School of Computing Sciences

Module: CMP-6058A/7058A Artificial Intelligence

Assignment 001: AI Game Hinger

Set by: Dr. Taoyang Wu (taoyang.wu@uea.ac.uk)
Date set: Monday 29 September 2025
Value: 40%

Date due: 24th October 2025 15:00 [week 5]
Returned by: 28th November 2025
Submission: Blackboard
Checked by: Dr. Daniel Paredes-Soto (d.paredes-soto@uea.ac.uk)

Learning outcomes

- Improve your understanding of basic and advanced search methods learnt in the module after applying and implementing them to solve problems for the Hinger game.
- Enhance your Python programming and problem-solving skills for Artificial Intelligence application.

Specification

Overview

The objective of this coursework is to design and implement solutions for a combinatorial game problem known as *Hinger*, with a focus on comparing the efficiency of different techniques learned in the module. You will collaborate with fellow students (your coursework partners) as a group, subject to approval from the assessment setter. All coding components for this coursework should be part of a single Python project, structured across multiple .py files.

Some tasks in this exercise require some research, particularly around efficient algorithms and better strategies for playing the game.

Each Python file created as part of this assessment should be appropriately commented and *must* include a header comment that contains your group number, your student ID and your assessment partners' student IDs. You may adapt the project header template *header.py* in the briefs folder on BlackBoard for use in each module file.

Python files without a header comment with your group number and student ID will not be marked.

Description

To support your progress through the coursework, several subtasks are marked with an asterisk (*). These may be deferred to a later stage, as they tend to be more complex and have less impact on subsequent tasks. This structure allows you to prioritize foundational tasks first, while reserving more complex components for when you are better prepared to tackle them.

Hinger Game

The **Hinger** game H(m,n) is played on an $m \times n$ rectangular board consisting of m rows and n columns. The top-left cell is designated as position (0, 0). Each cell on the board is either (i) empty, or (ii) **active** (meaning it contains a pile of one or more counters). Two cells (a, b) and (x, y) are defined as **adjacent** if $\max\{|a-x|, |b-y|\}$=1. This means they are directly next to each other: horizontally, vertically, or diagonally. A binary state refers to a board configuration where each cell contains either zero or one counter.

An **active region** is a maximal connected set of active cells, denoted as region R, with the following three properties: (i) each cell in R is active, (ii) any two cells in R are connected by a path of adjacent active cells within R, and (iii) all active cells adjacent to any cell in R are also included in R. For example, in *State A* of Figure 1, all cells containing one counter form a single active region, while the cell at position (0, 4), which contains two counters, constitutes a separate region.

1	1	0	0	2
1	1	0	0	0
0	0	1	1	1
0	0	0	1	1

1	1	0	0	1
1	1	0	0	0
0	0	**1**	1	1
0	0	0	1	1

1	1	0	0	1
1	1	0	0	0
0	0	0	1	1
0	0	0	1	1

Figure 1: Example of three states of the Hinger game played on a board with 4 rows and 5 columns. Left (state A) contains ten active cells forming two distinct active regions. Middle (State B) is a binary state and contains a hinger cell (2,2) highlighted in bold. Right (State C) displays three active regions. In a hypothetical gameplay scenario, player Alice performs a move at position (0,4), transitioning the board from State A to State B. Subsequently, player Bob performs a move at position (2,2), transforming State B into State C. Bob wins the game, as his move targets a hinger cell, which is a winning action under the game's rules. Note that the cost of Alice's move is 1 while that of Bob's move is 4 because there are three active cells adjacent to (2,2) when the move occurs.

In *Hinger,* a single legitimate move consists of removing one counter from an active pile. The cost of a move at cell (i, j) is one plus the number of active cells adjacent to (i, j) (see Figure 1 for an illustration). A **hinger** is a special type of active cell: it has only one counter and when a move is made on a hinger (i.e., the last counter is removed from this cell), the cell becomes empty and the number of active regions on the board increases by one. An example of a hinger is shown in bold in *State B* of Figure 1. The game is played by two players who take turns removing one counter from the board. The first player to make a move on a hinger **wins** the game, as this action triggers a critical change in the board's

structure. If all counters are removed from the board without either player acting on a hinger, the game ends in a **draw.**

A safe path between two states, S_1 and S_2, is defined as a sequence of moves that transforms S_1 into S_2, such that none of the intermediate states (including both S_1 and S_2) contains a hinger cell. In other words, the entire transition must avoid increasing the number of active regions on board. For example, no safe path can include *State B* in Figure 1, since it contains a hinger cell at position (2, 2).

Python Libraries

Only the Python standard libraries (see https://docs.python.org/3.12/library/index.html for a complete list) and Matplotlib library (https://matplotlib.org/) are allowed for Tasks 1 to 3. For Task 4, you may also use the Pygame library (https://www.pygame.org/docs/) for visualisation. If you are not sure which libraries are permitted, you should contact the assessment setter or module organiser for clarification. Your code should be runnable with **Python 3.12**, and it is your responsibility to ensure that your program runs smoothly on the lab PCs.

Tasks

1. Create a Python module file named *a1_state.py* within your project folder. In the module file, define a *State* class that encapsulates the basic structure and behavior of a game state in the game Hinger. The class should include one instance variable named *grid* representing the number of counters in each cell of the board. Your *State* class should also have:
   a. A proper initialiser, which receives a 2D list grid as a parameter.
   b. A sensible *__str__* method, which returns a readable string representation of the state (e.g., a matrix similar to those in Figure 1, without borders).
   c. A generator method named *moves()* that yields all possible states reachable in one move (i.e., removing one counter from any active cell).
   d. A method *numRegions()* that calculates and returns the number of active regions on the board.
   e. A method *numHingers()* that returns the number of hinger cells in the current state.
   f. Any additional utility methods that support gameplay logic or improve code clarity.
   g. Within this file, define a test function named *tester()* to validate your implementation. This function should execute automatically when the script *a1_state.py* is run directly, but not when imported as a module. Inside this function, implement the following tasks:
      i. Create a State instance named *sa* representing State A from Figure 1, and print its string representation.
      ii. Implement additional testing cases to verify your implementation.

2. Create a module file named *a2_path.py* within your project folder. This module should contain a set of functions for determining and retrieving safe paths between states in the Hinger game, along with a test harness to validate and compare your implementations. Your file should contain the following functions.
   a. A function *path_BFS(start,end)* that receives two binary states, *start* and *end,* and returns a safe path (as a list of moves) if such a path exists, or *None* otherwise, using the **Breadth-First Search** (BFS) algorithm.
   b. A function *path_DFS(start, end)* which receives two binary states, *start* and *end,* and returns a safe path (as a list of moves) if such a path exists, or *None* otherwise, using the **Depth-First Search** (DFS) algorithm.
   c. A function *path_IDDFS(start, end)* which receives two binary states, *start* and *end,* and returns a safe path (as a list of moves) if such a path exists, or *None* otherwise, using the **Iterative Deepening Depth-First Search** (IDDFS).
   d. A function *path_astar(start,end)* which receives two binary states, *start* and *end,* and returns a safe path if one exists, or *None* otherwise, using the **A*** search algorithm. You must include comments directly above the function definition to explain and justify the **heuristic function** used in your A* implementation.
   e. A test function named *tester()* to test the functions you implemented.
   f. Write a function *compare()* to evaluate and compare the performance of the four search algorithms (BFS,DFS, IDDFS, and A*) in finding a safe path between two game states in Hinger. You should design the output of this function to include metrics such as correctness, efficiency, and scalability.
   g. [*] A function *min_safe(start,end)* which receives two states *start* and *end* (which are not necessary binary), and returns a safe path with the minimal cost (i.e., the safe path with the lowest total sum of move costs) if such a path exists. Otherwise, it should return None. You need to choose and implement an efficient searching algorithm for this task, and justify your choice in the code comments directly above the function definition.

3. Create a module file named *a3_agent.py* within your project folder. This module will define an **Agent** class that models a player capable of making strategic decisions in the Hinger game. The agent should be designed to support multiple modes of play and demonstrate intelligent behavior based on the current game state. The class should include two instance variables: *name* is a string representing the agent's name (e.g., your group name, such as A9, as the default name) and *modes* is a list of game-playing strategies available to your agent. Your Agent class should also include the following:
   a. A proper initialiser, which receives two parameters: *size* and *name.* The first parameter *size* represents the board size as a tuple (m,n), where m is the number of rows and n is the number of columns. The second parameter *name* is an optional string representing the agent's name, with your group name as the default value. The initialiser should set up the agent accordingly.
   b. A sensible *__str__* method.
   c. A method *move(state, mode)* that receives a *State* object representing the current game board and a playing strategy, and returns a legitimate move for

your agent (e.g. a cell position (i, j) to act on). If no move is possible (e.g. there are no active cells), return *None*. Use the best strategy you have implemented as the default value for the *mode* parameter.

d. Additional supporting methods as needed to facilitate decision-making and strategy implementation for playing a binary Hinger game (that is, the game starts with a binary state). Your *modes* list should contain (i) an element named 'minimax', for which your agent plays using the minimax strategy, and (ii) an element named 'alphabeta', for which the agent plays using the alpha-beta pruning strategy.
e. Within this file, create a test function named *tester(),* which should run automatically when the file *a3_agent.py* is executed directly (but not when imported). In this function add appropriate code to demonstrate and validate the agent's behavior across scenarios.
f. [*] Implement other advanced strategies for your agent. For example, you may consider implementing a version of Monte Carlo Tree Search (MCTS).
g. [*] A method named *win(state)* that returns *True* if the given state (which is not necessarily binary) is a winning position for the player whose turn is to move, and *False* otherwise.

4. Create a new module file named *a4_game.*py within your project folder. This module will implement the core gameplay loop for the Hinger game, allowing two agents (or one agent and a human player) to take turns making moves on a shared game state. Within this file, define a function with the signature *play(state, agentA, agentB)* to simulate a complete game session between two players. It receives a *State* object representing the initial game board. Furthermore, agentA and agentB are instances of the *Agent* class (automated player) defined in Task 3, or a human player. For instance, *agentA=None* indicates that agentA is a human player. The function should perform all of the following tasks:
    - Alternate turns between agentA and agentB.
    - It should detect when the game is over: Declare a **win** and return the name of the winner if a player acts on a hinger cell, or declare a **draw** and returns *None* if all counters are removed and no hinger was triggered.
    - It should detect and handle illegal moves: If an agent or a human player attempts an invalid move (e.g. on an empty cell or non-existence cell), terminate the game and declare the opponent as the winner (e.g. return the name of the winner).

    Your module file should also have:

    a. A proper tester() function to validate your implementation.
    b. [*] Conduct you own research and implement additional features to make the game more engaging. Possible extensions could include: a timing-limited game where each turn has a time limit (e.g., 60 seconds), tracking total time used by each player, counting and displaying the number of moves made

during the game, saving the move history to a file for analysis or replay, or implementing a graphical user interface (GUI) for visual gameplay.

5. Polish and clean your code to improve its quality. This includes, but is not limited to, the following aspects: appropriate use of comments to explain logic and decision, improved readability through consistent formatting and naming conventions, enhanced robustness ensuring your code handles edge cases and errors gracefully, consideration of scalability (allowing your code to handle larger boards or more complex scenarios), and optimisation for efficiency (such as time usage and winning capacity).

Finally, each student in the group should write an individual reflection report (maximum two pages) structured into the following four sections:
(i) The first section should provide a clear summary of your progress across tasks 1-4, indicating which tasks have been fully completed (including any additional functionalities), partially done, or not completed. You should also include a description of your individual role in completing each task.
(ii) The second section should include a discussion of the performance of your group's search algorithms (e.g. BFS,DFS, IDDFS, and A*) based on Task 2. This may include a comparison of their relative performance, and suggestions for further improvement.
(iii) The third section should provide a discussion of the performance of your gaming-playing strategy from Task 3, including all the strategies that have been implemented and your assessment of their effectiveness.
(iv) The final section should include a personal summary of how you collaborated with your group members throughout this assignment. You should provide your own assessed contribution splits among the group, with a brief justification. For example, in a three-member group, you may suggest a 33%–33%–33% split for equal contributions, or a 25%–25%–50% split if one member contributed significantly more. In a two-member group, a 50%–50% split may be appropriate if contributions were balanced. You may include the number of times that your group has met (either virtually or face-to-face for this assignment), how the tasks are allocated, and any challenges encountered during collaboration and how they were addressed. You should also include suggestions for improving collaboration in future group projects, such as better task planning, clearer communication, or using collaborative tools more effectively to track progress and share updates.

You should name your PDF report file using your group name and your student ID. For example, use the name A9_12345678.pdf for a student with ID 12345678 in group A9. Note that each student working in a group should be submitting the same Python code, but their own copy of the reflection report. You should include the reflection report in your project folder.

Relationship to formative assessment

This assessment builds upon the work you have completed in your lab sessions and follows *similar* design methods. If you would like further feedback on your lab work, please approach your AT during further lab sessions.

Plagiarism, collusion, and contract cheating

The University takes academic integrity very seriously. **You must not commit plagiarism, collusion, or any form of cheating (such as using any large language models to generate code) in your submitted work**. Our Policy on Plagiarism, Collusion, and Contract Cheating explains:

- what is meant by the terms 'plagiarism', 'collusion', and 'contract cheating'
- how to avoid plagiarism, collusion, and contract cheating
- using a proof reader
- what will happen if we suspect that you have breached the policy.

It is essential that you read this policy and you undertake (or refresh your memory of) our school's training on this. You can find the policy and related guidance here: https://my.uea.ac.uk/departments/learning-and-teaching/students/academic-cycle/regulations-and-discipline/plagiarism-awareness

The policy allows us to make some rules specific to this assessment. Note that:

In this assessment, you are permitted to work within your group prescribed by the assessment setter. Discussing solutions to tasks between groups, or groups otherwise working together to perform the assessed tasks will be considered as a breach of university regulations. Please pay careful attention to the definitions of contract cheating, plagiarism and collusion in the policy and ask your module organiser if you are unsure about anything.

Deliverables

You are required to zip your entire project folder, **which contains only the module files and the files mentioned explicitly in the brief, and without any subfolders**, into a single .zip file. You should name your zip file using your group name and your student ID (e.g. B9_1234567.zip for a student with ID 1234567 in group B9), and submit the zip file via the submission point on BlackBoard, which will be made available closer to the submission deadline. You should contact the assessment setter for explicit permission if you need to include additional files in your submission and/or to use another file format for zipping your folder. If you encounter any issues surrounding zipping your folder in this manner, please contact a member of the teaching team who will assist you with this. A reminder that all code should be appropriately commented, including the required module headers on each module file.

A **live demonstration** of your Hinger game implementation will take place in **Week 6**. During the demo, you should be prepared to:

- Showcase the functionality of each module
- Explain your design choices and strategy implementations
- Demonstrate gameplay between agents (and optionally, a human player)

- Highlight any advanced features or extensions you have added
- Answer questions from the teaching team to demonstrate your understanding

Make sure that your code is well-organized, documented, and ready to run smoothly on the lab machines. This is your chance to show off your creativity and problem-solving skills.

Resources

Both the course texts and the lecture materials give guidance on how each of these tasks should be completed.

Marking scheme

1.	State Class (Task 1)	20
2.	PathFinder (Task 2)	30
3.	Agent (Task 3)	20
4.	Game (Task 4)	10
5.	Understanding & Report	20

Total marks available: 100