Assignment II: GamePlaying AI

Introduction

This assignment involves implementing a Game Playing AI agent. The goal is to learn how to make decisions in domains with a large space of possibilities while also taking into account the actions of the other agents in a time bound manner.

The setting we will consider is a game called Havannah (described below). This is a two player moves in which each player tries to create certain structures on a hexagonal board. The decision-making task is to determine what is a “good” move providing you higher chances of succeeding. While deciding the next move you need to consider the state (of the board and what actions the other player has taken) while considering a time budget.

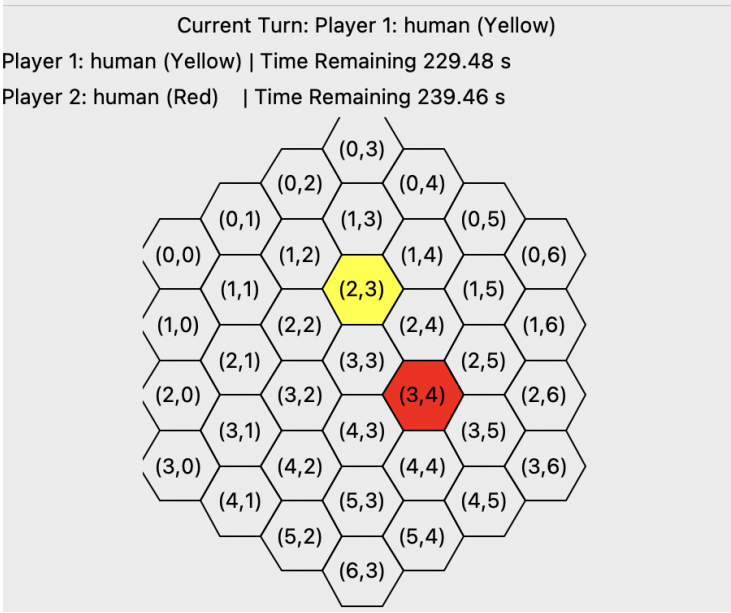
Developing an algorithm for this decision making task requires both thought and iterative implementation. Hence, it is recommended to engage with the assignment early.

Problem Statement

Havannah

Havannah is a board game played on a hexagonal grid of varying sizes. Two players, in alternate turns, place their coloured pieces on the hexagonal cells of the board, aiming to achieve one of the three winning conditions (explained below) with an unbroken chain of their pieces. The game continues until one player successfully meets one of these conditions and wins the game.

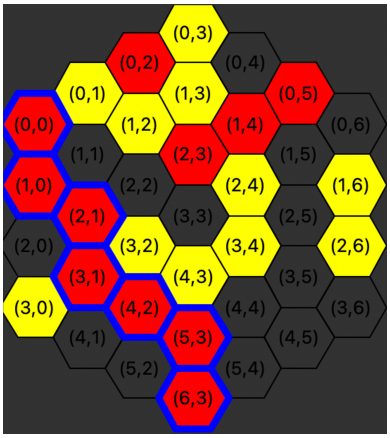
A board looks like the following, and in this state, players 1 and 2 have kept their pieces at (2,3) and (3,4) respectively, and player 1 is next to move :



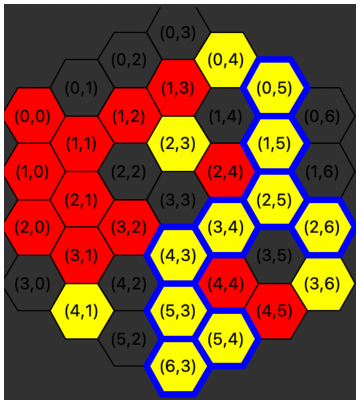
Winning Criteria

A win is declared when a player is successful in making any of the following three structures described below :

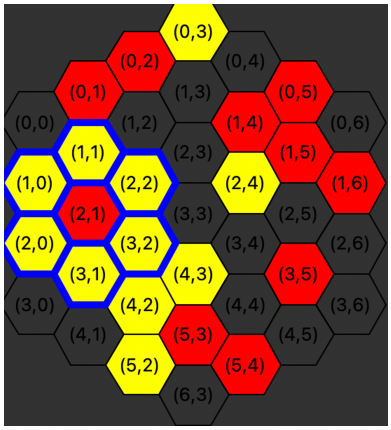
Bridge



Fork



Ring



- A **Fork** in Havannah occurs when three (or more) different edges of the board are connected through a path of adjacent, same-coloured cells. A cell is considered part of an edge if it lies on the boundary of the board but is not a corner. Importantly, corner cells are not part of any edge. The figure illustrates the yellow player creating a fork by connecting the edge cells at (0,5), (2,6),

and (5,4). In this example, the yellow player successfully connects three edge cells on different edges through a continuous path.

- A **Bridge** is a path (a sequence of adjacent, same-coloured cells) that connects any two corners of the board. Note that there are 6 corners on any given board. The figure above illustrates an example where the red player creates a bridge connecting the corners at coordinates (6,3) and (0,0). It is important to note that (1,0) is not a corner but an edge.
- A **Ring** is defined as a closed, cyclical path of same-coloured cells which must enclose one or more cells inside it. Note that the cell(s) inside the enclosure can be either the player's or the opponent's or an empty block.

End of the game

The game ends when one of the players is able to complete any of the three structures or the time budget for one of the players is exhausted. In the latter case, the agent whose time budget is not exhausted is considered the winner.

A win will receive 1 point, a loss receives 0 points. In the case of a draw, a partial credit (< 1) is awarded based on the time remaining relative to the opponent agent's time remaining

Across all our evaluations, this scheme will award higher marks to agents that give winning results faster.

Starter Code

Code Package and Environment Setup

- The starter code is available on Moodle. The downloaded zip has the following structure. Note that your implementation will only be modifying the files listed in red below. Do not modify any other file(s).

```
A2
├── game.py              ## Game engine - runs the loop/renders/
├── helper.py            ## Functions for traversing board/win c
├── initial_states       ## In case you need to load custom layou
```

```
│   ├── size* .txt          ## standard sample board representat
│   ├── custom.txt          ## (optional) if the need be, create
├── players             ## Implementations for agents are housed
│   ├── ai.py               ## Your implementation comes here, t
│   ├── ai2.py              ## To play ai vs ai games
│   ├── human.py            ## Input either via terminal or by c
│   └── random.py           ## Randomly selects one of the valid
└── readme.md           ## Instruction
```

- Please setup a **conda environment** with the allowed dependencies for the assignment with the commands provided below. First time conda users can download and install *Miniconda*.

```
cd A2
conda create -n aia2 python=3.10 numpy tk
conda activate aia2
```

Interaction with Simulator

The game between two players `<player1>` and `<player2>` can be initiated by the following command.

```
python game.py <player1> <player2> [--flag_field flag_value]*
```

Field	Description	Possible values	Default	Required argument?
player1, player2	Agent that plays as player 1 and player 2 (required)	[”ai”, “ai2”, “random”, “human”]	-	Yes
time	Time Budget in seconds, per	positive integer	240	Yes

Field	Description	Possible values	Default	Required argument?
	player, for the entire game			
dim	Size of the board	positive integer $\in$ [4, 10]	4	optional
mode	Render the GUI or not. If you are working with server that doesn't have a display, use "server" mode	["gui", "server"]	"gui"	optional
start_file	Use custom layout through text file (optional)	filename, and file populated with the custom layout	None	optional

Self-play (human vs human): You can understand the game rules better by playing a few games in human vs human mode. The inputs can be passed either via terminal (eg : "0 , 5") or just clicking on your desired position. You can initiate a game between two human agents on board dim 6, and a game time of 10 minutes as follows:

```
## human vs human
python game.py human human --dim 6 --time 600
```

Playing with a random agent: If you want to test your AI agent with a random agent, then you can initiate a game as follows

```
## AI vs Random
python game.py ai random --dim 5 --time 1000
```

Comparing two AI agents: You might also compare two of your implementations in `ai.py` and `ai2.py` , starting from a custom layout `initial_states/custom_layout.txt` as follows:

```
## AI vs AI
python game.py ai ai2 --start_file custom_layout.txt
```

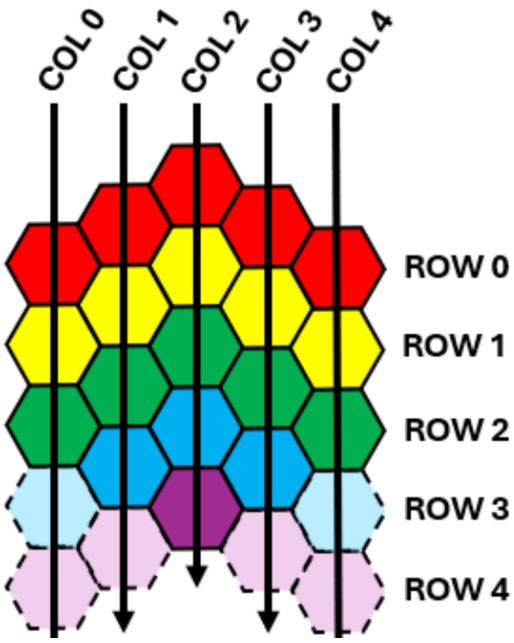
Implementation Guidelines

Board representation

A board of size N is represented as a numpy array of size $(2N-1, 2N-1)$. The entries in the numpy array can be one of the following:

- 0 : represents an unfilled location
- 1 : represents a location filled by player 1
- 2 : represents a location filled by player 2
- 3 : Dummy array entries that are not part of the board. These locations are considered invalid (dashed boundaries below), and is not to be filled

The state of the board is represented using the following coordinate frame :



The topmost layer of cells (highlighted in red on the left) constitutes Row 0.

For an N-sized board:

- The cells of the first row are $(0,0),(0,1)\ldots..(0,2N-1)$
- The 6 corners of the board, in clockwise order from the top-left corner, are : $(0,0)$, $(0,N-1)$, $(0,2N-1)$, $(N-1,2N-1)$, $(2N-1,N-1)$, $(N-1,0)$
- Rows N and onwards are only partially playable. These row may have dummy array elements that are not part of the board. The invalid locations are stored as 3 in the board, and pieces are not to be kept here.

Other Utility Functions

A few utility functions have been provided in `havannah/helper.py` . You may use these functions in your implementation.

1. `fetch_remaining_time(timer, player_num)` : Returns the remaining time for the player `player_num`
2. `get_valid_actions(board)` : Returns all the valid and unfilled positions in the provided state `board`
3. `get_neighbours(dim, pos)` : Returns all the neighbours around a given position `pos` for a `dim` sized board.
4. `get_all_corners(dim)` : Returns all the corner positions for a `dim` sized board.
5. `get_all_edges(dim)` : Returns all the edge positions for a `dim` sized board.

6. `get_vertices_on_edge(edge, dim)` : Returns the positions on an edge of the board for a particular `edge`, and a size `dim` board
7. `get_vertex_at_corner(corner, dim)` : Returns the position on of a `corner` of the board , and a size `dim` board.
8. `get_edge(pos, dim)` : Returns the edge on which a position `pos` lies for a size `dim` board, returns -1 if not an edge piece.
9. `get_corner(pos, dim)` : Returns the corner on which a position `pos` lies for a size `dim` board, returns -1 if not a corner piece.

Implementation Task

- In this assignment, your task is to implement the `AIPlayer` class in `ai.py` . All your code must be confined to this single file.
- Your `AIPlayer` class must implement the method `get_move(state)`, following the provided signature in the code. This method takes in a numpy array representing the current board state and should return the best position for placing your piece as an (i, j) integer tuple. Note that if the agent selects a position that is already occupied or an illegal position (denoted by an entry of 3 on the board), your turn will be skipped.
- You can use the helper function with the arguments `fetch_remaining_time(self.timer, self.player_number)` to find out the time budget remaining for your agent. Your agent must take into account the time left and plan accordingly.
- Please do not use multi-threading to accelerate your code, since the focus of this assignment is the core algorithm, and not parallelisation methods.
- Please Implement a search-based algorithm to find the best move at each step. You must focus exclusively on search-based methods, as discussed in class, or their extensions and adaptations. Avoid using any machine-learning techniques beyond search-based approaches. If you are unsure about your strategy, please raise a query on piazza.

General Suggestions

- Please formulate a model for the game in terms of its state space, actions possible and how the game terminates. Please analyse the branching factor and the guess the depth of the game tree. Play out the game your self (human option) with a reasonable time budget to get a sense of how you intuitively decide a move. Think of what aspects of the board state your consider and if you do any reasoning based on the moves of the other agent.
- Next, please consider the time aspect of deciding the next move. Think how you would speed you decision making. Typically, it is possible to implement a simple and effective agent that can produce its next move in about 5 seconds. While the budget allotted to you would be more, you could use this datapoint to get a general sense.
- Please refer to the material covered in class. Each class of algorithms - MiniMax/ExpectiMax/MCTS etc. - possess their own strengths and weaknesses. Your solution may incorporate elements of each. The use heuristics/state evaluation functions are likely to help in determining moves within a time budget.

Evaluation

The performance of your game-playing agent will be assessed in three stages: Preliminary, Group, and Finals, associated with 40, 30, and 30 points, respectively. The stages are described below.

Preliminary Stage

- Your agent will play two games with a random agent (i.e., a program that simply proposes moves at random). Your agent will play as yellow in the first game and as red in the second game. The games will take place on a board of **size 4.** Your agent and the opponent will get equal time to make their moves. Your agent will be provided a total duration of 3 minutes.
- Your agent will then play two games with a basic TA (Teaching Assistant) agent. The games will take place on a board of **size 4.** Your agent and the opponent will get equal time to make their moves. Your agent will be provided with a total duration of **6 minutes**.

- The performance in the preliminary stage will be used to provide a ranking of the submissions, as determined below. The rankings from this phase will be used to determine the seeding for the subsequent stages.

Criteria	Description
Number of Games Won	The primary ranking criterion will be the total number of games your agent wins.
Total Time Remaining	In case of a tie, agents will be further ranked based on the total time remaining across all four games.

- Points at the preliminary stage will be awarded as follows:

Scoring (out of 40)	Description
+10 points	for each game won by your agent.
+5 points	if your agent ties in the game
+0 points	for each game lost by your agent.

Group Stage

- Each agent will be assigned to one of the 16 groups based on their ranks from Phase 1, with the group assignment determined by the formula: group $g = ((r-1)\%16) + 1$, where r is the rank.
- Within each group, all member agents will play with each other, with each match consisting of two games similar to the previous stage. The games will be played on a board of **size 6**, and each player will have a total of **8 minutes** to make their moves.
- After all matches are completed, agents within each group will be ranked based on the number of wins and total time remaining. The top two agents from each group will advance to the next stage.

Scoring (out of 30)	Your submission will be awarded at most 30 points based on your standings in your group.

Final Stage

- The competition follows a tree-structured format, where submissions will progress up the ladder/levels by winning their matches. The fixtures will be released once group stages are completed.
- Each match will consist of two games: one game played as the red player and the other as the yellow player. The games will be conducted on a board of **size 6**, with each player allotted 10 **minutes**. The submission that does not win will not advance further.
- Winning criteria is the same as the group stage. If a match ends in a tie, a new match will be initiated with **5 minutes** allotted to each player. Should the tie persist, the group stage performance will be used as a tiebreaker. If the tie remains unresolved, a coin toss will finally decide the winner.

Submission Instructions

- **This assignment is to be done individually or in pairs. The choice is entirely yours.**
- **The assignment is intended for both COL333 and COL671 students.** If the assignment is being done in pairs then the partner is to be selected only within your class. That is COL333 students should be paired within COL333 and similarly masters students in COL671 should pair within their class of COL671.
- Please submit only single algorithm (your best algorithm). Please succinctly describe the core ideas of your algorithm in the text file called **report.txt** (upto 2 pages in standard 11 point Arial font). The first line of the report should list the name and entry number of student(s) who submit the assignment.
- The assignment is to be submitted on **Gradescope**. Exactly ONE of the team members needs to make the submission. All registered students have already been added to Gradescope. The details are given below.
    - You need to upload a single ZIP file named `"submission.zip"`. Upon unzipping, this zip **must** create a folder containing `ai.py`, `report.txt` and `group.txt`. Do not rename the zip file.

- The group.txt should include the entry numbers of all the group members (one per line). If you are working alone, simply include your own entry number. A sample `group.txt` is given below.

  ```
  2020CSZ1234
  2020CSZ5678
  ```

- Gradescope will run a preliminary check for your submission. It will flag issues such as an incorrect Agent class specification, use of disallowed imports and will further run your code with dummy inputs. Your submission must pass these preliminary test. Submissions failing the preliminary check will be excluded from evaluation.
- If you are working in pairs, you should select the partner using the "**Group Members**" option after uploading the submission in Gradescope.

- **Please submit the assignment by 6 PM on Thursday, 3rd October, 2024.**
- This assignment will carry $14(\pm 3)\%$ of the grade.

Other Guidelines

- Late submission deduction of (10% per day) will be awarded. Late submissions will be accepted till 2 days after the submission deadline. There are no buffer days. Hence, please submit by the submission date.
- Please follow the assignment guidelines. Failure to do so would cause exclusion from assessment and award of a reduction. Please strictly **adhere** to the input/output syntax specified in the assignment as the submissions are processed using scripts. Please do not modify files beyond the files you are supposed to modify. Failure to do so would cause exclusion from assessment/reduction.
- Queries (if any) should be raised on **Piazza**. Please keep track of Piazza posts. Any updates to the assignment statement will be discussed over Piazza. Please do not use email or message on Teams for assignment queries.
- **Please only submit work from your own efforts.** Do not look at or refer to code written by anyone else. You may discuss the problem, however the code

implementation must be original. Discussion will not be grounds to justify software plagiarism. Please do not copy existing assignment solutions from the internet or taken from submissions of related problems from previous years or submissions from other students; your submission will be compared against them using plagiarism detection software. Submissions violating the honor code will be excluded from evaluation for fairness to other students. Violations will also result in deductions as per course and institute policies.