] is
$$b \approx 0$$
, hence for searching depth d, the time complexity approximates to
$$O(10^d)$$
, where d is a hyper-parameter in iterative search part varies from 6 to 10. For **utility calculation**, the time is affected by checking every position on the chessboard, hence the time is
$$O(n^2)$$
where $n = 60$.
- **Space Complexity Analysis** The space complexity of alpha-beta search follows that of DFS, which comes down to
$$O(bd)$$
where $b \approx 10$ and d is the search depth.

F. Optimality

In the start and middle game, the alpha-beta cannot guarantee the optimality of the game search.

As the game proceeds into close-to-end state, where , the agent will perform optimal search thoroughly to the game end and replace the heuristics with the disc difference. Thus the optimality is

$$OPT = \begin{cases} True, & \text{if } a \le d \\ False, & \text{if } a > d \end{cases}$$

, where a represents the number of actions needed to reach game end.

IV. EXPERIMENTS

In this section, performance and efficiency of the algorithm will be tested. For the performance part, we test the heuristic functions by setting up a game environment where AIs play against each other and measure their win rate. For the efficiency test, we test different speed up approaches including **(a) move ordering**, **(b) back-propagation of chessboard** and **(c) Numba acceleration** and compared their running time.

*1) **Environment**:* The main testing environment is Windows 10 Home Edition with AMD Ryzen 7 5800H 3.20GHz. Python edition 3.8.8., Numpy 1.18.0 and Numba 0.53.1.

*2) **Performance Test**:* This section will hold a competition where AIs with different heuristics and hyper-parameters play against each other and their win rate is recorded as measurement of performance. And I also referred to the performance test in other's work[5].

TABLE III
MY PERFORMANCE TEST RESULT

	Mob.	Stab.	Front.	OW	Rate
Mob.	/	L	L	L	0%
Stab.	W	/	W	L	66.7%
Front.	W	L	/	L	33.3%
OW	W	W	W	/	100%

TABLE IV
OTHER WORK'S PERFORMANCE TEST RESULT

	All	Mob.	Stab.	Front.	OW
All	46%	44%	44%	45%	18%
Mob.	56%	50%	46%	71%	31%
Stab.	56%	54%	77%	66%	35%
Front.	55%	29%	34%	44%	19%
OW	82%	69%	65%	81%	48%

*3) **Efficiency Test:*** In this section, a variety of speed-up methods are introduced and evaluated. The results are measured by seconds, indicating the running time of a depth = 6 alpha-beta search in a mid-game chessboard.

- **Move Ordering** In alpha-beta pruning, the ordering of movements is often an essential factor affecting the running time of the algorithm. A good ordering is one where lots of pruning happens. The ordering here is a list of actions sorted by their *overall weights*.
- **Board Back-propagation** To perform an iterative search through the game, it is often needed to make copies from current chessboard. The board back-propagation is a method that can record the flipped discs and manually recover them on iteration exit, saving time and space from copying.
- **Numba** Numba is an open source JIT compiler that translates a subset of Python and Numpy code into fast machine code. It is introduced to the agent for algorithms concerning disc moves and utility calculations.
- **Bare** Original Algorithm which without any acceleration.
- **ALL** All methods combined.

TABLE V
EFFICIENCY TEST RESULT OF 5 ROUNDS

Bare	MO	BP	NB	ALL
9.751	5.513	9.697	2.053	1.891
9.744	5.431	9.753	2.009	1.920
9.739	5.420	9.729	2.017	1.945
9.793	5.470	9.668	2.041	1.917
9.727	5.419	9.881	2.025	1.923

*4) **Summary:*** **From the performance test**, *overall weight* has the highest remark unsurprisingly due to the importance of corner-taking strategy in Othello game. *Stability* also plays an important role in analyzing the mid game. For the *Mobility* and *Frontier discs*, their effects are less significant by comparison.

From the efficiency test result, we can see that **Numba** has brought the most significant improvement over all tested methods, showing the strength of machine code over interpreted language. The **move ordering** also decreased the search time dramatically. The credit will most probably be given to its effect in reducing **branching factor** b. Whilst improvement made by **back-propagation** is not evident. However, certain effects in reducing of memory use is still possible.

V. CONCLUSION

In this article, I proposed an alpha-beta search based Reversed Reversi AI agent, and analysed its complexity, performance, and efficiency.

The experimental results are supportive to my design principles of the algorithm. However, some other work come up with idea such as dynamically changing the weight of different heuristics, machine learning based method and so on. I tried several other methods, but the results didn't turn out better than mine, perhaps because of my bad implementation of those methods.

In the project's contest, the agent ranked at most top 15 at start. However, the agent cannot compare to others' due to poor effect of manually adjusting of heuristics and hyperparameters later in game.

To draw a conclusion of the algorithm, alpha-beta search relies heavily on computational resources and carefully adjusted heuristic parameters, which is easily converted to needs of speed-up methods and better tuning algorithm like **Genetic Programming(GP) [6]**. The pros of the algorithm is that it can perform nearly optimal search in end game and make seemingly reasonable actions with the help of heuristics.

From this project, I learned many new tricks in implementing, tuning and accelerating alpha-beta search, such as simplifying the data structure, improving the reusability of functions and code. And one of the most important tricks is Numba. Besides, I become more familiar with Python and libraries like numpy. Moreover, I learned a lot of strategies of Othello and AI strategies in gaming, which I think will be beneficial to my future career. Last but not least, the relatively boring and complicated process of adjusting the parameters taught me a lesson that you should never be too optimistic about anything, and even a slightly change can make a huge difference on the whole.

REFERENCES

[1] Norelli, A. , Panconesi, A. OLIVAW: Mastering Othello without Human Knowledge, nor a Fortune. Cornell University Library. (2020).

[2] D. Silver, J. Schrittwieser, K. Simonyan, et al., "Mastering the game of Go without human knowledge," Nature, vol. 550, no. 7676, 2017.

[3] B. Rose, Othello -A Minute to Learn. . . A Lifetime to Master, (2005).

[4] Alec Barber, Warren Pretorius, Uzair Qureshi, Dhruv Kumar. An Analysis of Othello AI Strategies. (2020)

[5] Vaishnavi S. , Muthukaruppa A.. An Analysis of Heuristics in Othello.

[6] Jean-Marc Alliot, Nicolas Durand. A Genetic Algorithm to Improve an Othello Program.