Homework 2

ECE 172A
Introduction to Intelligent Systems

Winter 2022

Make sure you follow these instructions carefully during submission. Not doing so may result in a significant penalty.

- All problems are to be solved using Python unless mentioned otherwise.
- Code templates and other files are available on GitHub.
- Submit your homework electronically by following the steps listed below:
  1. Upload a pdf file with your write-up on Gradescope. This should include your answers to each question and relevant code snippets. Make sure the report mentions your full name and PID. Problems without marked pages on Gradescope may receive a 0. Finally, carefully read and include the following sentences at the top of your report:
     Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind. By including this in my report, I agree to abide by the Academic Integrity Policy.
  2. Upload a zip file with all your scripts and files on Gradescope. Name this file: ECE_172A_hw2_lastname_studentid.zip. This should include all files necessary to run your code out of the box.

1 Better Robot Traversal (15 points)

In the last homework assignment, we looked at robots traversing a room using the sense-act paradigm discussed in class. One of the downsides to this approach is the restrictions it places on the space the bot is trying to traverse. In this problem, we will explore bot traversal using a more general approach. Take time to read and understand the script HW2_1.py, and modify the file as needed while working on each of the following parts.

1.1 Plotting the Potential Field

(i) Plot the 2D contour plot of the vector field.

(ii) Plot the gradient (dx, dy) as quivers on the same plot as the contour plot. Quiver plots are extremely useful for analyzing the gradients of a differentiable function. It is constructed by plotting a small arrow at every point on the domain of the function, with the arrow pointing in the direction of the gradient at that point.

(iii) Using the quiver plot as reference, indicate where the bot will begin (with a blue asterisk), and where the bot is intended to finish (with a red asterisk) on the contour quiver plot.

Provide answers to the following questions:

- Where are the obstacles in the room?
- How are obstacles in the room represented in the potential field, and why?
- What happens to the gradient as you approach the end location?

1.2 Gradient Descent

(i) Implement the gradient descent algorithm to navigate the potential field. Be sure to plot the position of the bot at every iteration.

(ii) Change the initial position of the bot to a different location within the room and run the gradient descent algorithm for the new case.

(iii) Change the position of the obstacles to different locations within the room and run the gradient descent algorithm for the new case. Make sure you revert the initial position of the robot back to the original location.

Answer the following questions:

- Why is this method better than the sense-act paradigm?
- Explain how the algorithm works and why it allows the bot to avoid obstacles.
- Is this robot an intelligent system? Why or why not?

2 Swarms (15 points)

As robots get smaller, the ability to maintain a large number of them becomes easier and easier. Increasing developments in machine intelligence have had a significant effect on the development of robotic swarms. In this problem, we'll look at the advantages of intelligent swarms versus non-intelligent ones.

Your task is to implement an efficient swarm of robots. In HW2_2.py, the bots are provided with the locations of all the walls. We will be using the function a_star, which will calculate a route for a bot to travel without colliding into a wall, to calculate our new position. You do not need to worry about anything inside the a_star function.

After reading through the code, write the following functions:

- get_unexplored_areas
- get_new_destination
- update_explore_map
- update_position

These functions are commented with "(TODO: write this function)" where called.

Variable details:

- curPos: 1x2 matrix containing the location of the bot
- dest: 1x2 matrix containing the next location the bot is headed towards (e.g. destination)
- route: Nx2 matrix containing a sequence of locations the bot takes in order reach the destination. The first row contains the position of the robot, followed in the second row by the next immediate step the bot should take. The last row contains the same location as dest.
- explore_map: a height x width matrix containing information on whether locations are:
  - walls: locations where bots cannot move to
  - unmapped: locations where bots have not visited
  - planned: locations where bots have planned to go (either en route or the final destination)
  - mapped: location where bots have visited

In your report, briefly describe your algorithm. After you implement the algorithm, run your code for number of bots = 5, 10, and 15. Report the number of iterations it takes to explore the whole map in each of these cases. You may use loops in your code if necessary. (Take a look at exampleOutput.avi to get an idea about what you're working towards.)

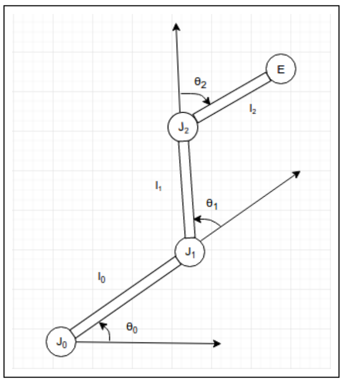

Figure 1: 2-D robotic arm

3 Robot Kinematics (15 points)

In this problem, we aim to design a simple 2-D robotic arm and learn how to control its movement. Figure 1 shows the robot to be designed. It has 3 links of lengths l_0, l_1, and l_2 that are free to rotate in the 2-D plane about the three joints J_0, J_1, and J_2. The joint J_0 remains fixed at the origin. The end point of the last link of the robot is called the 'end effector' E. We can control our robotic arm by varying the joint angles θ_0, θ_1, and θ_2

3.1 Forward Kinematics

Your first task will be to find out where the end effector and the joints of your robot arm move as you control the joint angles, called 'Forward Kinematics.

Write a Python function *forwardKinematics* that takes as inputs the joint angles θ_0, θ_1, θ_2 and the link lengths l_0, l_1, l_2 and outputs the positions of the J_1, J_2 and E. Once you have the joint positions, you can use *drawRobot* for visualizing the robot arm. Try varying the values of theta and l_1, l_2, l_3 and observe what happens to the robot.

In your report, include images of your robotic arm for:

1. $\theta_0 = \pi/3$, $\theta_1 = \pi/12$, $\theta_2 = -\pi/6$, $l_1 = 3$, $l_2 = 5$, $l_3 = 7$
2. $\theta_0 = \pi/4$, $\theta_1 = \pi/4$, $\theta_2 = \pi/4$, $l_1 = 3$, $l_2 = 5$, $l_3 = 2$

3.2 Inverse Kinematics

Now imagine that your robotic arm is to be deployed in an imaginary 2-D factory to carry objects at given locations. This requires the end effector of the robot to be matched to the location of the object of interest. This falls under 'Inverse Kinematics'. Essentially, you need to find the angles θ_0, θ_1, θ_2 for a given position (x_{target}, y_{target}) of the end effector.

Write a function *inverseKinematics* that takes as inputs the target end effector position (x_{target}, y_{target}) and link lengths l_0, l_1, l_2, and outputs the angles θ_0, θ_1, θ_2 that will lead to that end effector position. The comments in the code will walk you through the 'Jacobian method' for inverse kinematics.

Now, using your function, find the values of θ_0, θ_1 and θ_2 for $l_1 = l_2 = l_3 = 10$ and $(x_{target},\ y_{target}) = (6,\ 12)$. Try changing the initialization of the θ values in the function and see if it leads to different solutions. In particular try these two initializations:

1. $\theta_0 = \pi/6$, $\theta_1 = 0$, $\theta_2 = 0$
2. $\theta_0 = \pi/3$, $\theta_1 = 0$, $\theta_2 = 0$

In your report, include:

- Images of your robot for the solutions obtained with each of the two initializations.
- A plot showing the end effector positions through all the iterations. (There will be two plots, one for each initialization).
- Your code for *inverseKinematics* and any other function that you may write.

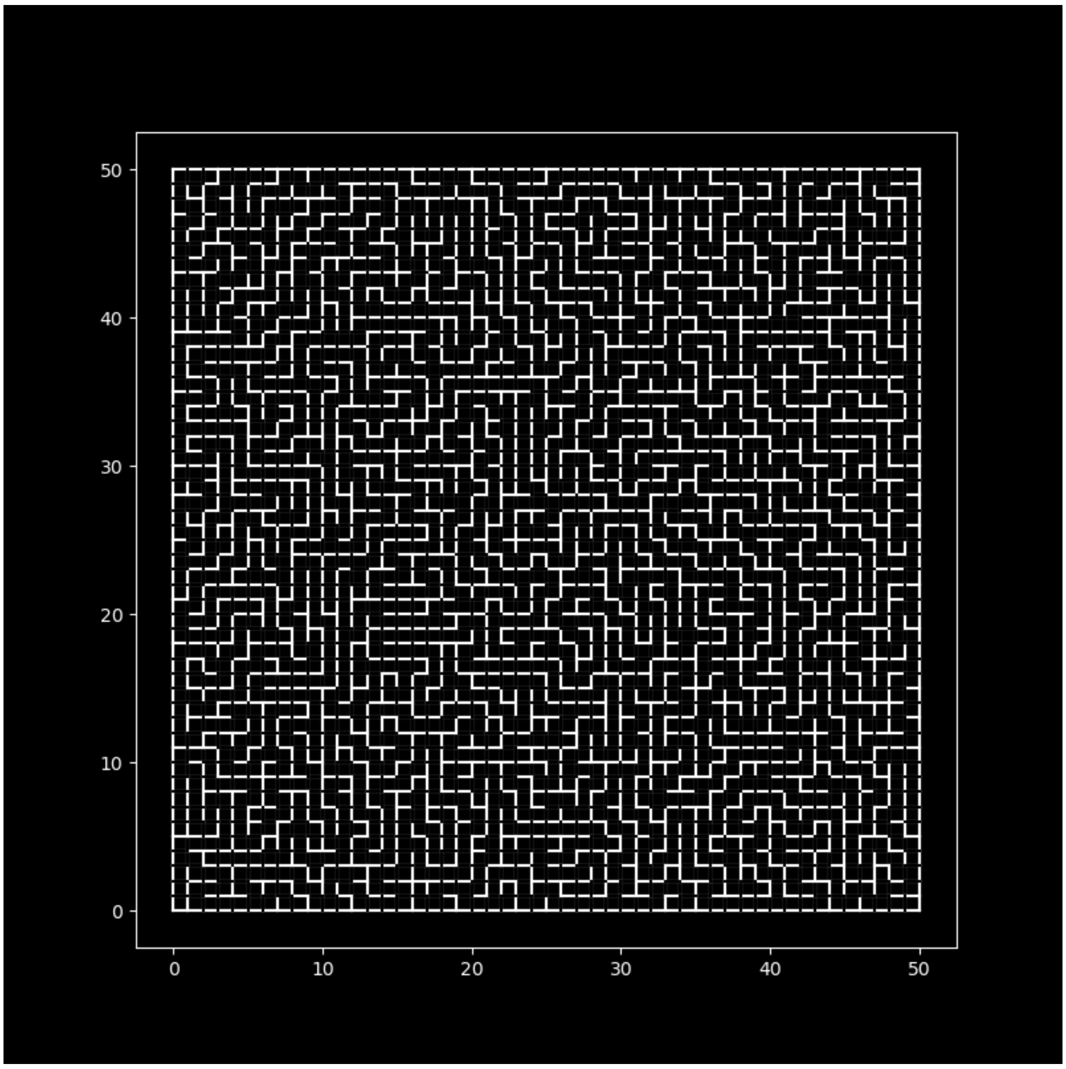

Figure 2: Maze for Problem 4. Begin at (0, 0) [bottom left], and end at (49, 49) [top right].

4 Maze Pathfinding

In this problem, you will implement path-finding algorithms in order to solve the maze shown in Figure 2.

Depth first search traverses the maze by selecting a direction at every node in the maze and proceeding until a dead end. Once it can't continue down a path, the algorithm will return to the most recent decision point and select another route. It does this until there are no more routes to explore or the goal has been reached.

The natural n-dimensional indexing of a matrix is $[idx_1, idx_2, ..., idx_n]$. For 2-dimensional matrices, this is $[row, col]$. The other way is called *linear* indexing where every $[row, col]$ pair gets a linear index. For example, in the maze above, we can label the starting position as $[0, 0]$ or we can use linear indexing and call it index 0. It is much easier to implement this search in linear indexing, where N is the total number of linear indices $num_{rows} * num_{cols}$.

In order to implement DFS, we utilize the Last-In-First-Out nature of a Stack (shown below). We will be *pushing* elements onto the stack and *popping* the last element that was pushed.

1. **Depth First Search**

   (a) Implement the DFS algorithm below and plot the cursor position every iteration. You can use the pop() and append() functions associated with the default Python list type to represent the stack.

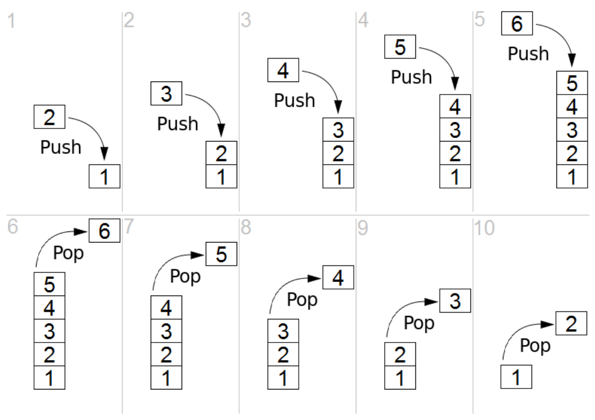

Figure 3: *push* and *pop* functionality of the stack

2. **Breadth First Search (BFS)**

   (a) Implement the BFS algorithm for the pathfinding. You can use pop(0) and append() functions associated with the default Python list type to represent the stack.

   (b) Briefly describe the BFS algorithm. What is the difference between the DFS and BFS algorithm?

   (c) Run the DFS and BFS algorithms on the full maze. How many iterations does it take to find the exit? Which algorithm works better in this problem?

Algorithm 1 Depth First Search

```
 1: Initialize an empty stack
 2: Initialize an $N \times 1$ array of zeros to keep track of the visited nodes (*visited*)
 3: Initialize an $N \times 1$ array of zeros for parent nodes (*parents*)
 4: Push the starting index onto the stack
 5: **while** the stack is not empty && you have not reached your goal **do**
 6:     pop the first index, $i$, from the stack
 7:     **if** we haven't visited the node at $i$ **then**
 8:         mark it as visited ($visited(i) = 1$)
 9:         get the neighbors of node $i$ ($neighbors = sense\_maze(i,\ data)$)
10:         **for** all unvisited neighbors of the node at $i$ **do**
11:             push the neighbor onto the stack
12:             set the parent of the neighbor to the current node
```

You may use loops in this problem. In you report, include a figure of the completed completed maze with all paths the DFS search tried, and a figure for BFS. In both cases, the final path should be visualized in a different color than other paths.

The maze is represented as a dictionary, available in the pickle file **172maze2021.p**. Each key in the dictionary is a tuple representing the (x, y) coordinates of a maze cell, with its value a boolean list [N, E, S, W], representing whether there is an opening in the corresponding cardinal direction. You may make use of the visualization script provided in HW2_4_draw.py.