Othello AI

|  |  |  |
| --- | --- | --- |
| 蔡柏毅 0610018 | 黃秉茂 0616098 | 鄭伯俞 0610021 |
|  |  |  |

Github repo link: <https://github.com/abt8601/NCTU-2021spring-AI-final>

# Introduction:

In this project, we create AI agents that play Othello[1] and compare their performances. The main motivation behind playing board games is mainly due to its fun nature. We applied all the adversarial search algorithms taught in class on the game, and let them play against each other. We also implement a GUI on which human players can play against different AI agents.

It is worth more to research the othello game than the pacman game, since the players have to take turns to take the action instead of acting simultaneously. The agents such as minimax and alpha-beta fit the ordering game more.

# Related work:

Othello is a widely researched game, and strong Othello programs play strongly against even the best human players.

As of 1994, all strong Othello programs are based on a deep alpha-beta search with a finely-tuned evaluation function, augmented with a large opening database and endgame search [2]. Construction of evaluation functions has evolved, from purely hand-crafted to partially learned. Optimisations in the search algorithm, like Multi-ProbCut, have been made to improve performance [3]. Other search algorithms, like Principal Variation Search, MTD(f), and NegaC\* have been developed and can be directly applied to playing Othello.

Other methods have been investigated as well. The Monte Carlo method is used to play Othello, and it could win a reasonably decent agent based on alpha-beta pruning when the number of random plays is large enough [4]. The study does not use MCTS, but some later studies have applied MCTS to playing Othello, most notably with learned domain knowledge applied to it. Reinforcement learning-based approaches have been studied as well [5].

# Methodology:

## Game Mechanics

We implement the mechanics of Othello games from scratch, in Python. The rules and game board definition are defined in the “othello.py” file. Also, the “README.md” on the Github repo explains the interface of the game simulator.

## Graphical Interface

We use the tkinter library to implement a GUI that allows human players to play against AI agents. The layout of the GUI is modified from [7]. At the beginning, players can choose the strength of the AI agent, the difficulty is represented by the stars on the button (see Fig 1 ). During the game we implement transition animation and highlight the possible next-move of each player (see Fig 2).

|  |  |
| --- | --- |
|  |  |
| Fig1. Beginning Menu | Fig2. Game Board |

## Random Agent:

Randomly pick a legal action. This agent is defined in “random\_agent.py”

## Max Agent :

Choose the action which can bring you to the maximum payoff. This agent is defined in “max\_agent.py”. This agent outperforms Random, and is extremely fast.

## Minimax Agent :

Suppose the adversary will minimize your score to defeat you, so you have to choose the action with considering the worst case. This agent is defined in “minimax\_agent.py” , the parameter “search\_depth = 2” used in the **Experiments** section.

## Minimax Agent with Alpha Beta Pruning:

Minimax with a pruning mechanism that speeds up the search process by eliminating branches that don't contain better candidate solutions. This agent is defined in “alpha\_beta\_agent.py”, the parameter “search\_depth=2” is used in the **Experiments** section**.**

## Expectimax Agent :

You can predict the behavior of your adversary, so you have to choose the action with considering the average case. This agent is defined in “expectedmax\_agent.py”, the parameter “search\_depth=2” is used in the **Experiments** section**.**

## Monte Carlo Tree Search Agent (MCTS):

MCTS is an algorithm similar to Minimax, except that it grows the game tree asymmetrically and uses random simulation instead of a heuristic evaluation function to determine which move is the best. Since this algorithm does not use a heuristic evaluation function, it can play games without expert knowledge of the game. As a result, it is frequently used in algorithms that aim to achieve general game playing, like that of AlphaZero. This study uses the UCT algorithm [6]. This agent is defined in “mcts\_agent.py”, the parameter ”n\_iters: int =100” in the **Experiments** section.

## Heuristic Evaluation:

Heuristic\_eval\_number:

In the end, you win the game and beat the adversary when your chess pieces are more than your opponent. Thus, the heuristic evaluation is the number of your chess pieces. The more chess pieces you have, the more probability you will win.

Heuristic\_eval\_edge:

The chess pieces on the edge are hard to be attacked, so you have the advantage of the game when you have many chess pieces on the edge. The heuristic evaluation is the number of your chess pieces on the edge. The more chess pieces you have on the edge, the more probability you will win.

Heuristic\_eval\_corner:

The chess pieces at the corner are impossible to be attacked, so you have the advantage of the game when you have many chess pieces at the corner. The heuristic evaluation is the number of your chess pieces at the corner. The more chess pieces you have on the corner, the more chances you will win.

Heuristic\_eval\_n\_action:

The more legal actions you can take, the more choice you have. On the other hand, you lose when you don’t have any legal moves. Therefore, the heuristic evaluation is the number of your legal actions. The more legal actions you have, the more chances you will win.

Heuristic\_eval\_adversary\_n\_action:

The more legal actions your opponent can take, the more choice he has. On the other hand, you win when your opponent doesn't have any legal moves. Therefore, the heuristic evaluation is the number of your opponent's legal actions. The less legal actions he has, the more probability you will win.

Heuristic\_eval\_comprehensive:

Consider each heuristic evaluation above and give them different weights. We decide to grant Heuristic\_eval\_corner the largest weight since the number of chess pieces at the corner is the most important.

# Experiments:

heuristic\_eval\_number

|  |  |  |  |
| --- | --- | --- | --- |
| Agents(Dark/Light) | winner | steps | times |
| random / random | former | 60 | 0.0594 |
| random / max | random | 63 | 0.0598 |
| random / minimax | minimax | 61 | 25.4874 |
| random / alpha-beta | alpha-beta | 44 | 0.1007 |
| random / expectimax | random | 61 | 16.6974 |
| random / mcts | mcts | 61 | 93.3939 |
| max / random | random | 63 | 0.0631 |
| max / max | later | 60 | 0.0689 |
| max / minimax | minimax | 60 | 13.6138 |
| max / alpha-beta | alpha-beta | 60 | 0.0893 |
| max / expectimax | max | 60 | 11.3421 |
| max / mcts | mcts | 62 | 97.1636 |
| minimax / random | minimax | 61 | 17.2466 |
| minimax / max | minimax | 60 | 49.4340 |
| minimax / minimax | former | 60 | 34.2234 |
| minimax / alpha-beta | alpha-beta | 61 | 34.4215 |
| Minimax / expectimax | minimax | 60 | 55.9900 |
| minimax / mcts | mcts | 64 | 120.5967 |
| alpha-beta / random | alpha-beta | 60 | 0.0978 |
| alpha-beta / max | max | 60 | 0.0944 |
| alpha-beta / minimax | minimax | 60 | 15.3207 |
| alpha-beta / alpha-beta | later | 60 | 0.1473 |
| alpha-beta / expectimax | alpha-beta | 60 | 12.8811 |
| alpha-beta / mcts | mcts | 60 | 97.5382 |
| expectimax / random | expectimax | 17 | 4.7969 |
| expectimax / max | draw | 60 | 23.8956 |
| expectimax / minimax | draw | 60 | 47.4639 |
| expectimax / alpha-beta | alpha-beta | 60 | 20.0277 |
| expectimax / expectimax | later | 60 | 24.9304 |
| expectimax / mcts | mcts | 61 | 106.2932 |
| mcts / random | mcts | 62 | 128.5235 |
| mcts / max | mcts | 60 | 127.7216 |
| mcts / minimax | mcts | 63 | 141.8833 |
| mcts / alpha-beta | mcts | 61 | 126.2533 |
| mcts / expectimax | mcts | 61 | 137.1520 |
| mcts / mcts | later | 61 | 218.2032 |

Winner of each pair of Agents

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | random | max | minimax | alpha-beta | expectimax | mcts |
| random | former | random | minimax | alpha-beta | expectimax | mcts |
| max | random | later | minimax | max | draw | mcts |
| minimax | minimax | minimax | former | minimax | draw | mcts |
| alpha-beta | alpha-beta | alpha-beta | alpha-beta | later | alpha-beta | mcts |
| expectimax | random | max | minimax | alpha-beta | later | mcts |
| mcts | mcts | mcts | mcts | mcts | mcts | later |

heuristic\_eval\_comprehensive

|  |  |  |  |
| --- | --- | --- | --- |
| Agents(Dark/Light) | winner | steps | times |
| random / random | random | 63 | 0.0542 |
| random / max | max | 60 | 0.2818 |
| random / minimax | minimax | 60 | 471.3425 |
| random / alpha-beta | alpha-beta | 60 | 0.5351 |
| random / expectimax | expectimax | 46 | 262.9170 |
| random / mcts | mcts | 60 | 94.8821 |
| max / random | max | 60 | 0.5317 |
| max / max | max | 61 | 0.6213 |
| max / minimax | minimax | 60 | 203.5885 |
| max / alpha-beta | max | 61 | 0.4953 |
| max / expectimax | expectimax | 60 | 230.2165 |
| max / mcts | mcts | 60 | 97.2234 |
| minimax / random | random | 60 | 313.8434 |
| minimax / max | minimax | 41 | 97.6054 |
| minimax /minimax | later | 60 | 440.9938 |
| minimax / alpha-beta | minimax | 59 | 168.9505 |
| minimax / expectimax | minimax | 61 | 371.5008 |
| minimax / mcts | mcts | 60 | 356.9488 |
| alpha-beta / random | alpha-beta | 60 | 0.4538 |
| alpha-beta / max | alpha-beta | 61 | 0.5057 |
| alpha-beta / minimax | alpha-beta | 60 | 216.8458 |
| alpha-beta / alpha-beta | later | 61 | 0.5140 |
| alpha-beta / expectimax | expectimax | 60 | 214.8458 |
| alpha-beta / mcts | mcts | 60 | 104.1896 |
| expectimax / random | expectimax | 60 | 332.0203 |
| expectimax / max | expectimax | 60 | 258.2807 |
| expectimax / minimax | expectimax | 60 | 411.8069 |
| expectimax / alpha-beta | expectimax | 60 | 260.3856 |
| expectimax / expectimax | expectimax | 60 | 322.9957 |
| expectimax / mcts | expectimax | 61 | 283.1464 |
| mcts / random | mcts | 60 | 101.9265 |
| mcts / max | mcts | 63 | 96.4052 |
| mcts / minimax | mcts | 64 | 290.1068 |
| mcts / alpha-beta | mcts | 60 | 96.46229 |
| mcts / expectimax | mcts | 61 | 201.4097 |
| mcts / mcts | mcts | 60 | 179.7360 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | random | max | minimax | alpha-beta | expectimax | mcts |
| random | former | max | random | alpha-beta | expectimax | mcts |
| max | max | former | minimax | alpha-beta | expectimax | mcts |
| minimax | minimax | minimax | later | alpha-beta | expectimax | mcts |
| alpha-beta | alpha-beta | max | minimax | later | expectimax | mcts |
| expectimax | expectimax | expectimax | minimax | expectimax | former | mcts |
| mcts | mcts | mcts | mcts | mcts | expectimax | former |

# Conclusion:

1. Comparison result

Heuristic\_eval\_number:

mcts > minimax > alpha-beta > random > max > expectimax

later > former

Heuristic\_eval\_comprehensive:

mcts > expectimax > minimax > alpha-beta > max > random

former > later

Since the random agent isn’t the worst when the heuristic evaluation is Heuristic\_eval\_number, we doubt the heuristic evaluation doesn't work well or even useless. Furthermore, when the same agents fight together, the former should win commonly, however, the situation in Heuristic\_eval\_number doesn’t act like that. Thus, we recognize that only considering the number of the check pieces isn’t good enough.

The result in Heuristic\_eval\_comprehensive is more like what we expected, and you can see the random agent and the later player are at a disadvantage. The mcts agent is the best and the random agent is the worst. The minimax agent performs similarly to alpha-beta agent, however, the former is also the most time-consuming.

We can see that heuristic evaluation takes a great part in the model. It impacts not only the performance but also the efficiency of agents. Thus, choosing the proper agent with the corresponding heuristic evaluation and who goes first are important.

1. Limitations

There are some limitations in this study. Due to the poor performances of the AI agents, the search depth (in the case of minimax-based algorithms) or the number of iterations (in the case of MCTS) is set to a very low value, in order to make the simulation complete in a reasonable amount of time. So the AIs tested are in fact very weak. We could identify two factors that impact the performance. First, the legal move generation routine is fairly slow, using a lot of loops. However, a fast version of this routine requires a lot of bitwise operations that are difficult to get right, and so far no success has been achieved on this front. Second, the CPython interpreter is itself a performance bottleneck, where loops are notoriously slow but are used extensively.

# Future Work:

We want to compare agents and heuristic evaluations based on the same amount of time in the future. Moreover, maybe we’ll create new rules, let the agent think deeper, or apply to deep reinforcement learning.

# References:

[1] Othello game <https://www.wikiwand.com/en/Reversi>

[2] Victor Allis (1994). Searching for Solutions in Games and Artificial Intelligence. PhD Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 90-900748-8-0.

[3] Michael Buro (2003). The Evolution of Strong Othello Programs. Entertainment Computing, pp. 81–88.

[4] J. A. M. Nijssen (2007). Playing othello using monte carlo. Strategies, pp. 1–9.

[5] Nees Jan van Eck and Michael van Wezel (2008). Application of reinforcement learning to the game of Othello. Computers & Operations Research, vol. 35, issue 6, pp. 1999–2017.

[6] Levente Kocsis, Csaba Szepesvári (2006). Bandit based Monte-Carlo Planning. In Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18–22, 2006, Proceedings. Lecture Notes in Computer Science. 4212. Springer. pp. 282–293.

[7] A simple game of othello with an AI opponent from <https://github.com/johnafish/othello>