

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

Отчет по лабораторной работе №3
по курсу «Алгоритмы и структуры данных»

Тема: Графы

Вариант 1

Выполнила:

Гайдук А. С.

К3241

Проверила:

Ромакина О. М.

Санкт-Петербург

2025 г.

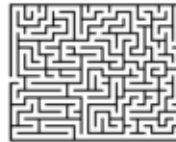
Содержание отчета

Содержание отчета.....	2
Задачи по варианту	3
Задача №1. Лабиринт [5 s, 512 Mb, 1 балл]	3
Задача №5. Город с односторонним движением [5 s, 512 Mb, 1.5 балла].	6
Задача №17. Слабая k-связность [1 s, 16 Mb, 4 балла]	10
Дополнительные задачи	14
Задача №2. Компоненты [5 s, 512 Mb, 1 балл]	14
Задача №3. Циклы [5 s, 512 Mb, 1 балл]	16
Задача №6. Количество пересадок [10 s, 512 Mb, 1 балл]	20
Задача №8. Стоимость полета [10 s, 512 Mb, 1.5 балла]	23
Задача №10. Оптимальный обмен валюты [10 s, 512 Mb, 2 балла]	27
Задача №11. Алхимия [1 s, 16 Mb, 3 балла]	31
Задача №14. Автобусы [1 s, 16 Mb, 3 балла]	35
Вывод	39

Задачи по варианту

Задача №1. Лабиринт [5 s, 512 Mb, 1 балл]

Лабиринт представляет собой прямоугольную сетку ячеек со стенками между некоторыми соседними ячейками. Вы хотите проверить, существует ли путь от данной ячейки к данному выходу из лабиринта, где выходом также является ячейка, лежащая на границе лабиринта (в примере, показанном на рисунке, есть два выхода: один на левой границе и один на правой границе). Для этого вы представляете лабиринт в виде неориентированного графа: вершины графа являются ячейками лабиринта, две вершины соединены неориентированным ребром, если они смежные и между ними нет стены. Тогда, чтобы проверить, существует ли путь между двумя заданными ячейками лабиринта, достаточно проверить, что существует путь между соответствующими двумя вершинами в графе.



Вам дан неориентированный граф и две различные вершины u и v . Проверьте, есть ли путь между u и v .

- **Формат ввода / входного файла (input.txt).** Неориентированный граф с n вершинами и m ребрами по формату 1. Следующая строка после ввода всего графа содержит две вершины u и v .
- **Ограничения на входные данные.** $2 \leq n \leq 10^3$, $1 \leq m \leq 10^3$, $1 \leq u, v \leq n$, $u \neq v$.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если есть путь между вершинами u и v ; выведите 0, если пути нет.
- **Ограничение по времени.** 5 сек.
- **Ограничение по памяти.** 512 мб.

Листинг кода:

```
import tracemalloc
import time
import sys
sys.setrecursionlimit(10**6)

tracemalloc.start()

def dfs(graph, start, end, visited):
    if start == end:
        return True
    visited[start] = True
    for neighbor in graph[start]:
        if not visited[neighbor]:
            if dfs(graph, neighbor, end, visited):
                return True
    return False

with open('input.txt', 'r') as file:
    n, m = map(int, file.readline().split())
    graph = {i: [] for i in range(1, n + 1)}
    for _ in range(m):
        u, v = map(int, file.readline().split())
        graph[u].append(v)
        graph[v].append(u)
    u, v = map(int, file.readline().split())
```

```

start_time = time.perf_counter()

visited = {i: False for i in range(1, n + 1)}
result = 1 if dfs(graph, u, v, visited) else 0

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Для решения этого задания я использовала алгоритм поиска в глубину для обхода графа. На вход функция dfs принимает граф в виде словаря (ключи – вершины, значения – списки смежных вершин), вершину, с которой начинается поиск, целевую вершину и словарь для отслеживания посещенных вершин. Если текущая вершина – целевая, путь найден. Иначе помечаем текущую вершину как посещенную и перебираем соседей текущей вершины. Если соседняя вершина еще не посещена, рекурсивно вызываем dfs для неё. Если рекурсивный вызов вернул True, значит путь найден.

Результат работы кода на примерах из текста задачи:

The image shows two screenshots of a code editor interface. The top screenshot displays the 'input.txt' file with the following content:

1	4 4
2	1 2
3	3 2
4	4 3
5	1 4
6	1 4

The bottom screenshot displays the 'output.txt' file with the following content:

1	1
---	---

```
task1.py  input.txt  output.txt
1 4 2
2 1 2
3 3 2
4 1 4
```

```
task1.py  input.txt  output.txt
1 1
```

Результат работы кода на максимальных и минимальных значениях:

```
task1.py  input.txt  output.txt
1 1000 1000
2 1 2
3 2 3
4 3 4
5 4 5
6 5 6
```

```
task1.py  input.txt  output.txt
1 1
```

```
task1.py  input.txt  output.txt
1 2 1
2 1 2
3 1 1
4 1 2
```

```
task1.py  input.txt  output.txt
1 1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000009 сек	0.037456 МВ
Пример из задачи	0.000009 сек	0.037929 МВ
Пример из задачи	0.000010 сек	0.037887 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.003364 сек	0.321070 МВ

Вывод по задаче:

Я реализовала алгоритм поиска в глубину для решения задачи на нахождение пути между вершинами графа.

Задача №5. Город с односторонним движением [5 s, 512 Mb, 1.5 балла]

Департамент полиции города сделал все улицы односторонними. Вы хотели бы проверить, можно ли законно проехать с любого перекрестка на какой-либо другой перекресток. Для этого строится ориентированный граф: вершины – это перекрестки, существует ребро (u, v) всякий раз, когда в городе есть улица (с односторонним движением) из u в v . Тогда достаточно проверить, все ли вершины графа лежат в одном компоненте сильной связности.

Нужно вычислить количество компонентов сильной связности заданного ориентированного графа с n вершинами и m ребрами.

- **Формат ввода / входного файла (input.txt).** Ориентированный граф с n вершинами и m ребрами по формату 1.
- **Ограничения на входные данные.** $1 \leq n \leq 10^4$, $0 \leq m \leq 10^4$.
- **Формат вывода / выходного файла (output.txt).** Выведите число – количество компонентов сильной связности.
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.

Листинг кода:

```
import tracemalloc
import time
import sys

sys.setrecursionlimit(10 ** 6)
tracemalloc.start()

def dfs(node, visited, graph, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs(neighbor, visited, graph, stack)
```

```

stack.append(node)

def reverse_graph(graph, n):
    reversed_graph = {i: [] for i in range(1, n + 1)}
    for u in graph:
        for v in graph[u]:
            reversed_graph[v].append(u)
    return reversed_graph

def count_scc(graph, n):
    visited = {i: False for i in range(1, n + 1)}
    stack = []
    for i in range(1, n + 1):
        if not visited[i]:
            dfs(i, visited, graph, stack)
    reversed_graph = reverse_graph(graph, n)
    visited = {i: False for i in range(1, n + 1)}
    count = 0
    while stack:
        node = stack.pop()
        if not visited[node]:
            dfs(node, visited, reversed_graph, [])
            count += 1

    return count

with open('input.txt', 'r') as file:
    n, m = map(int, file.readline().split())
    graph = {i: [] for i in range(1, n + 1)}
    for _ in range(m):
        u, v = map(int, file.readline().split())
        graph[u].append(v)

start_time = time.perf_counter()

result = count_scc(graph, n)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое  
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала функцию поиска в глубину, работающую как в задаче №1, однако теперь после обработки всех соседей текущая вершина добавляется в стек. Функция `reverse_graph` нужна для транспонирования графа (т. е. изменения направления всех ребер). Здесь я создаю пустой граф для транспонированного результата и перебираю все вершины и всех соседей

каждой вершины исходного графа. В транспонированный граф добавляю ребро в обратном направлении.

Функция `count_scc` считает компоненты сильной связности. Сначала я создаю словарь `visited` для отслеживания посещенных вершин и стек для хранения вершин в порядке завершения обхода. Перебираю все вершины, и, если вершина еще не посещена, вызываю `dfs`. После посещения всех вершин транспонирую граф, сбрасываю словарь `visited` и создаю счетчик компонентов. Пока стек не пуст, извлекаем вершины из него. Если она еще не посещена, вызываю `dfs` на транспонированном графе и увеличиваю счетчик компонентов сильной связности.

Первый проход `dfs` нужен для получения порядка, в котором вершины заканчиваются при обходе. Вторым проходом нужен для нахождения компонентов связности, используя порядок завершения вершин, полученный на первом шаге.

Результат работы кода на примерах из текста задачи:

task5.py	input.txt	output.txt
1	4 4	
2	1 2	
3	4 1	
4	2 3	
5	3 1	

task5.py	input.txt	output.txt
1	2	

task5.py	input.txt	output.txt
1	5 7	
2	2 1	
3	3 2	
4	3 1	
5	4 3	
6	4 1	
7	5 2	
8	5 3	


```
task5.py  input.txt  output.txt x
1  5
```

Результат работы кода на максимальных и минимальных значениях:

```
task5.py  input.txt x  output.txt
1  10000 10000
2  1 2
3  2 3
4  3 1
5  4 5
```

```
task5.py  input.txt  output.txt x
1  9926
```

```
task5.py  input.txt x  output.txt
1  1 0
2
```

```
task5.py  input.txt  output.txt x
1  1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000018 сек	0.037776 МВ
Пример из задачи	0.000040 сек	0.039012 МВ
Пример из задачи	0.000021 сек	0.039059 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.041052 сек	4.492733 МВ

Вывод по задаче:

Я реализовала алгоритм для подсчета количества компонентов сильной связности, используя два обхода DFS и транспонирование графа.

Задача №17. 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 $2 \leq N \leq 300$ и $1 \leq M \leq 10^5$ - 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.

Листинг кода:

```
import tracemalloc
import time

tracemalloc.start()

def floyd_warshall(graph, n):
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        dist[i][i] = 0

    for u in range(n):
        for v, w in graph[u]:
            dist[u][v] = min(dist[u][v], w)

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

def is_weakly_k_connected(dist, n, k):
    for u in range(n):
        for v in range(n):
            if u != v and dist[u][v] > k:
                return False
    return True
```

```

with open('input.txt', 'r') as file:
    n, m = map(int, file.readline().split())
    graph = [[] for _ in range(n)]
    for _ in range(m):
        u, v = map(int, file.readline().split())
        graph[u - 1].append((v - 1, 0))
        graph[v - 1].append((u - 1, 1))

start_time = time.perf_counter()

dist = floyd_warshall(graph, n)

low, high = 0, n
while low < high:
    mid = (low + high) // 2
    if is_weakly_k_connected(dist, n, mid):
        high = mid
    else:
        low = mid + 1

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(low))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я использовала алгоритм Флойда-Уоршала для вычисления кратчайших путей между всеми парами вершин в графе. Одноименная функция инициализирует матрицу расстояний. Изначально все значения являются недостижимыми (бесконечность). Далее я перебираю все вершины и рассматриваю базовый случай: расстояние вершины до самой себя равно нулю.

Снова перебираю все вершины и соседей каждой вершины с весом ребра w (вес ребра 0 \Rightarrow можно проехать без нарушения правил, прямое ребро; вес ребра 1 \Rightarrow обратное ребро, с нарушением правил). Обновляю расстояние между вершиной и её соседом, если найден более короткий путь.

Далее я перебираю все вершины как промежуточные, начальные и конечные. Проверяю, существует ли путь через промежуточную вершину, если да и он более короткий, то обновляю расстояние между начальной и конечной вершиной. Возвращаю матрицу кратчайших расстояний.

Функция `is_weakly_k_connected` проверяет, является ли граф слабо K -связным. Перебирая все вершины как начальные и конечные, если

расстояние между вершинами больше максимального количества нарушений, то граф не является слабо K -связным. Если все расстояния не превышают K , то возвращаем True.

После чтения файла с входными данными я инициализирую граф в виде списка смежности и перебираю все ребра. Считываю вершины, соединенные ребром, и добавляю ребра и их вес (1 или 0). После создания матрицы кратчайших расстояний инициализирую границы бинарного поиска (low – минимальное возможное K , high – максимальное). Выполняю поиск, пока границы не сойдутся: вычисляю середину диапазона и вызываю функцию для проверки графа на слабо mid -связность. Если проверка успешна, сужаю диапазон до $[low, mid]$. Иначе, сужаю диапазон до $[mid + 1, high]$. В файл записываю минимальное K , при котором граф является слабо K -связным.

Результат работы кода на примерах из текста задачи:

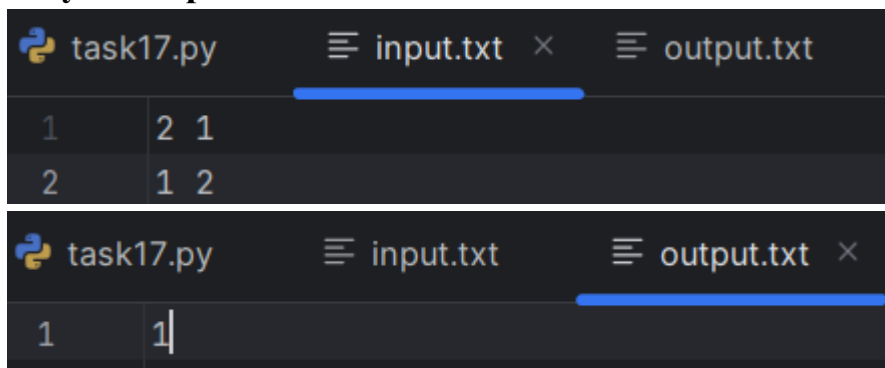
```
task17.py  input.txt  output.txt
1 3 2
2 1 2
3 1 3
```

```
task17.py  input.txt  output.txt
1 1
```

```
task17.py  input.txt  output.txt
1 4 4
2 2 4
3 1 3
4 4 1
5 3 2
```

```
task17.py  input.txt  output.txt
1 0
```

Результат работы кода на максимальных и минимальных значениях:



```
task17.py  input.txt  output.txt
1 2 1
2 1 2

task17.py  input.txt  output.txt
1 1
```

Проверка задачи на астр:

22732105	12.01.2025 3:09:20	Гайдук Аллина Сергеевна	0562	Python	Time limit exceeded	30	1,093	734 Kб
----------	--------------------	-------------------------	------	--------	---------------------	----	-------	--------

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000069 сек	0.037845 МВ
Пример из задачи	0.000102 сек	0.037914 МВ
Пример из задачи	0.000096 сек	0.037988 МВ
Верхняя граница диапазона значений входных данных из текста задачи	26.655079 сек	14.543669 МВ

Вывод по задаче:

Я реализовала алгоритм для определения слабой К-связности, используя алгоритм Флойда-Уоршала и бинарный поиск.

Дополнительные задачи

Задача №2. Компоненты [5 s, 512 Mb, 1 балл]

Теперь вы решаете сделать так, чтобы в лабиринте не было мертвых зон, то есть чтобы из каждой клетки был доступен хотя бы один выход. Для этого вы находите связные компоненты соответствующего неориентированного графа и следите за тем, чтобы каждый компонент содержал выходную ячейку.

Дан неориентированный граф с n вершинами и m ребрами. Нужно посчитать количество компонент связности в нем.

Листинг кода:

```
import tracemalloc
import time
import sys
sys.setrecursionlimit(10**6)

tracemalloc.start()

def dfs(graph, visited, vertex):
    visited[vertex] = True
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            dfs(graph, visited, neighbor)

def count_connected_components(graph):
    n = len(graph)
    visited = [False] * n
    count = 0
    for vertex in range(n):
        if not visited[vertex]:
            count += 1
            dfs(graph, visited, vertex)
    return count

with open('input.txt', 'r') as file:
    n, m = map(int, file.readline().split())
    graph = [[] for _ in range(n)]
    for _ in range(m):
        u, v = map(int, file.readline().split())
        graph[u - 1].append(v - 1)
        graph[v - 1].append(u - 1)

start_time = time.perf_counter()

component_count = count_connected_components(graph)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(component_count))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое  
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

Текстовое объяснение решения:

Для решения этой задачи я адаптировала алгоритм поиска в глубину, реализованный для задачи №1. Также я реализовала функцию для подсчета компонент связности. Она определяет количество вершин в графе, создает список для отслеживания посещенных вершин и счетчик компонент связности. Затем она проходит по всем вершинам графа и проверяет, была ли вершина посещена. Если нет, то к счетчику добавляется единица и вызывается DFS для текущей вершины.

Результат работы кода на примерах из текста задачи:

```
task2.py  input.txt  output.txt
1 4 2
2 1 2
3 3 2
```

```
task2.py  input.txt  output.txt
1 2
```

Результат работы кода на максимальных и минимальных значениях:

```
task2.py  input.txt  output.txt
1 1000 1000
2 1 2
3 2 3
4 3 4
5 4 5
```

```
task2.py  input.txt  output.txt
1 1
```

```
task2.py  input.txt  output.txt
1 1 0
```

```
task2.py  input.txt  output.txt
1 1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000004 сек	0.037728 МВ
Пример из задачи	0.000009 сек	0.037914 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.002952 сек	0.222731 МВ

Вывод по задаче:

Я адаптировала свой алгоритм для поиска в глубину и реализовала подсчет компонент связности.

Задача №3. Циклы [5 s, 512 Мб, 1 балл]

Учебная программа по инфокоммуникационным технологиям определяет пререквизиты для каждого курса в виде списка курсов, которые необходимо пройти перед тем, как начать этот курс. Вы хотите выполнить проверку согласованности учебного плана, то есть проверить отсутствие циклических зависимостей. Для этого строится следующий ориентированный граф: вершины соответствуют курсам, есть направленное ребро (u, v) – курс u следует пройти перед курсом v . Затем достаточно проверить, содержит ли полученный граф цикл.

Проверьте, содержит ли данный граф циклы.

- **Формат ввода / входного файла (input.txt).** Ориентированный граф с n вершинами и m ребрами по формату 1.
- **Ограничения на входные данные.** $1 \leq n \leq 10^3, 0 \leq m \leq 10^3$.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если данный граф содержит цикл; выведите 0, если не содержит.
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.

Листинг кода:

```
import tracemalloc
import time
import sys
sys.setrecursionlimit(10**6)

def dfs(v, visited, rec_stack, graph):
    visited[v] = True
    rec_stack[v] = True

    for neighbor in graph.get(v, []):
        if not visited[neighbor]:
            if dfs(neighbor, visited, rec_stack, graph):
                return True

    rec_stack[v] = False
    return False
```



```

        elif rec_stack[neighbor]:
            return True

    rec_stack[v] = False
    return False

def is_cyclic(graph, n):
    visited = {v: False for v in range(1, n + 1)}
    rec_stack = {v: False for v in range(1, n + 1)}
    for node in range(1, n + 1):
        if not visited[node]:
            if dfs(node, visited, rec_stack, graph):
                return 1
    return 0

with open('input.txt', 'r') as file:
    lines = file.readlines()
    n, m = map(int, lines[0].split())
    graph = {}
    for line in lines[1:m + 1]:
        u, v = map(int, line.split())
        if u not in graph:
            graph[u] = []
        graph[u].append(v)

start_time = time.perf_counter()

result = is_cyclic(graph, n)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое  
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Для решения этой задачи я создала рекурсивную функцию `dfs`, которая принимает на вход вершину, словарь для посещенных вершин, сам граф и словарь, который отслеживает вершины, находящиеся в текущем пути обхода для обнаружения циклов. Перебираю всех соседей текущей вершины, и, если вершина отсутствует в графе, используется пустой список. Если соседняя вершина еще не была посещена, вызываем `dfs` для нее. Если соседняя вершина уже находится в текущем пути обхода, то цикл обнаружен, возвращаем `True`. После завершения обработки вершины `v` она удаляется из текущего пути обхода. Если цикл не обнаружен, возвращаю `False`.

Функция `is_cyclic` нужна для проверки наличия циклов в графе. Инициализируются словари `visited`, `rec_stack` и начинается перебор всех вершин. Если вершина еще не посещена, вызываю `dfs` для нее, если `dfs` возвращает `True`, то цикл обнаружен, возвращаю 1.

Результат работы кода на примерах из текста задачи:

```
task3.py  input.txt  output.txt
1      5 7
2      1 2
3      2 3
4      1 3
5      3 4
6      1 4
7      2 5
8      3 5
```

```
task3.py  input.txt  output.txt
1      b
```

```
task3.py  input.txt  output.txt
1      4 4
2      1 2
3      4 1
4      2 3
5      3 1
```

```
task3.py  input.txt  output.txt
1      1
```

Результат работы кода на максимальных и минимальных значениях:

```
task3.py  input.txt  output.txt
1      1 0
```

```
task3.py  input.txt  output.txt x
1 0
```

```
task3.py  input.txt x  output.txt
1 1000 1000
2 1 2
3 2 3
```

```
task3.py  input.txt  output.txt x
1 1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000017 сек	0.037756 МВ
Пример из задачи	0.000043 сек	0.038071 МВ
Пример из задачи	0.000019 сек	0.037936 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.008450 сек	0.414600 МВ

Вывод по задаче:

Я реализовала алгоритм для определения циклов в графе, используя поиск в глубину.

Задача №6. 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 неориентированный 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).** Неориентированный 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.

Листинг кода:

```
import tracemalloc
import time
from collections import deque

tracemalloc.start()

def bfs_shortest_path(n, graph, start, end):
    if start == end:
        return 0
    visited = [False] * (n + 1)
    queue = deque()
    queue.append((start, 0))
    visited[start] = True
    while queue:
        current, distance = queue.popleft()
        for neighbor in graph[current]:
            if neighbor == end:
                return distance + 1
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append((neighbor, distance + 1))
    return -1

with open('input.txt', 'r') as file:
    lines = file.readlines()
    n, m = map(int, lines[0].split())
    graph = {i: [] for i in range(1, n + 1)}
    for line in lines[1:m + 1]:
        u, v = map(int, line.split())
        graph[u].append(v)
        graph[v].append(u)
    u, v = map(int, lines[m + 1].split())

start_time = time.perf_counter()

result = bfs_shortest_path(n, graph, u, v)

end_time = time.perf_counter()
```

```

current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала функцию для нахождения кратчайшего пути между вершинами start и end в графе. Функция принимает количество вершин, граф в виде словаря, начальную и конечную вершины. Если конечная и начальная вершины совпадают, возвращаю 0. Создаю список для отслеживания посещенных вершин и инициализирую очередь для BFS. Добавляю в очередь вершину start и расстояние до неё – 0, после чего помечаю её как посещенную. Пока очередь не пуста, извлекаю вершину и расстояние до неё из очереди. Перебираю всех соседей текущей вершины, если сосед – конечная вершина, возвращаю расстояние до неё. Если соседняя вершина не была посещена, отмечаю её как посещенную и добавляю в очередь с увеличенным расстоянием.

Результат работы кода на примерах из текста задачи:

task6.py	input.txt	output.txt
1	4 4	
2	1 2	
3	4 1	
4	2 3	
5	3 1	
6	2 4	

task6.py	input.txt	output.txt
1	2	

task6.py	input.txt	output.txt
1	5 4	
2	5 2	
3	1 3	
4	3 4	
5	1 4	
6	3 5	

```
task6.py  input.txt  output.txt x
1      -1
```

Результат работы кода на максимальных и минимальных значениях:

```
task6.py  input.txt x  output.txt
1      100000 100000
2      1 2
3      2 3
4      3 4
5      4 5
```

```
task6.py  input.txt  output.txt x
1      1
```

```
task6.py  input.txt x  output.txt
1      2 0
2      1 2
```

```
task6.py  input.txt  output.txt x
1      1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000009 сек	0.037641 МВ
Пример из задачи	0.000017 сек	0.037821 МВ
Пример из задачи	0.000017 сек	0.037821 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.000117 сек	29.726675 МВ

Вывод по задаче:

Я реализовала алгоритм для выведения минимального количества ребер в пути между двумя вершинами, используя поиск в ширину.

Задача №8. Стоимость полета [10 s, 512 Мб, 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.** $1 \leq n \leq 10^4$, $0 \leq m \leq 10^5$, $1 \leq u, v \leq n$, $u \neq v$, вес каждого ребра – неотрицательное целое число, не превосходящее 10^8 .
- **Формат вывода / выходного файла (output.txt).** Выведите минимальный вес пути из u в v . Введите -1, если пути нет.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.

Листинг кода:

```
import tracemalloc
import time
import heapq

tracemalloc.start()

def dijkstra(n, graph, 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_vertex == end:
            return current_distance

        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 -1

with open('input.txt', 'r') as file:
```

```

lines = file.readlines()
n, m = map(int, lines[0].split())
graph = {i: [] for i in range(1, n + 1)}
for line in lines[1:m + 1]:
    u, v, w = map(int, line.split())
    graph[u].append((v, w))
u, v = map(int, lines[m + 1].split())

start_time = time.perf_counter()

result = dijkstra(n, graph, u, v)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Для решения я использовала алгоритм Дейкстры. Функция Dijkstra принимает количество вершин в графе, начальную и конечную вершины, а также словарь, где ключи – это вершины, а значения – списки кортежей (сосед, вес). Далее я создала список для хранения минимального расстояния от начальной вершины до текущей, изначально все расстояния равны бесконечности. Расстояние вершины A до самой себя равно 0. Далее я инициализирую очередь с приоритетом (кучу) в виде кортежа (0, start), где 0 – текущее расстояние до вершины start.

Пока в очереди есть элементы, из кучи извлекается вершина с минимальным текущим расстоянием. Если текущая вершина – конечная, возвращаем текущее расстояние до нее. Если текущее расстояние до вершины больше, чем уже известное, эта вершина пропускается.

Для каждого соседа текущей вершины вычисляется новое расстояние до соседа. Если новое расстояние меньше известного, оно обновляется, и сосед добавляется в кучу.

Результат работы кода на примерах из текста задачи:

```
task8.py  input.txt  output.txt
1      4 4
2      1 2 1
3      4 1 2
4      2 3 2
5      1 3 5
6      1 3
```

```
task8.py  input.txt  output.txt
1      3
```

```
task8.py  input.txt  output.txt
1      5 9
2      1 2 4
3      1 3 2
4      2 3 2
5      3 2 1
6      2 4 2
7      3 5 4
8      5 4 1
9      2 5 3
10     3 4 4
11     1 5
```

```
task8.py  input.txt  output.txt
1      6
```

```
task8.py  input.txt  output.txt
1      3 3
2      1 2 7
3      1 3 5
4      2 3 2
5      3 2
```

```
task8.py  input.txt  output.txt x
1  |-1
```

Результат работы кода на максимальных и минимальных значениях:

```
task8.py  input.txt x  output.txt
1  10000 100000
2  2 3 1000000
3  3 4 1000000
4  4 5 1000000
```

```
task8.py  input.txt  output.txt x
1  9999000000
```

```
task8.py  input.txt x  output.txt
1  1 0
2  1 2
```

```
task8.py  input.txt  output.txt x
1  -1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000021 сек	0.037473 МВ
Пример из задачи	0.000024 сек	0.037661 МВ
Пример из задачи	0.000038 сек	0.037960 МВ
Пример из задачи	0.000012 сек	0.037614 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.056484 сек	21.081676 МВ

Вывод по задаче:

Я реализовала алгоритм Дейкстры для вычисления минимальной стоимости пути между двумя вершинами графа.

Задача №10. Оптимальный обмен валюты [10 s, 512 Mb, 2 балла]

Теперь вы хотите вычислить оптимальный способ обмена данной вам валюты c_i на *все* другие валюты. Для этого вы находите кратчайшие пути из вершины c_i во все остальные вершины.

Дан ориентированный граф с возможными отрицательными весами ребер, у которого n вершин и m ребер, а также задана одна его вершина s . Вычислите длину кратчайших путей из s во все остальные вершины графа.

- **Формат ввода / входного файла (input.txt).** Ориентированный взвешенный граф задан по формату 1.
- **Ограничения на входные данные.** $1 \leq n \leq 10^3$, $0 \leq m \leq 10^4$, $1 \leq s \leq n$, вес каждого ребра – целое число, не превосходящее *по модулю* 10^9 .
- **Формат вывода / выходного файла (output.txt).** Для каждой вершины i графа от 1 до n выведите в каждой отдельной строке следующее:
 - «*», если пути из s в i нет;
 - «-», если существует путь из s в i , но нет кратчайшего пути из s в i (то есть расстояние от s до i равно $-\infty$);
 - длину кратчайшего пути в остальных случаях.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.

Листинг кода:

```
import tracemalloc
import time

tracemalloc.start()

def bellman_ford(n, edges, start):
    distances = [float('inf')] * (n + 1)
    distances[start] = 0

    for _ in range(n - 1):
        for u, v, w in edges:
            if distances[u] != float('inf') and distances[u] + w < distances[v]:
                distances[v] = distances[u] + w

    negative_cycle = [False] * (n + 1)
    for _ in range(n - 1):
        for u, v, w in edges:
            if distances[u] != float('inf') and distances[u] + w < distances[v]:
                distances[v] = distances[u] + w
                negative_cycle[v] = True

    result = []
    for i in range(1, n + 1):
        if distances[i] == float('inf'):
            result.append("*")
        elif negative_cycle[i]:
            result.append("-")
        else:
            result.append(str(distances[i]))
```

```

    return result

with open('input.txt', 'r') as file:
    lines = file.readlines()
    n, m = map(int, lines[0].split())
    edges = []
    for line in lines[1:m + 1]:
        u, v, w = map(int, line.split())
        edges.append((u, v, w))
    s = int(lines[m + 1])

start_time = time.perf_counter()

result = bellman_ford(n, edges, s)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    for res in result:
        file.write(res + "\n")

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я использовала алгоритм Беллмана-Форда. Я создала список для хранения минимального расстояния от начальной вершины s до вершины i . Изначально все расстояния недостижимы, кроме расстояния до начальной вершины (0). Далее выполняется $n - 1$ итераций, на каждой итерации я прохожу по всем ребрам графа и обновляю расстояния до вершин, если найден более короткий путь. Если расстояние до вершины u не равно $+\infty$ и сумма расстояния до u и веса ребра w меньше текущего расстояния до v , обновляю расстояние до v .

Далее проверяю на отрицательные циклы: создаю список для хранения информации о том, достижима ли вершина через отрицательный цикл. Выполняю еще $n - 1$ итераций для проверки, можно ли еще уменьшить расстояния. Если да, то это значит, что в графе есть отрицательный цикл, и вершина v помечается как часть такого цикла.

Для формирования результата создаю список. Для каждой вершины проверяю, равно ли расстояние $+\infty$, если да, то пути нет. Если вершина достижима через отрицательный цикл, добавляю «-». В остальных случаях добавляю длину кратчайшего пути.

Результат работы кода на примерах из текста задачи:

```
task10.py  input.txt  output.txt
1      6 7
2      1 2 10
3      2 3 5
4      1 3 100
5      3 5 7
6      5 4 10
7      4 3 -18
8      6 1 -1
9      1
```

```
task10.py  input.txt  output.txt
1      0
2      10
3      -
4      -
5      -
6      *
```

```
task10.py  input.txt  output.txt
1      5 4
2      1 2 1
3      4 1 2
4      2 3 2
5      3 1 -5
6      4
```

```
task10.py  input.txt  output.txt
1      -
2      -
3      -
4      0
5      *
```

Результат работы кода на максимальных и минимальных значениях:

```
task10.py  input.txt  output.txt
1 1000 10000
2 2 3 10000000000
3 3 4 -10000000000
4 4 5 10000000000
5 5 6 -10000000000
6 6 7 10000000000
7 7 8 -10000000000
8 8 9 10000000000
```

```
task10.py  input.txt  output.txt
1 0
2 -10000000000
3 0
4 -10000000000
5 0
6 -10000000000
```

```
task10.py  input.txt  output.txt
1 1 0
2 1|
```

```
task10.py  input.txt  output.txt
1 0
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000024 сек	0.037597 МВ
Пример из задачи	0.000054 сек	0.037965 МВ
Пример из задачи	0.000051 сек	0.037786 МВ

Верхняя граница диапазона значений входных данных из текста задачи	8.516060 сек	2.259214 МВ
--	--------------	-------------

Вывод по задаче:

Задача №11. Алхимия [1 s, 16 Mb, 3 балла]

Алхимики средневековья владели знаниями о превращении различных химических веществ друг в друга. Это подтверждают и недавние исследования археологов.

В ходе археологических раскопок было обнаружено m глиняных табличек, каждая из которых была покрыта непонятными на первый взгляд символами. В результате расшифровки выяснилось, что каждая из табличек описывает одну алхимическую реакцию, которую умели проводить алхимики.

Результатом алхимической реакции является превращение одного вещества в другое. Задан набор алхимических реакций, описанных на найденных глиняных табличках, исходное вещество и требуемое вещество. Необходимо выяснить: возможно ли преобразовать исходное вещество в требуемое с помощью этого набора реакций, а в случае положительного ответа на этот вопрос – найти минимальное количество реакций, необходимое для осуществления такого преобразования.

- **Формат входных данных (input.txt) и ограничения.** Первая строка входного файла INPUT.TXT содержит целое число m ($0 \leq m \leq 1000$) – количество записей в книге. Каждая из последующих m строк описывает одну алхимическую реакцию и имеет формат «вещество1 -> вещество2», где «вещество1» – название исходного вещества, «вещество2» – название продукта алхимической реакции. $m + 2$ -ая строка входного файла содержит название вещества, которое имеется изначально, $m + 3$ -ая – название вещества, которое требуется получить.

Во входном файле упоминается не более 100 различных веществ. Название каждого из веществ состоит из строчных и заглавных английских букв и имеет длину не более 20 символов. Строчные и заглавные буквы различаются.

- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT выведите минимальное количество алхимических реакций, которое требуется для получения требуемого вещества из исходного, или -1, если требуемое вещество невозможно получить.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.

Листинг кода:

```
import tracemalloc
import time
from collections import deque

tracemalloc.start()

def bfs(start, end, graph):
    queue = deque([(start, 0)])
    visited = set()

    while queue:
        current, steps = queue.popleft()
        if current == end:
            return steps
        if current in visited:
            continue
```

```

        visited.add(current)

        if current in graph:
            for neighbor in graph[current]:
                if neighbor not in visited:
                    queue.append((neighbor, steps + 1))

    return -1

with open('input.txt', 'r') as file:
    lines = file.readlines()
    m = int(lines[0])
    graph = {}
    for i in range(1, m + 1):
        reaction = lines[i].strip().split(' -> ')
        if len(reaction) == 2:
            src, dest = reaction
            if src not in graph:
                graph[src] = []
            graph[src].append(dest)
    start = lines[m + 1].strip()
    end = lines[m + 2].strip()

start_time = time.perf_counter()

result = bfs(start, end, graph)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я реализовала очередь в алгоритме BFS с помощью deque. Функция bfs принимает начальное и требуемое вещества, а также граф реакций в виде словаря. Я создаю очередь, содержащую пары – вещество, количество реакций (изначально (start, 0)), а также множество для хранения уже посещенных веществ. Пока очередь не пуста, извлекаем из очереди текущее вещество и количество реакций, необходимых для его получения. Если текущее вещество – требуемое, возвращаем количество реакций. Если текущее вещество уже посещено – пропускаем его. Добавляем текущее вещество в множество посещенных и проверяем, есть ли оно в графе: если да, проходим по соседям, и если сосед еще не был посещен, добавляем его в очередь с увеличенным на 1 количеством реакций.

Результат работы кода на примерах из текста задачи:

```
task11.py  input.txt  output.txt
1  5
2  Aqua -> AquaVita
3  AquaVita -> PhilosopherStone
4  AquaVita -> Argentum
5  Argentum -> Aurum
6  AquaVita -> Aurum
7  Aqua
8  Aurum
```

```
task11.py  input.txt  output.txt  x
1  k
```

```
task11.py  input.txt  output.txt
1  5
2  Aqua -> AquaVita
3  AquaVita -> PhilosopherStone
4  AquaVita -> Argentum
5  Argentum -> Aurum
6  AquaVita -> Aurum
7  Aqua
8  Osmium
```

```
task11.py  input.txt  output.txt  x
1  -1
```

Результат работы кода на максимальных и минимальных значениях:

```
task11.py  input.txt  output.txt
1 1000
2 Substance1 -> Substance2
3 Substance2 -> Substance3
4 Substance3 -> Substance4
5 Substance4 -> Substance5
6 Substance5 -> Substance6
7 Substance6 -> Substance7
```

```
task11.py  input.txt  output.txt
1 0
```

```
task11.py  input.txt  output.txt
1 0
2 H2
3 Zn
```

```
task11.py  input.txt  output.txt
1 1
```

Проверка задачи на астр:

ID	Дата	Автор	Задача	Язык	Результат	Тест	Время	Память
22741752	14.01.2025 2:52:11	Гайдух Аллина Сергеевна	0743	PyPy	Accepted		0,265	4960 Кб

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000011 сек	0.037682 МВ
Пример из задачи	0.000016 сек	0.037995 МВ
Пример из задачи	0.000015 сек	0.037996 МВ
Верхняя граница	0.000288 сек	0.188708 МВ

диапазона значений ВХОДНЫХ ДАННЫХ ИЗ текста задачи		
--	--	--

Вывод по задаче:

Я реализовала алгоритм для подсчета минимального количества реакций для получения требуемого вещества из исходного, используя BFS через двухстороннюю очередь.

Задача №14. Автобусы [1 s, 16 Mb, 3 балла]

Между некоторыми деревнями края Власюки ходят автобусы. Поскольку пассажиропотоки здесь не очень большие, то автобусы ходят всего несколько раз в день.

Марии Ивановне требуется добраться из деревни d в деревню v как можно быстрее (считается, что в момент времени 0 она находится в деревне d).

- **Формат входных данных (input.txt) и ограничения.** Во входном файле INPUT.TXT записано число N - общее число деревень ($1 \leq N \leq 100$), номера деревень d и v , затем количество автобусных рейсов R ($0 \leq R \leq 10000$). Затем идут описания автобусных рейсов. Каждый рейс задается номером деревни отправления, временем отправления, деревней назначения и временем прибытия (все времена - целые от 0 до 10000). Если в момент t пассажир приезжает в деревню, то уехать из нее он может в любой момент времени, начиная с t .
- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT вывести минимальное время, когда Мария Ивановна может оказаться в деревне v . Если она не сможет с помощью указанных автобусных рейсов добраться из d в v , вывести -1.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.

Листинг кода:

```
import heapq
import tracemalloc
import time

tracemalloc.start()

def dijkstra(n, d, v, buses):
    arrival_time = [float('inf')] * (n + 1)
    arrival_time[d] = 0
    queue = [(0, d)]

    while queue:
        current_time, current_village = heapq.heappop(queue)

        if current_village == v:
            return current_time

        if current_time > arrival_time[current_village]:
            continue

        for bus in buses[current_village]:
            departure, destination, arrival = bus
```

```

        if current_time <= departure and arrival <
arrival_time[destination]:
            arrival_time[destination] = arrival
            heapq.heappush(queue, (arrival, destination))

    return -1

with open('input.txt', 'r') as file:
    lines = file.readlines()
    n = int(lines[0].strip())
    d, v = map(int, lines[1].strip().split())
    r = int(lines[2].strip())
    buses = [[] for _ in range(n + 1)]
    for i in range(3, 3 + r):
        departure_village, departure_time, destination_village,
arrival_time = map(int, lines[i].strip().split())
        buses[departure_village].append((departure_time,
destination_village, arrival_time))

start_time = time.perf_counter()

result = dijkstra(n, d, v, buses)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    file.write(str(result))

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я реализовала алгоритм Дейкстры, используя очередь с приоритетом (т. е. кучу). Основная функция Dijkstra принимает на вход количество деревень, начальную и конечную деревню, а также список автобусных рейсов для каждой деревни. Далее я создала список для хранения минимального времени прибытия в деревню, изначально установив все времена в $+\infty$, кроме времени для начальной деревни (0). Далее я инициализировала кучу парой (0, d), где 0 – время прибытия в начальную деревню, а d – сама деревня. Пока очередь не опустеет, извлекаю пару (текущее время, текущая деревня) из очереди. Если текущая деревня – конечная, возвращаю текущее время прибытия. Если текущее время больше известного времени прибытия в текущую деревню, пропускаю.

Далее для каждого автобусного рейса из текущей деревни проверяю, что время прибытия в текущую деревню меньше или равно времени отправления автобуса, и, что время прибытия в деревню назначения меньше известного. Если да, добавляю деревню в очередь.

Результат работы кода на примерах из текста задачи:

```
task14.py  input.txt  output.txt
1      3
2      1 3
3      4
4      1 0 2 5
5      1 1 2 3
6      2 3 3 5
7      1 1 3 10
```

```
task14.py  input.txt  output.txt
1      5
```

Результат работы кода на максимальных и минимальных значениях:

```
task14.py  input.txt  output.txt
1      100
2      1 100
3      10000
4      4 8444 72 9898
5      84 3654 62 9849
6      89 7049 99 9911
7      22 6247 2 9026
8      87 2702 57 8742
```

```
task14.py  input.txt  output.txt
1      4944
```

```
task14.py  input.txt  output.txt
1      1
2      1 2
3      0
```

```
task14.py  input.txt  output.txt
1      -1
```

Проверка задачи на астр:

22741766	14.01.2025 3:23:20	Гайдук Аллина Сергеевна	0134	PyPy	Accepted	0.234	3248 Кб
----------	--------------------	-------------------------	------	------	----------	-------	---------

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи		
Пример из задачи	0.000020 сек	0.037879 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.000727 сек	1.975964 МВ

Вывод по задаче:

Я реализовала алгоритм Дейкстры, используя кучу, для нахождения минимального времени пути из одной деревни в другую.

Вывод

В ходе данной выполнения лабораторной работы я изучила графы, поиск в глубину, поиск в ширину, алгоритм Дейкстры и алгоритм Беллмана-Форда, а также смогла применить свои знания на практике.