**PRACTICAL 1: BFS & DFS**

**BREADTH-FIRST SEARCHING IN GRAPH**

# Practical No. 01

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

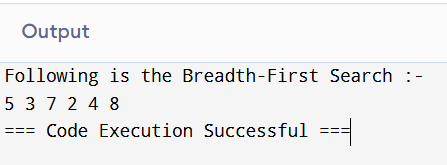
visited.append(neighbour)

queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search :- ")

bfs(visited, graph, '5') # function calling



**DEPTH-FIRST SEARCH**

# Practical No. 01

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

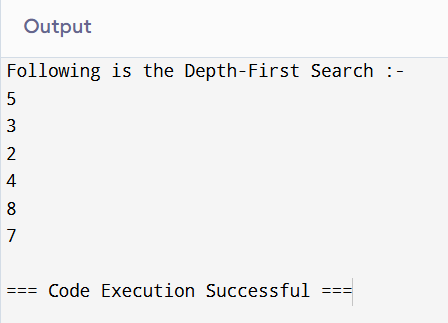
for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search :- ")

dfs(visited, graph, '5')



**PRACTICAL 2: A\* ALGORITHM**

# Practical No. 2

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} #store distance from starting node

parents = {} # parents contains an adjacency map of all nodes

#distance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes

#so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

#node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to first

#n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

else:

if g[m] > g[n] + weight:

#update g(m)

g[m] = g[n] + weight

#change parent of m to n

parents[m] = n

#if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 5,

'D': 7,

'E': 3,

'F': 6,

'G': 5,

'H': 3,

'I': 1,

'J': 0

}

return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

'A': [('B', 6), ('F', 3)],

'B': [('A', 6), ('C', 3), ('D', 2)],

'C': [('B', 3), ('D', 1), ('E', 5)],

'D': [('B', 2), ('C', 1), ('E', 8)],

'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

'F': [('A', 3), ('G', 1), ('H', 7)],

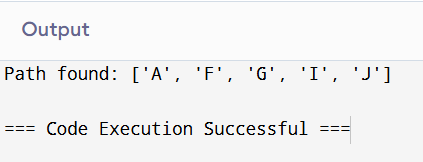
'G': [('F', 1), ('I', 3)],

'H': [('F', 7), ('I', 2)],

'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

aStarAlgo('A', 'J')



**PRACTICAL 3: QUEENS PROBLEM AND GRAPH COLORING**

**QUEENS PROBLEM**

# Practical No. 3

# Taking number of queens as input from user

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

#checking vertically and horizontally

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonally

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

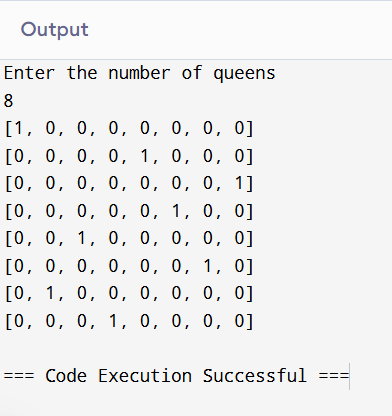
board[i][j] = 0

return False

N\_queens(N)

for i in board:

print (i)



**GRAPH COLORING**

# Practical No. 3

def isSafe(graph, color):

# check for every edge

for i in range(4):

for j in range(i + 1, 4):

if (graph[i][j] and color[j] == color[i]):

return False

return True

def graphColoring(graph, m, i, color):

# if current index reached end

if (i == 4):

# if coloring is safe

if (isSafe(graph, color)):

# Print the solution

printSolution(color)

return True

return False

# Assign each color from 1 to m

for j in range(1, m + 1):

color[i] = j

# Recur of the rest vertices

if (graphColoring(graph, m, i + 1, color)):

return True

color[i] = 0

return False

# /\* A utility function to print solution \*/

def printSolution(color):

print("Solution Exists:" " Following are the assigned colors ")

for i in range(4):

print(color[i], end=" ")

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

# /\* Create following graph and

# test whether it is 3 colorable

# (3)---(2)

# | / |

# | / |

# | / |

# (0)---(1)

# \*/

graph = [

[0, 1, 1, 1],

[1, 0, 1, 0],

[1, 1, 0, 1],

[1, 0, 1, 0],

]

m = 3 # Number of colors

# Initialize all color values as 0.

# This initialization is needed

# correct functioning of isSafe()

color = [0 for i in range(4)]

# Function call

if (not graphColoring(graph, m, 0, color)):

print("Solution does not exist")

