**//Activity 1:**

class node:

def \_\_init\_\_(self, state, parent, actions, totalcost):

self.state = state

self.parent= parent

self.actions actions

self.totalcost = totalcost

graph={ 'A': node('A', None, ['B', 'C', 'E'], None),

'B': node('B', None, ['A', 'D', 'E'], None),

'C': node('C', None, ['A', 'F', 'G', None),

'D': node('D', None, ['B', 'E'], None),

'E': node('E', None, ['A', 'B', 'D'], None),

'F': node('F', None, ['C'], None),

'G': node('G', None, ['C'], None)

}

**//Activity 2:**

class node:

def \_\_init\_\_(self, state, parent, actions, totalcost):

self.state = state

self.parent = parent

self.actions actions

self.totalcost = totalcost

def actionSequence (graph, initialstate, goalstate):

solution = [goalstate]

currentparent graph[goalstate].parent

while currentparent != None:

solution.append(currentparent)

currentparent = graph[currentparent].parent

solution.reverse()

return solution

def bfs(initialstate, goalstate):

graph = {

'A': node('A', None, ['B', 'C', 'E'], None),

'B': node('B', None, ['A', 'D', 'E'], None),

'C': node('C', None, ['A', 'F', 'G'], None),

'D': node('D', None, ['B', 'E'], None),

'E': node('E', None, ['A', 'B', 'D'], None),

'F': node('F', None, ['C'], None),

'G': node('G', None, ['C'], None)

}

frontier = [initialstate]

explored = []

while frontier:

currentnode = frontier.pop(0)

explored.append(currentnode)

for child in graph[currentnode].actions:

if child not in frontier and child not in explored:

graph[child].parent = currentnode

if graph[child].state == goalstate:

return actionSequence (graph, initialstate, goalstate)

frontier.append(child)

return None

solution bfs = ('D', 'C')

print(solution)

['D' , 'B' , 'A', 'C']

**//Activity 3:**

class node:

def \_\_init\_\_(self, state, parent, actions, total cost):

self.state = state

self.parent = parent

self.actions = actions

self.totalcost = totalcost

def actionSequence (graph, initialstate, goalstate):

solution = [goalstate]

currentparent graph[goalstate).parent

while currentparent is not None:

solution.append(currentparent)

currentparent = graph[currentparent].parent

solution.reverse()

return solution

def bfs(initialstate, goalstate):

graph = {

'A': node('A', None, ['B', 'C', 'E'], None),

'B': node('B', None, ['A', 'D', 'E'], None),

'C': node('C', None, ['A', 'F', 'G'], None),

'D': node('D', None, ['B', 'E'], None),

'E': node('E', None, ['A', 'B', 'D'], None),

'F': node('F', None, ['C'], None),

'G': node('G', None, ['C'], None)

}

frontier= [initialstate]

explored = []

while frontier:

currentnode frontier.pop(0)

explored.append(currentnode)

for child in graph[currentnode].actions:

if child not in frontier and child not in explored:

graph[child].parent = currentnode

if graph[child].state == goalstate:

return actionSequence(graph, initialstate, goalstate)

frontier.append(child)

return None

solution = bfs('D', 'C')

print(solution)

['D', 'B', 'A', 'C')

**//Activity 4:**

import heapq

#Define the graph as a dictionary

graph = {

'Arad': [('Zerind', 75), ('Timisoara', 118), ('Sibiu', 140)],

'Zerind': ['Oradea', 71), ('Arad', 75)],

'Oradea': ('Sibiu', 151), ('Zerind', 71)],

'Timisoara: [('Arad', 118), ('Lugoj, 111)],

'Lugoj': ['Timisoara', 111), ('Mehadia', 70)],

'Mehadia': ['Lugoj', 70), ('Drobeta', 75)],

'Drobeta': [('Mehadia', 75), ('Craiova, 120)],

'Sibiu: [('Arad', 140), ('Oradea', 151), ('Fagaras', 99), ('Rimnicu Vilcea', 80)],

'Fagaras': [('Sibiu', 99), ('Bucharest', 211)],

'Rimnicu Vilcea': [('Sibiu', 80), ('Craiova, 146), ('Pitesti', 97)],

'Craiova: [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)],

'Pitesti': ['Rimnicu Vilcea', 97), ('Craiova, 138), ('Bucharest', 101)],

'Bucharest': [('Fagaras", 211), ('Pitesti', 101)]

}

def uniform\_cost\_search(start, goal):

# Keep track of visited nodes and their distances from the start node

visited = {start: @}

# Keep track of the nodes in the path from the start node to the current node

path= (start: [start]}

# Initialize the heap with the start node and its cost

heap= [(0, start)]

while heap:

# Pop the node with the lowest cost from the heap

(cost, current) = heapq.heappop (heap)

# If we have reached the goal node, return the path

if current == goal:

return path[current]

#Loop through the neighboring nodes

for (neighbor, neighbor\_cost) in graph[current]:

# Calculate the new cost to reach the neighboring node

new\_cost = visited [current] + neighbor\_cost

# If the neighboring node hasn't been visited yet or the new cost is lower than the current cost

if neighbor not in visited or new\_cost < visited [neighbor]:

# Update the visited dictionary and the path dictionary

visited[neighbor] = new\_cost

path[neighbor] = path[current] + [neighbor]

# Add the neighboring node and its cost to the heap

heapq.heappush(heap, (new\_cost, neighbor))

return None

start = 'Arad'

goal ='Bucharest'

path = uniform\_cost\_search(start, goal)

print(path)

['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']

**//Activity 4**

import math

def findmin(frontier):

minV = math.inf

node = ''

for i in frontier:

if minV > frontier[1][1]:

minV=frontier[1][1]

node = i

return node

def actionSequence (graph, initialstate, goalstate):

solution [goalstate]

currentparent graph[goalstate].parent

while currentparent is not None:

solution.append(currentparent)

currentparent = graph[currentparent].parent

solution.reverse()

return solution

class node:

def \_\_init\_\_(self, state, parent, actions, totalcost):

self.state = state

self.parent = parent

self.actions = actions

self.totalcost = totalcost

def UCS(initialstate, goalstate):

graph = {

'A': node('A', None, [('B', 6), ('C', 9), ('E', 1)], 0),

'B': node('B', None, [('A', 6), ('D' ,3), ('E', 4)], 0),

'C': node('C', None, [('A', 9), ('F' ,2), ('G', 3)], 0),

'D': node('D', None, [('B', 3), ('E', 5), ('F', 7)], 0),

'E': node('E', None, [('A', 1), ('B', 4),('D', 5) , ('F', 6)], 0),

'F': node('F', None, [('C', 2), ('E', 6), ('D', 7)], 0),

'G': node('G', None, [('C', 3)], 0)

frontier dict()

frontier[initialstate] = (None, 0)

explored = []

while frontier:

currentnode = findmin (frontier)

del frontier[currentnode]

if graph[currentnode].state == goalstate:

return actionSequence (graph, initialstate, goalstate)

explored.append(currentnode)

for child in graph[currentnode].actions:

currentcost child [1] + graph[currentnode].totalcost

if child[e] not in frontier and child[e] not in explored:

graph[child[0]].parent = currentnode

graph[child[0]].totalcost = currentcost

frontier[child[0]] = (graph[child[0]].parent, graph[child[8]].totalcost)

elif child[0] in frontier:

if frontier[child[0]][1] > currentcost:

graph[child[0]].parent = currentnode:

graph[child[e]].totalcost = currentcost

frontier[child[@]] = (graph[child[0]].parent, graph[child[@]].totalcost)

solution = UCS('C', 'B')

print(solution)

G['C', 'F', 'E', 'B']