

Coding Competition: Maze Traversal AI

UCLA CSSI

Introduction

In this project, we will design an AI to find the shortest path through an $N \times N$ 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 $(0,0)^{\text{th}}$ cell, the AI iteratively selects new cells until it reaches the bottom right corner, the $(N-1,N-1)^{\text{th}}$ 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 $(N-1,N-1)^{\text{th}}$ cell.

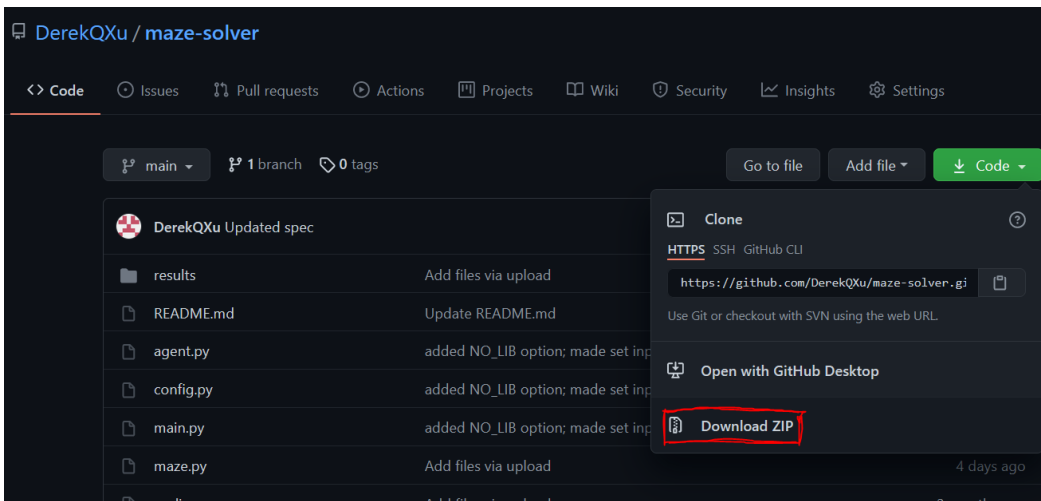
Once the AI is finished, I will compute the **shortest path** from $(0,0)$ to $(N-1, N-1)$ out of all cells your AI had explored. The goal is to (1) find the shortest path in (2) the least number of iterations. An example of this process is shown below.



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.

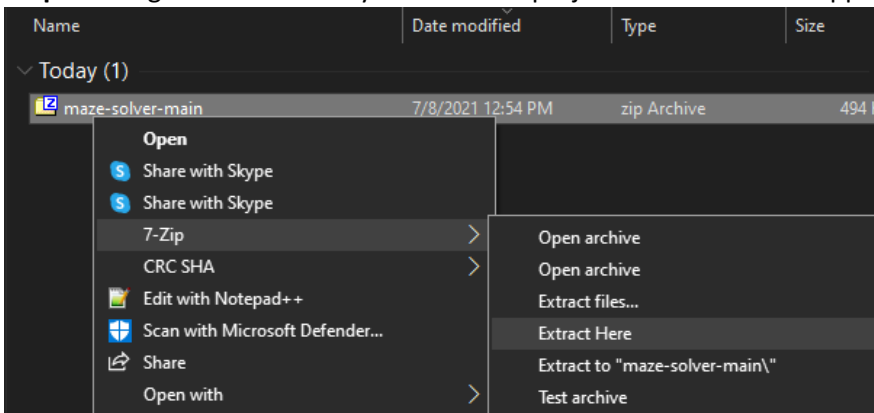
Installation

Click Code -> Download as ZIP from: <https://github.com/DerekQXu/maze-solver>.

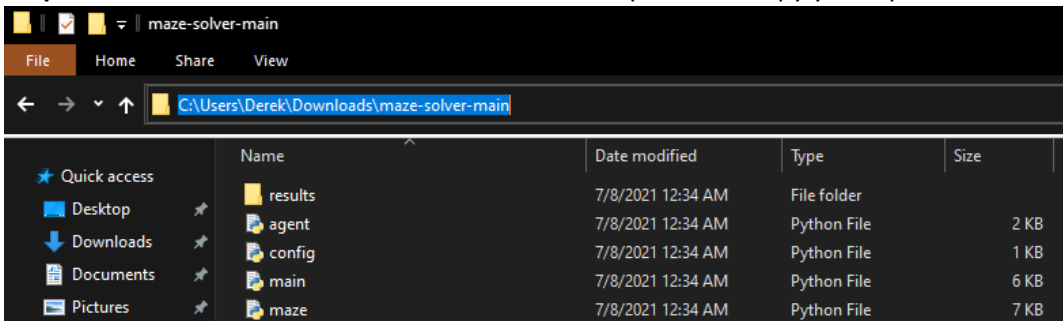


Windows

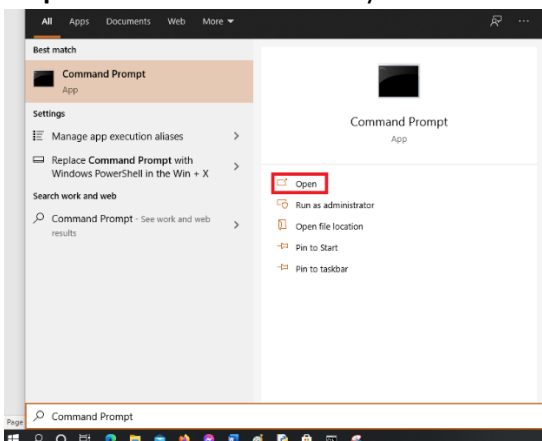
Step 1. Navigate to wherever you saved the project and extract the zipped folder.



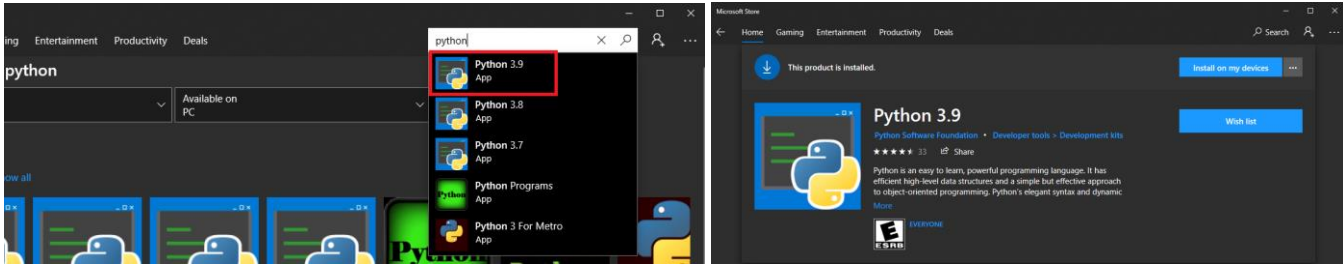
Step 2. Enter the extracted folder and click on the top bar and copy your <path>.



Step 3. Press the Window's key and search for and open Command Prompt.



Step 4. Type `python3` into Command Prompt and press Enter. **If it opens Microsoft Store**, install python from the Microsoft store. **If it gives you an error**, press the Window's key, search for and open Microsoft store, search for **python** in the Microsoft store, and install it. **If you have python**, close python by typing `exit(-1)` and press Enter.



Step 5. Type `cd <path>` into Command Prompt and press Enter.

```
C:\Users\Derek>cd C:\Users\Derek\Downloads\maze-solver-main
```

Step 6. Type `pip3 install -r requirements.txt` and press Enter.

```
C:\Users\Derek\Downloads\maze-solver-main>pip3 install -r requirements.txt
Collecting numpy==1.19.5
  Downloading numpy-1.19.5-cp39-cp39-win_amd64.whl (13.3 MB)
    | 13.3 MB 3.3 MB/s
Collecting Pillow==8.3.0
  Downloading Pillow-8.3.0-cp39-cp39-win_amd64.whl (2.3 MB)
    | 2.3 MB 3.3 MB/s
Collecting tqdm==4.61.1
  Using cached tqdm-4.61.1-py2.py3-none-any.whl (75 kB)
Installing collected packages: tqdm, Pillow, numpy
WARNING: The script tqdm.exe is installed in 'C:\Users\Derek\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
WARNING: The script f2py.exe is installed in 'C:\Users\Derek\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed Pillow-8.3.0 numpy-1.19.5 tqdm-4.61.1
```

Step 7. Type `python3 main.py` and press Enter to check if the program runs.

```
C:\Users\Derek\Downloads\maze-solver-main>python3 main.py
generating maze of size 5 (1/1)
maze generated!
1% | 15/1500 [00:00<?, ?it/s]
score: 93.6
  breakdown: completion_score:30.0,exploration_score:3.5999999999999996,path_score:40.0,path_score_bonus:20.0
  animating...
=====
generating maze of size 10 (1/1)
maze generated!
5% | 70/1500 [00:00<00:00, 8748.55it/s]
score: 92.9
  breakdown: completion_score:30.0,exploration_score:2.9,path_score:40.0,path_score_bonus:20.0
  animating...
=====
generating maze of size 20 (1/1)
maze generated!
16% | 243/1500 [00:00<00:00, 1472.35it/s]
score: 62.8983061734377
  breakdown: completion_score:30.0,exploration_score:3.9000000000000004,path_score:28.998306173437708,path_score_bonus:0
```

OS X

Step 1. Navigate to wherever you saved the project and double click on it to extract the zipped folder.

Step 2. Right click the **extracted “maze-solver-main folder”** and click **“copy”**. This will copy your <path> to clipboard.

Step 3. Go to **Finder/Application/Utilities/** and open **Terminal**.

Step 4. Type `python3` in Terminal. **If you do not have python**, type `brew install python3` and press Enter. **If you have python**, close python by typing `exit(-1)` and press Enter.

Step 5. Type `cd <path>` into the terminal and press Enter.

Step 6. Type `pip3 -r requirements.txt` and press Enter.

Step 7. Type `python3 main.py` and press Enter to check if the program runs.

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) 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.

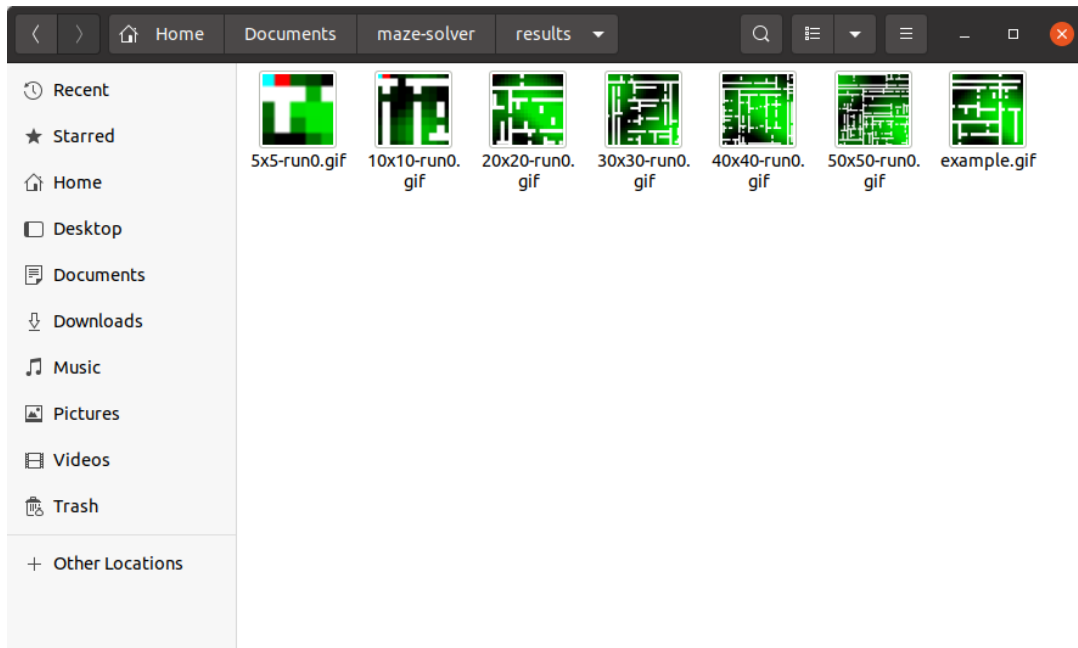


Figure 2: Example of the gifs.

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]**'s. The **select_action(...)** function will return **one cell** from this list. **You will decide which [x,y,cost] 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]**'s that are adjacent to the currently explored cells.
- **last_cell** (i.e. cyan cell): the last **[x,y,cost]** which was chosen.
- **adjacent_cells_to_last_cell** (i.e. terrain cells adjacent to cyan cell): a list of **[x,y,cost]**'s adjacent to last_cell.
- **return value**: any **[x,y,cost]** in **candidate_cells**.

You may not change the import statements at the beginning of **agent.py**. You may modify the **__init__(...)** function by adding new variables (see Hint 3). 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!*

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. You can access **N** in the **self.N** variable.

```
def select_action(self, candidate_cells, adjacent_cells_to_last_cell, last_cell):
    #####
    min_distance = 99999999 #big number
    action = candidate_cells[0] #default action

    # select action with smallest l1 distance from origin
    for cell in candidate_cells:
        x, y = cell[0], cell[1]
        num_cells_from_origin = self.N - 1 - x + self.N - 1 - y
        if num_cells_from_origin < min_distance:
            min_distance = num_cells_from_origin
            action = cell
    #####
```

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 need to keep track of **the cost required to reach each cell** throughout search, even from previous calls to `select_action(...)`. Thus, we must define a member variable, which is like **a global variable** but only accessible by functions declared under the same class (ex. class Agent). To use a member variable, `x`, we write `self.x`. We declare the new member variables in the `__init__(...)` function.

We define **2 member variable lists** representing **the cost required to reach each cell**. Think of this similarly to a banking application, with a “persons” list and a “savings” list, where the *ith* person has the *ith* saving. Here, we have a `cell_li` and `cost_li`, where the *ith* cell will have the *ith* 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.cell_li = []
    self.cost_li = []
    #####
```

On each iteration, we can update the **2 lists** 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**.

After updating the 2 lists, we choose **the candidate cell** with the least cost.

```
def select_action(self, candidate_cells, adjacent_cells_to_last_cell, last_cell):
    #####
    # update the cell costs
    for cell in adjacent_cells_to_last_cell:
        x, y, terrain = cell[0], cell[1], cell[2]
        if len(last_cell) == 0:
            # first iteration
            self.cell_li.append([x,y])
            self.cost_li.append(terrain)
        else:
            # compute cost required to reach cell
            x_last, y_last = last_cell[0], last_cell[1]
            cost_last_cell = self.cost_li[self.cell_li.index([x_last, y_last])]
            cost_new = terrain + cost_last_cell
            if [x, y] in self.cell_li:
                # update the cell's cost if the new path is cheaper
                cost_old = self.cost_li[self.cell_li.index([x, y])]
                if cost_new < cost_old:
                    self.cost_li[self.cell_li.index([x, y])] = cost_new
            else:
                self.cell_li.append([x, y])
                self.cost_li.append(cost_new)

    # select the cell with smallest cumulative cost
    min_cost = 99999999 #big number
    action = candidate_cells[0] #default action
    for cell in candidate_cells:
        x, y = cell[0], cell[1]
        if self.cost_li[self.cell_li.index([x, y])] < min_cost:
            min_cost = self.cost_li[self.cell_li.index([x, y])]
            action = cell
    #####
```

This is called **Dijkstra'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 **Dijkstra'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?

----- end -----

Appendix

Scoring

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

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

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

- 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.

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

- 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**.

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

- 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 $N = [10, 30, 50, 100]$ will win the competition!