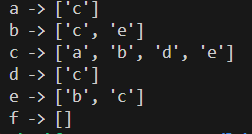
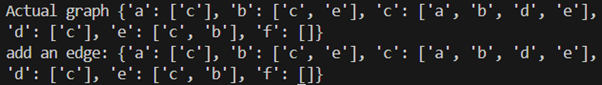
Name : Abdul Rehman Section :B

Course : AI&ES Roll No : CT-22052

---------------------------------------------------------------------------------------------------------------------  
 **Example 1:**

  
  
**Example 2:** **Example 3:** **Example 4:** **Example 5:** **Example 6:** **Example 7:**

**Example 8:** **Example 9:** **Example 10:** **Exercise:  
  
Question 1:  
Regular Graph**: A graph where each vertex has the same degree

**Null Graph**: A graph with no edges

**Trivial Graph**: A graph with only one vertex and no edges.

**Simple Graph**: A graph that has no loops and no parallel edges

**Connected Graph**: A graph where there is a path between every pair of vertices

**Disconnected Graph**: A graph where at least one pair of vertices is not connected by a path.

**Complete Graph**: A graph in which every pair of distinct vertices is connected by exactly one edge.

**Cyclic Graph**: A graph that contains at least one cycle, meaning a path that starts and ends at the same vertex without repeating any edge.

**Degree of a Vertex**: The number of edges connected to a particular vertex.

**Loop**: An edge that connects a vertex to itself.

**Parallel Edges**: Multiple edges that connect the same pair of vertices.  
  
**Question 2:**class Graph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_edge(self, u, v):

        if u not in self.graph:

            self.graph[u] = []

        if v not in self.graph:

            self.graph[v] = []

        self.graph[u].append(v)

        self.graph[v].append(u)

    def find\_isolated\_nodes(self):

        return [node for node in self.graph if not self.graph[node]]

    def find\_paths(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return [path]

        if start not in self.graph:

            return []

        paths = []

        for node in self.graph[start]:

            if node not in path:

                new\_paths = self.find\_paths(node, end, path)

                for p in new\_paths:

                    paths.append(p)

        return paths

    def find\_all\_paths(self):

        all\_paths = []

        nodes = list(self.graph.keys())

        for i in range(len(nodes)):

            for j in range(i + 1, len(nodes)):

                paths = self.find\_paths(nodes[i], nodes[j])

                all\_paths.extend(paths)

        return all\_paths

    def find\_shortest\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.graph:

            return None

        shortest = None

        for node in self.graph[start]:

            if node not in path:

                new\_path = self.find\_shortest\_path(node, end, path)

                if new\_path:

                    if not shortest or len(new\_path) < len(shortest):

                        shortest = new\_path

        return shortest

    def has\_cycle\_util(self, v, visited, parent):

        visited[v] = True

        for neighbor in self.graph[v]:

            if not visited.get(neighbor, False):

                if self.has\_cycle\_util(neighbor, visited, v):

                    return True

            elif parent != neighbor:

                return True

        return False

    def has\_cycle(self):

        visited = {}

        for node in self.graph:

            if not visited.get(node, False):

                if self.has\_cycle\_util(node, visited, -1):

                    return True

        return False

    def add\_vertex(self, v):

        if v not in self.graph:

            self.graph[v] = []

    def find\_degree(self, v):

        return len(self.graph.get(v, []))

    def is\_connected(self):

        visited = set()

        nodes = list(self.graph.keys())

        if not nodes:

            return True

        self.dfs(nodes[0], visited)

        return len(visited) == len(nodes)

    def dfs(self, node, visited):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                self.dfs(neighbor, visited)

g = Graph()

g.add\_edge(1, 4)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(4, 2)

g.add\_edge(4, 3)

g.add\_edge(4, 5)

g.add\_edge(5, 8)

g.add\_edge(5, 7)

g.add\_edge(5, 6)

g.add\_edge(7, 8)

g.add\_vertex(9)

print("isolated nodes:", g.find\_isolated\_nodes())

print("paths between 1 and 7:", g.find\_paths(1, 7))

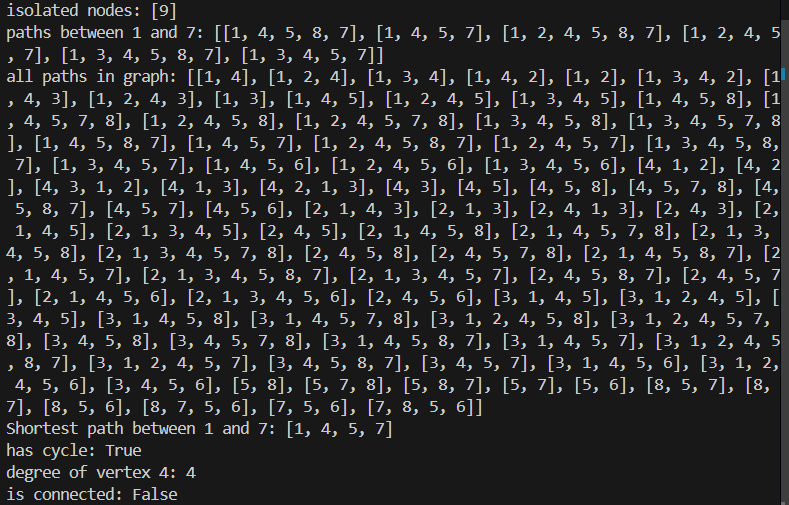
print("all paths in graph:", g.find\_all\_paths())

print("Shortest path between 1 and 7:", g.find\_shortest\_path(1, 7))

print("has cycle:", g.has\_cycle())

print("degree of vertex 4:", g.find\_degree(4))

print("is connected:", g.is\_connected())

****

**Question 2:**class Graph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_edge(self, u, v):

        if u not in self.graph:

            self.graph[u] = []

        if v not in self.graph:

            self.graph[v] = []

        self.graph[u].append(v)

        self.graph[v].append(u)

    def find\_isolated\_nodes(self):

        return [node for node in self.graph if not self.graph[node]]

    def find\_paths(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return [path]

        if start not in self.graph:

            return []

        paths = []

        for node in self.graph[start]:

            if node not in path:

                new\_paths = self.find\_paths(node, end, path)

                for p in new\_paths:

                    paths.append(p)

        return paths

    def find\_all\_paths(self):

        all\_paths = []

        nodes = list(self.graph.keys())

        for i in range(len(nodes)):

            for j in range(i + 1, len(nodes)):

                paths = self.find\_paths(nodes[i], nodes[j])

                all\_paths.extend(paths)

        return all\_paths

    def find\_shortest\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.graph:

            return None

        shortest = None

        for node in self.graph[start]:

            if node not in path:

                new\_path = self.find\_shortest\_path(node, end, path)

                if new\_path:

                    if not shortest or len(new\_path) < len(shortest):

                        shortest = new\_path

        return shortest

    def has\_cycle\_util(self, v, visited, parent):

        visited[v] = True

        for neighbor in self.graph[v]:

            if not visited.get(neighbor, False):

                if self.has\_cycle\_util(neighbor, visited, v):

                    return True

            elif parent != neighbor:

                return True

        return False

    def has\_cycle(self):

        visited = {}

        for node in self.graph:

            if not visited.get(node, False):

                if self.has\_cycle\_util(node, visited, -1):

                    return True

        return False

    def add\_vertex(self, v):

        if v not in self.graph:

            self.graph[v] = []

    def find\_degree(self, v):

        return len(self.graph.get(v, []))

    def is\_connected(self):

        visited = set()

        nodes = list(self.graph.keys())

        if not nodes:

            return True

        self.dfs(nodes[0], visited)

        return len(visited) == len(nodes)

    def dfs(self, node, visited):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                self.dfs(neighbor, visited)

g = Graph()

g.add\_edge('A', 'C')

g.add\_edge('A', 'B')

g.add\_edge('B', 'G')

g.add\_edge('G', 'F')

g.add\_edge('G', 'H')

g.add\_edge('G', 'I')

g.add\_edge('G', 'D')

g.add\_edge('H', 'J')

g.add\_edge('H', 'E')

g.add\_edge('I', 'D')

g.add\_edge('I', 'E')

g.add\_vertex('K')

print("iisolated nodes:", g.find\_isolated\_nodes())

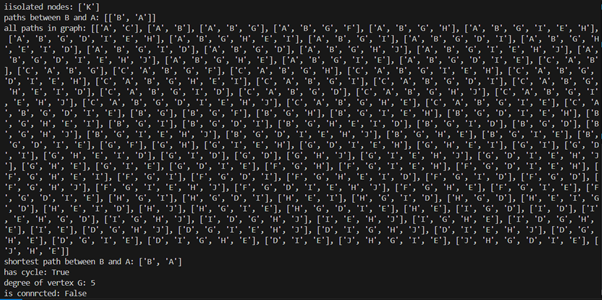
print("paths between B and A:", g.find\_paths('B', 'A'))

print("all paths in graph:", g.find\_all\_paths())

print("shortest path between B and A:", g.find\_shortest\_path('B', 'A'))

print("has cycle:", g.has\_cycle())

print("degree of vertex G:", g.find\_degree('G'))

print("is connrcted:", g.is\_connected())

**Question 3:**class Graph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_edge(self, u, v):

        if u not in self.graph:

            self.graph[u] = []

        if v not in self.graph:

            self.graph[v] = []

        self.graph[u].append(v)

        self.graph[v].append(u)

    def find\_isolated\_nodes(self):

        return [node for node in self.graph if not self.graph[node]]

    def find\_paths(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return [path]

        if start not in self.graph:

            return []

        paths = []

        for node in self.graph[start]:

            if node not in path:

                new\_paths = self.find\_paths(node, end, path)

                for p in new\_paths:

                    paths.append(p)

        return paths

    def find\_all\_paths(self):

        all\_paths = []

        nodes = list(self.graph.keys())

        for i in range(len(nodes)):

            for j in range(i + 1, len(nodes)):

                paths = self.find\_paths(nodes[i], nodes[j])

                all\_paths.extend(paths)

        return all\_paths

    def find\_shortest\_path(self, start, end, path=[]):

        path = path + [start]

        if start == end:

            return path

        if start not in self.graph:

            return None

        shortest = None

        for node in self.graph[start]:

            if node not in path:

                new\_path = self.find\_shortest\_path(node, end, path)

                if new\_path:

                    if not shortest or len(new\_path) < len(shortest):

                        shortest = new\_path

        return shortest

    def has\_cycle\_util(self, v, visited, parent):

        visited[v] = True

        for neighbor in self.graph[v]:

            if not visited.get(neighbor, False):

                if self.has\_cycle\_util(neighbor, visited, v):

                    return True

            elif parent != neighbor:

                return True

        return False

    def has\_cycle(self):

        visited = {}

        for node in self.graph:

            if not visited.get(node, False):

                if self.has\_cycle\_util(node, visited, -1):

                    return True

        return False

    def add\_vertex(self, v):

        if v not in self.graph:

            self.graph[v] = []

    def find\_degree(self, v):

        return len(self.graph.get(v, []))

    def is\_connected(self):

        visited = set()

        nodes = list(self.graph.keys())

        if not nodes:

            return True

        self.dfs(nodes[0], visited)

        return len(visited) == len(nodes)

    def dfs(self, node, visited):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                self.dfs(neighbor, visited)

g = Graph()

g.add\_edge("Thomas Farm", "McFanes Farm")

g.add\_edge("McFanes Farm", "Bakery")

g.add\_edge("Bakery", "Mayors House")

g.add\_edge("Mayors House", "Brewery")

g.add\_edge("Brewery", "Inn")

g.add\_edge("Inn", "Library")

g.add\_edge("Library", "City Hall")

g.add\_edge("City Hall", "Dry Cleaner")

g.add\_edge("Dry Cleaner", "Inn")

g.add\_vertex("John's House")

print("isolated Nodes:", g.find\_isolated\_nodes())

print("paths between Thomas Farm and Library:", g.find\_paths("Thomas Farm", "Library"))

print("all paths in graph:", g.find\_all\_paths())

print("shortest path between Thomas Farm and Library:", g.find\_shortest\_path("Thomas Farm", "Library"))

print("has Cycle:", g.has\_cycle())

print("degree of vertex Bakery:", g.find\_degree("Bakery"))

print("is Connected:", g.is\_connected())

