KAHENI PETER

P101/1740G/21

COM 405E: PRACTICAL ASSIGNMENT

1. **Maze Solving Using Greedy Best-First Search with Visualization**
   * **Question**: Implement a Greedy Best-First Search algorithm to solve a maze where 0 represents an open path and 1 represents a wall. Use the Manhattan distance as the heuristic. Visualize the explored nodes and the final path from start to goal on the grid.
   * **Input**: A 2D grid (maze), start coordinates, and goal coordinates.
   * **Output**: Display the grid showing the explored nodes, the path found, and obstacles.

The code

import matplotlib.pyplot as plt

import numpy as np

from matplotlib.colors import ListedColormap

from queue import PriorityQueue

# Heuristic function - Manhattan distance

def manhattan\_distance(x1, y1, x2, y2):

return abs(x1 - x2) + abs(y1 - y2)

# Greedy Best-First Search function

def greedy\_best\_first\_search(maze, start, goal):

rows, cols = len(maze), len(maze[0])

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {start: None}

explored = set()

while not open\_list.empty():

\_, current = open\_list.get()

if current == goal:

break

explored.add(current)

x, y = current

# Check neighbors (up, down, left, right)

for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

nx, ny = x + dx, y + dy

neighbor = (nx, ny)

if 0 <= nx < rows and 0 <= ny < cols and maze[nx][ny] == 0 and neighbor not in explored:

if neighbor not in came\_from:

priority = manhattan\_distance(nx, ny, goal[0], goal[1])

open\_list.put((priority, neighbor))

came\_from[neighbor] = current

# Reconstruct path from start to goal

path = []

if goal in came\_from:

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, explored

# Visualization function

def visualize\_maze(maze, path, explored, start, goal):

# Create a copy of the maze to modify it without changing the original

maze\_copy = np.copy(maze)

# Mark the explored nodes

for x, y in explored:

maze\_copy[x][y] = 0.5 # Gray for explored nodes

# Mark the path

for x, y in path:

maze\_copy[x][y] = 0.7 # Yellow for path

# Mark the start and goal

sx, sy = start

gx, gy = goal

maze\_copy[sx][sy] = 0.3 # Green for start

maze\_copy[gx][gy] = 0.9 # Red for goal

# Define a custom colormap

cmap = ListedColormap(['blue', 'gray', 'green', 'yellow', 'red'])

# Plot the maze

plt.imshow(maze\_copy, cmap=cmap)

plt.title("Maze Solution Using Greedy Best-First Search")

plt.colorbar(ticks=[0, 0.5, 0.3, 0.7, 0.9], label='Cell Types')

plt.show()

# Define maze, start, and goal

maze = [

[0, 1, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

start = (0, 0)

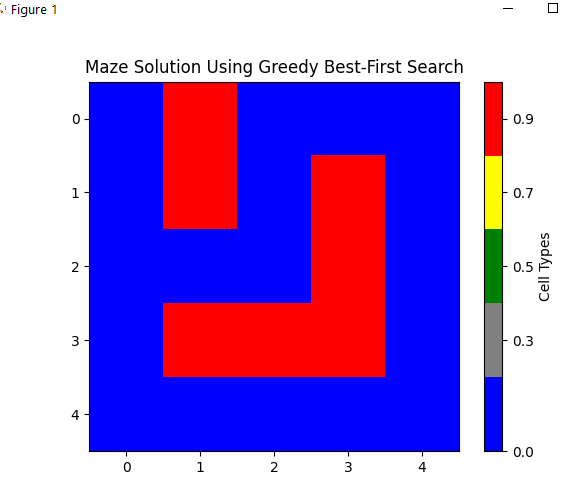
goal = (4, 4)

# Run GBFS and visualize

path, explored = greedy\_best\_first\_search(maze, start, goal)

visualize\_maze(maze, path, explored, start, goal)

The output



 **ListedColormap**: Used to define specific colors for each type of cell:

* **Blue** for walls
* **Gray** for explored nodes
* **Green** for start
* **Yellow** for the path
* **Red** for the goal

1. **City Navigation Using Greedy Best-First Search with Visualization**
   * **Question**: Implement GBFS to navigate through a city map represented by a graph. Nodes represent cities and edges represent roads. Use straight-line distance as the heuristic. Visualize the cities explored and the final route.
   * **Input**: A graph of cities with coordinates, start city, and destination city.
   * **Output**: Visualize the cities explored during the search and highlight the path found on the graph.

The code

import matplotlib.pyplot as plt

import networkx as nx

import numpy as np

from queue import PriorityQueue

# Heuristic function - Euclidean distance

def euclidean\_distance(coord1, coord2):

return np.sqrt((coord1[0] - coord2[0]) \*\* 2 + (coord1[1] - coord2[1]) \*\* 2)

# Greedy Best-First Search algorithm for city navigation

def greedy\_best\_first\_search(graph, start, goal, positions):

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {start: None}

explored = set()

while not open\_list.empty():

\_, current = open\_list.get()

if current == goal:

break

explored.add(current)

for neighbor in graph.neighbors(current):

if neighbor not in explored:

if neighbor not in came\_from:

priority = euclidean\_distance(positions[neighbor], positions[goal])

open\_list.put((priority, neighbor))

came\_from[neighbor] = current

# Reconstruct path from start to goal

path = []

if goal in came\_from:

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, explored

# Visualization function for the city graph

def visualize\_city\_graph(graph, path, explored, positions, start, goal):

plt.figure(figsize=(10, 8))

# Draw the graph

nx.draw(graph, pos=positions, with\_labels=True, node\_color="lightblue", node\_size=500, font\_size=10, font\_weight="bold")

# Highlight explored nodes

explored\_nodes = list(explored)

nx.draw\_networkx\_nodes(graph, positions, nodelist=explored\_nodes, node\_color="gray")

# Highlight the path found

path\_edges = [(path[i], path[i+1]) for i in range(len(path)-1)]

nx.draw\_networkx\_edges(graph, positions, edgelist=path\_edges, edge\_color="yellow", width=2)

# Mark the start and goal nodes

nx.draw\_networkx\_nodes(graph, positions, nodelist=[start], node\_color="green", node\_size=700, label="Start")

nx.draw\_networkx\_nodes(graph, positions, nodelist=[goal], node\_color="red", node\_size=700, label="Goal")

plt.title("City Navigation Using Greedy Best-First Search")

plt.legend(["Explored Cities", "Path", "Start", "Goal"])

plt.show()

# Define city graph with positions

city\_graph = nx.Graph()

# Add cities as nodes

cities = {

'A': (1, 1), 'B': (2, 3), 'C': (3, 1), 'D': (4, 4), 'E': (5, 1),

'F': (6, 3), 'G': (7, 1), 'H': (8, 4)

}

for city, position in cities.items():

city\_graph.add\_node(city, pos=position)

# Define edges (roads between cities)

edges = [

('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('C', 'E'),

('E', 'F'), ('D', 'F'), ('E', 'G'), ('F', 'H'), ('G', 'H')

]

city\_graph.add\_edges\_from(edges)

# Starting and goal cities

start\_city = 'A'

goal\_city = 'H'

# Extract positions for visualization

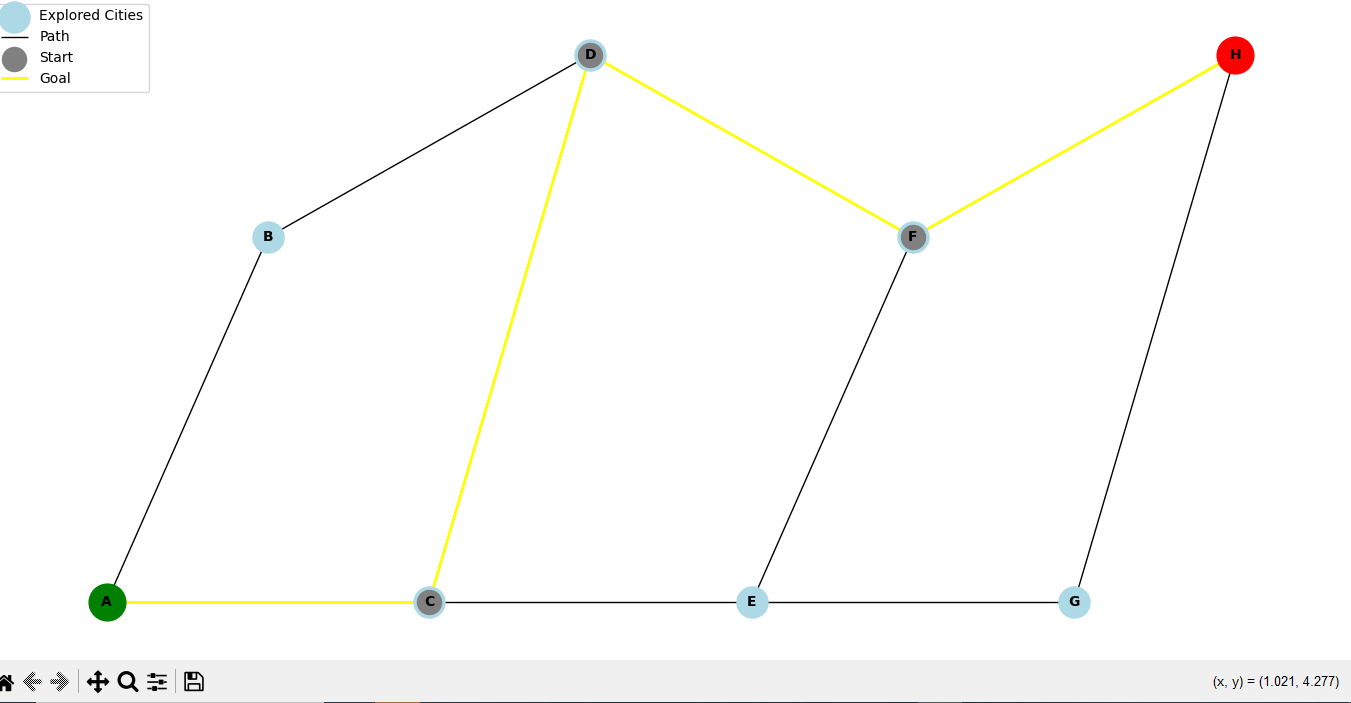
positions = nx.get\_node\_attributes(city\_graph, 'pos')

# Run GBFS and visualize

path, explored = greedy\_best\_first\_search(city\_graph, start\_city, goal\_city, positions)

visualize\_city\_graph(city\_graph, path, explored, positions, start\_city, goal\_city)

The Output



1. **Grid Pathfinding with Diagonal Movement Using GBFS and Visualization**
   * **Question**: Implement GBFS to find a path in a grid where diagonal movements are allowed. Use Euclidean distance as the heuristic. Visualize the search process by displaying the explored nodes, the path taken, and obstacles.
   * **Input**: A 2D grid, start and goal coordinates.
   * **Output**: Visualize the search progress, highlighting explored areas and the final path.

The Code

import matplotlib.pyplot as plt

import numpy as np

from queue import PriorityQueue

# Heuristic function - Euclidean distance

def euclidean\_distance(x1, y1, x2, y2):

return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5

# Greedy Best-First Search function with diagonal movement

def greedy\_best\_first\_search(maze, start, goal):

rows, cols = len(maze), len(maze[0])

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {start: None}

explored = set()

# Define movement directions (up, down, left, right, and diagonals)

directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]

while not open\_list.empty():

\_, current = open\_list.get()

if current == goal:

break

explored.add(current)

x, y = current

for dx, dy in directions:

nx, ny = x + dx, y + dy

neighbor = (nx, ny)

# Check if the neighbor is within bounds, is not an obstacle, and hasn't been explored

if 0 <= nx < rows and 0 <= ny < cols and maze[nx][ny] == 0 and neighbor not in explored:

if neighbor not in came\_from:

priority = euclidean\_distance(nx, ny, goal[0], goal[1])

open\_list.put((priority, neighbor))

came\_from[neighbor] = current

# Reconstruct path from start to goal

path = []

if goal in came\_from:

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, explored

# Visualization function for grid

def visualize\_grid(maze, path, explored, start, goal):

maze\_copy = np.copy(maze)

# Mark the explored nodes

for x, y in explored:

maze\_copy[x][y] = 0.5 # Gray for explored nodes

# Mark the path

for x, y in path:

maze\_copy[x][y] = 0.7 # Yellow for path

# Mark the start and goal

sx, sy = start

gx, gy = goal

maze\_copy[sx][sy] = 0.3 # Green for start

maze\_copy[gx][gy] = 0.9 # Red for goal

plt.imshow(maze\_copy, cmap="coolwarm")

plt.title("Grid Pathfinding with Diagonal Movement Using GBFS")

plt.show()

# Define grid, start, and goal

maze = [

[0, 1, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

start = (0, 0)

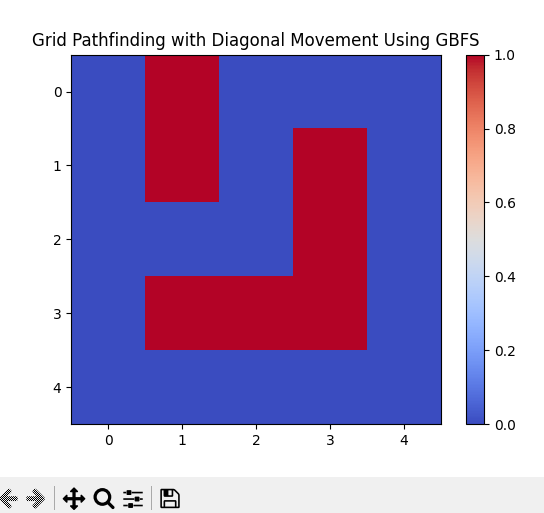
goal = (4, 4)

# Run GBFS and visualize

path, explored = greedy\_best\_first\_search(maze, start, goal)

visualize\_grid(maze, path, explored, start, goal)

The Ouput



1. **Robot Pathfinding in a 3D Environment Using GBFS**
   * **Question**: Implement GBFS to control a robot navigating through a 3D environment filled with obstacles. Use the Euclidean distance between the robot and the target as the heuristic. Visualize the robot's exploration of the 3D space and the final path taken.
   * **Input**: A 3D grid, start, and goal points.
   * **Output**: Display a 3D visualization of the robot’s pathfinding, showing explored nodes and the final path.

The Code

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

from queue import PriorityQueue

# Heuristic function - Euclidean distance in 3D

def euclidean\_distance\_3d(x1, y1, z1, x2, y2, z2):

return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2 + (z1 - z2) \*\* 2) \*\* 0.5

# Greedy Best-First Search function for 3D pathfinding

def greedy\_best\_first\_search\_3d(grid, start, goal):

depth, rows, cols = len(grid), len(grid[0]), len(grid[0][0])

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {start: None}

explored = set()

# Define movement directions (6 possible moves in 3D: front, back, left, right, up, down)

directions = [

(-1, 0, 0), (1, 0, 0), # move along the x-axis

(0, -1, 0), (0, 1, 0), # move along the y-axis

(0, 0, -1), (0, 0, 1) # move along the z-axis

]

while not open\_list.empty():

\_, current = open\_list.get()

if current == goal:

break

explored.add(current)

x, y, z = current

for dx, dy, dz in directions:

nx, ny, nz = x + dx, y + dy, z + dz

neighbor = (nx, ny, nz)

# Check if the neighbor is within bounds, is not an obstacle, and hasn't been explored

if (0 <= nx < depth and 0 <= ny < rows and 0 <= nz < cols and

grid[nx][ny][nz] == 0 and neighbor not in explored):

if neighbor not in came\_from:

priority = euclidean\_distance\_3d(nx, ny, nz, goal[0], goal[1], goal[2])

open\_list.put((priority, neighbor))

came\_from[neighbor] = current

# Reconstruct path from start to goal

path = []

if goal in came\_from:

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, explored

# Visualization function for 3D grid

def visualize\_3d\_path(grid, path, explored, start, goal):

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

# Plot the obstacles

depth, rows, cols = len(grid), len(grid[0]), len(grid[0][0])

for x in range(depth):

for y in range(rows):

for z in range(cols):

if grid[x][y][z] == 1: # obstacle

ax.scatter(x, y, z, color='black', marker='s', s=20)

# Plot the explored nodes

for (x, y, z) in explored:

ax.scatter(x, y, z, color='gray', marker='o', s=10)

# Plot the path

for (x, y, z) in path:

ax.scatter(x, y, z, color='yellow', marker='o', s=20)

# Plot the start and goal

ax.scatter(\*start, color='green', marker='o', s=100, label="Start")

ax.scatter(\*goal, color='red', marker='o', s=100, label="Goal")

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

plt.title("3D Pathfinding with Greedy Best-First Search")

plt.legend()

plt.show()

# Define 3D grid (0 = open, 1 = obstacle), start, and goal points

grid = np.zeros((5, 5, 5))

grid[1][2][1] = grid[1][2][2] = grid[2][1][3] = grid[3][3][3] = 1 # Obstacles

start = (0, 0, 0)

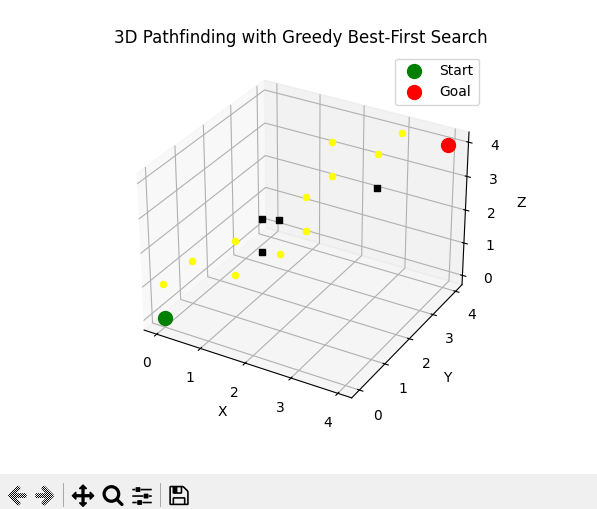
goal = (4, 4, 4)

# Run GBFS and visualize

path, explored = greedy\_best\_first\_search\_3d(grid, start, goal)

visualize\_3d\_path(grid, path, explored, start, goal)

The Output



1. **Treasure Hunt with GBFS in a Dynamic Grid**
   * **Question**: Implement GBFS to find a treasure in a dynamic grid where obstacles may change positions during the search. Use Manhattan distance as the heuristic. Visualize the search, showing the treasure’s location, path, and obstacle changes.
   * **Input**: A 2D grid, start and goal coordinates, with dynamic obstacle placement.
   * **Output**: Display the grid in real-time, showing the robot's search progress and how it reacts to moving obstacles.

The Code

import numpy as np

import matplotlib.pyplot as plt

from queue import PriorityQueue

import random

import time

# Heuristic function - Manhattan distance

def manhattan\_distance(x1, y1, x2, y2):

return abs(x1 - x2) + abs(y1 - y2)

# Greedy Best-First Search function for dynamic grid

def greedy\_best\_first\_search\_dynamic(grid, start, goal, obstacle\_count):

rows, cols = len(grid), len(grid[0])

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {start: None}

explored = set()

while not open\_list.empty():

\_, current = open\_list.get()

explored.add(current)

if current == goal:

break

x, y = current

for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]: # Directions (up, down, left, right)

nx, ny = x + dx, y + dy

neighbor = (nx, ny)

if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0 and neighbor not in explored:

if neighbor not in came\_from:

priority = manhattan\_distance(nx, ny, goal[0], goal[1])

open\_list.put((priority, neighbor))

came\_from[neighbor] = current

# Dynamically change obstacles

if len(explored) % obstacle\_count == 0:

update\_obstacles(grid)

# Reconstruct path from start to goal

path = []

if goal in came\_from:

node = goal

while node:

path.append(node)

node = came\_from[node]

path.reverse()

return path, explored

# Function to update obstacles in the grid randomly

def update\_obstacles(grid):

rows, cols = len(grid), len(grid[0])

for \_ in range(2): # Move two obstacles

x, y = random.randint(0, rows - 1), random.randint(0, cols - 1)

if grid[x][y] == 0: # Only move if it's open space

grid[x][y] = 1 # Place an obstacle

else:

# If an obstacle is already there, remove it

grid[x][y] = 0

# Visualization function

def visualize\_dynamic\_search(grid, path, explored, start, goal):

plt.ion() # Turn on interactive mode

fig, ax = plt.subplots()

while True:

ax.clear()

ax.imshow(grid, cmap="coolwarm", interpolation='nearest')

# Mark the path

for (x, y) in path:

grid[x][y] = 0.5 # Gray for path

# Mark the explored nodes

for (x, y) in explored:

if grid[x][y] != 0.5: # Avoid overwriting the path

grid[x][y] = 0.3 # Blue for explored nodes

# Mark start and goal

sx, sy = start

gx, gy = goal

grid[sx][sy] = 0.2 # Green for start

grid[gx][gy] = 0.9 # Red for goal

ax.imshow(grid, cmap="coolwarm", interpolation='nearest')

plt.title("Treasure Hunt with GBFS - Dynamic Grid")

plt.pause(0.5) # Pause for visualization

if path:

# Show the last part of the path being taken

last\_step = path.pop(0) # Remove the first step

grid[last\_step] = 0.5 # Update the grid

if not path: # Exit if the path is complete

break

plt.ioff() # Turn off interactive mode

plt.show()

# Define 2D grid, start and goal coordinates

grid\_size = 10

grid = np.zeros((grid\_size, grid\_size))

grid[1][1] = grid[1][2] = grid[2][1] = 1 # Initial obstacles

start = (0, 0)

goal = (9, 9)

# Run GBFS with dynamic obstacles and visualize

path, explored = greedy\_best\_first\_search\_dynamic(grid, start, goal, obstacle\_count=5)

visualize\_dynamic\_search(grid, path, explored, start, goal)

The Output

