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:

Смирнов Георгий Валерьевич

К3139

Проверил:

Афанасьев А.В.

Санкт-Петербург

2024 г.

[**Задачи по варианту**](#_gjdgxs) **2**

[Задача №6. Количество пересадок.](#_30j0zll) 2

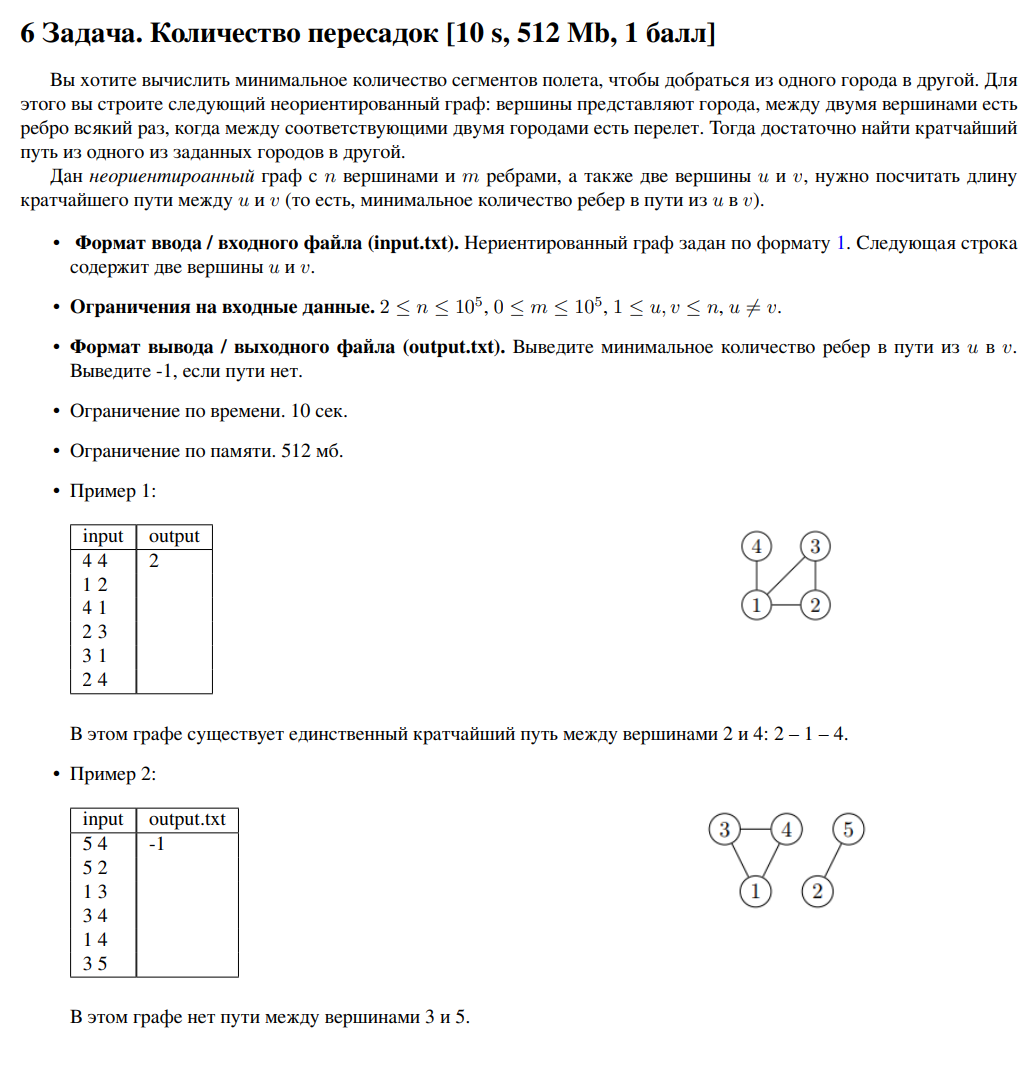
[Задача №8. Стоимость полёта.](#_1fob9te) 7

[Задача №17. Слабая K-связность.](#_3znysh7) 11

[**Вывод**](#_2et92p0) **14**

# Задачи по варианту

## Задача №6. Количество пересадок



**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)

# Поиск кратчайшего пути с помощью BFS

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")

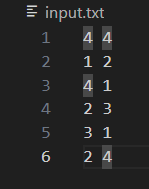
**Текстовое объяснение решения**

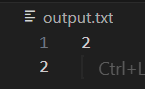
Основная логика программы заключена в функции solve\_task. Сначала происходит чтение входных данных из файла input.txt. Считываются количество вершин n и количество ребер m, а также начальная (u) и конечная (v) вершины для поиска пути. Затем создается граф в виде списка смежности, и в него добавляются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

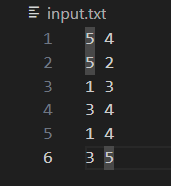
Результат работы кода на примерах из текста задачи:

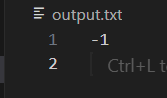
1)





2)

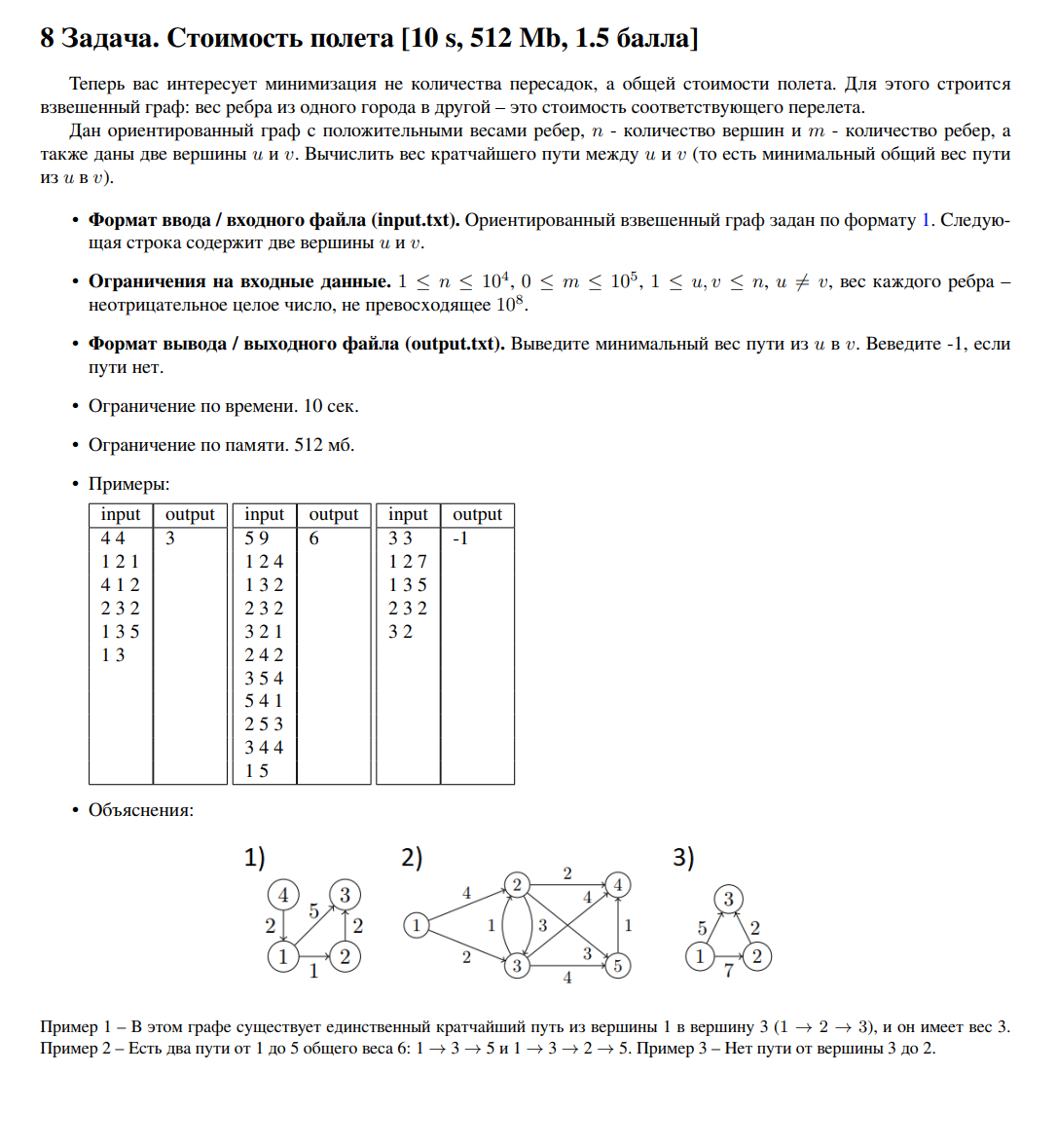




|  | Время выполнения, с | Затраты памяти, МБ |
| --- | --- | --- |
| Пример из задачи | 0.000599 секунд | 15.16 МБ |
| Пример из задачи | 0.000589 секунд | 14.88 МБ |

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

## Задача №8. Стоимость полета.



**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")

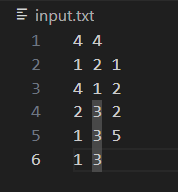
print(f"Использованная память: {memory\_usage / (1024 \* 1024):.2f} МБ")

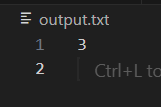
**Текстовое объяснение решения**

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.

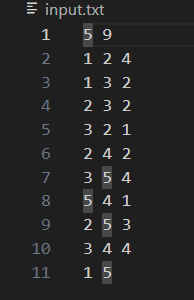
Результат работы кода на примерах из текста задачи:

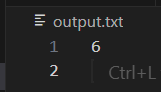
1)





2)

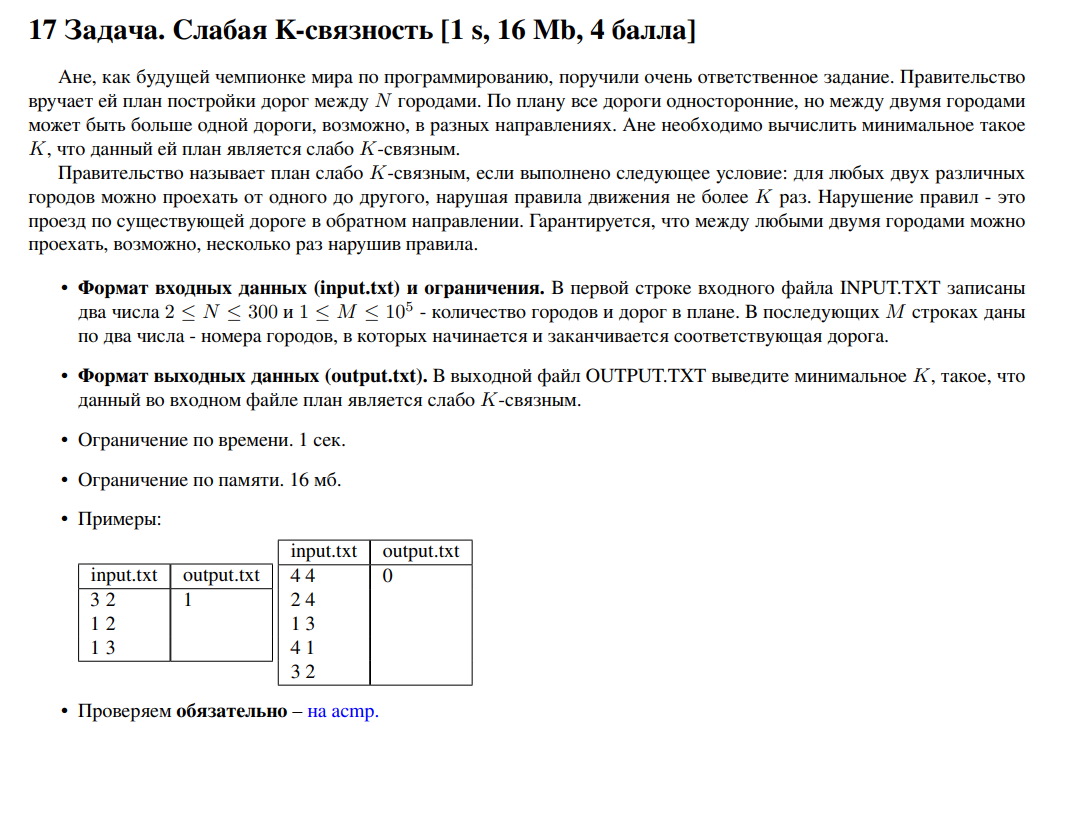




|  | Время выполнения, с | Затраты памяти, МБ |
| --- | --- | --- |
| Пример из задачи | 0.000574 секунд | 15.16 МБ |
|  | 0.000584 секунд | 15.13 МБ |

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

## Задача №17. Слабая K-свяанность



**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.

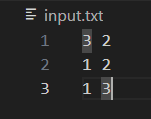
Заполняет матрицу расстояний на основе входных данных, устанавливая 0 для прямых дорог и 1 для обратных дорог.

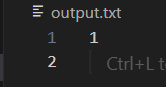
Применяет алгоритм Флойда-Уоршелла для нахождения минимального количества нарушений правил движения между всеми парами городов.

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

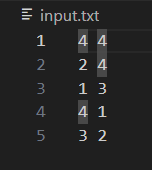
Результат работы кода на примерах из текста задачи:

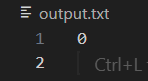
1)





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.