Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №3

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Унарные и бинарные операции над графами»

Выполнил:

студент группы 20ВВ2

Беляков А.М.

Проверили:

Митрохин М.А.

Юрова О.В.

Пенза 2021

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) две матрицы M 1 ,

М 2 смежности неориентированных помеченных графов G 1 , G 2 . Выведите

сгенерированные матрицы на экран.

2. \* Для указанных графов преобразуйте представление матриц

смежности в списки смежности. Выведите полученные списки на экран.

Задание 2

1. Для матричной формы представления графов выполните операцию:

а) отождествления вершин

б) стягивания ребра

в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

2. \* Для представления графов в виде списков смежности выполните

операцию:

а) отождествления вершин

б) стягивания ребра

в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

Задание 3

1. Для матричной формы представления графов выполните операцию:

а) объединения G = G 1 и G 2

б) пересечения G = G 1 и G 2

в) кольцевой суммы G = G 1 и G 2

Результат выполнения операции выведите на экран.

Задание 4 \*

1. Для матричной формы представления графов выполните операцию

декартова произведения графов G = G 1 X G2.

Результат выполнения операции выведите на экран.

Задание 1:   
Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

Листинг:

from random import randint  
from Graph import Graph  
  
graph = Graph(name='V')  
graph2 = Graph(name='X')  
  
for i in range(5):  
 graph.add\_node(i)  
for i in range(5):  
 graph2.add\_node(i)  
  
for i in range(5):  
 graph.connect(randint(0, 4), randint(0, 4))  
for i in range(5):  
 graph2.connect(randint(0, 4), randint(0, 4))  
  
graph.print()  
graph2.print()

Задание 2:

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание  
Листинг:

from Graph import Graph  
  
graph = Graph(name='V')  
  
for i in range(5):  
 graph.add\_node(i)  
  
graph.connect(0, 4)  
graph.connect(1, 3)  
graph.connect(2, 4)  
graph.connect(3, 4)  
graph.connect(0, 2)  
graph.connect(2, 3)  
  
graph.print()  
graph.node\_duplication(4)  
print("Дублирование вершины")  
graph.print()  
graph.rib\_retraction(2, 5)  
print("Стягивание ребра")  
graph.print()  
graph.node\_association(3, 4)  
print("Объединение вершин")  
graph.print()

Задание 3:

Объединение 2х графов

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

Листинг:

from Graph import Graph  
  
graph1 = Graph(name='V')  
graph2 = Graph(name='X')  
  
for i in range(5):  
 graph1.add\_node(i)  
  
for i in range(5):  
 graph2.add\_node(i)  
  
graph1.connect(0, 1)  
graph1.connect(2, 1)  
  
graph2.connect(3, 4)  
graph2.connect(3, 2)  
  
graph1.print()  
graph2.print()  
graph3 = graph1+graph2  
graph3.print()

Пересечение 2х графов

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

Листинг:

from Graph import Graph  
  
graph1 = Graph(name='V')  
graph2 = Graph(name='X')  
  
for i in range(5):  
 graph1.add\_node(i)  
  
for i in range(5):  
 graph2.add\_node(i)  
  
graph1.connect(0, 1)  
graph1.connect(2, 1)  
graph1.connect(3, 4)  
  
graph2.connect(3, 4)  
graph2.connect(3, 2)  
  
graph1.print()  
graph2.print()  
graph3 = graph1.graph\_crossing(graph2)  
graph3.print()

Кольцевая сумма 2х графов

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

Листинг:

from Graph import Graph  
  
graph1 = Graph(name='V')  
graph2 = Graph(name='X')  
  
for i in range(5):  
 graph1.add\_node(i)  
  
for i in range(5):  
 graph2.add\_node(i)  
  
graph1.connect(0, 1)  
graph1.connect(2, 1)  
  
  
graph2.connect(3, 4)  
graph2.connect(3, 2)  
  
graph1.print()  
graph2.print()  
graph3 = graph1.annular\_sum(graph2)  
graph3.print()

Задание 4:

Декартово произведение графов

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание  
Листинг:

from Graph import Graph  
  
graph1 = Graph(name='V')  
graph2 = Graph(name='X')  
  
for i in range(2):  
 graph1.add\_node(i)  
  
for i in range(3):  
 graph2.add\_node(i)  
  
graph1.connect(0, 1)  
  
graph2.connect(0, 1)  
graph2.connect(1, 2)  
  
graph1.print()  
graph2.print()  
graph3 = graph1\*graph2  
graph3.print()

Вывод: во время работы над данной лабораторной работой мы научились работать с графами и создали несколько основных действий над графами.

Листинг:

Graph.py  
import Exceptions\_Graph  
import Exceptions\_Matrix  
import Exceptions\_list  
import Node  
import Matrix  
import AdjacencyList  
  
  
class Graph():  
 def \_\_init\_\_(self, name, nodes=None):  
 self.name = str(name)  
 self.\_last\_element = 0  
 self.\_nodes = list()  
 self.\_matrix = Matrix.Matrix()  
 self.\_adjacency\_list = AdjacencyList.AdjacencyList()  
 if nodes is not None:  
 if isinstance(nodes, list):  
 for i in nodes:  
 self.add\_node(i)  
 else:  
 raise Exceptions\_Graph.GraphError  
  
 @property  
 def matrix(self):  
 return self.\_matrix  
  
 @matrix.setter  
 def matrix(self, new):  
 if isinstance(new, Matrix.Matrix):  
 self.\_matrix = new  
 self.\_adjacency\_list = self.matrix.convert\_to\_adj\_list()  
 else:  
 raise Exceptions\_Matrix.MatrixError  
  
 @property  
 def adj\_list(self):  
 return self.\_adjacency\_list  
  
 @adj\_list.setter  
 def adj\_list(self, new):  
 if isinstance(new, AdjacencyList.AdjacencyList):  
 self.\_adjacency\_list = new  
 self.\_matrix = self.\_adjacency\_list.convert\_to\_matrix()  
 else:  
 raise Exceptions\_list.AdjacencyListError  
  
 @property  
 def nodes(self):  
 return self.\_nodes  
  
 def get\_index\_on\_data(self, data, default=None):  
 for i in self.\_nodes:  
 if i == data:  
 return i.index  
 return default  
  
 def add\_node(self, data, name=None):  
 if not (self.\_check\_node\_in\_graph(data)):  
 if name is None:  
 name = self.name + str(self.\_last\_element)  
 node = Node.Node(data=data, name=name, index=self.\_last\_element)  
 self.\_nodes.append(node)  
 self.\_last\_element += 1  
 self.\_matrix.add\_node(self.\_nodes[-1])  
 self.\_adjacency\_list.add\_node(self.\_nodes[-1])  
 return node.index  
 else:  
 raise Exceptions\_Graph.NodeAlreadyIncluded  
  
 def \_check\_node\_in\_graph(self, data):  
 for i in [i.data for i in self.\_nodes]:  
 try:  
 if data == i:  
 return True  
 except TypeError:  
 return False  
 return False  
  
 def del\_node(self, node):  
 if node.data in [i.data for i in self.\_nodes]:  
 self.\_matrix.del\_node(self.\_nodes[node.index])  
 self.\_adjacency\_list.del\_node(self.\_nodes[node.index])  
 self.\_nodes.pop(node.index)  
 self.\_index\_update()  
 else:  
 raise Exceptions\_Graph.NodeNotIncludedError  
  
 def \_index\_update(self):  
 for i in range(len(self.\_nodes)):  
 self.\_nodes[i].index = i  
  
 def print(self):  
 print('Граф =', self.name)  
 print("Матрица смежности")  
 self.\_matrix.print()  
 print("Список смежности")  
 self.\_adjacency\_list.print()  
 print("Список нод")  
 print([i.name for i in self.\_nodes])  
  
 def connect(self, from\_node, to\_node, weight=1):  
 self.connect\_dir(from\_node, to\_node, weight)  
 self.connect\_dir(to\_node, from\_node, weight)  
  
 def connect\_dir(self, from\_node, to\_node, weight=1):  
 from\_node, to\_node = self.\_node\_review(from\_node, to\_node)  
 self.\_matrix.connect\_dir(from\_node, to\_node, weight)  
 self.\_adjacency\_list.connect\_dir(from\_node, to\_node, weight)  
  
 def disconnect(self, from\_node, to\_node, weight=1):  
 self.disconnect\_dir(from\_node, to\_node, weight)  
 self.disconnect\_dir(to\_node, from\_node, weight)  
  
 def disconnect\_dir(self, from\_node, to\_node, weight\_input=1):  
 from\_node, to\_node = self.\_node\_review(from\_node, to\_node)  
 try:  
 self.\_matrix.disconnect\_dir(from\_node, to\_node, weight\_input)  
 self.\_adjacency\_list.disconnect\_dir(from\_node, to\_node, weight\_input)  
 except ValueError:  
 raise Exceptions\_Graph.RibNotIncludedError  
  
 def convert\_matrix\_to\_adj\_list(self):  
 return self.\_matrix.convert\_to\_adj\_list()  
  
 def convert\_adj\_list\_to\_matrix(self):  
 return self.\_adjacency\_list.convert\_to\_matrix()  
  
 def node\_association(self, from\_node, to\_node):  
 from\_node, to\_node = self.\_node\_review(from\_node, to\_node)  
 self.\_matrix.node\_association(from\_node, to\_node)  
 self.\_adjacency\_list.node\_association(from\_node, to\_node)  
 self.del\_node(from\_node)  
  
 def rib\_retraction(self, from\_node, to\_node, weight=1):  
 from\_node, to\_node = self.\_node\_review(from\_node, to\_node)  
 try:  
 self.\_matrix.rib\_retraction(from\_node, to\_node, weight)  
 self.\_adjacency\_list.rib\_retraction(from\_node, to\_node, weight)  
 except Exceptions\_Graph.RibNotIncludedError:  
 raise Exceptions\_Graph.RibNotIncludedError  
 self.del\_node(from\_node)  
  
 def \_node\_review(self, \*nodes):  
 array = []  
 for node in nodes:  
 if node is None:  
 raise Exceptions\_Graph.NodeNotIncludedError  
 else:  
 array.append(self.\_nodes[node])  
 return array  
  
 def node\_duplication(self, from\_node):  
 from\_node = self.\_node\_review(from\_node)  
 self.add\_node(str(from\_node[0].data) + 'd', name=from\_node[0].name + 'd')  
 self.\_matrix.node\_duplication(from\_node[0], self.\_nodes[-1])  
 self.\_adjacency\_list.node\_duplication(from\_node[0], self.\_nodes[-1])  
  
 def \_\_add\_\_(self, other):  
 if not isinstance(other, Graph):  
 raise Exceptions\_Graph.GraphError  
 to\_ret = Graph(name=self.name + '+' + other.name)  
 links = dict()  
 for i in self.\_nodes:  
 links[i] = to\_ret.add\_node(i.data, name=i.name)  
  
 for i in other.nodes:  
 if i in [j for j in to\_ret.nodes]:  
 for j in to\_ret.nodes:  
 if j == i:  
 links[i] = j.node\_name\_update(i)  
 break  
 else:  
 links[i] = to\_ret.add\_node(i.data, name=i.name)  
  
 to\_ret.matrix.plus(self.\_matrix, other.matrix, links)  
 to\_ret.adj\_list = to\_ret.matrix.convert\_to\_adj\_list()  
 return to\_ret  
  
 def graph\_crossing(self, other):  
 if not isinstance(other, Graph):  
 raise Exceptions\_Graph.GraphError  
 to\_ret = Graph(name=self.name + '+' + other.name)  
 links = dict()  
 for i in self.\_nodes:  
 links[i] = to\_ret.add\_node(i.data, name=i.name)  
 for i in other.nodes:  
 if i in [j for j in to\_ret.nodes]:  
 for j in to\_ret.nodes:  
 if j == i:  
 links[j.node\_name\_update(i)] = i  
 break  
 to\_ret.matrix.matrix\_crossing(self.matrix, other.matrix, links)  
 to\_ret.adj\_list = to\_ret.matrix.convert\_to\_adj\_list()  
 return to\_ret  
  
 def annular\_sum(self, other):  
 if not isinstance(other, Graph):  
 raise Exceptions\_Graph.GraphError  
 to\_ret = Graph(name=self.name + '+' + other.name)  
 links = dict()  
 for i in self.\_nodes:  
 links[i] = to\_ret.add\_node(i.data, name=i.name)  
 for i in other.nodes:  
 if i in [j for j in to\_ret.nodes]:  
 for j in to\_ret.nodes:  
 if j == i:  
 links[j.node\_name\_update(i)] = i  
 break  
 else:  
 links[i] = to\_ret.add\_node(i.data, name=i.name)  
  
 to\_ret.matrix.annular\_sum(self.matrix, other.matrix, links)  
  
 i = 0  
 while i < len(to\_ret.matrix.matrix):  
 row = [to\_ret.matrix.matrix[f][i] for f in range(len(to\_ret.matrix.matrix))]  
 if to\_ret.\_matrix.matrix[i] == [[] for \_ in range(len(to\_ret.matrix.matrix))] and row == [[] for \_ in range(  
 len(to\_ret.matrix.matrix))]:  
 to\_ret.del\_node(to\_ret.\_nodes[i])  
 else:  
 i += 1  
 to\_ret.adj\_list = to\_ret.matrix.convert\_to\_adj\_list()  
 return to\_ret  
  
 def graph\_rand\_gen(self, long, connects):  
 for i in range(long):  
 self.add\_node(i)  
 for \_ in range(connects):  
 self.connect(randint(0, long - 1), randint(0, long - 1))

def \_\_mul\_\_(self, other):  
 if not isinstance(other, Graph):  
 raise Exceptions\_Graph.GraphError  
 to\_ret = Graph(name=self.name + '+' + other.name)  
 links = dict()  
 for node in self.\_nodes:  
 for node2 in other.nodes:  
 links[to\_ret.add\_node(node.data + node2.data, name=node.name + node2.name)] = [node, node2]  
 to\_ret.matrix.cartesian\_product(self.\_matrix, other.matrix, links)  
 to\_ret.\_adjacency\_list = to\_ret.\_matrix.convert\_to\_adj\_list()  
 return to\_ret

Matrix.py

import Exceptions\_Matrix  
import Exceptions\_Graph  
import AdjacencyList  
  
  
class Matrix():  
 def \_\_init\_\_(self):  
 self.\_nodes = []  
 self.\_matrix = []  
  
 @property  
 def nodes(self):  
 return self.\_nodes  
  
 @property  
 def matrix(self):  
 return self.\_matrix  
  
 def add\_node(self, node):  
 self.\_nodes.append(node)  
 self.\_matrix\_update(1)  
  
 def \_matrix\_update(self, add\_nodes):  
 for i in self.\_matrix:  
 for \_ in range(add\_nodes):  
 i.append([])  
 for i in range(add\_nodes):  
 self.\_matrix.append([[] for \_ in range(len(self.\_nodes))])  
  
 def del\_node(self, node):  
 self.\_nodes.pop(node.index)  
 self.\_matrix.pop(node.index)  
 for i in self.\_matrix:  
 i.pop(node.index)  
  
 def print(self):  
 for node in self.\_nodes:  
 print(" " + node.name, end="")  
 print()  
 for i in range(len(self.\_matrix)):  
 print(str(self.\_nodes[i].name) + " " + str(self.\_matrix[i])[1:-1])  
  
 def connect\_dir(self, from\_node, to\_node, weight=1):  
 self.\_matrix[from\_node.index][to\_node.index].append(weight)  
  
 def disconnect\_dir(self, from\_node, to\_node, weight=1):  
 self.\_matrix[from\_node.index][to\_node.index].remove(weight)  
  
 def convert\_to\_adj\_list(self):  
 adj\_list = AdjacencyList.AdjacencyList()  
 for i in self.\_nodes:  
 adj\_list.add\_node(i)  
 for i in range(len(self.\_matrix)):  
 for j in range(len(self.\_matrix[i])):  
 for k in self.\_matrix[i][j]:  
 adj\_list.connect\_dir(self.\_nodes[i], self.\_nodes[j], k)  
 return adj\_list  
  
 def node\_association(self, from\_node, to\_node):  
 for i in range(len(self.\_matrix)):  
 for k in self.\_matrix[i][from\_node.index]:  
 self.connect\_dir(to\_node, self.\_nodes[i], k)  
 self.disconnect\_dir(self.\_nodes[i], from\_node, k)  
  
 for j in range(len(self.\_matrix[from\_node.index])):  
 for k in self.\_matrix[from\_node.index][j]:  
 self.connect\_dir(self.\_nodes[j], to\_node, k)  
 self.disconnect\_dir(from\_node, self.\_nodes[j], k)  
  
 def rib\_retraction(self, from\_node, to\_node, weight):  
 if len(self.\_matrix[from\_node.index][to\_node.index]) > 0:  
 self.node\_association(from\_node, to\_node)  
 self.disconnect\_dir(to\_node, to\_node, weight)  
 else:  
 raise Exceptions\_Graph.RibNotIncludedError  
  
 def node\_duplication(self, from\_node, node):  
 for i in range(len(self.\_matrix[from\_node.index])):  
 for j in self.\_matrix[from\_node.index][i]:  
 self.connect\_dir(node, self.\_nodes[i], j)  
 for i in range(len(self.\_matrix)):  
 for j in self.\_matrix[i][from\_node.index]:  
 self.connect\_dir(self.\_nodes[i], node, j)  
  
 def plus(self, first\_matrix, second\_matrix, links):  
 for i in range(len(first\_matrix.matrix)):  
 for j in range(len(first\_matrix.matrix[i])):  
 for k in first\_matrix.matrix[i][j]:  
 self.connect\_dir(self.\_nodes[links[first\_matrix.nodes[i]]],  
 self.\_nodes[links[first\_matrix.nodes[j]]], k)  
  
 for i in range(len(second\_matrix.matrix)):  
 for j in range(len(second\_matrix.matrix[i])):  
 for k in second\_matrix.matrix[i][j]:  
 self.connect\_dir(self.nodes[links[second\_matrix.nodes[i]]],  
 self.\_nodes[links[second\_matrix.nodes[j]]], k)  
  
 def matrix\_crossing(self, first\_matrix, second\_matrix, links):  
 for i in range(len(self.\_matrix)):  
 for j in range(len(self.\_matrix[i])):  
 first\_array = first\_matrix.matrix[links[i].index][links[j].index]  
 second\_array = second\_matrix.matrix[links[i].index][links[j].index]  
 for k in first\_array:  
 for g in second\_array:  
 if k == g:  
 self.connect\_dir(self.\_nodes[i], self.\_nodes[j], k)  
  
 def annular\_sum(self, first\_matrix, second\_matrix, links):  
 for i in range(len(self.\_matrix)):  
 for j in range(len(self.\_matrix[i])):  
 first\_array = first\_matrix.matrix[links[i].index][links[j].index]  
 second\_array = second\_matrix.matrix[links[i].index][links[j].index]  
 for k in first\_array:  
 if k not in [g for g in second\_array]:  
 self.connect\_dir(self.\_nodes[i], self.\_nodes[j], k)  
 for k in second\_array:  
 if k not in [g for g in first\_array]:  
 self.connect\_dir(self.\_nodes[i], self.\_nodes[j], k)  
  
 def cartesian\_product(self, first\_matrix, second\_matrix, links):  
 for i in range(len(self.\_matrix)):  
 for j in range(len(self.\_matrix[i])):  
 if links[i][0] == links[j][0] and len(second\_matrix.matrix[links[i][1].index][links[j][1].index]) > 0:  
 self.\_matrix[i][j] = (second\_matrix.matrix[links[i][1].index][links[j][1].index])  
 elif links[i][1] == links[j][1] and len(first\_matrix.matrix[links[i][0].index][links[j][0].index]) > 0:  
 self.\_matrix[i][j] = (first\_matrix.matrix[links[i][0].index][links[j][0].index])

AdjacencyList.py

import Exceptions\_list  
import Exceptions\_Graph  
import Matrix  
  
  
class AdjacencyList():  
 def \_\_init\_\_(self):  
 self.\_adj\_list = dict()  
  
 @property  
 def adj\_list(self):  
 return self.\_adj\_list  
  
 def add\_node(self, node):  
 self.\_adj\_list[node] = list()  
  
 def del\_node(self, node):  
 self.\_adj\_list.pop(node)  
 for i in self.\_adj\_list.keys():  
 count = 0  
 for j in self.\_adj\_list[i]:  
 if j == node:  
 count += 1  
 for \_ in range(count):  
 self.\_adj\_list[i].pop(node)  
  
 def print(self):  
 for i in self.\_adj\_list.keys():  
 print(str(i.name) + ":", end="")  
 for j in self.\_adj\_list[i]:  
 print(str(j[0].name + "[" + str(j[1]) + "]") + ", ", end="")  
 print()  
  
 def connect\_dir(self, from\_node, to\_node, weight=1):  
 self.\_adj\_list[from\_node].append([to\_node, weight])  
  
 def disconnect\_dir(self, from\_node, to\_node, weight=1):  
 self.\_adj\_list[from\_node].remove([to\_node, weight])  
  
 def convert\_to\_matrix(self):  
 matrix = Matrix.Matrix()  
 for i in self.\_adj\_list.keys():  
 matrix.add\_node(i)  
 for i in self.\_adj\_list.keys():  
 for j in self.\_adj\_list[i]:  
 matrix.connect\_dir(i, j[0], j[1])  
 return matrix  
  
 def node\_association(self, from\_node, to\_node):  
 for i in self.\_adj\_list.keys():  
 local\_copy = self.\_adj\_list[i][:]  
 for j in local\_copy:  
 if j[0] == from\_node:  
 self.connect\_dir(i, to\_node, j[1])  
 self.disconnect\_dir(i, from\_node, j[1])  
 local\_copy = self.\_adj\_list[from\_node][:]  
 for i in local\_copy:  
 self.connect\_dir(to\_node, i[0], i[1])  
 self.disconnect\_dir(from\_node, i[0], i[1])  
  
 def rib\_retraction(self, from\_node, to\_node, weight):  
 if from\_node in [i[0] for i in self.\_adj\_list[to\_node]] or to\_node in [i[0] for i in self.\_adj\_list[from\_node]]:  
 self.node\_association(from\_node, to\_node)  
 self.disconnect\_dir(to\_node, to\_node, weight)  
 else:  
 raise Exceptions\_Graph.RibNotIncludedError  
  
 def node\_duplication(self, from\_node, node):  
 for i in self.\_adj\_list.keys():  
 for j in self.\_adj\_list[i]:  
 if j[0] == from\_node:  
 self.connect\_dir(i, node, j[1])  
  
 for i in self.\_adj\_list[from\_node]:  
 self.connect\_dir(node, i[0], i[1])

Node.py

import Exceptions\_Node  
  
  
class Node():  
 def \_\_init\_\_(self, name=None, data=None, index=None):  
 self.\_name = name  
 self.\_data = str(data)  
 self.\_index = index  
 self.\_flag = []  
  
 @property  
 def name(self):  
 return self.\_name  
  
 @name.setter  
 def name(self, new\_name):  
 self.\_name = new\_name  
  
 @property  
 def data(self):  
 return str(self.\_data)  
  
 @data.setter  
 def data(self, new\_data):  
 self.\_data = str(new\_data)  
  
 @property  
 def index(self):  
 return self.\_index  
  
 @index.setter  
 def index(self, new\_index):  
 self.\_index = new\_index  
  
 @property  
 def flag(self):  
 return self.\_flag  
  
 def node\_name\_update(self, node):  
 self.\_name = self.\_name + node.name  
 return self.\_index  
  
 def node\_data\_update(self, node):  
 self.\_data = self.data + node.data  
  
 def \_\_eq\_\_(self, other):  
 if isinstance(other, Node):  
 return self.\_equal(other.data)  
 else:  
 return self.\_equal(other)  
  
 def \_equal(self, other):  
 try:  
 if self.data == other:  
 return True  
 else:  
 return False  
 except TypeError:  
 return False  
  
 def \_\_hash\_\_(self):  
 return hash(self.data)  
  
 def add\_flag(self, flag):  
 self.\_flag.append(flag)  
  
 def remove\_flag(self, flag):  
 self.\_flag.remove(flag)

Exceptions\_Graph.py

class GraphError(Exception):  
 def \_\_str\_\_(self):  
 return "Impossible action"  
  
  
class RibNotIncludedError(GraphError):  
 def \_\_str\_\_(self):  
 return "No rib"  
  
  
class NodeNotIncluded(GraphError):  
 def \_\_str\_\_(self):  
 return "Node not included"  
  
  
class NodeAlreadyIncluded(GraphError):  
 def \_\_str\_\_(self):  
 return "Node already included"  
  
  
class NodeNotIncludedError(GraphError):  
 def \_\_str\_\_(self):  
 return "Node not included"  
  
  
class NodeError(GraphError):  
 def \_\_str\_\_(self):  
 return "This is not a node"

Exceptions\_Matrix.py

class MatrixError(Exception):  
 def \_\_str\_\_(self):  
 return "This is not a matrix"

Exceptions\_list.py

class AdjacencyListError(Exception):  
 def \_\_str\_\_(self):  
 return "This is not a adjacency list"