CS 165A – Artificial Intelligence, Fall 2025
Machine Problem 2

Responsible TA: Xuan Luo, xuan_luo@ucsb.edu

Due: December 5, 2025, 11:59 PM

1 Notes

- Please review the "Policy on Academic Integrity" in the course syllabus carefully.
- Check Canvas regularly for any updates or corrections to this assignment.
- You can use any algorithm to solve this problem.
- Submit your report and code electronically as instructed.
- Direct any questions about MP2 to the Canvas MP2 Q&A forum.
- **AI tools like ChatGPT are permitted for this assignment.** However, you must include a **ai_usage.txt** file with your submission, detailing the prompts and code snippets generated. Note that while AI tools are allowed, **sharing your prompts or code with others is strictly prohibited**. We will use plagiarism detection software to enforce this policy.
- The assets are from Minecraft Education Edition and used in compliance with the EULA.

2 Background: Search in Dynamic Environments

In the vast landscapes of robotics and artificial intelligence, the challenge of navigating dynamic environments lies at the heart of innovation. From self-driving cars weaving through bustling city streets to virtual agents exploring uncharted digital realms, the ability to search and adapt in ever-changing worlds powers progress across countless domains.

This project invites you into the world of GauchoMiner, a Minecraft-inspired mining game designed to sharpen your skills in search-based challenges. In this machine problem, you will tackle navigation on a static map, mastering classic algorithms like A*, Expectimax, and Minimax to uncover efficient paths. Figure 1 shows an example.

3 Game Overview

In this assignment, you'll develop a program to play a 2D Minecraft-inspired mining game on an $X \times Y$ grid. Your task is to control a miner who navigates and digs through a map filled with various blocks, aiming to collect gold scattered across the grid to achieve the highest possible score. The miner starts at a designated position with a limited amount of energy, and every move or dig consumes some of this energy, with the cost varying depending on the type of block encountered. The map is fully visible, revealing all block locations and gold positions from the start, but energy limits how much you can explore and dig. Additionally, zombies move around the map; you must avoid them, as failing to do so will require the miner to expend energy to defeat them. Your program must decide the best sequence of moves and digs to maximize the total score before the miner's energy is depleted, at which point the game ends.

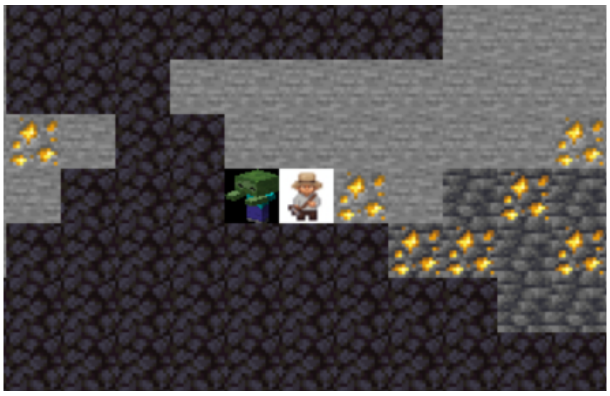
Figure 1: An illustration of the game environment.

3.1 State Representation

The game's state keeps track of everything happening as you play. It's defined as:

$$s = (A, M)$$

Here, A represents the agent's situation—your status as the miner—and M is the current map, showing the world you're digging through. These two parts together give a complete picture of the game at any moment.

Agent State (A)

The agent state $A = (p, e, r)$ tracks your personal progress:

- $p = (x, y)$: Your current position on the grid, where $P = \{(x, y) \mid 0 \leq x < X, 0 \leq y < Y\}$. This shows where you are, and you can move using four actions: W (up), A (left), S (down), or D (right) to an adjacent spot, as long as it's within the grid.
- $e \geq 0$: Your remaining energy, starting from a total E. Moving and digging drain this, and how much depends on the block you hit.
- r: Your total score, or reward, which starts at 0 and grows as you collect gold. It's the running tally of points you've earned.

This tells us where you are, how much energy you've got left, and how well you're doing.

Map State (M)

The map state M is an $X \times Y$ matrix, where each position (x, y) holds a block. It is randomly generated at the start and represents the world. All actions consume energy to move to a new position. The map has boundaries; attempting to move beyond them results in a failed move and costs 1 energy. The possible blocks are:

Notice that there will be 3-10 zombies ( ) on the map. Zombie will attempt to move one square closer to the player (i.e., reduce the Manhattan distance) with a probability $P_{\text{move}} = \min(4.0/d_{\text{manhattan}}, 1.0)$, where $d_{\text{manhattan}}$ is the current Manhattan distance to the player. With probability $1.0 - P_{\text{move}}$, the zombie will randomly move for that turn.

Zombies are not smart pathfinders: they only attempt to move to an adjacent square that directly reduces the Manhattan distance. If all such moves are blocked by an obstacle or the boundary, the zombie "bumps the wall" and remains in its current spot for that turn.

3.2 Initial State

The initial state is $s_0 = (A_0, M_0)$, where $A_0 = (p_0, E, 0)$ represents the agent's initial configuration: p_0 is the starting position (randomly assigned), E is the full energy, and the score is zero. The initial map M_0, an $X \times Y$ matrix, is randomly generated based on a random seed, which will be detailed later.

Table 1: Map State Blocks. Please note that their actual name stored in the map is different from the notation here.

Name	Icon	Energy Cost	Reward
Empty	⬛	1	0
Dirt	🟫	2	0
Stone	⬜	4	0
Deepslate	▩	10	0
Gold ore	🟨	4	5
Deepslate gold ore	🟨	10	5
Zombie	🧟	50	0
Boundary	-	1	0

3.3 Actions

At each timestep, the agent selects one action from the set $\{W, A, S, D, I\}$ to navigate or interact with the $X \times Y$ grid. Moving to a block (via W, A, S, D) automatically digs it, consuming energy based on the block type (see Table 1). Digging updates the agent's score and will turn the block to 'Empty'. The actions are:

- W: Move up one cell (to $(x, y - 1)$) and dig the block there.
- A: Move left one cell (to $(x - 1, y)$) and dig the block there.
- S: Move down one cell (to $(x, y + 1)$) and dig the block there.
- D: Move right one cell (to $(x + 1, y)$) and dig the block there.
- I: Stay in place at (x, y) without moving or digging. Consumes 1 energy.

Each action determines the agent's movement and interaction with the map, updating the agent's position, energy, score, and the map state as applicable.

3.4 Termination Conditions

The game terminates under one of two conditions:

- The agent's energy is depleted, i.e., $e \leq 0$ in the agent's state A.
- All ores on the map M have been dug up.

3.5 Run the Code

To run the GauchoMiner game, you need to set up the required Python libraries and execute the provided script. The game runs with a graphical interface (using Pygame) for visualization, allowing you to observe the miner's actions on the grid. You have to install the following Python libraries:

```
pip install pygame opensimplex
```

Ensure you have Python 3.11 installed. You can use "python --version" to check it.

To launch the game with default settings, execute the following command in your terminal or command prompt from the directory containing `new_game.py`:

```
python new_game.py
```

You can customize the game environment using optional command-line arguments to modify the random seed for map generation and the game speed. The available arguments are:

- `--seed`: Sets the seed for random map generation. Use the same seed to reproduce the same map.
- `--fps`: Controls the game speed in frames per second. Lower values (e.g., 1) slow down the visualization for easier debugging.
- `--grid_size`: Controls the grid size for visualization.
- `--zombies`: Controls the number of zombies on the map.

4 Implementation guidelines

In this task, you will implement a search algorithm to solve the mining game on a large 40×30 grid. You must implement the function `agent_logic` in the file `agent_logic.py`. The parameters of the function are:

- `cave_map`: A 2D array representing the game map. Accessing `cave_map[x][y]` provides the block type at position (x, y), where x is the column index (increasing from left to right) and y is the row index (increasing from top to bottom).
- `position`: A tuple (x, y) indicating the agent's current position on the map.
- `energy`: An integer representing the agent's remaining energy.

Your task is to implement the search algorithm to collect golds. At each timestep, the function must return a single character from $\{'W', 'A', 'S', 'D', 'I'\}$, indicating the agent's next action.

4.1 Hints:

- You can start by implementing the simplest possible strategy: a greedy algorithm. At each step, identify the locations of all remaining gold on the map. Use a pathfinding algorithm (like A* or Dijkstra's) to find the minimum energy cost to reach each piece of gold. Then, choose to move towards the "closest" gold (the one with the minimum path cost). Your `agent_logic` function will return the first action (e.g., 'W', 'A', 'S', 'D') on that path.
- Be careful of oscillation issues. A naive agent might get stuck in a loop, moving back and forth between two positions (e.g., moving 'W' then 'S', 'W' then 'S', repeatedly). This can happen if the "best" target keeps changing. You can mitigate this by recording your recent path (e.g., maintaining a visited list for your current path) and either forbidding or adding a high cost to moves that immediately revisit the same locations.

5 Extra Credits: Up to 20% bonus.

We will open a MP2_code submission portal for the regular submission, then open another one named MP2_Competition in the last 24 hours. The submission to MP2_Competition is optional, it will use different test cases but still follow the same distribution. The submission to MP2_Competition is optional, and we will maintain a leaderboard. The 1st place receives 20% extra credit; 2nd and 3rd place receives 10%; top-10 receives 5%. You can use any algorithm for MP2_code and MP2_Competition.

6 Submission Guidelines

6.1 Program Execution

Submit your implementation to the Gradescope assignment named `MP2_code`. Include the following files:

- `agent_logic.py`: The Python script containing your A* implementation. **Note: Your program must not print any output and should only return a single character from** $\{'W', 'A', 'S', 'D', 'I'\}$.
- Any additional files required for execution, such as a JSON file to store the agent's memory of previous actions, if applicable.

The autograder will automatically execute and evaluate your code on a secret test set. The final score of your submission will be determined based on the average points earned in the game.

6.2 Report Writing

The machine problem report should be submitted in a **Markdown format** and must clearly document your approach, implementation details, and findings. Your report must include your name, email, and perm number at the top of the first page. The report should have less than 2000 words. The structure of the report should be as follows:

- Describe in detail the algorithm and high-level search strategy you implemented to control the agent.
- Explain your rationale for choosing this particular design.
- Describe your agent's **target selection** logic. How does it decide *which* piece of gold (or other objective) to pursue at any given time?

6.3 Gradescope Submission

We will create two distinct assignments on Gradescope(`https://www.gradescope.com/courses/1153753`) for this module:

MP2_report: Submit only your report to this assignment. Your report should be a markdown file named report.md.

MP2_code: Submit all code-related documents here, including:

- Your source code named `agent_logic.py`
- The ai_usage.txt file that details the prompts and code snippets generated.

7 Grading Criteria

The final grade for this project will be determined based on a **100% base score**, with the opportunity to earn extra credit from the competition.

- **Report Submission (Required):** 20% of the base grade.
- **Average on Gradescope (Required):** 80% of the base grade.
  - **Testing Accuracy:** Assessed on a private, hidden test dataset. For average score $0 \leq \text{Score} < 500$, the score will be linearly interpolated between 0 and 100.
  - **Runtime Constraint:** Your code must complete execution within **10 minutes** (600 seconds). If execution time exceeds this limit, the process will be terminated, and the accuracy score awarded will be zero.
- **Extra Credit (Optional):** Up to 20% based on the ranking in MP2_Competition.