# ASSIGNMENT 4

**4) a. Create a Python program to find and print one valid solution for the 8- queens problem on an 8x8 chessboard..**

**Code:**

```
N = 8
def solveNQueens(board, col): if col == N: print(board)
		return True
	for i in range(N):
		if isSafe(board, i, col): board[i][col] = 1
			if solveNQueens(board, col + 1): return True
			board[i][col] = 0
	return False
def isSafe(board, row, col):
	for x in range(col):
		if board[row][x] == 1: return False
	for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
		if board[x][y] == 1:
			return False
	for x, y in zip(range(row, N, 1), range(col, -1, -1)):
		if board[x][y] == 1:
			return False
	return True
board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0): print("No solution found")
```

**Output:**

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

=== Code Execution Successful ===
```

**b. Extend the previous program to find and print all valid solutions for the 8- queens problem on an 8x8 chessboard.**

**Code:**

```
result = []
def isSafe(board, row, col):
   for i in range(col):
      if (board[row][i]):  return False
   i = row
   j = col
   while i >= 0 and j >= 0:
      if(board[i][j]):  return False
```

```
        i -= 1
        j -= 1
    i = row
    j = col
    while j >= 0 and i < 8:
        if(board[i][j]):  return False
        i = i + 1
        j = j - 1
    return True
def solveNQUtil(board,  col):  if (col == 8):
        v = []
        for i in board:
          for j in range(len(i)):
            if i[j] == 1: v.append(j+1)
        result.append(v)
        return True
    res = False
    for i in range(8):
        if (isSafe(board, i, col)):
            board[i][col] = 1
            res = solveNQUtil(board,  col + 1) or res
            board[i][col] = 0
    return res
def solveNQ(n):  result.clear();
    board = [[0 for j in range(n)]
             for i in range(n)]
    solveNQUtil(board,  0)
    result.sort()
    return result
n = 8
res = solveNQ(n)
print(res)
```

## Output:

[[1, 5, 8, 6, 3, 7, 2, 4], [1, 6, 8, 3, 7, 4, 2, 5], [1, 7, 4, 6, 8, 2, 5, 3], [1, 7, 5, 8, 2, 4, 6, 3], [2, 4, 6, 8, 3, 1, 7 , 5], [2, 5, 7, 1, 3, 8, 6, 4], [2, 5, 7, 4, 1, 8, 6, 3], [2, 6, 1, 7, 4, 8, 3, 5], [2, 6, 8, 3, 1, 4, 7, 5], [2, 7, 3, 6, 8, 5, 1, 4], [2, 7, 5, 8, 1, 4, 6, 3], [2, 8, 6, 1, 3, 5, 7, 4], [3, 1, 7, 5, 8, 2, 4, 6], [3, 5, 2, 8, 1, 7, 4, 6], [3, 5, 2, 8, 6, 4, 7, 1], [3, 5, 7, 1, 4, 2, 8, 6], [3, 5, 8, 4, 1, 7, 2, 6], [3, 6, 2, 5, 8, 1, 7, 4], [3, 6, 2, 7, 1, 4, 8, 5], [3, 6, 2, 7, 5, 1, 8, 4], [3, 6, 4, 1, 8, 5, 7, 2], [3, 6, 4, 2, 8, 5, 7, 1], [3, 6, 8, 1, 4, 7, 5, 2], [3, 6, 8, 1, 5, 7, 2, 4], [3, 6, 8, 2, 4, 1, 7, 5], [3, 7, 2, 8, 5, 1, 4, 6], [3, 7, 2, 8, 6, 4, 1, 5], [3, 8, 4, 7, 1, 6, 2, 5], [4, 1, 5, 8, 2, 7, 3, 6], [4, 1, 5, 8, 6, 3, 7, 2], [4, 2, 5, 8, 6, 1, 3, 7], [4, 2, 7, 3, 6, 8, 1, 5], [4, 2, 7, 3, 6, 8, 5, 1], [4, 2, 7, 5, 1, 8, 6, 3], [4, 2, 8, 5, 7, 1, 3, 6], [4, 2, 8, 6, 1, 3, 5, 7], [4, 6, 1, 5, 2, 8, 3, 7], [4, 6, 8, 2, 7, 1, 3, 5], [4, 6, 8, 3, 1, 7, 5, 2], [4, 7, 1, 8, 5, 2, 6, 3], [4, 7, 3, 8, 2, 5, 1, 6], [4, 7, 5, 2, 6, 1, 3, 8], [5, 2, 4, 6, 8, 3, 1, 7], [6, 3, 1, 7, 5, 8, 2, 4], [6, 3, 1, 8, 4, 2, 7, 5], [6, 3, 1, 8, 5, 2, 4, 7], [6, 3, 5, 7, 1, 4, 2, 8], [6, 3, 5, 8, 1, 4, 2, 7], [6, 3, 7, 2, 4, 8, 1, 5], [6, 3, 7, 2, 8, 5, 1, 4], [6, 3, 7, 4, 1, 8, 2, 5], [6, 4, 1, 5, 8, 2, 7, 3], [6, 4, 2, 8, 5, 7, 1, 3], [6, 4, 7, 1, 3, 5, 2, 8], [6, 4, 7, 1, 8, 2, 5, 3], [6, 8, 2, 4, 1, 7, 5, 3], [7, 1, 3, 8, 6, 4, 2, 5], [7, 2, 4, 1, 8, 5, 3, 6], [7, 2, 6, 3, 1, 4, 8, 5], [7, 3, 1, 6, 8, 5, 2, 4], [7, 3, 8, 2, 5, 1, 6, 4], [7, 4, 2, 5, 8, 1, 3, 6], [7, 4, 2, 8, 6, 1, 3, 5], [7, 5, 3, 1, 6, 8, 2, 4], [8, 2, 4, 1, 7, 5, 3, 6], [8, 2, 5, 3, 1, 7, 4, 6], [8, 3, 1, 6, 2, 5, 7, 4], [8, 4, 1, 3, 6, 2, 7, 5]]

=== Code Execution Successful ===

## ASSIGNMENT 5

**5) a. Write a Python program to find the shortest path between two nodes in an unweighted, undirected graph using BFS.**

**Code:**

```
from collections import deque
def bfs(graph, S, par, dist):
    q = deque()
    dist[S] = 0
    q.append(S)
    while q: node = q.popleft()
        for neighbor in graph[node]:
            if dist[neighbor] == float('inf'):
                par[neighbor] = node
                dist[neighbor] = dist[node] + 1
                q.append(neighbor)
def print_shortest_distance(graph, S, D, V):
    par = [-1] * V
    dist = [float('inf')] * V
    bfs(graph, S, par, dist)
    if dist[D] == float('inf'): print("Source and Destination are not connected")
        return
    path = []
    current_node = D
    path.append(D)
    while par[current_node] != -1:
        path.append(par[current_node])
        current_node = par[current_node]
    for i in range(len(path) - 1, -1, -1):
        print(path[i], end=" ")
if __name__ == "__main__":
    V = 8
    S, D = 2, 6
    edges = [ [0, 1], [1, 2], [0, 3], [3, 4], [4, 7], [3, 7], [6, 7], [4, 5], [4, 6], [5, 6] ]
    graph = [[] for _ in range(V)]
    for edge in edges: graph[edge[0]].append(edge[1])
        graph[edge[1]].append(edge[0])
    print_shortest_distance(graph, S, D, V)
```

**Output:**

2 1 0 3 4 6

=== Code Execution Successful ===

**b. Implement BFS to find the shortest path from a given starting point to all other nodes in an unweighted, directed graph.**

**Code:**

```
from collections import defaultdict
def build_graph():
   edges = [ ["A", "B"], ["A", "E"],
      ["A", "C"], ["B", "D"],
      ["B", "E"], ["C", "F"],
      ["C", "G"], ["D", "E"]
   ]
   graph = defaultdict(list)
   for edge in edges:
      a, b = edge[0], edge[1]
      graph[a].append(b)
      graph[b].append(a)
   return graph
if __name__ == "__main__":
   graph = build_graph()
   print(graph)
```

**Output:**

defaultdict(<class 'list'>, {'A': ['B', 'E', 'C'], 'B': ['A', 'D', 'E'], 'E': ['A', 'B', 'D'], 'C': ['A', 'F', 'G'], 'D': ['B', 'E'], 'F': ['C'], 'G': ['C']})

=== Code Execution Successful ==

**c. Create a Python program to find the connected components in an undirected graph using BFS.**

**Code:**

```
class Graph_structure:
  def __init__(self, V):  self.V = V
    self.adj = [[] for i in range(V)]
  def DFS_Utility(self, temp, v, visited):
    visited[v] = True
    temp.append(v)
    for i in self.adj[v]: if visited[i] == False:
        temp = self.DFS_Utility(temp, i, visited)
    return temp
```

```
  def add_edge(self, v, w):  self.adj[v].append(w)
    self.adj[w].append(v)
  def find_connected_components(self):
    visited = []
    connected_comp = []
    for i in range(self.V):  visited.append(False)
    for v in range(self.V):
      if visited[v] == False: temp = []
        connected_comp.append(self.DFS_Utility(temp, v, visited))
    return connected_comp
my_instance = Graph_structure(6)
my_instance.add_edge(1, 0)
my_instance.add_edge(2, 3)
my_instance.add_edge(3, 4)
my_instance.add_edge(5, 0)
print("There are 6 edges. They are : ")
print("1-->0")
print("2-->3")
print("3-->4")
print("5-->0")
connected_comp = my_instance.find_connected_components()
print("The connected components are...")
print(connected_comp)
```

**Output:**

```
There are 6 edges. They are :
1-->0
2-->3
3-->4
5-->0
The connected components are...
[[0, 1, 5], [2, 3, 4]]

=== Code Execution Successful ===
```

## ASSIGNMENT 6

**6) a. Write a Python program to find all possible paths from a given starting node to a destination node in an unweighted, directed graph using DFS.**

**Code:**

```
from collections import defaultdict
class Graph:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = defaultdict(list)
  def addEdge(self, u, v):
```

```
        self.graph[u].append(v)
    def printAllPathsUtil(self, u, d, visited, path):
        visited[u]= True
        path.append(u)
        if u == d:
            print (path)
        else:
            for i in self.graph[u]:
                if visited[i]== False:
                    self.printAllPathsUtil(i, d, visited, path)
        path.pop()
        visited[u]= False
    def printAllPaths(self, s, d):
        visited =[False]*(self.V)
        path = []
        self.printAllPathsUtil(s, d, visited, path)
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(2, 0)
g.addEdge(2, 1)
g.addEdge(1, 3)
s = 2 ; d = 3
print ("Following are all different paths from % d to % d :" %(s, d))
g.printAllPaths(s, d)
```

## Output:

```
Following are all different paths from  2 to  3 :
[2, 0, 1, 3]
[2, 0, 3]
[2, 1, 3]
=== Code Execution Successful ===
```

**b. Implement DFS to check if a cycle exists in an undirected graph.**

## Code:

```
from collections import defaultdict
class Graph:
    def __init__(self, v):
        self.v = v
        self.adjacency_list = defaultdict(list)
    def add_edge(self, v, u):
        self.adjacency_list[v].append(u)
        self.adjacency_list[u].append(v)
```

```
    def is_cyclic_util(self,  v, visited, parent):
        visited[v] = True
        for neighbor in self.adjacency_list[v]:
            if not visited[neighbor]:
                if self.is_cyclic_util(neighbor,  visited, v):
                    return True
            elif neighbor != parent:
                return True
        return False
    def is_cyclic(self):
        visited = [False] * self.v
        for i in range(self.v):
            if not visited[i] and self.is_cyclic_util(i,  visited, -1):
                return True
        return False
g = Graph(6)
g.add_edge(0, 1)
g.add_edge(1, 5)
g.add_edge(5, 4)
g.add_edge(4, 0)
g.add_edge(4, 3)
g.add_edge(3, 2)
g.add_edge(0, 2)
if g.is_cyclic():
    print("Graph is cyclic")
else:
    print("Graph is acyclic")
```

**Output:**

```
Graph is cyclic


=== Code Execution  Successful ===
```

**c. Create a Python program to find the topological ordering of nodes in a directed acyclic graph (DAG) using DFS.**

**Code:**

```
from collections import defaultdict
class Graph: def __init__(self,n):
        self.graph = defaultdict(list)
        self.N = n
    def addEdge(self,m,n):
        self.graph[m].append(n)
```

```
    def sortUtil(self,n,visited,stack): visited[n] = True
        for element in self.graph[n]:
            if visited[element] == False:
                self.sortUtil(element,visited,stack)
        stack.insert(0,n)
    def topologicalSort(self):
        visited = [False]*self.N
        stack =[]
        for element in range(self.N):
            if visited[element] == False:
                self.sortUtil(element,visited,stack)
        print(stack)
graph = Graph(6)
graph.addEdge(0,1);
graph.addEdge(0,3);
graph.addEdge(1,2);
graph.addEdge(2,3);
graph.addEdge(2,4);
graph.addEdge(3,4);
graph.addEdge(4,5)
print("The Topological Sort of The Graph Is:  ")
graph.topologicalSort()
```

**Output:**

```
The Topological Sort of The Graph Is:
[0, 1, 2, 3, 4, 5]

=== Code Execution Successful ===
```

# ASSIGNMENT 7

**7) a. Write a Python program to solve the 8-Puzzle problem using the A* search algorithm.**

**Code:**

```
import copy
from heapq import heappush, heappop
n = 3
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
```

```
        if not self.heap:
            return True
        else:
class nodes:
    def __init__(self, parent, mats, empty_tile_posi,  costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels
    def __lt__(self, nxt):
        return self.costs < nxt.costs
def calculateCosts(mats,  final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1
    return count
def newNodes(mats, empty_tile_posi,  new_empty_tile_posi,
            levels, parent, final) -> nodes:
    new_mats = copy.deepcopy(mats)
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]
    costs = calculateCosts(new_mats,  final)
    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,  costs, levels)
    return new_nodes
def printMatsrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")
        print()
def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):
    if root == None:    return
    printPath(root.parent)
    printMatsrix(root.mats)
    print()
def solve(initial, empty_tile_posi,  final):
    pq = priorityQueue()
    costs = calculateCosts(initial,  final)
```

```
    root = nodes(None, initial,
            empty_tile_posi, costs, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.costs == 0:  printPath(minimum)
            return
        for i in range(n):
            new_tile_posi = [  minimum.empty_tile_posi[0] + rows[i], minimum.empty_tile_posi[1] +cols[i],]
            if isSafe(new_tile_posi[0], new_tile_posi[1]):
            child = newNodes(minimum.mats,minimum.empty_tile_posi, new_tile_posi, minimum.levels+ 1,
                            minimum, final,)
                pq.push(child)
initial = [ [ 1, 2, 3 ],  [ 5, 6, 0 ],  [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],  [ 5, 8, 6 ],  [ 0, 7, 4 ] ]
empty_tile_posi = [ 1, 2 ]
solve(initial, empty_tile_posi, final)
```

## Output:

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4

=== Code Execution Successful ===
```

**b. Implement a Python program to check if a given 8-Puzzle configuration is solvable or not.**

## Code:

```
def getInvCount(arr):
    inv_count = 0
    empty_value = -1
    for i in range(0, 9):
        for j in range(i + 1, 9):
            if arr[j] != empty_value and arr[i] != empty_value and arr[i] > arr[j]:
                inv_count += 1
    return inv_count
```

```
def isSolvable(puzzle) :
   inv_count = getInvCount([j for sub in puzzle for j in sub])
   return (inv_count % 2 == 0)
puzzle = [[1, 8, 2],[-1, 4, 3],[7, 6, 5]]
if(isSolvable(puzzle)) :
   print("Solvable")
else :
   print("Not Solvable")
```

**Output:**

```
Solvable

=== Code Execution Successful ===
```

# ASSIGNMENT 8

**8) a. Implement a Python program to find the optimal route for a traveling salesman to visit a given set of cities and return to the starting city, minimizing the total distance traveled.**

**Code:**

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
   vertex = []
   for i in range(V):
      if i != s:
         vertex.append(i)
   min_path = maxsize
   next_permutation=permutations(vertex)
   for i in next_permutation:
      current_pathweight = 0
      k = s
      for j in i:
         current_pathweight += graph[k][j]
         k = j
      current_pathweight += graph[k][s]
      min_path = min(min_path, current_pathweight)
   return min_path
if __name__ == "__main__":
   graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
   s = 0
   print(travellingSalesmanProblem(graph, s))
```

**Output:**

80

=== Code Execution Successful ===

**b. Design a heuristic algorithm, such as the Nearest Neighbor algorithm or the Genetic Algorithm, to find an approximate solution for the TSP in a reasonable amount of time for a large number of cities.**

**Code:**

```
import numpy as np
import random
def calculate_tour_length(tour, dist_matrix):
   return sum(dist_matrix[tour[i]][tour[i  + 1]] for i in range(len(tour) - 1)) + dist_matrix[tour[-1]][tour[0]]
def create_initial_population(pop_size, num_cities): population = []
   for _ in range(pop_size):
      tour = list(range(num_cities))
      random.shuffle(tour)
      population.append(tour)
   return population
def select_parents(population, dist_matrix):
   population.sort(key=lambda tour: calculate_tour_length(tour, dist_matrix))
   return population[:2]  # Select the best two
def crossover(parent1, parent2):
   size = len(parent1)
   start, end = sorted(random.sample(range(size), 2))
   child = [None] * size
   child[start:end] = parent1[start:end]
   for city in parent2:
      if city not in child:
         for i in range(size):
            if child[i] is None:
               child[i] = city
               break
   return child
def mutate(tour):  idx1, idx2 = random.sample(range(len(tour)), 2)
   tour[idx1], tour[idx2] = tour[idx2], tour[idx1]
def genetic_algorithm_tsp(dist_matrix, pop_size=100, generations=500):
   num_cities = dist_matrix.shape[0]
   population = create_initial_population(pop_size, num_cities)
   for _ in range(generations): new_population = []
      while len(new_population) < pop_size:
         parent1, parent2 = select_parents(population, dist_matrix)
         child = crossover(parent1, parent2)
         if random.random() < 0.1:  # Mutation chance
            mutate(child)
         new_population.append(child)
      population = new_population
```

```
    best_tour = min(population, key=lambda tour: calculate_tour_length(tour, dist_matrix))
    return best_tour
if __name__ == "__main__":
    num_cities = 10
    np.random.seed(0)
    dist_matrix = np.random.randint(1, 100, size=(num_cities, num_cities))
    dist_matrix = (dist_matrix + dist_matrix.T) / 2  # Make it symmetric
    best_tour = genetic_algorithm_tsp(dist_matrix)
    best_length = calculate_tour_length(best_tour, dist_matrix)
    print("Best Genetic Algorithm Tour:", best_tour)
    print("Best Tour Length:", best_length)
```

## Output:

Best Genetic Algorithm Tour: [5, 3, 1, 6, 2, 4, 7, 0, 9, 8]

Best Tour Length: 299.0

=== Code Execution Successful ===

# ASSIGNMENT 9

**9) a. Design a Python program to find the shortest path for a monkey to reach the banana in a grid-based maze. The monkey can move up, down, left, or right, and the grid may contain obstacles that the monkey cannot pass through.**

## Code:

```
from collections import deque
def is_valid_move(grid, visited, row, col):
    rows = len(grid)
    cols = len(grid[0])
    return (0 <= row < rows) and (0 <= col < cols) and not grid[row][col] and not visited[row][col]
def bfs_shortest_path(grid, start, goal):
    rows = len(grid)
    cols = len(grid[0])
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    queue = deque([start])
    visited = [[False] * cols for _ in range(rows)]
    visited[start[0]][start[1]] = True
    parent = {start: None}
    while queue: current = queue.popleft()
        if current == goal:  path = []
            while current:  path.append(current)
                current = parent[current]
            return path[::-1]  # Return reversed path
        for direction in directions:
            next_row, next_col = current[0] + direction[0], current[1] + direction[1]
            next_pos = (next_row, next_col)
            if is_valid_move(grid, visited, next_row, next_col):
```

```
            visited[next_row][next_col] = True
            queue.append(next_pos)
            parent[next_pos] = current
    return []  # No path found
def print_path(grid, path):
    for i in range(len(grid)):  row = ""
        for j in range(len(grid[0])):
            if (i, j) in path:
                row += "P "
            elif grid[i][j]:
                row += "# "
            else:  row += ". "
        print(row)
    print()
if __name__ == "__main__":
    grid = [ [0, 0, 0, 0, 0], [0, 1, 1, 1, 0], [0, 0, 0, 1, 0], [0, 1, 0, 0, 0], [0, 0, 0, 0, 0]  ]
    start = (0, 0)  # Starting point (row, col)
    goal = (4, 4)   # Goal point (row, col)
    path = bfs_shortest_path(grid, start, goal)
    if path:  print("Shortest path found:")
        print_path(grid, path)
    else:  print("No path found")
```

## Output:

```
Shortest path found:

P P P P P

. # # # P

. . . # P

. # . . P

. . . . P

=== Code Execution Successful ===
```

**b. Implement a heuristic search algorithm, such as A* search, to find an optimal path for the monkey to reach the banana while avoiding obstacles in the grid.**

## Code:

```
import heapq
class Node:
    def __init__(self, position, parent=None):
        self.position = position  # (x, y) coordinates
        self.parent = parent    # Parent node
        self.g = 0 # Cost from start to this node
        self.h = 0  # Heuristic cost to target
```

```
        self.f = 0  # Total cost (g + h)
    def __lt__(self, other):
        return self.f < other.f
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def a_star_search(start, goal, grid):
    open_list = []
    closed_set = set()
    start_node = Node(start)
    goal_node = Node(goal)
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)
        if current_node.position == goal_node.position:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path
        for new_position in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
neighbor_pos = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1]
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_set):
                neighbor_node = Node(neighbor_pos, current_node)  neighbor_node.g = current_node.g + 1
                neighbor_node.h = heuristic(neighbor_pos, goal_node.position)
                neighbor_node.f = neighbor_node.g + neighbor_node.h
                if not any(neighbor_pos == n.position and neighbor_node.g > n.g for n in open_list):
                    heapq.heappush(open_list, neighbor_node)
    return None
if __name__ == "__main__":
    grid = [ [0, 1, 0, 0, 0], [0, 1, 0, 1, 0], [0, 0, 0, 1, 0], [0, 1, 0, 0, 0],  [0, 0, 0, 1, 0]]
    start = (0, 0)  # Monkey's starting position
 goal = (4, 4)
    path = a_star_search(start, goal, grid)
    if path:
        print("Path to banana:", path)
    else:
        print("No path found!")
```

## Output:

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

=== Code Execution Successful ===