1 Цель работы

Познакомиться с основными алгоритмами, используемыми при работе с графами.

2 Задание

Вариант 4

Задание на реализацию:

Задание 4:

Объявите структуру данных «Граф» на основе матрицы смежности с возможностью вывода ее текущего представления в терминал, после чего используйте его для реализации алгоритма Дейкстры и Форда-Беллмана. Добавьте возможность сохранения текущего представления графа в файл и загрузки из него.

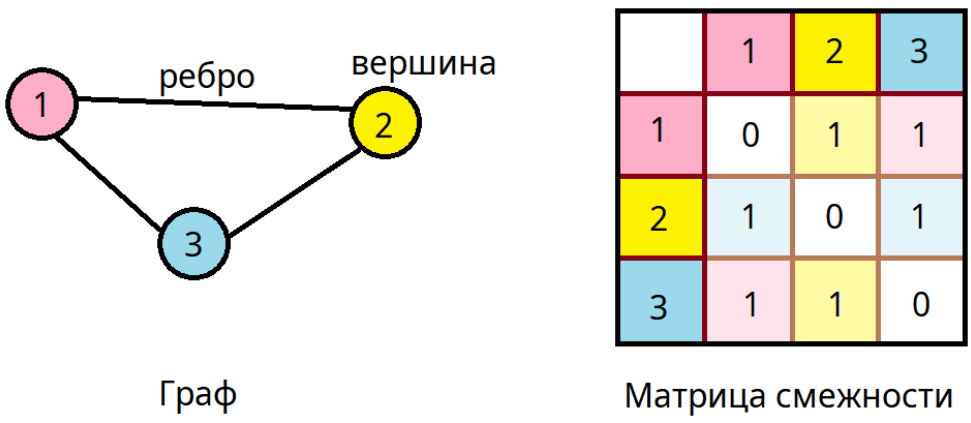
3 Теория по заданиям

Структура данных Граф – это нелинейная структура данных, состоящая из вершин и ребер.

Вершины (Vertex) – узлы;

Рёбра (Edge) – отношение между узлами.

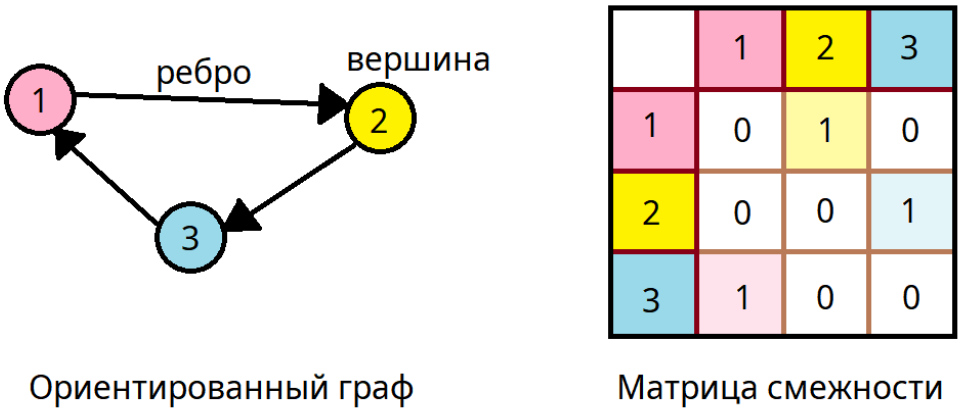
Матрица смежности – это квадратная матрица, используемая для представления конечного графа (рисунок 3.1).



	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Рисунок 3.1 – Граф с матрицей

Ориентированный граф – граф, в котором ребро имеет направление. То есть узлы являются упорядоченными парами в определении каждого ребра (рисунок 3.2).



	1	2	3
1	0	1	0
2	0	0	1
3	1	0	0

Рисунок 3.2 – Ориентированный граф с матрицей

Алгоритм Дейкстры – это метод, который находит кратчайший путь от одной вершины графа к другой. Работает для взвешенных граф и без отрицательных масс, расстояний (рисунок 3.3).

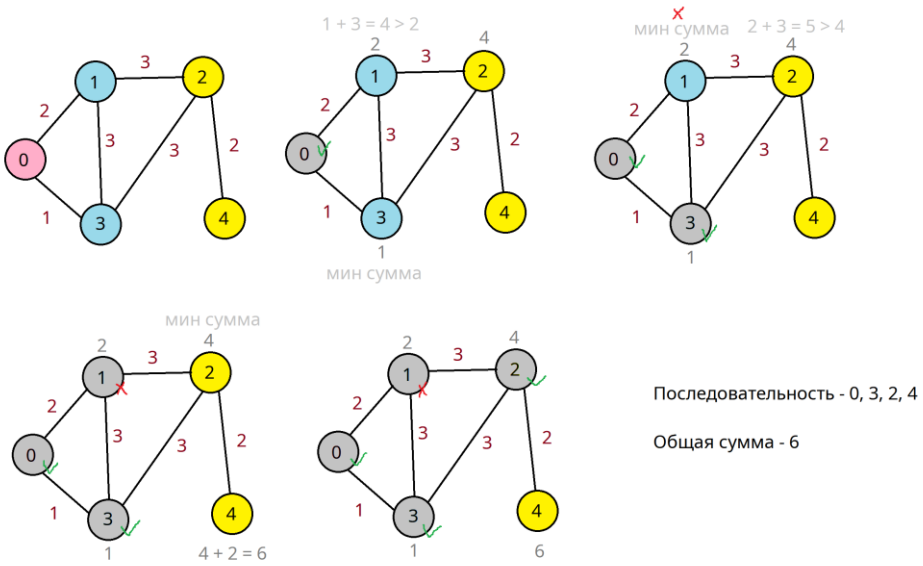


Рисунок 3.3 – Схематический пример алгоритма Дейкстры

Алгоритм нахождения кратчайшего пути:

За начальный текущий индекс берётся стартовый индекс;

Цикл алгоритма идёт, пока текущая вершина не станет конечной;

Перебираем вершины текущей вершины, которые соединены (рёбра) и не пройденные:

- Ребро каждой вершины суммируется с пройденным расстоянием(суммой прошлых расстояний);

- Если получившаяся сумма меньше суммы текущей вершины, то перезаписываем в вершину его новую минимальную сумму и сохраняем вершину в кратчайший путь для его индекса с путём текущей вершины;

После перебора вершин, считаем текущую вершину как пройденную;

Берём за новый текущий индекс, индекс вершины с минимальной суммы на данный момент и не пройденный.

Пример нахождения кратчайшего пути от 0 индекса до 4 индекса алгоритмом Дейкстры с помощью программы на рисунке 3.4:

```
Adjacenc matrix of Graph:
0       2       0       1       0
2       0       3       3       0
0       3       0       3       2
1       3       3       0       0
0       0       2       0       0

Index: 0
[V] 1: 0 + 2[2] < inf
[V] 3: 0 + 1[1] < inf
Path: [0] | min_index: 3 [1]

Index: 3
[X] 1: 1 + 3[4] >= 2
[V] 2: 1 + 3[4] < inf
Path: [0, 3] | min_index: 1 [2]

Index: 1
[X] 2: 2 + 3[5] >= 4
Path: [0, 1] | min_index: 2 [4]

Index: 2
[V] 4: 4 + 2[6] < inf
Path: [0, 3, 2] | min_index: 4 [6]

Short path: [0, 3, 2, 4] | Sum of path: 6
```

Рисунок 3.4 – Пример работы алгоритма Дейкстры в программе

Алгоритм Форда-Беллмана – алгоритм поиска кратчайшего пути во взвешенном графе. Он находит кратчайшие пути от одной вершины графа до всех остальных. Допускаются рёбра с отрицательным весом (рисунок 3.5).

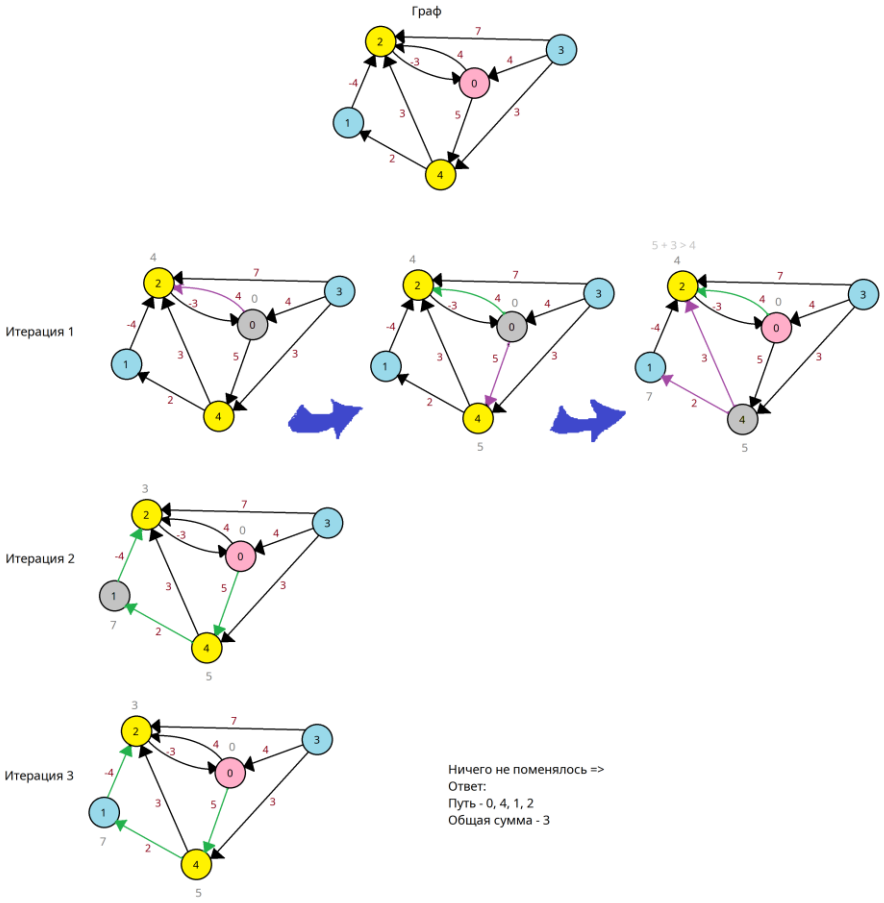


Рисунок 3.5 – Схематический пример алгоритма Форда-Беллмана

Алгоритм нахождения кратчайшего пути:

Идёт цикл из (количество вершин – 1):

В каждой итерации цикла выполняется:

- Перебор всех существующих рёбер
- где в каждом ребре, если получившаяся сумма из конечной вершины ребра меньше суммы текущей вершины и её минимальной суммы, то перезаписываем в вершину его новую минимальную сумму и сохраняем вершину в кратчайший путь для его индекса с путём текущей вершины. А также возможно это отрицательный цикл и ставится флаг, который обнуляется в каждой новой итерации;

После пройденного перебора, если изменений не было, то можно завершить алгоритм, иначе итерации продолжаются;

По окончанию всех итерации, идёт проверка на отрицательный цикл, если он есть (флаг не обнулился), то алгоритм не может иметь правильного решения. Иначе алгоритм завершается с найденным коротким путём и её суммой.

Пример нахождения кратчайшего пути от 0 индекса до 2 индекса алгоритмом Форда-Беллмана с помощью программы на рисунке 3.6:

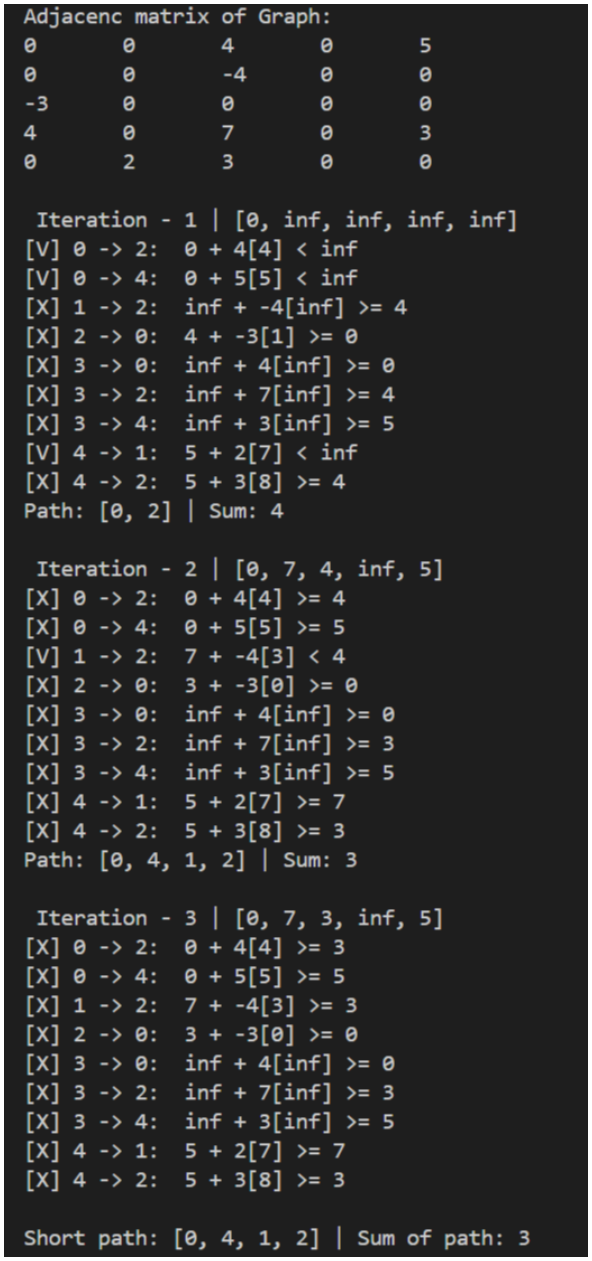


Рисунок 3.6 – Пример работы алгоритма Форда-Беллмана в программе

4 Код программы для задания

Структура данных Граф – Graph для задания со всеми методами.

Путь файла с Графом – structuredata/graph.py

```
import json
from typing import List, Optional

# Класс - Граф на основе матрицы смежности
class Graph:
    # принимает - True|False (ориентированный граф или нет) и хранит
матрицу
```

```
    def __init__(self, directed: bool = False) -> None:
        # матрица смежности
        self.adjacenc_matrix: List[List[Optional[int | float]]] = []
        # ориентированный - True | неориентированный - False
(соединяется вершины с двух сторон)
        self.directed: bool = directed

    # возвращает длину N матрицы NxN
    def get_length(self) -> int:
        return len(self.adjacenc_matrix)

    # добавляет вершину
    def add_vertex(self) -> None:
        matrix = self.adjacenc_matrix
        # если пуст, то добавляем одну вершину
        if len(matrix) == 0:
            matrix.append([None])
            return
        # к каждой вершине добавляем место для новой вершины
        for el in matrix:
            el.append(None)
        # добавление новой вершины (учитывая количество вершин)
        matrix.append([None] * len(el))

    # добавляет ребро с весом от одной вершине к другой по индексам
    def add_edge(self, from_index: int, to_index: int, weight: int |
float) -> bool:
        matrix = self.adjacenc_matrix
        # если индекс меньше 0 или превышает или равен количество
вершин или индексы равны
        if min(from_index, to_index) < 0 or max(from_index, to_index)
>= len(matrix) or from_index == to_index:
            return False
        # если пуст или только одна вершина, то выход
        if len(matrix) < 2:
            return False

        # добавляем ребро от одной вершины к другой
        matrix[from_index][to_index] = weight
        # и наоборот добавляем от другой к одной вершине, если
ненаправленный
        if not self.directed:
            matrix[to_index][from_index] = weight
        return True

    # возвращает список вершины, хранящий все вершины с которыми
соединён и нет
    def get_vertex(self, index: int) -> Optional[List[Optional[int |
float]]]:
        matrix = self.adjacenc_matrix
        if index < 0 or index >= len(matrix):
```

```
            return
        return matrix[index]

    # возвращает вес ребра
    def get_edge(self, from_index: int, to_index: int) -> Optional[int
| float]:
        matrix = self.adjacenc_matrix
        if min(from_index, to_index) < 0 or max(from_index, to_index)
>= len(matrix) or from_index == to_index:
            return
        return matrix[from_index][to_index]

    # удаляет вершину
    def remove_vertex(self, index: int) -> None:
        matrix = self.adjacenc_matrix
        # если пуст ил индекс вышел из диапазона списка, то выход
        if len(matrix) == 0 or index < 0 or index >= len(matrix):
            return
        # удаление всех мест индекса
        for el in matrix:
           el.pop(index)
        matrix.pop(index)

    # удаляет ребро (равняет в None)
    def remove_edge(self, from_index: int, to_index: int) ->
Optional[int | float]:
        matrix = self.adjacenc_matrix
        # если индекс меньше 0 или превышает или равен количество
вершин или индексы равны
        if min(from_index, to_index) < 0 or max(from_index, to_index)
>= len(matrix) or from_index == to_index:
            return
        # если пуст или только одна вершина, то выход
        if len(matrix) < 2:
            return
        # вес ребра
        weight = matrix[from_index][to_index]
        # удаляем ребро от одной вершины к другой
        matrix[from_index][to_index] = None
        # и наоборот удаляем от другой к одной вершине, если
ненаправленный
        if not self.directed:
            matrix[to_index][from_index] = None
        return weight

    # Очистка матрицы Графа
    def clear(self) -> None:
        self.adjacenc_matrix.clear()

    # Сохранение состояния Графа в json формате
    def save(self, path: str) -> None:
```

```
        with open(path, "w", encoding="UTF-8") as file_save:
            json.dump(self.adjacenc_matrix, file_save,
ensure_ascii=False, default=str)

    # Загрузка состояния Графа в json формате
    def load(self, path: str) -> None:
        with open(path, "r", encoding="UTF-8") as file_load:
            self.adjacenc_matrix.clear()
            self.adjacenc_matrix = json.load(file_load)

    # Алгоритм Дейкстры - поиск кратчайшего пути (возвращает список
последовательности индексов и сумму масс их рёбер)
    def shortest_path_Dijkstra(self, start: int, end: int) ->
Optional[tuple[List[int], int | float]]:
        # если граф пустой - выход
        if self.get_length() == 0:
            return
        matrix = self.adjacenc_matrix
        # текущий индекс вершины, из которого будет отмеряется
расстояние к другим вершинам
        index = start
        # к каждой вершине назначаем бесконечную сумму (в будущем это
будут их минимальные суммы)
        min_sum = [float("inf") for _ in range(self.get_length())]
        # у начальной вершины расстояние = 0
        min_sum[index] = 0
        # обработанные вершины (на которых уже были)
        vers = []
        # сохранение индексов вершин минимальных путей расстояний
        path = [[] for _ in range(self.get_length())]
        # начальная вершина имеет себя, в качестве первой вершины
        path[index] = [index]
        # алгоритм продолжается пока индекс не дойдёт до конечной
вершины
        while index != end:
            # существующие рёбра вершины индекса (index)
            ed = []
            for i in range(self.get_length()):
                # если это ребро, то записываем при условии, что
вершина ещё не пройдена
                if matrix[index][i] is not None and i not in vers:
                    ed += [i]
            # перебираем рёбра (индексы вершин)
            for i in ed:
                # получаем расстояние ребра
                mas = self.get_edge(index, i)
                # если ребро меньше, нынешней суммы вершины, к которой
идём то переписываем сумму этого индекса и вписываем в пути
                if min_sum[index] + mas < min_sum[i]:
                    min_sum[i] = min_sum[index] + mas
```

```
                    # записывает весь накопленный путь вершины вместе
с вершиной, к которой идём
                    path[i] = path[index] + [i]
            # записываем текущий индекс, как уже пройдённый
            vers += [index]
            # вычитае через set уберёт из списка пройденные вершины
            index_min = list(set(range(self.get_length())) -
set(vers))
            # берём за индекс самую минимальную сумму, но не берём уже
пройденные
            index = min(index_min, key=lambda x: min_sum[x])
        # возвращаем список самой короткой последовательности вершин и
её сумму
        return (path[end], min_sum[end])

    # Алгоритм Форда-Беллмана - поиск кратчайшего пути (возвращает
список последовательности индексов и сумму масс их рёбер)
    def shortest_path_Bellman_Ford(self, start: int, end: int) ->
Optional[tuple[List[int], int | float]]:
        # если граф пустой - выход
        if self.get_length() == 0:
            return
        # все существующие рёбра - в виде записи (от вершины, до
вершины, масса ребра)
        edges = []
        for i in range(self.get_length()):
            for j in range(self.get_length()):
                # если индексы равны - пропускаем
                if i == j:
                    continue
                # если есть у вершин соединение (ребро), то записываем
в edges
                w = self.get_edge(i, j)
                if w is not None:
                    edges += [(i, j, w)]
        # к каждой вершине назначаем бесконечную сумму (в будущем это
будут их минимальные суммы)
        min_sum = [float("inf") for _ in range(self.get_length())]
        # у начальной вершины расстояние = 0
        min_sum[start] = 0
        # сохранение индексов вершин минимальных путей расстояний
        path = [[] for _ in range(self.get_length())]
        # начальная вершина имеет себя, в качестве первой вершины
        path[start] = [start]
        # есть ли отрицательный цикл
        cycle_minus = True
        # итерации идут до максимального количества возможных вершин в
пути
        for _ in range(self.get_length() - 1):
            # если не изменится суммы, то нахождение пути и её суммы
возможна
```

```
            cycle_minus = False
            # перебор всех рёбер
            for i, j, w in edges:
                # если ребро меньше, нынешней суммы вершины, к которой
идём то переписываем сумму этого индекса и вписываем в пути
                if min_sum[i] + w < min_sum[j]:
                    min_sum[j] = min_sum[i] + w
                    # записывает весь накопленный путь вершины вместе
с вершиной, к которой идём
                    path[j] = path[i] + [j]
                    # возможен отрицательный цикл
                    cycle_minus = True
            # возвращаем путь до заданной вершины (end) и её сумму -
так как итерация ничего не изменила
            if not(cycle_minus):
                return (path[end], min_sum[end])
        # Бесконечный отрицательный цикл либо нету рёбер
        if cycle_minus:
            return None
        # Иначе возвращаем путь до заданной вершины (end) и её сумму
        return (path[end], min_sum[end])

    # Магический метод вывода Графа
    def __str__(self) -> str:
        txt = ""
        for i in self.adjacenc_matrix:
            txt += ("\t".join(map(str, i)).replace("None", "0") +
"\n")
        else:
            txt = txt[:-1]
        return txt
```

Для аннотации типа берём из библиотеки **typing** функции – Optional, List:

- **Optional** – может хранить либо None либо указный тип;
- **List** – тип списка python.

Методы класса Графа (Graph):

- **__init__(directed) –** принимает - True|False (ориентированный граф или нет) и хранит матрицу;
- **get_length() –** возвращает длину N матрицы NxN;
- **add_vertex() –** добавляет вершину;
- **add_edge(from_index, to_index, weight) –** добавляет ребро с весом от одной вершине к другой по индексам;
- **get_edge(from_index, to_index) –** возвращает вес ребра;

- **get_vertex(index)** – возвращает список вершины, хранящий все вершины с которыми соединён и нет;
- **remove_vertex(index)** – удаляет вершину;
- **remove_edge(from_index, to_index)** – удаляет ребро (равняет в None);
- **clear()** – очистка матрицы Графа;
- **save(path)** – сохранение состояния Графа в json формате;
- **load(path)** – загрузка состояния Графа в json формате;
- **__str__()** – магический метод вывода Графа.

Метод алгоритма Графа – Дейкстры:

```
    # Алгоритм Дейкстры - поиск кратчайшего пути (возвращает список
последовательности индексов и сумму масс их рёбер)
    def shortest_path_Dijkstra(self, start: int, end: int) ->
Optional[tuple[List[int], int | float]]:
        # если граф пустой - выход
        if self.get_length() == 0:
            return
        matrix = self.adjacenc_matrix
        # текущий индекс вершины, из которого будет отмеряется
расстояние к другим вершинам
        index = start
        # к каждой вершине назначаем бесконечную сумму (в будущем это
будут их минимальные суммы)
        min_sum = [float("inf") for _ in range(self.get_length())]
        # у начальной вершины расстояние = 0
        min_sum[index] = 0
        # обработанные вершины (на которых уже были)
        vers = []
        # сохранение индексов вершин минимальных путей расстояний
        path = [[] for _ in range(self.get_length())]
        # начальная вершина имеет себя, в качестве первой вершины
        path[index] = [index]
        # алгоритм продолжается пока индекс не дойдёт до конечной
вершины
        while index != end:
            # существующие рёбра вершины индекса (index)
            ed = []
            for i in range(self.get_length()):
                # если это ребро, то записываем при условии, что
вершина ещё не пройдена
                if matrix[index][i] is not None and i not in vers:
                    ed += [i]
            # перебираем рёбра (индексы вершин)
            for i in ed:
                # получаем расстояние ребра
                mas = self.get_edge(index, i)
```

```
                # если ребро меньше, нынешней суммы вершины, к которой
идём то переписываем сумму этого индекса и вписываем в пути
                if min_sum[index] + mas < min_sum[i]:
                    min_sum[i] = min_sum[index] + mas
                    # записывает весь накопленный путь вершины вместе
с вершиной, к которой идём
                    path[i] = path[index] + [i]
            # записываем текущий индекс, как уже пройдённый
            vers += [index]

            # вычитает через set уберёт из списка пройденные вершины
            index_min = list(set(range(self.get_length())) -
set(vers))
            # берём за индекс самую минимальную сумму, но не берём уже
пройденные
            index = min(index_min, key=lambda x: min_sum[x])
        # возвращаем список самой короткой последовательности вершин и
её сумму
        return (path[end], min_sum[end])
```

index – текущий индекс вершины;

index_min – следующий текущий индекс вершины, у которой самая минимальная сумма;

min_sum – запись хранения минимальных сумм для каждой итерации (за начальные числа, берём бесконечность или для это Python - float("inf"));

Но для начальной вершины сумма = 0;

vers – запись посещённых вершин после итераций;

path – запись кротчайшего пути (обновляется с каждой итерации);

ed – список соединённых вершин с текущей вершиной;

mas – расстояние текущей вершины до соединяющей вершины с ней.

Метод алгоритма Графа – Форда-Беллмана:

```
    # Алгоритм Форда-Беллмана - поиск кратчайшего пути (возвращает
список последовательности индексов и сумму масс их рёбер)
    def shortest_path_Bellman_Ford(self, start: int, end: int) ->
Optional[tuple[List[int], int | float]]:
        # если граф пустой - выход
        if self.get_length() == 0:
            return
        # все существующие рёбра - в виде записи (от вершины, до
вершины, масса ребра)
        edges = []
        for i in range(self.get_length()):
            for j in range(self.get_length()):
                # если индексы равны - пропускаем
```

```
                if i == j:
                    continue
                # если есть у вершин соединение (ребро), то записываем
в edges
                w = self.get_edge(i, j)
                if w is not None:
                    edges += [(i, j, w)]
        # к каждой вершине назначаем бесконечную сумму (в будущем это
будут их минимальные суммы)
        min_sum = [float("inf") for _ in range(self.get_length())]
        # у начальной вершины расстояние = 0
        min_sum[start] = 0
        # сохранение индексов вершин минимальных путей расстояний
        path = [[] for _ in range(self.get_length())]
        # начальная вершина имеет себя, в качестве первой вершины
        path[start] = [start]
        # есть ли отрицательный цикл
        cycle_minus = True
        # итерации идут до максимального количества возможных вершин в
пути
        for _ in range(self.get_length() - 1):
            # если не изменится суммы, то нахождение пути и её суммы
возможна
            cycle_minus = False
            # перебор всех рёбер
            for i, j, w in edges:
                # если ребро меньше, нынешней суммы вершины, к которой
идём то переписываем сумму этого индекса и вписываем в пути
                if min_sum[i] + w < min_sum[j]:
                    min_sum[j] = min_sum[i] + w
                    # записывает весь накопленный путь вершины вместе
с вершиной, к которой идём
                    path[j] = path[i] + [j]
                    # возможен отрицательный цикл
                    cycle_minus = True
            # возвращаем путь до заданной вершины (end) и её сумму -
так как итерация ничего не изменила
            if not(cycle_minus):
                return (path[end], min_sum[end])
        # Бесконечный отрицательный цикл либо нету рёбер
        if cycle_minus:
            return None
        # Иначе возвращаем путь до заданной вершины (end) и её сумму
        return (path[end], min_sum[end])
```

edges - все существующие рёбра;

min_sum – запись хранения минимальных сумм для каждой итерации (за начальные числа, берём бесконечность или для это Python - float("inf"));

Но для начальной вершины сумма = 0;

path – запись кротчайшего пути (обновляется с каждой итерации);

cycle_minus – проверка на отрицательный цикл (если отрицателен, то результат алгоритма не может считаться верным).

4.1 Тесты

Код теста:

```
from structuredata.graph import Graph
import os.path

matrix = Graph()
size = 4
m1 = [[None] * size for _ in range(size)]
m2_direct = [[None] * size for _ in range(size)]
m3 = [[None] * size for _ in range(size)]
element = [
    (0, 1, 1),
    (1, 2, 1),
    (2, 3, 1),
    (0, 3, 1)
]

for i, j, w in element:
    m2_direct[i][j] = w
    m3[i][j] = w
    m3[j][i] = w

grafs = """ A   B   C   D   E
A       1
B   1       2   2   7
C       2           3
D       2           4
E       7   3   4
NON
    A   B   C   D   E
A       5   3
B   5       1   4
C   3   1       6
D       4   6       1
E               1
NON
    A   B   C   D   E
A       3   7
B   3       2       8
C   7   2       4
D           4       1
E       8       1   """

negative_graph = """    A   B   C   D   E
```

```
A           4       5
B           -4
C   -3
D   4       7       3
E       2   3
NON
    A   B   C   D   E
A           -4      5
B           -4
C   -3
D   4       -7      3
E       2   3       """

otv = [
    ([0, 1, 2, 4], 6),
    ([0, 2, 1, 3, 4], 9),
    ([0, 1, 2, 3, 4], 10)
    ]

negative_otv = [
    ([0, 4, 1, 2], 3),
    None,
    ]

def to_graph(txt: str, graph: Graph) -> int:
    col:list = txt.rstrip("\n").split("\n")[1:]
    for vs in range(len(col)):
        graph.add_vertex()
    for vs in range(len(col)):
        lt = col[vs].split("\t")[1:]
        for v in range(len(lt)):
            if lt[v].replace("-", "").isdigit():
                graph.add_edge(vs, v, int(lt[v]))
    return len(col)

# добавление в конец вершины
def test_add_vertex():
    for _ in element:
        matrix.add_vertex()
    assert m1 == matrix.adjacenc_matrix

# добавление ребра с весом
def test_add_edge():
    for i, j, w in element:
        matrix.add_edge(i, j, w)
    assert matrix.adjacenc_matrix == m3

# проверка функций на get_ дающие значение
def test_gets():
    assert matrix.get_length() == len(element)
    for i in range(matrix.get_length()):
```

```
        assert matrix.get_vertex(i) == m3[i]
        for j in range(matrix.get_length()):
            assert matrix.get_edge(i, j) == m3[i][j]

# удаление вершины
def test_remove_vertex():
    ind = 1
    matrix.remove_vertex(ind)
    assert matrix.get_length() == size - ind
    assert matrix.get_edge(ind, 0) == m3[ind + 1][0]
    assert matrix.get_edge(ind, 1) == m3[ind + 1][2]
    assert matrix.get_edge(ind, 2) == m3[ind + 1][3]

# удаление ребра
def test_remove_edge():
    weight = matrix.remove_edge(element[0][0], element[0][1])
    assert weight == None
    assert matrix.get_edge(element[0][0], element[0][1]) == None

# удаление и очистка
def test_clear():
    matrix.clear()
    assert matrix.get_length() == 0
    assert matrix.adjacenc_matrix == []

# если граф ориентированный
def test_add_edge_direct():
    matrix.directed = True
    for _ in element:
        matrix.add_vertex()
    for i, j, w in element:
        matrix.add_edge(i, j, w)
    assert matrix.adjacenc_matrix == m2_direct
    matrix.directed = False

# проверка вывода
def test_str():
    list_str = ""
    for i in m2_direct:
        list_str += "\t".join(map(str, i)).replace("None", "0") + "\n"
    else:
        list_str = list_str[:-1]
    assert list_str == str(matrix)

# сохранение состояния в файл (проверка существует ли файл)
def test_save():
    path = "test_json1.json"
    matrix.adjacenc_matrix = [i for i in m3]
    matrix.save(path)
    assert os.path.exists(path)
```

```python
# загрузка файла в пустое дерево
def test_load():
    path = "test_json1.json"
    matrix.clear()
    matrix.load(path)
    assert matrix.adjacenc_matrix == m3

# сверка ответов Алгоритма Дейкстры
def test_dijkstra():
    n = -1
    for txt in grafs.split("NON\n"):
        n += 1
        matrix.clear()
        size = to_graph(txt, matrix)
        path, sm = matrix.shortest_path_Dijkstra(0, size - 1)
        assert path == otv[n][0]
        assert sm == otv[n][1]

# сверка ответов Алгоритма Форда-Беллмана
def test_bellman_ford():
    matrix.directed = True
    n = -1
    for txt in negative_graph.split("NON\n"):
        n += 1
        matrix.clear()
        size = to_graph(txt, matrix)
        res = matrix.shortest_path_Bellman_Ford(0, 2)
        assert res == negative_otv[n]
    n = -1
    for txt in grafs.split("NON\n"):
        n += 1
        matrix.clear()
        size = to_graph(txt, matrix)
        path, sm = matrix.shortest_path_Bellman_Ford(0, size - 1)
        assert path == otv[n][0]
        assert sm == otv[n][1]
```

Результат пройденных тестов (рисунок 4.1).

```
PS D:\Saves\GUAP\SEM3\AuSD\laba6> pytest test_4.py
============================= test session starts =============================
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba6
collected 12 items

test_4.py ............                                                   [100%]

============================== 12 passed in 0.03s ==============================
```

Рисунок 4.1 – результат проверки теста 4 задания

Все 12 тестов были пройдены успешно.

Выводы

В данной лабораторной работе я создала структуру данную Граф на основе матрицы смежности с применением таких алгоритмов поиска кратчайшего пути как:

- Алгоритм Дейкстры;
- Алгоритм Форда-Беллмана.

Научилась применять аннотацию типов с помощью библиотеки typing. А также провела тесты своей структуры данной с алгоритмами поиска кратчайшего пути.