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

Отчет по лабораторной работе №3

по курсу «Алгоритмы и структуры данных»

Тема: Графы

Вариант 4

Выполнила:

Буй Тхук Хуен - К3139

Проверила:

Петросян А.М.

Санкт-Петербург

2025 г.

Содержание отчета

I. Задачи по варианту

Задача №4. Порядок курсов

Теперь, когда вы уверены, что в данном учебном плане нет циклических зависимостей, вам нужно найти порядок всех курсов, соответствующий всем зависимостям. Для этого нужно сделать топологическую сортировку соответствующего ориентированного графа.

Дан ориентированный ациклический граф (DAG) с n вершинами и m ребрами. Выполните топологическую сортировку.

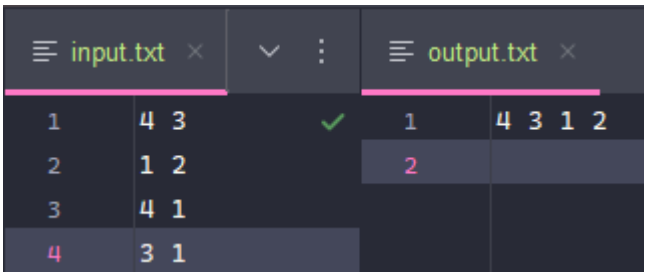
- **Формат ввода / входного файла (input.txt).** Ориентированный ациклический граф с n вершинами и m ребрами по формату 1.
- **Ограничения на входные данные.** $1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$. Графы во входных файлах гарантированно ациклические.
- **Формат вывода / выходного файла (output.txt).** Выведите ***любое*** линейное упорядочение данного графа (Многие ациклические графы имеют более одного варианта упорядочения, вы можете вывести любой из них).
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.

- Листинг кода

```python
import sys
from Lab3.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

sys.setrecursionlimit(200000)# Увеличиваем лимит рекурсии для больших графов

def topological_sort(n, edges):
    from collections import defaultdict

    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)

    visited = [0] * (n + 1)
    order = []

    def dfs(node):
        visited[node] = 1
        for neighbor in graph[node]:
            if not visited[neighbor]:
                dfs(neighbor)
        order.append(node)

    for node in range(1, n + 1):
        if not visited[node]:
            dfs(node)

    return order[::-1]

def main():
    n, edges = read(PATH_INPUT, 4)
```

```
    results = topological_sort(n, edges)
    write(PATH_OUTPUT, results, 4)

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

- Текстовое объяснение решения
  + Мы увеличиваем ограничение глубины рекурсии, так как для больших графов (до 100 000 вершин) рекурсивный DFS может уйти глубоко.
  + Функция topological_sort принимает: n — количество вершин,edges — список рёбер.
  + Используем defaultdict(list), чтобы представлять граф в виде списка смежности: Здесь u -> v означает, что вершина u указывает на вершину v.
  + visited отслеживает, какие вершины уже были посещены.
  + order хранит порядок вершин после обхода.
  + Функция dfs(node): Помечаем вершину как посещённую. Обходим всех её соседей, которые ещё не посещены. После завершения рекурсии добавляем вершину в order.
  + Обходим все вершины и запускаем DFS, если они ещё не посещены. Порядок вершин получается обратным из-за специфики DFS, поэтому переворачиваем список.
- Результат работы код на примерах из текста задачи:



  - Test:

```
Test Case: test1
Execution Time = 0.00002670 s, Memory Usage = 1.1064 KB


Ran 3 tests in 0.003s

OK

Test Case: test2
Execution Time = 0.00002230 s, Memory Usage = 1.3018 KB

Test Case: test3
Execution Time = 0.00002560 s, Memory Usage = 1.5547 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
4 1 3 1	0.00002670	1.1064
5 7 2 1 3 2 3 1 4 3 4 1 5 2 5 3	0.00002230	1.3018
6 6 6 3 6 1 5 1 5 2 3 4 4 2	0.00002560	1.5547

- Вывод

  + Алгоритмы могут применяться во многих областях, таких как планирование курсов, составление графиков работ с учетом зависимостей или обработка цепочек задач.

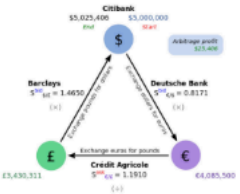
+ Сложность: O(n+m), где n — количество вершин, а m — количество ребер, что обеспечивает хорошую производительность в рамках задачи.
+ Использование списков смежности экономит память
+ DFS использует рекурсию, но sys.setrecursionlimit(200000), чтобы избежать переполнения стека.

Задача №9. Аномалии курсов валют

Вам дан список валют $c_1, c_2, \ldots, c_n$ вместе со списком обменных курсов: r_{ij} – количество единиц валюты c_j, которое можно получить за одну единицу c_i. Вы хотите проверить, можно ли начать делать обмен с одной единицы какой-либо валюты, выполнить последовательность обменов и получить более одной единицы той же валюты, с которой вы начали обмен. Другими словами, вы хотите найти валюты $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ такие, что $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$.

Для этого построим следующий граф: вершинами являются валюты $c_1, c_2, \ldots, c_n$, вес ребра из c_i в c_j равен $-\log r_{ij}$. Тогда достаточно проверить, есть ли в этом графе отрицательный цикл. Пусть цикл $c_i \to c_j \to c_k \to c_i$ имеет отрицательный вес. Это означает, что $-(\log c_{ij} + \log c_{jk} + \log c_{ki}) < 0$ и, следовательно, $\log c_{ij} + \log c_{jk} + \log c_{ki} > 0$. Это, в свою очередь, означает, что

$$r_{ij} r_{jk} r_{ki} = 2^{\log c_{ij}} 2^{\log c_{jk}} 2^{\log c_{ki}} = 2^{\log c_{ij} + \log c_{jk} + \log c_{ki}} > 1.$$



Для заданного ориентированного графа с возможными отрицательными весами ребер, у которого n вершин и m ребер, проверьте, содержит ли он цикл с отрицательным суммарным весом.

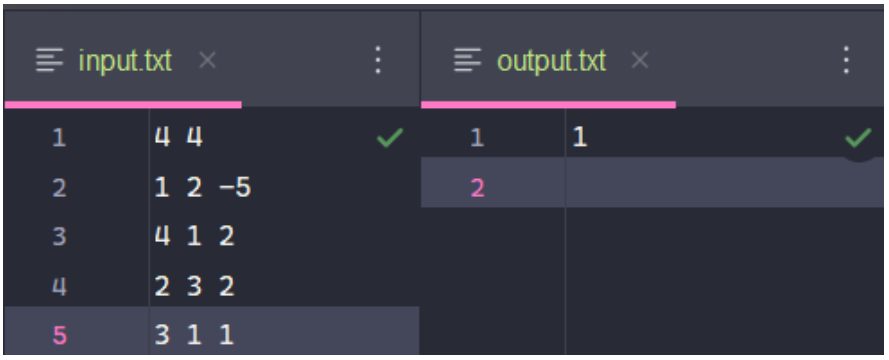
- **Формат ввода / входного файла (input.txt).** Ориентированный взвешенный граф задан по формату 1.
- **Ограничения на входные данные.** $1 \le n \le 10^3$, $0 \le m \le 10^4$, вес каждого ребра – целое число, не превосходящее ***по модулю*** 10^4.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если граф содержит цикл с отрицательным суммарным весом. Выведите 0 в противном случае.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.

- Листинг кода

```
from Lab3.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

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

    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] != INF and dist[u] + w < dist[v]:
```

```
                dist[v] = dist[u] + w

    for u, v, w in edges:
        if dist[u] != INF and dist[u] + w < dist[v]:
            return 1

    return 0

def main():
    n, edges = read(PATH_INPUT, 9)
    result = bellman_ford(n, edges)
    write(PATH_OUTPUT, result, 9)

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

- Текстовое объяснение решения:
  + INF = float('inf') — устанавливаем бесконечность для всех расстояний.
  + dist = [INF] * (n + 1) — массив расстояний, где dist[i] хранит минимальное расстояние до вершины i.
  + dist[1] = 0 — начинаем с произвольной вершины (в данном случае, 1).
  + Проходим (n - 1) раз, так как в худшем случае кратчайшие пути обновляются $n-1$n - 1$n-1$ раз.
  + Для каждого ребра (u, v, w):
    - Если dist[u] + w < dist[v], обновляем dist[v].
    - Если после $n-1$ итераций можно ещё раз уменьшить расстояние, то есть отрицательный цикл.
- Результат работы код на примерах из текста задачи:



- Test

```
Test Case: test_case_1
Execution Time = 0.00001680 s, Memory Usage = 0.2656 KB

Test Case: test_case_2
Execution Time = 0.00001400 s, Memory Usage = 0.2422 KB

Test Case: test_case_3
Execution Time = 0.00000710 s, Memory Usage = 0.1250 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
3 3 1 2 -2 2 3 -2 3 1 -2	0.00001680	0.2656
4 4 1 2 3 2 3 4 3 4 -8 4 1 -2	0.00001400	0.2422
5 0	0.00000710	0.1250

- Вывод
  - + Этот код эффективно обнаруживает наличие отрицательного цикла в графе, что важно для анализа валютного арбитража.
  - + Временная сложность: O(nm), так как у нас n-1 итераций по m рёбрам.
  - + Пространственная сложность: O(n+m) (храним массив расстояний и список рёбер).

Задача №15. Герои

Коварный кардинал Ришелье вновь организовал похищение подвесок королевы Анны; вновь спасать королеву приходится героическим мушкетерам. Атос, Портос, Арамис и д'Артаньян уже перехватили агентов кардинала и вернули украденное; осталось лишь передать подвески королеве Анне. Королева ждет мушкетеров в дворцовом саду. Дворцовый сад имеет форму прямоугольника и разбит на участки, представляющие собой небольшие садики, содержащие коллекции растений из разных климатических зон. К сожалению, на некоторых участках, в том числе на всех участках, расположенных на границах сада, уже притаились в засаде гвардейцы кардинала; на бой с ними времени у мушкетеров нет. Мушкетерам удалось добыть карту сада с отмеченными местами засад; теперь им предстоит выбрать наиболее оптимальные пути к королеве. Для надежности друзья разделили между собой спасенные подвески и проникли в сад поодиночке, поэтому начинают свой путь к королеве с разных участков сада. Двигаются герои по максимально короткой возможной траектории.

Марлезонский балет вот-вот начнется; королева не в состоянии ждать героев больше L минут; ровно в начале $L+1$-ой минуты королева покинет парк, и те мушкетеры, что не успеют к этому времени до нее добраться, не смогут передать ей подвески. На преодоление одного участка у мушкетеров уйдет ровно по минуте. С каждого участка мушкетеры могут перейти на 4 соседние. Требуется выяснить, сколько подвесок будет красоваться на платье королевы, когда она придет на бал.

- **Формат входных данных (input.txt) и ограничения.** Первая строка входного файла INPUT.TXT содержит целые числа N и M ($1 \le N, M \le 20$) – размеры сада. Далее идут N строк по M символов в каждом; символы '0' соответствуют участкам, на которых нет засады, символы '1' – участкам, на которых разместились гвардейцы. В $N+2$-ой строке теста записано три целых числа: координаты участка, на котором королева будет ждать мушкетёров (Q_x, Q_y) ($1 < Q_x < N$, $1 < Q_y < M$) и время в минутах до начала балета ($1 \le L \le 1000$). В $N+3$-ей строке записаны через пробел целые числа координаты участка, с которого стартует Атос (A_x, A_y) ($1 < A_x < N$, $1 < A_y < M$) и количество подвесок, хранящихся у него ($1 \le P_a \le 1000$). В $N+4$, $N+5$ и $N+6$-ой строках аналогично записаны стартовые координаты и количество подвесок у Портоса, Арамиса и д'Артаньяна.
- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT выведите количество подвесок, которое королева успеет получить у мушкетеров до начала балета.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.

- Листинг кода

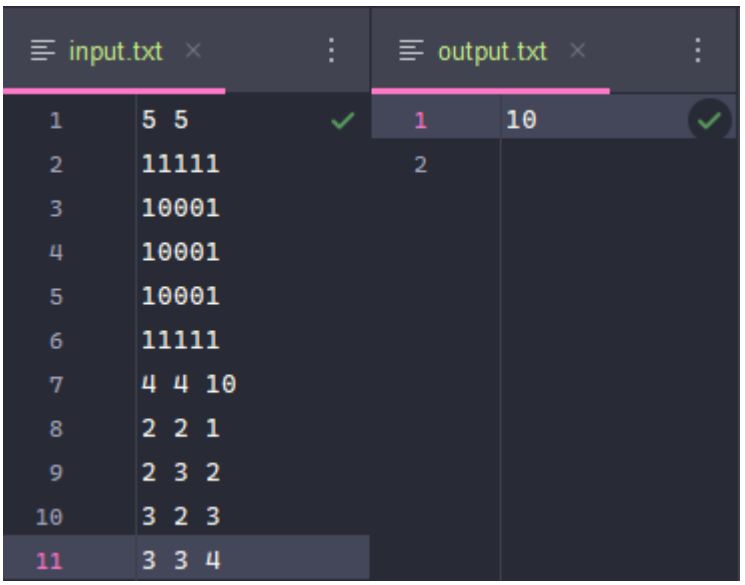
```
from Lab3.utils import read, write
from collections import deque

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def bfs(start_x, start_y, qx, qy, garden, n, m):
    queue = deque([(start_x, start_y, 0)])
    visited = set([(start_x, start_y)])

    while queue:
        x, y, dist = queue.popleft()
        if (x, y) == (qx, qy):
            return dist
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and (nx, ny) not in visited and
garden[nx][ny] == '0':
                visited.add((nx, ny))
            queue.append((nx, ny, dist + 1))

    return float('inf')

def main():
```

```
    n, m, garden, (qx, qy, L), musketeers = read(PATH_INPUT, 15)

    # Переключиться на индекс, начинающийся с 0
    qx, qy = qx - 1, qy - 1
    musketeers = [(x - 1, y - 1, p) for x, y, p in musketeers]

    # Считаем подвески, которые успеют к королеве
    total_pendants = 0
    for Ax, Ay, Pa in musketeers:
        min_time = bfs(Ax, Ay, qx, qy, garden, n, m)  # Приводим к 0-индексации
        if min_time <= L:
            total_pendants += Pa

    # Запись результата
    write(PATH_OUTPUT, total_pendants, 15)

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

- Текстовое объяснение решения
  + Эта функция находит кратчайшее расстояние от точки (start_x, start_y) до королевы (qx, qy).
  + Начиная с позиции рейнджера, сохраним в очереди на расстоянии 0.
  + Пройдем по каждой ячейке, взяв первый элемент из очереди (popleft()).
  + Если королева найдена, вернет минимальное количество минут на путешествие.
  + Двигается в 4 направлениях, если:
    - Не выходит в сад.
    - Не входит в ограждение («1»).
    - Никогда там раньше не был. Добавить в очередь, отметить как пропущенное.
+ n, m: Размер сада (N x M).
+ garden: Карта сада (0: тропа, 1: охрана).
+ (qx, qy, L): Положение ферзя и время ожидания.
+ musketeers: Список мушкетеров и количество ожерелий, которые они носят.
+ Поскольку Python использует индексацию, начинающуюся с 0, нам необходимо уменьшить все координаты на 1.
+ Просмотрим каждого мушкетера:

- Найти кратчайший путь к королеве с помощью bfs().
- Если успеют сделать это до ухода королевы, добавит количество ожерелий к total_pendants.

- Результат работы код на примерах из текста задачи



- Test

```
Test Case: test_case_1
Execution Time = 0.00007980 s, Memory Usage = 1.6250 KB

Test Case: test_case_2
Execution Time = 0.00005470 s, Memory Usage = 1.6250 KB

Test Case: test_case_3
Execution Time = 0.00006760 s, Memory Usage = 2.0938 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
5 5 11111 10001 10001 10001 11111 4 4 10 2 2 1 2 3 2 3 2 3	0.00007980	1.6250

3 3 4		
5 5 11111 10001 10111 10101 11111 4 4 10 2 2 1 2 2 2 2 2 3 2 2 4	0.00005470	1.6250
5 5 11111 10001 10101 10001 11111 4 4 3 2 2 1 2 3 2 3 2 3 3 3 4	0.00006760	2.0938

- Вывод
  + Алгоритм BFS: оптимальное решение задачи поиска кратчайшего пути в невзвешенном графе, помогает быстро находить результаты. Гарантированно находит кратчайший путь в саду со сложностью O(N * M), где N — количество строк, а M — количество столбцов сада, поскольку каждая ячейка в саду посещается только один раз.

II. Дополнительные задачи

Задача №2. Компоненты

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.

- **Формат ввода / входного файла (input.txt).** Неориентированный граф с n вершинами и m ребрами по формату 1.
- **Ограничения на входные данные.** $1 \le n \le 10^3, 0 \le m \le 10^3$.
- **Формат вывода / выходного файла (output.txt).** Выведите количество компонент связности.
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.

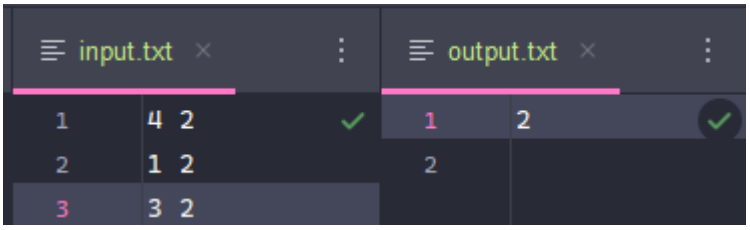
- Листинг кода

```python
from Lab3.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'


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

def find_connected_components(n, m, edges):
    # Создаём список смежности для графа
    graph = [[] for _ in range(n)]

    for edge in edges:
        u, v = edge
        graph[u - 1].append(v - 1)  # Индексы вершин начинаются с 0
        graph[v - 1].append(u - 1)

    visited = [False] * n
    component_count = 0

    for i in range(n):
        if not visited[i]:
            dfs(i, graph, visited)
            component_count += 1

    return component_count

def main():
    n, m, edges = read(PATH_INPUT, 2)
    results = find_connected_components(n, m, edges)
    write(PATH_OUTPUT, results, 2)
```

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

- Текстовое объяснение решения:
  + Сначала создаём список смежности для графа, где для каждой вершины будет храниться список её соседей.
  + Далее используем DFS для поиска компонент связности. Когда мы встречаем непосещённую вершину, это значит, что мы нашли новую компоненту, и запускаем DFS, чтобы отметить все вершины этой компоненты.
  + Количество запусков DFS будет равно количеству компонент связности в графе.

- Результат работы кода:



- Test

```
Test Case: test1
Execution Time = 0.00001390 s, Memory Usage = 0.1797 KB

Test Case: test2
Execution Time = 0.00001590 s, Memory Usage = 0.4688 KB

Test Case: test3
Execution Time = 0.00002090 s, Memory Usage = 0.5000 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
5 0	0.00001390	0.1797
6 3 1 2	0.00001590	0.4688

Тест	Время выполнения (s)	Затраты памяти (KB)
3 4 4 5		
6 4 1 2 2 3 4 5 5 6	0.00002090	0.5000

- Вывод
    + Проблему можно эффективно решить с помощью DFS, а конечным результатом будет подсчет количества инициализаций DFS, соответствующий количеству связанных компонентов в графе.
    + Временная сложность: O(n + m), где n - количество вершин, а m - количество ребер. Для каждой вершины мы выполняем один проход по списку смежности, и каждое ребро проходится только один раз.

III. Вывод

Сравнение алгоритмов BFS, DFS и Bellman-Ford

Критерии	BFS	DFS	Bellman-Ford
Метод	Обход вершин по уровням, от ближайшей к самой дальней (FIFO)	Обход в глубину, прохождение всех вершин в ветви и последующий возврат (LIFO)	Использует итерационный метод, обновит значение кратчайшего пути через каждое ребро.
Временная сложность	O(V + E), где V — количество вершин, а E — количество ребер	O(V + E) для неориентированных графов, $O(V^2)$ для ориентированных графов	O(V * E), где V — количество вершин, а E — количество ребер.

Пространственная сложность	O(V), где V — количество вершин в графе	O(V), хранит посещённые вершины	O(V) хранит расстояния от корневой вершины
Типы графов	Неориентированные и ориентированные графы.	Неориентированные и ориентированные графы.	Взвешенные графы могут иметь отрицательные веса.
Преимущества	Простота реализации, эффективность для поиска кратчайших путей в невзвешенных неориентированных графах	Подходит для задач, связанных с обходом графа в глубину.	Обнаружение отрицательного цикла, применимое к графикам с отрицательным весом.
Недостатки	Плохо работает с взвешенными или ориентированными графами.	Не всегда возможно найти кратчайший путь.	Медленнее алгоритма Дейкстры на графах без отрицательных весов.