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»

Зав. кафедрой «КБИС»

_____ Д.А. Короченцев
(подпись)

«___» _____ 2023 г.

ОТЧЕТ

по учебной, ознакомительной практике

в ФГАУНИ НИИ «Спецвузавтоматика»

Обучающийся _____ Д.П. Ковалев
подпись, дата

Обозначение отчета УП.99000.000 Группа ВКБ12

Направление 10.05.01 Информатика и вычислительная техника

Направленность (профиль) Математические методы защиты информации

Руководитель практики:

от предприятия зам. директора по научной работе _____ К.Ю. Гуфан
подпись, дата

от кафедры ст. преподаватель _____ Э.Р. Типаева
подпись, дата

Оценка _____ _____ _____
дата подпись преподавателя

Ростов-на-Дону

2023 г.

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»

ЗАДАНИЕ

по учебной, экспериментально-исследовательской практике

в ФГАУНИ НИИ «Спецвузавтоматика»

в период с «__» _____ 2023 г. по «__» _____ 2023 г.

Обучающийся Ковалев Данил Петрович

Обозначение отчета УП.99000.000 Группа ВКБ12

Срок представления отчета на кафедру «__» _____ 2023 г.

Содержание индивидуального задания

Реализация алгоритма Флойда в задачах спортивного программирования.
Создание скрипта, реализующего алгоритм Флойда и взаимодействующий с пользователем через консоль.

Руководитель практики от кафедры	_____ подпись, дата	Э.Р. Типаева
Руководитель практики от предприятия	_____ подпись, дата	К.Ю. Гуфан
Задание принял к исполнению	_____ подпись, дата	Д.П. Ковалев

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»

Зав. кафедрой «КБИС»

_____ Д.А. Короченцев
(подпись)

«___» _____ 2023 г.

Рабочий график (план) проведения практики

№	Мероприятие	Срок выполнения
1	Прохождение вводного и первичного инструктажа по охране труда на рабочем месте, и инструктажа по пожарной безопасности на объекте.	02.06.2023
2	Получение индивидуального задания.	05.06.2023
3	Ознакомление с теоретическим материалом.	05.06.2023
4	Разработка скрипта, реализующего алгоритм Флойда.	06.06.2023-10.06.2023
5	Решение задач с Интернет-ресурсов, используя алгоритм Флойда.	11.06.2023
6	Оформление отчёта по практике.	13.06.2023-15.06.2023
7	Защита отчёта по практике на предприятии.	20.06.2023
8	Защита отчёта по практике на кафедре.	21.06.2023

Руководитель практики:

от предприятия зам. директора по научной работе _____ К.Ю Гуфан
подпись, дата

от кафедры старший преподаватель _____ Э.Р. Типаева
подпись, дата

Ростов-на-Дону

2023 г.

ДНЕВНИК ПРОХОЖДЕНИЯ ПРАКТИКИ

В данном разделе ежедневно, кратко и четко записываются выполняемые работы, и в конце каждой недели журнал представляется для проверки руководителю (от предприятия и университета) практики. При выполнении одной и той же работы несколько дней, в графе «дата» сделать запись «с ___по___».

Дата	Место работы	Выполняемые работы	Оценка руководителя
02.06.2023	ДГТУ	Прохождение вводного и первичного инструктажа по охране труда на рабочем месте, и инструктажа по пожарной безопасности на объекте.	
05.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Получение индивидуального задания.	
06.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Изучение теоретического материала, первые наброски алгоритма Флойда.	
07.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Разработка основной логики алгоритма.	
08.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Оптимизации основного алгоритма, изучение multiprocessing в Python.	
09.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Создание основного скрипта для консольного взаимодействия с пользователем.	
10.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Разработка декораторов, чтобы сделать программу более устойчивой к некорректному вводу, вывод матрицы.	
11.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Решение задач с Интернет-ресурсов, используя алгоритм Флойда.	
с 13.06.2023 по 15.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Оформление отчёта по практике.	
20.06.2023	ФГАУНИ НИИ «Спецвузавтоматика»	Защита отчёта по практике на предприятии.	
21.06.2023	каф. «КБИС»	Защита отчёта по практике на кафедре.	

от предприятия зам. директора по научной работе _____ К.Ю. Гуфан
подпись, дата

от кафедры ст. преподаватель _____ Э.Р. Типаева
подпись, дата

ОТЗЫВ-ХАРАКТЕРИСТИКА

Обучающийся Ковалев Данил Петрович
группа ВКБ12
кафедра «Кибербезопасность информационных систем»
вид практики учебная практика

наименование места практики ФГАУНИ НИИ «Спецвузавтоматика»

Обучающийся выполнил задания программы практики

_____
_____
_____
_____
_____
_____
_____
_____

Дополнительно ознакомился/изучил

_____
_____
_____
_____
_____
_____
_____
_____

Заслуживает оценки
_____

Руководитель практики
от предприятия
_____
«____» _____2023 г.

М.П.

Содержание

Оглавление

Введение

Алгоритм Флойда – Уоршелла, или просто алгоритм Флойда, является классическим алгоритмом динамического программирования для поиска кратчайшего пути в графе. Он был разработан Робертом Флойдом и Стивеном Уоршеллом в 1962 году независимо друг от друга.

Алгоритм Флойда предназначен для нахождения кратчайших путей между всеми парами вершин во взвешенном ориентированном или неориентированном графе. Он основан на пошаговом обновлении дистанций между парами вершин и использует динамическое программирование для нахождения оптимальных путей.

Идея алгоритма заключается в том, чтобы рассмотреть все возможные промежуточные вершины на пути между каждой парой вершин и обновить длины путей, если найден путь с меньшей длиной. Алгоритм постепенно строит таблицу с длинами кратчайших путей между всеми парами вершин.

Алгоритм Флойда является эффективным и универсальным, так как работает с графами любого типа и может быть использован для поиска кратчайших путей в графах с отрицательными ребрами, что отличает его от других алгоритмов, таких как алгоритм Дейкстры.

Алгоритм работает за $\Theta(n^3)$ времени и использует $\Theta(n^2)$ памяти, где n – количество вершин в графе. Нужно учитывать, что не стоит использовать его для очень больших графов.

Алгоритм Флойда широко применяется в различных областях, включая транспортное планирование, маршрутизацию сети, анализ данных и другие. Он является одним из основных алгоритмов для решения задачи о кратчайших путях и является важным инструментом в теории графов и алгоритмическом программировании.

1 Алгоритм Флойда: динамическое программирование, описание работы и оптимизация

1.1 Принципы динамического программирования

Перед тем, как говорить о алгоритме, следует понять, что такое динамическое программирование, так реализация строится на данном способе. Динамическое программирование — это метод решения задач, основанный на разбиении исходной задачи на более мелкие подзадачи и сохранении результатов решения этих подзадач для последующего использования.

Основная идея динамического программирования заключается в том, что если мы можем решить подзадачу один раз и сохранить ее результат, то мы можем использовать этот результат в дальнейшем, вместо того чтобы решать эту же подзадачу снова и снова.

Динамическое программирование часто используется для оптимизации алгоритмов, которые имеют повторяющиеся подзадачи. Это позволяет уменьшить количество вычислений и ускорить работу алгоритма. Как раз алгоритм Флойда строится именно на принципах динамического программирования.

1.2 Описание работы алгоритма Флойда

1. Создается матрица смежности для графа, в нашем случае - *«matrix»*, в которой каждый элемент «*matrix[i][j]*» равен весу ребра между вершинами *«i»* и «*j*». Если между вершинами «*i*» и «*j*» нет ребра, то «*matrix[i][j]* = ∞».

2. Для каждой пары вершин «*i*» и «*j*» находим кратчайший путь между ними, используя все промежуточные вершины. Для этого применяем следующий алгоритм:

- Выбираем вершину, которая будет ключевой, то есть через нее будут проходить пути для нахождения расстояния между двумя другими точками. Условно назовем её «*top*» для дальнейшего удобства;
- Находим минимальное расстояние, используя формулу: *matrix[i][j] = min(matrix[i][j], matrix[i][top] + matrix[top][j])*, где *top, i, j = 1,....,N*. Проще говоря, если путь от «*i*» до «*j*» через «*top*» короче, чем текущий путь от «*i*» до «*j*», то обновляем значение «*matrix[i][j]*» на значение пути от «*i*» до «*j*» через «*top*».

3. После завершения алгоритма «*matrix*» будет содержать кратчайшие пути между всеми парами вершин в графе.

1.3 Оптимизация алгоритма Флойда

Есть несколько способов как можно оптимизировать данный алгоритм, уменьшив потребление ресурсов. Приведу некоторые:

1. Оптимизация по памяти: вместо хранения всей матрицы расстояний между всеми парами вершин, можно хранить только две строки или столбца матрицы. Это возможно благодаря тому, что при вычислении каждой новой строки или столбца нам нужны только предыдущие две строки или столбца.

2. Оптимизация по времени: можно использовать параллельные вычисления для ускорения работы алгоритма на многоядерных процессорах.

2 Реализация алгоритма Флойда на Python 3.11

2.1 Создание декораторов

Для того, чтобы не засорять основной файл «*algorithm_console.py*», создам отдельный файл *«decorators.py»* для описания логики декораторов. В самом начале импортирую нужные функции с модулей.

2.1.1 Импорт модулей

```
from terminaltables import DoubleTable
from time import sleep
from tqdm import tqdm
from colorama import Style, Fore
```

Рисунок 1 – Импортирование нужных функций

Первый модуль, не встроенный в стандартный библиотеку, предназначен для создания таблиц в терминальных и консольных приложениях из списка списков строк. Из «*terminaltables*» импортирую класс *«DoubleTable»* для вывода матриц в консоль.

Второй модуль, встроенный в стандартную библиотеку, предназначен для работы со временем в Python. Здесь я импортирую функцию *«sleep»* для работы с прогресс баром. Данная функция умеет приостанавливать программу на заданное количество секунд, что как раз подходит.

Третий модуль, не встроенный в стандартную библиотеку, предназначен для работы с «прогрессом», проще говоря, с прогресс баром для взаимодействия с пользователем. Здесь мы импортируем класс «*tqdm*» для прогресс бара.

Четвертый модуль, не встроенный в стандартную библиотеку, предназначен для работы с текстом в консоли: изменение цвета текста, шрифта, стиля и т.п. Здесь мы импортируем атрибуты класса, чтобы управлять цветом и стилем.

2.1.2 Разработка прогресс бара

```
comments = Style.BRIGHT + Fore.GREEN + f"\N{snake} Работу выполнил: Ковалев Данил ВКБ12 \N{snake}\nП


1 usage
def progress_bar(func):
    def wrapper(*args, **kwargs):
        print(comments)
        for _ in tqdm(range(100)):
            sleep(0.05)
        return func(*args, **kwargs)

    return wrapper
```

Рисунок 2 – разработка прогресс бара

Здесь мы создаем глобальную переменную, это будет моим водяным знаком. Как раз здесь добавлю оформление текста, которое я желаю видеть в консоли.

Создадим функцию «*progress_bar*». Данная функция является декоратором и принимает в качестве аргумента другую функцию «*func*». Декоратор создает новую функцию-обертку «*wrapper*», которая будет выполняться вместо функции, переданной в качестве аргумента. Функция-обертка использует библиотеку «*tqdm*» для создания прогресс-бара, который будет отображаться в процессе выполнения функции, переданной в качестве аргумента.

Аргументы и ключевые слова, переданные в функцию-обертку, передаются далее в функцию, переданную в качестве аргумента «*func*» после того, как прогресс-бар завершится.

2.1.3 Разработка функции-ошибки

```
def retry_on_value_error(func):
    """
    Функция - декоратор для проверки на ошибки функции.
    В случае чего перезапускает функцию, если неверно ввели данные
    :param func: любая функция, где может произойти ValueError
    :return: результат действия функции в случае успешного выполнения
    """

    def wrapper(*args, **kwargs):
        while True:
            try:
                result = func(*args, **kwargs)
                return result
            except ValueError:
                print("Ошибка! Попробуйте еще раз.")

    return wrapper
```

Рисунок 3 – разработка функции - ошибки

Данная функция является декоратором, который оборачивает другую функцию и добавляет повторное выполнение этой функции в случае возникновения исключения «*ValueError*».

Общая структура декоратора состоит из внутренней функции «*wrapper*», которая выполняет повторные попытки вызова функции «*func*» с переданными аргументами «**args*» и «***kwargs*». Если вызов функции происходит успешно без возникновения «*ValueError*», то результат возвращается из декорированной функции. В случае возникновения исключения «*ValueError*», выводится сообщение об ошибке, и происходит вызов функции для получения правильного результата.

2.1.4 Разработка функции, которая распечатывает матрицу

```
def pprint_matrix(func):
    """
    Функция - декоратор, которая распечатывает матрицу, если функция возвращает её.
    Если вам нужно распечатать матрицу, то надо поставить вторым аргументом.
    :param func: Любая функция, где требуется распечатать матрицу.
    :return: Результат действия функции.
    """

    def inner(*args, **kwargs):
        if isinstance(result := func(*args, **kwargs), tuple):
            _, matrix = result
        else:
            matrix = result
        table_data = [[" "] + [chr(x + 65) for x in range(len(matrix))]]
        table_data.extend([(chr(number + 65), *raw) for number, raw in enumerate(matrix)])
        table_instance = DoubleTable(table_data, "Матрица=смежности")
        print("\n" + table_instance.table, "\n", sep='')
        return result
    return inner
```

Рисунок 4 – разработка функции, которая распечатывает матрицу

Основная структура декоратора состоит из внутренней функции «*inner*», которая выполняет вызов функции «*func*» с переданными аргументами «**args*», «***kwargs*». Затем проверяется тип возвращаемого значения «*result*» функции «*func*». Если «*result*» является кортежем, то предполагается, что возвращаемое значение состоит из двух частей, где первая часть не нуждается в распечатке, а вторая часть – матрица, которую необходимо распечатать. В противном случае, считается, что возвращаемое значение полностью является матрицей.

Для распечатки матрицы, она преобразуется в формат таблицы, где символы A, B, C … используются в качестве заголовков столбцов и строк. Затем создается экземпляр класса «*DoubleTable*» из библиотеки «*terminaltables*». Наконец, возвращается исходное значение «*result*» функции «*func*».

Листинг 1 - Программа «*decorators.py» (для языка Python 3.11)*

```
"""
Данный модуль разработан, чтобы описывать логику декораторов.
"""
all = ["progress_bar", "pprint_matrix", "retry_on_value_error"]


from terminaltables import DoubleTable
from time import sleep
from tqdm import tqdm
from colorama import Style, Fore


comments = Style.BRIGHT + Fore.GREEN + f"\N{snake} Работу
выполнил: Ковалев Данил ВКБ12 \N{snake}\nПриветствую " \
                                       f"пользователя
^_^\nДанный скрипт направлен на реализацию алгоритма " \
                                       f"Флойда.\nЕсли хотите
посмотреть реализацию в реальных задачах, то перейдите " \
                                       f"в другие файлы, в ином
случае:\nначинайте с этого файла."


def progress_bar(func):
    """
    Функция - декоратор для создания прогресс бара.
    Делает так, чтобы программа останавливалась на 5 с,
показывая прогресс бар в консоли.
    Не измеряет само время выполнения программы, только красивая
анимация для взаимодействия.
    :param func: Любая функция.
    :return: Результат действия функции
    """
    def wrapper(*args, **kwargs):
        print(comments)
        for _ in tqdm(range(100)):
            sleep(0.05)
        return func(*args, **kwargs)
```

```
    return wrapper


def retry_on_value_error(func):
    """
    Функция - декоратор для проверки на ошибки функции.
    В случае чего перезапускает функцию, если неверно ввели данные.
    :param func: Любая функция, где может произойти ValueError.
    :return: Результат действия функции в случае успешного выполнения.
    """

    def wrapper(*args, **kwargs):
        while True:
            try:
                result = func(*args, **kwargs)
                return result
            except ValueError:
                print("Ошибка! Попробуйте еще раз.")

    return wrapper


def pprint_matrix(func):
    """
    Функция - декоратор, которая распечатывает матрицу, если функция возвращает её.
    Если вам нужно распечатать матрицу, то надо поставить вторым аргументом.
    :param func: Любая функция, где требуется распечатать матрицу.
    :return: Результат действия функции.
    """
```

```
    def inner(*args, **kwargs):
        if isinstance(result := func(*args, **kwargs), tuple):
            _, matrix = result
        else:
            matrix = result

        # Создание списка узлов для
        table_data = [[" "] + [chr(x + 65) for x in
range(len(matrix))]]
        table_data.extend([(chr(number + 65), *raw) for number,
raw in enumerate(matrix)])
        table_instance = DoubleTable(table_data,
"Матрица=смежности")

        print("\n" + table_instance.table, "\n", sep='')
        return result

    return inner
```

2.2 Визуализация графа

```
import matplotlib.pyplot as plt
import networkx as nx
from itertools import product


2 usages
def graph(matrix):
    g = nx.Graph()  # создаём объект графа
    length = len(matrix)  # матрица
    nodes = [chr(x + 65) for x in range(length)]  # определяем список узлов (ID узлов)
    edges = [(chr(raw + 65), chr(column + 65)) for raw, column in product(range(length), repeat=2) if
             matrix[raw][column]]
    g.add_nodes_from(nodes)
    g.add_edges_from(edges)

    nx.draw(g, with_labels=True, font_weight='bold')
    plt.show()
```

Рисунок 5 – создание графа

Данный код создает граф на основе заданной матрицы смежности и визуализирует его с помощью библиотеки «*matplotlib*» и «*networkx*».

Основная функция «*graph*» принимает матрицу «matrix» в качестве аргумента. Сначала создается объект графа «*g*» с помощью «*nx.Graph*». Затем определяется длина матрицы «*length*» и создается список узлов «*edges*» графа на основе ненулевых значений в матрице смежности. Для каждой пары вершин (узлов), для которых значение в матрице не равно нулю, создается ребро между ними.

После того, как граф был построен с помощью узлов и ребер, вызывается функция «*nx.draw*», которая рисует граф с помощью библиотеки «*networkx*». Параметры «*with_label=True*» и «*font_weight='bold'*» указывают, что надо отображать метки узлов и использовать жирный шрифт для них.

Наконец, с помощью «*plt.show()*» граф отображается на экране.

Листинг 2 – Программа «visual.py» ***(для языка Python 3.11)***

```
"""
    Отдельная сущность. Благодаря это функции из матрицы смежности можно получить граф.
    """
    all = ["graph"]
    import matplotlib.pyplot as plt
    import networkx as nx
    from itertools import product


    def graph(matrix):
        g = nx.Graph()  # создаём объект графа
        length = len(matrix)
        nodes = [chr(x + 65) for x in range(length)]  # определяем список узлов (ID узлов)
        edges = [(chr(raw + 65), chr(column + 65)) for raw, column in product(range(length), repeat=2) if
                 matrix[raw][column]]
        g.add_nodes_from(nodes)
        g.add_edges_from(edges)
        nx.draw(g, with_labels=True, font_weight='bold')
        plt.show()
```

2.3 Взаимодействие с пользователем

2.3.1 Импорт модулей

```
from itertools import permutations, combinations
from math import inf   C3EQUALZ, Yesterday • first commit
from string import ascii_letters
from time import sleep
from typing import Callable

from colorama import Fore

import algorithm_console as alg
from decorators import progress_bar, retry_on_value_error, pprint_matrix
from visual import graph
```

Рисунок 6 – импорт модулей в «*interact_with_user*»

В самом начале я импортирую модули, которые встроены в стандартную библиотеку Python.

1. *itertools* — это модуль, который содержит функции для создания итераторов для эффективной обработки последовательностей.
- *permutations* - функция, которая возвращает все возможные перестановки элементов переданной последовательности.
- *combinations* - функция, которая возвращает все возможные комбинации элементов переданной последовательности заданной длины.

2. *math* — это модуль, который содержит математические функции.
- *inf* — это константа, которая представляет бесконечность.

3. *string* — это модуль, который содержит наборы символов.
- *ascii_letters* — это строка, которая содержит все буквы латинского алфавита (в верхнем и нижнем регистрах).

4. *typing* – это модуль, который предоставляет инструменты для определения и аннотирования типов данных в коде. Использую для повышения читабельности в коде.

– *Callable* - используется для указания типа функции или метода, которые могут быть вызваны.

Последние самые модули я импортировал в данную часть, так как здесь они будут вызываться для результата, предварительно оформив *__init__.py*.

2.3.2 Функция questionary.

```python
@progress_bar
@retry_on_value_error
def questionary() -> tuple[tuple[list[list], list[list]], tuple]:
    """
    Функция, которая задает начальные данные для остальных функций
    """
    start_node = input(Fore.LIGHTWHITE_EX + '\nВведите начальную вершину по лексикограф. порядку (буква англ.) ')
    end_node = input('Введите конечную вершину по лексикограф порядку (буква англ.) ')
    start_node, end_node = sorted((start_node, end_node))

    find_start = input("Введите начало (с какой вершины поиск): ")
    find_end = input("Введите конец: ")
    params = (end_node, start_node, find_end, find_start)

    check: Callable[[str], bool] = lambda alpha: len(alpha) == (alpha in ascii_letters)
    if all(check(i) for i in params) and start_node != end_node:
        params = map(lambda x: ord(x.title()) - 65, params)
        info_for_graph = alg.floyd(next(params) - next(params) + 1)
        shortest_path = alg.get_path(info_for_graph, next(params), next(params))
        return info_for_graph, shortest_path
    raise ValueError()
```

Рисунок 7 – функция для взаимодействия с пользователем

Функция задает начальные данные для других функций, связанных с поиском кратчайшего пути в графе. Она запрашивает у пользователя начальную и конечную вершины графа, а также вершины, откуда и до которых нужно искать кратчайший путь. Затем функция проверяет корректность введенных данных и преобразует их в числовой формат. Далее она использует алгоритм Флойда для нахождения кратчайшего пути между заданными вершинами и возвращает информацию о графе и кратчайший путь в виде кортежа. Если данные введены некорректно, функция вызывает исключение «*ValueError*».

В данном случае, аннотация типов возвращаемого значения указывает на то, что функция должна вернуть кортеж, состоящий из двух элементов:

первый элемент - кортеж, состоящий из двух элементов (список списков и список списков), а второй элемент - кортеж.

2.3.3 Функция create_matrix

```
1 usage
@retry_on_value_error
@pprint_matrix
def create_matrix(n: int) → list[list[int]]:
    """
    Функция создания матрицы смежности на основе входных данных с консоли
    :param n: количество вершин
    :return: матрицу для алгоритма Флойда
    """
    comment = "Введите какой граф вы изначально хотите (oriented, undirected)? По умолчанию undirected. "
    generate_iter = permutations if input(comment).lower() == 'oriented' else combinations
    matrix = [[inf if i != j else 0 for i in range(n)] for j in range(n)]
    for start, end in generate_iter(range(n), 2):
        query = input(f"Есть ли направленное ребро между {chr(65 + start)} и {chr(65 + end)}? ")
        if query.lower() in approval:
            matrix[start][end] = int(input('Введите вес '))
            matrix[end][start] = matrix[start][end] if generate_iter is combinations else matrix[end][start]
    return matrix
```

Рисунок 8 – функция для создания матрицы

Данная функция создает матрицу смежности для графа на основе введенных пользователем данных. В качестве аргумента функция принимает целое число n - количество вершин графа.

Сначала функция запрашивает у пользователя тип графа (ориентированный или неориентированный) и в зависимости от этого выбирает соответствующую функцию для генерации ребер графа: «*permutations*» или «*combinations*». Затем создается матрица matrix размером n x n, заполненная значениями «*inf*» (бесконечность) для всех элементов, кроме диагональных, которые равны 0.

Далее функция запрашивает у пользователя наличие ребер между каждой парой вершин графа и их вес, если такое ребро есть. Если пользователь подтверждает наличие ребра, то в матрицу записывается его вес. Если граф неориентированный, то значение в матрице записывается и

для обратного направления. В конце функция возвращает полученную матрицу matrix.

2.3.4 Функция main и точка запуска.

```
1 usage
def main() -> None:
    info_for_graph, shortest_path = questionary()
    print(f"Самый короткий путь - это {shortest_path[0]} с весом {shortest_path[1]}")
    sleep(10)
    graph(info_for_graph)


if __name__ == "__main__":
    main()
```

Рисунок 9 – функция «*main*»

Данный код запускает функцию «*main*», которая в свою очередь вызывает функцию «*questionary*» для получения информации о графе и кратчайшем пути. Затем выводится сообщение о кратчайшем пути и вызывается функция «*graph*» для создания матрицы смежности графа. Если код запущен как основная программа «*if __name__ == "__main__"*», то функция «*main*» будет выполнена.

Листинг 3 – Программа «interact_with_user» (для языка Python 3.11)

```
"""
Главный модуль, где происходит взаимодействие с пользователем.
"""
from itertools import permutations, combinations
from math import inf
from string import ascii_letters
from time import sleep
from typing import Callable
from colorama import Fore
```

```
import algorithm_console as alg
from decorators import progress_bar, retry_on_value_error,
pprint_matrix
from visual import graph

approval = ('да', 'y', 'yes', 'ofc')

@progress_bar
@retry_on_value_error
def questionary() -> tuple[tuple[list[list], list[list]],
tuple]:
    """
    Функция, которая задает начальные данные для остальных
функций
    """
    start_node = input(Fore.LIGHTWHITE_EX + '\nВведите
начальную вершину по лексикограф. порядку (буква англ.) ')
    end_node = input('Введите конечную вершину по лексикограф
порядку (буква англ.) ')
    start_node, end_node = sorted((start_node, end_node))
    find_start = input("Введите начало (с какой вершины поиск):
")
    find_end = input("Введите конец: ")
    params = (end_node, start_node, find_end, find_start)
    check: Callable[[str], bool] = lambda alpha: len(alpha) ==
(alpha in ascii_letters)
    if all(check(i) for i in params) and start_node !=
end_node:
        params = map(lambda x: ord(x.title()) - 65, params)
        info_for_graph = alg.floyd(next(params) - next(params)
+ 1)
        shortest_path = alg.get_path(info_for_graph,
next(params), next(params))
        return info_for_graph, shortest_path
    raise ValueError()
```

```
@retry_on_value_error
@pprint_matrix
def create_matrix(n: int) -> list[list[int]]:
    """
    Функция создания матрицы смежности на основе входных данных
с консоли
    :param n: количество вершин
    :return: матрицу для алгоритма Флойда
    """
    comment = "Введите какой граф вы изначально хотите
(oriented, undirected)? По умолчанию undirected. "
    generate_iter = permutations if input(comment).lower() ==
'oriented' else combinations
    matrix = [[inf if i != j else 0 for i in range(n)] for j in
range(n)]
    for start, end in generate_iter(range(n), 2):
        query = input(f"Есть ли направленное ребро между
{chr(65 + start)} и {chr(65 + end)}? ")
        if query.lower() in approval:
            matrix[start][end] = int(input('Введите вес '))
            matrix[end][start] = matrix[start][end] if
generate_iter is combinations else matrix[end][start]
    return matrix
def main() -> None:
    info_for_graph, shortest_path = questionary()
    print(f"Самый короткий путь - это {shortest_path[0]} с
весом {shortest_path[1]}")
    sleep(10)
    graph(info_for_graph)

if name == "main":
    main()
```

2.4 Основная логика алгоритма

2.4.1 Импорт модулей

```
import threading

from Practics.console_alg.decorators import pprint_matrix
import Practics.console_alg.interact_with_user as interact
```

Рисунок 10 – импорт модулей в algorithm_console.py

Модуль *«threading»* в Python предоставляет возможность создания и управления потоками выполнения в программе. Потоки позволяют выполнять несколько задач одновременно, что увеличивает производительность программы.

Последние импорты – это функции, которые я описал ранее.

2.4.2 Функция update_path

```
def update_path(i: int, top: int, matrix: list[list], all_path: list[list]) → None:
    """
    Обновление кратчайшего пути между двумя вершинами i и j через промежуточную вершину top
    :param i: номер первой вершины
    :param top: номер промежуточной вершины при движении от i к j
    :param matrix: матрица смежности для графа
    :param all_path: матрица, показывающая вершину, где проходит самый короткий путь через 2 других
    """
    for j in range(len(matrix)):
        if matrix[i][j] > matrix[i][top] + matrix[top][j]:
            matrix[i][j] = matrix[i][top] + matrix[top][j]
            all_path[i][j] = top
```

Рисунок 11 – функция «*update_path*»

Данная функция сделана для того, чтобы можно было её вызывать асинхронно. Здесь мы ищем минимальный вес между двумя вершинами, сохраняя вершину, через которую у нас минимальный вес.

2.4.3 Функция floyd

```
    Функция, реализующая алгоритм Флойда
    :param n: количество вершин
    :return: матрицу смежности, где заполнены всевозможные пути,
    и матрицу, показывающей вершину, где проходит самый короткий путь через 2 других
    """
    matrix = interact.create_matrix(n)
    length = len(matrix)
    all_path = [[0] * length for _ in range(length)]
    for top in range(n):
        threads = []
        for i in range(n):
            thread = threading.Thread(target=update_path, args=(i, top, matrix, all_path))
            thread.start()
            threads.append(thread)
        for thread in threads:
            thread.join()
    for i in range(n):
        if matrix[i][i] < 0:
            raise ValueError("Отрицательный цикл обнаружен! Алгоритм Флойда работает некорректно!")
    return all_path, matrix
```

Рисунок 12 – функция «*floyd*»

Данный код реализует алгоритм Флойда для поиска кратчайших путей между всеми парами вершин в графе. Входным параметром является количество вершин в графе *n*. Функция «*create_matrix*» создает матрицу смежности графа. Затем создается матрица «*all_path*», в которую будут записываться все возможные пути между вершинами, и заполняется нулями. Далее выполняется цикл по всем вершинам графа. Для каждой вершины запускаются *n* потоков, каждый из которых вызывает функцию «*update_path*» с аргументами *i, top, matrix, all_path.*

Функция «update_path» обновляет значения в матрице «all_path», находящиеся на пересечении *i*-ой строки и *top*-го столбца, если новый путь короче текущего. После завершения работы всех потоков проверяется наличие отрицательных циклов в графе, и если они есть, выбрасывается исключение ValueError.

В конце функция возвращает матрицу all_path и матрицу смежности графа matrix.

2.4.4 Функция get_path

```
def get_path(shortest_vertex, end: int, start: int) -> tuple:
    """
    Функция, которая возвращает полный путь от конечной точки к начальной.
    :param shortest_vertex: Матрица путей, которая показывает вершину, соединяющую эти точки.
    :param end: Конечная точка.
    :param start: Начальная точка.
    """
    shortest_path_matrix, distance_matrix = shortest_vertex
    path, summary = [end], 0
    while end != start:
        if shortest_path_matrix[end][start] == 0:
            path.insert(0, start)
            summary += distance_matrix[end][start]
            break
        new_point = shortest_path_matrix[end][start]
        summary += distance_matrix[end][new_point]
        end = new_point
        path.insert(0, end)
    return [chr(65 + vertex) for vertex in path], summary
```

Рисунок 13 – функция *get_path*

Данная функция используется для получения полного пути от конечной точки к начальной на основе кортежа, состоящего из матрицы путей и матрицы весов «*shortest_vertex*». Разбивается на две отдельные матрицы: «*shortest_path_matrix*», которая содержит информацию о вершинах на пути, и «*distance_matrix*», которая содержит расстояния между вершинами. Затем инициализируются переменные «*path*», в которой будет храниться путь, и «*summary*», которая будет содержать суммарную длину пути.

В цикле происходит построение пути от конечной точки «*end*» к начальной точке «*start*». Если значение в матрице путей «*shortest_path_matrix*» для текущих вершин «*end*» и «*start*» равно 0, это означает, что достигнута начальная точка и цикл прерывается. В противном случае, обновляется текущая вершина «*end*» на основе значения в матрице путей, добавляется текущая вершина в начало списка «*path*», и обновляется суммарное расстояние «*summary*».

Функция возвращает путь в виде списка символов, где каждый символ представляет вершину пути (используется символьное представление вершин, начиная с кода символа 'A'), и суммарную длину пути «*summary*».

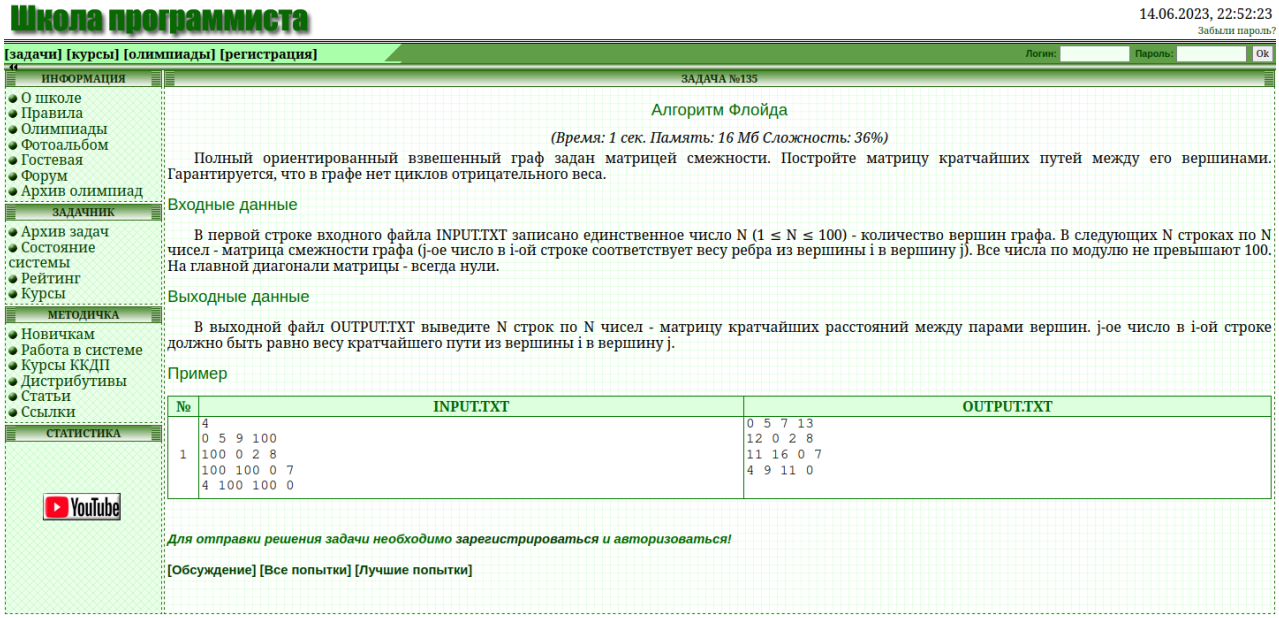
Листинг 4 – Программа «algorithm_console» (для языка Python 3.11)

```
"""
Реализация алгоритма Флойда для взаимодействия через консоль с пользователем.

Разработан в учебных целях, здесь активно используется декораторы во избежание того, чтобы
пользователь не вводил некорректные данные. В случае неверного ввода будет вызвано исключение.

Из оптимизаций: использование параллельных вычислений.
Является только логикой, основное взаимодействие осуществляется в interact_with_user.py
"""
import threading

from Practics.console_alg.decorators import pprint_matrix
import Practics.console_alg.interact_with_user as interact


def update_path(i: int, top: int, matrix: list[list], all_path: list[list]) -> None:
    """
    Обновление кратчайшего пути между двумя вершинами i и j через промежуточную вершину top
    :param i: номер первой вершины
    :param top: номер промежуточной вершины при движении от i к j
    :param matrix: матрица смежности для графа
    :param all_path: матрица, показывающая вершину, где проходит самый короткий путь через 2 других
    """
    for j in range(len(matrix)):
        if matrix[i][j] > matrix[i][top] + matrix[top][j]:
            matrix[i][j] = matrix[i][top] + matrix[top][j]
            all_path[i][j] = top


@pprint_matrix
def floyd(n: int) -> tuple[list[list], list[list]]:
    """
    Функция, реализующая алгоритм Флойда
    :param n: количество вершин
    :return: матрицу смежности, где заполнены всевозможные пути,
```

```
    и матрицу, показывающей вершину, где проходит самый короткий
путь через 2 других
    """
    matrix = interact.create_matrix(n)
    length = len(matrix)
    all_path = [[0] * length for _ in range(length)]
    for top in range(n):
        threads = []
        for i in range(n):
            thread     =     threading.Thread(target=update_path,
args=(i, top, matrix, all_path))
            thread.start()
            threads.append(thread)
        for thread in threads:
            thread.join()
    for i in range(n):
        if matrix[i][i] < 0:
            raise   ValueError("Отрицательный   цикл   обнаружен!
Алгоритм Флойда работает некорректно!")
    return all_path, matrix


def get_path(shortest_vertex, end: int, start: int) -> tuple:
    """
    Функция, которая возвращает полный путь от конечной точки к
начальной.
    :param shortest_vertex: Матрица путей, которая показывает
вершину, соединяющую эти точки.
    :param end: Конечная точка.
    :param start: Начальная точка.
    """
    shortest_path_matrix, distance_matrix = shortest_vertex
    path, summary = [end], 0
    while end != start:

        if shortest_path_matrix[end][start] == 0:
            path.insert(0, start)
            summary += distance_matrix[end][start]
            break

        new_point = shortest_path_matrix[end][start]
        summary += distance_matrix[end][new_point]
        end = new_point
        path.insert(0, end)

    return [chr(65 + vertex) for vertex in path], summary
```

3 Решение задач, используя алгоритм Флойда

3.1 Решение первой задачи с acmp.ru



Школа программиста

14.06.2023, 22:52:23

Забыли пароль?

[задачи] [курсы] [олимпиады] [регистрация]

Логин: Пароль: Ok

ИНФОРМАЦИЯ
- О школе
- Правила
- Олимпиады
- Фотоальбом
- Гостевая
- Форум
- Архив олимпиад

ЗАДАЧНИК
- Архив задач
- Состояние системы
- Рейтинг
- Курсы

МЕТОДИЧКА
- Новичкам
- Работа в системе
- Курсы ККДП
- Дистрибутивы
- Статьи
- Ссылки

СТАТИСТИКА

YouTube

ЗАДАЧА №135

Алгоритм Флойда

(Время: 1 сек. Память: 16 Мб Сложность: 36%)

Полный ориентированный взвешенный граф задан матрицей смежности. Постройте матрицу кратчайших путей между его вершинами. Гарантируется, что в графе нет циклов отрицательного веса.

Входные данные

В первой строке входного файла INPUT.TXT записано единственное число N (1 ≤ N ≤ 100) - количество вершин графа. В следующих N строках по N чисел - матрица смежности графа (j-ое число в i-ой строке соответствует весу ребра из вершины i в вершину j). Все числа по модулю не превышают 100. На главной диагонали матрицы - всегда нули.

Выходные данные

В выходной файл OUTPUT.TXT выведите N строк по N чисел - матрицу кратчайших расстояний между парами вершин. j-ое число в i-ой строке должно быть равно весу кратчайшего пути из вершины i в вершину j.

Пример

№	INPUT.TXT	OUTPUT.TXT
1	4 0 5 9 100 100 0 2 8 100 100 0 7 4 100 100 0	0 5 7 13 12 0 2 8 11 16 0 7 4 9 11 0

Для отправки решения задачи необходимо зарегистрироваться и авторизоваться!

[Обсуждение] [Все попытки] [Лучшие попытки]

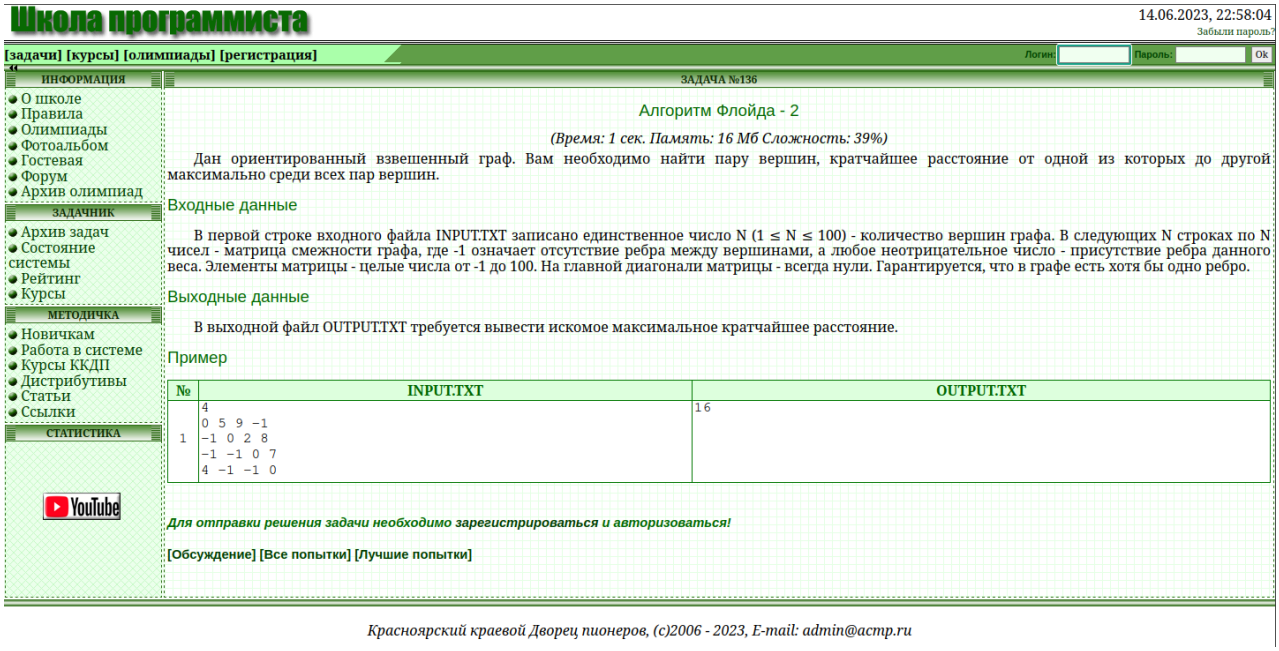
Рисунок 14 – условие первой задачи

```python
from itertools import product

# https://acmp.ru/index.asp?main=task&id_task=135

with open(r"data1.txt", mode='r', encoding='UTF-8') as input_file:
    matrix = [[int(x) for x in input_file.readline().split()] for _ in range(int(input_file.readline()))]

for const_top, i, j in product(range(len(matrix)), repeat=3):
    matrix[i][j] = min(matrix[i][j], matrix[i][const_top] + matrix[const_top][j])
    C3EQUALZ, Yesterday • first commit
[print(' '.join(str(x) for x in raw)) for raw in matrix]
```

Рисунок 15 – решение первой задачи

Ссылка на данную задачу: https://acmp.ru/index.asp?main=task&id_task=135

3.2 Решение второй задачи с acmp.ru



Школа программиста

14.06.2023, 22:58:04

Забыли пароль?

[задачи] [курсы] [олимпиады] [регистрация]

Логин: Пароль: Ok

ИНФОРМАЦИЯ
- О школе
- Правила
- Олимпиады
- Фотоальбом
- Гостевая
- Форум
- Архив олимпиад

ЗАДАЧНИК
- Архив задач
- Состояние системы
- Рейтинг
- Курсы

МЕТОДИЧКА
- Новичкам
- Работа в системе
- Курсы ККДП
- Дистрибутивы
- Статьи
- Ссылки

СТАТИСТИКА

YouTube

ЗАДАЧА №136

Алгоритм Флойда - 2

(Время: 1 сек. Память: 16 Мб Сложность: 39%)

Дан ориентированный взвешенный граф. Вам необходимо найти пару вершин, кратчайшее расстояние от одной из которых до другой максимально среди всех пар вершин.

Входные данные

В первой строке входного файла INPUT.TXT записано единственное число N (1 ≤ N ≤ 100) - количество вершин графа. В следующих N строках по N чисел - матрица смежности графа, где -1 означает отсутствие ребра между вершинами, а любое неотрицательное число - присутствие ребра данного веса. Элементы матрицы - целые числа от -1 до 100. На главной диагонали матрицы - всегда нули. Гарантируется, что в графе есть хотя бы одно ребро.

Выходные данные

В выходной файл OUTPUT.TXT требуется вывести искомое максимальное кратчайшее расстояние.

Пример

№	INPUT.TXT	OUTPUT.TXT
1	4 0 5 9 -1 -1 0 2 8 -1 -1 0 7 4 -1 -1 0	16

Для отправки решения задачи необходимо зарегистрироваться и авторизоваться!

[Обсуждение] [Все попытки] [Лучшие попытки]

Красноярский краевой Дворец пионеров, (c)2006 - 2023, E-mail: admin@acmp.ru

Рисунок 16 – условие второй задачи

```python
from itertools import product

# https://acmp.ru/index.asp?main=task&id_task=136

with open(r"data2.txt", mode='r', encoding='UTF-8') as input_file:
    matrix = [[int(x) for x in input_file.readline().split()] for _ in range(int(input_file.readline()))]

for const_top, i, j in product(range(len(matrix)), repeat=3):
    if matrix[i][j] < 0:
        matrix[i][j] = float('inf')
    matrix[i][j] = min(matrix[i][j], matrix[i][const_top] + matrix[const_top][j])

print(max((max(raw) for raw in matrix)))
```

Рисунок 17 – решение второй задачи

Ссылка на данную задачу: https://acmp.ru/index.asp?main=task&id_task=136

3.3 Решение третей задачи с leetcode

Description Editorial Solutions (801) Submissions

1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

Hint

Medium 2.1K 75

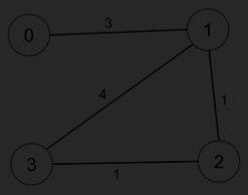
Companies

There are `n` cities numbered from `0` to `n-1`. Given the array `edges` where `edges[i] = [from_i, to_i, weight_i]` represents a bidirectional and weighted edge between cities `from_i` and `to_i`, and given the integer `distanceThreshold`.

Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** `distanceThreshold`, If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

Example 1:




```
Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4
Output: 3
Explanation: The figure above describes the graph.
The neighboring cities at a distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.
```

Example 2:

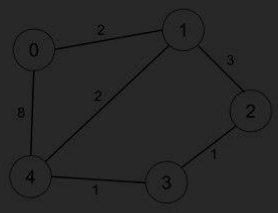


Рисунок 18 – условие третей задачи

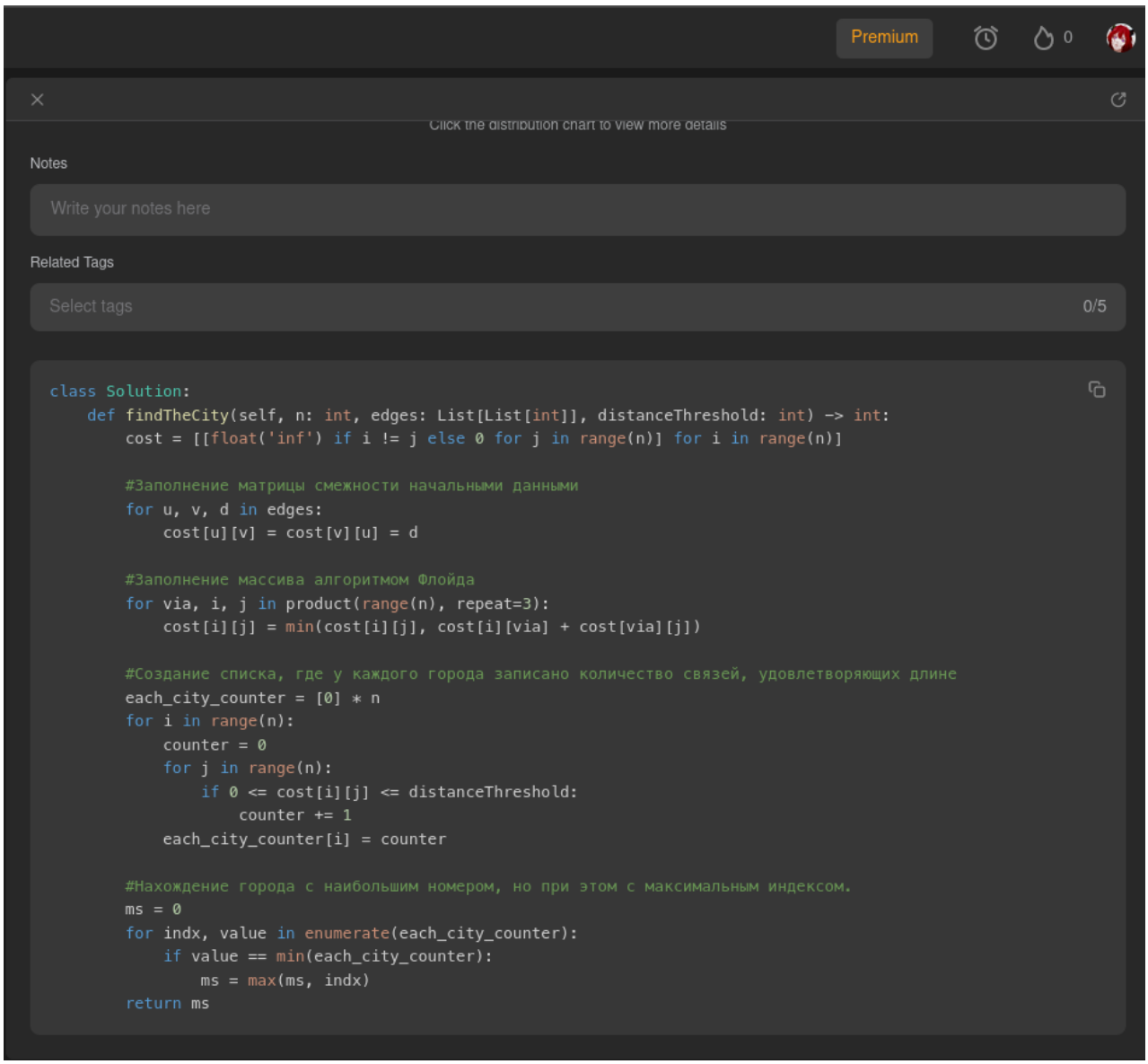


Рисунок 19 – решение третей задачи

Ссылка на данную задачу: https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/description/

Заключение

В результате проделанной работы, я изучил алгоритм Флойда и его применение для решения задач на графах. Алгоритм Флойда является эффективным инструментом для нахождения кратчайших путей между всеми парами вершин во взвешенном графе.

Во время изучения алгоритма Флойда я освоил его основные принципы и понял, как он работает. Алгоритм Флойда основан на пошаговом обновлении матрицы расстояний и матрицы путей до всех пар вершин. Он позволяет найти не только самые короткие пути между всеми парами вершин, но и информацию о промежуточных вершинах, через которые проходят эти пути.

В процессе реализации алгоритма Флойда, я обратил внимание на возможности оптимизации. Одной из таких оптимизаций является параллельное выполнение вычислений с использованием многопоточности. Это позволяет ускорить процесс обновления матрицы путей и сократить время выполнения алгоритма на больших графах.

Изучение и реализация алгоритма Флойда позволили мне расширить свои знания в области алгоритмов и графов. Я освоил принципы работы алгоритма, научился оптимизировать его и применять в различных задачах. Алгоритм Флойда является мощным инструментом для решения задач на графах и может быть полезным во многих областях, таких как маршрутизация, транспортная логистика, анализ социальных сетей и других.

Перечень использованных информационных ресурсов

1. Форум URL: ***https://qoo.su/Gqtrpo*** (дата обращения 13.06.2023)

2. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 2-е изд — М.: Издательский дом «Вильямс», 2009. — ISBN 978-5-8459-0857-5. (дата обращения 13.06.2023)

3. Романовский И. В. Дискретный анализ: Учебное пособие для студентов, специализирующихся по прикладной математике и информатике. Изд. 3-е. — СПб.: Невский диалект, 2003. — 320 с. — ISBN 5-7940-0114-3. (дата обращение 13.06.2023)