|  |  |
| --- | --- |
| |  | | --- | | МИНОБРНАУКИ РОССИИ  Федеральное государственное  бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | |

Институт Информационных технологий

Кафедра корпоративных информационных систем

**ЛАБОРАТОРНАЯ РАБОТА**

по дисциплине «Структуры и алгоритмы обработки данных»

**Тема лабораторной работы** Теория графов.

**Студент группы** ИКБО-12-17 Лисовой А.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись студента)*

**Руководитель работы** Андрианова Е.Г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись руководителя)*

Работа представлена к защите «29» марта 2019 г.

Допущен к защите «\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2019 г.

Москва, 2019

**Оглавление**

[Постановка задачи 2](#_Toc2375733)

[Теоретическое введение 3](#_Toc2375734)

[Описание алгоритма решения задачи 4](#_Toc2375735)

[Тестирование программы 6](#_Toc2375736)

[Кодирование алгоритма программы 8](#_Toc2375737)

# Постановка задачи

1. Проверить, будет ли граф являться деревом, если из него удалить N вершин, указанных пользователем.
2. Посчитать количество путей между всеми вершинами ориентированного графа.
3. Определить, содержит ли граф Эйлеров цикл, и, если содержит, указать его.

# Теоретическое введение

**Граф:**

Граф — абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин.

**Дерево:**

Дерево - связный ациклический граф с не ориентированными и невзвешенными ребрами рёбрами. Дерево состоит из узлов, один из которых обязательно является корневым – то есть самым верхним узлом дерева. Внутренним узлом является любой узел дерева, имеющий потомков. Соответственно листом называется узел, потомков лишенный.

**Лес:**

Лес — упорядоченное множество упорядоченных деревьев.

**Путь:**

Путь - это конечная последовательность вершин графа, в которой каждые две вершины идущие подряд соединены ребром. Путь может быть ориентированным или неориентированным в зависимости от графа.

**Эйлеров путь:**

Эйлеров путь (Эйлерова цепь) в графе — это путь, проходящий по всем рёбрам графа и притом только по одному разу.

**Эйлеров цикл:**

Эйлеров цикл — Эйлеров путь, являющийся циклом. То есть замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

**Степень вершины:**

Степень вершины графа – это количество ребер, связывающих вершину графа с соседними. Степень вершины может быть входящая и исходящая (для неориентированных графов входящая степень равна исходящей):

# Описание алгоритмов решения задачи

**Задание 1:**

1. Проверить, является ли исходный граф деревом, если не является – продолжить.

2. Последовательно удалить из графа указанные пользователем вершины.

3. Применить алгоритм поиску в глубину для выявления циклов в оставшемся графе:

3.1 Выбираем вершину, входящая степень которой равна нулю, после чего начинаем обход в глубину.

3.2 После того, как вершина обработана, заносим ее в стек.

3.3 По окончании обхода в глубину вершины достаются из стека. Новые номера присваиваются в порядке вытаскивания из стека.

3.4 Во время обхода в глубину для обозначения состояния вершин используется 3 цвета. Изначально все вершины белые. В тот момент, когда мы впервые посещаем вершину, красим ее в серый цвет.

3.5 Когда просмотрен список всех смежных с ней вершин, то есть, из неё больше невозможно никуда пройти дальше, красим ее в черный цвет.

3.6 Если же из вершины можно снова пройти в смежную с ней вершину, это означает, что граф содержит цикл, и, следовательно, деревом не является.

4. Если граф стал деревом – сообщить об этом и отобразить получившийся граф, иначе – сообщить о том, что граф так и не стал деревом.

**Задание 2:**

1. С помощью вложенного цикла начать перебор всех вершин графа, вызывая для каждой пары функцию поиска пути:

1.1 Используя обход в ширину внутри цикла найти пути между данной парой вершин графа

1.2 Если получен путь, и его нет в списке путей между данными вершинами графа, то поместить путь в список всех путей между данными вершинами графа.

1.3 Повторять цикл, пока все возможные пути между данными вершинами графа не будут найдены.

2. После того, как итерация цикла для каждой пары вершин завершилась, получить список всех возможных путей

3. После того, как был получен список списков всех возможных путей между всеми парами вершин графа, то, с помощью цикла посчитать длину каждого входящего в него списка всех путей для каждой из пар вершин, это и количество всех возможных путей между всеми вершинами графа.

**Задание 3:**

1. Последовательно проходя по всем вершинам графа:

1.1 Пока для текущей вершины есть цикл, находим его и помещаем его вершины в массив циклов, сохраняя порядок обхода.

1.2 Идем по массиву циклов, рекурсивно вызывая функцию поиска циклов для каждой из вершин в массиве.

2. Если найден цикл, один раз проходящий через все ребра графа, вывести его. Иначе сообщить о том, что граф не содержит Эйлеров цикл.

# Тестирование программы

**Задание 1:**

Тест №1: Определить, будет ли граф являться деревом, после указанных пользователем вершин в указанном им же количестве.

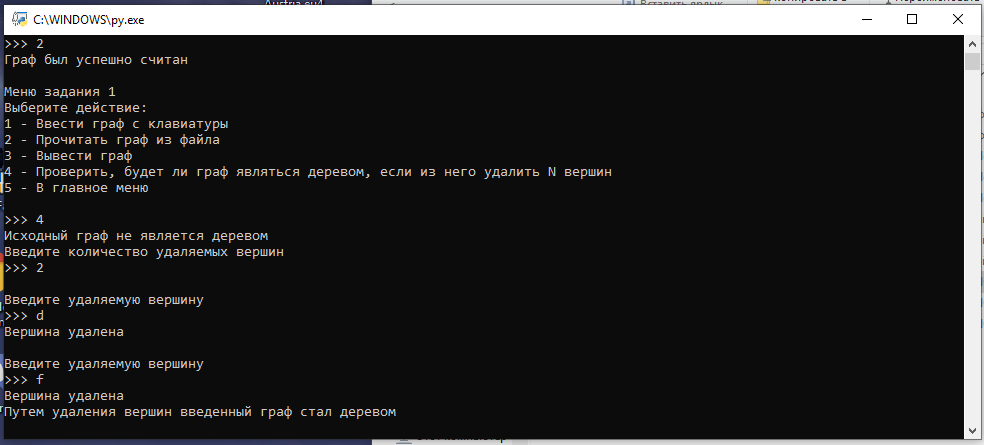


Рис. 1 Подтверждает корректную работу, так как после удаления указанных вершин граф перестал содержать циклы, и, следовательно, стал деревом.

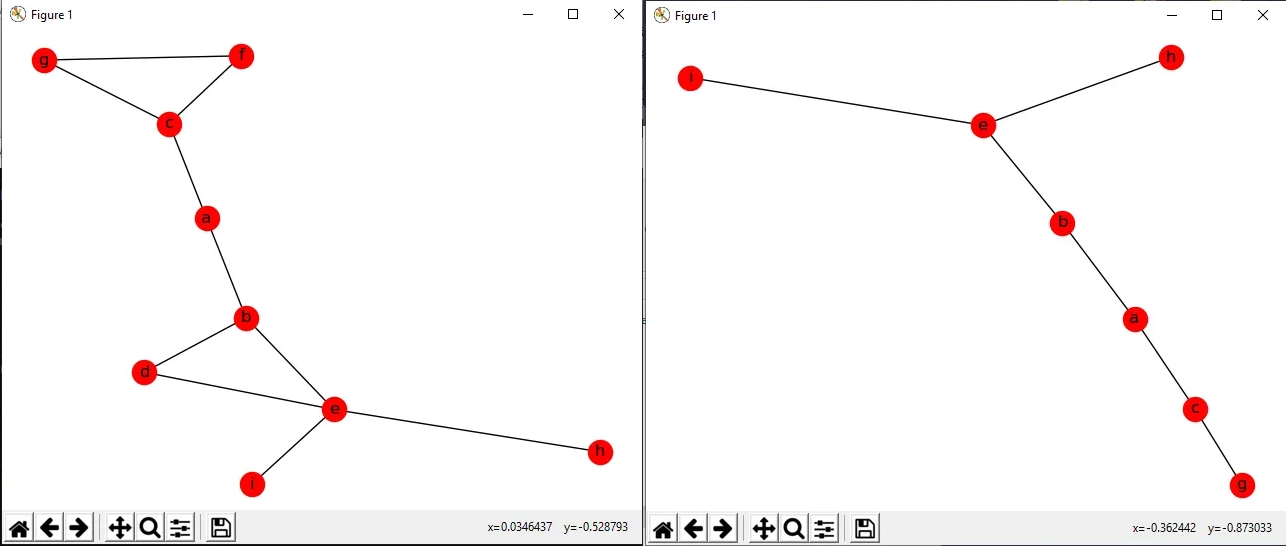


Рис. 2 Графическое отображение графа до и после удаления вершин в программе.

**Задание 2:**

Тест №2: Посчитать количество всевозможных путей между всеми вершинами графа.

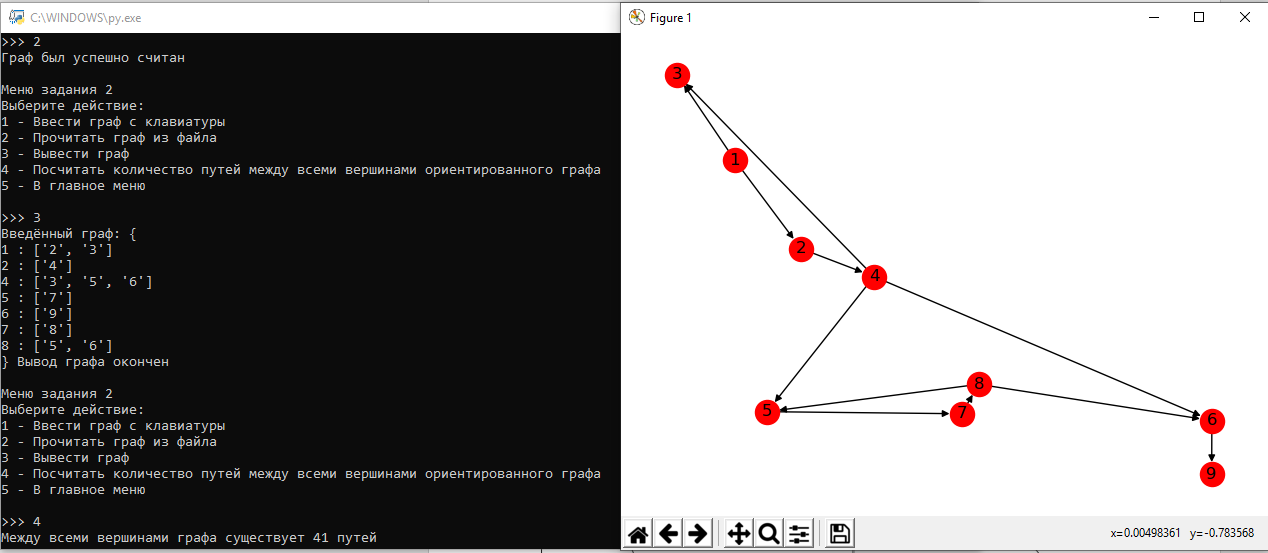


Рис. 3 Подтверждает корректную работу программы, было получено верное количество всех возможных путей графа.

**Задание 3:**

Тест №3: Поиск Эйлерова цикла в введенном графе.

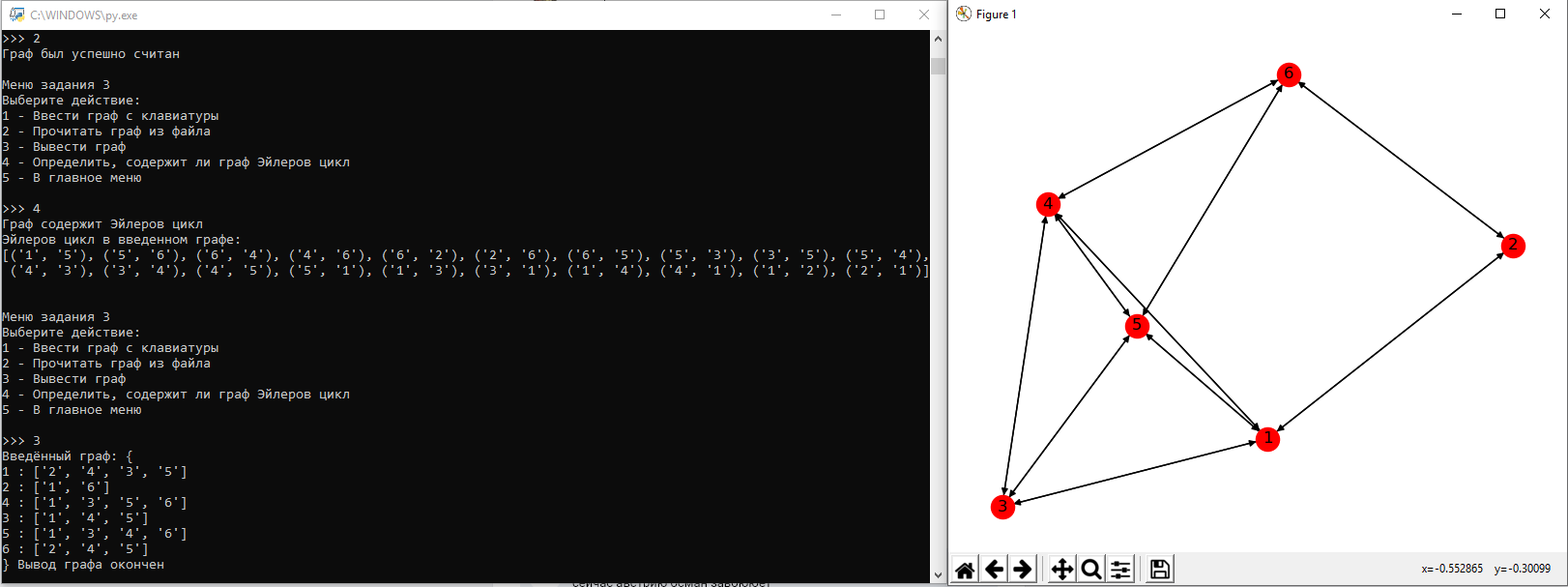


Рис. 4 Подтверждает корректную работу программы, так как так был получен Эйлеров цикл для содержащего его графа.

# Кодирование алгоритма программы

**Модуль main.py:**

# Список импортов

from Graph import Graph

from Interface import Interface

# Консольное меню программы

menu = Interface()

# Вызов метода отображения главного меню

menu.interface\_Main()

# Цикл обработки команд

while True:

# Ввод выбранного действия

Step = menu.input\_Controller()

# Задание 1

if Step == 1:

print("")

# Создание объекта класса Граф

graph = Graph()

menu.interface\_Task1()

while True:

Step = menu.input\_Controller()

# Ввести неоринетированный граф с клавиатуры

if Step == 1:

graph.input\_Undirected\_Graph()

#print(graph.undirected\_cycle\_Detector())

menu.interface\_Task1()

# Ввести граф из файла

elif Step == 2:

graph.file\_Undirected\_Graph("Task1.txt")

menu.interface\_Task1()

# Вывести граф

elif Step == 3:

graph.output\_Graph()

if graph.\_edges\_value > 0:

graph.nx\_Undirected\_Graph\_Draw()

menu.interface\_Task1()

# Удалить из графа N вершин и проверить его на древовидность

elif Step == 4:

graph.tree\_Detector()

menu.interface\_Task1()

# Выход в главное меню

elif Step == 5:

print("")

menu.interface\_Main()

break

# Остальные случаи

else:

print("Ошибка ввода")

print("")

# Задание 2

elif Step == 2:

print("")

# Создание объекта класса Граф

graph = Graph()

menu.interface\_Task2()

while True:

Step = menu.input\_Controller()

# Ввести ориентированный граф с клавиатуры

if Step == 1:

graph.input\_Directed\_Graph()

#print(graph.undirected\_cycle\_Detector())

menu.interface\_Task2()

# Ввести граф из файла

elif Step == 2:

graph.file\_Directed\_Graph("Task2.txt")

menu.interface\_Task2()

# Вывести граф

elif Step == 3:

graph.output\_Graph()

if graph.\_edges\_value > 0:

graph.nx\_Directed\_Graph\_Draw()

menu.interface\_Task2()

# Посчитать количество всех путей между вершинами графа

elif Step == 4:

graph.all\_Graph\_Paths()

menu.interface\_Task2()

# Выход в главное меню

elif Step == 5:

print("")

menu.interface\_Main()

break

# Остальные случаи

else:

print("Ошибка ввода")

print("")

# Задание 3

elif Step == 3:

print("")

# Создание объекта класса Граф

graph = Graph()

menu.interface\_Task3()

while True:

Step = menu.input\_Controller()

# Ввести ориентированный или неориентированный граф с

клавиатуры

if Step == 1:

graph.input\_Directed\_Graph()

#print(graph.undirected\_cycle\_Detector())

menu.interface\_Task3()

# Ввести граф из файла

elif Step == 2:

graph.file\_Undirected\_Graph("Task3.txt")

menu.interface\_Task3()

# Вывести граф

elif Step == 3:

graph.output\_Graph()

if graph.\_edges\_value > 0:

graph.nx\_Directed\_Graph\_Draw()

menu.interface\_Task3()

# Определить, содержит ли граф Гамильтонов цикл

elif Step == 4:

graph.eulerian\_Detector()

menu.interface\_Task3()

# Выход в главное меню

elif Step == 5:

print("")

menu.interface\_Main()

break

# Остальные случаи

else:

print("Ошибка ввода")

print("")

# Выход из программы

elif Step == 4:

print("Программа завершена")

break

# Остальные случаи

else:

print("Ошибка ввода")

print("")

**Модуль Interface.py:**

# Класс интерфейса

class Interface:

# Конструктор

def \_\_init\_\_(self):

pass

# Главное меню:

def interface\_Main(self):

print ("Главное меню:")

print ("Выберите действие:")

print ("1 - Первое задание")

print ("2 - Второе задание")

print ("3 - Третье задание")

print ("4 - Выход из программы.")

print("")

pass

# Задание 1:

def interface\_Task1(self):

print ("Меню задания 1")

print ("Выберите действие:")

print ("1 - Ввести граф с клавиатуры")

print ("2 - Прочитать граф из файла")

print ("3 - Вывести граф")

print ("4 - Проверить, будет ли граф являться деревом, если из

него удалить N вершин")

print ("5 - В главное меню")

print("")

pass

# Задание 2:

def interface\_Task2(self):

print ("Меню задания 2")

print ("Выберите действие:")

print ("1 - Ввести граф с клавиатуры")

print ("2 - Прочитать граф из файла")

print ("3 - Вывести граф")

print ("4 - Посчитать количество путей между всеми вершинами

ориентированного графа")

print ("5 - В главное меню")

print("")

pass

# Задание 3:

def interface\_Task3(self):

print ("Меню задания 3")

print ("Выберите действие:")

print ("1 - Ввести граф с клавиатуры")

print ("2 - Прочитать граф из файла")

print ("3 - Вывести граф")

print ("4 - Определить, содержит ли граф Эйлеров цикл")

print ("5 - В главное меню")

print("")

pass

# Защита от некорректного ввода:

def input\_Controller(self):

while True:

try:

Value = int(input(">>> "))

if Value > 0:

return Value

else:

print("Ошибка ввода!")

except ValueError:

print("Ошибка ввода!")

print("")

**Модуль Graph.py**

# Список импортов

import re

import networkx as nx

import matplotlib.pyplot as plt

import copy

# Класс графа

class Graph(object):

# Конструктор

def \_\_init\_\_(self, edges\_value = 0 , vertex\_value = 0):

# Граф строится на основе словаря

self.\_edges = {}

# Количество рёбер

self.\_edges\_value = edges\_value

# Метод заполнения направленного графа c клавиатуры

def input\_Directed\_Graph(self):

self.\_edges.clear()

print("Введите количество рёбер графа: ")

self.\_edges\_value = self.\_graph\_Edges\_Input\_Controller()

print("Последовательно введите ребра графа: ")

for i in range(self.\_edges\_value):

print("Идет ввод ", i + 1, " ребра графа: ")

current\_node = self.lexema\_Filter()

next\_node = self.lexema\_Filter()

current\_list = []

current\_list.append(next\_node)

if current\_node in self.\_edges:

if next\_node not in self.\_edges[current\_node]:

self.\_edges[current\_node].append(next\_node)

else:

self.\_edges[current\_node] = current\_list

print("Граф был успешно заполнен")

print("")

# Метод заполнения ненаправленного графа с клавиатуры

def input\_Undirected\_Graph(self):

self.\_edges.clear()

print("Введите количество рёбер графа: ")

self.\_edges\_value = self.\_graph\_Edges\_Input\_Controller()

print("Последовательно введите ребра графа: ")

for i in range(self.\_edges\_value):

print("Идет ввод ", i + 1, " ребра графа: ")

current\_node = self.lexema\_Filter()

next\_node = self.lexema\_Filter()

current\_list\_first = []

current\_list\_second = []

current\_list\_first.append(next\_node)

current\_list\_second.append(current\_node)

if current\_node in self.\_edges:

if next\_node not in self.\_edges[current\_node]:

self.\_edges[current\_node].append(next\_node)

else:

self.\_edges[current\_node] = current\_list\_first

if next\_node in self.\_edges:

if current\_node not in self.\_edges[next\_node]:

self.\_edges[next\_node].append(current\_node)

else:

self.\_edges[next\_node] = current\_list\_second

print("Граф был успешно заполнен")

print("")

# Метод заполнения неоринетированного графа из файла

def file\_Undirected\_Graph(self, name):

self.\_edges.clear()

file = open(name)

for line in file:

pattern = re.compile("\n")

match = pattern.search(line)

if match:

line = line[0:-1]

values = line.split(' ')

current\_node = values[0]

next\_node = values[1]

current\_list\_first = []

current\_list\_second = []

current\_list\_first.append(next\_node)

current\_list\_second.append(current\_node)

if current\_node in self.\_edges:

if next\_node not in self.\_edges[current\_node]:

self.\_edges[current\_node].append(next\_node)

else:

self.\_edges[current\_node] = current\_list\_first

if next\_node in self.\_edges:

if current\_node not in self.\_edges[next\_node]:

self.\_edges[next\_node].append(current\_node)

else:

self.\_edges[next\_node] = current\_list\_second

self.\_edges\_value += 1

print("Граф был успешно считан")

print("")

# Метод заполнения ориентированного графа из файла

def file\_Directed\_Graph(self, name):

self.\_edges.clear()

file = open(name)

for line in file:

pattern = re.compile("\n")

match = pattern.search(line)

if match:

line = line[0:-1]

values = line.split(' ')

current\_node = values[0]

next\_node = values[1]

current\_list\_first = []

current\_list\_second = []

current\_list\_first.append(next\_node)

current\_list\_second.append(current\_node)

if current\_node in self.\_edges:

if next\_node not in self.\_edges[current\_node]:

self.\_edges[current\_node].append(next\_node)

else:

self.\_edges[current\_node] = current\_list\_first

self.\_edges\_value += 1

print("Граф был успешно считан")

print("")

# Метод вывода графа

def output\_Graph(self):

if self.\_edges\_value > 0:

print("Введённый граф: {")

for key in self.\_edges:

print(key, ":", self.\_edges[key])

print("} Вывод графа окончен ")

print("")

else:

print("Граф не был введён ")

print("")

# Метод защиты от некорректного ввода чисел

def \_graph\_Edges\_Input\_Controller(self):

while True:

try:

value = int(input(">>> "))

if value > 0:

return value

else:

print("Ошибка ввода")

print("")

except ValueError:

print("Ошибка ввода")

print("")

# Метод фильтра лексем

def lexema\_Filter(self):

# Паттерн всего, что является лексемой

Pattern = r'[\.a-zA-Zа-яА-я0-9]+'

while True:

lexem = input(">>> ")

if re.search(Pattern, lexem):

return lexem

else:

print("Ошибка ввода")

print("")

# Метод удаления вершины из графа

def vertex\_Remove(self):

if self.\_edges is not None:

print("")

print("Введите удаляемую вершину")

remove\_vertex = self.lexema\_Filter()

if remove\_vertex in self.\_edges.keys():

self.\_edges.pop(remove\_vertex)

for vertex in self.\_edges:

if remove\_vertex in self.\_edges[vertex]:

self.\_edges[vertex].remove(remove\_vertex)

self.\_edges\_value -= 1

print("Вершина удалена")

else:

print("Введенной вершины не существует в графе")

else:

print("Граф пуст")

# Метод определния есть ли в графе любые циклы

def directed\_cycle\_Detector(self):

path = set()

visited = set()

# Рекурсивный обход графа

def visit(vertex):

if vertex in visited:

return False

visited.add(vertex)

path.add(vertex)

for neighbour in self.\_edges.get(vertex, ()):

if neighbour in path or visit(neighbour):

return True

path.remove(vertex)

return False

return any(visit(vertex) for vertex in self.\_edges)

# Метод проверки наличия циклов в неориентированном графе (т.е.

игнорируя циклы вида {a -> b, b -> a})

def undirected\_cycle\_Detector(self):

marker = {vertex : False for vertex in self.\_edges}

cycle = [False]

# Рекрусивный обход графа

def visit(vertex, cycle, pred\_node, marker):

if cycle[0]:

return

marker[vertex] = True

for jertex in self.\_edges[vertex]:

if marker[jertex] and jertex != pred\_node:

cycle[0] = True

return

if not marker[jertex]:

visit(jertex, cycle, vertex, marker)

# Вызов обхода графа

for vertex in self.\_edges:

if not marker[vertex]:

visit(vertex, cycle, vertex, marker)

if cycle[0]:

break

return cycle[0]

# Приведение графа из неориентированного вида к ориентированному

def utod(self):

graph = {}

for vertex in self.\_edges:

graph[vertex] = []

values = self.\_edges.get(vertex)

for value in values:

if value not in graph.keys():

graph[vertex].append(value)

if len(graph[vertex]) == 0:

graph.pop(vertex)

self.\_edges = graph

# Удалить из графа N вершин и проверить на древовидность

def tree\_Detector(self):

if self.\_edges\_value > 0:

if self.undirected\_cycle\_Detector():

print("Исходный граф не является деревом")

else:

print("Исходный граф уже является деревом, выполнение

задания невозможно")

return

print("Введите количество удаляемых вершин")

N = self.\_graph\_Edges\_Input\_Controller()

if N < self.\_edges\_value:

for index in range(N):

self.vertex\_Remove()

if self.undirected\_cycle\_Detector():

print("После удаления вершин граф так и не стал

деревом")

print("")

else:

print("Путем удаления вершин введенный граф стал

деревом")

print("")

self.output\_Graph()

self.nx\_Undirected\_Graph\_Draw()

#print("Приведем граф к ориентированному виду для

удобства восприятия")

#self.utod()

#self.output\_Graph()

else:

print("Введено некорректное число удаляемых вершин")

print("")

else:

print("Граф не был введен")

# Метод поиска всех путей между двумя вершинами графа

def find\_All\_Paths(self, start, end, path=[]):

path = path + [start]

if start == end:

return [path]

if start not in self.\_edges.keys():

return []

paths = []

for node in self.\_edges[start]:

if node not in path:

newpaths = self.find\_All\_Paths(node, end, path)

for newpath in newpaths:

paths.append(newpath)

return paths

# Метод подсчета количества путей между всеми вершинами графа

def all\_Graph\_Paths(self):

vertexes = set()

for vertex in self.\_edges.keys():

vertexes.add(vertex)

for value in self.\_edges[vertex]:

vertexes.add(value)

count = 0

all\_paths = []

for vertex in vertexes:

for jertex in vertexes:

if jertex != vertex:

all\_paths.append(self.find\_All\_Paths(vertex, jertex,

[]))

for paths in all\_paths:

count += len(paths)

print("Между всеми вершинами графа существует " + str(count) + "

путей")

print("")

self.nx\_Directed\_Graph\_Draw()

# Метод отрисовки ориентированного графа с помощью библиотеки networkx и matplotlib

def nx\_Undirected\_Graph\_Draw(self):

G = nx.Graph()

for vertex in self.\_edges.keys():

values = self.\_edges[vertex]

for value in values:

G.add\_edge(vertex, value)

nx.draw(G, with\_labels = True)

plt.show()

# Метод отрисовки неориентированного графа с помощью библиотеки networkx и matplotlib

def nx\_Directed\_Graph\_Draw(self):

G = nx.DiGraph()

for vertex in self.\_edges.keys():

values = self.\_edges[vertex]

for value in values:

G.add\_edge(vertex, value)

nx.draw(G, with\_labels = True)

plt.show()

# Метод поиска цила Эйлера в графе

def eulerian\_Detector(self):

"""

Для неориентированного графа

if self.\_edges\_value > 0:

graph = copy.deepcopy(self.\_edges) # Буферный граф

odd = [ vertex for vertex in graph.keys() if

len(graph[vertex]) & 1 ]

keys = []

for key in graph.keys():

keys.append(key)

odd.append(keys[0])

if len(odd) > 3:

print("Граф не содержит Эйлерова цикла")

print("")

return

stack = [ odd[0] ]

path = []

# Цикл обхода графа

while stack:

vertex = stack[-1]

if graph[vertex]:

jertex = graph[vertex][0]

stack.append(jertex)

del graph[jertex][ graph[jertex].index(vertex) ]

del graph[vertex][0]

else:

path.append(stack.pop())

print("Эйлеров цикл в введенном графе: ")

print (str(path))

print("")

return path

else:

print("Граф пуст")

print("")

"""

# Для всех типов графов:

if self.\_edges\_value > 0:

G = nx.DiGraph()

for vertex in self.\_edges.keys():

values = self.\_edges[vertex]

for value in values:

G.add\_edge(vertex, value)

if nx.is\_eulerian(G):

print("Граф содержит Эйлеров цикл")

print("Эйлеров цикл в введенном графе: ")

print (str(list(nx.eulerian\_circuit(G))))

print("")

return list(nx.eulerian\_circuit(G))

else:

print("Граф не содержит Эйлеров цикл")

print("")

else:

print("Граф пуст")

print("")