10 Write the python program to implement A\* algorithm python code

import heapq

class Node:

def \_\_init\_\_(self, position, parent=None):

self.position = position

self.parent = parent

self.g = 0 # Cost from start to current node

self.h = 0 # Heuristic (Manhattan distance)

self.f = 0 # Total cost

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

return self.f < other.f

def heuristic(a, b):

"""Manhattan distance"""

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

def astar(grid, start, end):

open\_list = []

closed\_list = set()

start\_node = Node(start)

end\_node = Node(end)

heapq.heappush(open\_list, start\_node)

while open\_list:

current\_node = heapq.heappop(open\_list)

closed\_list.add(current\_node.position)

# Goal reached

if current\_node.position == end\_node.position:

path = []

while current\_node:

path.append(current\_node.position)

current\_node = current\_node.parent

return path[::-1] # Return reversed path

# Neighbor positions (up, down, left, right)

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

for dx, dy in neighbors:

neighbor\_pos = (current\_node.position[0] + dx, current\_node.position[1] + dy)

# Check boundaries and obstacles

if (0 <= neighbor\_pos[0] < len(grid) and

0 <= neighbor\_pos[1] < len(grid[0]) and

grid[neighbor\_pos[0]][neighbor\_pos[1]] == 0 and

neighbor\_pos not in closed\_list):

neighbor\_node = Node(neighbor\_pos, current\_node)

neighbor\_node.g = current\_node.g + 1

neighbor\_node.h = heuristic(neighbor\_pos, end\_node.position)

neighbor\_node.f = neighbor\_node.g + neighbor\_node.h

# If not already in open list or better path found

if all(neighbor\_pos != n.position or neighbor\_node.g < n.g for n in open\_list):

heapq.heappush(open\_list, neighbor\_node)

return None # No path found

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

grid = [

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

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

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

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

[0, 0, 0, 0, 0]

]

start = (0, 0)

end = (4, 4)

path = astar(grid, start, end)

print("Path found:", path) import heapq

class Node:

def \_\_init\_\_(self, position, parent=None):

self.position = position

self.parent = parent

self.g = 0 # Cost from start to current node

self.h = 0 # Heuristic (Manhattan distance)

self.f = 0 # Total cost

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

return self.f < other.f

def heuristic(a, b):

"""Manhattan distance"""

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

def astar(grid, start, end):

open\_list = []

closed\_list = set()

start\_node = Node(start)

end\_node = Node(end)

heapq.heappush(open\_list, start\_node)

while open\_list:

current\_node = heapq.heappop(open\_list)

closed\_list.add(current\_node.position)

# Goal reached

if current\_node.position == end\_node.position:

path = []

while current\_node:

path.append(current\_node.position)

current\_node = current\_node.parent

return path[::-1] # Return reversed path

# Neighbor positions (up, down, left, right)

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

for dx, dy in neighbors:

neighbor\_pos = (current\_node.position[0] + dx, current\_node.position[1] + dy)

# Check boundaries and obstacles

if (0 <= neighbor\_pos[0] < len(grid) and

0 <= neighbor\_pos[1] < len(grid[0]) and

grid[neighbor\_pos[0]][neighbor\_pos[1]] == 0 and

neighbor\_pos not in closed\_list):

neighbor\_node = Node(neighbor\_pos, current\_node)

neighbor\_node.g = current\_node.g + 1

neighbor\_node.h = heuristic(neighbor\_pos, end\_node.position)

neighbor\_node.f = neighbor\_node.g + neighbor\_node.h

# If not already in open list or better path found

if all(neighbor\_pos != n.position or neighbor\_node.g < n.g for n in open\_list):

heapq.heappush(open\_list, neighbor\_node)

return None # No path found

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

grid = [

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

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

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

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

[0, 0, 0, 0, 0]

]

start = (0, 0)

end = (4, 4)

path = astar(grid, start, end)

print("Path found:", path)

**OUTPUT:**

