Game of the Amazon

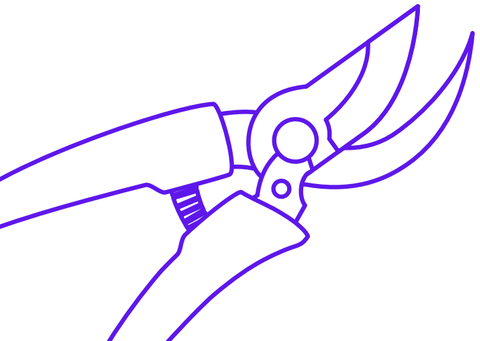
Name: 陳彥佑
Student ID: 112550145
Department: CS, Year 2
Team ID: 52
Professor: Yi-Ting, Chen

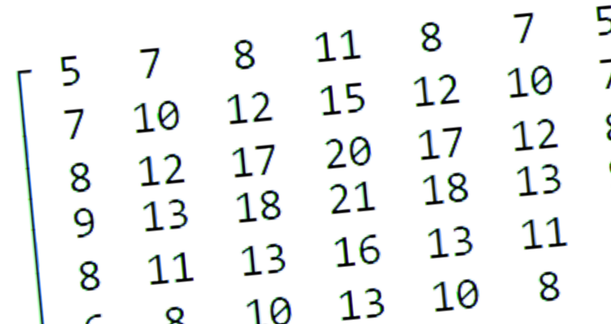
Introduction

1 Inspiration

This project was inspired by HW2: Connect Four. In HW2, I implemented several searching algorithms, including:

- Minimax
- Alpha-beta pruning
- Heuristic Functions based on position-weight matrix

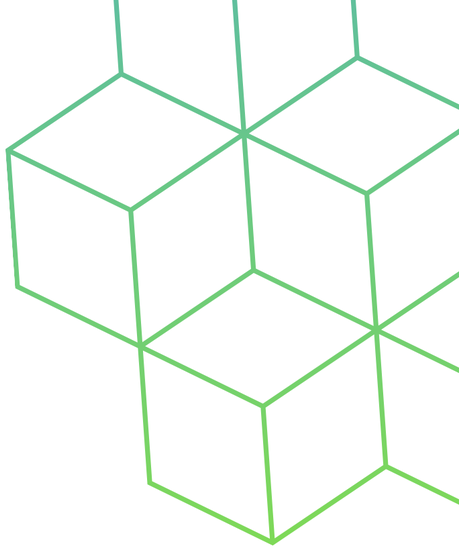




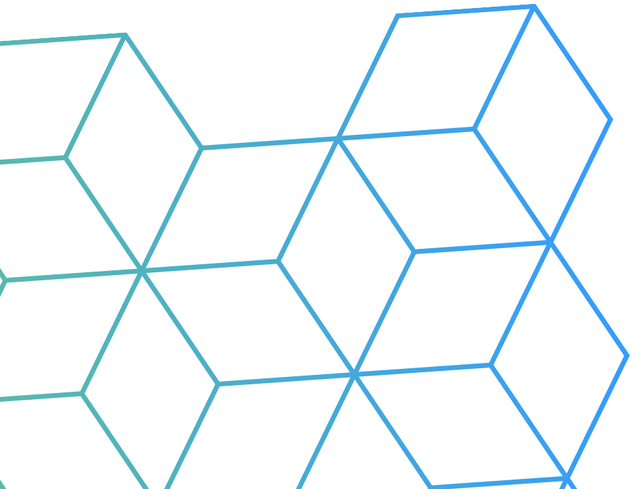
2 What's the difference?

In this project, I have configured four agents as follows:

- Random_agent: Randomly move and do attacks.
- Heuristic_agent: Use heuristic function to evaluate and take the best action.
- MCTS_agent: Implement MCTS search.
- Iterative_agent: Implement IDS search.



Platform



1 Board initialization

In this project, when the board is initialized, all amazons are placed symmetrically. With margin considered in, a list that contains all candidate tuples (x , y, opp_x, opp_y) is generated, and one of them is selected randomly to determine where the amazons should be placed.

Baseline

1 Random_agent

Random_agent will call functions to retrieve all possible moves and arrow targets. Then, it selects an amazon, a move destination, and an arrow target randomly.

```
<Pseudocode>
class RandomAgent:
    def __init__(self, agent_id: int):
        self.agent_id = agent_id    # Assign the agent_id

    def get_action(self, board: Board):
        pos_list  --- retrieve the position list
        from_pos  --- from pos_list randomly choose which amazon to move
        to_pos    --- from from_pos randomly choose the destination
        arrow_pos --- from to_pos randomly choose where to do attack
        return from_pos, to_pos, arrow_pos
```

2 Heuristic_agent

The heuristic evaluation combines three components: territory estimation, mobility estimation, and center-based weighting.

```
<Pseudocode>
class HeuristicAgent:
    def board_eval(self, agent_id: int):
        mob_eval, terr_eval --- find the mobility and territory estimation score
        centrality          --- find the center-weighted estimation score
        return c_1 * mob_eval + c_2 * terr_eval + c_3 * centrality
        (Currently, c_1 = c_2 = Ø.4, c_3 = Ø.2.)
```

2 Heuristic_agent

Heuristic_agent also implements alpha-beta pruning, doing board evaluation mainly based on it.

```
<Pseudocode>
class HeuristicAgent:
    def alpha_beta(**kwargs, depth, maximizing: bool) -> float:
        if depth == Ø: return self.board_eval(board)
        if maximizing:
            max_eval = -inf
            for valid_action:
                eval = self.alpha_beta(**kwargs, depth - 1, False)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if is_time_to_stop: return max_eval
            return max_eval
        else:
            ...
```

2 Heuristic_agent

After searching, Heuristic_agent chooses the best action based on the results.

```
<Pseudocode>
class HeuristicAgent:
    def __init__(self, **kwargs):
        Assign parameters to the self.
    def get_action(self, board: Board):
        best_score = -inf
        best_action = None
        for valid_action:
            score = self.alpha_beta(**kwargs, depth = self.max_depth - 1, False)
            if score > best_score:
                best_score = score
                best_action --- retrieve position
        if best_action: return best_action
```

Main Approach

1 MCTS_agent

Before the agent takes any action, the search tree must be initialized by creating nodes.

```
<Pseudocode>
class MCTSNode:
    def __init__(**kwargs):
        Assign parameters and create variables.
    def get_all_actions(self):
        return a list consists of all valid actions.
    def expand(self):
        Choose an unexpanded action and create child_node.
    def best_child(self, c_param: float):
        Use formula to find the upper confidence bound (UCB) and the best node.
    def backpropagate(self, result: int):
        Update the data. If a player wins, result += 1; if tie, result += Ø.5.
        Also, visit += 1 unconditionally.
```

1 MCTS_agent

Now, MCTS_agent can start working. First, it simulates the game based on the mobility and territory heuristics, instead of playing until the end.

```
<Pseudocode>
class MCTSAgent:
    def simulate(self, board: Board) -> int:
        my_pos, opp_pos   --- retrieve the position lists of two players
        my_mob, opp_mob   --- find the sum of all valid moves in each list
        my_terr, opp_terr --- do territory estimation, same as Heuristic_agent
        mob_eval = my_mob - opp_mob
        terr_eval = my_terr - opp_terr
        score = mob_eval + Ø.5 * terr_eval
        return winner_id based on score (1 / 2 / Ø for draw)
```

1 MCTS_agent

Then, based on the simulation, agent can take actions under the given time limit.

```
<Pseudocode>
class MCTSAgent:
    def __init__(self, agent_id: int, time_limit: float = 1.0):
        Assign parameters.
    def get_action(self, board: Board):
        Initialize root node.
        while is_time_enough:
            Selection: walk down tree via best_child()
            Expansion: if not terminal, expand node
            Simulation: run simulate(node.board)
            Backpropagation: propagate result up the tree
```

2 Iterative_agent

Iterative_agent implements IDS search, which performs depth-limited search iteratively, increasing depth each time

```
<Pseudocode>
class IterativeAgent:
    def search_at_depth(self, board: Broad, depth: int):
        best_score = -inf
        best_action = None
        for valid_action:
            if not is_time_enough:
                raise TimeoutError
            score = self.minimax(board, depth - 1, False)
            best_action = Tuple(valid_action)
        return best_action
```

2 Iterative_agent

Iterative_agent also implements minimax search, doing board evaluation mainly based on it.

```
<Pseudocode>
class IterativeAgent:
    def minimax(**kwargs, depth, maximizing: bool) -> float:
        if depth == Ø: return self.board_eval(board)
        if maximizing:
            best = -inf
            for valid_action:
                score = self.minimax(**kwargs, depth - 1, False)
                best = max(best, score)
            return max_eval
        else:
            ...
```

2 Iterative_agent

Then, based on the result, agent can take actions under the given

time limit.

```
<Pseudocode>
class IterativeAgent:
    def __init__(self, agent_id: int, time_limit: float, max_depth: int):
        Assign parameters.
    def get_action(self, board: Board):
        best_action = None
        for depth in range (1, self.max_depth + 1):
            if is_time_enough:
                best_action = self.search_at_depth(board, depth)
        return best_action
```

Evaluation Metric

1 Quantitative metric: Win rate

In this project, I will compare two types of agents—baseline and main approach—by having them play against each other. The win rate is used as a quantitative metric to determine which agent performs better.

2

Quantitative metric: Time consumption

I also evaluate the agents based on their average time consumption during the game. Longer decision times typically indicate higher computational cost and lower efficiency.

Result & Analysis

1 Controlled variables

Across the four agents, the simulation will be conducted under the condition that the following variables are controlled:

- Board size: 15
- time_limit: 2.0
- max_depth: 4

1 Independent variable

The only independent variable in the simulation is the number of the games. In this project, the number of the games will be set to 50, 100, and 500. For each setting, I repeat the simulation five times and take the arithmetic mean of the results, including the win rate and time consumption.

2 Result

The following four pages show the result of four groups of matches. The results may slightly differ from the ones on E3 platform since there are randomness in the simulations, and they are calculated based on the data attached in the Appendix 3.

2 Result: Random vs. MCTS

Game = 50:

- Win rate (Random) = 35.2%
- Win rate (MCTS) = 64.8%
- Time consumption = 30.2s

Game = 100:

- Win rate (Random) = 39.8%
- Win rate (MCTS) = 60.2%
- Time consumption = 68.8s

Game = 500:

- Win rate (Random) = 36.96%
- Win rate (MCTS) = 63.04%
- Time consumption = 337.4s

2 Result: Random vs. Iterative

Game = 50:

- Win rate (Random) = 56.8%
- Win rate (Iterative) = 43.2%
- Time consumption = 0s

Game = 100:

- Win rate (Random) = 61%
- Win rate (Iterative) = 39%
- Time consumption = 0s

Game = 500:

- Win rate (Random) = 60.88%
- Win rate (Iterative) = 39.12%
- Time consumption = 0s

2 Result: Heuristic vs. MCTS

Game = 50:

- Win rate (Heuristic) = 30%
- Win rate (MCTS) = 70%
- Time consumption = 74.4s

Game = 100:

- Win rate (Heuristic) = 31.2%
- Win rate (MCTS) = 68.8%
- Time consumption = 142.6s

Game = 500:

- Win rate (Heuristic) = 33.48%
- Win rate (MCTS) = 66.52%
- Time consumption = 702.2s

2 Result: Heuristic vs. Iterative

Game = 50:

- Win rate (Heuristic) = 60.8%
- Win rate (Iterative) = 39.2%
- Time consumption = 22.6s

Game = 100:

- Win rate (Heuristic) = 57.4%
- Win rate (Iterative) = 42.6%
- Time consumption = 48.6s

Game = 500:

- Win rate (Heuristic) = 55.76%
- Win rate (Iterative) = 44.24%
- Time consumption = 258.8s

3 Analysis

According to the results, it is evident that MCTS search is more effective than the baseline agent. However, the iterative agent performs poorly. I believe this is because the time limit I set was too short for it to complete a deep enough search. On the other hand, MCTS also incurs significantly higher time consumption compared to the iterative agent. Therefore, despite its better performance, MCTS may still be inappropriate for low-resource or real-time environments.

Appendix

1 Amazon Game: Introduction

The Amazon Game is like a chess game. There are two players. Each of them has several “amazons”. Players should alternately move their amazons and do attack. The first player unable to move or do attack loses the game.

2 Amazon Game: Board configuration

When setting up a board, the following rules must be satisfied:

- Given an integer N as the size of the board. There should be N*N grids on the board.
- Given an integer M as the number of amazons each player possesses. Amazons should be put in the grids. Currently, we put amazons in symmetrical positions.
- Currently, in this project, N = 15 and M = 4.

2 Amazon Game: Grid State

There are four states for each grid:

- Empty: This grid can be moved onto, traversed, and attacked.
- Occupied: When an amazon lies on a grid, the grid is occupied. It cannot be moved onto, traversed, and attacked.
- Temporarily blocked: When an amazon moves to other grid, the grid that it lies on before moving will be temporarily blocked. After 5 moves, the grid will turn back to empty.
- Blocked: When a grid is shot by an arrow, it is destroyed, and then permanently blocked.

2 Amazon Game: Rules for actions

The amazons move like the “queen” in the Chess game. They can move on a horizontal, vertical, or diagonal straight line. However, it is not allowed to move onto and traverse the blocked grids.

As for attacking, the amazon attacks after moving, and it shoots arrow like moving – on a horizontal, vertical, or diagonal straight line. The arrow cannot shoot and traverse the blocked grids.

3 Simulation data: Random vs. MCTS

Group 1: Random vs. MCTS									
Simulations			01	02	03	04	05	Win	Average
Games	50	Random	22	17	16	17	16	88	35.20%
		MCTS	28	33	34	33	34	162	64.80%
		Time	31	28	32	32	28		30.2
	100	Random	37	43	39	42	38	199	39.80%
		MCTS	63	57	61	58	62	301	60.20%
		Time	64	72	72	67	69		68.8
	500	Random	179	188	197	179	181	924	36.96%
		MCTS	321	312	303	321	319	1576	63.04%
		Time	320	340	336	340	351		337.4

3 Simulation data: Random vs. Iterative

Group 2: Random vs. Iterative									
Simulations			01	02	03	04	05	Win	Average
Games	50	Random	31	26	26	31	28	142	56.80%
		Iterative	19	24	24	19	22	108	43.20%
		Time	0	0	0	0	0		0
	100	Random	60	55	51	65	74	305	61.00%
		Iterative	40	45	49	35	26	195	39.00%
		Time	0	0	0	0	0		0
	500	Random	300	300	308	309	305	1522	60.88%
		Iterative	200	200	192	191	195	978	39.12%
		Time	0	0	0	0	0		0

3 Simulation data: Heuristic vs. MCTS

Group 3: Heuristic vs. MCTS									
Simulations			01	02	03	04	05	Win	Average
Games	50	Heuristic	11	14	22	10	18	75	30.00%
		MCTS	39	36	28	40	32	175	70.00%
		Time	74	72	72	80	74		74.4
	100	Heuristic	32	28	39	27	30	156	31.20%
		MCTS	68	72	61	73	70	344	68.80%
		Time	140	143	144	145	141		142.6
	500	Heuristic	168	172	171	158	168	837	33.48%
		MCTS	332	328	329	342	332	1663	66.52%
		Time	713	704	664	702	728		702.2

3 Simulation data: Heuristic vs. Iterative

Group 4: Heuristic vs. Iterative									
Simulations			01	02	03	04	05	Win	Average
Games	50	Heuristic	28	34	33	27	30	152	60.80%
		Iterative	22	16	17	23	20	98	39.20%
		Time	24	19	21	28	21		22.6
	100	Heuristic	53	51	62	58	63	287	57.40%
		Iterative	47	49	38	42	37	213	42.60%
		Time	53	56	44	48	42		48.6
	500	Heuristic	275	288	287	276	268	1394	55.76%
		Iterative	225	212	213	224	232	1106	44.24%
		Time	255	249	251	266	273		258.8