

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет Информатика и вычислительная техника

Кафедра Кибербезопасность информационных систем

**ПРАКТИЧЕСКАЯ РАБОТА № 4**

на тему «Вероятностное программирование»

Выполнила студентка гр. ВКБ32

Дедиченко Дарья Валентиновна

Проверил

Доцент, Савельев Василий Александрович

Ростов-на-Дону

2022

***Задание 1.***

import rwgraph

class Node:

    def \_\_init\_\_(self, data, indexloc = None):

        self.data = data

        self.index = indexloc

class Graph:

    @classmethod

    def create\_from\_nodes(self, nodes):

        return Graph(len(nodes), len(nodes), nodes)

    def \_\_init\_\_(self, row, col, nodes = None):

        # установка матрица смежности

        self.adj\_mat = [[0] \* col for \_ in range(row)]

        self.nodes = nodes

        for i in range(len(self.nodes)):

            self.nodes[i].index = i

    # Связывает node1 с node2

    # Ряд - источник, столбец - назначение

    def connect\_dir(self, node1, node2, weight = 1):

        node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)

        self.adj\_mat[node1][node2] = weight

    # Опциональный весовой аргумент для поддержки алгоритма Дейкстры

    def connect(self, node1, node2, weight = 1):

        self.connect\_dir(node1, node2, weight)

        self.connect\_dir(node2, node1, weight)

    # Получает ряд узла, отметить ненулевые объекты с их узлами в массиве self.nodes

    # Выбирает любые ненулевые элементы, оставляя массив узлов

    # которые являются connections\_to (для ориентированного графа)

    # Возвращает значение: массив кортежей (узел, вес)

    def connections\_from(self, node):

        node = self.get\_index\_from\_node(node)

        return [(self.nodes[col\_num], self.adj\_mat[node][col\_num]) for col\_num in range(len(self.adj\_mat[node])) if self.adj\_mat[node][col\_num] != 0]

    # Проводит матрицу к столбцу узлов

    # Проводит любые ненулевые элементы узлу данного индекса ряда

    # Выбирает только ненулевые элементы

    # используется connections\_to ИЛИ connections\_from

    # Возвращает значение: массив кортежей (узел, вес)

    def connections\_to(self, node):

      node = self.get\_index\_from\_node(node)

      column = [row[node] for row in self.adj\_mat]

      return [(self.nodes[row\_num], column[row\_num]) for row\_num in range(len(column)) if column[row\_num] != 0]

    def print\_adj\_mat(self):

      for row in self.adj\_mat:

          print(row)

    def node(self, index):

      return self.nodes[index]

    def remove\_conn(self, node1, node2):

      self.remove\_conn\_dir(node1, node2)

      self.remove\_conn\_dir(node2, node1)

    # Убирает связь в направленной манере (nod1 к node2)

    # Может принять номер индекса ИЛИ объект узла

    def remove\_conn\_dir(self, node1, node2):

      node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)

      self.adj\_mat[node1][node2] = 0

    # Может пройти от node1 к node2

    def can\_traverse\_dir(self, node1, node2):

      node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)

      return self.adj\_mat[node1][node2] != 0

    def has\_conn(self, node1, node2):

      return self.can\_traverse\_dir(node1, node2) or self.can\_traverse\_dir(node2, node1)

    def add\_node(self,node):

      self.nodes.append(node)

      node.index = len(self.nodes) - 1

      for row in self.adj\_mat:

        row.append(0)

      self.adj\_mat.append([0] \* (len(self.adj\_mat) + 1))

    # Получает вес, представленный перемещением от n1

    # к n2. Принимает номера индексов ИЛИ объекты узлов

    def get\_weight(self, n1, n2):

        node1, node2 = self.get\_index\_from\_node(n1), self.get\_index\_from\_node(n2)

        return self.adj\_mat[node1][node2]

    # Разрешает проводить узлы ИЛИ индексы узлов

    def get\_index\_from\_node(self, node):

        if not isinstance(node, Node) and not isinstance(node, int):

            raise ValueError("node must be an integer or a Node object")

        if isinstance(node, int):

            return node

        else:

            return node.index

    def dijkstra(self, node):

        # Получает индекс узла (или поддерживает передачу int)

        nodenum = self.get\_index\_from\_node(node)

        # Заставляет массив отслеживать расстояние от одного до любого узла

        # в self.nodes. Инициализирует до бесконечности для всех узлов, кроме

        # начального узла, сохраняет "путь", связанный с расстоянием.

        # Индекс 0 = расстояние, индекс 1 = перескоки узла

        dist = [None] \* len(self.nodes)

        for i in range(len(dist)):

            dist[i] = [float("inf")]

            dist[i].append([self.nodes[nodenum]])

        dist[nodenum][0] = 0

        # Добавляет в очередь все узлы графа

        # Отмечает целые числа в очереди, соответствующие индексам узла

        # локаций в массиве self.nodes

        queue = [i for i in range(len(self.nodes))]

        # Набор увиденных на данный момент номеров

        seen = set()

        while len(queue) > 0:

            # Получает узел в очереди, который еще не был рассмотрен

            # и который находится на кратчайшем расстоянии от источника

            min\_dist = float("inf")

            min\_node = None

            for n in queue:

                if dist[n][0] < min\_dist and n not in seen:

                    min\_dist = dist[n][0]

                    min\_node = n

            # Добавляет мин. расстояние узла до увиденного, убирает очередь

            queue.remove(min\_node)

            seen.add(min\_node)

            # Получает все следующие перескоки

            connections = self.connections\_from(min\_node)

            # Для каждой связи обновляет путь и полное расстояние от

            # исходного узла, если полное расстояние меньше

            # чем текущее расстояние в массиве dist

            for (node, weight) in connections:

                tot\_dist = weight + min\_dist

                if tot\_dist < dist[node.index][0]:

                    dist[node.index][0] = tot\_dist

                    dist[node.index][1] = list(dist[min\_node][1])

                    dist[node.index][1].append(node)

        return dist

def main():

    print('input file name:')

    file\_name = input()

    vertex = set()

    node\_dict = {}

    graph\_matrix = rwgraph.read\_graph(file\_name)

    for i in graph\_matrix:

        vertex.add(i[0])

        vertex.add(i[1])

    for i in vertex:

        node\_dict[i] = Node(i)

    graph = Graph.create\_from\_nodes(list(node\_dict.values()))

    for i in graph\_matrix:

        graph.connect(node\_dict[i[0]], node\_dict[i[1]], int(i[2]))

    print('input start vertex:')

    start = int(input())

    print([(weight, [n.data for n in node]) for (weight, node) in graph.dijkstra(node\_dict[start])])

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Библиотека ввода/вывода:**

def read\_graph(file):

    graph = open(file, 'r')

    reblist=[[int(value) for value in line.split()] for line in graph.readlines()]

    graph.close()

    return reblist

def write\_graph(graph, file):

    new\_file = open(file\_name, 'w')

    for e in graph:

        newf.write('{} {} {}\n'.format(e[0],e[1],e[2]))

    newf.close()

**Тесты:**

class TestShortPath(unittest.TestCase):

    def test\_1(self):

        self.assertEqual(graph\_short\_path\_find(0,3,graph), 20)

    def test\_2(self):

        self.assertEqual(graph\_short\_path\_find(2,3,graph), 11)

    def test\_3(self):

        self.assertEqual(graph\_short\_path\_find(3,0,graph), 20)

    def test\_4(self):

        self.assertEqual(graph\_short\_path\_find(3,2,graph), 11)

    def test\_5(self):

        self.assertEqual(graph\_short\_path\_find(2,4,graph), '2 4 -1')

    def test\_6(self):

        self.assertEqual(graph\_short\_path\_find(4,2,graph), '4 2 -1')

    def test\_7(self):

        self.assertEqual(graph\_short\_path\_find(5,6,graph), '5 6 -1')

    def test\_8(self):

        self.assertEqual(graph\_short\_path\_find(6,5,graph), '6 5 -1')

    def test\_9(self):

        self.assertEqual(graph\_short\_path\_find(1,3,graph), 15)

    def test\_10(self):

        self.assertEqual(graph\_short\_path\_find(3,1,graph), 15)