LAB # 4

Statement Purpose:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **Depth First Search** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node. Students will also see the working of Uniform Cost Search.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using Breadth First Search.
- How to find a solution of those problems using Uniform Cost Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Stage J (Journey)

Introduction:

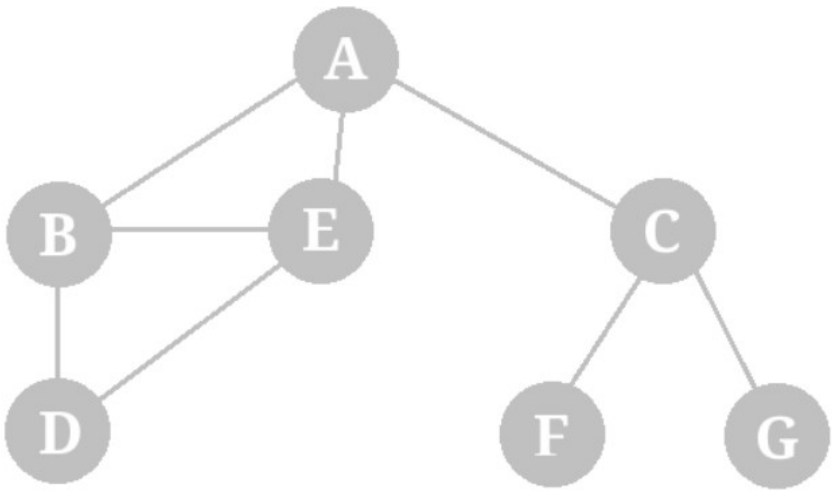
In the previous lab we studied breadth first search, which is the most naïve way of searching trees. The path returned by breadth search is not optimal. Uniform cost search always returns the optimal path. In this lab students will be able to implement a uniform cost search approach. Students will also be able to apply depth first search to their problems.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

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



Solution:

Remember a node of a tree can be represented by four attributes. We consider node as a class having four attributes namely,

1- State of the node
2- Parent of the node
3- Actions applicable to that node (In the book/theory class, this attribute referred to the parent action that generated current node, however in this lab we will use this attribute to refer to all possible children states that are achievable from current state).
4- The total path cost of that node starting from the initial state until that particular node is reached

```
class Node:
    #state = state          # class variable shared by all instances
    def __init__(self, state, parent, actions, totalCost):
        self.state = state    # instance variable unique to each instance
        self.parent = parent
        self.actions = actions # we are not saving actions themselves,
                               # only output states of those actions
        self.totalCost = totalCost
```

We can now implement this class in a dictionary. This dictionary will represent our state space graph. As we will traverse through the graph, we will keep updating parent and cost of each node.

```
# we think of a graph as a dictionary, items comprise of nodes, where
# each node has a key and a value. Key is simply the state of the node
# and value are actual attributes that node object

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),
         '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.

Solution:

Remember that we can implement depth first search simply by using LIFO approach instead of FIFO that was used in breadth first search. Additionally we also don't keep leaf nodes (nodes without children) in explored set.

```
def DFS():
    initialState = 'A'
    goalState = 'D'

    # we think of a graph as a dictionary, items comprise of nodes, where
    # each node has a key and a value. Key is simply the state of the node
    # and value are actual attributes that node object

    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),
             'G': Node('G', None, ['C'], None)}

    frontier = [initialState]
    explored=[]
    while len(frontier)!=0:
        currentNode = frontier.pop(len(frontier)-1)
        print(currentNode)
        explored.append(currentNode)
        currentChildren=0
        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]
```

Now the function definition is complete which can be called as follows,

```
solution = DFS()
print(solution)
```

Notice the difference in two portions of the code between breadth first search and depth first search. In the first we just pop out the last entry from the queue and in the 2nd difference we delete leaf nodes from the graph.

```
def actionSequence(graph, initialState, goalState):
    # returns a list of states starting from goal state moving upwards towards
    # parents until root node is reached
    solution=[goalState]
    currentParent=graph[goalState].parent
    while currentParent!=None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

Calling DFS() will return the following 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?

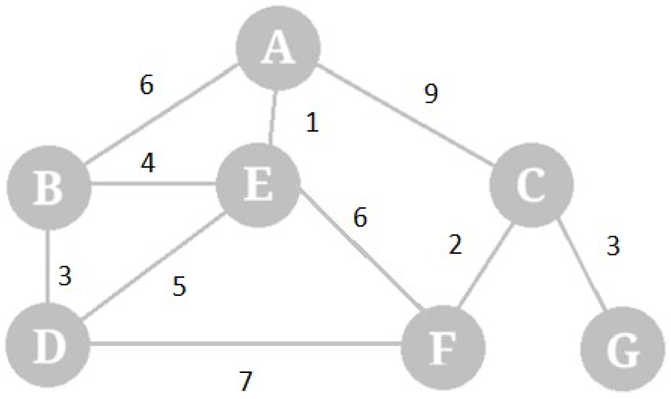
Solution:

Final path is ['D', 'B', 'A', 'C']

Explored node sequence is D, E, A.

Activity 4:

Imagine the same tree but this time we also mention the cost of each edge.



Implement a uniform cost solution to find the path from C to B.

Solution:

First we modify our graph structure so that in Node.actions is an array of tuples where each tuple contains a vertex and its associated weight.

```
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)}
```

We also modify the frontier format which will now be a dictionary. This dictionary will contain each node (the state of the node will act as a key and its parent and accumulated cost from the initial state will be two attributes of a particular key). We now define a function which will give the node/key for which the cost is minimum. This will be implementation of pop method from the priority queue.

```
import math

def findMin(frontier):
    # returns that node in the frontier which has a lowest cost
    minV=math.inf
    node=''
    for i in frontier:
        if minV>frontier[i][1]:
            minV=frontier[i][1]
            node = i
    return node
```

The rest of the functions are the same as in BFS i.e.,

```
def actionSequence(graph, initialState, goalState):
    # returns a list of states starting from goal state moving upwards towards
    # parents until root node is reached
    solution=[goalState]
    currentParent=graph[goalState].parent
    while currentParent!=None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution

class Node:
    #state = state          # class variable shared by all instances
    def __init__(self, state, parent, actions, totalCost):
        self.state = state    # instance variable unique to each instance
        self.parent = parent
        self.actions = actions # we are not saving actions themselves,
                               # only output states of those actions
        self.totalCost = totalCost
```

Finally we define UCS()

```
def UCS():
    initialState = 'C'
    goalState = 'B'

    # we think of a graph as a dictionary, items comprise of nodes, where
    # each node has a key and a value. Key is simply the state of the node
    # and value are actual attributes that node object
##
    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=[] # parent of initial node is None and its cost is 0
```

```
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[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()
print(solution)
```

The above two lines will print

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

3) Stage v (verify)

Activity 1:

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.

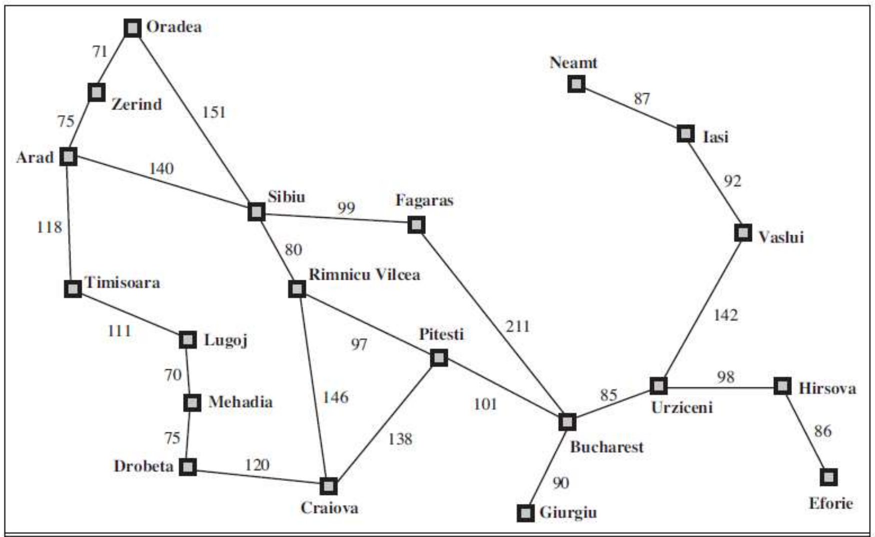


Figure 3.2 A simplified road map of part of Romania.

4) Stage a2 (assess)

Assignment:

Create a graph and then set initial and goal states such that the number of nodes visited for BFS is smaller than that in DFS. Now modify the initial and goal state such that the number of nodes visited for BFS is larger than that in DFS.