фСАНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3
по курсу «Алгоритмы и структуры данных»
Тема: Двоичные деревья поиска
Вариант 20

Выполнил:
Галилей Кирилл Дмитриевич
К3240

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Задачи по варианту

Задача №6. Количество пересадок

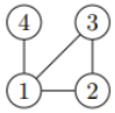
6 Задача. Количество пересадок [10 s, 512 Mb, 1 балл]

Вы хотите вычислить минимальное количество сегментов полета, чтобы добраться из одного города в другой. Для этого вы строите следующий неориентированный граф: вершины представляют города, между двумя вершинами есть ребро всякий раз, когда между соответствующими двумя городами есть перелет. Тогда достаточно найти кратчайший путь из одного из заданных городов в другой.

Дан *неориентироанный* граф с n вершинами и m ребрами, а также две вершины u и v, нужно посчитать длину кратчайшего пути между u и v (то есть, минимальное количество ребер в пути из u в v).

- **Формат ввода / входного файла (input.txt).** Нериентированный граф задан по формату 1. Следующая строка содержит две вершины u и v.
- **Ограничения на входные данные.** $2 \leq n \leq 10^5, 0 \leq m \leq 10^5, 1 \leq u, v \leq n, u \neq v$.
- **Формат вывода / выходного файла (output.txt).** Выведите минимальное количество ребер в пути из u в v. Выведите -1, если пути нет.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

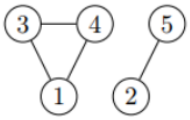
input	output
4 4 1 2 4 1 2 3 3 1 2 4	2



В этом графе существует единственный кратчайший путь между вершинами 2 и 4: 2 – 1 – 4.

- Пример 2:

input	output.txt
5 4 5 2 1 3 3 4 1 4 3 5	-1



В этом графе нет пути между вершинами 3 и 5.

Код программы

```
import time
import psutil
from collections import deque

def solve_task():
```

```
    # Чтение входных данных
    with open('input.txt', 'r') as file:
        n, m = map(int, file.readline().split())
        u, v = map(int, file.readline().split())

        # Создание графа
        graph = [[] for _ in range(n + 1)]
        for _ in range(m):
            a, b = map(int, file.readline().split())
            graph[a].append(b)
            graph[b].append(a)

    def bfs(start, end):
        visited = [False] * (n + 1)
        queue = deque([(start, 0)])
        visited[start] = True

        while queue:
            current, depth = queue.popleft()
            if current == end:
                return depth
            for neighbor in graph[current]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append((neighbor, depth + 1))
        return -1

    # Нахождение кратчайшего пути
    result = bfs(u, v)

    # Запись результата в выходной файл
    with open('output.txt', 'w') as file:
        file.write(str(result) + '\n')

if __name__ == "__main__":
    # Начало отсчета времени
    start_time = time.time()

    # Выполнение задачи
    solve_task()

    # Конец отсчета времени
    end_time = time.time()
```

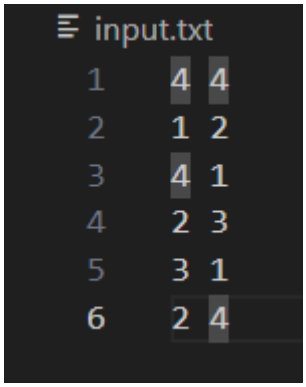
```
    # Подсчет использованного времени
    elapsed_time = end_time - start_time

    # Получение информации о текущем процессе
    process = psutil.Process()

    # Подсчет использованной памяти в байтах
    memory_usage = process.memory_info().rss

    print(f"Время выполнения: {elapsed_time:.2f} секунд")
    print(f"Использованная память: {memory_usage / (1024 * 1024):.2f}
МБ")
```

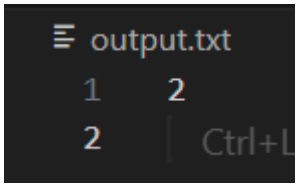
Текстовое объяснение решения

Основная логика программы заключена в функции solve_task. Сначала происходит чтение входных данных из файла input.txt. Считываются количество вершин n и количество ребер m, а также начальная (u) и конечная (v) вершины для поиска пути. Затем создается граф в виде списка смежности, и в него добавляются ребра, считанные из файла.

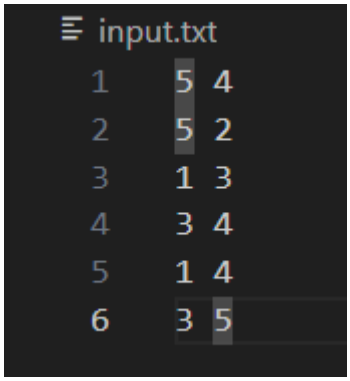
Для поиска кратчайшего пути используется вспомогательная функция bfs, которая принимает начальную и конечную вершины. В этой функции инициализируется список посещенных вершин и очередь для BFS. Пока очередь не пуста, извлекается текущая вершина и ее глубина. Если текущая вершина совпадает с конечной, возвращается глубина (длина пути). Для всех соседей текущей вершины, если они не посещены, они добавляются в очередь с увеличенной глубиной. Если путь не найден, возвращается -1.

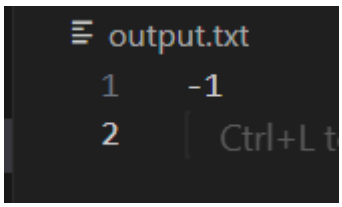
Результат работы кода на примерах из текста задачи:
1)





2)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.000599 секунд	15.16 МБ
Пример из задачи	0.000589 секунд	14.88 МБ

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №8. Стоимость полета.

8 Задача. Стоимость полета [10 s, 512 Mb, 1.5 балла]

Теперь вас интересует минимизация не количества пересадок, а общей стоимости полета. Для этого строится взвешенный граф: вес ребра из одного города в другой – это стоимость соответствующего перелета.

Дан ориентированный граф с положительными весами ребер, n - количество вершин и m - количество ребер, а также даны две вершины u и v. Вычислить вес кратчайшего пути между u и v (то есть минимальный общий вес пути из u в v).

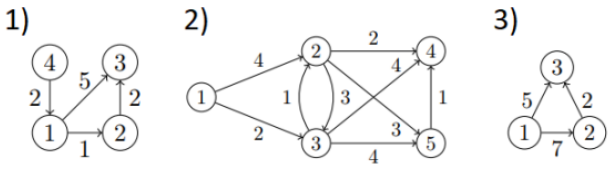
- **Формат ввода / входного файла (input.txt).** Ориентированный взвешенный граф задан по формату 1. Следующая строка содержит две вершины u и v.
- **Ограничения на входные данные.** $1 \le n \le 10^4$, $0 \le m \le 10^5$, $1 \le u, v \le n$, $u \ne v$, вес каждого ребра – неотрицательное целое число, не превосходящее 10^8.
- **Формат вывода / выходного файла (output.txt).** Выведите минимальный вес пути из u в v. Веведите -1, если пути нет.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Примеры:

input	output
4 4 1 2 1 4 1 2 2 3 2 1 3 5 1 3	3

input	output
5 9 1 2 4 1 3 2 2 3 2 3 2 1 2 4 2 3 5 4 5 4 1 2 5 3 3 4 4 1 5	6

input	output
3 3 1 2 7 1 3 5 2 3 2 3 2	-1

- Объяснения:



Пример 1 – В этом графе существует единственный кратчайший путь из вершины 1 в вершину 3 ($1 \to 2 \to 3$), и он имеет вес 3. Пример 2 – Есть два пути от 1 до 5 общего веса 6: $1 \to 3 \to 5$ и $1 \to 3 \to 2 \to 5$. Пример 3 – Нет пути от вершины 3 до 2.

Код программы

```
import time
import psutil
import heapq

def solve_task():
    # Чтение входных данных
    with open('input.txt', 'r') as file:
        n, m = map(int, file.readline().split())
        u, v = map(int, file.readline().split())
```

```
        # Создание графа
        graph = [[] for _ in range(n + 1)]
        for _ in range(m):
            a, b, w = map(int, file.readline().split())
            graph[a].append((b, w))

    # Поиск кратчайшего пути с помощью алгоритма Дейкстры
    def dijkstra(start, end):
        distances = [float('inf')] * (n + 1)
        distances[start] = 0
        priority_queue = [(0, start)]

        while priority_queue:
            current_distance,              current_vertex              =
heapq.heappop(priority_queue)

            if current_distance > distances[current_vertex]:
                continue

            for neighbor, weight in graph[current_vertex]:
                distance = current_distance + weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))

        return distances[end] if distances[end] != float('inf') else -1

    # Нахождение кратчайшего пути
    result = dijkstra(u, v)

    # Запись результата в выходной файл
    with open('output.txt', 'w') as file:
        file.write(str(result) + '\n')

if __name__ == "__main__":
    # Начало отсчета времени
    start_time = time.time()

    # Выполнение задачи
    solve_task()
```

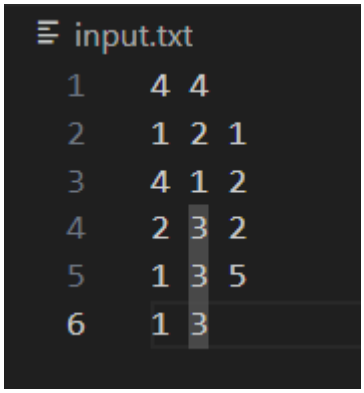
```
    # Конец отсчета времени
    end_time = time.time()

    # Подсчет использованного времени
    elapsed_time = end_time - start_time

    # Получение информации о текущем процессе
    process = psutil.Process()

    # Подсчет использованной памяти в байтах
    memory_usage = process.memory_info().rss

    print(f"Время выполнения: {elapsed_time:.2f} секунд")
    print(f"Использованная память: {memory_usage / (1024 * 1024):.2f}
МБ")
```

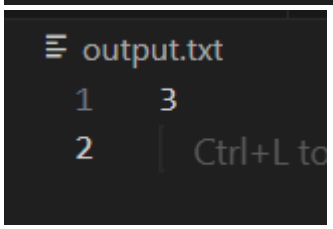
Текстовое объяснение решения

Для поиска кратчайшего пути используется вспомогательная функция dijkstra, которая принимает начальную и конечную вершины. В этой функции инициализируется список расстояний до всех вершин, устанавливая начальное расстояние до стартовой вершины равным нулю. Создается приоритетная очередь, в которую добавляется стартовая вершина с расстоянием 0. Пока очередь не пуста, извлекается вершина с наименьшим текущим расстоянием. Если текущее расстояние больше уже известного расстояния до этой вершины, то вершина пропускается. Для всех соседей текущей вершины вычисляется новое расстояние, и если оно меньше уже известного, то обновляется расстояние и сосед добавляется в очередь. В конце функция возвращает расстояние до конечной вершины или -1, если путь не найден.

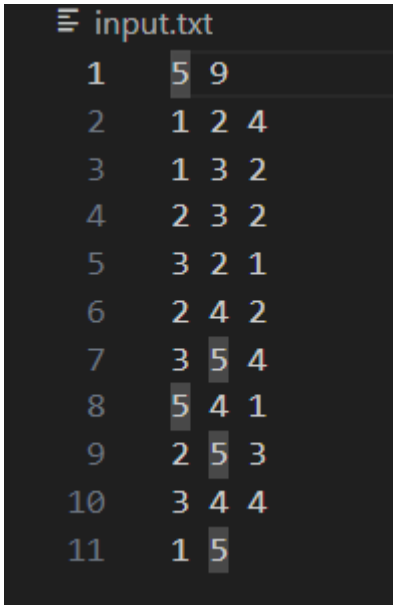
Результат работы кода на примерах из текста задачи:

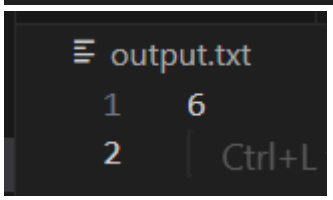
1)





2)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.000574 секунд	15.16 МБ

	0.000584 секунд	15.13 МБ

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №17. Слабая K-свяанность

17 Задача. Слабая K-связность [1 s, 16 Mb, 4 балла]

Ане, как будущей чемпионке мира по программированию, поручили очень ответственное задание. Правительство вручает ей план постройки дорог между N городами. По плану все дороги односторонние, но между двумя городами может быть больше одной дороги, возможно, в разных направлениях. Ане необходимо вычислить минимальное такое K, что данный ей план является слабо K-связным.

Правительство называет план слабо K-связным, если выполнено следующее условие: для любых двух различных городов можно проехать от одного до другого, нарушая правила движения не более K раз. Нарушение правил - это проезд по существующей дороге в обратном направлении. Гарантируется, что между любыми двумя городами можно проехать, возможно, несколько раз нарушив правила.

- **Формат входных данных (input.txt) и ограничения.** В первой строке входного файла INPUT.TXT записаны два числа $2 \le N \le 300$ и $1 \le M \le 10^5$ - количество городов и дорог в плане. В последующих M строках даны по два числа - номера городов, в которых начинается и заканчивается соответствующая дорога.
- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT выведите минимальное K, такое, что данный во входном файле план является слабо K-связным.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.
- Примеры:

input.txt	output.txt
3 2 1 2 1 3	1

input.txt	output.txt
4 4 2 4 1 3 4 1 3 2	0

- Проверяем **обязательно** – на acmp.

Код программы

```
import sys
import timeit
import psutil


def solve():
    with open('input.txt', 'r') as f:
        n, m = map(int, f.readline().strip().split())
        dist = [[float('inf')] * n for _ in range(n)]

        for _ in range(m):
```

```
            u, v = map(int, f.readline().strip().split())
            dist[u-1][v-1] = 0  # Прямая дорога
            dist[v-1][u-1] = 1  # Обратная дорога (нарушение правила)

        # Алгоритм Флойда-Уоршелла
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if dist[i][j] > dist[i][k] + dist[k][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

        # Найти максимальное значение K
        max_k = 0
        for i in range(n):
            for j in range(n):
                if i != j:
                    max_k = max(max_k, dist[i][j])

        with open('output.txt', 'w') as f:
            f.write(str(max_k) + '\n')


if __name__ == "__main__":
    start_time = timeit.default_timer()
    solve()
    end_time = timeit.default_timer()
    elapsed_time = end_time - start_time
    process = psutil.Process()
    memory_usage = process.memory_info().rss
    print(f"Время выполнения: {elapsed_time:.6f} секунд")
    print(f"Использованная память: {memory_usage / (1024 * 1024):.2f}
МБ")
```

Текстовое объяснение решения

Программа считывает количество городов и дорог из файла input.txt.

Инициализирует матрицу расстояний, устанавливая бесконечность для всех пар городов, кроме диагональных элементов.

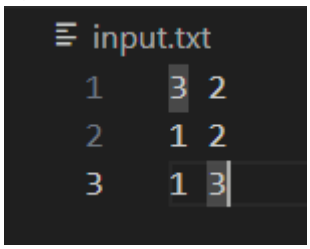
Заполняет матрицу расстояний на основе входных данных, устанавливая 0 для прямых дорог и 1 для обратных дорог.

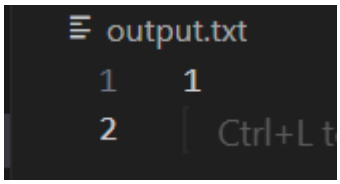
Применяет алгоритм Флойда-Уоршелла для нахождения минимального количества нарушений правил движения между всеми парами городов.

Находит максимальное значение нарушений и записывает его в файл output.txt, а также выводит время выполнения и использование памяти.

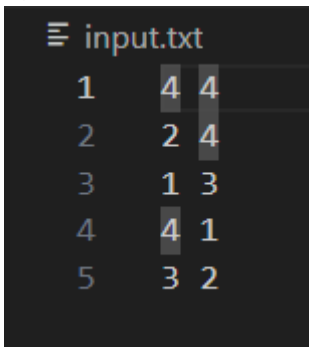
Результат работы кода на примерах из текста задачи:

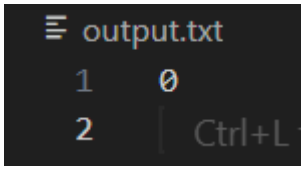
1)





2)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.002188 секунд	0.02
Пример из задачи	0.001862 секунд	0.02

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти
раничения по времени и памяти

Вывод

В ходе данной лабораторной работы я научился решать задачи. Написанные программы были протестированы, а также были измерены потребляемый ими объём памяти и время работы. Все программы работаю корректно и укладываются в установленные ограничения по времени и памяти на примерах из задач.