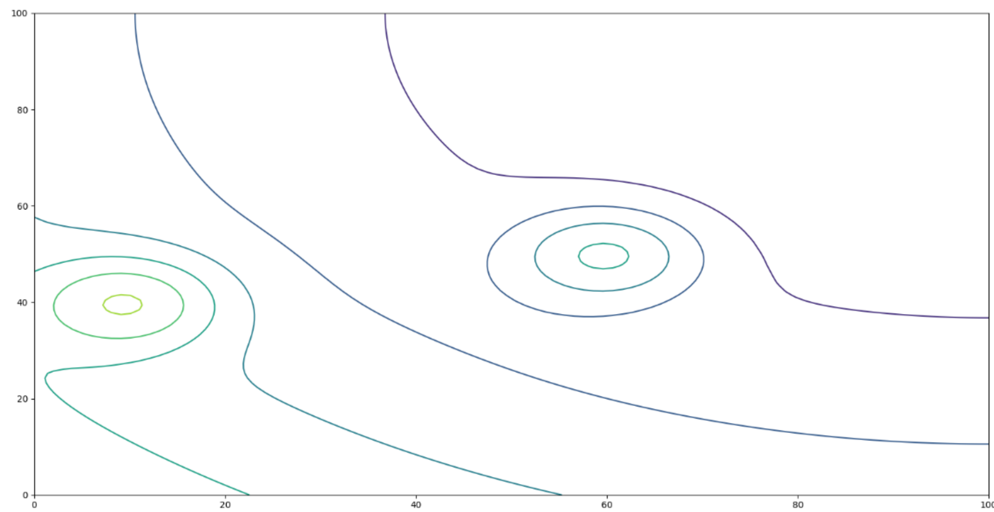
Hao Le
A15547504
ECE 172A Winter 2022 HW2

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.

1. Better Robot Traversal

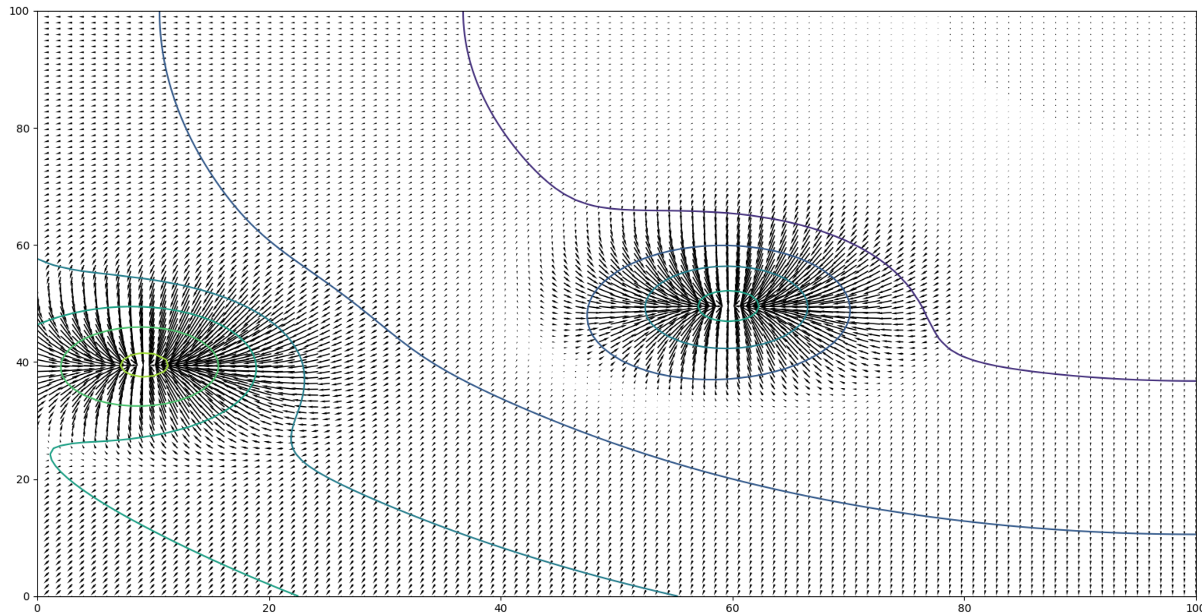
1.1

i.



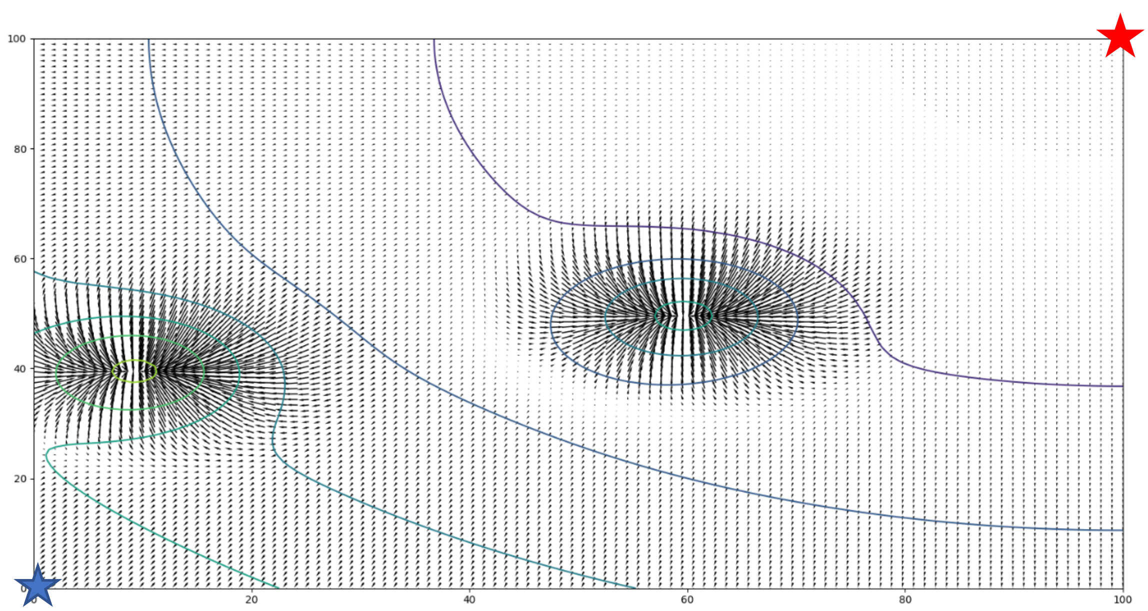

```
fig2 = plt.figure()
plt.contour(x,y,z)
```

ii.




```
dy, dx = np.gradient(z,1,1)
plt.quiver(x,y,dx,dy,width=0.001,headwidth=2) #comment this line out to toggle quiver arrows
```

iii.



There are two obstacles located at (60,50) and (10,40) determine by the *mu* variable.

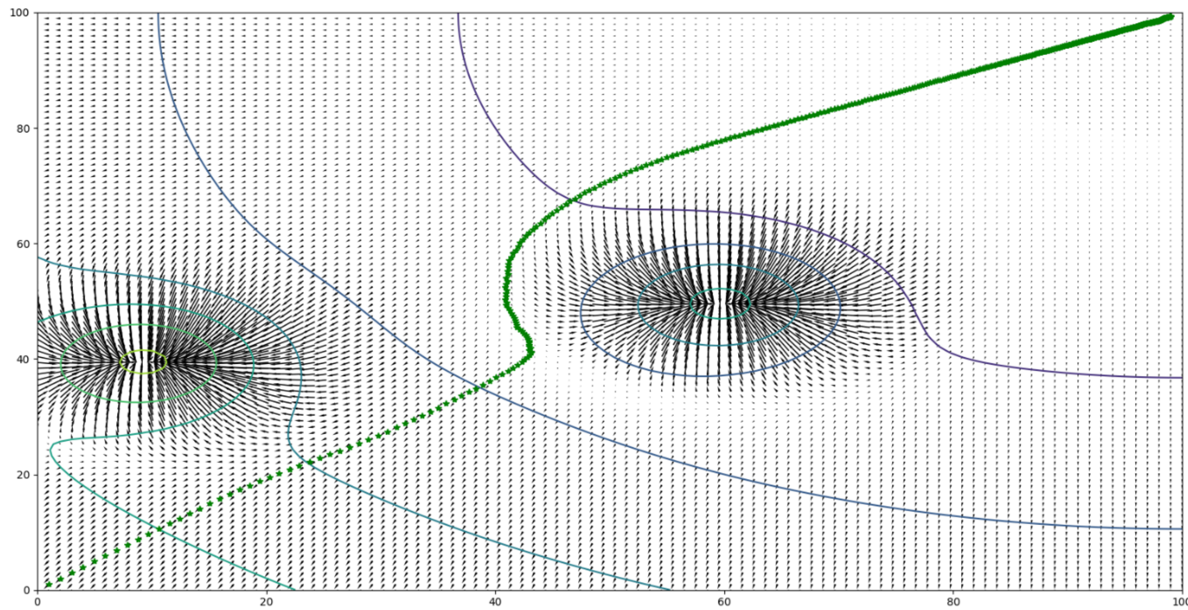
We see this on the potential field, where the quiver arrows converge and point to two single points. The potential field represent the gradient vectors at each position; gradients point towards the direction of greatest ascent. Since the obstacles are the high “peaks”, the arrows would point towards the directions to their tips, hence we see the convergence of arrows.

The gradient gets “smaller” in terms of the magnitude of the vector. In other words, there is no one particular direction that produces an ascent, so the landscape “flattens” out.

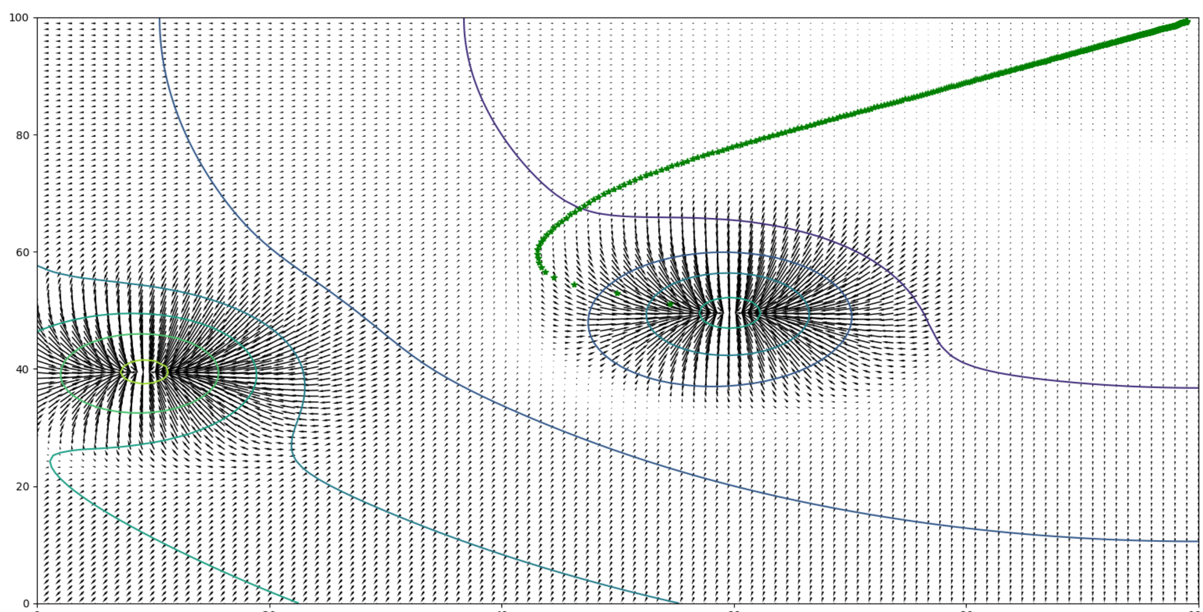
1.2

i.

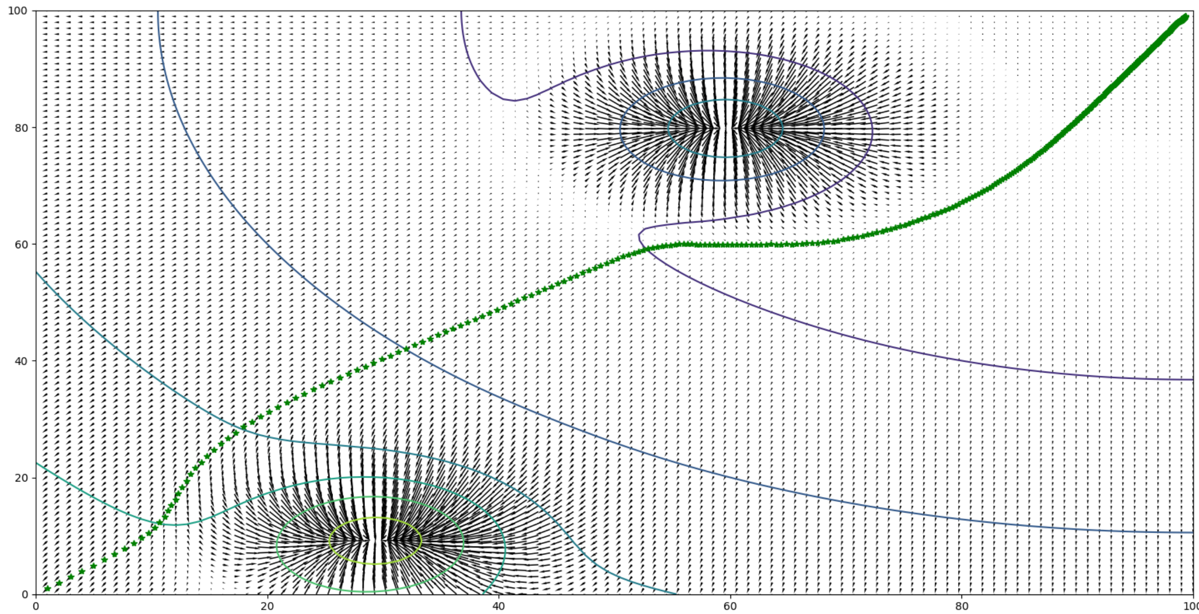
```
step_size = 100
current_loc = initial_loc.astype(float)
close_enough_metric = 0.0001

plan_path = True #change to True to plan path


if plan_path == True:
    try:

        while True:

            dx_ = -dx[(int(current_loc[1])), (int(current_loc[0]))]
            dy_ = -dy[(int(current_loc[1])), (int(current_loc[0]))]


            current_loc[0] = current_loc[0] + step_size * dx_
            current_loc[1] = current_loc[1] + step_size * dy_

            plt.plot(current_loc[0], current_loc[1], marker="*",color="g")

            if np.linalg.norm(np.array([dx_, dy_])) < close_enough_metric:
                break

    except:

        plt.show()


plt.show()
```



ii.



iii.



This method is better than a sense-act paradigm because it considers the position of obstacles as well as the uncertainties of their actual placement. This can be seen as Gaussian distributions of the two obstacles, where their means represent their expected positions. Unlike a reactive paradigm that must first collide with obstacles to change trajectory, this deliberative paradigm helps the robot plan its motion and potentially avoid damage caused by obstacle collision.

The gradient descent algorithm works by following the direction of greatest descent informed by the negation of the gradient vector. This will lead the robot to increment its position towards the nearest local minimum, analogous to a ball rolling into a pit.

in this case because there are no local minimums for the robot to traverse into and never ascend back out. Second, since the goal position is on the top right corner, this is also where the gradient has smallest magnitude i.e., the overall shape of the plot “funnels” into the corner, so the robot will always “roll” into that corner. Obstacles in this case are hills and need great ascent. Since the robot follows directions opposite to ascent, it will never go up these hills, or even try to go near them, much like how a ball will never go back up a hill – hence, the path will avoid these obstacles.

This system is intelligent because it uses information from its surrounds to inform and reason its path planning process, to ensure that the path is safe and not detrimental to its existence. If the state of the environment changes, it will act upon those changes via changing the planned path.

2. Swarm

```python
def get_unexplored_areas(explore_map, unmapped_value):

    rows, cols = np.where(explore_map == 0)

    if not np.any(rows): #if row is empty meaning no unexplored areas
        return []

    unexplored_areas = np.zeros((len(rows),2))

    unexplored_areas[:,0] = rows
    unexplored_areas[:,1] = cols

    return unexplored_areas

def get_new_destination(current_position, unexplored_areas):

    euclidianDistances = []

    for i in range(len(unexplored_areas)):
        euclidianDistance = np.sqrt((current_position[0] - unexplored_areas[i][0])**2 + (current_position[1] - unexplored_areas[i][1])**2)
        euclidianDistances.append(euclidianDistance)

    minimumIndex = np.argmin(euclidianDistances) #find the index of the smallest distance

    return unexplored_areas[minimumIndex]
```

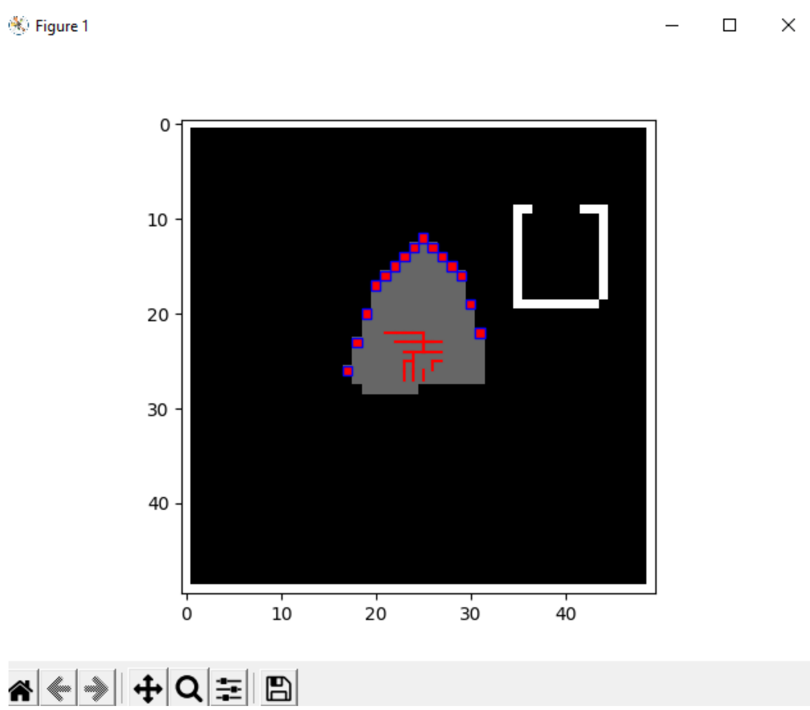
```python
def update_explore_map(dest, route, explore_map, planned, unmapped):

    for location in route:
        if explore_map[location[0],location[1]] == unmapped:
            explore_map[location[0],location[1]] = planned

    return explore_map

def update_position(curPos, route, dest, explore_map, mapped):

    curPos = route[1]

    explore_map[curPos[0],curPos[1]] = mapped

    if np.array_equal(dest,curPos):
        dest = []

    route = np.delete(route, 0, 0)

    return curPos, route, dest, explore_map
```

This swarm algorithm uses the collective power of many agents to quickly map out an environment with walls. First, we instantiate an arbitrary amount of bots that have the capability to read the current state of the map: its current location, location of walls, and location of mapped and unmapped areas. Next, the algorithm for each iteration iterates through the list of instantiated robots i.e. robot 1 goes first, then robot 2 and so on.

For every iteration, every robot checks for the nearest unmapped spot on the map, provided that it does not have an existing destination already. The robot uses A* to plan an efficient route to the nearest unmapped spot. At this point, the robot will have a destination and a route. On the same iteration, it takes 1 step closer to the destination via the route, simultaneously updating the explore map to mark its previous position as mapped. The algorithm repeats this for all the other bots. Effectively, each bot takes its turn to advance one step to its current or a new unmapped destination – this ensures that each bot has a unique unmapped destination.

If there are no more unmapped areas on the map, the bot stops.

When the mapped spots + number of walls equals the area of the map, this means everything has been mapped, and the algorithm stops.



Number of bots : iterations for complete mapping

5 : 528
10: 277
15: 211

3. Robot Kinematics

3.1

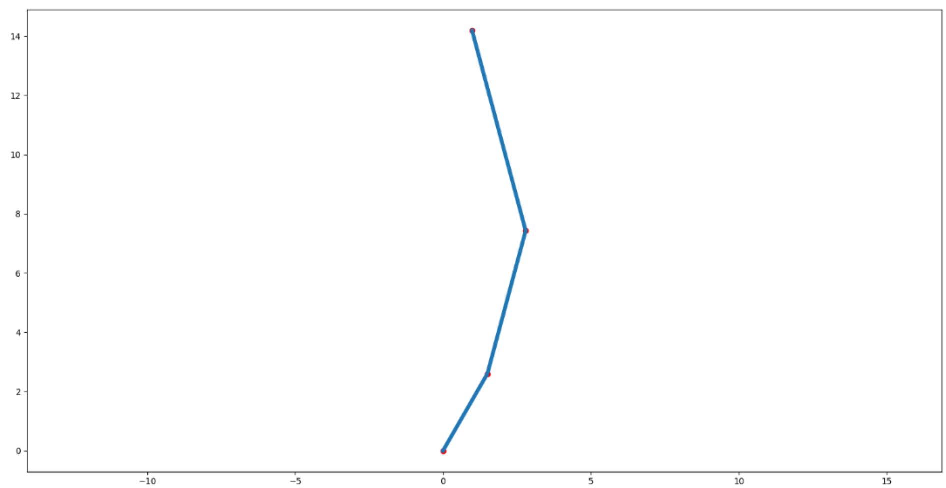
```python
def forwardKinematics(theta0, theta1, theta2, l0, l1, l2):

    T_2E = np.array([l2, 0]) #translation to get EE frame to J2 frame
    T_12 = np.array([l1, 0]) #translation to get J2 frame to J1 frame
    T_01 = np.array([l0, 0]) #translation to get J1 frame to J0 frame

    R_12 = createRotationMatrix(-theta2) #clockwise rotation about J2; J2 frame to J1 frame
    R_01 = createRotationMatrix(theta1) #anticlockwise rotation about J1; J1 frame to J0 frame
    R_G0 = createRotationMatrix(theta0) #anticlockwise rotation about J0; J0 frame to global frame

    #get position of E in global frame

    P_E_1 = np.matmul(R_12,T_2E) + T_12 #position in J1 frame
    P_E_0 = np.matmul(R_01,P_E_1) + T_01 #position in J0 frame
    P_E_G = np.matmul(R_G0,P_E_0) #position in global frame

    #get position of J2 in global frame

    P_J2_0 = np.matmul(R_01,T_12) + T_01 #position in J0 frame
    P_J2_G = np.matmul(R_G0,P_J2_0) #position in global frame

    #get position of J1 in global frame

    P_J1_G = np.matmul(R_G0,T_01) #position in global frame

    return [P_J1_G[0], P_J1_G[1], P_J2_G[0], P_J2_G[1], P_E_G[0], P_E_G[1]] #return global positions of joints and EE



def createRotationMatrix(theta): #rotation clockwise about origin, in radians
    return np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(theta)]])
```

```python
jointPositions = forwardKinematics(pi/3, pi/12, -pi/6, 3, 5, 7)
print(jointPositions)
drawRobot(jointPositions[0], jointPositions[1], jointPositions[2], jointPositions[3], jointPositions[4], jointPositions[5])

jointPositions = forwardKinematics(pi/4, pi/4, -pi/4, 3, 5, 2)
print(jointPositions)
drawRobot(jointPositions[0], jointPositions[1], jointPositions[2], jointPositions[3], jointPositions[4], jointPositions[5])
```

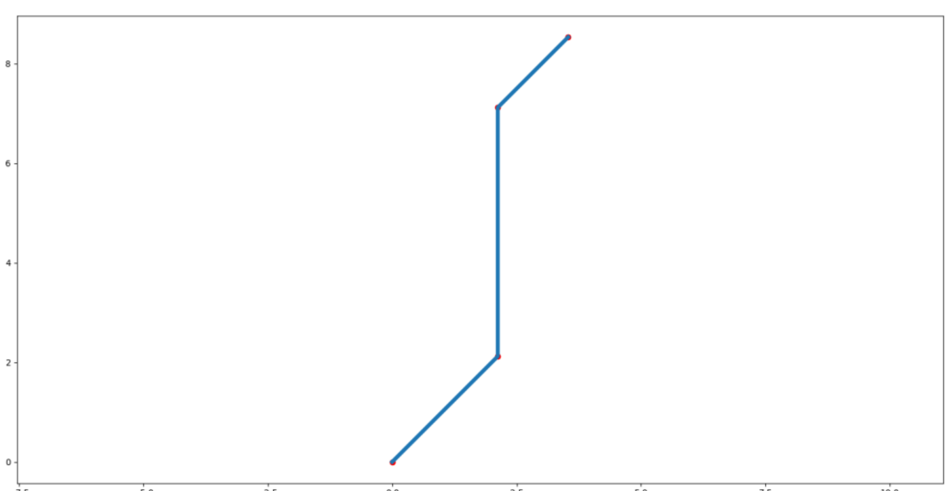
I used a matrix approach and found the individual transformations across each frame of reference. Then each joint position can be found by compositing the transforms of the frames that come before each joint.

Main script is used to first calculate the 3 joint positions in the global frame, print out the positions, and visualize them. The structure of the log is as follows: [J1_x, J1_y, J2_x, J2_y, E_x, E_y]

```
jointPositions = forwardKinematics(pi/3, pi/12, -pi/6, 3, 5, 7)
print(jointPositions)
drawRobot(jointPositions[0], jointPositions[1], jointPositions[2], jointPositions[3], jointPositions[4], jointPositions[5])

jointPositions = forwardKinematics(pi/4, pi/4, -pi/4, 3, 5, 2)
print(jointPositions)
drawRobot(jointPositions[0], jointPositions[1], jointPositions[2], jointPositions[3], jointPositions[4], jointPositions[5])
```




```
[1.5000000000000004, 2.598076211353316, 2.7940952255126046, 7.4277053427986575, 0.9823619097949603, 14.189186126822136]
```




```
[2.121320343559643, 2.121320343559643, 2.1213203435596424, 7.121320343559644, 3.5355339059327373, 8.535533905932738]
```

3.2

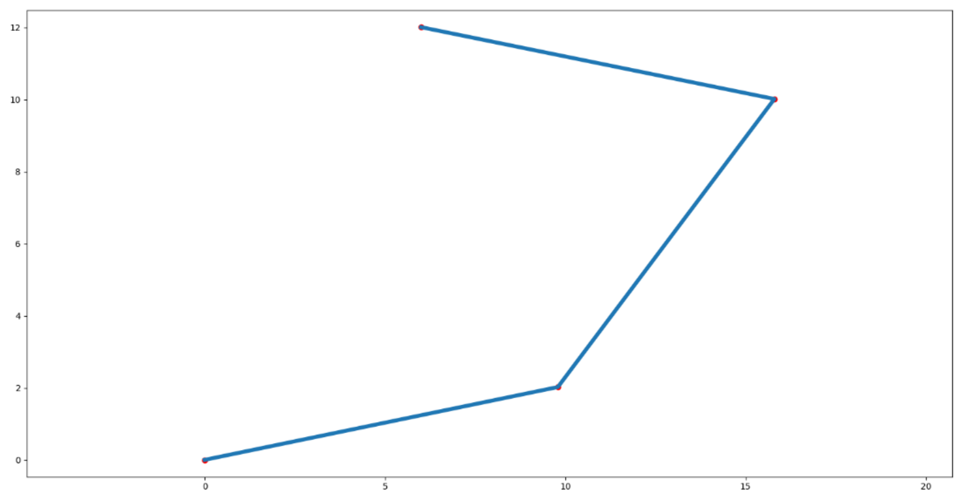
```python
def inverseKinematics(l0,l1,l2,x_e_target,y_e_target,thetas):

    jointPositions = forwardKinematics(thetas[0],thetas[1],thetas[2],l0,l1,l2)

    closeEnoughDistance = 0.01

    targetEEPosition = np.array([x_e_target,y_e_target])
    currentEEPosition = np.array([jointPositions[4],jointPositions[5]])

    stepSize = 0.1

    e_x = []
    e_y = [] #keep track of end effector position

    while np.linalg.norm(targetEEPosition - currentEEPosition) > closeEnoughDistance: #if distance is smaller than "close enough" metric, stop

        e_x.append(jointPositions[4])
        e_y.append(jointPositions[5]) #record down starting EE pos as well as the subsequent iterations

        EEPositionIncrement = (targetEEPosition - currentEEPosition) * stepSize  #get delta e in the direction of target

        jacobian = calulateJacobian(thetas[0],thetas[1],thetas[2],l0,l1,l2)
        pinvJacobian = np.linalg.pinv(jacobian)

        deltaJointAngles = np.matmul(pinvJacobian,EEPositionIncrement) #get delta thetas to move EE in that direction

        thetas = thetas + deltaJointAngles #change thetas

        jointPositions = forwardKinematics(thetas[0],thetas[1],thetas[2],l0,l1,l2)
        currentEEPosition = np.array([jointPositions[4],jointPositions[5]]) #calculate new position of EE


    jointPositions = forwardKinematics(thetas[0],thetas[1],thetas[2],l0,l1,l2) #get the final state of the robot with the calculated thetas

    drawRobot(jointPositions[0], jointPositions[1], jointPositions[2], jointPositions[3], jointPositions[4], jointPositions[5])
    plt.scatter(e_x, e_y)
    plt.show()

    return thetas
```

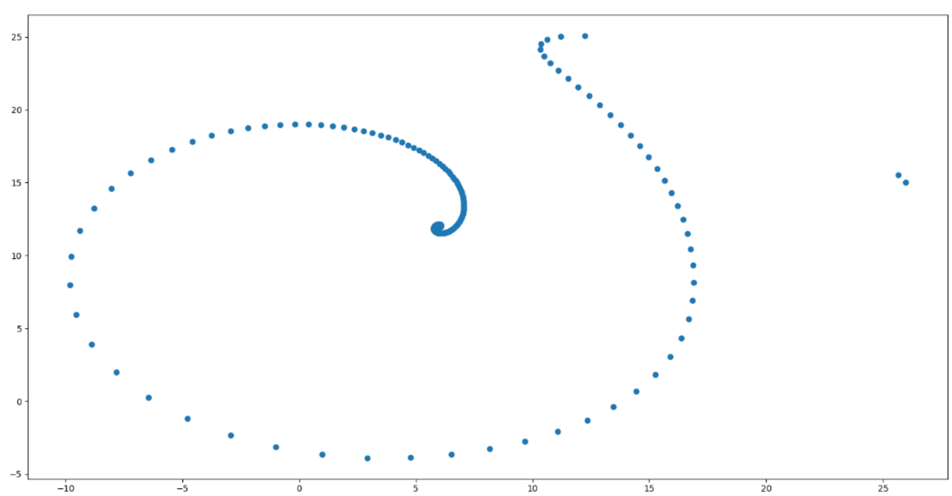
```python
def calulateJacobian(theta0, theta1, theta2, l0, l1, l2): #analytically calculated by algebraically getting EE position in terms of thetas and l's. Then doing partial derivates.

    dx_theta0 = - l0 * np.sin(theta0) - l1 * np.sin(theta0 + theta1) - l2 * np.sin(theta0 + theta1 + theta2)
    dx_theta1 = - l1 * np.sin(theta0 + theta1) - l2 * np.sin(theta0 + theta1 + theta2)
    dx_theta2 = - l2 * np.sin(theta0 + theta1 + theta2)

    dy_theta0 = l0 * np.cos(theta0) + l1 * np.cos(theta0 + theta1) + l2 * np.cos(theta0 + theta1 + theta2)
    dy_theta1 = l1 * np.cos(theta0 + theta1) + l2 * np.cos(theta0 + theta1 + theta2)
    dy_theta2 = l2 * np.cos(theta0 + theta1 + theta2)

    jacobian = np.array([[dx_theta0,dx_theta1,dx_theta2],[dy_theta0,dy_theta1,dy_theta2]])

    return jacobian
```

The main script for actual calculation and plotting:

```python
print(inverseKinematics(10,10,10,6,12,np.array([pi/6,0,0])))
print(inverseKinematics(10,10,10,6,12,np.array([pi/3,0,0])))
```

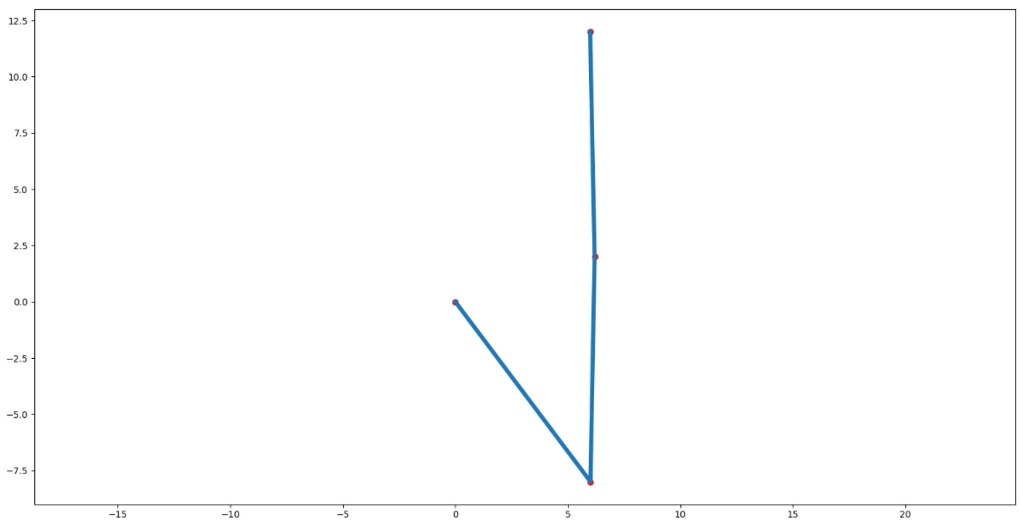
1st test case:

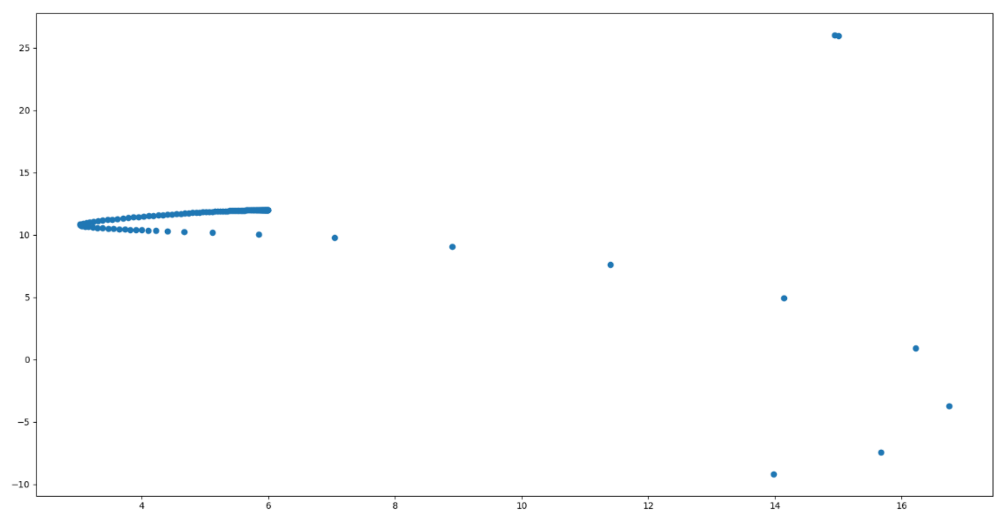





```
[-12.36287873   7.00635613  10.55206603]
```

2^{nd} test case:






```
[-95.17433571  96.72690667  94.20974224]
```

Angles are logged in radians: [theta0,theta1,theta2]

4. Maze Pathfinding

1. Depth First Search

```
class Stack:
    def __init__ (self):
        self.items = []
    def push(self, item):
        self.items.append(item) #add item to the end of list
    def pop(self):
        self.items.pop() #remove the last item in the list

class Queue:
    def __init__ (self):
        self.items = []
    def enqueue(self, item):
        self.items.insert(0,item) #add item to the start of list
    def dequeue(self):
        self.items.pop() #remove the last item in the list
```

Stack and queue classes defined. Stack is used in DFS. Queue will be used for BFS later.

Stack is like a FIFO pile, where we can add things on top of the pile, and take them off.

```python
def DFS(maze, startNode, goalNode):

    stack = Stack()

    exploredNodes = []

    stack.push(startNode)
    currentNode = None
    iterations = 0

    while len(stack.items) != 0 and currentNode != goalNode:


        currentNode = stack.items[-1]

        if not currentNode in exploredNodes:
            exploredNodes.append(currentNode)

        if maze[currentNode][0] == True and not (currentNode[0],currentNode[1]+1) in exploredNodes: #north
            newNode = (currentNode[0],currentNode[1]+1)
            stack.push(newNode)
        elif maze[currentNode][1] == True and not (currentNode[0]+1,currentNode[1]) in exploredNodes: #east
            newNode = (currentNode[0]+1,currentNode[1])
            stack.push(newNode)
        elif maze[currentNode][2] == True and not (currentNode[0],currentNode[1]-1) in exploredNodes: #south
            newNode = (currentNode[0],currentNode[1]-1)
            stack.push(newNode)
        elif maze[currentNode][3] == True and not (currentNode[0]-1,currentNode[1]) in exploredNodes: #west
            newNode = (currentNode[0]-1,currentNode[1])
            stack.push(newNode)
        else:
            stack.pop()

        iterations = iterations + 1

    if len(stack.items) == 0:
        print("no path found")
        return 1
    else:
        print("path found")
        print("\n")
        print(iterations)
        print("iterations")
        return stack.items, exploredNodes
```

DFS uses a stack to keep track of the latest parent node; since things that come in also come out first, it will traverse deeper and deeper down the child nodes until a dead end is reached. Then, removing the previous nodes off the stack, it will keep on doing so until it has reached a node with another possible path, repeating the process. This produces a greedy behavior. When it has reached the goal node, the items currently in the stack represent the actual shortest path, since there are no “branches”. The algorithm always takes one path, not leaving another for “later.”

2. Breadth First Search

```python
def BFS(maze, startNode, goalNode):

    queue = Queue()

    exploredNodes = []

    queue.enqueue(startNode)
    currentNode = None
    iterations = 0

    adjacentNodes = {}

    while len(queue.items) != 0 and currentNode != goalNode:

        currentNode = queue.items[-1]

        if not currentNode in exploredNodes:
            exploredNodes.append(currentNode)

        if maze[currentNode][0] == True and not (currentNode[0],currentNode[1]+1) in exploredNodes: #north
            newNode = (currentNode[0],currentNode[1]+1)
            queue.enqueue(newNode)
            adjacentNodes[newNode] = currentNode
        if maze[currentNode][1] == True and not (currentNode[0]+1,currentNode[1]) in exploredNodes: #east
            newNode = (currentNode[0]+1,currentNode[1])
            queue.enqueue(newNode)
            adjacentNodes[newNode] = currentNode
        if maze[currentNode][2] == True and not (currentNode[0],currentNode[1]-1) in exploredNodes: #south
            newNode = (currentNode[0],currentNode[1]-1)
            queue.enqueue(newNode)
            adjacentNodes[newNode] = currentNode
        if maze[currentNode][3] == True and not (currentNode[0]-1,currentNode[1]) in exploredNodes: #west
            newNode = (currentNode[0]-1,currentNode[1])
            queue.enqueue(newNode)
            adjacentNodes[newNode] = currentNode

        queue.dequeue()

        iterations = iterations + 1

    if len(queue.items) == 0:
        print("no path found")
        return 1

    else:
        foundPath = []
        currentNode = goalNode

        while currentNode != startNode:
            currentNode = adjacentNodes[currentNode]
            print(currentNode)
            foundPath.append(currentNode)

        print("path found")
        print("\n")
        print(iterations)
        print("iterations")
        return foundPath, exploredNodes
```

The BFS algorithm utilizes a first in last out queue instead of a stack which DFS uses. This allows a broader searching behavior since it will first explore all its options on the frontier node. In other words, upon encountering a new unvisited node, it will add to the queue all the possible next node options, with priority starting in the north direction, and ending in the west. However, these new options will eventually be explored, but this depends on how long the queue is; first in line can be an option for another frontier node. Once a branch has reached a dead end, BFS does not backtrack to previous nodes, but instead continues regularly on the next node in line, adding its options to the back of the queue, and so on. Eventually, when the next node in line is the goal, the shortest path will have to be traced – unlike DFS, it is not simply the queue items.

We can trace back a path by keeping a dictionary of adjacent nodes. On every iteration, when a new unexplored node is found, we add it as a key, and the corresponding value is the current node. Thus, it is possible that a single node can have at most 4 adjacent nodes i.e. at most 4 keys in the dictionary can have the same value. We do this until the goal has reached.

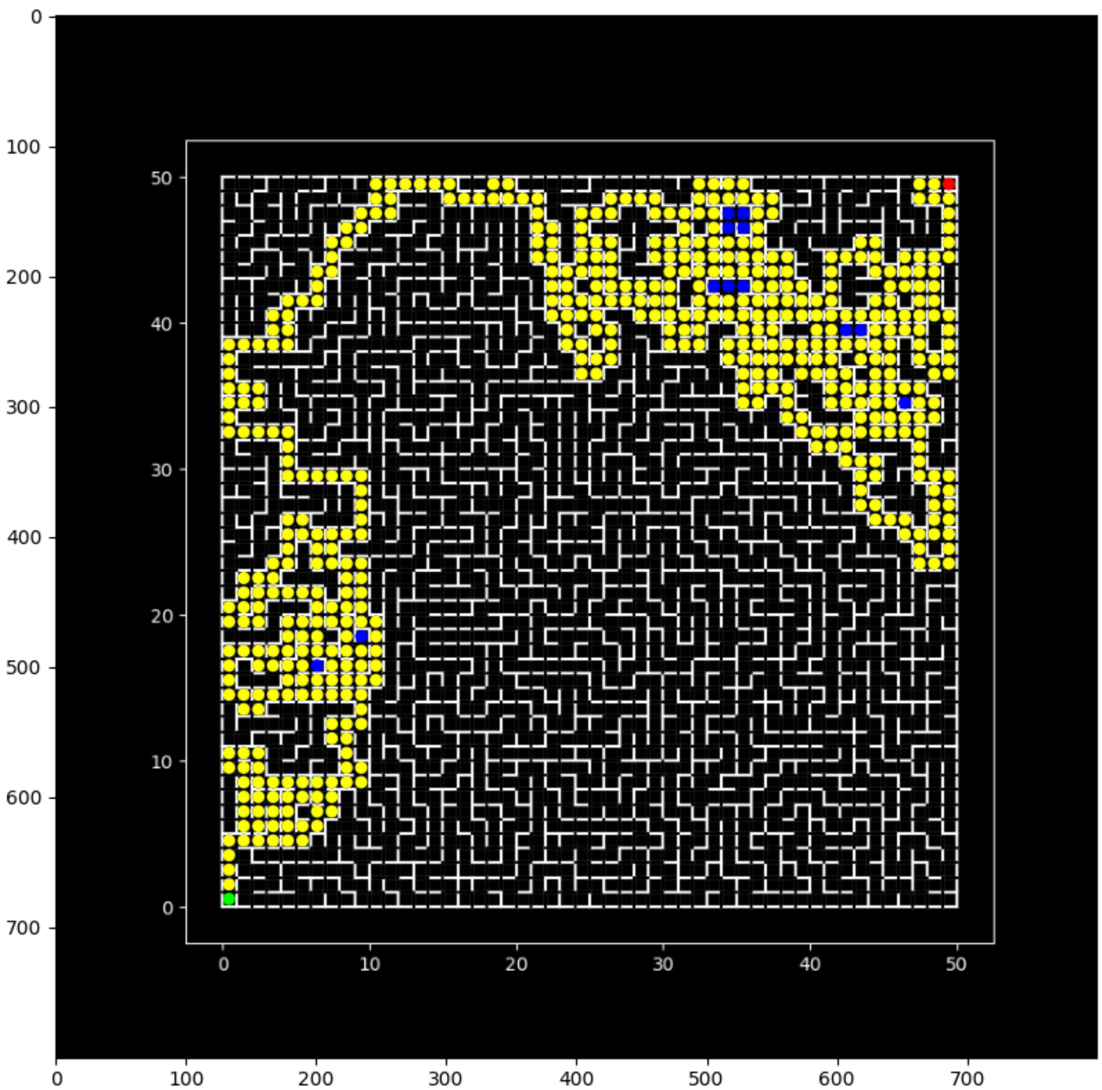
To start the trace back process, start by getting the value of the key as the node. This should point to the next node. Then use that next node as the current node and add this current node to the shortest path list and look up the value of the key as the current node. Repeat this until the value is the starting node. The shortest path list should now be valid.

The main script to calculate the shortest paths and plot them:

```python
maze = pickle.load(open("172maze2021.p","rb"))

start = (0,0)
goal = (49,49)

pathNodes, exploredNodes = DFS(maze,start,goal)
draw_path(pathNodes, exploredNodes)

pathNodes, exploredNodes = BFS(maze,start,goal)
draw_path(pathNodes, exploredNodes)
```

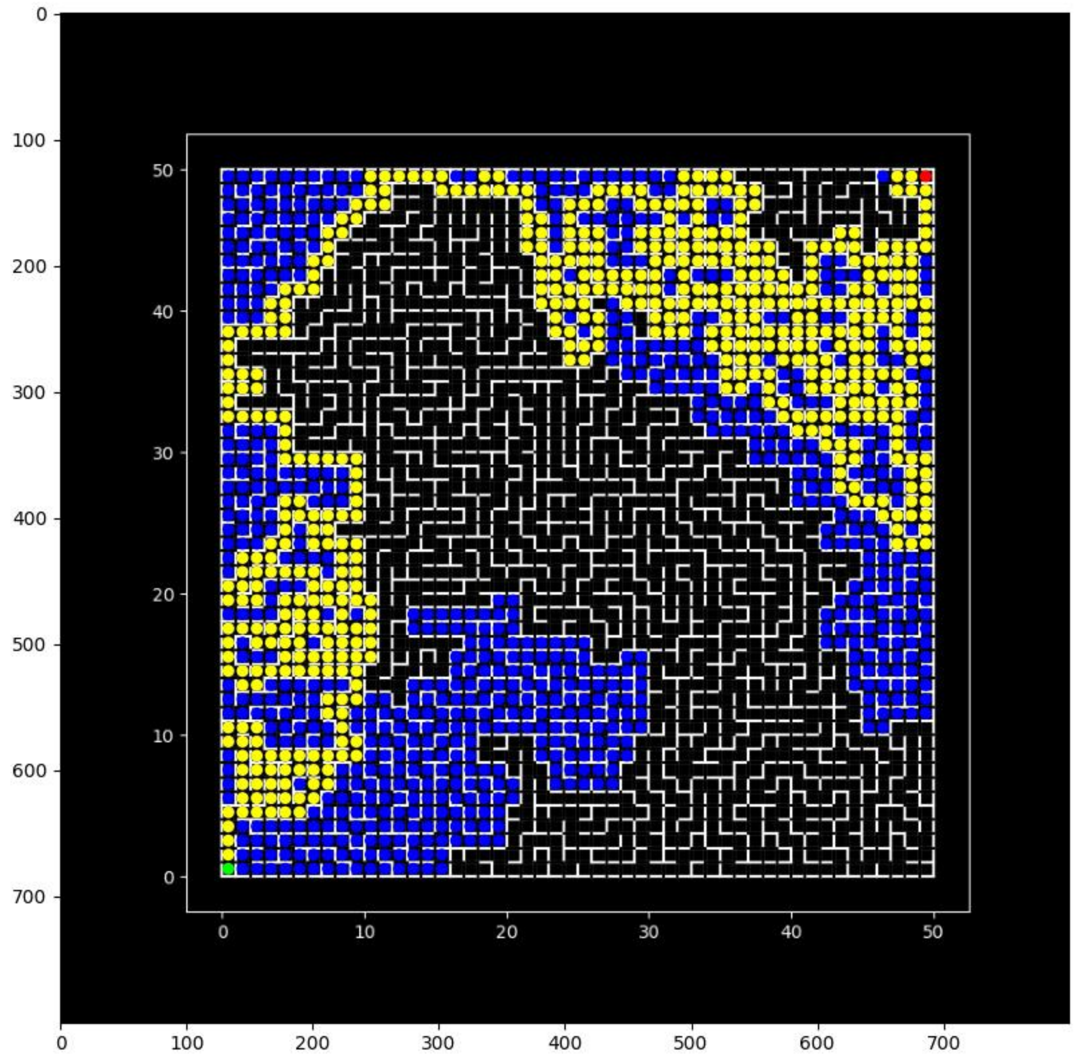
Yellow represents the shortest path, and blue represents explored areas. Iteration count is logged.

DFS:




```
537
iterations
```

BFS:




```
1263
iterations
```

DFS clearly does better than BFS on the maze because the former found the shortest path in fewer iterations; BFS spent more iterations exploring areas, as shown by the prominence of blue areas.