Practical work no. 1 – Oriented Graph

Specification

I defined the class Graph representing a directed graph.

The class **Graph** provides the following methods:

- def add_vertex(self, vertex):

This function adds a new vertex to a graph. It checks if the vertex already exists in the graph, and if so, it raises a ValueError. Otherwise, it adds the vertex to the graph by initializing its in-degree and out-degree dictionaries as empty lists. Finally, it increments the total number of vertices in the graph.

Preconditions: Vertex should not exist in the graph.

```
def add_vertex(self, vertex):
    """
    Adds a vertex to the graph
    :param vertex: the vertex to be added
    :raises: ValueError if the vertex already exists
    """
    if self.is_vertex(vertex):
        raise ValueError(f"Vertex {vertex} already exists!")
    self.__din[vertex] = []
    self.__dout[vertex] = []
    self.__length += 1
```

- def add_edge(self, vertex):

This function adds a new edge to the graph between two vertices, vertex1 and vertex2, with a specified cost. It checks if the edge already exists and raises a ValueError if it does. Otherwise, it updates the graph's internal representation to include the new edge and its cost.

Preconditions: Edge should not exist in the graph.

```
def add_edge(self, vertex1, vertex2, cost):
    """
    Adds an edge to the graph
    :param vertex1: the starting vertex
    :param vertex2: the ending vertex
    :param cost: the cost of the edge
    :raises: ValueError if the edge already exists
    """
    if self.is_edge(vertex1, vertex2):
        raise ValueError(f"Edge ({vertex1}, {vertex2}) already exists!")
    self.__din[vertex2].append(vertex1)
    self.__dout[vertex1].append(vertex2)
    self.__dcost[(vertex1, vertex2)] = cost
```

- def is_vertex(self, vertex):

This function checks if a vertex exists in the graph by verifying if it appears as a key in both the in-degree and out-degree dictionaries of the graph. If the vertex exists, it returns True; otherwise, it returns False.

```
def is_vertex(self, vertex):
    """
    Checks if a vertex exists in the graph or not
    :param vertex: the vertex to be checked
    :return: true if the vertex exists, false otherwise
    """
    if (vertex in self.__din.keys()) and (vertex in self.__dout.keys()):
        return True
    return False
```

- def is_edge(self, vertex1, vertex2):

This function checks if an edge exists in the graph between two given vertices, vertex1 and vertex2. It first verifies if both vertices exist in the graph. Then, it checks if there exists an edge from vertex1 to vertex2. If such edges exist, it returns True, indicating that the edge exists in the graph. Otherwise, it implicitly returns None, meaning the edge does not exist.

Preconditions: Vertices should exist in the graph.

```
def is_edge(self, vertex1, vertex2):
    """
    Checks if an edge exists in the graph or not
    :param vertex1: the starting vertex
    :param vertex2: the ending vertex
    :raises: ValueError if one of the vertices does not exist
    """
    if not self.is_vertex(vertex1):
        raise ValueError(f"Vertex {vertex1} does not exist! Try another one.")
    if not self.is_vertex(vertex2):
        raise ValueError(f"Vertex {vertex2} does not exist! Try another one.")

    if vertex1 in self.__din[vertex2] and vertex2 in self.__dout[vertex1]:
        return True
```

- def remove_vertex(self, vertex):

This function deletes a vertex from the graph along with all edges connected to it. It first verifies the existence of the vertex. Then, it iterates over outgoing and incoming vertices, removing corresponding edges and their costs. Afterward, it removes the vertex itself from the graph and decrements the total number of vertices.

Preconditions: Vertex should exist in the graph.

```
def remove_vertex(self, vertex):
    """
    Removes a vertex from the graph
    :param vertex: the vertex to be removed
    :raises: ValueError if the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError(f"Vertex {vertex} does not exist!")
    for vertex2 in self.parse_vertex_out(vertex):
        self.__din[vertex2].remove(vertex)
        del self.__dcost[(vertex, vertex2)]
    for vertex1 in self.parse_vertex_in(vertex):
        self.__dout[vertex1].remove(vertex)
        del self.__dcost[(vertex1, vertex)]
    del self.__din[vertex]
    del self.__dout[vertex]
    self.__length -= 1
```

- def remove_edge(self, vertex1, vertex2):

This remove_edge function removes an edge from the graph between the given vertex1 and vertex2. It first checks if the edge exists. If the edge does not exist, it raises a ValueError. Then, it removes the edge from the graph's internal representation by deleting the corresponding entries in the in-degree and out-degree dictionaries, along with removing the associated cost from the cost dictionary.

Preconditions: Edge should not exist.

```
def remove_edge(self, vertex1, vertex2):
    """
    Removes an edge from the graph
    :param vertex1: the starting vertex
    :param vertex2: the ending vertex
    :raises: ValueError if the edge does not exist
    """
    if not self.is_edge(vertex1, vertex2):
        raise ValueError(f"Edge ({vertex1}, {vertex2}) does not exist!")
    self.__din[vertex2].remove(vertex1)
    self.__dout[vertex1].remove(vertex2)
    del self.__dcost[(vertex1, vertex2)]
```

- @property def number_vertices(self):

This property provides a getter method for accessing the number of vertices in the graph. It returns the value representing the total number of vertices in the graph.

```
@property
def number_vertices(self):
    """
    Getter for the number of vertices
    :return: the number of vertices of the graph
    """
    return self.__length
```

- @setter def number_vertices(self):

This setter method allows modification of the number of vertices in the graph. It updates the length of the graph with the provided value, effectively changing the total number of vertices in the graph.

```
@number_vertices.setter
def number_vertices(self, value):
    """
    Setter for the number of vertices
    :param value: the new number of vertices
    """
    self.__length = value
```

- @property number_edges(self):

This property provides a getter method for accessing the number of edges in the graph. It calculates and returns the number of edges.

```
@property
def number_edges(self):
    """
    Getter for the number of edges
    :return: the number of edges of the graph
    """
    return len(self.__dcost)
```

- def get_cost(self, vertex1, vertex2):

This function retrieves the cost associated with an edge between two given vertices in the graph. It first checks if the edge exists using the is_edge method. If the edge does not exist, it raises a ValueError. Then, it returns the cost corresponding to the provided edge.

Preconditions: Edge should exist.

```
def get_cost(self, vertex1, vertex2):
    """
    Getter for the cost of an edge
    :param vertex1: the starting vertex
    :param vertex2: the ending vertex
    :return: the cost of the edge
    :raises: ValueError if the edge does not exist
    """
    if not self.is_edge(vertex1, vertex2):
        raise ValueError(f"The edge ({vertex1}, {vertex2}) does not exist!")
    return self.__dcost[(vertex1, vertex2)]
```

- def modify_cost(self, vertex1, vertex2):

This function updates the cost associated with the edge between two given vertices in the graph to the provided new_cost. It first verifies the existence of the . If the edge does not exist, it raises a ValueError. Othewise, it updates the cost for the specified edge with the new cost.

Preconditions: Edge should exist.

```
def modify_cost(self, vertex1, vertex2, new_cost):
    """
    Modifies the cost of an edge
    :param vertex1: the starting vertex
    :param vertex2: the ending vertex
    :param new_cost: the new cost of the edge
    :raises: ValueError if the edge does not exist
    """
    if not self.is_edge(vertex1, vertex2):
        raise ValueError(f"The edge ({vertex1}, {vertex2}) does not exist!")
    self.__dcost[(vertex1, vertex2)] = new_cost
```

- def parse_vertices(self):

This method retrieves and returns the list of vertices in the graph. It achieves this by extracting the keys from the dictionary, which represents the vertices of the graph along with their incoming edges.

```
def parse_vertices(self):
    """
    Parses the vertices of the graph
    :return: the list of vertices
    """
    return list(self.__din.keys())
```

- def parse_vertex_in(self, vertex):

This method retrieves and returns the list of inbound vertices for a given vertex. It first checks if the vertex exists in the graph. If the vertex does not exist, it raises a ValueError. Otherwise, it extracts the list of inbound vertices from the dictionary, representing the incoming edges to the specified vertex.

Preconditions: Vertex should exist.

```
def parse_vertex_in(self, vertex):
    """
    Parses the inbound vertices of a vertex
    :param vertex: the vertex to be parsed
    :return: the list of inbound vertices
    :raises: ValueError if the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist!")
    return list(self.__din[vertex])
```

- def parse_vertex_out(self, vertex):

This method retrieves and returns the list of outbound vertices for a given vertex. It first checks if the vertex exists in the graph. If the vertex does not exist, it raises a ValueError. Otherwise, it extracts the list of outbound vertices from the dictionary, representing the outgoing edges to the specified vertex.

Preconditions: Vertex should exist.

```
def parse_vertex_out(self, vertex):
    """
    Parses the outbound vertices of a vertex
    :param vertex: the vertex to be parsed
    :return: the list of outbound vertices
    :raises: ValueError if the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist!")
    return list(self.__dout[vertex])
```

- def in_degree(self, vertex):

This method calculates and returns the in-degree of a specified vertex in the graph. It first checks if the vertex exists. If the vertex does not exist, it raises a ValueError. Then, it computes the in-degree by determining the number of incoming edges to the vertex.

Preconditions: Vertex should exist.

```
def in_degree(self, vertex):
    """
    Computes the in degree of a vertex
    :param vertex: the vertex for whom the in degree is computed
    :return: the in degree of the vertex
    :raises: ValueError if the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist!")
    return len(self.__din[vertex])
```

- def out_degree(self, vertex):

This method calculates and returns the out-degree of a specified vertex in the graph. It first checks if the vertex exists. If the vertex does not exist, it raises a ValueError. Then, it computes the out-degree by determining the number of outgoing edges to the vertex.

Preconditions: Vertex should exist.

```
def out_degree(self, vertex):
    """
    Computes the out degree of a vertex
    :param vertex: the vertex for whom the out degree is computed
    :return: the out degree of the vertex
    :raises: ValueError if the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist!")
    return len(self.__dout[vertex])
```

- def parse_outbound_edges(self, vertex1):

The method retrieves and returns the list of outbound edges from a specified vertex. It first verifies the existence of the vertex otherwise, it raises a ValueError. Then, it iterates over the outbound vertices of the given vertex, constructing edges in the form of tuples (vertex1, vertex2) and adding them to a list. Finally, it returns the list of outbound edges.

Preconditions: Vertex should exist.

```
def parse_outbound_edges(self, vertex1):
    """
    Parses the outbound edges of a vertex
    :param vertex1: the vertex for which the outbound edges are parsed
    :return: the list of outbound edges
    :raises: ValueError if the vertex does not exist
    """
    edges = []
    if not self.is_vertex(vertex1):
        raise ValueError("Vertex does not exist!")
    for vertex2 in self.parse_vertex_out(vertex1):
        edges.append((vertex1, vertex2))
    return list(edges)
```

- def parse_inbound_edges(self, vertex2):

The method retrieves and returns the list of inbound edges from a specified vertex. It first verifies the existence of the vertex otherwise, it raises a ValueError. Then, it iterates over the inbound vertices of the given vertex, constructing edges in the form of tuples (vertex1, vertex2) and adding them to a list. Finally, it returns the list of outbound edges.

Preconditions: Vertex should exist.

```
def parse_inbound_edges(self, vertex2):
    """
    Parses the inbound edges of a vertex
    :param vertex2: the vertex for which the inbound edges are parsed
    :return: the list of inbound edges
    :raises: ValueError if the vertex does not exist
    """
    edges = []
    if not self.is_vertex(vertex2):
        raise ValueError("Vertex does not exist!")
    for vertex1 in self.parse_vertex_in(vertex2):
        edges.append((vertex1, vertex2))
    return list(edges)
```

- def parse_edges(self):

The method retrieves and returns the list of edges in the graph along with their associated costs. It accesses the items in the dictionary, which stores the edges and their costs as key-value pairs.

```
def parse_edges(self):
    """
    Parses the edges of the graph
    :return: the list of edges
    """
    return self.__dcost.items()
```

- def save_graph_to_file(self, file_name):

The method saves the graph to a specified file. It iterates through the vertices in the graph and writes each edge along with its associated cost to the file. If a vertex is isolated (i.e., it has no outgoing or incoming edges), it writes the vertex followed by two -1 values. After writing all edges, it closes the file.

```
def save_graph_to_file(self, file_name):
    """
    Saves the graph to a file
    :param file_name: the name of the file
    """
    file = open(file_name, 'w')
    found = False
    for vertex1 in self.__dout.keys():
        # if the vertex is isolated, then we write the end vertex and the cost as -1
        if len(self.__dout[vertex1]) == 0 and len(self.__din[vertex1]) == 0:
            found = True
    if found:
        for vertex1 in self.__dout.keys():
            # if the vertex is isolated, then we write the end vertex and the cost as -1
            if len(self.__dout[vertex1]) == 0 and len(self.__din[vertex1]) == 0:
                file.write(f"{vertex1} {-1} {-1}\n")
            else:
                # if is not isolated, then we write the edge and the cost
                for vertex2 in self.__dout[vertex1]:
                    cost = self.__dcost[(vertex1, vertex2)]
                    file.write(f"{vertex1} {vertex2} {cost}\n")
    else:
        file.write(f"{self.__length} {self.number_edges}\n")
        for edge, cost in self.parse_edges():
            vertex1, vertex2 = edge
            file.write(f"{vertex1} {vertex2} {cost}\n")
    file.close()
```

- def load_graph_from_file(self, file_name):

The method loads a graph from a specified file. It reads the contents of the file and constructs the graph accordingly. If the file format indicates the number of vertices and edges on the first line, it reads the vertices and edges accordingly. Otherwise, it reads the edges with their associated costs from subsequent lines. It handles cases where vertices may be isolated or connected, adding them as necessary along with their corresponding edges and costs. If the file does not exist, it raises a FileNotFoundError.

```python
def load_graph_from_file(self, file_name):
    """
    Loads a graph from a file
    :param file_name: the name of the file
    :raises: FileNotFoundError if the file does not exist
    """
    if os.path.exists(file_name):
        file = open(file_name, 'r')
        # Read the first line
        first_line = file.readline().split()
        # if the length of the first line is 2, then the graph needs to be read in a way
        if len(first_line) == 2:
            # saving the vertices and edges
            lines = file.readlines()
            for line in lines:
                vertex1, vertex2, cost = line.strip().split()
                if not self.is_vertex(int(vertex1)):
                    self.add_vertex(int(vertex1))
                if not self.is_vertex(int(vertex2)):
                    self.add_vertex(int(vertex2))
                self.add_edge(int(vertex1), int(vertex2), int(cost))
        # if the length of the first line is 3, then the graph needs to be read in a different way
        elif len(first_line) == 3:
            # saving the first line
            vertex1, vertex2, cost = first_line
            if int(vertex2) == -1:
                self.add_vertex(int(vertex1))
            else:
                if not self.is_vertex(int(vertex1)):
                    self.add_vertex(int(vertex1))
                if not self.is_vertex(int(vertex2)):
                    self.add_vertex(int(vertex2))
                self.add_edge(int(vertex1), int(vertex2), int(cost))
        lines = file.readlines()
        # taking all the lines
        for line in lines:
            vertex1, vertex2, cost = line.strip().split()
            # if the second vertex is -1, then the first vertex is isolated
            if int(vertex2) == -1:
                self.add_vertex(int(vertex1))
            # else we add the vertices and the edge
            else:
                if not self.is_vertex(int(vertex1)):
                    self.add_vertex(int(vertex1))
                if not self.is_vertex(int(vertex2)):
                    self.add_vertex(int(vertex2))
                self.add_edge(int(vertex1), int(vertex2), int(cost))
    file.close()
```

Implementation

The class Graph uses 3 dictionaries:

- **self.__din = {}**
- **self.__dout = {}**
- **self.__cost = {}**

In and Out dictionaries contain a vertex as a key and every key has a list which shows the vertices connected to that key, depending on whether the edge starts or ends in that vertex.The cost dictionary has a tuple of vertices as key and the value is the cost of the edge between them.

Another structure that is implemented is self.__length which represents the number of vertices is the graph.

```
class Graph:
    def __init__(self, n=0):
        self.__length = n
        self.__din = {}
        self.__dout = {}
        for i in range(self.__length):
            self.__din[i] = []
            self.__dout[i] = []
        self.__dcost = {}
```