LAB # 3

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 **Breadth 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.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to use find a solution of those problems using Breadth First 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:

Sometimes, very different-sounding problems turn out to be similar when you think about how to solve them. What do Pac-Man, the royal family of Britain, and driving to Orlando have in common? They all involve route-finding or path-search problems:

How is the current Prince William related to King William III, who endowed the College of William and Mary in 1693?

What path should a ghost follow to get to Pac-Man as quickly as possible?

What's the best way to drive from Dallas, Texas to Orlando, Florida?

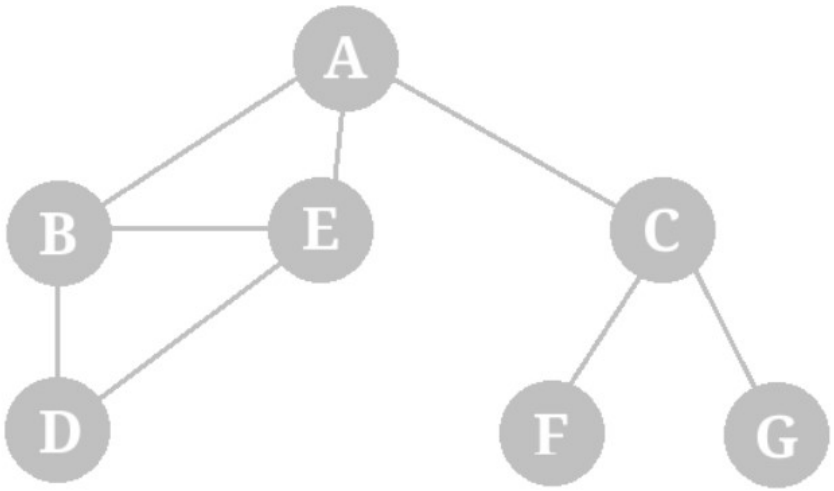
We have to be given some information to answer any of these questions. This information about each question can be represented in terms of graph. This lab will enable students to represent problems in terms of graph and then finding a solution in terms of a path using breadth first search.

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
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', '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 breadth first search in mind, describe a sequence of actions that you must take to reach that goal state.

Solution:

Remember that in theory class, we discussed the following implementation of breadth first search.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

What follows is python implementation of above mentioned algorithm,

```
def BFS():
    initialState = 'D'
    goalState = 'F'

    # 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', '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 len(frontier)!=0:
        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)
```

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

```
solution = BFS()
print(solution)
```

There is one additional line of graph[child].parent=currentNode in python code which is missing in pseudocode. This allows us to update each parent of a node as we traverse the graph. Afterwards, the function actionSequence() is called which returns a series of actions when a goal state is reached. Given a start state, end state and a graph, this function recursively iterated through each parent until the starting state is reached.

```
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 BFS() will return the following solution,
['A', 'C', 'F']

Activity 3:

Change initial state to D and set goal state as C. What will be resulting path of BFS search?

Can we calculate cost as well?

Solution:

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

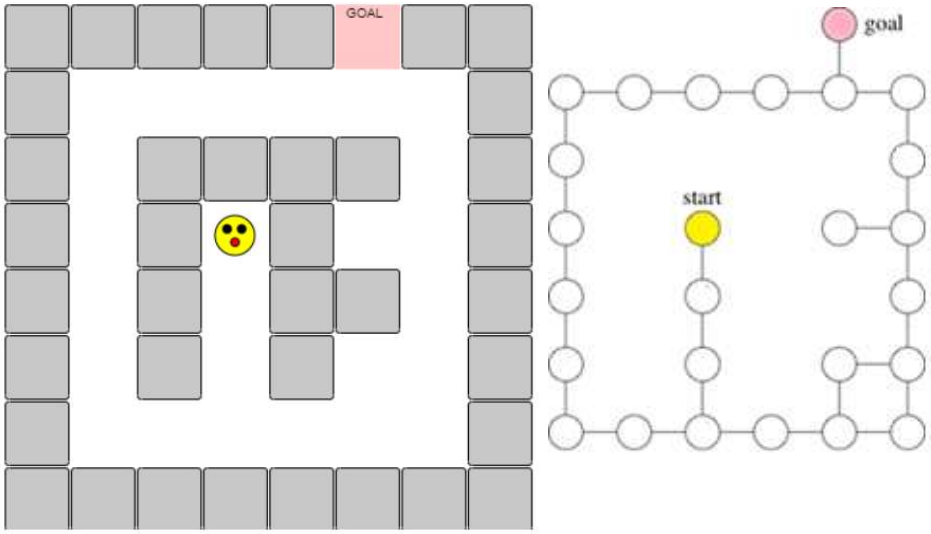
3) Stage a2 (assess)

Assignment:

Consider a maze as shown below. Each empty tile represents a separate node in the graph. There are maximum of four possible actions i.e., to move up, down, left or right on any given tile/node. Using BFS, find out how to get out of the maze if you're in the start position depicted below.

1. Design a state space
2. Design the possible actions
3. Specify cost with actions
4. Create graph:
   We consider node as a class having four attributes namely,
   5- State of the node
   6- Parent of the node
   7- Actions applicable to that node
   8- The total path cost of that node starting from the initial state until that particular node is reached

   No need to run it, just create a representation [can be paper based as well]



4) Stage v (verify)

Activity 1:

Imagine going from Arad to Bucharest in the following map. Implement a BFS to find the corresponding path.

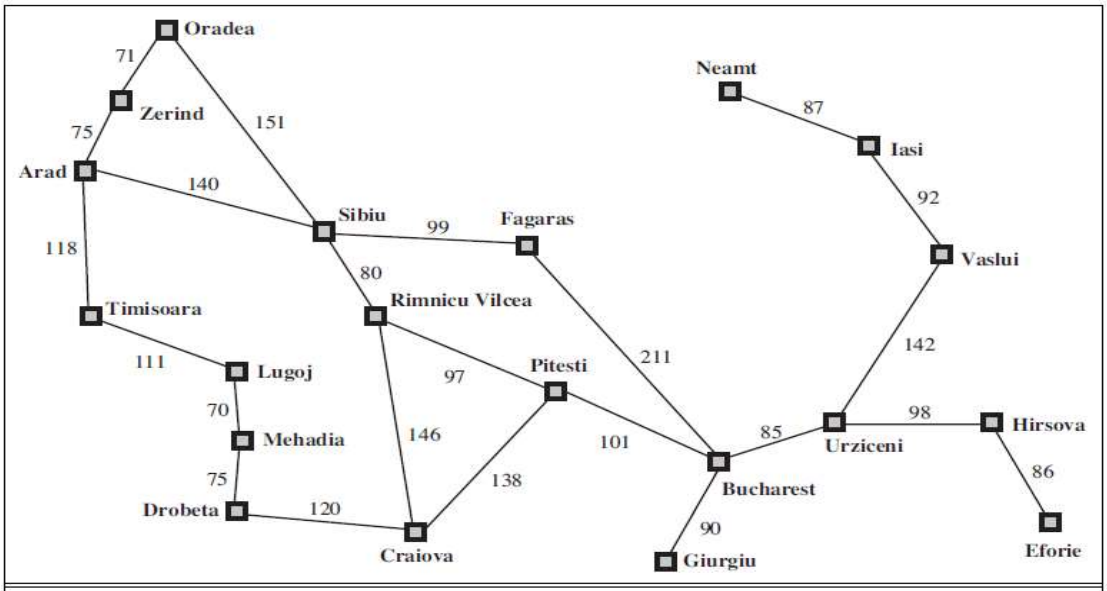


Figure 3.2 A simplified road map of part of Romania.