import math

import time

import random

import heapq

import matplotlib.pyplot as plt

# Define the initial and goal states

start\_state = (0, 0) # Initial state represented as (x, y)

goal\_state = (9, 9) # Goal state represented as (x, y)

# Define the successor function

def get\_successors(state, grid):

x, y = state

successors = []

# Generate all possible successors

for dx in [-1, 0, 1]:

for dy in [-1, 0, 1]:

if dx == 0 and dy == 0:

continue # Skip current state

new\_x = x + dx

new\_y = y + dy

# Check if the successor is within the grid boundaries

if 0 <= new\_x < 10 and 0 <= new\_y < 10 and grid[new\_x][new\_y] != 1:

if dx != 0 or dy != 0:

# If moving diagonally, check if the adjacent cells are obstacles

if grid[x + dx][y] == 1 or grid[x][y + dy] == 1:

continue # Skip this successor

successors.append((new\_x, new\_y))

return successors

class Node:

def \_\_init\_\_(self, state, cost, heuristic):

self.state = state

self.cost = cost

self.heuristic = heuristic

self.total\_cost = cost + heuristic

self.parent = None

def \_\_lt\_\_(self, other):

return self.total\_cost < other.total\_cost

def \_\_eq\_\_(self, other):

return self.total\_cost == other.total\_cost

def \_\_gt\_\_(self, other):

return self.total\_cost > other.total\_cost

fringe = [] # Priority queue for the fringe

def add\_to\_fringe(node):

heapq.heappush(fringe, (node.total\_cost, node))

def get\_from\_fringe():

\_, node = heapq.heappop(fringe)

return node

def is\_fringe\_empty():

return len(fringe) == 0

def heuristic(state, goal\_state):

x1, y1 = state

x2, y2 = goal\_state

return math.sqrt((x2 - x1) \* 2 + (y2 - y1) \* 2)

def get\_cost(state, successor\_state):

x1, y1 = state

x2, y2 = successor\_state

dx = abs(x2 - x1)

dy = abs(y2 - y1)

if dx == 1 and dy == 1 or dx == 1 or dy == 1:

return math.sqrt(2)

else:

return 1

def A\_star\_search(start\_state, goal\_state, grid):

start\_node = Node(start\_state, 0, heuristic(start\_state, goal\_state))

add\_to\_fringe(start\_node)

visited = set() # Keep track of visited states

while not is\_fringe\_empty():

current\_node = get\_from\_fringe()

if current\_node.state == goal\_state:

path = []

while current\_node:

path.append(current\_node.state)

current\_node = current\_node.parent

path.reverse()

return path

visited.add(current\_node.state)

successors = get\_successors(current\_node.state, grid)

for successor\_state in successors:

if successor\_state in visited:

continue # Skip already visited states

successor\_cost = current\_node.cost + get\_cost(current\_node.state, successor\_state)

successor\_heuristic = heuristic(successor\_state, goal\_state)

# Check if the successor is already in the fringe with a lower total cost

existing\_successor = next((node for \_, node in fringe if node.state == successor\_state), None)

if existing\_successor and existing\_successor.total\_cost <= successor\_cost + successor\_heuristic:

continue # Skip this successor

successor\_node = Node(successor\_state, successor\_cost, successor\_heuristic)

successor\_node.parent = current\_node

add\_to\_fringe(successor\_node)

visited.add(successor\_state)

return None

# Required Code Final output

grid\_size = 10

problem\_tests = 20

obstacle\_percentages = range(10, 100, 10)

results = []

for obstacle\_percentage in obstacle\_percentages:

total\_time = 0

worst\_case = float(0)

best\_case = float(0)

obstacle\_count = int(grid\_size \* grid\_size \* obstacle\_percentage / 100)

for \_ in range(problem\_tests):

# Generate a random grid with obstacles

flag = True

while flag:

obstacles = random.sample(range(grid\_size \* grid\_size), obstacle\_count)

start\_value = int(''.join(str(value) for value in start\_state))

goal\_value = int(''.join(str(value) for value in goal\_state))

if start\_value and goal\_value in obstacle\_count:

flag = True

else:

flag = False

grid = [[1 if i \* grid\_size + j in obstacles else 0 for j in range(grid\_size)] for i in range(grid\_size)]

# Measure the execution time

start\_time = time.time()

path = A\_star\_search(start\_state, goal\_state, grid)

end\_time = time.time()

if path:

execution\_time = end\_time - start\_time

total\_time += execution\_time

worst\_case = max(worst\_case, execution\_time)

best\_case = min(best\_case, execution\_time)

average\_time = total\_time / problem\_tests

results.append((obstacle\_percentage, average\_time, worst\_case, best\_case))

# Print the results as a table

print("Obstacle % | Average Time | Worst Case | Best Case")

for result in results:

print("{:<12} | {:<12} | {:<10} | {:<9}".format(\*result))