AI 500: Foundations of AI

Programming Assignment 1

13 September, 2025

Important Policies and Guidelines

Academic Integrity

WARNING: ZERO TOLERANCE FOR PLAGIARISM

All submissions will be checked for plagiarism. Any case of academic dishonesty will result in a grade of zero for the entire assignment and will be referred to the appropriate disciplinary committee.

- **AI-Generated Code:** Submitting code that is identical or substantially similar to that of another student is considered plagiarism, **even if both students used an AI tool to generate it**. You are responsible for ensuring your final submission is your own work.
- **Clarification:** If you are unsure about what constitutes plagiarism, it is your responsibility to contact the TAs for clarification **before the submission deadline**. Ignorance will not be considered a valid excuse.

Deadline and Late Submissions

- **Due Date:** Sunday, 28th September 2025, 11:55 PM.
- Submissions must be made on LMS.
- No late submissions will be accepted under any circumstances. Please submit well before the deadline to account for any potential technical issues.

Submission Guidelines

You are required to submit a single '.zip' file containing your three completed notebooks and your comparison text file. Please follow the naming and folder structure conventions precisely.

1. Create a main folder and name it using your roll number: `RollNumber_PA1`
2. Place your four completed files inside this folder. Rename your files according to the following convention:
   - `RollNumber_Part1_Search.py`
   - `RollNumber_Part2_AStar.ipynb`
   - `RollNumber_Part3_MiniMax.py`
   - `RollNumber_comparison.txt`
3. Compress (zip) the main folder. The final submitted file should be named: `RollNumber_PA1.zip`
4. Upload the single '.zip' file to the assignment submission portal on LMS.

Part 1: Search Algorithms

Assigned TA: Muhammad Ashhad Ali

Search in a Maze (60 Marks)

In this task, you will implement three fundamental search algorithms: **Depth-First Search (DFS)**, **Breadth-First Search (BFS)**, and **A* Search**. Your goal is to use these algorithms to find a path through a 20x20 maze from the bottom-right corner to the top-left corner.

Getting Started: The Maze Configuration

The maze is represented as a grid where each cell has a coordinate `(x,y)`. The `x` value corresponds to the row (1 is the top row) and `y` to the column (1 is the leftmost column).

The maze's wall layout is loaded into a dictionary, which you can access via `m.maze_map`. This dictionary maps each cell coordinate to another dictionary that shows open paths. A value of `1` means a path is open, and `0` means it is blocked.

For example, the entry for cell `(1, 1)` might look like this:

```
(1, 1): {'E': 1, 'N': 0, 'S': 1, 'W': 0}
```

This indicates that from cell `(1, 1)`, you can move East ('E') and South ('S'), but not North ('N') or West ('W').

Your Implementation Tasks

For all algorithms, you must adhere to the following general requirements:

- The **start position** is the bottom-right corner, `(20, 20)`.
- The **goal position** is the top-left corner, `(1, 1)`.
- When multiple paths are available, your algorithm must explore them in the priority order: **North, West, South, East**.
- Each search function must return two lists: one containing all visited cell coordinates and another containing the coordinates in the final solution path.

- **Task 1.1: Depth-First Search (DFS) (15 Marks)**
  Implement the DFS algorithm in the `DFS` function in `task1.py`.
- **Task 1.2: Breadth-First Search (BFS) (15 Marks)**
  Implement the BFS algorithm in the `BFS` function.

- **Task 1.3: A* Search (15 Marks)**
  Implement the A* Search algorithm in the `AStar` function. You must use the **Euclidean Distance** for the heuristic and a step **cost of 1** for moving between cells.

Comparison and Analysis (15 Marks)

After implementing the algorithms, answer the following questions in a plain text file named `comparison.txt`. Provide a clear justification for each answer based on your observations and the theoretical properties of the algorithms.

1. Which search algorithm(s) perform better in terms of time efficiency (i.e., runs fastest)? Why?
2. Which search algorithm(s) is better in terms of returning the shortest path? Why?

Running and Evaluating Your Code

To test your implementations, run the `task1.py` script from your terminal with the appropriate command-line argument:

- Run DFS: `python task1.py -d`
- Run BFS: `python task1.py -b`
- Run A* Search: `python task1.py -a`

After an algorithm completes its search, performance statistics (Execution Time, Path Length, Nodes Explored) will be printed to the terminal. A visualization window will then open, showing the explored cells in one color and the final path in another.

Informed Search with A*

(50 Marks)

Introduction

In this part, your goal is to implement the **A* search algorithm** to find the shortest walking path between the SSE building and the SDSB building on a map of the LUMS campus.

Getting Started

All your work will be done in the `RollNumber_Part2_AStar.ipynb` notebook. The campus map is provided as a *networkx* graph where nodes are locations and edges are paths with a *weight* attribute for distance.

Your Task

Task 2.1: Implement the Heuristic Complete the `heuristic()` function. For this map, you must implement the **Haversine formula** to calculate the straight-line distance.

- **Variables:** ϕ is Latitude, λ is Longitude, and r is the Earth's radius (6,371,000 meters).
- **Formula Steps:**

$$\begin{aligned} a &= \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right) \\ c &= 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \\ d &= r \cdot c \end{aligned}$$

- **Important:** Python's math functions work with radians. You must convert latitude and longitude values from degrees to radians first using `math.radians()`.

Task 2.2: Implement A* Search Complete the `a_star_search()` function. Your implementation must use a **priority queue** to efficiently explore the graph, always expanding the node with the lowest $f(n)$ score, where:

$$f(n) = g(n) + h(n)$$

To implement this algorithm, you should follow these general steps:

1. **Initialization:** You will need three main data structures:
    - A **priority queue** to act as your frontier. It should be initialized with the start node. The priority of each item in the queue will be its f-score.
    - A **dictionary** to store the `g_score` for each node (the known cost from the start to that node). The g-score for the start node is 0.
    - A **dictionary** to keep track of the path. For each node, it will store the node that came immediately before it on the best path found so far.

2. **The Loop:** Your algorithm should loop as long as the frontier (priority queue) is not empty. In each iteration, you will extract the node with the **lowest f-score**.

3. **Neighbor Exploration:** Once you have a node, get all of its neighbors. For each neighbor, calculate a "tentative" g-score, which is the g-score of the current node plus the distance to that neighbor.

4. **Update if Better:** If this tentative g-score is better (lower) than any previously recorded g-score for that neighbor, it means you've found a better path to it. You should:

   - Record the current node as the parent of the neighbor in your dictionary.
   - Update the neighbor's g-score.
   - Calculate the neighbor's new f-score and add it to the priority queue.

5. **Path Reconstruction:** Once your loop reaches the goal node, your algorithm is done. Use the dictionary to backtrack from the goal all the way to the start, recording each node along the way. This sequence of nodes is your final, shortest path.

Visualization

After implementing the functions, the final cell of the notebook will run your code and display the shortest path on a static visualization of the campus map. At the end of the notebook, you are required to answer the theoretical questions in the final markdown cell. This is a graded component of the assignment.

Part 3: Adversarial Search (MiniMax)

Assigned TA: Eeman Adnan

Math Duel Game

This part of the assignment focuses on the **minimax algorithm**, which is used for decision-making in two-player games. To test your understanding and application of the concept, we will implement it for a simple mathematical game called **Math Duel**.

i) Understanding the Game

Math Duel is a two-player subtraction game:

- The game starts with a number (e.g., 21).
- Player 1 (human) and the AI take turns subtracting an **allowed move** (e.g., 1, 2, or 3) from the current number.
- The player who makes the number reach 0 **wins the game**.

Example Play

- Starting number: 10
- Allowed moves: {1, 2, 3}
- Player 1 subtracts 2 → New number = 8
- AI subtracts 3 → New number = 5
- Player 1 subtracts 1 → New number = 4
- AI subtracts 3 → New number = 1
- Player 1 subtracts 1 → New number = 0 ⇒ **Player 1 wins!**

This is the game we will analyze using **minimax** to determine the optimal strategy.

ii) Understanding the Code Files

For this assignment, you are provided with two files:

1. MiniMax.py – Contains the **class skeleton** for the game mechanics. Several methods are left as **TODOs** – your task is to implement them.
2. `main.py` – Contains code to run your implementation.

iii) Assignment (To Do!)

You need to implement four key functions inside `MiniMax.py`.

A – `make_move()` (5 marks)

This function makes a move in the game.

Arguments:

- `move`: the number to subtract from the current state.

Tasks:

- Subtract `move` from `current_number`.
- Check if the game has ended (`current_number == 0`).
- If game is over, set the winner.
- Switch turn to the other player.

B – `evaluate()` (5 marks)

This function evaluates a finished game state.

Tasks:

- Return +1 if the MAX player wins.
- Return -1 if the MIN player wins.
- Return 0 if the game is not yet over.

This is a basic utility function that the minimax algorithm relies on.

C – `minimax()` (20 marks)

This function implements the **minimax algorithm with alpha-beta pruning**.

Arguments:

- `state`: current number
- `depth`: recursion depth
- `is_maximizing`: True if it's MAX's turn, False if MIN's
- `alpha`, `beta`: pruning boundaries

Tasks:

- If `state == 0`, return the evaluation (terminal state).
- If maximizing:
  - Explore all possible allowed moves.
  - Return the maximum evaluation.
- If minimizing:
  - Explore all possible allowed moves.
  - Return the minimum evaluation.
- Use **alpha-beta pruning** to cut off unnecessary branches.

D – `get_best_move()` (10 marks)

This function finds the **optimal move** using minimax.

Tasks:

- For the current player, simulate each legal move.
- Use `minimax()` to evaluate the outcome.
- Pick the move with the best evaluation (max for Player 1, min for AI).

iv) Example Run

Once you finish the assignment, you should be able to run `main.py` to see results. You can test with a different starting number by changing the following line in `main.py`:

game = MathDuel(start_number=16, allowed_moves=[1, 2, 3])

Please do not change anything else in `main.py`.

Guide: Running `main.py` in VS Code

1. Open Your Project

1. Open **Visual Studio Code (VS Code)**.
2. Go to **File → Open Folder...**
3. Select the folder where your files are saved.
4. Make sure both files are in the same folder:
   - `MiniMax.py`
   - `main.py`

2. Open a Terminal in VS Code

1. In VS Code, click **Terminal → New Terminal**.
2. A terminal will open at the bottom of VS Code.

3. Navigate to Your Folder (if needed)

If the terminal is not already inside your project folder, move there using the `cd` command:

```
cd path/to/your/folder
```

4. Run Your Program

Now type:

- Windows:

  ```
  python main.py
  ```

- Mac:

  ```
  python3 main.py
  ```

5. Example Output

If everything is set up, you'll see something like:

```
Current number: 10
Player 1's turn (MAX)
Allowed moves: [1, 2, 3]
Your move: 2
Current number: 8
AI's turn (MIN)
Allowed moves: [1, 2, 3]
AI plays: 3
```