LAB MID 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' , 'E' , 'C'], None),
         'B': Node('B', None, ['D' , 'E' , 'A'], 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),
         'F': 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 dfs(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 = []
  currentChildren = 0
  while frontier:
    currentnode = frontier.pop(len(frontier)-1)
    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:
          # print(explored)
          return actionSequence(graph,initialstate,goalstate)
        currentChildren=currentChildren+1
        frontier.append(child)
  if currentChildren == 0 :
    del explored[len(explored)-1]
solution = dfs('A','D')
print(solution)
```

```
['A', 'E', 'D']
```

ACTIVITY 3

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

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

ACTIVITY 4

```
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', 6), ('C',9), ('E',1)], 0),
         'B': Node('B', None, [('A', 6), ('D',3), ('E',4)], 0),
         'C': Node('C', None, [('A',9), ('E',2), ('G',3)], 0),
         'D': Node('D', None, [('B',3), ('E',5), ('E',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))
import math
def findMin (frontier):
  minV=math.inf
  node=''
  for i in frontier:
    if minV>frontier[i][1]:
      minV-frontier [i] [1]
```

```
        node = i
    return node
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
class Node:
    def __init__(self, state, parent, actions, totalCost):
      self.state = state #instance variable unique to each instance
      self.parent = parent
      self.actions = actions
      self. totalCost = totalCost
def UCS ():
    initialState = 'C'
    goalState = 'B'
    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), ('E',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 len(frontier) != 0:
    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[0] not in frontier and child[0] not in explored:
          graph [child[0]].parent-currentNode
          graph[child[0]].totalCost-currentCost
          frontier [child[0]]=(graph [child[0]].parent, graph[child[0]].totalCost)
        elif child[0] in frontier:
          if frontier[child[01][1] < currentCost:
            graph [child[0]] parent frontier [child[0]][0]
            graph[child[01].totalCost frontier [child[0]][1]
          else:
            frontier [child[0]]=(currentNode, currentCost)
            graph (child[0]].parent=frontier [child[0]][0]
            graph[child[0]].totalCost frontier (child[0]] [1]

         File "<ipython-input-26-2ef954aa231e>", line 63
           if frontier [child[01][1] < currentCost:
                                     ^
      SyntaxError: leading zeros in decimal integer literals are not permitted; use an 0o
      prefix for octal integers
```

SEARCH STACK OVERFLOW

HOME ACTIVITY

```
import heapq
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):
        visited = {start: 0}
        path = {start: [start]}
        heap = [(0, start)]
        while heap:
            (cost, current) = heapq.heappop(heap)
            if current == goal:
                return path[current]
            for (neighbor, neighbor_cost) in graph[current]:
                new_cost = visited[current] + neighbor_cost
                if neighbor not in visited or new_cost < visited[neighbor]:
                    visited[neighbor] = new_cost
                    path[neighbor] = path[current] + [neighbor]
                    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']
```

0s completed at 4:05 AM