Lab-11: 19CSE212 Data Structures and Algorithms

Graph Representation using Linked List and Adjacency Matrix and implementation of the Graph ADT

Dt. 10-05-2024

1. Graph Representation

(a) Representation of Graph Using Linked List

Python Code Template

```
class Node:
  def __init__(self, vertex = None):
    self. vertex = vertex
    self. next = None

#implementation of linked list for the graph
class LinkedList:
  def __init__(self):
    self. head = None

  #add a vertex in the linked list
  def insert(self, vertex):
    node = Node(vertex)    #create a node
    if self. head == None:   #if list is empty insert a vertex(node) at the start
      self. head = node
    else:
      temp = self. head
      #iterate through the list till the last node is found
      while temp. next:
        temp = temp. next
```

```
    temp. next = node       #adding a new node

  #traverse through a linked list
  def traverse(self):
    temp = self. head
    while temp:
      print(temp. vertex, end='-->')
      temp = temp. next

#representation of graph using linked lists
class Graph:
  def __init__(self, no_vertices, directed = True):
    self. no_vertices = no_vertices
    self.vertices_list = [LinkedList() for i in range(0, no_vertices)]    #create a list to represent the graph
    self. directed = directed

  #insert an edge in the graph
  def insert_edge(self, edge):
    vertex1, vertex2 = edge
    if (vertex1 >=0 and vertex1 < self.no_vertices) and (vertex2 >=0 and vertex2 < self.no_vertices):
      #add an edge from vertex1 to vertex2
      self.vertices_list[vertex1].insert(vertex2)
      if not self.directed: #if the graph is undirected, add an edge from vertex2 to vertex1 as well
        self.vertices_list[vertex2].insert(vertex1)
    else:
      raise ValueError("Invalid vertices")

  #display the graph
  def display_graph(self):
    for i in range(0, self.no_vertices):
      print(i, end="\t")
```

```
        self.vertices_list[i].traverse()
        print("None")
```

Sample Inputs:

```
#Creation of undirected Graph
n = 5   #number of vertices

graph = Graph(n, directed=False)    #create an undirected graph

#insert edges
graph.insert_edge((0, 3))
graph.insert_edge((1, 3))
graph.insert_edge((1, 2))
graph.insert_edge((2, 4))
graph.insert_edge((3, 4))

#display graph
graph.display_graph()
```

Output

```
0    3-->None
1    3-->2-->None
2    1-->4-->None
3    0-->1-->4-->None
4    2-->3-->None
```

```
#Creation of directed Graph

n = 5 #number of vertices

graph = Graph(n, directed=True)     #create a directed graph
```

```
#insert edges
graph.insert_edge((0, 3))
graph.insert_edge((2, 1))
graph.insert_edge((2, 4))
graph.insert_edge((3, 1))
graph.insert_edge((3, 4))
graph.insert_edge((4, 2))

#display graph
graph.display_graph()
```

Output

```
0   3-->None
1   None
2   1-->4-->None
3   1-->4-->None
4   2 - ->None
```

(b)Representation of Graph Using Adjacency Matrix

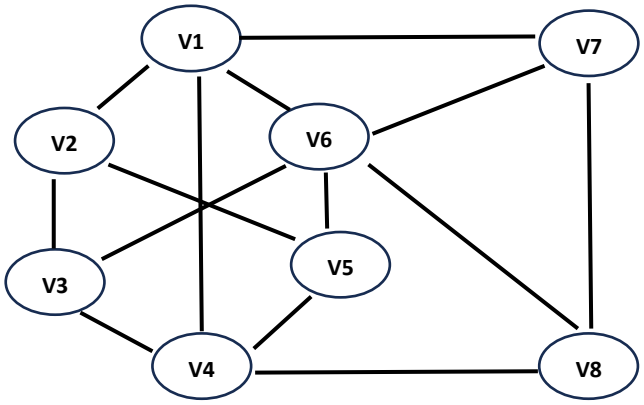
Python Code Template

```
class Graph(object):
    # Initialize the matrix
    def __init__(self, size):
        self.adjMatrix = []
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])
        self.size = size
    # Add edges
    def add_edge(self, v1, v2):
        if v1 == v2:
```

```
            print("Same vertex %d and %d" % (v1, v2))
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1
    # Remove edges
    def remove_edge(self, v1, v2):
        if self.adjMatrix[v1][v2] == 0:
            print("No edge between %d and %d" % (v1, v2))
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0
    def __len__(self):
        return self.size
    # Print the matrix
    def print_matrix(self):
        for row in self.adjMatrix:
            for val in row:
                print('{:4}'.format(val)),
            print
def main():
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.print_matrix()
if __name__ == '__main__':
    main()
```

Assignment (based on Graph Representation)

1. **Write the Python code to represent the following unweighted undirected simple graph using an adjacency list and an adjacency matrix.**



2. Implementation of ADTs for Graph Data Structure

List of Methods

incidentEdges(v): returns the list of edges incident on the vertex v.
endVertices(e): returns the pair of vertices (v, w) that forms the edge e.
isDirected(e): returns 'Yes', if e is directed, else returns 'No'
origin(e): returns the origin vertex v of the edge e, where e is a directed edge and e = v -> w
destination(e): returns the destination vertex w of the edge e, where e = v -> w
opposite(v, e): returns the vertex w, which is opposite to v in the edge e.
areAdjacent(v, w): returns 'Yes', if there exists an edge between (v, w), else, returns 'False'.
removeVertex(v): removes the vertex v from the graph and returns the updated vertex list.
removeEdge(e): removes the edge e from the graph and returns the updated edge list.
numVertices(): returns the number of vertices in the graph.
numEdges(): returns the number of edges in the graph.
vertices(): returns the list of vertices in the graph.
edges(): returns the list of edges in the graph.

Python Code Template:

```
class Graph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}

    def add_vertex(self, v):   // Insert vertex
        if v not in self.vertices:
```

```
            self.vertices[v] = []

    def add_edge(self, e, v1, v2, directed=False):   // Insert Edge
        self.edges[e] = (v1, v2)
        self.vertices[v1].append(e)
        self.vertices[v2].append(e)
        if not directed:
            self.edges[(v2, v1)] = e
            self.vertices[v1].append((v2, v1))

    def incident_edges(self, v):
        return self.vertices[v]

    def end_vertices(self, e):
        return self.edges[e]

    def is_directed(self, e):
        return isinstance(e, tuple)

    def origin(self, e):
        if self.is_directed(e):
            return e[0]
        else:
            return None

    def destination(self, e):
        if self.is_directed(e):
            return e[1]
        else:
            return None

    def opposite(self, v, e):
        v1, v2 = self.end_vertices(e)
        if v == v1:
            return v2
        elif v == v2:
            return v1
        else:
            return None

    def are_adjacent(self, v1, v2):
        for e in self.vertices[v1]:
            if self.destination(e) == v2:
                return True
        return False

def remove_vertex(self, v):
        incident_edges = self.incident_edges(v)
```

```
        for e in incident_edges:
            self.remove_edge(e)
        del self.vertices[v]

    def remove_edge(self, e):
        v1, v2 = self.end_vertices(e)
        self.vertices[v1].remove(e)
        self.vertices[v2].remove(e)
        del self.edges[e]

    def num_vertices(self):
        return len(self.vertices)

    def num_edges(self):
        return len(self.edges)

    def vertices(self):
        return set(self.vertices.keys())

    def edges(self):
        return set(self.edges.keys())

# Sample Inputs
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_vertex('C')
g.add_vertex('D')

g.add_edge('e1', 'A', 'B')
g.add_edge('e2', 'A', 'C')
g.add_edge('e3', 'B', 'C')
g.add_edge('e4', 'B', 'D')

print("Incident Edges to vertex 'A':", g.incident_edges('A'))
print("End Vertices of edge 'e1':", g.end_vertices('e1'))
print("Is 'e1' directed?", g.is_directed('e1'))
print("Origin of edge 'e1':", g.origin('e1'))
print("Destination of edge 'e1':", g.destination('e1'))
print("Opposite of vertex 'A' in edge 'e1':", g.opposite('A', 'e1'))
print("Are 'A' and 'B' adjacent?", g.are_adjacent('A', 'B'))

print("Number of Vertices:", g.num_vertices())
print("Number of Edges:", g.num_edges())
print("Vertices:", g.vertices())
print("Edges:", g.edges())

g.remove_edge('e1')
```

```
print("After removing 'e1' edge, Edges:", g.edges())

g.remove_vertex('C')
print("After removing 'C' vertex, Vertices:", g.vertices())
```

Assignment (Based on the implementation of Graph ADT)

Scenario 1:

Suppose you are developing a social networking platform where users can connect with each other by forming friendships. You decide to represent the network of users as a graph, where each user is a vertex, and each friendship connection between users is an edge. You have implemented the necessary Abstract Data Types (ADTs) for the graph, including methods to add vertices and edges, remove vertices and edges, count the number of vertices and edges, and retrieve the set of vertices and edges.

Requirements:

1. **Initialization:**
   You've just initialized the graph to represent the social network of your platform. How many vertices and edges does the graph currently have? Also, can you list all the vertices and edges present in the network?

2. **Adding Connections:**
   A new user, Karthik, joins the platform and becomes friends with Srinivas and Surya. Describe the steps you would take to represent these new connections in the graph. How does this affect the count of vertices and edges in the graph?

3. **Removing a Connection:**
   Unfortunately, Karthik and Surya disagreed and decided to unfriend each other. How would you remove the friendship connection between Karthik and Surya from the graph? What is the new count of edges in the graph after this operation?

4. **Analyzing Network Size:**
   As the platform grows, you're interested in analyzing the size of the social network. How would you determine the total number of users (vertices) and friendships (edges) currently present in the network?

5. **Cleanup:**
   It's important to keep the network tidy by removing inactive users. Suppose Karthik decides to leave the platform. Describe the steps you would take to remove Karthik from the graph, along with all her friendship connections. What is the new count of vertices and edges in the graph after this cleanup operation?

Based on the above requirements, write a Python code to implement the social networking platform.

Sample Inputs and Outputs

1. **Initialization:**
   Graph Initialized:
   Number of Vertices: 0
   Number of Edges: 0
   Vertices: {}
   Edges: {}

2. **Adding Connections:**
   Adding connections for Karthik:
   Karthik becomes friends with Srinivas
   Karthik becomes friends with Surya

   After adding connections:
   Number of Vertices: 3
   Number of Edges: 2
   Vertices: {'Karthik', 'Srinivas', 'Surya'}
   Edges: {('Karthik', 'Srinivas'), ('Karthik', 'Surya')}

3. **Removing Connection:**
   Removing the connection between Karthik and Surya:

   After removing the connection:
   Number of Vertices: 3
   Number of Edges: 1
   Vertices: {'Karthik', 'Srinivas', 'Surya'}
   Edges: {('Karthik', 'Srinivas')}

4. **Analyzing the Network size:**

   Total Number of Users (Vertices): 3
   Total Number of Friendships (Edges): 1

5. **Cleanup:**
   Cleaning up inactive user: Karthik

   After cleanup:
   Number of Vertices: 2
   Number of Edges: 0
   Vertices: {'Srinivas', 'Surya'}
   Edges: {}

Scenario 2:

You're working on a project to model a transportation network for a city using graphs. Each intersection in the city represents a vertex in the graph, and each road connecting two intersections represents an edge. You've implemented a graph using an adjacency matrix, where the value in each cell represents the distance between the intersections (vertices). The city planners want to perform various operations on this graph to optimize transportation routes.

Requirements

1. **Initialization:**
   You've initialized the transportation network graph for the city, representing intersections and roads as vertices and edges, respectively. Can you display the adjacency matrix representing this network? Also, how many intersections (vertices) and roads (edges) are currently in the network?

2. **Adding a New Road:**
   A new road connecting Intersection A and Intersection B is constructed. How would you update the adjacency matrix to represent this new road? What are the implications for the transportation network in terms of connectivity and distance?

3. **Removing a Road:**
   Due to road maintenance, the road connecting Intersection B and Intersection C needs to be temporarily closed. How would you update the adjacency matrix to reflect this change? How does this affect the transportation options for commuters traveling between Intersection B and Intersection C?

4. **Analyzing Connectivity:**
   The city council is interested in understanding the connectivity of the transportation network. How would you determine if there is a direct road connection between Intersection A and Intersection C? If not, what alternative routes could commuters take to travel between these intersections?

Based on the above requirements, write a Python code to implement the transportation network system.

Sample Inputs and Outputs

1. **Initialization**
   Initializing transportation network graph using weighted adjacency matrix:
   # The numbers in the weighted adjacency matrix indicate the edge weights.
   [[0, 10, 0, 0],
   [10, 0, 20, 0],
   [0, 20, 0, 15],
   [0, 0, 15, 0]]
   Number of Intersections (Vertices): 4
   Number of Roads (Edges): 4

2. **Adding a New Road:**
   Adding a new road between Intersection A and Intersection B:
   Updated Adjacency Matrix:
   [[0, 5, 0, 0],
   [5, 0, 20, 0],
   [0, 20, 0, 15],
   [0, 0, 15, 0]]

3. **Removing a Road:**
   Removing the road between Intersection B and Intersection C:
   Updated Adjacency Matrix:
   [[0, 5, 0, 0],
   [5, 0, 0, 0],
   [0, 0, 0, 15],
   [0, 0, 15, 0]]

4. **Analyzing Connectivity:**
   Analyzing connectivity between Intersection A and Intersection C:
   There is no direct road connection between Intersection A and Intersection C.
   Alternative routes:
   - Intersection A -> Intersection B -> Intersection D -> Intersection C
   - Intersection A -> Intersection D -> Intersection C