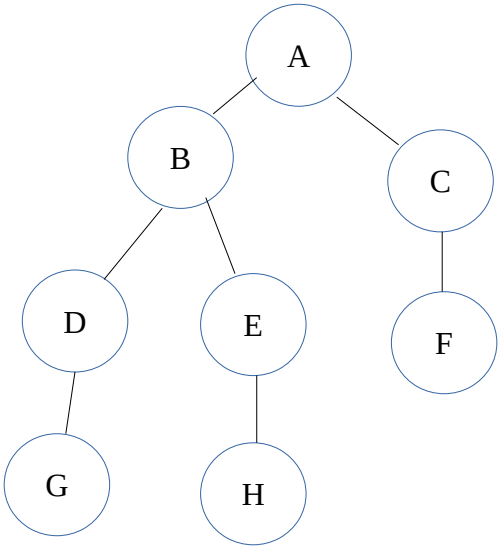
Lab 3: Problem Solving by Searching: DFS and BFS

An uninformed search algorithm generates the search tree without using any domain specific knowledge. Uninformed search algorithms are also called blind search algorithms. The search algorithm produces the search tree without using any domain knowledge, which is a brute force in nature. They don't have any background information on how to approach the goal or whatsoever. But these are the basics of search algorithms in AI. In this lab we will implement the following searching strategies in the graphs shown in figure below.

Depth First Search (DFS): It is a search algorithm where the search tree will be traversed from the root node. It will be traversing, searching for a key at the leaf of a particular branch. If the key is not found the searching retraces its steps back to the point from where the other branch was left unexplored and the same procedure is repeated for that other branch.

Breadth-First Search(BFS): It is another search algorithm in AI which traverses breadth-wise to search the goal in a tree. It begins searching from the root node and expands the successor node. It further expands along breadth-wise and traverses those nodes rather than searching depth-wise.



Our goal is to find a path from node **A** to node **H**

Source Code:

```
#We represent the graph in adjacent list
#representation as dictionary of lists
G  = {
  'A' : ['B', 'C'],
  'B' : ['A', 'D', 'E'],
  'C' : ['A', 'F'],
  'D' : ['B', 'G'],
  'E' : ['B', 'D', 'G', 'H'],
  'F' : ['C'],
  'G' : ['D', 'E'],
  'H' : ['E']
  }

#Specify start node and goal node
start = 'A'
goal = 'H'

def DFS(G, start, goal):
  #We maintain a stack which stores a list
  #containing vertices which are not visited
  stack = [start]
  #We maintain a set of visited vertices
  visited = set()
  #We maintain a dictionary to keep track of
  #previous vertex
  previousV = dict()
  #The starting vertex has no previous vertex
  previousV[start] = ""
  #Repeat the following until the stack is empty
  while stack:
    poppedVertex = stack.pop()
    if poppedVertex not in visited:
      #If the popped vertex is the goal we return True
      if poppedVertex == goal:
        return (True,previousV)
      visited.add(poppedVertex)
      for vertex in G[poppedVertex]:
        if vertex not in visited and vertex not in stack:
          stack.append(vertex)
          previousV[vertex] = poppedVertex
  #If no goal found until the end we return False
  return (False,previousV)
```

```
def BFS(G, start, goal):
  #We maintain a queue which stores a list
  #containing vertices which are not visited
  queue = [start]
  #We maintain a set of visited vertices
  visited = set()
  #We maintain a dictionary to keep track of
  #previous vertex
  previousV = dict()
  #The starting vertex has no previous vertex
  previousV[start] = ""
  #Repeat the following until the stack is empty
  while queue:
    poppedVertex = queue.pop(0)
    if poppedVertex not in visited:
      #If the popped vertex is the goal we return True
      if poppedVertex == goal:
        return (True,previousV)
      visited.add(poppedVertex)
      for vertex in G[poppedVertex]:
        if vertex not in visited and vertex not in queue:
          queue.append(vertex)
          previousV[vertex] = poppedVertex
  #If no goal found until the end we return False
  return (False,previousV)

def constructPath(previous,start,goal):
  temp = goal
  path = temp
  while(previous[temp] != ""):
    path = previous[temp] + "->" + path
    temp = previous[temp]
  return path

#Driver Code
if __name__ ==  "__main__":
  print("-------------------------------------")
  print("Using DFS")
  pathFound,previous = DFS(G, start, goal)
  if(pathFound):
    print(constructPath(previous, start, goal))
  else:
```

```
    print("Path Not Found")
  print("-------------------------------------")
  print("Using BFS")
  pathFound,previous = BFS(G, start, goal)
  if(pathFound):
    print(constructPath(previous, start, goal))
  else:
    print("Path Not Found")
```

Output:

```
----------------------------------------
Using DFS
A->B->E->H
----------------------------------------
Using BFS
A->B->E->H
```

Source Code: