**DAY 2**

**DAY 2 ( DEPTH FIRST SEARCH ) :**

**# 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')**

**DAY 2 ( BREADTH FIRST SEARCH ) :**

**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**

**DAY 2 ( TRAVELLING SALESMAN PROBLEM ) :**

**# Python3 program to implement traveling salesman**

**# problem using naive approach.**

**from sys import maxsize**

**from itertools import permutations**

**V = 4**

**# implementation of traveling Salesman Problem**

**def travellingSalesmanProblem(graph, s):**

**# store all vertex apart from source vertex**

**vertex = []**

**for i in range(V):**

**if i != s:**

**vertex.append(i)**

**# store minimum weight Hamiltonian Cycle**

**min\_path = maxsize**

**next\_permutation=permutations(vertex)**

**for i in next\_permutation:**

**# store current Path weight(cost)**

**current\_pathweight = 0**

**# compute current path weight**

**k = s**

**for j in i:**

**current\_pathweight += graph[k][j]**

**k = j**

**current\_pathweight += graph[k][s]**

**# update minimum**

**min\_path = min(min\_path, current\_pathweight)**

**return min\_path**

**# Driver Code**

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

**# matrix representation of graph**

**graph = [[0, 10, 15, 20], [10, 0, 35, 25],**

**[15, 35, 0, 30], [20, 25, 30, 0]]**

**s = 0**

**print(travellingSalesmanProblem(graph, s)**

**DAY 2 ( A\* ALGORITHM ) :**

**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**

**#ditance 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': 99,**

**'D': 1,**

**'E': 7,**

**'G': 0,**

**}**

**return H\_dist[n]**

**#Describe your graph here**

**Graph\_nodes = {**

**'A': [('B', 2), ('E', 3)],**

**'B': [('C', 1),('G', 9)],**

**'C': None,**

**'E': [('D', 6)],**

**'D': [('G', 1)],**

**}**

**aStarAlgo('A', 'G')**

**DAY 2 ( MAP COLORING TO IMPLEMENT CSP ) :**

**colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black']**

**states = ['Andhra', 'Karnataka', 'TamilNadu', 'Kerala']**

**neighbors = {}**

**neighbors['Andhra'] = ['Karnataka', 'TamilNadu']**

**neighbors['Karnataka'] = ['Andhra', 'TamilNadu', 'Kerala']**

**neighbors['TamilNadu'] = ['Andhra', 'Karnataka', 'Kerala']**

**neighbors['Kerala'] = ['Karnataka', 'TamilNadu']**

**colors\_of\_states = {}**

**def promising(state, color):**

**for neighbor in neighbors.get(state):**

**color\_of\_neighbor = colors\_of\_states.get(neighbor)**

**if color\_of\_neighbor == color:**

**return False**

**return True**

**def get\_color\_for\_state(state):**

**for color in colors:**

**if promising(state, color):**

**return color**

**def main():**

**for state in states:**

**colors\_of\_states[state] = get\_color\_for\_state(state)**

**print (colors\_of\_states);**

**}**