```
M Kaleem ullah
FA20-BSE-013
```

Actvity 1: Consider a toy problem that can be represented as a following graph. How would you represent this graph in python?

```
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: For the graph in previous activity, imagine node A as starting node and your goal is to reach F. Keeping depth first search in mind, describe a sequence of actions that you must take to reach that goal state.

```
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: Change initial state to D and set goal state as C. What will be resulting path of BFS search? What will be the sequence of nodes explored?

```
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: Imagine the same tree but this time we also mention the cost of each edge. mplement a uniform cost solution to find the path from C to B.

```
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,intialstate,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
       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[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[0]][1] < currentcost:
        graph[child[0]].parent = frontier[child[0]][0]
        graph[child[0]].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]
    solution = UCS('C','B')
    print(solution)
```

```
  File "<ipython-input-13-d59ba81d73e4>", line 41
    return: actionSequence(graph,initialstate,goalstate)
    ^
IndentationError: expected an indented block after 'if' statement on line 40
```

SEARCH STACK OVERFLOW

Home Activity:Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a uniform cost search to find the corresponding path.

```
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: 0}
    # 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']
```

0s completed at 4:02 AM