Graph Theory

Organizational information

Today we will solve the whole laboratory on small data, yours homework is to update the code to run on big data

We have a lot of things to do, please be careful and work quickly

The statement of the problem is at the address

http://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/lab1.html

The common requirements for the whole year are on teams

Practical work no.1
Scadentă la 7 Aprilie 2023 23:59

Start

A **graph** is a data structure that consists of vertices that are connected via edges. It can be implemented with an:

1. Adjacency list

For every vertex, its adjacent vertices are stored.

In the case of a weighted graph, the edge weights are stored along with the vertices.

2. Adjacency matrix

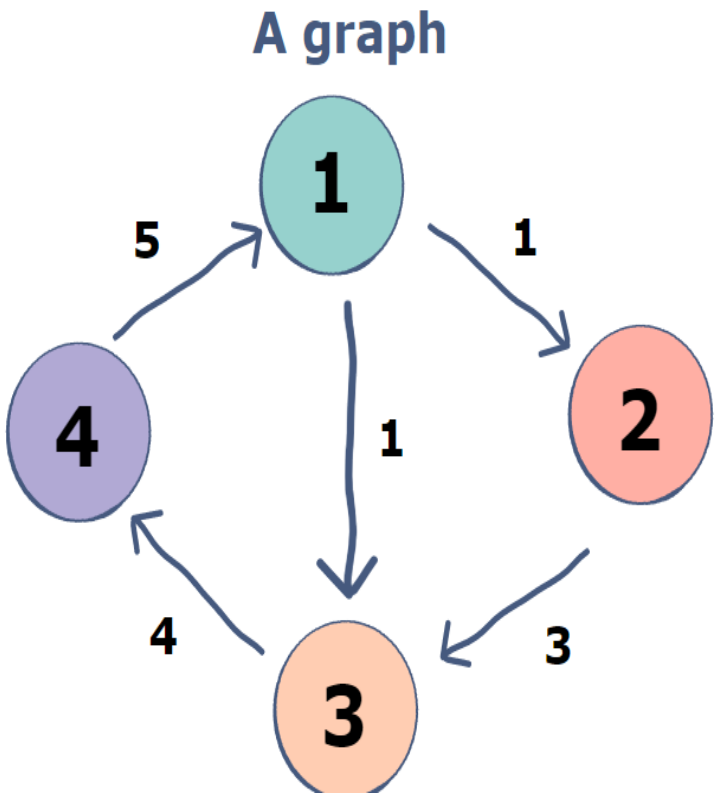
- The row and column indices represent the vertices:

matrix[*i*][*j*]=1 means that there is an edge from vertices *i* to *j*, and *matrix*[*i*][*j*]=0 denotes that there is no edge between *i* and *j*.

For a weighted graph,the edge weight is usually written in place of 1.

just to refresh the information

We will work with



Adjacency list	Adjacency matrix
{"1" : [[2, 1], [3,1]], "2" : [[3, 3]], "3" : [[4, 4]], "4" : [[1, 5]]}	[[0, 1, 1, 0], [0, 0, 3, 0], [0, 0 ,0 , 4], [5, 0, 0 ,0]]

a simple implementation without TAD is presented on the next slide

```
# Add a vertex to the dictionary
def add_vertex(v):
  global graph
  global vertices_no
  if v in graph:
    print("Vertex ", v, " already exists.")
  else:
    vertices_no = vertices_no + 1
    graph[v] = []
 # Add an edge between vertex v1 and v2 with edge weight e
def add_edge(v1, v2, e):
  global graph
  # Check if vertex v1 is a valid vertex
  if v1 not in graph:
    print("Vertex ", v1, " does not exist.")
  # Check if vertex v2 is a valid vertex
  elif v2 not in graph:
    print("Vertex ", v2, " does not exist.")
  else:
    # Since this code is not restricted to a directed or
    # an undirected graph, an edge between v1 v2 does not
    # imply that an edge exists between v2 and v1
    temp = [v2, e]
    graph[v1].append(temp)
 # Print the graph
def print_graph():
  global graph
  for vertex in graph:
    for edges in graph[vertex]:
      print(vertex, " -> ", edges[0], " edge weight: ", edges[1])
  # driver code
  graph = {}
  # stores the number of vertices in the graph
  vertices_no = 0
  add_vertex(1)
  add_vertex(2)
  add_vertex(3)
  add_vertex(4)
  # Add the edges between the vertices by specifying
  # the from and to vertex along with the edge weights.
  add_edge(1, 2, 1)
  add_edge(1, 3, 1)
  add_edge(2, 3, 3)
  add_edge(3, 4, 4)
  add_edge(4, 1, 5)
  print_graph()
  print(vertices_no)
  # Reminder: the second element of each list inside the
  dictionary
  # denotes the edge weight.
  print ("Internal representation: ", graph)
```

The self.graph

is likely a variable that is defined within a class, possibly the Graph class. Without additional context, it's difficult to provide a more specific explanation.

In general, self.graph could refer to a data structure that represents a graph. Depending on the implementation, this data structure could store information such as the vertices and edges of the graph, along with any associated attributes or properties of each vertex or edge.

For example, self.graph could be implemented as an adjacency list, where each vertex is a key in a dictionary, and its associated value is a list of adjacent vertices. Alternatively, self.graph could be implemented as an adjacency matrix, where the edges are represented as entries in a matrix.

In the context of a class, self.graph may be a member variable that is used throughout the class methods to manipulate or analyze the graph. For example, self.graph may be passed as a parameter to other methods that operate on the graph data structure.

What is abstract data type in graph theory?

implement with python

The answer

In graph theory, an abstract data type (ADT) is a mathematical model of a graph that defines a set of operations that can be performed on the graph. These operations include adding and removing vertices and edges, finding paths between vertices, and computing various properties of the graph, such as its connectivity and degree sequence.

In Python, an ADT for graphs can be implemented using a combination of classes and methods. One approach is to define a class for the graph itself, along with classes for vertices and edges, and methods for performing the various operations on the graph.

This implementation defines a Graph class with methods for adding and removing vertices and edges, finding paths between vertices, and checking whether the graph is connected. The Graph class contains a dictionary vertices that maps each vertex to a list of its adjacent vertices, and a list edges that stores the edges in the graph.

The implementation of answer

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

    def add_vertex(self, vertex):
        self.vertices[vertex] = []

    def add_edge(self, vertex1, vertex2):
        self.edges.append((vertex1, vertex2))
        self.vertices[vertex1].append(vertex2)
        self.vertices[vertex2].append(vertex1)

    def remove_vertex(self, vertex):
        for v in self.vertices[vertex]:
            self.vertices[v].remove(vertex)
            self.edges.remove((vertex, v))
        del self.vertices[vertex]

    def remove_edge(self, vertex1, vertex2):
        self.edges.remove((vertex1, vertex2))
        self.vertices[vertex1].remove(vertex2)
        self.vertices[vertex2].remove(vertex1)
```

```
    def find_path(self, start, end, path=None):
        if path is None:
            path = []
        path = path + [start]
        if start == end:
            return path
        if start not in self.vertices:
            return None
        for vertex in self.vertices[start]:
            if vertex not in path:
                extended_path = self.find_path(vertex, end, path)
                if extended_path:
                    return extended_path
        return None
```

```
    def is_connected(self):
        visited = set()
        vertices = list(self.vertices.keys())
        self._dfs(vertices[0], visited)
        return len(visited) == len(vertices)

    def _dfs(self, vertex, visited):
        visited.add(vertex)
        for v in self.vertices[vertex]:
            if v not in visited:
                self._dfs(v, visited)
```

Note that this is just one possible implementation of a graph ADT in Python. Depending on the specific requirements of a given problem, other implementations may be more appropriate or efficient.

Algorithm to remove vertex

```
def remove_vertex(self, vertex):
    for v in self.vertices[vertex]:
        self.vertices[v].remove(vertex)
        self.edges.remove((vertex, v))
    del self.vertices[vertex]
```

The method takes one parameter, vertex, which is the vertex to be removed from the graph.

The first line of the method starts a loop that iterates over all the adjacent vertices of the vertex being removed:

```
for v in self.vertices[vertex]:
```

For each adjacent vertex, the code removes the vertex being removed from its adjacency list:

```
self.vertices[v].remove(vertex)
```

This line is removing the vertex from the list of adjacent vertices of v. This ensures that the graph remains connected after the vertex is removed.

The next line removes the edge between the vertex being removed and the adjacent vertex:

```
self.edges.remove((vertex, v))
```

This line removes the tuple (vertex, v) from the list of edges of the graph. This ensures that the edge between the vertex being removed and the adjacent vertex is no longer part of the graph.

Finally, the last line of the method deletes the vertex being removed from the vertices dictionary:

```
del self.vertices[vertex]
```

This line removes the vertex from the vertices dictionary, effectively removing it from the graph.

In summary, the remove_vertex method removes a vertex from a graph, updating the adjacency lists of its adjacent vertices and removing any edges that were incident to the vertex being removed

OR

Based on this simple example we will go today

The statement of the problem we will solve today

Design and implement an abstract data type directed graph and a function (either a member function or an external one, as your choice) for reading a directed graph from a text file.

The vertices will be specified as integers from 0 to n-1, where n is the number of vertices.

Edges may be specified either by the two endpoints (that is, by the source and target), or by some abstract data type Edge_id (that data type may be a pointer or reference to the edge representation, but without exposing the implementation details of the graph).

Additionally, create a map that associates to an edge an integer value (for instance, a cost).

Required operations:

get the number of vertices;
parse (iterate) the set of vertices;

is the problem stated on the website http://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/lab1.html

How implement a graph

Adjacency list

The DirectedGraph class uses a dictionary to store the vertices and their respective edges.

- The __init__ method is the constructor method of a class in Python. It is called when an instance of the class is created.
- DirectedGraph class is being defined and it has an __init__ method that initializes an empty dictionary **self.graph**
- The **add_vertex** method can be used to add a vertex to the graph object. The add_vertex method adds a vertex to the dictionary, and the add_edge method adds an edge between two vertices
- The __str__ method prints the graph by iterating through the dictionary and its edges.

In the following we created an instance of the **DirectedGraph class** and added three vertices to the graph with the **add_vertex** method. If a vertex is not already present in the graph, the method creates a new key in the graph dictionary with an empty list as its value.

```
class DirectedGraph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, v):
        if v in self.graph:
            print("Vertex ", v, " already exists.")
        else:
            self.graph[v] = []

    def add_edge(self, v1, v2, e):
        if v1 not in self.graph:
            print("Vertex ", v1, " does not exist.")
        elif v2 not in self.graph:
            print("Vertex ", v2, " does not exist.")
        else:
            self.graph[v1].append((v2, e))

    def __str__(self):
        s = ''
        for vertex in self.graph:
            for edges in self.graph[vertex]:
                s += f"{vertex} -> {edges[0]} edge weight:
{edges[1]}\n"
        return s
```

```
def add_edge(self, v1, v2, e):
    if v1 not in self.graph:
        print("Vertex ", v1, " does not exist.")
    elif v2 not in self.graph:
        print("Vertex ", v2, " does not exist.")
    else:
        self.graph[v1].append((v2, e))
```

the line graph.add_edge(1, 2, 5) adds an edge from vertex 1 to vertex 2 with a weight of 5. The add_edge method checks to make sure that both v1 and v2 are valid vertices in the graph before adding the edge. If either vertex is not in the graph, an error message is printed.

After adding the edges, we print the graph using the __str__ method of the DirectedGraph class, which returns a string representation of the graph in the format vertex: (neighbor1, weight1), (neighbor2, weight2), ...

```
# Create an instance of the DirectedGraph class
graph = DirectedGraph()

# Add three vertices to the graph
graph.add_vertex(1)
graph.add_vertex(2)
graph.add_vertex(3)

# Add edges between the vertices
graph.add_edge(1, 2, 5)
graph.add_edge(1, 3, 2)
graph.add_edge(2, 3, 1)
```

we first create an empty DirectedGraph object called graph. Then we add three vertices to the graph using the add_vertex method. Next, we add edges between the vertices using the add_edge method. Now if you print the graph dictionary, you will see that it contains the vertices we just added:

```
print(graph.graph)
# Output: {'1': [], '2': [], '3': []}
```

```
# Print the graph to verify the edges were added correctly
print(graph)

# Output:
# 1: (2, 5), (3, 2)
# 2: (3, 1)
# 3:
```

The output shows that vertex 1 has edges to vertices 2 and 3 with weights 5 and 2, respectively. Vertex 2 has an edge to vertex 3 with weight 1. Vertex 3 has no outgoing edges.

```
def __str__(self):
    s = ''
    for vertex in self.graph:
        for edges in self.graph[vertex]:
            s += f"{vertex} -> {edges[0]} edge weight:
{edges[1]}\n"
    return s
```

The code def __str__(self): is defining a method for a class in Python, which is used to return a string representation of an instance of that class. In this case, the method is used to represent a graph object.

The method first initializes an empty string s, which will be used to build up the string representation of the graph. It then iterates over each vertex in the graph using a for loop, and for each vertex, it iterates over its edges using another for loop. For each edge, the method adds a string to s that indicates the vertex, the vertex it is connected to via the edge, and the weight of the edge.

Finally, the method returns the complete string s that represents the graph.

```
1 -> 2 edge weight: 1
1 -> 3 edge weight: 1
2 -> 3 edge weight: 3
3 -> 4 edge weight: 4
4 -> 1 edge weight: 5
```

Finally, we **call print(graph)** to print the string representation of the graph, which is generated by the __str__() method. The output is shown above.

Input.txt

Each line of the file represents a directed edge in the graph, in the format: v1 v2 e

Where
v1 is the starting vertex of the edge,
v2 is the ending vertex of the edge, and
e is the weight of the edge.

For example, if you want to create the directed graph with vertices 1, 2, 3, and 4, and edges (1, 2, 1), (1, 3, 1), (2, 3, 3), (3, 4, 4), and (4, 1, 5), then the contents of input.txt would be:

```
1 2 1
1 3 1
2 3 3
3 4 4
4 1 5
```

In this updated code, we define a new method read_graph(filename) which takes a filename as input and reads the directed graph from the file. The file should contain one directed edge per line in the format "v1 v2 e", where v1 and v2 are integers representing the vertices and e is a float representing the edge weight.

The read_graph() method opens the file, reads each line, and adds the vertices and edges to the graph using the add_vertex() and add_edge() methods. If a vertex already exists in the graph, it will not be added again.

In the driver code, we call the read_graph() method to read the graph from the input file "input.txt". Then we print the number of vertices, the vertices themselves, and the graph itself using the num_vertices(), vertices(), and __str__() methods, respectively.

```
class DirectedGraph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, v):
        if v in self.graph:
            print("Vertex ", v, " already exists.")
        else:
            self.graph[v] = []

    def add_edge(self, v1, v2, e):
        if v1 not in self.graph:
            print("Vertex ", v1, " does not exist.")
        elif v2 not in self.graph:
            print("Vertex ", v2, " does not exist.")
        else:
            self.graph[v1].append((v2, e))
```

```
def num_vertices(self):
        return len(self.graph)

    def vertices(self):
        return self.graph.keys()
```

The num_vertices() method returns the number of vertices in the graph, which is simply the length of the dictionary self.graph.
The vertices() method returns a set of all the vertices in the graph, which is just the keys of the dictionary self.graph.
In the driver code, we call these two methods and print their output. The output shows that the graph has four vertices and we iterate over the vertices to print them one by one.

```
def read_graph(self, filename):
    with open(filename, 'r') as f:
        for line in f:
            v1, v2, e = line.strip().split()
            v1 = int(v1)
            v2 = int(v2)
            e = float(e)
            if v1 not in self.graph:
                self.add_vertex(v1)
            if v2 not in self.graph:
                self.add_vertex(v2)
            self.add_edge(v1, v2, e)
```

The method **read_graph** reads a directed graph from a text file with the given filename. It assumes that the file contains one **directed edge per line**, with the starting vertex, ending vertex, and edge weight separated by spaces.

The method **opens** the file using a with statement, which automatically closes the file when the block is exited. It then reads each line of the file using a for **loop**, and processes it as follows:

It **strips** any whitespace from the beginning and end of the line using the strip method.
It **splits** the line into three parts using the **split** method, **which** returns a list of strings.
The **v1, v2, and e variables** are assigned to the first, second, and third elements of the list, respectively.
It **converts the v1 and v2 variables from strings to integers using the int** function, and the e variable from a string to a float using the float function.
It checks if the starting vertex v1 is already in the graph using an if statement. If it is not, it adds it to the graph using the **add_vertex** method.
It checks if the ending vertex v2 is already in the graph. If it is not, it adds it to the graph using the add_vertex method.
It adds the directed edge to the graph using the add_edge method, with the starting vertex v1, ending vertex v2, and edge weight e.
Overall, the read_graph method allows you to easily read a directed graph from a text file and add it to an instance of the DirectedGraph class.

The Edge_id

In graph theory, an edge is a connection between two vertices in a graph. The edge is usually represented as an unordered pair of vertices, indicating that the edge is bidirectional. For example, an edge between vertices u and v can be represented as (u, v) or (v, u).

The Edge_id is an additional identifier assigned to each edge in the graph. This identifier can be any unique value that distinguishes one edge from another. It is not required to represent an edge as an (u, v) pair, but instead, each edge can be assigned a unique identifier, and then stored in a separate data structure.

For example, suppose we have a graph with the following edges:

```
(1, 2)
(2, 3)
(1, 3)
```

We can assign each edge a unique identifier as follows:

```
edge_ids = {
    (1, 2): 1,
    (2, 3): 2,
    (1, 3): 3
}
```

In this case, we are using a dictionary to map each edge to a unique identifier. The Edge_id serves as an additional piece of information that can be associated with each edge in the graph. This information can be useful for certain graph algorithms or applications that require more information about each edge beyond its endpoint vertices.

However, it's worth noting that not all graph representations or algorithms require an Edge_id. In many cases, representing an edge as an unordered pair of vertices is sufficient for performing various graph operations.

- given two vertices, find out whether there is an edge from the first one to the second one, and retrieve the Edge_id if there is an edge (the latter is not required if an edge is represented simply as a pair of vertex identifiers);
- get the in degree and the out degree of a specified vertex;

has_edge, get_edge_id, in_degree, and out_degree:

```
def has_edge(self, v1, v2):
    for edges in self.graph[v1]:
        if edges[0] == v2:
            return True, edges[1]
    return False, None

def in_degree(self, v):
    degree = 0
    for vertex in self.graph:
        for edges in self.graph[vertex]:
            if edges[0] == v:
                degree += 1
    return degree

def out_degree(self, v):
    if v not in self.graph:
        print("Vertex ", v, " does not exist.")
        return None
    return len(self.graph[v])
```

The has_edge method takes two vertices v1 and v2 as input and returns True if there is an edge from v1 to v2 and False otherwise.

The get_edge_id method takes two vertices v1 and v2 as input and returns the index of the edge

The get_edge_id method takes two vertex identifiers as input and returns the ID of the edge from the first vertex to the second vertex, if such an edge exists. Note that in this implementation, an edge is represented simply as a pair of vertex identifiers, so the edge ID is the same as the ending vertex ID.

The in_degree method takes a vertex identifier as input and returns the in-degree of that vertex, i.e., the number of edges that point to that vertex.

The out_degree method takes a vertex identifier as input and returns the out-degree of that vertex, i.e., the number of edges that start from that vertex.

```
graph = DirectedGraph()
graph.read_graph("C:/Users/user/Documents/2022_2023/GrafTheories/Lab2/input.txt")
 print("Number of vertices:", graph.num_vertices())


print("Vertices:")
for v in graph.vertices():
    print(v)


print("Graph:")
print(graph)


# check if edge exists and get its weight
edge_exists, weight = graph.has_edge(1, 2)
if edge_exists:
    print("There is an edge from 1 to 2 with weight", weight)
else:
    print("There is no edge from 1 to 2")
```

```
# get the in-degree and out-degree of vertex 3
in_degree = graph.in_degree(3)
if in_degree is not None:
    print("The in-degree of vertex 3 is", in_degree)

out_degree = graph.out_degree(3)
if out_degree is not None:
    print("The out-degree of vertex 3 is", out_degree))
```

To get the endpoints of an edge specified by an Edge_id, we first need to modify the add_edge method in the DirectedGraph class to keep track of the edge ids:

Here's an updated version of the add_edge() method that tracks the edge ids:

```
class DirectedGraph:
    def __init__(self):
        self.graph = {}

        # Initialize an empty dictionary to store edge IDs
        self.edge_ids = {}

    def add_edge(self, source, dest, weight):
        # Generate a unique ID for the new edge
        edge_id = len(self.edge_ids)

        # Add the edge to the graph
        if source in self.graph:
            self.graph[source].append((dest, weight, edge_id))
        else:
            self.graph[source] = [(dest, weight, edge_id)]

        # Add the edge ID to the dictionary
        self.edge_ids[(source, dest)] = edge_id
```

Once the edge ids are being tracked, we can define a new method get_edge_endpoints() that takes an edge_id as input and returns the source and destination vertices of the corresponding edge:

Once the edge ids are being tracked, we can define a new method get_edge_endpoints() that takes an edge_id as input and returns the source and destination vertices of the corresponding edge:

```
class DirectedGraph:
    # ... existing code ...

    def get_edge_endpoints(self, edge_id):
        for source in self.graph:
            for dest, weight, id_ in self.graph[source]:
                if id_ == edge_id:
                    return source, dest
        return None
```

This method iterates over all the edges in the graph and checks if the edge id matches the input edge_id.

If it finds a match, it returns the source and destination vertices of the edge as a tuple. If it doesn't find a match, it returns None.

Note that this assumes that each edge id is unique within the graph. If that's not the case, this method may return arbitrary results.

or

To get the endpoints of an edge specified by an Edge_id, we first need to modify the add_edge method in the DirectedGraph class to keep track of the edge ids:

```
class DirectedGraph:
    # ... existing code ...

    def get_edge_endpoints(self, edge_id):
        if edge_id not in self.edges:
            print("Edge ", edge_id, " does not exist.")
            return None
        else:
            return self.edges[edge_id]
```

To get the endpoints of an edge specified by an Edge_id, we first need to modify the add_edge method in the DirectedGraph class to keep track of the edge ids:

Note that we have added a new edges dictionary to keep track of the edges and their corresponding vertices.

Next, we can add a new method called get_edge_endpoints to the DirectedGraph class that takes an edge id as input and returns a tuple containing the source and target vertices of the edge:

This method first checks if the specified edge id exists in the edges dictionary. If it does not exist, the method prints an error message and returns None. Otherwise, the method returns a tuple containing the source and target vertices of the edge.

Here's an example of how to use the get_edge_endpoints method. Suppose we have the following directed graph:

```
endpoints = graph.get_edge_endpoints(2)
if endpoints is not None:
    source, target = endpoints
    print(f"Endpoints of edge {2}: {source} -> {target}")
```

This will output:

Endpoints of edge 2: 2 -> 3

```
def get_edge_endpoints(self, edge_id):
        if edge_id not in self.edges:
            print("Edge ", edge_id, " does not exist.")
            return None
        else:
            return self.edges[edge_id]
```

In this updated implementation, the add_edge method has been modified to store a unique identifier (edge_id) for each edge, and the edge information is stored in the self.edges dictionary. The __str__ method has been updated to print the edge id and weight for each edge. The get_edge_endpoints method returns the endpoints of the edge specified by the edge_id (if the edge exists), and None otherwise.

```
0 1 4 1
0 7 8 2
1 2 8 3
1 7 11 4
2 3 7 5
2 8 2 6
2 5 4 7
3 4 9 8
```

Based on the previous code that you shared, it looks like each line in the input file should contain 4 values: v1, v2, e, and edge_id. It's possible that the formatting of the input file is incorrect, or that there is an issue with the code that reads the input file.

Drive code

```
graph.read_graph("C:/Users/user/Documents/2022_2023/Graf
Theories/Lab2/input2.txt")
print("Number of vertices:", graph.num_vertices())

print("Vertices:")
for v in graph.vertices():
    print(v)

print("Graph:")
print(graph)

for v in graph.vertices():
    print(f"Inbound edges of vertex {v}:")
    for source, weight in graph.inbound_edges(v):
        print(f"{source} -> {v} edge weight: {weight}")
    print(f"Outbound edges of vertex {v}:")
    for target, weight in graph.outbound_edges(v):
        print(f"{v} -> {target} edge weight: {weight}")

for v in graph.vertices():
    print(f"Node {v}:")
    for target, weight in graph.outbound_edges(v):
        edge_id = list(filter(lambda x:
graph.get_edge_endpoints(x) == (v, target)
```

The code appears to be using a custom implementation of a directed graph data structure, with methods to read a graph from a file, print information about the graph, and retrieve and modify edge weights.

The first few lines of the code create a new instance of a directed graph object, and then read the graph from a file located at "C:/Users/user/Documents/2022_2023/GrafTheories/Lab2/input2.txt".

The code then prints the number of vertices in the graph, and the vertices themselves. It then prints a representation of the entire graph, which likely includes all vertices and edges.

The next part of the code iterates through each vertex in the graph, and for each vertex, prints information about the inbound and outbound edges of that vertex.

Specifically, for each vertex, it prints a list of inbound edges, which are edges that point to the vertex from another vertex, along with the weight of each inbound edge. It then prints a list of outbound edges, which are edges that start at the vertex and point to another vertex, along with the weight of each outbound edge.

The code then iterates through each vertex in the graph again, and for each vertex, prints more detailed information about the inbound and outbound edges.

For each outbound edge, it prints the target vertex, and the ID of the edge connecting the source vertex to the target vertex.

For each inbound edge, it prints the source vertex, and the ID of the edge connecting the source vertex to the target vertex.

PB8 retrieve or modify the information (the integer) attached to a specified edge.

```
def get_edge_endpoints(self, edge_id):
    if edge_id not in self.edges:
        print("Edge ", edge_id, " does not exist.")
        return None
    else:
        return self.edges[edge_id]

def get_edge_weight(self, edge_id):
    if edge_id not in self.edges:
        print("Edge ", edge_id, " does not exist.")
        return None
    else:
        return self.edges[edge_id]

def set_edge_weight(self, edge_id, weight):
    if edge_id not in self.edges:
        print("Edge ", edge_id, " does not exist.")
    else:
        self.edges[edge_id] = weight
```

These three methods (get_edge_endpoints, get_edge_weight, and set_edge_weight) are used to retrieve or modify the information (the integer weight) attached to a specified edge.

get_edge_endpoints(self, edge_id) takes an edge_id as input and returns the endpoints of the edge as a tuple (source, target). If the edge_id does not exist in the graph's edges dictionary, it returns None and prints an error message.

get_edge_weight(self, edge_id) takes an edge_id as input and returns the weight of the edge. If the edge_id does not exist in the graph's edges dictionary, it returns None and prints an error message.

set_edge_weight(self, edge_id, weight) takes an edge_id and a weight as input and sets the weight of the specified edge to the input weight. If the edge_id does not exist in the graph's edges dictionary, it prints an error message.

Example

Assume we have the following directed graph with edges labeled by their endpoints and weights:

```
graph = DirectedGraph()
graph.read_graph("input.txt")
```

0 -> 1 (4)
0 -> 2 (3)
1 -> 3 (5)
2 -> 3 (6)
3 -> 4 (2)

Now let's say we want to retrieve the endpoints and weight of the edge with edge_id 2 (which corresponds to the edge (0, 2)):

```
endpoints = graph.get_edge_endpoints(2)
weight = graph.get_edge_weight(2)
print(endpoints)  # Output: (0, 2)
print(weight)  # Output: 3
```

let's say we want to modify the weight of the edge with edge_id 3 (which corresponds to the edge (1, 3)):

```
graph.set_edge_weight(3, 10)
new_weight = graph.get_edge_weight(3)
print(new_weight)  # Output: 10
```

After running these operations, the graph now looks like this:

0 -> 1 (4)
0 -> 2 (3)
1 -> 3 (10)
2 -> 3 (6)
3 -> 4 (2)

DirectedGraph class that includes the ability to add and remove vertices and edges, as well as the ability to make copies of the graph:

However, I can provide a general outline of how these functionalities can be implemented in a directed graph data structure.

Adding a vertex:

Add a new vertex to the set of vertices
Initialize an empty list for storing outbound edges from the new vertex
Removing a vertex:

Remove the vertex from the set of vertices
Remove all edges that connect to the vertex (both inbound and outbound)
Adding an edge:

Check if the source and target vertices exist in the graph
If they do, add a new edge with the specified weight from the source to the target vertex
Add the new edge to the list of outbound edges for the source vertex
Add the new edge to the list of inbound edges for the target vertex
Removing an edge:

Find the edge with the specified source and target vertices
Remove the edge from the list of outbound edges for the source vertex
Remove the edge from the list of inbound edges for the target vertex

Define a class for the TAD representation of a graph

This code is implementing a graph data structure using an adjacency list representation.
The graph is represented as a dictionary where the keys are the vertices and the values are also dictionaries.
Each vertex dictionary contains the adjacent vertices as keys and the edge IDs as values.

STEP 1

```
class Graph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.next_edge_id = 0
```

The constructor method initializes the graph object by creating empty dictionaries for vertices and edges, and sets next_edge_id to 0.

The vertices dictionary is used to store the vertices in the graph. It is an empty dictionary at the start, but it will be populated with vertices as they are added to the graph using the add_vertex method.

The edges dictionary is used to store the edges in the graph. Like the vertices dictionary, it is initially empty, and edges are added using the add_edge method.

The next_edge_id variable is used to assign unique identifiers to the edges in the graph. This is useful when multiple edges between the same pair of vertices are allowed in the graph, or when there are directed edges in the graph. Whenever a new edge is added to the graph, next_edge_id is incremented to ensure that each edge is assigned a unique identifier.

Overall, the constructor method creates an empty graph object that is ready to have vertices and edges added to it using the add_vertex and add_edge methods.

```
class Graph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.next_edge_id = 0

    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = {}

    def remove_vertex(self, vertex):
        if vertex in self.vertices:
            for dest in list(self.vertices[vertex]):
                edge_id = self.vertices[vertex][dest]
                self.remove_edge(edge_id)
            del self.vertices[vertex]
    def add_edge(self, source, dest, weight):
        self.add_vertex(source)
        self.add_vertex(dest)
        edge_id = self.next_edge_id
        self.next_edge_id += 1
        self.vertices[source][dest] = edge_id
        self.edges[edge_id] = (source, dest, weight)
```

The functions add_vertex and remove_vertex add and remove a vertex respectively. If the vertex being added is not already in the graph, it is added to the vertices dictionary as a key with an empty dictionary as its value. If the vertex being removed exists in the vertices dictionary, it removes all the edges that are connected to it by iterating over its adjacent vertices, removing the edge from the edges dictionary, and then removing the vertex itself from the vertices dictionary.

The function add_edge adds an edge to the graph. If the source vertex and destination vertex do not exist in the graph, they are added to the vertices dictionary using the add_vertex function. A unique edge ID is then generated and the edge is added to the edges dictionary with the source vertex, destination vertex, and weight as its value. Finally, the edge ID is added to the adjacent vertices' dictionary of the source vertex with the destination vertex as the key and the edge ID as the value.

Note that this code does not support multi-edges or self-loops, and the weight of each edge is assumed to be a numerical value.

```
def remove_edge(self, edge_id):
    if edge_id in self.edges:
        source, dest, weight = self.edges[edge_id]
        del self.edges[edge_id]
        if dest in self.vertices.get(source, {}):
            del self.vertices[source][dest]

def get_weight(self, edge_id):
    if edge_id in self.edges:
        return self.edges[edge_id][2]
    else:
        return None

def set_weight(self, edge_id, weight):
    if edge_id in self.edges:
        source, dest, old_weight = self.edges[edge_id]
        self.edges[edge_id] = (source, dest, weight)
    else:
        print(f"Edge {edge_id} does not exist in the graph")

def __str__(self):
    output = ''
    for edge_id, (source, dest, weight) in self.edges.items():
        output += f"{source} {dest} {weight} {edge_id}\n"
    return output
```

These functions are related to managing the edges of the graph.

The function remove_edge takes an edge ID and removes the edge from the edges dictionary if it exists. It also removes the edge from the adjacent vertices' dictionaries by deleting the corresponding key-value pair in the source vertex's dictionary.

The function get_weight takes an edge ID and returns the weight of the edge if it exists in the edges dictionary. If the edge ID does not exist in the edges dictionary, it returns None.

The function set_weight takes an edge ID and a weight value and sets the weight of the edge to the new value if the edge exists in the edges dictionary. It updates the edges dictionary by replacing the old weight value with the new weight value for the given edge ID. If the edge ID does not exist in the edges dictionary, it prints a message saying that the edge does not exist in the graph.

Note that these functions assume that edges are uniquely identified by their edge ID, and there are no duplicate edge IDs.

```
# Step 1: Read graph from text file and convert to TAD
graph = Graph()
with open('graph.txt') as file:
    for line in file:
        source, dest, weight = line.split()
        graph.add_edge(source, dest, int(weight))

# Step 2: Add and remove vertices and edges
# Add a new vertex
new_vertex = 'D'
graph.add_vertex(new_vertex)

# Remove a vertex and all its incident edges
vertex_to_remove = 'C'
graph.remove_vertex(vertex_to_remove)

# Add a new edge
new_source = 'A'
new_dest = 'D'
new_weight = 5
graph.add_edge(new_source, new_dest, new_weight)

# Remove an existing edge
edge_to_remove = 0  # Replace with the edge ID to remove
graph.remove_edge(edge_to_remove)

# Step 3: Retrieve or modify information attached to specified edge using its ID
edge_id = 1  # Replace with the edge ID to retrieve or modify
# To retrieve the information attached to the edge:
weight = graph.get_weight(edge_id)
if weight is not None:
    source, dest, _ = graph.edges[edge_id]
    print(f"The weight of edge {source}-{dest} (ID {edge_id}) is {weight}")
else:
    print(f"Edge ID {edge_id} does not exist in the graph")
```

```
# To modify the information attached to the edge:
new_weight = 10  # Replace with the new weight value
graph.set_weight(edge_id, new_weight)
source, dest, _ = graph.edges[edge_id]
print(f"New weight of edge {source}-{dest} (ID {edge_id}) is
{new_weight}")
```

Copying the graph:

Create a new instance of the DirectedGraph class
Copy all vertices from the original graph to the new graph
Copy all edges from the original graph to the new graph, along with their weights and endpoints
Note that the specific implementation of these functionalities may depend on the data structures and algorithms used to represent the graph. For example, the implementation may use adjacency matrices, adjacency lists, or other data structures to store the vertices and edges.

```
class DirectedGraph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.edge_weights = {}
        self.next_edge_id = 0

    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = {}

    def remove_vertex(self, vertex):
        if vertex in self.vertices:
            for dest in list(self.vertices[vertex]):
                edge_id = self.vertices[vertex][dest]
                self.remove_edge(edge_id)
            del self.vertices[vertex]
```

```
def add_edge(self, source, dest, weight):
    self.add_vertex(source)
    self.add_vertex(dest)
    edge_id = self.next_edge_id
    self.next_edge_id += 1
    self.vertices[source][dest] = edge_id
    self.edges[edge_id] = (source, dest)
    self.edge_weights[edge_id] = weight

def remove_edge(self, edge_id):
    if edge_id in self.edges:
        source, dest = self.edges[edge_id]
        del self.edges[edge_id]
        if dest in self.vertices.get(source, {}):
            del self.vertices[source][dest]
        if edge_id in self.edge_weights:
            del self.edge_weights[edge_id]

def get_weight(self, edge_id):
    if edge_id in self.edge_weights:
        return self.edge_weights[edge_id]
    else:
        return None
```

```
def set_weight(self, edge_id, weight):
    if edge_id in self.edge_weights:
        self.edge_weights[edge_id] = weight
    else:
        print(f"Edge {edge_id} does not exist in the graph")

    def copy_graph(self):
        new_graph = DirectedGraph()
        # copy all vertices
        for vertex in self.vertices:
            new_graph.add_vertex(vertex)
        # copy all edges with their weights
        for edge_id in self.edges:
            source, dest = self.edges[edge_id]
            weight = self.edge_weights[edge_id]
            new_graph.add_edge(source, dest, weight)
        return new_graph
```

```
# create a new graph
graph1 = DirectedGraph()

# add vertices and edges to the graph
graph1.add_vertex('A')
graph1.add_vertex('B')
graph1.add_vertex('C')
graph1.add_edge('A', 'B', 1)
graph1.add_edge('B', 'C', 2)
graph1.add_edge('C', 'A', 3)

# copy the graph
graph2 = graph1.copy_graph()

# print the vertices and edges of the original graph
print("Original graph:")
print(graph1.vertices)
print(graph1.edges)
print(graph1.edge_weights)

# print the vertices and edges of the copied graph
print("Copied graph:")
print(graph2.vertices)
print(graph2.edges)
print(graph2.edge_weights)
```

The output of the above code will be:

```
Original graph:
{'A': {'B': 0}, 'B': {'C': 1}, 'C': {'A': 2}}
{0: ('A', 'B'), 1: ('B', 'C'), 2: ('C', 'A')}
{0: 1, 1: 2, 2: 3}
Copied graph:
{'A': {}, 'B': {}, 'C': {}}
{0: ('A', 'B'), 1: ('B', 'C'), 2: ('C', 'A')}
{0: 1, 1: 2, 2: 3}
```

In your opinion, is the solution, offered to you on this slide, ok?

If yes why?
If not why?

```
class DirectedGraph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.next_edge_id = 0

    def copy_graph(self):
        new_graph = DirectedGraph()
        new_graph.vertices = self.vertices.copy()
        new_graph.edges = {}
        new_graph.next_edge_id = self.next_edge_id

        def lazy_copy_edge(source, dest, weight):
            edge_id = self.vertices[source][dest]
            new_graph.edges[edge_id] = (source, dest, weight)

        for source in self.vertices:
            for dest, weight in self.vertices[source].items():
                lazy_copy_edge(source, dest, weight)

        return new_graph
```

This implementation creates a new DirectedGraph instance and copies the vertex dictionary from the original graph to the new graph using the dict.copy() method, which creates a shallow copy of the dictionary.

The method then defines a lazy_copy_edge function that takes the source, destination, and weight of an edge in the original graph, and uses the edge ID to copy the edge to the new graph. Rather than copying the edge immediately, the function adds the edge ID and its data to the edges dictionary of the new graph.

Finally, the method iterates over all vertices in the original graph and uses the lazy_copy_edge function to copy each edge to the new graph. It then returns the new graph instance.

This implementation has a time complexity of O(|V| + |E|), since it iterates over all vertices and edges in the original graph.

However, it has a memory complexity of O(|V|), since it only stores the edges in the edges dictionary of the new graph when they are accessed. This can be useful for large graphs where memory usage is a concern.

Random graph

import random

is a statement in Python that allows you to use the random module in your code.

The random module provides a suite of functions for generating random numbers. These functions are useful for many purposes, such as generating a random number within a given range or shuffling a list randomly.

In the context of the Graph class, the random module is used to generate random edges and weights for the create_random_graph method. This method creates a graph with a specified number of vertices and edges, and assigns a random weight to each edge.

The @classmethod decorator is used to define a class method that can be called on the class itself, rather than on an instance of the class. This allows you to create new instances of the class using the class method, such as Graph.create_random_graph(5, 6).

Algoritm

Here is an algorithm for creating a random graph with a specified number of vertices and edges:

Define a function create_random_graph that takes two arguments: num_vertices and num_edges.

Create an empty graph object graph using the Graph class.

If num_edges is greater than num_vertices * (num_vertices - 1) / 2, raise a ValueError with a message indicating that there are too many edges for the number of vertices.

Generate a list of num_vertices vertices labeled from 0 to num_vertices - 1.

Shuffle the list of vertices using the random.shuffle() function from the random module.

While the number of edges in the graph is less than num_edges, repeat the following steps:

Choose two distinct vertices randomly from the shuffled list of vertices.
Add an edge between the two vertices with a random weight between 1 and 10 using the graph.add_edge() method.
Return the graph object.

Here is the Python code implementing the algorithm:

```
import random
class Graph:
    def __init__(self):
        self.adj_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, u, v, weight):
        self.adj_list[u].append((v, weight))
        self.adj_list[v].append((u, weight))

    @classmethod
    def create_random_graph(cls, num_vertices, num_edges):
        if num_edges > num_vertices * (num_vertices - 1) / 2:
            raise ValueError("Too many edges")
        graph = cls()
        vertices = list(range(num_vertices))
        random.shuffle(vertices)
        num_edges_added = 0
        while num_edges_added < num_edges:
            u, v = random.sample(vertices, 2)
            graph.add_vertex(u)
            graph.add_vertex(v)
            graph.add_edge(u, v, random.randint(1, 10))
            num_edges_added += 1
        return graph
```

You can call the create_random_graph function as follows:

graph = Graph.create_random_graph(5, 6)
print(graph.adj_list)

This will create a random graph with 5 vertices and 6 edges, and print its adjacency list.

```
def __init__(self):
    self.adj_list = {}

def add_vertex(self, vertex):
    if vertex not in self.adj_list:
        self.adj_list[vertex] = []
```

These two methods are part of the Graph class and are responsible for creating and managing the adjacency list of the graph.

__init__(self) is a special method in Python classes that is called when a new instance of the class is created. In this case, it initializes the adj_list attribute of the class as an empty dictionary, which will store the adjacency list of the graph.

add_vertex(self, vertex) is a method that adds a vertex to the graph by adding an empty list to the adjacency list if the vertex is not already present. The keys of the dictionary adj_list are the vertices, and the corresponding values are lists that contain tuples representing the edges of the graph.

The add_edge method in the Graph class is used to add an edge between two vertices of the graph. It takes three parameters: u, v, and weight.

```
def add_edge(self, u, v, weight):
    self.adj_list[u].append((v, weight))
    self.adj_list[v].append((u, weight))
```

The first two parameters u and v are the two vertices between which the edge is being added, and the third parameter weight is the weight or cost of the edge.

The method first appends a tuple containing the destination vertex and the weight to the adjacency list of the source vertex u, using the append() method. Then it appends a similar tuple to the adjacency list of the destination vertex v.

By appending the tuple to both vertices' adjacency lists, it ensures that the edge is added to the graph for both vertices, as the graph is undirected.

```
def create_random_graph(cls, num_vertices, num_edges):
    if num_edges > num_vertices * (num_vertices - 1) / 2:
        raise ValueError("Too many edges")
    graph = cls()
    vertices = list(range(num_vertices))
    random.shuffle(vertices)
    num_edges_added = 0
    while num_edges_added < num_edges:
        u, v = random.sample(vertices, 2)
        graph.add_vertex(u)
        graph.add_vertex(v)
        graph.add_edge(u, v, random.randint(1, 10))
        num_edges_added += 1
    return graph
```

The create_random_graph method is a class method that takes three arguments: cls, num_vertices, and num_edges. It creates a random graph with the specified number of vertices and edges.

First, the method checks if the number of edges is greater than the maximum possible number of edges in a graph with the specified number of vertices. If it is, a ValueError is raised.

The method creates an instance of the Graph class and initializes an empty list of vertices. Then, it shuffles the list of vertices randomly.

Next, the method enters a loop to add edges to the graph. The loop continues until the number of edges added is equal to the specified number of edges.

In each iteration of the loop, the method selects two random vertices from the shuffled list of vertices. It then adds these vertices to the graph and creates an edge between them with a randomly generated weight between 1 and 10.

Finally, the method returns the completed random graph.

Representation of the directed graph 3 dictionaries

The internal representation of the directed graph can be done using three dictionaries.

1.The first dictionary is used to represent the vertices. The keys of the dictionary are the vertices, and the values are the adjacent vertices of the corresponding vertex. Each adjacent vertex is represented as a key in a sub-dictionary, and the value of the sub-dictionary can be used to store information about the edge (such as the weight of the edge). Here is an example of the vertices dictionary:

```
{
   "A": {"B": 2, "C": 1},
   "B": {"C": 3, "D": 2},
   "C": {"D": 1},
   "D": {"A": 1}
}
```

The second dictionary is used to represent the edges. The keys of the dictionary are the unique edge IDs, and the values are tuples that contain the source vertex, the destination vertex, and the weight of the edge. Here is an example of the edges dictionary:

```
{
   0: ("A", "B", 2),
   1: ("A", "C", 1),
   2: ("B", "C", 3),
   3: ("B", "D", 2),
   4: ("C", "D", 1),
   5: ("D", "A", 1)
}
```

The third dictionary is used to keep track of the next available edge ID. This can be initialized to 0 and incremented every time a new edge is added to the graph. Here is an example of the next_edge_id dictionary:

```
{
   "next_edge_id": 6
}
```

Overall, this representation allows for efficient access to both the vertices and edges of the graph. The first dictionary allows for constant time lookup of adjacent vertices and edge information for a given vertex, and the second dictionary allows for constant time lookup of edge information for a given edge ID.

```
class DirectedGraph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.next_edge_id = 0

    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = {}

    def remove_vertex(self, vertex):
        if vertex in self.vertices:
            for dest in list(self.vertices[vertex]):
                edge_id = self.vertices[vertex][dest]
                self.remove_edge(edge_id)
            del self.vertices[vertex]

    def add_edge(self, source, dest, weight):
        self.add_vertex(source)
        self.add_vertex(dest)
        edge_id = self.next_edge_id
        self.next_edge_id += 1
        self.vertices[source][dest] = edge_id
        self.edges[edge_id] = (source, dest, weight)

    def remove_edge(self, edge_id):
        if edge_id in self.edges:
            source, dest, weight = self.edges[edge_id]
            del self.edges[edge_id]
            if dest in self.vertices.get(source, {}):
                del self.vertices[source][dest]

    def get_weight(self, edge_id):
        if edge_id in self.edges:
            return self.edges[edge_id][2]
        else:
            return None

    def set_weight(self, edge_id, weight):
        if edge_id in self.edges:
            source, dest, old_weight = self.edges[edge_id]
            self.edges[edge_id] = (source, dest, weight)
        else:
            print(f"Edge {edge_id} does not exist in the graph")
```

This code defines a DirectedGraph class that has three instance variables for the three dictionaries that represent the graph: vertices, edges, and next_edge_id. The class also has methods for adding and removing vertices and edges, as well as getting and setting the weight of an edge.

```
g = DirectedGraph()
g.add_vertex('A')
g.add_vertex('B')
g.add_edge('A', 'B', 2)
```

This creates a directed graph with two vertices ('A' and 'B') and one edge between them with weight 2. You can then use the remove_vertex, remove_edge, get_weight, and set_weight methods to modify the graph as needed.

The preconditions (i.e. requirements that must be satisfied before a given operation can be performed) for the operations in the DirectedGraph TAD:

add_vertex(vertex): Adds a new vertex to the graph.

Precondition: vertex must be a hashable object (e.g. string, number, tuple, etc.).
remove_vertex(vertex): Removes a vertex from the graph.

Precondition: vertex must be a vertex in the graph.
add_edge(source, dest, weight): Adds a new edge with the given weight between the source and destination vertices.

Precondition: source and dest must be vertices in the graph, and weight must be a non-negative number.
remove_edge(edge_id): Removes the edge with the given edge_id from the graph.

Precondition: edge_id must be an edge ID that corresponds to an edge in the graph.
get_weight(edge_id): Returns the weight of the edge with the given edge_id.

Precondition: edge_id must be an edge ID that corresponds to an edge in the graph.
set_weight(edge_id, weight): Sets the weight of the edge with the given edge_id to the given weight.

Precondition: edge_id must be an edge ID that corresponds to an edge in the graph, and weight must be a non-negative number.
Note that these preconditions ensure that the operations are performed correctly and that the internal representation of the graph remains consistent. If these preconditions are not satisfied, the operations may result in unexpected behavior or errors.

Here is a possible solution

```
class DirectedGraph:
    def __init__(self):
        self.vertices = {}
        self.edges = {}
        self.next_edge_id = 0

    def add_vertex(self, vertex):
        assert isinstance(vertex, Hashable), "Vertex must be a hashable object"
        if vertex not in self.vertices:
            self.vertices[vertex] = {}

    def remove_vertex(self, vertex):
        assert vertex in self.vertices, "Vertex must exist in the graph"
        for dest in list(self.vertices[vertex]):
            edge_id = self.vertices[vertex][dest]
            self.remove_edge(edge_id)
        del self.vertices[vertex]

    def add_edge(self, source, dest, weight):
        assert source in self.vertices, "Source vertex must exist in the graph"
        assert dest in self.vertices, "Destination vertex must exist in the graph"
        assert isinstance(weight, (int, float)) and weight >= 0, "Weight must be a non-negative number"
```

```
edge_id = self.next_edge_id
        self.next_edge_id += 1
        self.vertices[source][dest] = edge_id
        self.edges[edge_id] = (source, dest, weight)

    def remove_edge(self, edge_id):
        assert edge_id in self.edges, "Edge ID must exist in the graph"
        source, dest, weight = self.edges[edge_id]
        del self.edges[edge_id]
        if dest in self.vertices.get(source, {}):
            del self.vertices[source][dest]

    def get_weight(self, edge_id):
        assert edge_id in self.edges, "Edge ID must exist in the graph"
        return self.edges[edge_id][2]

    def set_weight(self, edge_id, weight):
        assert edge_id in self.edges, "Edge ID must exist in the graph"
        assert isinstance(weight, (int, float)) and weight >= 0, "Weight must be a non-negative
number"
        source, dest, old_weight = self.edges[edge_id]
        self.edges[edge_id] = (source, dest, weight)
```

```
# create a new DirectedGraph instance
graph = DirectedGraph()

# add some vertices to the graph
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_vertex('D')

# add some edges to the graph
graph.add_edge('A', 'B', 5)
graph.add_edge('B', 'C', 2)
graph.add_edge('C', 'D', 3)
graph.add_edge('D', 'A', 1)
graph.add_edge('A', 'C', 4)

# remove an edge from the graph
edge_id = graph.vertices['A']['B']
graph.remove_edge(edge_id)

# remove a vertex from the graph
graph.remove_vertex('C')
```

```
# set the weight of an edge in the graph
edge_id = graph.vertices['A']['C']
graph.set_weight(edge_id, 2)

# get the weight of an edge in the graph
edge_id = graph.vertices['A']['D']
weight = graph.get_weight(edge_id)
print(f"The weight of the edge from A to D is {weight}")
```

Homework

Implement a menu and using it test all the operations on small graphs and on the large files: graph1k, graph10k, graph100k …

please run and adopt the code created in the laboratory for the large files: graph 1k, graph 10k, graph 100k ...
which exit on site