DEPARTMENT OF
COMPUTER SYSTEMS ENGINEERING
MEHRAN UNIVERSITY OF ENGINEERING AND
TECHNOLOGY, JAMSHORO

ARTIFICIAL INTELLIGENCE

PRACTICAL 02

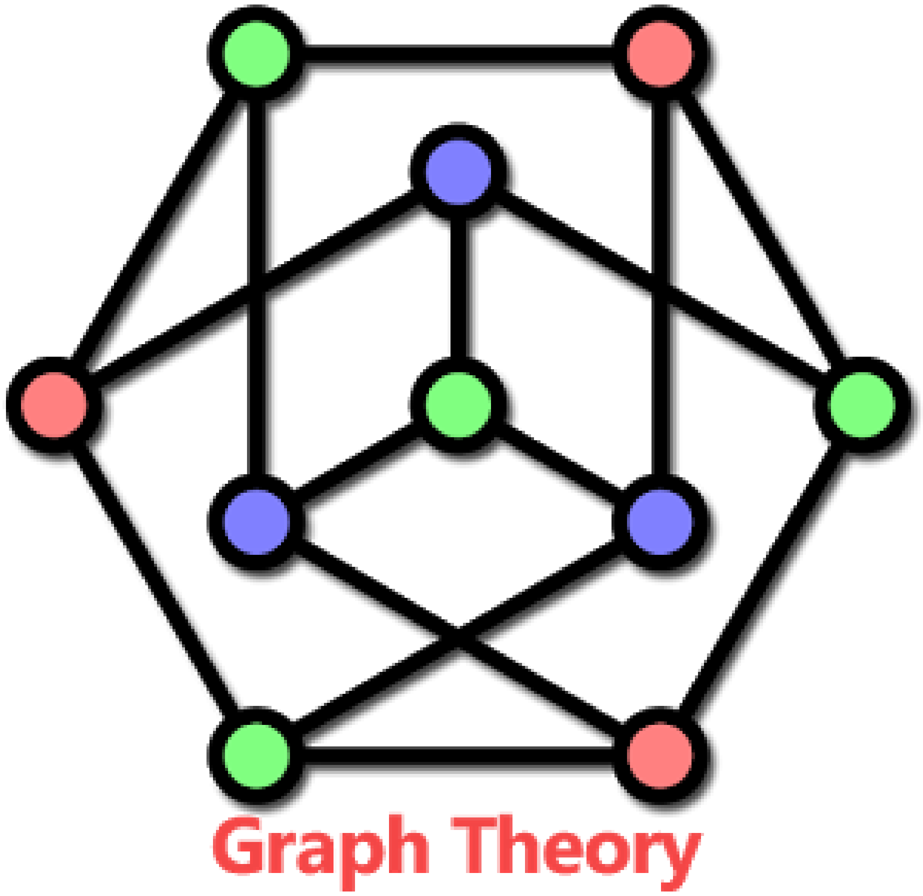
OBJECT: INTRODUCTION TO SEARCHING PROBLEMS, REPRESENTATION OF GRAPHS

Outline

- Fundamentals of graphs
- Applications of graphs
- Creation of graphs using dictionaries
- Generating edges
- Find isolated nodes
- Finding path between two vertex/ node
- Finding all paths in graphs
- Finding shortest path between nodes
- Determine cycles in graphs
- Add an edge
- Find degree of vertex
- Find if the graph is connected

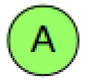
Required Tools

- PC with windows
- Anaconda environment



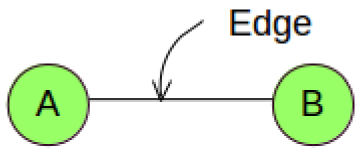
FUNDAMENTALS OF GRAPHS

Vertex:



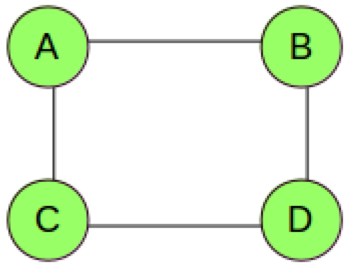
It is a point where multiple lines meet. It's also known as the **node**.
A vertex is generally denoted by an alphabet as shown above.

Edge:



It is a line that connects two vertices.

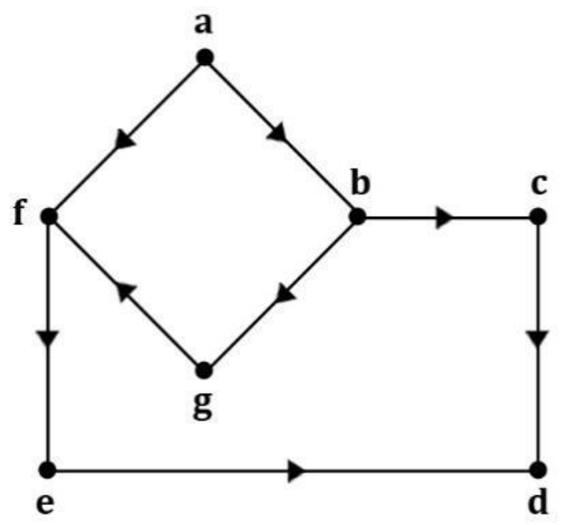
Graph:



As discussed in the previous section, graph is a combination of vertices (nodes) and edges.
G = (V, E) where V represents the set of all vertices and E represents the set of all edges of the graph.

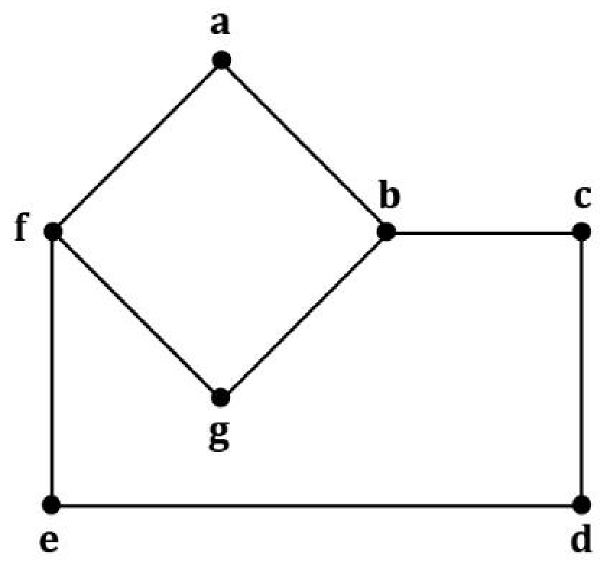
FUNDAMENTALS OF GRAPHS

- **Directed graphs**
  have edges with direction. The edges indicate a one-way relationship, in that each edge can only be traversed in a single direction.



Directed Graph

- **Undirected graphs**
  have edges that do not have a direction. The edges indicate a two-way relationship, in that each edge can be traversed in both directions.



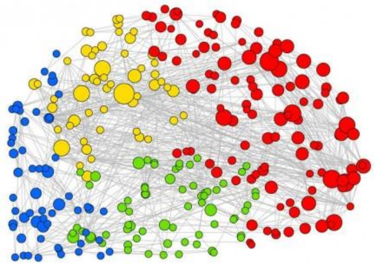
Undirected Graph

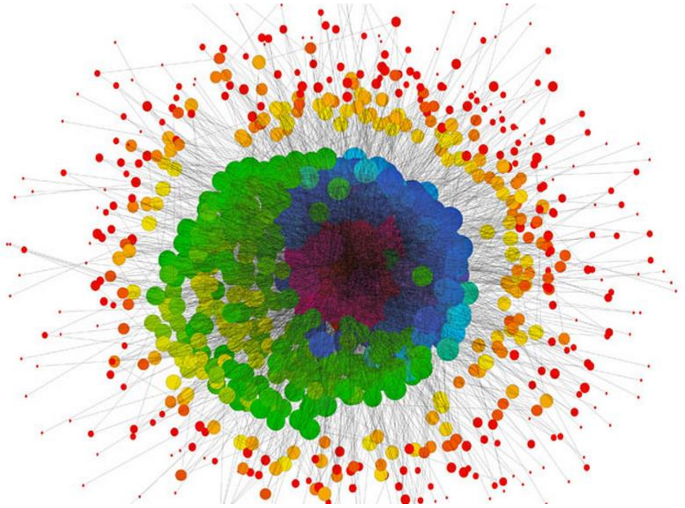
APPLICATIONS OF GRAPHS

- Many practical problems can be represented by graphs.
- They are often used to model problems or situations in physics, biology, psychology and above all in computer science.
- In computer science, graphs are used to represent:

✓ Networks of communication,
✓ Data organization,
✓ Computational devices,
✓ The flow of computation,
The are used to represent the data organization like the file system of an operating system
✓ Communication networks.

Airport network graph


Brain graph


Data graph

APPLICATIONS OF GRAPHS

1. THE INTERNET:
ONE OF THE LARGEST GRAPHS IN LIFE.

In social network like facebook, whatsapp, messenger, twitter, google+, linkedin etc. users are connected via a large graph.

- Facebook Friend
- Twitter follower
- Page Ranking
- Scientific Computation (Atom, Protein, etc)
- Network Traffic flow/Shortest path/Minimum spanning tree
- Website analysis
- Biological analysis
- VLSI

APPLICATIONS OF GRAPHS

2. IN FLIGHT NETWORKS:

For Airlines to connect countless cities in the most efficient way.

For air traffic controllers to avoid crash.

APPLICATIONS OF GRAPHS

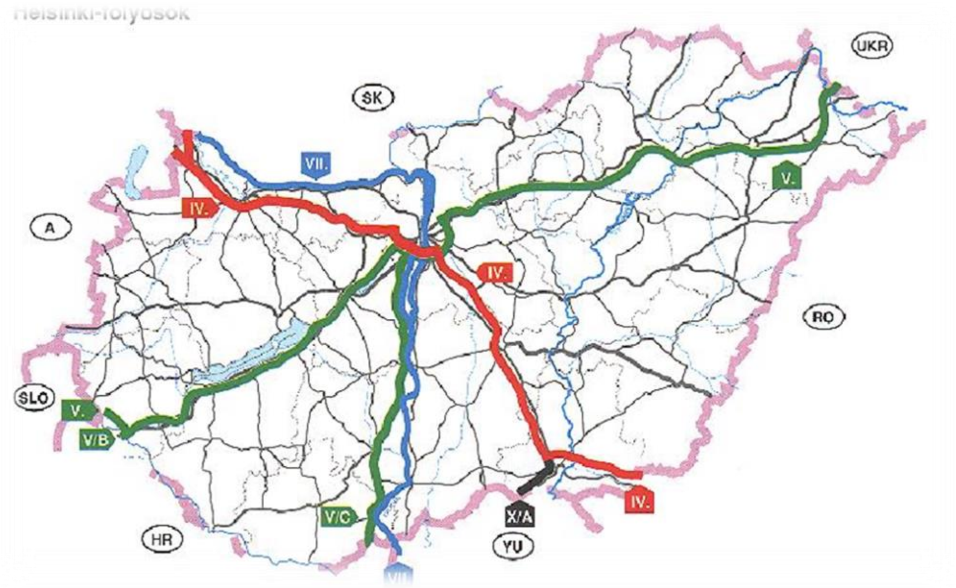
3. ROAD NETWORKS:

To develop Intelligent Transportation Systems

To reduce traffic congestion.

To travel faster.

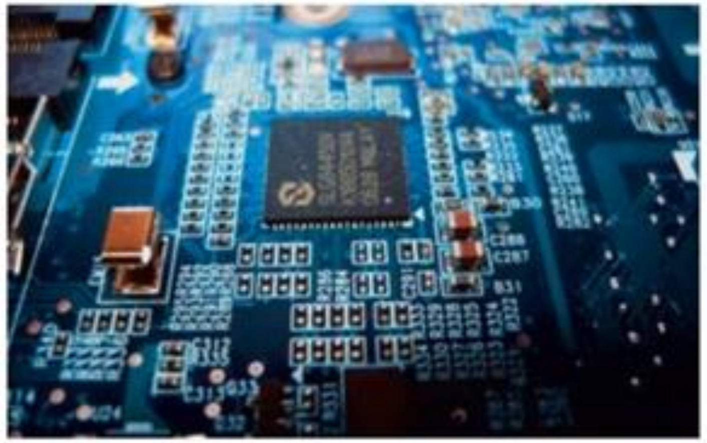
To avoid car accidents.

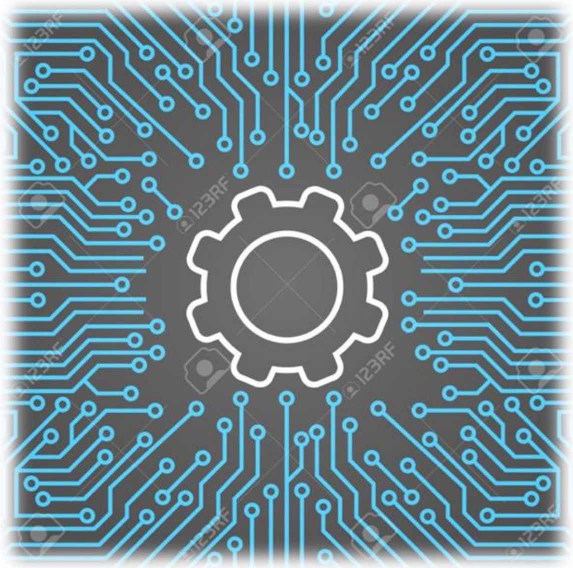


APPLICATIONS OF GRAPHS

4. DESIGN OF COMPUTER CHIPS.

To connect millions of transistors that consist in Integrated circuits .



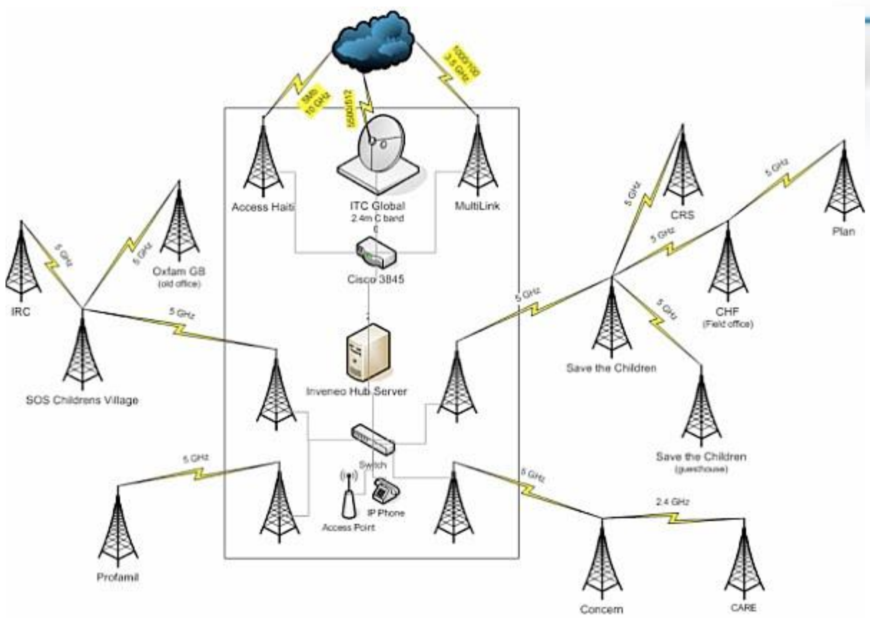


APPLICATIONS OF GRAPHS

5. COMPUTER NETWORK SECURITY

6. GSM MOBILE PHONE NETWORKS

7. MAP COLORING



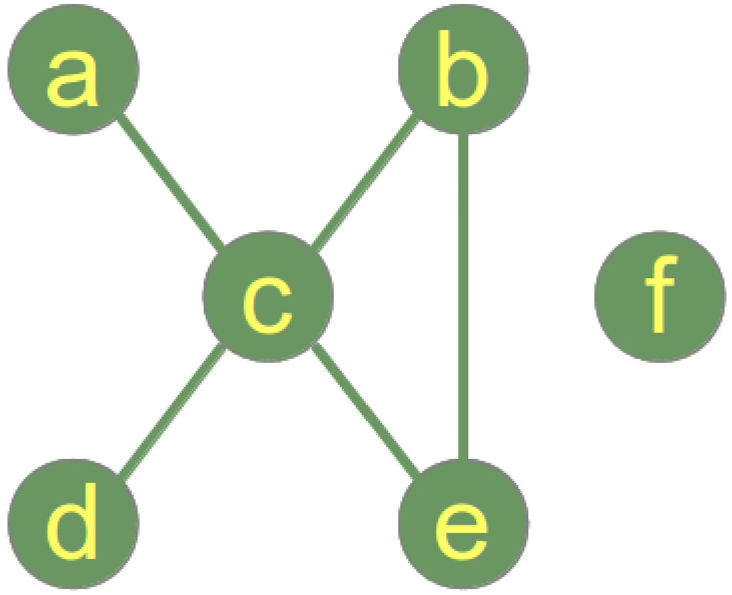
GENERATE GRAPH USING PYTHON

Python has no built-in data type or class for graphs, but it is easy to implement them in Python.

One data type is ideal for representing graphs in Python, i.e. **dictionaries(**A **dictionary** is a collection which is unordered, changeable and indexed. In **Python dictionaries** are written with curly brackets, and they have keys and values.**).**

Example 1: The graph in our illustration can be implemented in the following way:

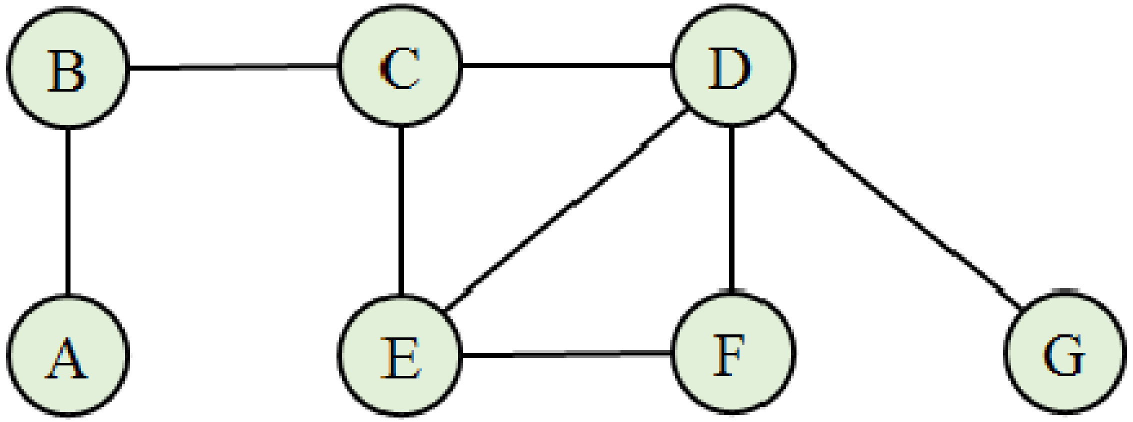
```
graph =
{
"a" : ["c"],
 "b" : ["c", "e"],
"c" : ["a", "b", "d", "e"],
"d" : ["c"],
"e" : ["c", "b"],
"f" : []
}
```



GENERATE GRAPH USING PYTHON

```
print(graph["a"])

print("The nodes/vertices of graph:", graph.keys() )
#Returns list of dictionary keys

print("The edges of the graph for keys{0} are {1}".format( graph.keys(), graph.values()))
#Returns list of dictionary values
```

TASK1: Generate the output of above code & Use the same code for following graph.



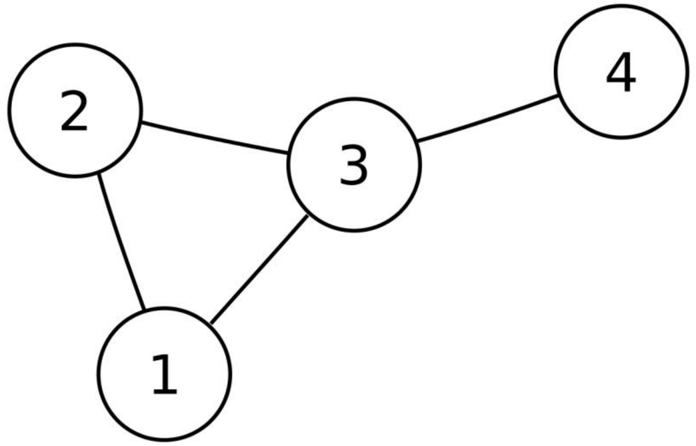
GENERATING EDGES

We can create a user-defined function to generate the list of all edges present in graph.

Example 2

```
#Generating Edges
graph = {                "a" : ["c"],
                         "b" : ["c", "e"],
                         "c"  : ["a", "b", "d", "e"],
                          "d" : ["c"],
                          "e" : ["c", "b"],
                          "f" : []
             }
def generate_edges(graph):
   edges = [] # empty list
   for node in graph: #traversing through graph
      for neighbour in graph[node]: # takes particular node
         edges.append((node, neighbour))
   return edges

print("Generating Edges:",generate_edges(graph)) #using function
```

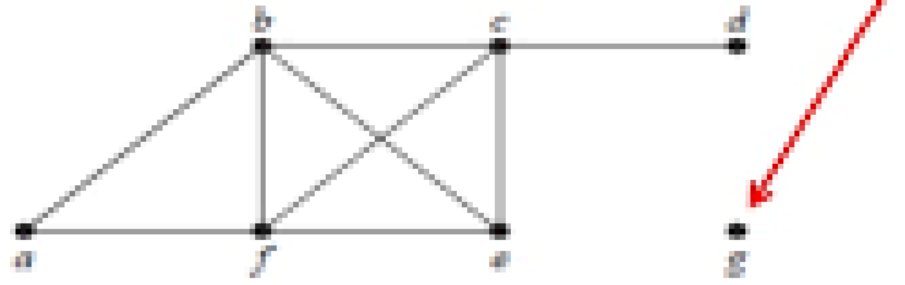
TASK2: Generate the output of above code & Use the same code to find edges of following graph.



What is an isolated node?

Isolated vertex

- **A vertex of degree zero** is called **isolated.** It follows that an isolated vertex is not adjacent to any vertex.



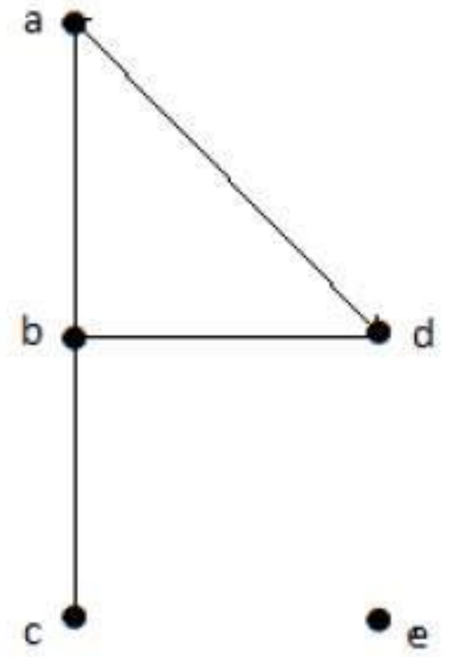
FINDING ISOLATED NODES

As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph.
The following Python function calculates the isolated nodes of a given graph:

Example 3

```
#Finding Isolated nodes

def  find_isolated_nodes(graph):
    """ returns a list of isolated nodes. """
    isolated = []
    for node in graph: #traverse each node
        if not graph[node]: #value for this node is nothing
            isolated += node # add the node to list
    return isolated

print("Isolated nodes", find_isolated_nodes(graph))
```

TASK3: Generate the output of above code & Use the same code to find isolated node of following graph.



FINDING PATHS BETWEEN TWO NODES/ VERTEX

Let's write a simple function to determine a path between two nodes. It takes a graph and the start and end nodes as arguments.

It will return a list of nodes (including the start and end nodes) comprising the path. When no path can be found, it returns None. The same node will not occur more than once on the path returned.

The algorithm uses an important technique called *backtracking*: it tries each possibility in turn until it finds a solution.

TASK4: Generate the output of above code & Use the same code to find the path between 'd' and 'b' in the same graph.

Example 4

```
#Finding path between two nodes
def find_path(graph, start, end, path=[]):#empty list to save
values of path
        path = path + [start] # adding first node
        if start == end: # check if both are same
            return path
        if not graph.has_key(start):
#if start is not present in graph; returns true if key is in
dictionary , false otherwise
            return None
#if start and end are not same find the path
        for node in graph[start]: #for particular node 'a'
traverse the neighbours
            if node not in path:
                newpath = find_path(graph, node, end, path)
#recursive function; backtrack finds the vertices which are
not present in path yet
                if newpath: return newpath
        return None
print("Path between nodes",find_path(graph, 'e', 'd'))
```

FINDING PATHS BETWEEN TWO NODES/ VERTEX

The second 'if' statement is necessary only in case there are nodes that are listed as end points for arcs but that don't have outgoing arcs themselves, and aren't listed in the graph at all.

Such nodes could also be contained in the graph, with an empty list of outgoing arcs, but sometimes it is more convenient not to require this.

Note that while the user calls find_graph() with three arguments, it calls itself with a fourth argument: the path that has already been traversed. The default value for this argument is the empty list, '[]', meaning no nodes have been traversed yet.

This argument is used to avoid cycles (the first 'if' inside the 'for' loop). The 'path' argument is not modified: the assignment "path = path + [start]" creates a new list.

If we had written "path.append(start)" instead, we would have modified the variable 'path' in the caller, with disastrous results. (Using tuples, we could have been sure this would not happen, at the cost of having to write "path = path + (start,)" since "(start)" isn't a singleton tuple -- it is just a parenthesized expression.)

FINDING ALL PATHS BETWEEN TWO NODES/ VERTEX

It is simple to change this function to return a list of **all paths (without cycles)** instead of the first path it finds:

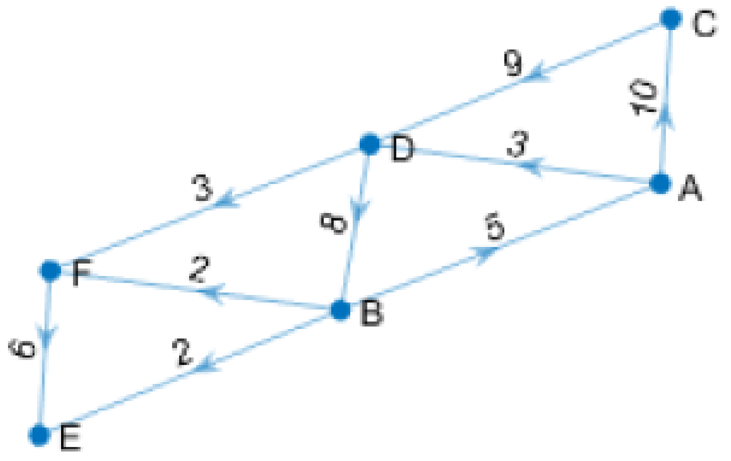
Example 5:

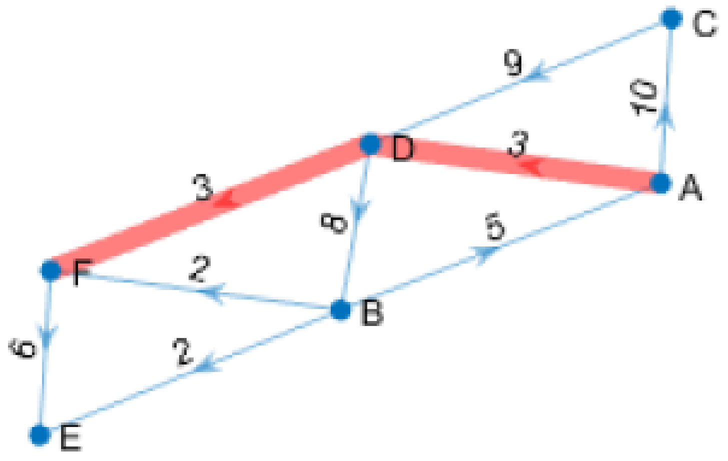
```
#Finding all paths between two nodes
def find_all_paths(graph, start, end, path=[]):
        path = path + [start]
        if start == end:
            return [path]
        if not graph.has_key(start):
            return []
        paths = []
        for node in graph[start]:
            if node not in path:
                newpaths = find_all_paths(graph, node, end, path)
                for newpath in newpaths:
                    paths.append(newpath)
        return paths

print("All Paths between nodes",find_all_paths(graph, 'a', 'b'))
```

TASK5: Generate the output of above code & Use the same code to find the path between ‘d’ and ‘b’ in the same graph.

SHORTEST PATHS BETWEEN TWO NODES

- The shortest path is a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. Consider the following graph:





SHORTEST PATHS BETWEEN TWO NODES/ VERTEX

Another variant finds the **shortest path:**

Example 6:

```
#Finding shortest path between two nodes
def find_shortest_path(graph, start, end, path=[]):
        path = path + [start]
        if start == end:
            return path
        if not graph.has_key(start):
            return None
        shortest = None
        for node in graph[start]:#traverse through graph and find neighbour
            if node not in path: # if neighbor node is not in path
                newpath = find_shortest_path(graph, node, end, path)# find shortest path between adjacent
nodes and return the result
                if newpath: # if you find new path
                    if not shortest or len(newpath) < len(shortest): # compare the length of the acquired path
with the shortest; if shortest path is found assign acquired path to shortest variable
                        shortest = newpath
        return shortest
print("Shortest Path between nodes",find_shortest_path(graph, 'a', 'b'))
```

TASK 6: Generate the output of above code & Use the same code to find the path between 'd' and 'b' in the same graph.

ADDING EDGES

We can create user defined function to add nodes to already created graphs.

Example 7:

```
print("Actual graph",graph)
#Adding an edge
def add_edge(graph,edge):
    edge=set(edge)# if you don’t want duplicates in list than you
will use set
    (n1,n2)=tuple(edge)# same as list; can’t be changed
    if n1 in graph:
        graph[n1]=n2
        #graph[n1].append(n2)
    return graph
print("add an edge:", add_edge(graph,{"a",“g"}))
```

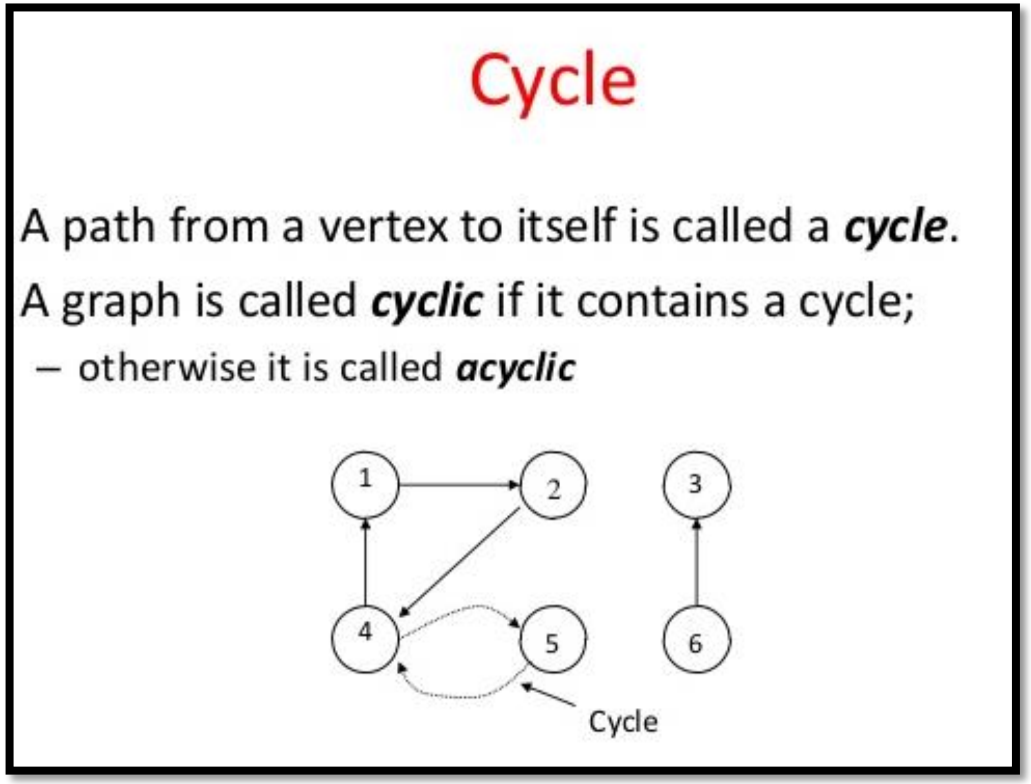
TASK 7: Generate the output of above code & Use the same code to add node to ‘d’ in ‘c’

CYCLES IN GRAPHS

Cycles are basically the loops present for a given node. For finding out if loops exist in a given graph function given below can be used.

Example 8:

```
graph = { "a" : ["a","c"],
        "b" : ["c", "e"],
        "c" : ["a", "b", "d", "e"],
        "d" : ["c"],
        "e" : ["c", "b"],
        "f" : []
      }
graph1 = { "a" : ["c"],
        "c" : []
      }
def find_cycle_single_node(graph, start):
   for node in graph[start]:
      if node==start:
         return "cycle exist"
   return "cycle does not exist"
print(find_cycle_single_node(graph,"a"))
print(find_cycle_single_node(graph1,"a"))
```



Cycle

A path from a vertex to itself is called a ***cycle***.

A graph is called ***cyclic*** if it contains a cycle;

– otherwise it is called ***acyclic***

1
2
3
4
5
6
Cycle

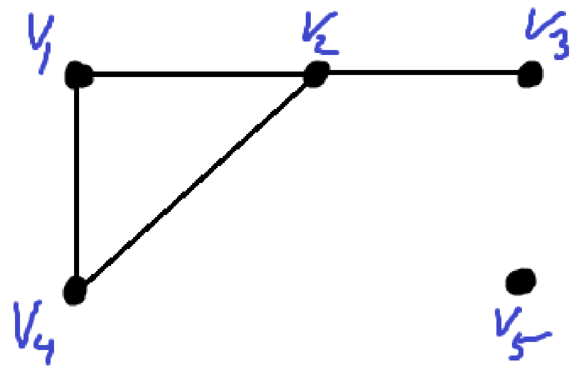
TASK 8: Generate the output of above code

DEGREE OF VERTEX

The degree of a vertex v in a graph is the number of edges connecting it, with loops counted twice. The degree of a vertex v is denoted deg(v). The maximum degree of a graph G, denoted by Δ(G), and the minimum degree of a graph, denoted by δ(G), are the maximum and minimum degree of its vertices. In the graph on the right side, the maximum degree is 5 at vertex c and the minimum degree is 0, i.e. the isolated vertex f.
If all the degrees in a graph are the same, the graph is a regular graph.

Example 9:

```
def find_degree(graph,node):
    degree=0
    t=[]
    for neighbour in graph[node]: # traverse through the neighbors
        t.append(neighbour) # append the neighbors which eventually will let us find the degree
        degree=degree+1
    return degree
Degree=find_degree(graph,"c")
print("degree of the vertex:",Degree)
```



$deg(v_1) = 2$

$deg(v_2) = 3$

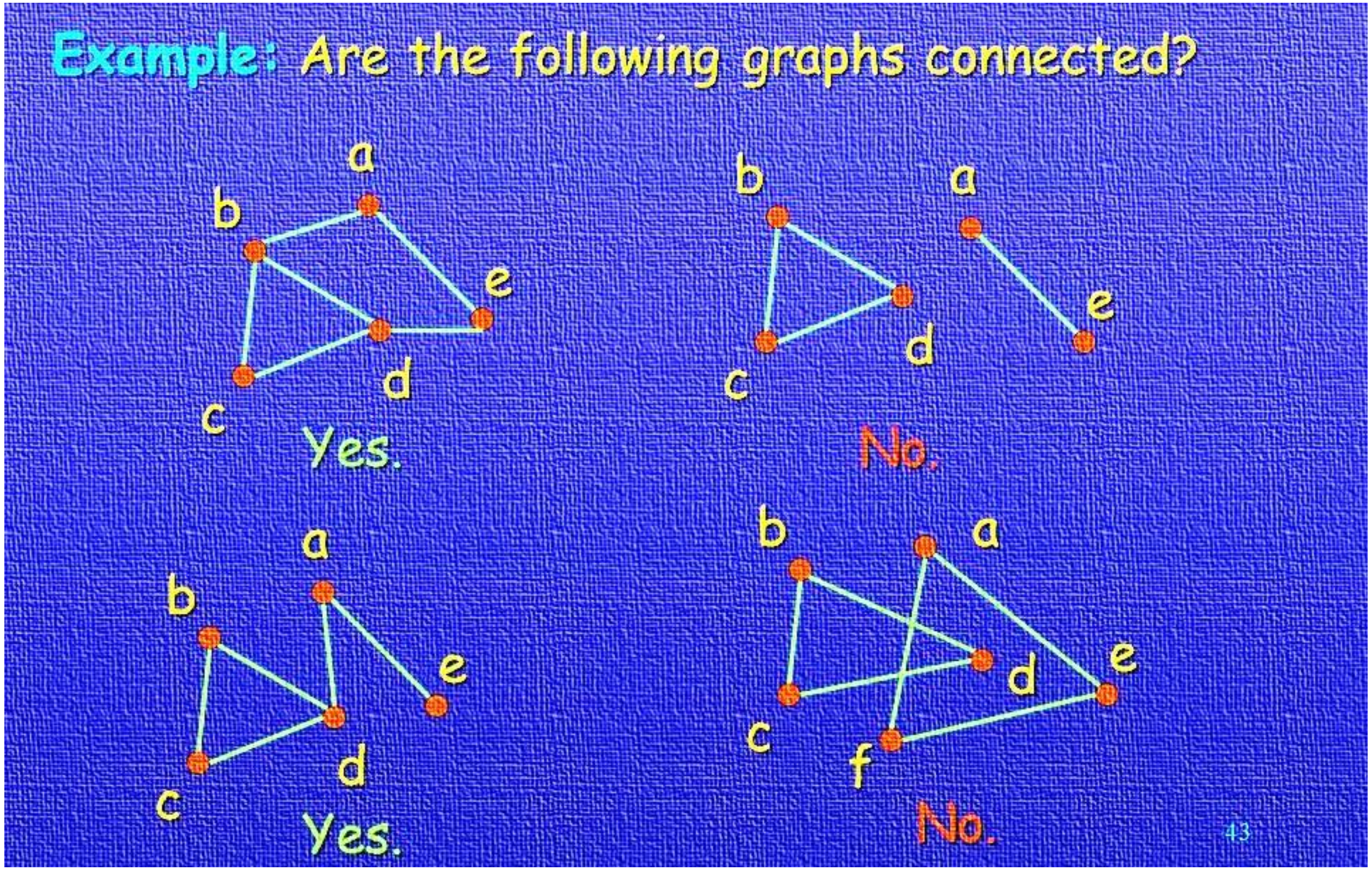
$deg(v_4) = 2$

$deg(v_3) = 1$

$deg(v_5) = 0$

TASK 9: Generate the output of above code

CONNECTED GRAPHS


Example: Are the following graphs connected?

CONNECTED GRAPHS

Example 10:

```
def graph_connected(graph,seen_node=None,start=None):
    if seen_node==None:
        seen_node=set()
        nodes=list(graph.keys())#list of all the graph keys
        if not start: #start is not given
         start=nodes[0] #vertex at the 0th index will be start
         seen_node.add(start)
         if len(seen_node)<len(nodes):
             for othernodes in graph[start]:# for the adjacent nodes of start
                 if othernodes not in seen_node: #if adjacent nodes are not present in
seen_node it will recursively call itself
                     if graph_connected(graph,seen_node,othernodes):
                         return True
         else:
             return True
             return False
conn=graph_connected(graph,seen_node=None,start=None)
if conn:
    print("The graph is connected")
else:
    print("The graph is not connected")
```

A graph is said to be connected if every pair of vertices in the graph is connected. The example graph on the right side is a connected graph. It possible to determine with a simple algorithm whether a graph is connected:

- Choose an arbitrary node x of the graph G as the starting point
- Determine the set A of all the nodes which can be reached from x.

If A is equal to the set of nodes of G, the graph is connected; otherwise it is disconnected.
We implement a method to check if a graph is a connected graph.

TASK 10: Generate the output of above code

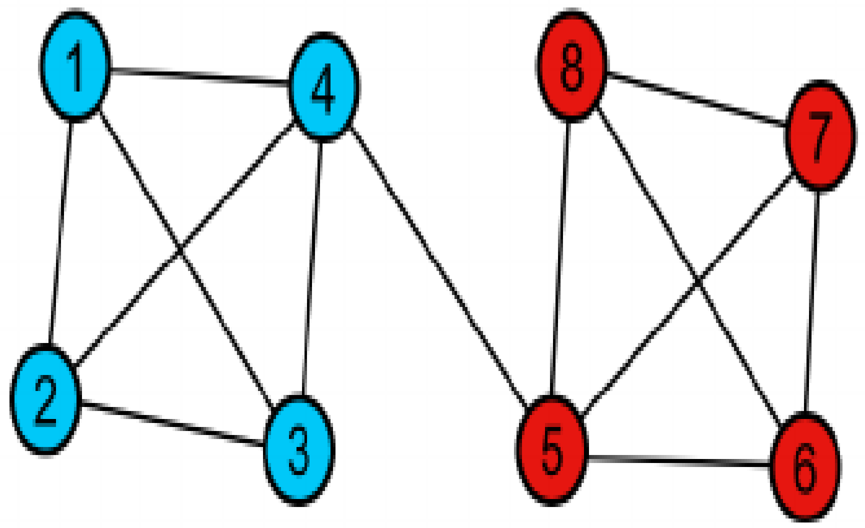
EXERCISE

Question 1: Define the following terms:

- Regular Graph
- Null Graph
- Trivial Graph
- Simple Graph
- Connected Graph
- Disconnected Graph
- Complete Graph
- Cyclic Graph
- Degree of vertex
- Loop
- Parallel Edges

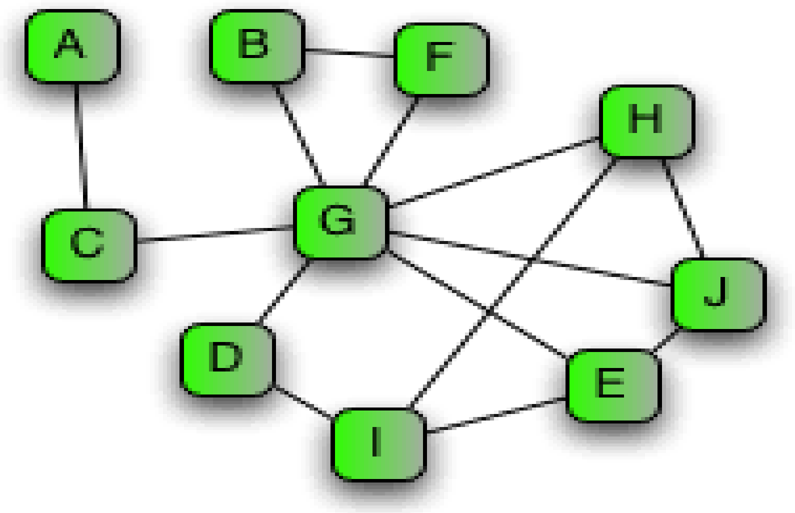
Question 2: Consider the following graph:

a) Find isolated nodes.
b) Find path between two vertex/ node 1 and 7.
c) Find all paths in graphs.
d) Find shortest path between nodes 1 and 7.
e) Determine cycles in graphs.
f) Add an edge named 9.
g) Find degree of vertex 4.
h) Find if the graph is connected.



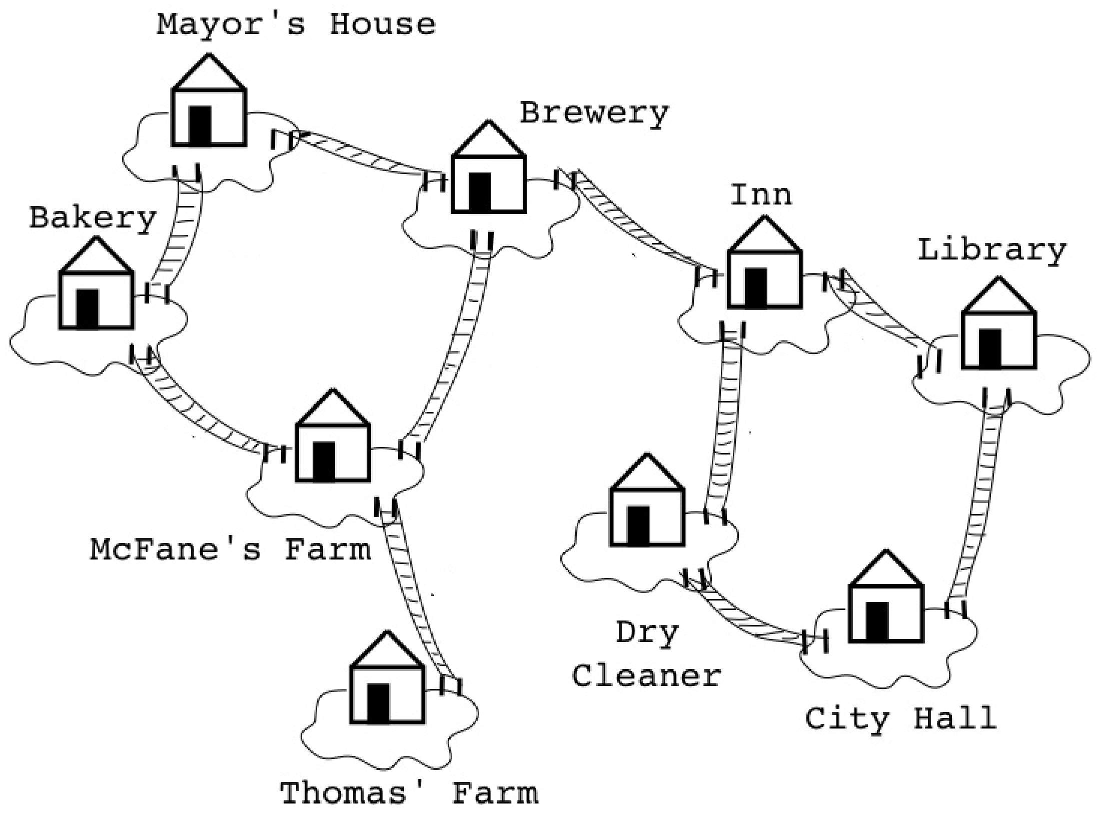
Question 2: Consider the following graph:

a) Find isolated nodes.
b) Find path between two vertex/node B and A.
c) Find all paths in graphs.
d) Find shortest path between nodes B and A.
e) Determine cycles in graphs.
f) Add an edge named K.
g) Find degree of vertex G.
h) Find if the graph is connected.



Question 3: Consider the following graph:

a) Find isolated nodes.
b) Find path between two vertex/node Thomas' Farm and Library.
c) Find all paths in graph.
d) Finding shortest path between nodes Thomas' Farm and Library.
e) Determine cycles in graphs.
f) Add an edge named John's House.
g) Find degree of vertex Bakery.
h) Find if the graph is connected.


Mayor's House
Brewery
Inn
Library
Bakery
McFane's Farm
Dry Cleaner
City Hall
Thomas' Farm