Documentation

```
from graph_exceptions import GraphException
import copy
from random import randint


class DirectedGraph:
```

The constructor initializes the directed graph object.

It takes an optional parameter number_of_vertices, which specifies the number of vertices in the graph. If not provided, it defaults to 0.

It initializes several attributes:

- number_of_vertices: Stores the number of vertices in the graph.
- number_of_edges: Stores the number of edges in the graph.
- inbound_edges: A dictionary to store the inbound edges for each vertex. Each vertex is a key, and its value is a list of vertices from which there are edges coming into the key vertex.
- outbound_edges: A dictionary to store the outbound edges for each vertex. Each vertex is a key, and its value is a list of vertices to which there are edges going out from the key vertex.
- costs: A dictionary to store the cost associated with each edge. The keys are tuples representing edges, and the values are the costs associated with those edges.

```
    def __init__(self, number_of_vertices=0):
        """
        Constructor for the DirectedGraph class
        :param number_of_vertices: the number of vertices in the
graph
        """
        self.number_of_vertices = number_of_vertices
        self.number_of_edges = 0
        self.inbound_edges = {}
        self.outbound_edges = {}
        self.costs = {}
```

Provides a string representation of the graph in it's current state.

It starts by constructing a header string indicating the number of vertices and edges in the graph. Then, it iterates through each vertex in the inbound_edges dictionary. For each vertex, it iterates through the list of inbound edges associated with that vertex. Inside the nested loop, it constructs a string representation for each inbound edge, including its associated cost. If there are no inbound edges or outbound edges for a vertex (for example an isolated vertex), it notes that in the string representation. It appends these representations to the main string and returns this string.

```
def __str__(self):
    """
    Returns a string representation of the graph
    """
    string = f"DirectedGraph: {self.number_of_vertices}
vertices, {self.number_of_edges} edges"
    for vertex, inbound_edges in self.inbound_edges.items():
        for inbound_edge in inbound_edges:
            if not inbound_edges and not
self.outbound_edges[vertex]:
                string += f"\n {vertex} -> isolated vertex"
                continue

            string += f"\n {inbound_edge} -> {vertex} ->:
{self.costs[(inbound_edge, vertex)]}"

    return string
```

This method retrieves and returns the number of vertices in the directed graph. Used to access the number of vertices if needed from outside the class. Returns the number of vertices of the current graph.

```
def get_number_of_vertices(self):
    """
    Getter for the number of vertices
    :return: the number of vertices
    """
    return self.number_of_vertices
```

This method is used to set the number of vertices in the directed graph to a new value. It takes one parameter, number_of_vertices, which represents the new number of vertices that the graph should be set to.

```
def set_number_of_vertices(self, number_of_vertices):
    """
    Setter for the number of vertices
    :param number_of_vertices: the new number of vertices
    :return: None
    """
    self.number_of_vertices = number_of_vertices
```

Getter method used to retrieve the cost associated with a specific edge in the graph.

It takes two parameters:

- source: Represents the source vertex of the edge.
- destination: Represents the destination vertex of the edge.

Inside the method, it checks whether the specified edge exists in the graph by calling the check_if_edge_exists method, if the edge exists, it proceeds to retrieve its cost otherwise it raises a GraphException.

```
def get_cost_of_an_edge(self, source, destination):
    """
    Getter for the cost of an edge
    :param source: the source vertex
    :param destination: the destination vertex
    :return: the cost of the edge
    """
    if self.check_if_edge_exists(source, destination):
        return self.costs[(source, destination)]
    else:
        raise GraphException("There is no edge from {source} to
{destination}")
```

Setter method used to set the cost associated with a specific edge in the graph.

It takes three parameters:

- source: Represents the source vertex of the edge.
- destination: Represents the destination vertex of the edge.
- cost: Represents the new cost to be assigned to the edge.

Inside the method, it checks whether the specified edge already exists in the graph by calling the check_if_edge_exists method. If the edge exists, it updates its cost. If not, it adds the edge to the graph with the specified cost.

```
def set_cost_of_an_edge(self, source, destination, cost):
    """
    Setter for the cost of an edge
    :param source: the source vertex
    :param destination: the destination vertex
    :param cost: the new cost of the edge
    :return: None
    """
    if self.check_if_edge_exists(source, destination):
        self.costs[(source, destination)] = cost
    else:
        self.add_edge(source, destination, cost)
```

Getter method used to retrieve all the vertices present in the graph. Returns a list of all the vertices present in the directed graph.

```
def get_vertices(self):
    """
    Getter for all the vertices in the graph
    :return: a list of vertices
    """
    return list(self.inbound_edges.keys())
```

Method used to check whether an edge exists between two vertices in the graph.

It takes two parameters:

- source: Represents the source vertex of the edge.
- destination: Represents the destination vertex of the edge.

Inside the method, it checks whether the specified edge exists in the graph by verifying if the tuple (source, destination) is present in the costs dictionary keys. If the edge exists in the costs dictionary, it returns True, indicating that the edge exists. Otherwise, it returns False.

```
def check_if_edge_exists(self, source, destination) -> bool:
    """
    Checks if an edge exists between two vertices
    :param source: the source vertex
    :param destination: the destination vertex
    :return: True if the edge exists, False otherwise
    """
    return (source, destination) in self.costs
```

Method used to add a new vertex to the graph. It takes one parameter:

- vertex: Represents the vertex to be added to the graph.

It first checks if the vertex already exists in the graph. It does so by verifying if the vertex is present in the keys of the inbound_edges dictionary. If the vertex already exists, it raises a GraphException indicating that the vertex already exists in the graph otherwise it adds the vertex.

It initializes the dictionaries inbound_edges and outbound_edges of that vertex with empty lists and increments the number_of_vertices.

```
def add_vertex(self, vertex):
    """
    Adds a vertex to the graph
    :param vertex: the vertex to be added
    :return: None
    """
    if vertex in self.inbound_edges:
        raise GraphException("The vertex already exists")

    self.inbound_edges[vertex] = []
    self.outbound_edges[vertex] = []
    self.number_of_vertices += 1
```

Method used to remove a vertex from the graph. It takes one parameter:

- vertex: Represents the vertex to be removed from the graph.

It first checks if the vertex exists in the graph by verifying if vertex is present in the keys of the inbound_edges dictionary. If the vertex does not exist in the graph, it raises a GraphException indicating that the vertex does not exist. If the vertex exists in the graph, it proceeds to remove it.

Decremets the number_of_vertices and updates the number_of_edges attribute by subtracting the number of inbound and outbound edges associated with the vertex.

Iterates over the inbound_edges of the vertex, deleting the corresponding edges from the costs dictionary and removing the vertex from the outbound edges of its inbound neighbors, similarly for the outbound_edges. Then it removes the vertex from the inbound_edges and outbound_edges dictionaries.

```
def remove_vertex(self, vertex):
    """
    Removes a vertex from the graph
    :param vertex: the vertex to be removed
    :return: None
    """
    if vertex not in self.inbound_edges:
        raise GraphException("The vertex does not exist")

    if vertex in self.inbound_edges:
        self.number_of_vertices -= 1
        self.number_of_edges -= len(self.inbound_edges[vertex])
+ len(self.outbound_edges[vertex])
        for inbound_vertex in self.inbound_edges[vertex]:
            del self.costs[(vertex, inbound_vertex)]
            del self.outbound_edges[inbound_vertex][vertex]
        for outbound_vertex in self.outbound_edges[vertex]:
            del self.costs[(vertex, outbound_vertex)]
            del self.inbound_edges[outbound_vertex][vertex]
        del self.inbound_edges[vertex]
        del self.outbound_edges[vertex]
```

Method used to add an edge between two vertices in the graph. It takes three parameters:

- source: Represents the source vertex of the edge.
- destination: Represents the destination vertex of the edge.
- cost: Represents the cost associated with the edge.

It first checks if the edge already exists in the graph by calling the check_if_edge_exists method. If the edge already exists, it raises a GraphException.

It attempts to add both the source and destination vertices using the add_vertex method. If either vertex already exists, it catches the GraphException raised by add_vertex and continues.

It adds the destination vertex to the outbound edges of the source vertex and adds the source vertex to the inbound edges of the destination vertex. It assigns the specified cost to the edge by updating the costs dictionary with the tuple (source, destination) as the key, increments the number_of_edges.

```
def add_edge(self, source, destination, cost):
    """
    Adds an edge to the graph
    :param source: the source vertex
    :param destination: the destination vertex
    :param cost: the cost of the edge
    :return: None
    """
    if self.check_if_edge_exists(source, destination):
        raise GraphException("The edge already exists")
    try:
        self.add_vertex(source)
    except GraphException:
        pass
    try:
        self.add_vertex(destination)
    except GraphException:
        pass

    if source not in self.outbound_edges:
        self.outbound_edges[source] = []
    if destination not in self.inbound_edges:
        self.inbound_edges[destination] = []

    self.outbound_edges[source].append(destination)
    self.inbound_edges[destination].append(source)
    self.costs[(source, destination)] = cost
    self.number_of_edges += 1
```

Method used to remove an edge from the graph. It takes two parameters:

- source: Represents the source vertex of the edge.
- destination: Represents the destination vertex of the edge.

It first checks if the specified edge exists in the graph by calling the check_if_edge_exists method. If the edge does not exist, it raises a GraphException.

It deletes the cost associated with the edge from the costs dictionary. It removes the destination vertex from the outbound edges of the source vertex and removes the source vertex from the inbound edges of the destination vertex, decrements the number_of_edges.

```
def remove_edge(self, source, destination):
    """
    Removes an edge from the graph
    :param source: the source vertex
    :param destination: the destination vertex
    :return: None
    """
    if self.check_if_edge_exists(source, destination):
        del self.costs[(source, destination)]
        self.outbound_edges[source].remove(destination)
        self.inbound_edges[destination].remove(source)
        self.number_of_edges -= 1
        return
    raise GraphException("The edge does not exist")
```

Method used to create a copy of the graph.

It creates a new instance of the DirectedGraph class named graph_copy with the same number of vertices as the original graph. It then deep copies the inbound_edges, outbound_edges, and costs dictionaries from the original graph to the graph_copy.

It returns a new DirectedGraph object that is a copy of the original graph.

```
def get_copy_of_graph(self):
    """
    Creates a copy of the graph
    :return: a copy of the graph
    """
    graph_copy = DirectedGraph(self.number_of_vertices)
    graph_copy.inbound_edges = copy.deepcopy(self.inbound_edges)
    graph_copy.outbound_edges =
copy.deepcopy(self.outbound_edges)
    graph_copy.costs = copy.deepcopy(self.costs)
    return graph_copy
```

Method used to generate a random graph with a specified number of vertices and edges. It takes two parameters:

- number_of_vertices: Represents the desired number of vertices in the random graph.
- number_of_edges: Represents the desired number of edges in the random graph.

It first checks whether the specified number of edges is greater than the square of the number of vertices. If so, it raises a GraphException indicating that the input is invalid. It also checks whether the current graph already contains vertices. If so, it raises a GraphException indicating that the graph already exists.

If the input passes the validation checks, it proceeds to generate the random graph. It iterates over the range of number_of_vertices and adds each vertex to the graph using the add_vertex method. For each edge, it randomly selects a source and destination vertex from the possible_edges, as well as a random cost, and adds the edge using the add_edge method.

```
def create_random_graph(self, number_of_vertices,
number_of_edges):
    """
    Creates a random graph with a given number of vertices and
edges
    :param number_of_vertices: the number of vertices
    :param number_of_edges: the number of edges
    :return: None
    """
    if number_of_edges > number_of_vertices *
(number_of_vertices - 1):
        raise GraphException("Invalid input! The number of edges
must be less than the number of vertices * (number of vertices -
1)")
    if self.number_of_vertices != 0:
        raise GraphException("The graph already exists.")

    for i in range(number_of_vertices):
        self.add_vertex(i)

    possible_edges = [(i, j) for i in range(number_of_vertices)
for j in range(number_of_vertices) if i != j]
    num_edges = number_of_edges
    for source, destination in possible_edges:
        cost = randint(1, 100)
        self.add_edge(source, destination, cost)
        num_edges -= 1
        if num_edges == 0:
            break
```

Method used to read a graph from a file. It takes one parameter:

- file_name: Represents the name of the file to read the graph from

It opens the specified file in read mode ('r') using a with statement, ensuring that the file is properly closed after reading. It initializes the attributes number_of_edges, number_of_vertices, inbound_edges, outbound_edges, and costs to empty dictionaries and resets the counts of edges and vertices. It reads the first line of the file, which contains the number of vertices and edges. It converts the read values to integers and assigns them to the variables number_of_vertices and number_of_edges. It iterates over the range of number_of_vertices, adding each vertex to the graph using the add_vertex method. It then iterates until the specified number of edges, reading each line from the file. For each line, it splits the line and extracts the source, destination, and cost values. It converts these values to integers and adds the corresponding edge to the graph using the add_edge method.

```
def read_graph_from_file(self, file_name):
    """
    Reads a graph from a file
    :param file_name: the name of the file to read from
    :return: None
    """
    with open(file_name, 'r') as file:
        self.number_of_edges = 0
        self.number_of_vertices = 0
        self.inbound_edges = {}
        self.outbound_edges = {}
        self.costs = {}

        number_of_vertices, number_of_edges =
file.readline().split()
        number_of_edges = int(number_of_edges)
        number_of_vertices = int(number_of_vertices)

        for vertex in range(number_of_vertices):
            self.add_vertex(vertex)

        while number_of_edges:
            line = file.readline().split()
            source, destination, cost = line[0], line[1],
line[2]
            self.add_edge(int(source), int(destination),
int(cost))
            number_of_edges -= 1
```

Method used to write the graph to a file. It takes one parameter:

- file_name: Represents the name of the file to write the graph to.

It opens the specified file in write mode ('w'). It writes the number of vertices and edges of the graph on the first line of the file in the format: {number_of_vertices} {number_of_edges}. It then iterates over each vertex in the graph's outbound_edges dictionary and writes each outbound edge along with its associated cost to the file. Each edge is written as a separate line in the format: {source_vertex} {destination_vertex} {cost}.

```
def write_graph_to_file(self, file_name):
    """
    Writes the graph to a file
    :param file_name: the name of the file to write to
    :return: None
    """
    with open(file_name, 'w') as file:
        file.write(f"{self.number_of_vertices}
{self.number_of_edges}\n")
        for vertex, outbound_edges in
self.outbound_edges.items():
            for outbound_edge in outbound_edges:
                file.write(f"{vertex} {outbound_edge}
{self.costs[(vertex, outbound_edge)]}\n")
```