НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«Московский энергетический институт»
Кафедра математического и компьютерного моделирования

«Технологии программирования»

Лабораторная работа
Вариант №16

Выполнил: Сошников С. А.
Группа: А-16-19

Преподаватель: Князев А.В.

Москва 2021г

Задание на лабораторную работу

Общее:

Разработать класс для представления неориентированного графа и поиска цикла или пути между двумя вершинами при заданных условиях.
Разработать программу на языке Python, реализующую конкретное задание.
Программа должна обеспечивать ввод описания графа из текстового файла.
В скобках указана форма внутреннего представления графа: граф_матр (матрица смежности), граф_спис (множество списков смежных вершин).
Программа должна обеспечивать представление исходного графа и результата в графическом виде.
Отчёт по лабораторной работе должен содержать:

- Титульный лист
- Задание на работу (общее и индивидуальное)
- Описание работы программы
- Алгоритмы выполнения основных операций
- Тесты
- Распечатки экранов при работе программы
- Листинг программы

Индивидуальное:

Найти путь между двумя вершинами, не проходящий через центр графа (граф_спис).

Описание работы программы

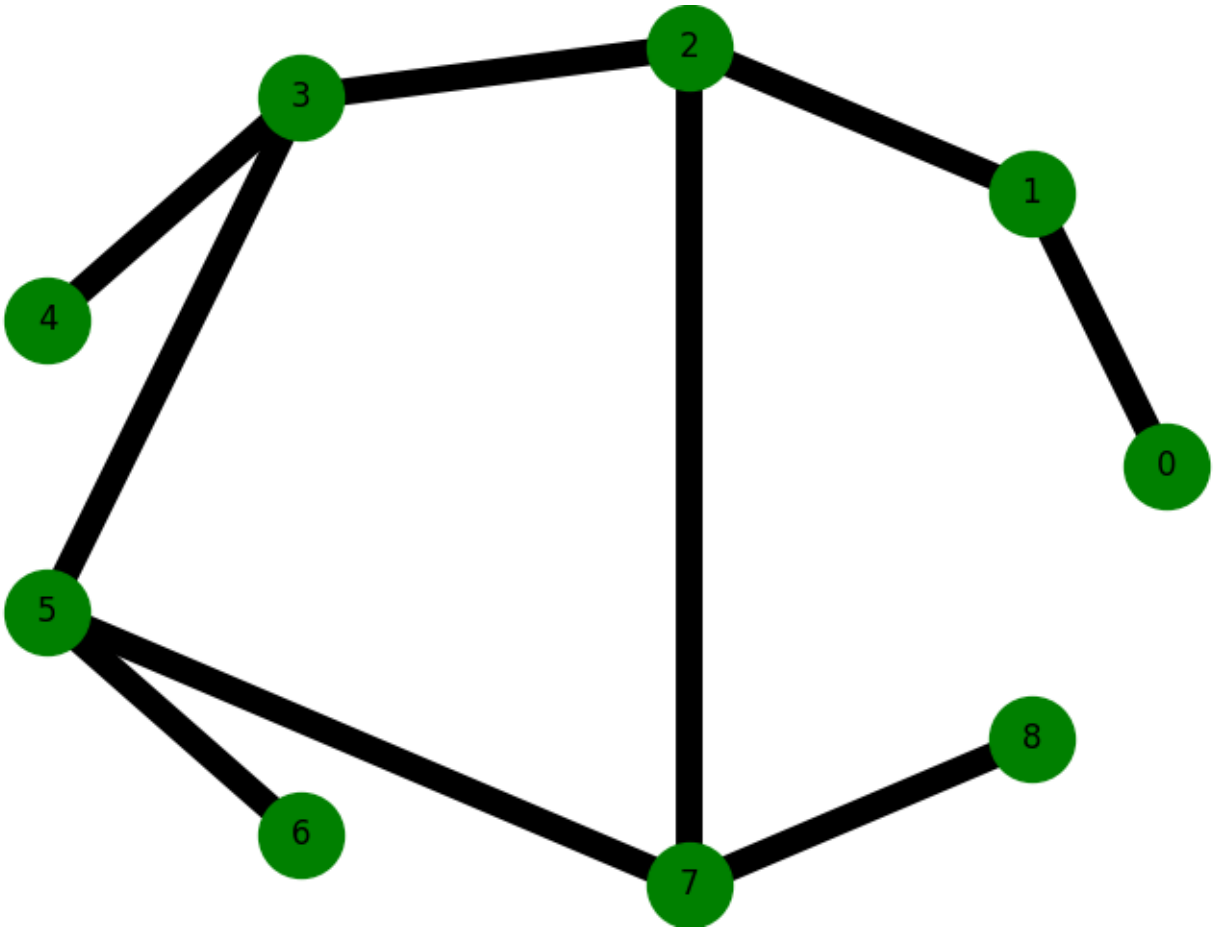
Из файла считываются данные для графика: количество вершин и количество рёбер, ну а по этим данным строится граф: вершины нумеруются от 0. Граф строится путём добавления необходимых рёбер с определённым весом (расстояние между вершинами), при этом граф реализован через список смежности, т.е. граф описан списком, каждый i-ый элемент которого – список смежных вершин по отношению к i-ой вершине.
После построения графа (список смежности заполнен) производится поиск вершин, составляющих центр графа. Для этого через алгоритм Дейкстры рассчитываются расстояния от i-ой вершины до всех остальных, после чего берется максимум в каждом случае. Таким образом будут получены эксцентриситеты всех точек графа, из них выбираются точки с наименьшим эксцентриситетом и заносятся в соответствующий список.
Имея на руках список центральных вершин, переходим к поиску пути, не проходящего через центры.
Последующий алгоритм реализован рекурсивно: из начальной точки проходим по всем смежным, потом по смежным смежных и т.д., минуя центр и точки, в которых уже побывали. Как только путь из А в Б будет проложен – задача решена. Далее строится граф, в котором все вершины отмечены зелёным, а рёбра – черным. Необходимый путь выделяется красным цветом. Если путь невозможен, то выводится чистый граф.

Описание алгоритмов

Открыть файл на чтение => Считать строку => Записать кол-во вершин => Считать строку => Записать кол-во рёбер => Считываем элементы строк до пробела, либо символа переноса строки => Держим левую вершину (первое значение), правую вершину (второе значение), вес (третье значение) => Создаём ребро, записывая левую вершину в список смежности правой, правую – в список смежности левой + записываем вес => граф построен => Алгоритм Дейкстры => находим максимумы в расстояниях от каждой точки => Записываем список эксцентриситетов => находим в нём минимум => Создаём список центральных вершин, т.е. тех, что с минимальных эксцентриситетом
=> Ищем требуемый путь из точки А в точку Б => А != Б ? => Создаём список смежных вершин для точки А => Удаляем из списка центральные вершины и те, в которых уже побывали => Повторяем рекурсивно алгоритм для всех смежных точек к А, пока не будет выполнено А = Б
=> Строим граф с зелёными вершинами и чёрными рёбрами => Если путь найден, то выделяем его красным цветом, иначе оставляем граф в чистом виде => Выводим граф

Тесты

Рассмотрим следующий граф:



Л – левая вершина, П – правая вершина, В – вес
Л П В

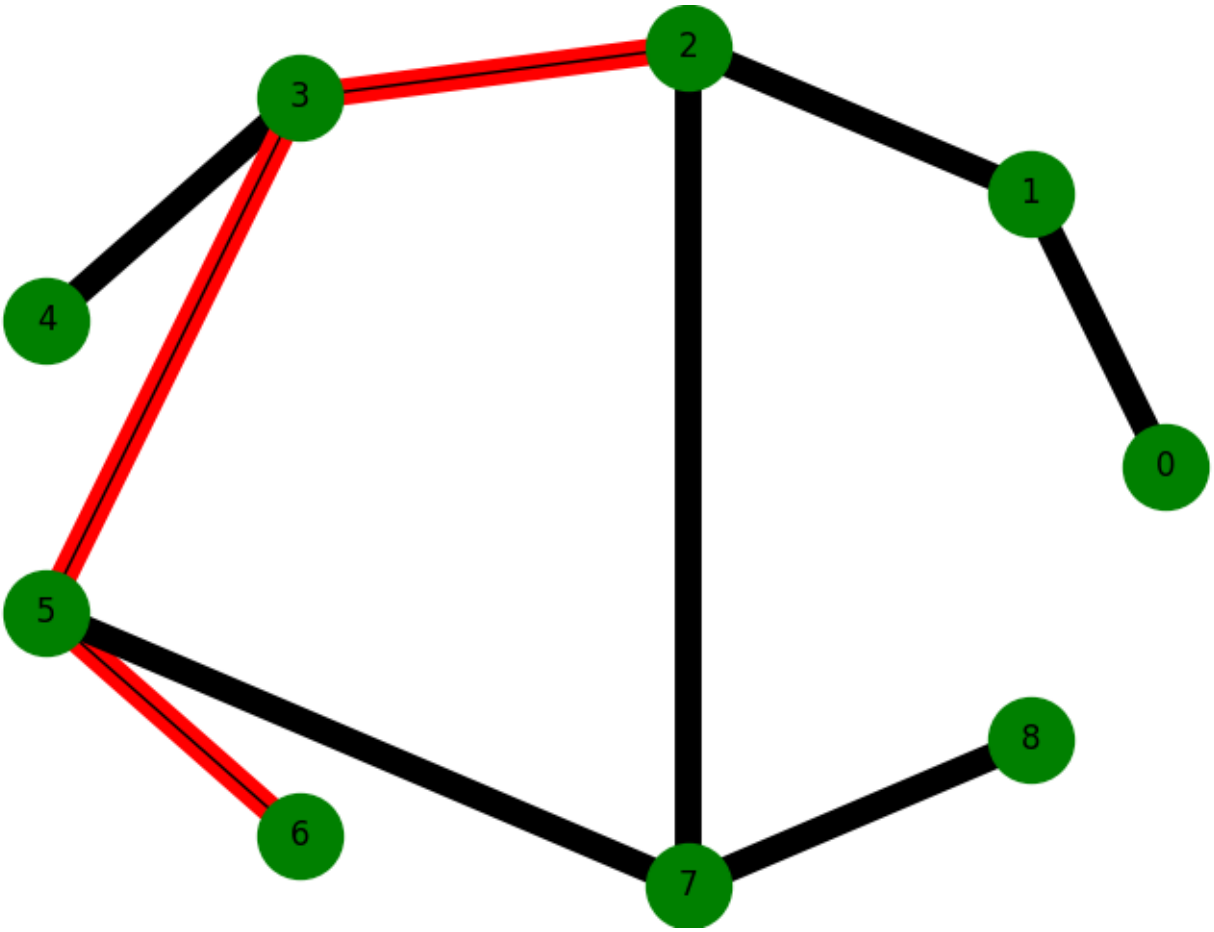
```
0 1 2
1 2 3
2 3 2
3 4 7
3 5 8
5 6 2
5 7 6
7 8 10
7 2 3
```

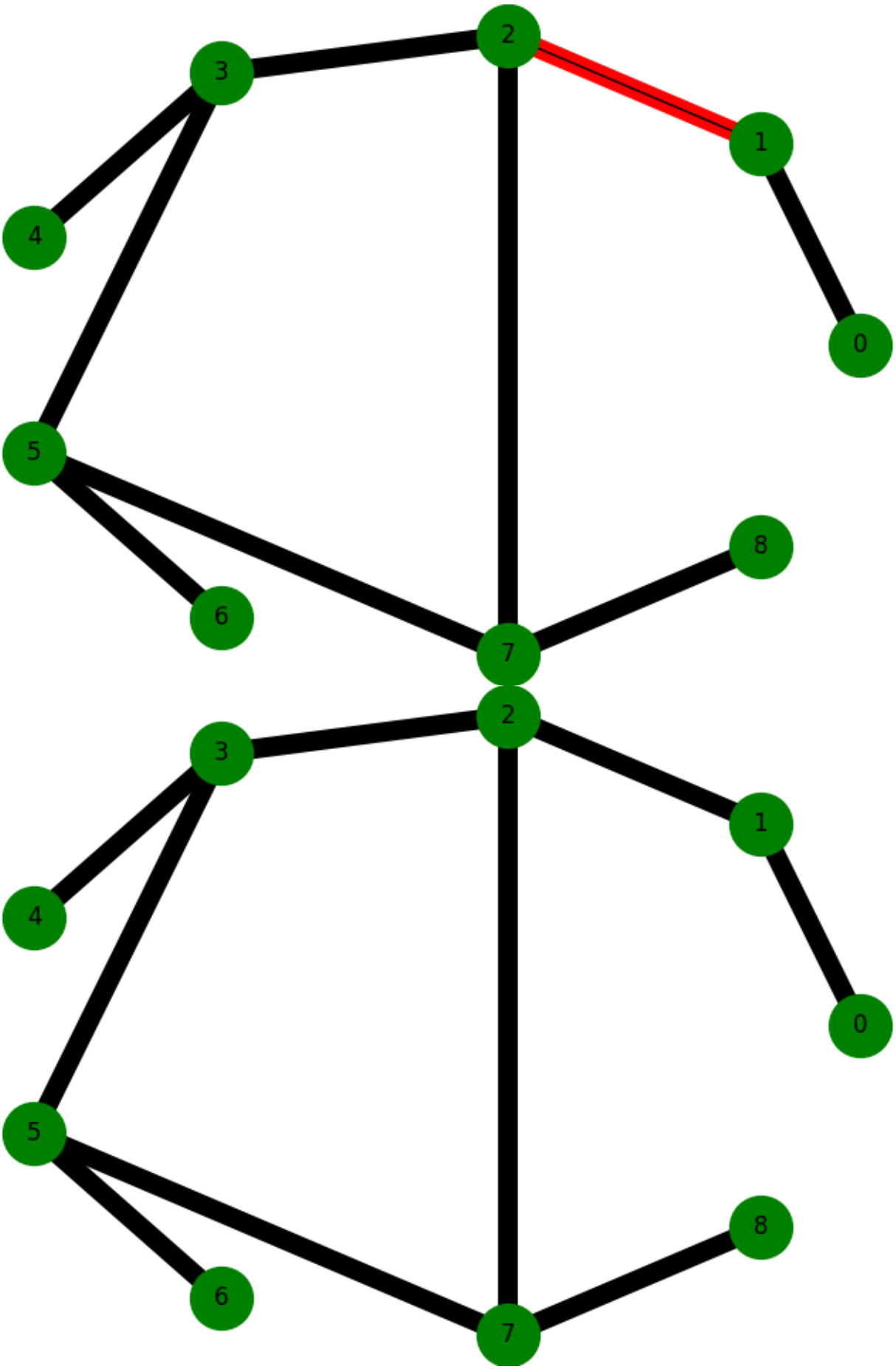
Центр: вершина 7

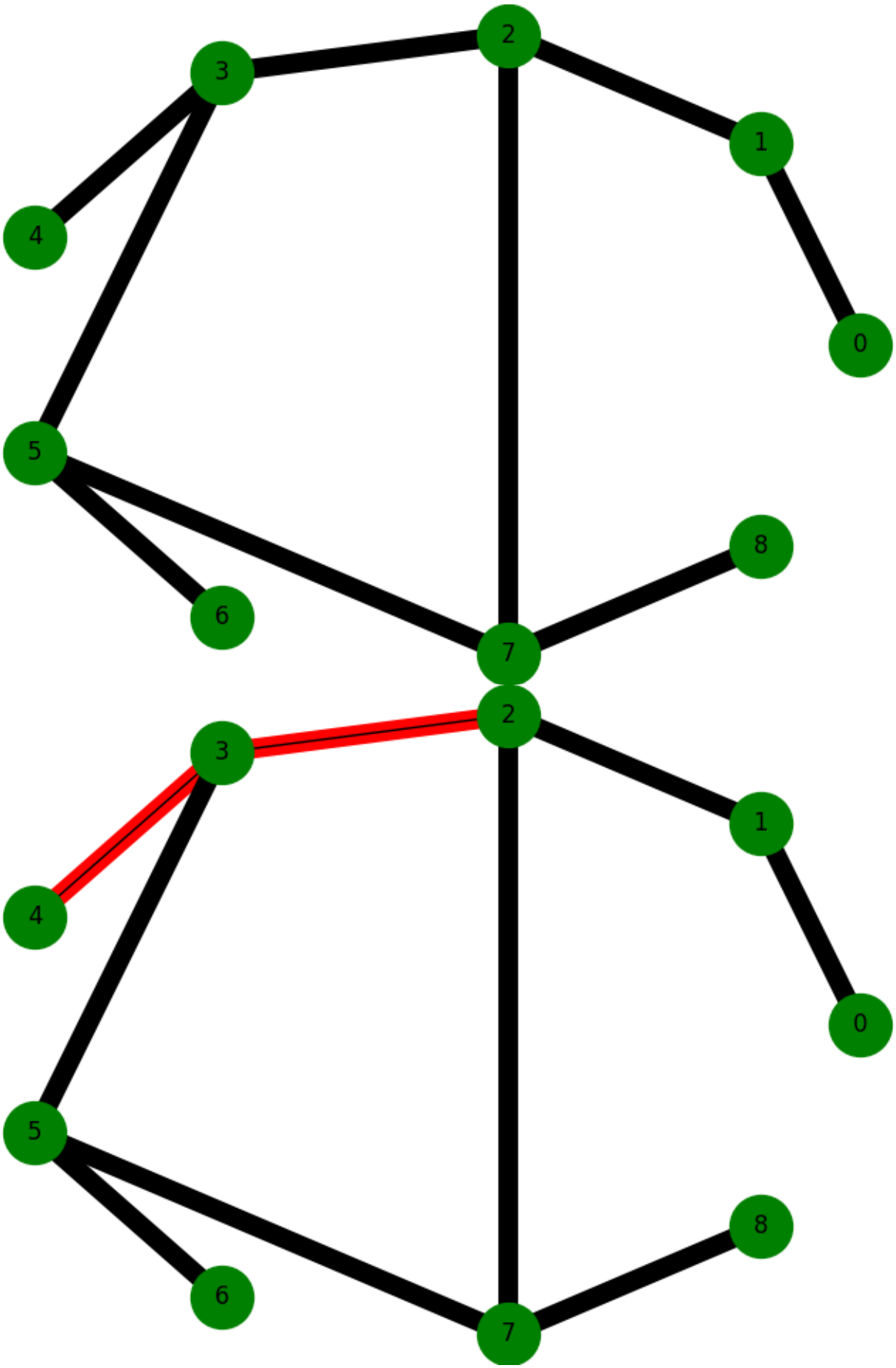
Перейдём к тестам:
Из 4 в 1 : 4 > 3 > 2 > 1

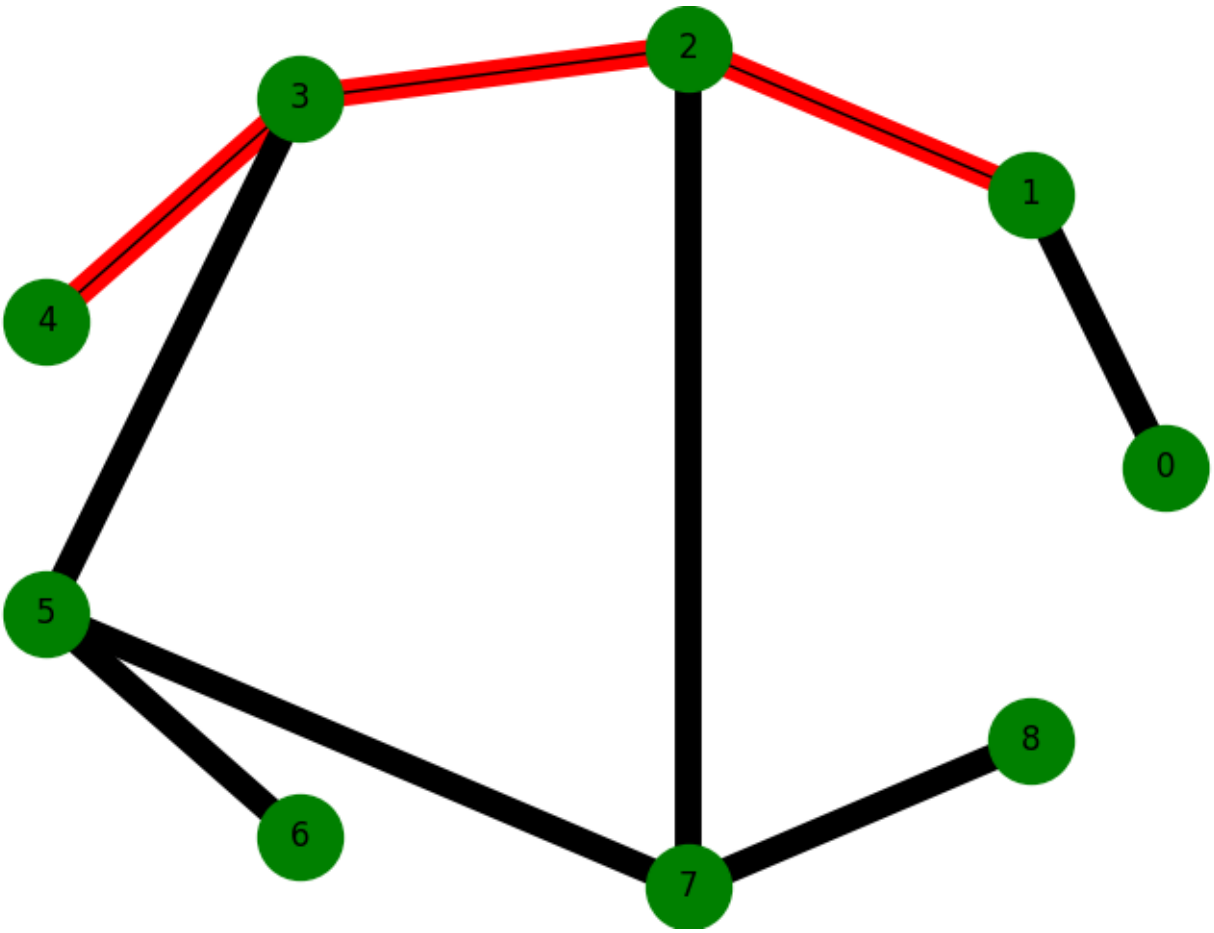
Из 2 в 6 : 2 > 3 > 5 > 6
Из 1 в 2 : 1 > 2
Из 0 в 8 : Нет пути
Из 2 в 7 : Нет пути
Из 2 в 4 : 2 > 3 > 4

Распечатки экрана при работе программы




2
3
1
4
0
5
8
6
7
2
3
1
4
0
5
8
6
7


2
3
1
4
0
5
8
6
7
2
3
1
4
0
5
8
6
7


2
3
1
4
0
5
8
6
7

Листинг программы

```
import matplotlib.pyplot as plt
import networkx as nx


class AdjNode:
    def __init__(self, data, distance):
        self.vertex = data
        self.distance = distance
        self.next = None


class Graph:
   def __init__(self, vertices):
      self.V = vertices
      self.graph = [None] * self.V

   def add_edge(self, src, dest, distance):
      node = AdjNode(dest, distance)
      node.next = self.graph[src]
      self.graph[src] = node

      node = AdjNode(src, distance)
      node.next = self.graph[dest]
      self.graph[dest] = node

   def print_graph(self):
      for i in range(self.V):
         print("Adjacency list of vertex {}\n head".format(i), end="")
         temp = self.graph[i]
         while temp:
            print(" -> {} (dist: {})".format(temp.vertex, temp.distance),
end="")
            temp = temp.next
         print(" \n")

   def dekstra_algorythm(self, src):
      seen = [None] * self.V
      provisional_distance = [float('inf')] * self.V
      fake_distance = {}
      seen[src] = True
      provisional_distance[src] = 0
      for i in range(self.V):
         fake_distance[i] = provisional_distance[i]
      del fake_distance[src]
      current_node = src
```

```
      coefficient = 0
      while None in seen:
         current_dist_dict = {}
         temp = self.graph[current_node]
         while temp:
            current_dist_dict[temp.vertex] = temp.distance
            temp = temp.next
         for i in current_dist_dict:
            if i not in fake_distance:
               continue
            if fake_distance[i] > current_dist_dict[i] + coefficient:
               fake_distance[i] = current_dist_dict[i] + coefficient
         min_in_fake_distance = float('inf')
         min_vert_in_fake_distance = -1
         for i in fake_distance:
            if fake_distance[i] < min_in_fake_distance:
               min_in_fake_distance = fake_distance[i]
               min_vert_in_fake_distance = i
         seen[min_vert_in_fake_distance] = True
         provisional_distance[min_vert_in_fake_distance] =
min_in_fake_distance
         del fake_distance[min_vert_in_fake_distance]
         current_node = min_vert_in_fake_distance
         coefficient = provisional_distance[min_vert_in_fake_distance]
      return provisional_distance

   def find_path_no_center(self, src, dest):
      list_of_eccentricities = [None] * graph.V
      for i in range(graph.V):
         list_of_eccentricities[i] = max(graph.dekstra_algorythm(i))
      min_eccentricity = min(list_of_eccentricities)
      center_list = list(())
      for i in range(len(list_of_eccentricities)):
         if list_of_eccentricities[i] == min_eccentricity:
            center_list.append(i)
      been = [None] * self.V
      path = list(())
      flag = False
      time_res = None
      flag2 = False
      def path_process(current_node, path, been):
         global flag, flag2
         flag = False
         flag2 = False
         been[current_node] = True
         path.append(current_node)
         if current_node == dest:
            flag = True
            flag2 = True
            return path
         temp = self.graph[current_node]
         adjacency_list = {}
         t = 0
         while temp:
            adjacency_list[t] = temp.vertex
            t += 1
            temp = temp.next
         for i in range(len(adjacency_list)):
            if adjacency_list[i] in center_list or been[adjacency_list[i]]:
               del adjacency_list[i]
         if adjacency_list:
            for i in adjacency_list:
               time_res = path_process(adjacency_list[i], list(path),
list(been))
```

```
                if flag2:
                    return time_res
                elif flag:
                    flag = False
                    return path
        return path_process(src, path, been)



def add_edge_for_nx(first_vertex, second_vertex, weight, color, visual_graph
= None):
    visual_graph.add_edge(first_vertex, second_vertex, weight=weight,
color=color)
    visual_graph.add_edge(second_vertex, first_vertex, weight=weight,
color=color)

def is_path(left_vertex, right_vertex):
    for i in range(len(result) - 1):
        if left_vertex == result[i] and right_vertex == result[i + 1] or
left_vertex == result[i + 1] and right_vertex == result[i]:
            return True
    return False







f  = open("input.txt", "r")
V = int(f.readline())
graph = Graph(V)
E = int(f.readline())
for i in range(E):
    left_vert = ""
    t = f.read(1)
    while t != " " and t != "\n":
        left_vert += t
        t = f.read(1)
    right_vert = ""
    t = f.read(1)
    while t != " " and t != "\n":
        right_vert += t
        t = f.read(1)
    weight = ""
    t = f.read(1)
    while t != " " and t != "\n":
        weight += t
        t = f.read(1)
    graph.add_edge(int(left_vert), int(right_vert), int(weight))
f.readline()

source = ""
t = f.read(1)
while t != " " and t != "\n" and t != ".":
    source += t
    t = f.read(1)
destination = ""
t = f.read(1)
while t != " " and t != "\n" and t != "." and t != "":
    destination += t
    t = f.read(1)

result = graph.find_path_no_center(int(source), int(destination))
if not result:
```

```
   print("We cannot build a route between these vertexes without visiting
center")
else:
   print("To avoid visiting center your path is: ")
   for i in result:
      if i != result[len(result) - 1]:
         print(i, "-> ", end="")
      else:
         print(i)

visual_graph = nx.Graph()
for i in range(V):
   visual_graph.add_node(i)
f.seek(0); f.readline(); f.readline()
for i in range(E):
   left_vert = ""
   t = f.read(1)
   while t != " " and t != "\n":
      left_vert += t
      t = f.read(1)
   right_vert = ""
   t = f.read(1)
   while t != " " and t != "\n":
      right_vert += t
      t = f.read(1)
   weight = ""
   t = f.read(1)
   while t != " " and t != "\n":
      weight += t
      t = f.read(1)
   if result and is_path(int(left_vert), int(right_vert)):
      add_edge_for_nx(int(left_vert), int(right_vert), int(weight), 'r',
visual_graph)
   else:
      add_edge_for_nx(int(left_vert), int(right_vert), int(weight), 'black',
visual_graph)

pos = nx.circular_layout(visual_graph)
edges = visual_graph.edges()
colors = [visual_graph[u][v]['color'] for u,v in edges]

nx.draw(visual_graph, pos, edge_color=colors, width=10)

nx.draw_circular(visual_graph, node_color='green', node_size=1000,
with_labels=True)


f.close()
plt.show()
```

Вид файла

```
9
9
0 1 2
1 2 3
2 3 2
3 4 7
3 5 8
5 6 2
5 7 6
7 8 10
7 2 3
# below you have to set start and final point of route
1 6
```

1 строка – кол-во вершин
2 строка – кол-во рёбер
Последующие строки – рёбра в формате «леваявершина праваявершина вес»
Последняя строка – первое значение – пункт отправления, второе – пункт прибытия