**"""2nd asgn\_2203A51240-K.Bhavana**

**\*\*Implementing Breadth First Search Algorithm using a QUEUE\*\***

from collections import defaultdict, deque

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

def bfs(self, start):

visited = set()

queue = deque([start])

while queue:

vertex = queue.popleft()

if vertex not in visited:

print(vertex, end=" ")

visited.add(vertex)

for neighbor in self.graph[vertex]:

if neighbor not in visited:

queue.append(neighbor)

# Exam

graph = Graph()

graph.add\_edge(0, 1)

graph.add\_edge(0, 2)

graph.add\_edge(1, 2)

graph.add\_edge(2, 0)

graph.add\_edge(2, 3)

graph.add\_edge(3, 3)

graph.add\_edge(3, 4)

graph.add\_edge(4, 5)

graph.add\_edge(5, 2)

print("Breadth First Traversal (starting from vertex 2):")

graph.bfs(2)

**OUTPUT:**

Breadth First Traversal (starting from vertex 2):

2 0 3 1 4 5

**"""\*\*Implement Depth First Search Algorithm using a Stack\*\*"""**

class Graph:

def \_\_init\_\_(self):

self.adj\_list = {}

def add\_edge(self, u, v):

if u not in self.adj\_list:

self.adj\_list[u] = []

self.adj\_list[u].append(v)

def dfs(self, start):

visited = set()

stack = [start]

while stack:

node = stack.pop()

if node not in visited:

print(node, end=" ")

visited.add(node)

if node in self.adj\_list:

neighbors = reversed(self.adj\_list[node]) # To maintain the order specified in the problem statement

for neighbor in neighbors:

if neighbor not in visited:

stack.append(neighbor)

# Example usage:

g = Graph()

g.add\_edge('S', 'A')

g.add\_edge('S', 'B')

g.add\_edge('S', 'C')

g.add\_edge('A', 'D')

g.add\_edge('B', 'E')

g.add\_edge('B', 'F')

g.add\_edge('C', 'G')

g.add\_edge('D', 'H')

print("Depth First Search starting from node 'S':")

g.dfs('S')

**OUTPUT**:

Depth First Search starting from node 'S':

S A D H B E F C G

**"""# Implement A\* Algorithm using Numpy"""**

import numpy as np

from heapq import heappush, heappop

class AStar:

def \_\_init\_\_(self, grid):

self.grid = grid

self.num\_rows, self.num\_cols = grid.shape

self.start = None

self.goal = None

def find\_path(self, start, goal):

self.start = start

self.goal = goal

open\_list = []

closed\_list = set()

heappush(open\_list, (0, start))

parent = {}

g\_score = {start: 0}

while open\_list:

\_, current = heappop(open\_list)

if current == goal:

return self.reconstruct\_path(parent, current)

closed\_list.add(current)

for neighbor in self.get\_neighbors(current):

if neighbor in closed\_list:

continue

tentative\_g\_score = g\_score[current] + 1

if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:

parent[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score = tentative\_g\_score + self.heuristic(neighbor, goal)

heappush(open\_list, (f\_score, neighbor))

return None

def get\_neighbors(self, node):

row, col = node

neighbors = []

for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

new\_row, new\_col = row + dr, col + dc

if 0 <= new\_row < self.num\_rows and 0 <= new\_col < self.num\_cols and self.grid[new\_row][new\_col] != 1:

neighbors.append((new\_row, new\_col))

return neighbors

def heuristic(self, a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def reconstruct\_path(self, parent, current):

path = []

while current in parent:

path.append(current)

current = parent[current]

path.append(self.start)

path.reverse()

return path

# Example usage:

grid = np.array([

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

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

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

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

[0, 0, 0, 0, 0]

])

astar = AStar(grid)

start = (0, 0)

goal = (4, 4)

path = astar.find\_path(start, goal)

if path:

print("Path found:", path)

else:

print("No path found")

**OUTPUT:**

Path found: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]