Maze Traversal AI Project

UCLA CSSI

Introduction

In this project, we will design an AI to find the shortest path through an **NxN maze**. Every **cell** in the maze is either a **wall** or some **terrain**. Different terrains have different **costs** of traversal. The **shortest path** is defined as a *path of terrain cells with the least total cost from entrance to exit*.

Starting from the top left corner, the **(0,0)th cell**, the AI iteratively selects new cells until it reaches the bottom right corner, the **(N-1,N-1)th cell**. At each iteration, there are some cells that have been explored (blue) and **candidate cells** adjacent to the explored cells (red). Given the **(x,y) locations** of all red cells (and their associated **terrain costs**), *the AI will choose which red cell to select next*. The AI will stop once it selects the **(N-1,N-1)th cell**.

Once the AI is finished, I will compute the **shortest path** from **(0,0)** to **(N-1, N-1)** out of **all cells your AI had explored**. The goal is to (1) find the shortest path in (2) the least number of iterations. An example of this process is shown below.

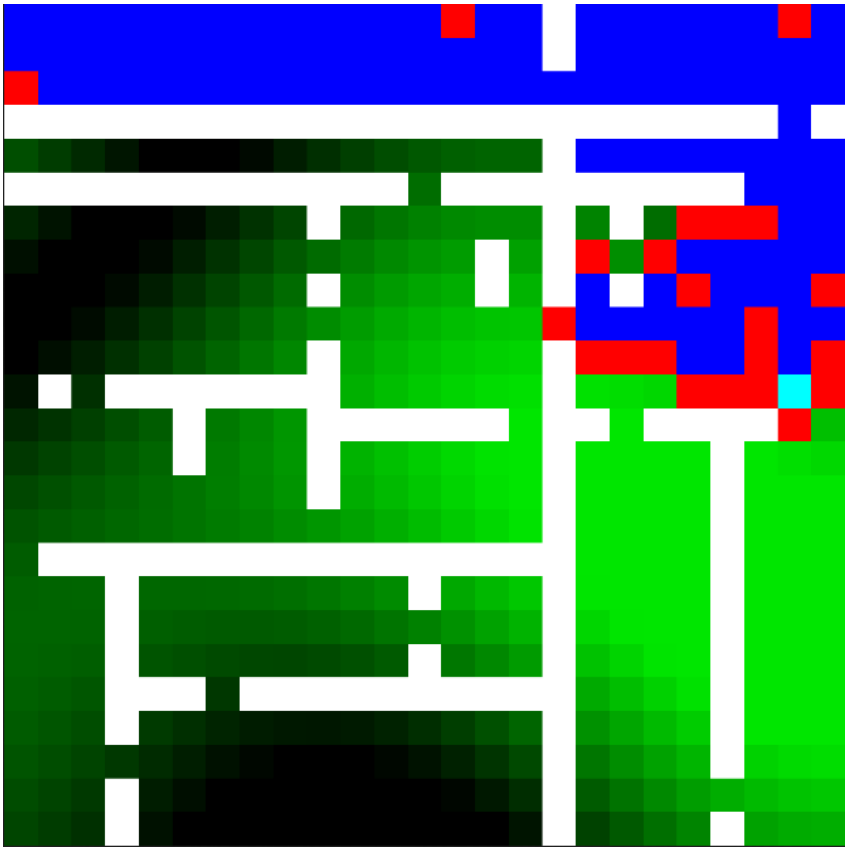

Figure 0: The blue and cyan cells represent the currently explored cells. The red cells represent the candidate cells. The cyan cell was the selected cell of the last iteration. The white cells are the walls. The green cells are terrain, where a lighter terrain corresponds with a higher cost of traversal.

Installation

Download the project from https://github.com/DerekQXu/maze-solver.

Install required packages by running "**pip3 install -r requirements.txt"** in the terminal tab of Pycharm. Change the **ROOT_DIR** variable in **config.py** to where you saved your project. Feel free to ping me or Ethan or the TA's if you have any issues!

Run

Run **“python3 main.py”**. This will test your **agent.py** on many mazes and grade it based on the quality of the final path and the number of iterations used. You want the “**final score: …”** value to be as large as possible. If you wish to see how the final score is computed or how to interpret the breakdown, please check the **Appendix** section. There are also animated gifs of each run in the **results** folder.

The actual competition will use different mazes (sizes: 10, 30, 50, 100, 200) than the default setup.

Examples are shown below:

```
============================
generating maze of size 30 (1/1)
maze generated!
 35%|███       | 519/1500 [00:01<00:03, 275.07it/s]
score: 74.09908688560421
    breakdown: completion_score:30.0,exploration_score:4.222222222222222,path_score:39.87686466338198,path_score_bonus:0.0
    animating...
============================
generating maze of size 40 (1/1)
maze generated!
 57%|█████     | 855/1500 [00:08<00:06, 96.39it/s]
score: 72.809381480399
    breakdown: completion_score:30.0,exploration_score:4.65,path_score:38.159381480399006,path_score_bonus:0.0
    animating...
============================
generating maze of size 50 (1/1)
maze generated!
100%|██████████| 1500/1500 [00:36<00:00, 41.63it/s]
score: 26.50966706602305
    breakdown: completion_score:26.50966706602305,exploration_score:0.0,path_score:0.0,path_score_bonus:0.0
    animating...
============================
final score: 66.41321850381054
```

Figure 1: Example of the terminal output.

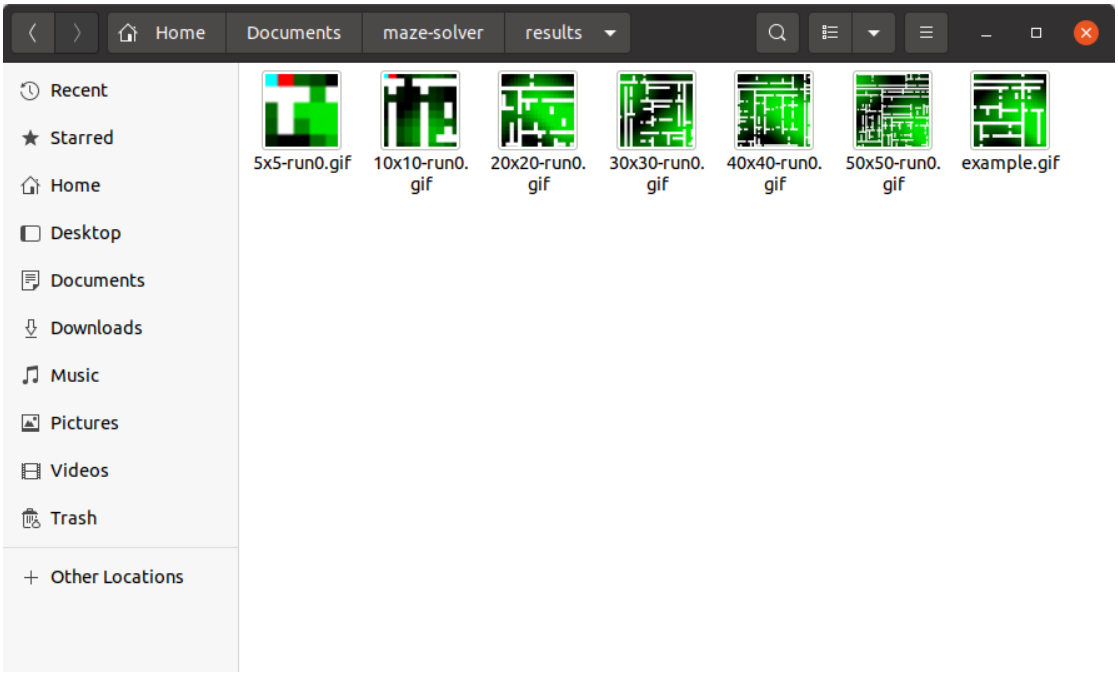

Figure 2: Example of the gifs.

Interface

The AI you will design is in **agent.py**. Specifically, you will write the **select_action(…)** function.

The **select_action(…)** function is given a list of red **candidate cells**, formatted as a list of **(x,y,terrain_cost)** tuples. The **select_action(…)** function will return **one cell/tuple** from this list. **You will decide which tuple to select** to (1) find the *shortest path* in (2) the *smallest number of iterations*. **You do not need to keep track of the shortest path.**

```
def select_action(self, candidate_cells, adjacent_cells_to_last_cell, last_cell):
    ###########################################################
    # write your algorithm here!
    random_index = int(random.random())*len(candidate_cells)
    action = candidate_cells[random_index]
    ###########################################################

    assert action in candidate_cells # you must choose from one of the candidate cells!
    if get_loc(action) == (self.N-1,self.N-1):
        self.done = True

    return action
```

In addition to the list of **candidate cells**, we include **2 additional inputs** you may find helpful. We summarize all 3 inputs and the return value of **select_action(…)** below:

- **candidate_cells** *(i.e. red cells)*: a **list** of **(x,y,cost) tuples** that are adjacent to the currently explored cells.
- **last_cell** *(i.e. cyan cell)*: the last **(x,y,cost) tuple** which was chosen.
- **adjacent_cells_to_last_cell** (*i.e. terrain cells adjacent to cyan cell*): a **list** of **(x,y,cost) tuples** adjacent to last_cell.
- **return value**: any **(x,y,cost) tuple** in **candidate_cells**.

You may not change the import statements at the beginning of **agent.py**. You may modify the **constructor** by adding new variables. You may use the size of the current maze, **self.N**, the maximum number of iterations, **MAX_ITERS**, the minimum terrain cost, **MIN_COST**, and the maximum terrain cost, **MAX_COST**. *Note: it is perfectly fine not to use any of these additional variables or function inputs!*

```
def __init__(self, N):
    self.N = N
    self.backtracking_path = {}
    self.done = False
    random.seed(123)

    ###########################################################
    # add any extra data_structures you need here!
    ###########################################################
```

Most importantly: **Have Fun and Good Luck!**

Hints (Read Me!)

Hint 1: The current **agent.py** selects a **random cell** from the set of **candidate cells**. Do you think you can do better? Which cell should we select first?

Hint 2: One idea is to choose *the candidate cell which is closest to the (N-1, N-1)th cell*. To do so, we can compute **the number of cells** between each **candidate cell, (x,y),** and the **(N-1, N-1)th cell**, which is **"(N-1-x) + (N-1-y)"**, and select the candidate cell with the least number of cells between it and the exit.

```
def select_action(self, candidate_cells, adjacent_cells_to_last_cell, last_cell):
    ############################################################
    # write your algorithm here!
    # return cell with smallest L1 distance from origin
    min_num_cells = float('inf')
    action = None
    for cell in candidate_cells:
        x,y,_ = cell
        num_cells_from_origin = self.N - 1 - x + self.N - 1 - y
        if num_cells_from_origin < min_num_cells:
            min_num_cells = num_cells_from_origin
            action = cell
    ############################################################

    assert action in candidate_cells  # you must choose from one of the candidate cells!
    if get_loc(action) == (self.N - 1, self.N - 1):
        self.done = True

    return action
```

This is called **Greedy Best First Search**, which *aims to reach the exit in the fewest number of iterations.*

Hint 3: Another idea is to select the *cell that adds the least cost*. To do so, we can initiate **a dictionary** that maps **each cell** to **the cost required to reach that cell** and choose the **candidate cell with the least cost**.

```
def __init__(self, N):
    self.N = N
    self.backtracking_path = {}
    self.done = False
    random.seed(123)

    ############################################################
    # add any extra data_structures you need here!
    self.cumulative_cost_dict = {}
    ############################################################
```

On each iteration, we can populate the dictionary with **cells adjacent to the one we last selected** (i.e. **red cells who are next to the cyan cell**). The **cost to reach this cell** is equal to **the terrain cost of the red cell + the cost to reach the cyan cell**. Once we have update the dictionary with the new red cells, we then choose **the candidate cell** with the least cost.

```
def select_action(self, candidate_cells, adjacent_cells_to_last_cell, last_cell):
    ############################################################
    # write your algorithm here!
    # update the dictionary
    for cell in adjacent_cells_to_last_cell:
        _, _, terrain = cell
        if last_cell is None:
            self.cumulative_cost_dict[cell] = terrain
        else:
            self.cumulative_cost_dict[cell] = terrain + self.cumulative_cost_dict[last_cell]

    # return the cell with smallest cumulative cost
    min_cost = float('inf')
    action = None
    for cell in candidate_cells:
        if self.cumulative_cost_dict[cell] < min_cost:
            min_cost = self.cumulative_cost_dict[cell]
            action = cell
    ############################################################

    assert action in candidate_cells # you must choose from one of the candidate cells!
    if get_loc(action) == (self.N-1,self.N-1):
        self.done = True

    return action
```

This is called **Djikstra's Algorithm**, which *guarantees we will find the shortest path before reaching the exit of the maze*.

Hint 4: How can we combine the **Greedy Best First Search** and **Djikstra's Algorithm** to *find the shortest path in the least number of iterations*? It seems both algorithms are first assigning some score to each candidate cell and then selecting the cell with the least score. Is there some way to *combine* the two?

Hint 5: In our experiments, is finding the shortest path more important or reaching it in the least number of iterations more important? Is this different from maze to maze, and can you detect when one is more important than the other?

Maze Traversal AI Project: Appendix

Scoring

To prevent the actor from running forever, we have a maximum number of iterations we can run for, called **MAX_ITERS**. The AI *will exit the maze early* whenever it has reached the exit before **MAX_ITERS** iterations. Our AI aims to (1) find the *shortest path* in (2) the *smallest number of iterations*. AIs are assigned a **grade** as follows:

$$grade = grade_{done} + \mathbb{1}["done"] \cdot \left(grade_{explor.} + grade_{path}\right) + \mathbb{1}["shortest\ path\ found"] \cdot grade_{bonus}$$

$$grade_{done} = 30 \cdot "the\ closest\ L2\ distance\ the\ agent\ got\ to\ (N-1, N-1)\ throughout\ search"$$

- Notice, the AI would score the full 50 points if it **reached the exit**. It would score some fraction of 50 points if it did not.

$$grade_{explor.} = 10 \cdot \frac{MAX\_ITER - "number\ of\ iterations\ traversed\ by\ agent"}{MAX\_ITER}$$

- Notice, the AI will score higher by **exiting the maze at an earlier iteration**, up to at most 10 points.
- This score will only apply if the AI **reached the exit**.

$$grade_{path} = 40 \cdot \frac{"shortest\ path\ cost" - \frac{"agent's\ path\ cost" - "shortest\ path\ cost"}{2}}{"shortest\ path\ cost"}$$

- Notice, the AI will score the full 40 points if it **found the shortest path**. The AI will score 0 points if the path it found has a cost 3x higher than the shortest path.
- This score will only apply if the AI **reached the exit**.

$$grade_{bonus} = 20$$

- This score will only apply if the AI **found the shortest path**.

The AI with the highest **average final grade** on a set of mazes with sizes *N = [10, 30, 50, 100, 200]* will win the competition!