Maze Traversal AI Project

UCLA CSSI

Introduction

In this project, we would like to design an AI agent to find the shortest path through a maze. Starting from the top left corner (0,0), the agent iteratively selects new cells until they reach the bottom right corner (N-1,N-1). At each iteration, the agent selects from cells that are adjacent to the currently explored cells, called **candidate cells**. Furthermore, every cell in the maze as a different terrain. Each terrain has an associated cost of traversing it. Thus, some paths may be "shorter in the number of cells" but actually "longer in terms of total cost." An example of this process is shown below.

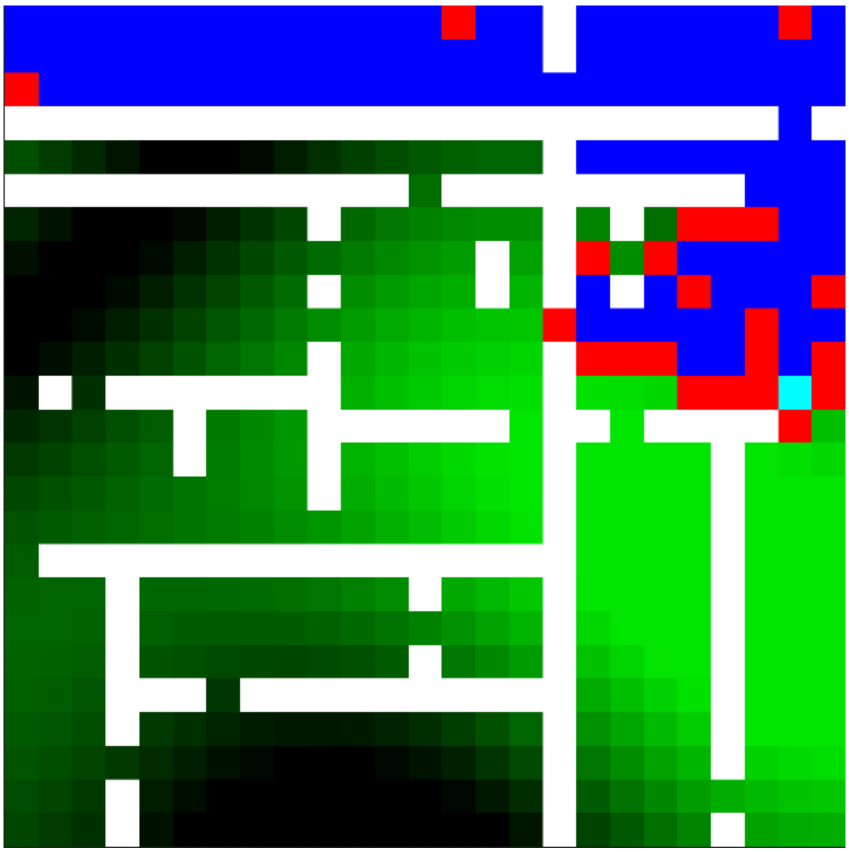

Figure 0: The blue cells represent the currently explored cells. The red cells represent the candidate cells. The cyan cell is the selected cell. The white cells are the walls. The green cells are terrain, where lighter terrain means higher cost.

Code and installation can be found here: https://github.com/DerekQXu/maze-solver.

Goal

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

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

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

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

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

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

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

- Notice, the agent will score the full 40 points if it **found the shortest path**. The agent 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 agent **reached the exit**.

$$grade_{bonus} = 20$$

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

The agent with the highest **average final grade** on a set of mazes with size *N = [1, 5, 10, 20, 30, 40, 50, 75, 100]* will win the competition!

Rules

The **AI agent you design** will choose which **cell** out of a set of **candidate cells** to select next. Remember, we wish to choose the cells that allow us to find (1) the *shortest path (in terms of total cost)* in (2) the *smallest number of iterations*. There is a separate agent already written which will choose the shortest path out of all the cells you have explored. Thus, as long as you select *all the cells contained in the shortest path* by the time your agent reaches the exit, you will get a full score for **path_grade**.

You will write your code in the agent.py file of the codebase, specifically the **select_action** function.

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

There are 4 inputs and 1 output to this function:

- **candidate_cells**: a **set** of **(x,y,cost) tuples** that are adjacent to the currently explored cells.
- **last_cell**: the last **(x,y,cost) tuple** which was chosen.
- **adjacent_cells_to_last_cell**: the **(x,y,cost) tuples** that are adjacent to **last_cell** (i.e. not walls).
- **iteration**: the current iteration we are on.

- **output**: any **(x,y,cost)** tuple in **candidate_cells**.

You may also modify the constructor by adding any data structures you like.

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

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

Notice, you also have access to the size of the current maze in **self.N**, and maximum number of iterations in **MAX_ITERS**.

Have Fun and Good Luck!

Testing

To test the program, simply run **$python3 main.py**. The final score as well as the score breakdown for each section will appear in the terminal output. There will also be animated gifs of each run in the **results** directory of your root folder. Notice, the actual competition will use different mazes than the default test 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.

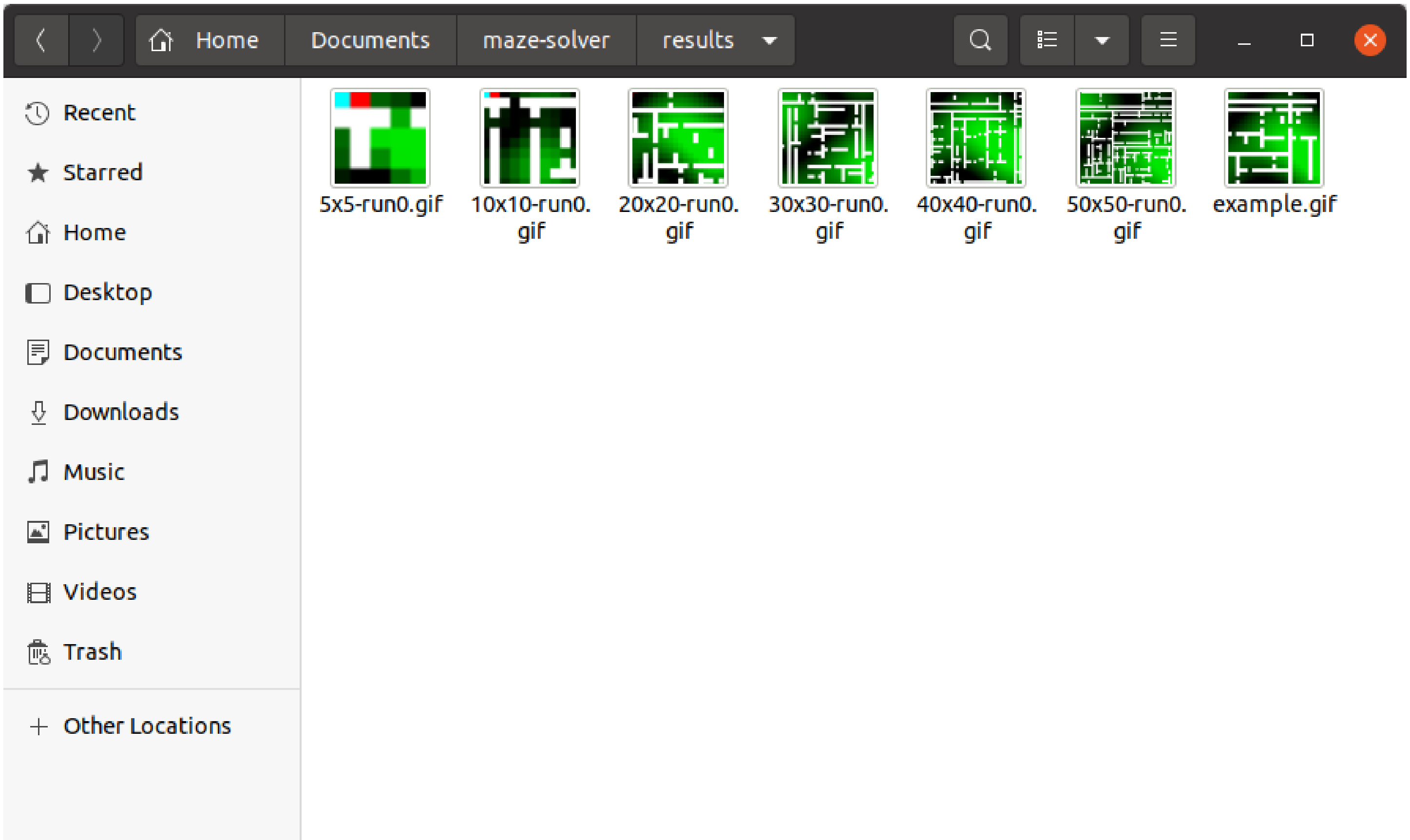

Figure 2: Example of the gifs.

Maze Traversal AI Project: Appendix

Hint 1: Notice, the current codebase just 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 first select *cells that may belong to the path with the least cost*. This is called **Djikstra's Algorithm**. To do so, we need to keep track of the cumulative cost required to reach each candidate cell in a dictionary.

```
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 every iteration, we can update this dictionary by computing the <u>cost it takes to reach adjacent cells from the one we just visited</u>. Notice the set of candidate cells at the current iteration must have been in a set of all adjacent cells from previous iterations. Thus, we can directly select the cell with the <u>smallest cumulative cost</u> from our dictionary as our action.

```
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
```

Djikstra's Algorithm *guarantees we will find the shortest path* before reaching the exit of the maze.

Hint 3: It seems on larger mazes, Djikstra's algorithm fails to complete within **MAX_ITERS** iterations. How can we speed up Djikstra's Algorithm? Perhaps if we *sacrifice the optimality of our path*, we can focus on *finding the action that will get us closest to the exit*. One simple method is to compute the <u>L1 distance from the candidate cells to the exit</u> and select the cell with the <u>smallest L1 distance</u>. This is called **Greedy Best First Search**, as shown below.

```
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
```

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