**LAB 2**

PATHFINDING IN A HAUNTED HOUSE

* Objective: Implement Greedy Best-First Search & A\* algorithms in a 2D haunted house grid.

Explore different heuristics for efficient pathfinding

* Problem Setup:

Grid map S: Start, G: Goal, 0: Open, 1: Wall

Example:

S0010/11016/00010/11011/00000

* Tasks:
* Part A: Greedy Best-First Search (h(n))
* Part B: A\* Search (f(n) = g(n) + h(n))
* Part C: Try heuristics: Manhattan, Euclidean ,diagnol
* Compare :

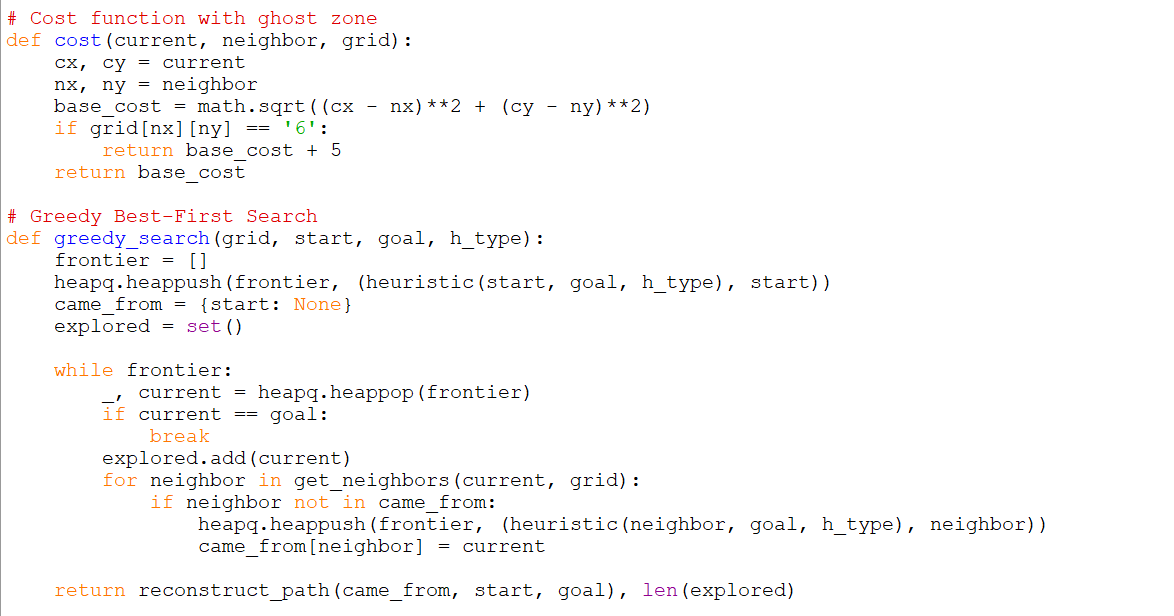
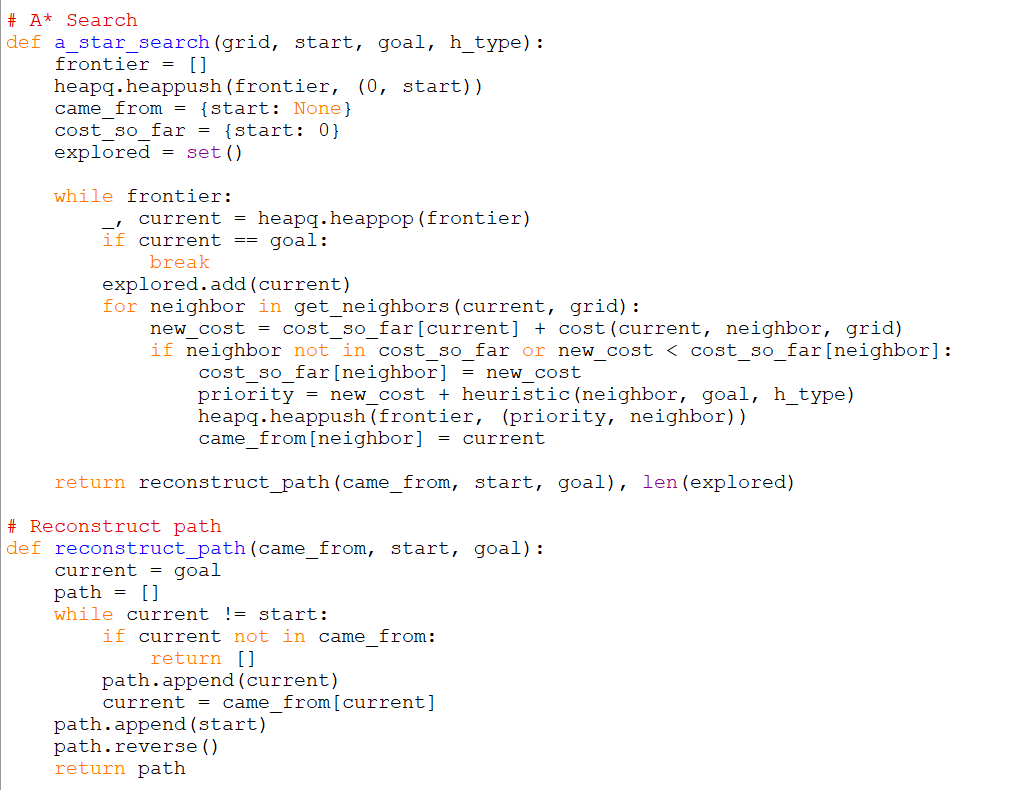
Path length,nodes explored ,(execution time optional)

* Bonus challenge:

Ghost zones(high cost), diagnol moves , visualize path

A screen shot of a computer code

AI-generated content may be incorrect.

  A screenshot of a computer code

AI-generated content may be incorrect.

OUTPUT

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a computer code

AI-generated content may be incorrect.

**CODE**

import heapq

import math

import time

# Heuristic function

def heuristic(a, b, h\_type):

dx, dy = abs(a[0] - b[0]), abs(a[1] - b[1])

if h\_type == "manhattan":

return dx + dy

elif h\_type == "euclidean":

return math.sqrt(dx\*\*2 + dy\*\*2)

elif h\_type == "diagonal":

return max(dx, dy)

return 0

# Parse grid string

def parse\_grid(grid\_str):

grid = [list(row) for row in grid\_str.split("/")]

start = goal = None

for i in range(len(grid)):

for j in range(len(grid[0])):

if grid[i][j] == 'S':

start = (i, j)

elif grid[i][j] == 'G':

goal = (i, j)

return grid, start, goal

# Get valid neighbors (8 directions)

def get\_neighbors(pos, grid):

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

neighbors = []

for dx, dy in directions:

x, y = pos[0] + dx, pos[1] + dy

if 0 <= x < len(grid) and 0 <= y < len(grid[0]):

if grid[x][y] != '1':

neighbors.append((x, y))

return neighbors

# Cost function with ghost zone

def cost(current, neighbor, grid):

cx, cy = current

nx, ny = neighbor

base\_cost = math.sqrt((cx - nx)\*\*2 + (cy - ny)\*\*2)

if grid[nx][ny] == '6':

return base\_cost + 5

return base\_cost

# Greedy Best-First Search

def greedy\_search(grid, start, goal, h\_type):

frontier = []

heapq.heappush(frontier, (heuristic(start, goal, h\_type), start))

came\_from = {start: None}

explored = set()

while frontier:

\_, current = heapq.heappop(frontier)

if current == goal:

break

explored.add(current)

for neighbor in get\_neighbors(current, grid):

if neighbor not in came\_from:

heapq.heappush(frontier, (heuristic(neighbor, goal, h\_type), neighbor))

came\_from[neighbor] = current

return reconstruct\_path(came\_from, start, goal), len(explored)

# A\* Search

def a\_star\_search(grid, start, goal, h\_type):

frontier = []

heapq.heappush(frontier, (0, start))

came\_from = {start: None}

cost\_so\_far = {start: 0}

explored = set()

while frontier:

\_, current = heapq.heappop(frontier)

if current == goal:

break

explored.add(current)

for neighbor in get\_neighbors(current, grid):

new\_cost = cost\_so\_far[current] + cost(current, neighbor, grid)

if neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]:

cost\_so\_far[neighbor] = new\_cost

priority = new\_cost + heuristic(neighbor, goal, h\_type)

heapq.heappush(frontier, (priority, neighbor))

came\_from[neighbor] = current

return reconstruct\_path(came\_from, start, goal), len(explored)

# Reconstruct path

def reconstruct\_path(came\_from, start, goal):

current = goal

path = []

while current != start:

if current not in came\_from:

return []

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path

# Visualize path

def visualize\_path(grid, path):

visual = [row[:] for row in grid]

for x, y in path:

if visual[x][y] not in ('S', 'G'):

visual[x][y] = '\*'

for row in visual:

print("".join(row))

# Run all heuristics

def run\_all(grid\_str):

grid, start, goal = parse\_grid(grid\_str)

if start is None or goal is None:

print("Error: Grid must contain both 'S' (start) and 'G' (goal).")

return

heuristics = ["manhattan", "euclidean", "diagonal"]

for h in heuristics:

print(f"\nGreedy Best-First Search ({h})")

t1 = time.time()

path, explored = greedy\_search(grid, start, goal, h)

t2 = time.time()

print(f"Path length: {len(path)} | Nodes explored: {explored} | Time: {t2 - t1:.4f}s")

visualize\_path(grid, path)

print(f"\nA\* Search ({h})")

t1 = time.time()

path, explored = a\_star\_search(grid, start, goal, h)

t2 = time.time()

print(f"Path length: {len(path)} | Nodes explored: {explored} | Time: {t2 - t1:.4f}s")

visualize\_path(grid, path)

# Fixed grid input (includes both 'S' and 'G')

grid\_input = "S0010/11016/00010/11011/0000G"

run\_all(grid\_input)