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)
          }
```

home activity:

```
from queue import PriorityQueue

# Define the graph as a dictionary of dictionaries
graph = {'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
         'Zerind': {'Arad': 75, 'Oradea': 71},
         'Oradea': {'Zerind': 71, 'Sibiu': 151},
         'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
         'Timisoara': {'Arad': 118, 'Lugoj': 111},
         'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
         'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
         'Drobeta': {'Mehadia': 75, 'Craiova': 120},
         'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
         'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
         'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
         'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
         'Bucharest': {'Fagaras': 211, 'Pitesti': 101}}

def uniform_cost_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put((0, start))
    explored = []
    path = {}
    path[start] = None

    while not frontier.empty():
        cost, current_node = frontier.get()
        explored.append(current_node)

        if current_node == goal:
            final_path = []
            while current_node in path:
                final_path.append(current_node)
                current_node = path[current_node]
            final_path.reverse()
            return final_path

        for neighbor, neighbor_cost in graph[current_node].items():
            if neighbor not in explored:
                new_cost = cost + neighbor_cost
                if neighbor not in [node[1] for node in frontier.queue]:
                    frontier.put((new_cost, neighbor))
                    path[neighbor] = current_node
                elif new_cost < [node[0] for node in frontier.queue if node[1] == neighbor][0]:
                    frontier.get([node for node in frontier.queue if node[1] == neighbor][0])
                    frontier.put((new_cost, neighbor))
                    path[neighbor] = current_node

    return None

# Test the uniform cost search algorithm
start_node = 'Arad'
goal_node = 'Bucharest'
```

```
result_path = uniform_cost_search(graph, start_node, goal_node)

if result_path:
    print("The minimum distance path from", start_node, "to", goal_node, "is:")
    print(result_path)
    print("The total distance is:", sum(graph[result_path[i]][result_path[i+1]] for i in range(len(result_path)-1)))
else:
    print("Goal not reachable from the starting node")
```

```
The minimum distance path from Arad to Bucharest is:
['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
The total distance is: 418
```

Double-click (or enter) to edit

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

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

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 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 4:

```
import heapq

tree = {
    'C': [('A', 3), ('D', 2)],
    'A': [('B', 5)],
    'B': [],
    'D': [('E', 4), ('F', 6)],
    'E': [],
    'F': [('G', 1)],
    'G': []
}

def uniform_cost(start, goal):
    # Initialize the PQ and visited dictionary
    pq = [(0, start)]
    visited = {start: 0}
    while pq:
        # Get the node with the lowest cost from the PQ
        (cost, current) = heapq.heappop(pq)
        # If we reach the goal, return the path and its cost
        if current == goal:
            path = []
            while current in visited:
                path.insert(0, current)
                current = visited[current][1]
            return (path, visited[goal])
        # Explore the neighbors of the current node
        for (neighbor, neighbor_cost) in tree[current]:
            neighbor_cost += cost
            if neighbor not in visited or neighbor_cost < visited[neighbor]:
                visited[neighbor] = (neighbor_cost, current)
                heapq.heappush(pq, (neighbor_cost, neighbor))
```

```
    # If we reach here, there is no path between the start and goal
    return None

# Test the implementation
path, cost = uniform_cost('C', 'B')
print('The path from C to B is:', path)
print('The cost of the path is:', cost)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-3635b12998e3> in <cell line: 38>()
     36 
     37 # Test the implementation
---> 38 path, cost = uniform_cost('C', 'B')
     39 print('The path from C to B is:', path)
     40 print('The cost of the path is:', cost)

<ipython-input-13-3635b12998e3> in uniform_cost(start, goal)
     24             while current in visited:
     25                 path.insert(0, current)
---> 26                 current = visited[current][1]
     27             return (path, visited[goal])
     28         # Explore the neighbors of the current node

TypeError: 'int' object is not subscriptable
```

SEARCH STACK OVERFLOW

0s completed at 3:18 PM