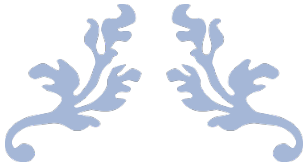

Artificial Intelligence in Othello Game

COMP3270 Mini-Project Report

Abstract

Nowadays, artificial intelligence (AI) has become one of the most animated topics in both the field of academic research and industrial use. This project contributes to a simple exploration of the application of AI technology to Othello game, and it delivers a Python program providing Othello game between the human player and the computer. The artificial agent is implemented based on the minimax search algorithm with alpha-beta pruning. The corresponding experiment has been conducted to examine the performance of the game AI.

December 1, 2017

YAN KAI

UID 3035141231

1. Introduction

Artificial intelligence (AI) has made significant progress in many fields of academic research and real-world applications. Applying AI technologies to traditional chess games brings a lot of attraction and helps people to improve their chess skills. Based on the interest in AI, this project explores the application of artificial intelligence in Othello games, and develops a simple Othello program providing competition between a human and a computer.

Othello is a strategic chess game for two players, played on an 8×8 board. There are sixty-four identical game pieces which are black on the one side and white on the other side. The initial state of the game is shown in Figure 1. Players take turns placing pieces with the side of their own color facing up in the empty squares on the board. During a play, any pieces of the opponent's color that are in a straight line and bounded by the pieces just placed and another disk of the current player's color are reversed to the current player's color. The player with the majority of pieces at the end of the game is the winner.

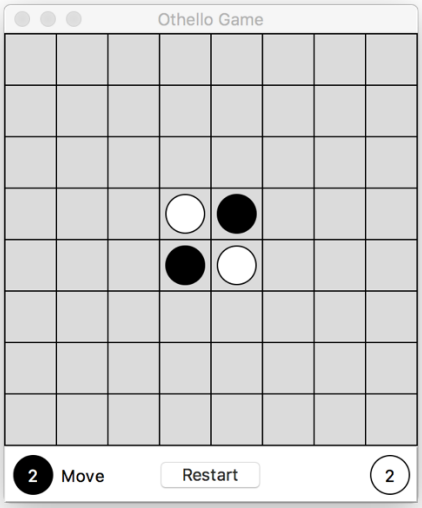

Figure 1. Initial state of an Othello Game

This project makes two contributions. First, it combines AI technology with the theories of Othello games, providing the first-hand experience on the real-world application of AI. Second, this project delivered a lightweight program for beginners to play against the artificial agent in Othello game.

The remainder of this report proceeds as follows. First, the AI algorithms are illuminated, including the minimax search with alpha-beta pruning, the evaluation function embraced by the artificial agent, and the additional strategies improving the search efficiency. Next, the implementation of the Othello game is demonstrated in terms of the basic functions and the

architecture of the application. Then, a brief experiment result is elaborated for the justification of the intelligence of artificial agent. Finally, this report is closed with a conclusion summarizing the design and performance of the AI.

2. Artificial Agent Design

In this section, the design of the AI algorithms is illustrated, including the minimax search with alpha-beta pruning, the evaluation function, and the history table used to facilitate the game tree search.

2.1. Minimax search with alpha-beta pruning

Othello game is a 0-sum game, so that the more a state of the board benefits one player, the more it is in disadvantageous to the other one. The game playing can be treated as a progress that two players take turns to choose the states with maximum benefit to themselves. Every available step allowing players to place a piece generates a new state. Each of these states can be interpreted to a node of a game tree, and the whole game is a travel from the root to the leaves of the tree.

The game tree search algorithm adopted in this project is the minimax search algorithm with alpha-beta pruning. Minimax search requires each node explored should be assigned with a value indicating the benefit to the player, and it attempts to reach the node with the maximum benefit. Additionally, the algorithm assumes that the adversary will also try to maximize its own benefits. In another word, the adversary will always make the rational choice that is most disadvantageous to the player. Therefore, in minimax search algorithm, the value of a node is determined by its children. For a node where the player plays, its value is the minimum of its children, while for a node where the adversary plays, its value is the maximum of its children.

Alpha-beta pruning is an algorithm improving the efficiency of minimax search by pruning the branches that are unnecessary to be explored. Figure 2 shows an example of applying alpha-beta pruning to the game tree. As B is a minimum node, its value should be the minimum value its children have, which is 6. Hence, the node A must have a value that is not smaller than B. The node E has a value 4, then the value of node C will not be larger than 4, which is smaller than the value of B. Therefore, it's unnecessary to continue to search F and G since

the value of their parent C will never be adopted by A. In this situation, the branches of F and G are pruned.

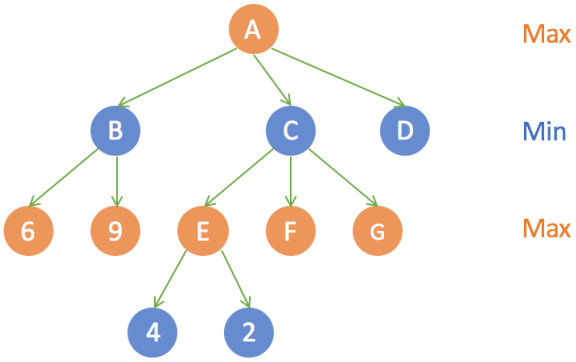

Figure 2. Minimax search tree with alpha-beta pruning

The implementation of this algorithm introduces two values for each node, which are α and β, indicating the floor and ceiling of this node. α and β are dynamically updated during the search progress, and once β ≤ α, the pruning happens. The following is the pseudocode of this algorithm.

```
def alpha_beta(state, depth, -∞, +∞, player):
    if depth == 0 or state is a terminal state:
        return evaluation(state)
    if player plays at this state:
        for child in children of state:
            α = max(α, alpha_beta(child, depth-1, α, β, adversary)
            if β <= α:
                break
        return α
    else:
        for child in children of state:
            β = min(β, alpha_beta(child, depth-1, α, β, player))
            if β <= α:
                break
        return β
```

2.2. Evaluation function

In this project, the evaluation function embraced by the AI mainly consider three factors: the mobility, the number of pieces, and the corner of the board.

The mobility refers to how many moves the current player can choose when facing a certain state. Generally, to acquire the upper hand in an Othello game, the AI should have the ability to limit the mobility of its adversary and maximize its own mobility. Therefore, the ratio

between the available moves of the current player and the adversary is used to evaluate the mobility of the current player in the state. It is demonstrated by the following equation:

$$\text{Mobility}(\text{self}, \text{state}) = \frac{Moves(self, state) + 1}{Moves(adversary, state) + 1}$$

The winning condition of Othello is to occupy more squares than those occupied by the adversary. Hence, the number of pieces is a cardinal factor to be considered when a state is evaluated. In this project, the ratio between the numbers of pieces is used to reflect the imparity between two players. Generally, during the first half of the game process, more pieces means fewer advantages, because the less pieces you have, the less mobility your adversary has. When the game comes to the end, more pieces become indicating a higher dominance. The following equation shows the method to evaluate the number of pieces.

$$\text{Imparity}(\text{self}, \text{state}) = \alpha \frac{Pieces(self)}{Pieces(adversary)} \quad \text{where } \alpha = \begin{cases} 1 & if\ empty\ square < 20 \\ -1 & otherwise \end{cases}$$

Othello is a game where a player tries to flip the pieces owned by the adversary. However, there are some pieces that can never be flipped. In this project, only pieces at four corners are taken into consideration, and "stability" is used to describe this characteristic. The more corners are occupied and are able to be occupied by the player, the more benefit can be gained from this state, and vice versa. This factor is evaluated by the following equation:

$$\begin{aligned} &\text{Stability}(\text{self}, \text{state}) \\ &\quad = \text{Corners}(\text{self}) + \text{PotentialCorners}(\text{self}) - \text{Corners}(\text{adversary}) \\ &\quad - \text{PotentialCorners (adversary)} \end{aligned}$$

Finally, the evaluation function is:

$$\begin{aligned} &\text{Evaluation}(\text{self}, \text{state}) \\ &\quad = (\text{Mobility}(\text{black}, \text{state}) + \text{Imarity}(\text{black}, \text{state}) \\ &\quad + 10\ \text{Stability}(\text{black}, \text{state})) \times \text{Color}(\text{self}) \end{aligned}$$

$$where\ Color(black) = 1\ and\ Color(white) = -1$$

2.3. History table

The experiment reveals that the most time-consuming step is to find all valid moves for the given state, and a lot of states are searched for more than one time. To improve the searching efficiency, a hash table is used to store the states that have been searched, so that for the

second time a state is reached, the AI agent can refer to the table to retrieve all valid moves of the state. The time efficiency becomes O(1). The following pseudocode demonstrates the mechanism of this history table.

```
def find_moves(state, player):
    key = str(state)+str(player)
    if key in move_table:
        return move_table[key]
    moves = []
    for square in board:
        if player can place piece on square:
            moves.append(square)
    move_table[key] = moves
    return moves
```

3. Implementation

In this project, the implementation of the Othello program is based on Python3. The graphical user interface (GUI) is implemented using Tkinter, which is a de-facto standard GUI package for Python. To run this program, users must make sure that Python3 has been correctly installed on their machine.

This program provides 4 modes of the game. The initial panel is shown in Figure 3. Users can choose to hold black or white pieces to play against the computer, or play locally without AI,

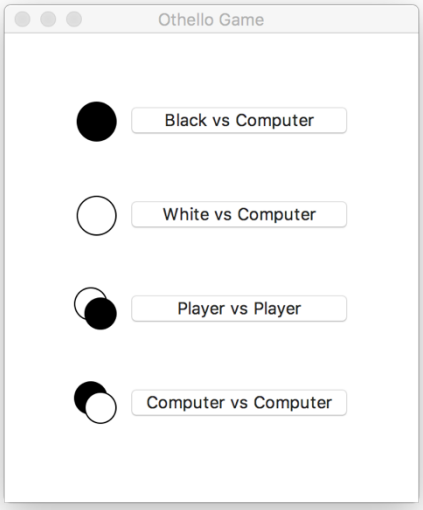

Figure 3. Initial panel of the Othello program.

or simply watch two AI agents play against each other. The game board is shown in Figure 1, at the bottom of which the number of pieces and the current player is displayed. Users place pieces by clicking the corresponding square. Figure 4 shows the end of a game where the white wins.

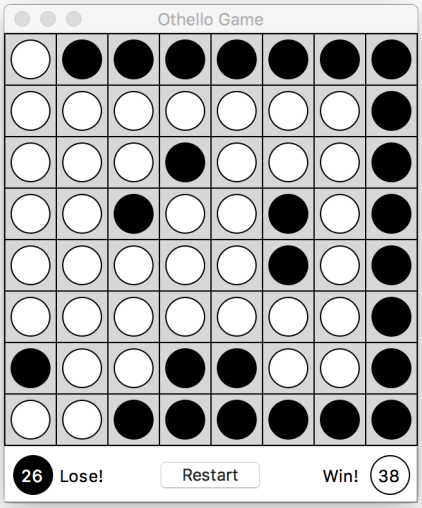

Figure 4. White wins this Othello game.

4. Experiment Results

The experiment is conducted on the machine of which the configuration is shown in the following table:

OS	macOS High Sierra 10.13.1
Python Version	Python 3.6.3
CPU	2.5 GHz Intel Core i7
Memory	16 GB 1600 MHz DDR3

In this project, under the restriction that the AI should make a decision within no more than 1 second on average, the maximum search depth is adjusted to 4. According to the statistic result of the experiment where 2 AI agents play against each other, the AI explores 1063 nodes in 0.297 seconds to make a move on average, with a maximum 2765 nodes 0.763s and a maximum time 0.792s within which 2728 nodes are explored. After the game, the length

of the history table is 47865, and it has been referenced for 31565 times in total, with a utility of 65.9%. Details can be seen in Appendix 1.

Additionally, the writer, who is a beginner of Othello game, has tried to play against the computer using black or white pieces for dozens of times, but unfortunately, never did he win a single game.

5. Conclusion

This project applies several AI technologies to the Othello game, and it delivers a pragmatic Othello application. The major technology adopted in this project to implement the artificial agent is the minimax search with alpha-beta pruning, and a history table is introduced to improve the search efficiency. The project has conducted experiments on the AI designed, which reveals that the AI can perform the game tree search exploring into depth 4 without perceptible decreasing in search speed. It takes around 0.3 seconds to finish the search with the maximum time no more than 1 second. The agent has sufficient intelligence to beat an Othello beginner. The application provides functions that allow users to assign black or white to the artificial agent and play against the computer.

Appendices

Step	Moving player	nodes explored	time per move (s)	time per node (ms)	table referenced
1	Black	129	0.050674	0.393	54
2	White	235	0.098145	0.418	119
3	Black	436	0.174256	0.400	229
4	White	376	0.154250	0.410	171
5	Black	605	0.220105	0.364	291
6	White	931	0.330142	0.355	417
7	Black	895	0.302574	0.338	454
8	White	1207	0.507631	0.421	548
9	Black	1902	0.671159	0.353	737
10	White	2136	0.730862	0.342	872
11	Black	2033	0.719189	0.354	727
12	White	1351	0.449530	0.333	534
13	Black	1817	0.602546	0.332	733
14	White	1474	0.493984	0.335	671
15	Black	1256	0.390119	0.311	580
16	White	1379	0.416369	0.302	769
17	Black	800	0.235262	0.294	431
18	White	1589	0.459280	0.289	934
19	Black	776	0.231250	0.298	436
20	White	1594	0.459373	0.288	882
21	Black	1884	0.556521	0.295	879
22	White	1704	0.470021	0.276	943
23	Black	2728	0.792350	0.290	1042
24	White	2765	0.763189	0.276	1397
25	Black	2491	0.696954	0.280	1013
26	White	2208	0.579512	0.262	1161
27	Black	2127	0.598222	0.281	905
28	White	1366	0.366979	0.269	789
29	Black	1557	0.413853	0.266	613
30	White	1726	0.432417	0.251	915
31	Black	1683	0.408131	0.243	894
32	White	1469	0.341328	0.232	923
33	Black	1348	0.333335	0.247	631
34	White	843	0.196140	0.233	472
35	Black	1104	0.275423	0.249	513
36	White	1573	0.358767	0.228	781

37	Black	1102	0.261132	0.237	589
38	White	1423	0.315826	0.222	668
39	Black	1967	0.433031	0.220	873
40	White	1976	0.425975	0.216	961
41	Black	878	0.177744	0.202	546
42	White	916	0.189994	0.207	516
43	Black	911	0.185988	0.204	527
44	White	651	0.127187	0.195	439
45	Black	250	0.045063	0.180	225
46	White	170	0.028409	0.167	197
47	Black	152	0.029109	0.192	112
48	White	237	0.042107	0.178	193
49	Black	232	0.041467	0.179	160
50	White	348	0.062602	0.180	262
51	Black	315	0.052079	0.165	176
52	White	294	0.047459	0.161	218
53	Black	200	0.034614	0.173	133
54	White	92	0.013177	0.143	91
55	Black	94	0.016531	0.176	86
56	White	52	0.007165	0.138	67
57	Black	15	0.003490	0.233	21
58	White	14	0.003218	0.230	29
59	Black	5	0.001237	0.247	11
60	White	2	0.000422	0.211	5
Total	—	63793	17.824868	—	31565
Average	—	1063.22	0.297081	0.263	526.08
			Length of history table: 47865		

Appendix 1. Statistic results of the experiment where 2 AI agents play against each other