CS 165A – Artificial Intelligence, Fall 2024
Machine Problem 2

Responsible TA: Xuan Luo, xuan_luo@ucsb.edu

Due: Dec 7, 2024 11:59pm

1 Notes

■ Make sure to read the "Policy on Academic Integrity" in the course syllabus.
■ Any updates or corrections will be posted on Canvas.
■ You must work individually on this assignment.
■ Each student must submit their report and code electronically.
■ If you have any questions about MP2, please post them on the Canvas MP2 Q&A.
■ Plagiarism Warning: We will use software to check for plagiarism. You are expected to complete the project independently without using any code from others.
■ **There are no restrictions on which algorithm you choose to solve this machine problem.**

2 Background: Search in Dynamic Environments

Search in dynamic environments is a fundamental problem in robotics and artificial intelligence, with applications spanning diverse fields such as autonomous driving, urban traffic management, and virtual simulations. Effective search strategies allow systems to navigate complex, often unpredictable environments by identifying optimal routes and avoiding obstacles. For instance, in autonomous vehicles, real-time search is essential for safe navigation as it accounts for moving obstacles (e.g., pedestrians, other vehicles) and adjusts routes based on the current traffic flow.

This project aims to simulate a search-based challenge through a dynamic "Parkour" game. In this scenario, the player-controlled agent must continuously move forward, collecting coins to maximize their score, while avoiding moving obstacles (other cars and walls). The project simulates critical elements of real-world search scenarios: dynamic obstacles and the need to balance goal achievement (collecting coins) with safety (avoiding collisions).

3 Problem Definition

In this task, students are required to design and implement an intelligent searching agent, $\mathcal{A}$, to operate in a dynamic, grid-based environment. This environment simulates a "Parkour" scenario where Agent $\mathcal{A}$'s primary objectives are to:

- **Collect Coins:** Gather as many coins as possible to maximize its score.
- **Reach the Goal:** Navigate the map to reach a designated goal area while avoiding obstacles and collisions.
- **Optimize Path:** Balance between coin collection and movement penalties to achieve the highest possible score.

3.1 Environment

The environment, represented by a grid-based map M with dimensions 50×30, is procedurally generated with the following elements:

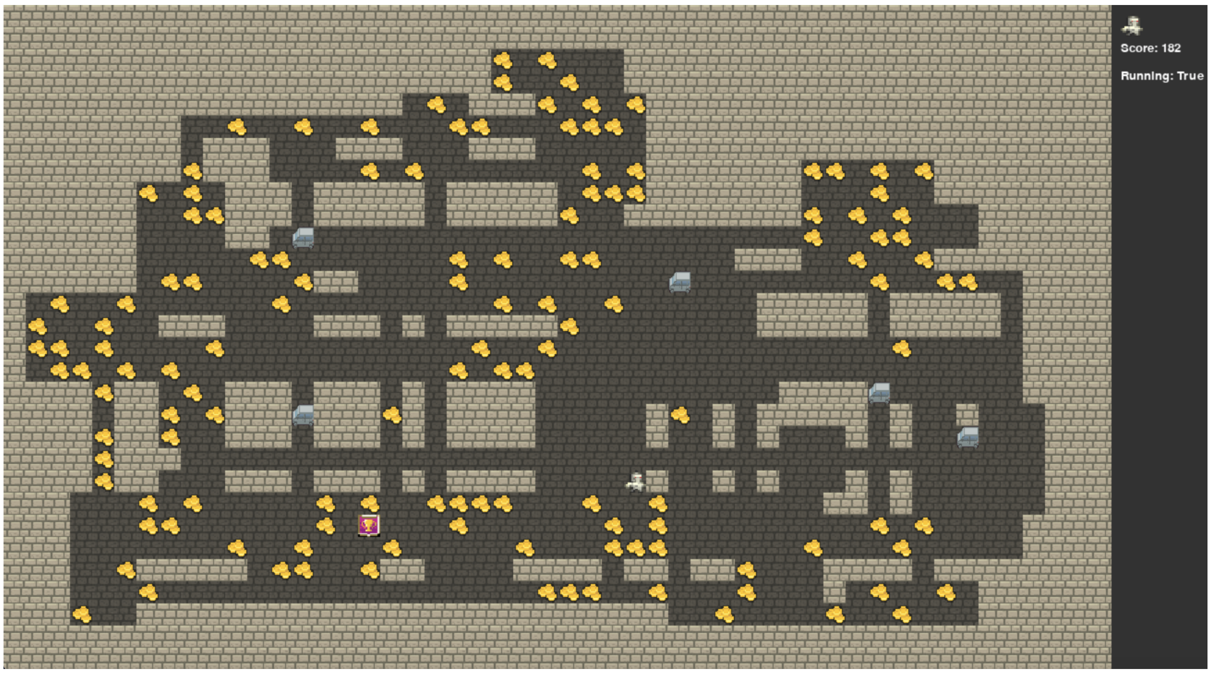

Figure 1: Example Environment. The robot is the player's agent.

- **Roads:** Open cells that allow movement for both agents and other dynamic elements.
- **Walls:** Static obstacles that no agent can cross.
- **Goal:** A specific area on the grid, randomly placed on a road cell, which ends the game when reached. This is denoted as the trophy on the map.
- **Coins:** Collectible items randomly distributed on road cells.

3.2 Dynamic Elements

The environment includes randomly moving cars $\{C_1, C_2, ..., C_k\}$ where:

- Cars move randomly each turn to adjacent road cells
- Cars collect coins when moving to a coin-containing cell
- Cars cannot move through walls or other cars
- Cars move after Agent $\mathcal{A}$'s movement each turn

3.3 Game Mechanics and Rules

Scoring System

- **Coin Collection:**
  - Each coin collected adds +10 to Agent $\mathcal{A}$'s score
  - Collected coins are immediately removed from the map
- **Turn Penalty:**
  - Every turn incurs a penalty of $-k$ points. Different environment have different k value
  - Score cannot go below 0
  - Penalty applies to all actions, including "Stay in place"

Termination Conditions

- **Collision with Obstacles:**
  - If Agent $\mathcal{A}$ collides with any moving car C_i or a wall
  - Game ends immediately
  - Score is set to 0
- **Reaching the Goal:**
  - If Agent $\mathcal{A}$ reaches the goal
  - Game ends successfully
  - Current score is claimed

3.4 Available Actions

Each agent has five possible actions per turn:

- **W:** Move up one cell
- **A:** Move left one cell
- **S:** Move down one cell
- **D:** Move right one cell
- **I:** Stay in current position

All movements are restricted to one cell per turn and must be to a valid road cell.

Performance Scoring

We will test your implementation on multiple environments. For each environment E_i:

- Let S_i be the baseline score (pre-determined)
- Let Stu_i be the student's achieved score
- The performance ratio is calculated as $R_i = \frac{Stu_i}{S_i}$

The test score is calculated as:

$$S_{test} = \frac{1}{n} \sum_{i=1}^{n} R_i$$

where n is the total number of test environments.

4 Hints

- Consider carefully how to balance between collecting coins and reaching the goal. When to collect coins and when to stop?
- Avoid oscillation behavior where your agent moves back and forth between objectives. This can be triggered if your objective functions are conflict with each other.
- You may implement "memory" by saving and loading agent states through a JSON file to track historical movements

5 Program Specification

Your primary task is to complete a script named `agent_A.py`. This script must contain a function called `logic_A`, which defines the movement logic of your AI agent. This function is invoked at each time step during the game simulation and determines the agent's action based on the current state of the game environment.

5.1 Function Parameters

```
def logic_A(cur_map, cur_position, cur_coins, cur_car_positions, penalty_k):
    # Your implementation here
    return action  # One of: 'W', 'A', 'S', 'D', 'I'
```

Input parameters:

- `cur_map`: 2D list of size 50×30, containing strings 'wall', 'road', or 'goal'
- `cur_position`: Tuple (x, y) of current position coordinates
- `cur_coins`: List of tuples, each (x, y) representing a coin position
- `cur_car_positions`: List of tuples, each (x, y) representing a car position
- `penalty_k`: Integer value of the movement penalty for each turn. This value will between 1 and 3.

Return value: String 'W', 'A', 'S', 'D', or 'I' representing the chosen action

5.2 Running the Simulation

Please download the MP2-codepack from Canvas. To initiate the simulation, execute the following command in your terminal:

```
python run_game.py --speed 5 --seed 777
```

The first parameter 'speed' specifies the simulation speed, which determines how quickly the game progresses. The second parameter 'seed', the random seed, influences the map generation. Using the same seed will consistently regenerate the same map configuration. To thoroughly test your agent's adaptability, vary the seed to expose it to different map layouts. Notice that you have to install the package 'pygame' on Python 3 to run the code.

6 Submission Guidelines

Please follow these instructions to ensure your submissions are processed correctly by the autograder and eligible for manual grading if necessary.

6.1 Gradescope Submission

We will create two distinct assignments on Gradescope for this module:

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

- A python script named agent_A.py. **Notice that your program should not print anything and only return a single value in [W, A, S, D, I].**
- Any additional scripts required for execution, e.g. a json file to store the agent's memory.

MP2_report: Submit only your PDF report to this assignment.

Important: Do not place your scripts or code inside a directory or zip file. The autograder should be able to run your code and provide accuracy results within a few minutes. While there is no limit to the number of submissions, only the last submission will be considered for grading.

7 Grading Criteria

7.1 Grading Breakdown

The final grade for this project will be determined based on the following criteria:

- **Complete Report:** 20%

- **Execution Specifications:** 10%
- **Correct Program Specifications:** 10%
  - Reads file names from command-line arguments
  - Produces correct standard output results
  - No segfaults or unhandled exceptions
- **Test Accuracy and Runtime:** 60%

7.2 Leaderboard and Bonuses

- The top scorer will receive a 50% bonus on their project grade.
- The top three scorers will each receive a 25% bonus.
- A real-time leaderboard will be available on Gradescope to track these rankings.

7.3 Accuracy and Time

The majority of the grade (60%) is based on the testing accuracy and runtime:

- **Testing Accuracy:** Assessed on a private dataset.
- **Runtime:** Your code must complete execution within 10 minutes; otherwise, it will be terminated, and the accuracy score will be zero.

Score Calculation

- Scores based on the test score S_{test}:
  - $0.0 \leq S_{test} < 0.60$: Scores will be linearly interpolated between 0% and 80%.
  - $0.60 \leq S_{test} < 1.00$: Scores will be linearly interpolated between 80% and 100%.
  - $1.00 \leq S_{test} \leq 2.00$: Scores will be linearly interpolated between 100% and 120%. The maximum score is 120%.

8 Report Guidelines

The project report should adhere to the following specifications:

- **Length:** The report must be between 1 and 3 pages, with no exceptions exceeding this limit.
- **Font Size:** The text must be at least 11pt in size.
- **Content Requirements:** Your report should comprehensively cover the following sections:
  1. **Algorithm:** Provide a concise explanation of your code's algorithm, including a description of classes and their basic functionalities.
  2. **Results:** Discuss your results on the provided datasets, including accuracy and running time.
  3. **Challenges:** Detail the challenges you encountered during the project and how you addressed them.
- **Submission Format:** Submit only one PDF report containing all the sections listed above. Do not include additional README files.
- **Identification:** Your report must include your name, email, and perm number at the top of the first page.