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

Отчет по лабораторной работе №3 (семестр 2)
по курсу «Алгоритмы и структуры данных»
Тема: Графы

Выполнил:
Нгуен Хыу Жанг
Мобильные и сетевые технологии
К3140

Проверила:
Петросян Анна Мнацакановна

Санкт-Петербург
2025 г

Содержание

Вариант 2

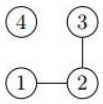
Задание 2 : Компоненты

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.
- Пример:

input	output
4 2 1 2 3 2	2



В этом графе есть два компонента связности: 1, 2, 3 и 4.

```
import os
from collections import defaultdict

def count_components(input_file='input.txt', output_file='output.txt', to_file=True):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as f:
        n, m = map(int, f.readline().strip().split())
        graph = defaultdict(list)

        for _ in range(m):
            u, v = map(int, f.readline().strip().split())
            graph[u].append(v)
            graph[v].append(u)

    def dfs(node, visited):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
```

```
                dfs(neighbor, visited)

    visited = set()
    component_count = 0

    for vertex in range(1, n + 1):
        if vertex not in visited:
            dfs(vertex, visited)
            component_count += 1

    if to_file:
        with open(output_path, 'w') as f:
            f.write(f"{component_count}\n")
    return component_count
```

input.txt:

```
4 2
1 2
3 2
```

output.txt:

```
2
```

Объяснение кода задания о компонентах связности:

- В данной задаче необходимо найти количество компонент связности в неориентированном графе, представленном в виде списка смежности. Для решения задачи используется обход в глубину (DFS).

Структура кода:

1. **Импорт библиотек**:
   - Импортируется модуль `os` для работы с файловой системой и `defaultdict` из модуля `collections` для создания графа в виде списка смежности.
2. **Функция `count_components`**:
   - Основная функция, которая считывает входные данные, строит граф и считает количество компонент связности.
3. **Чтение входных данных**:
   - Открывается файл `input.txt` и считывается количество вершин `n` и количество ребер `m`.
   - Создается граф в виде словаря, где ключом является вершина, а значениями — список соседних вершин.
4. **Построение графа**:
   - Для каждого ребра (u, v) добавляем v в список соседей для u и u в список соседей для v, тем самым создавая неориентированный граф.
5. **Функция `dfs(node, visited)`**:
   - Рекурсивная функция, реализующая обход в глубину.
   - Добавляет текущую вершину `node` в множество `visited` и рекурсивно обходит всех соседей, которые еще не были посещены.
6. **Счетчик компонентов**:
   - Создается множество `visited` для отслеживания посещенных вершин и переменная `component_count` для подсчета количества компонент связности.
   - Для каждой вершины от 1 до n проверяется, была ли она посещена. Если нет, запускается DFS из этой вершины, и увеличивается счетчик компонент.
7. **Запись результата в файл**:
   - После подсчета компонент связности результат записывается в файл `output.txt`.

Пример работы:

- Для входных данных:

```
4 2
1 2
3 2
```

- **Граф**:
  - Вершины: 4
  - Ребра: 2 (1 соединяется с 2, 2 соединяется с 3).
- **Компоненты связности**:
  - Первая компонента: 1, 2, 3 (все связаны между собой).
  - Вторая компонента: 4 (отдельная вершина).

- Таким образом, количество компонент связности в графе равно 2, что и будет выведено в `output.txt`.

Заключение:

- Код эффективно находит количество компонент связности в неориентированном графе с помощью метода обхода в глубину, обеспечивая выполнение в рамках заданных ограничений по времени и памяти.

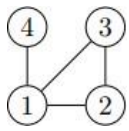
Задание 7 : Двудольный граф

Неориентированный граф называется **двудольным**, если его вершины можно разбить на две части так, что каждое ребро графа соединяет вершины из разных частей, то есть не существует рёбер между вершинами одной и той же части графа. Двудольные графы естественным образом возникают в задачах, где граф используется для моделирования связей между объектами двух разных типов (например, мальчиками и девочками, или студентами и общежитиями). Альтернативное определение таково: граф двудольный, если его вершины можно раскрасить двумя цветами (например, черным и белым) так, что концы каждого ребра окрашены в разные цвета.

Дан неориентированный граф с n вершинами и m ребрами, проверьте, является ли он двудольным.

- **Формат ввода / входного файла (input.txt).** Неориентированный граф задан по формату 1.
- **Ограничения на входные данные.** $1 \le n \le 10^5$, $0 \le m \le 10^5$.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если граф двудольный; и 0 в противном случае.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Пример 1:

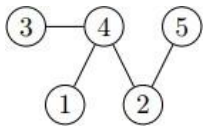
input	output
4 4 1 2 4 1 2 3 3 1	0



Этот граф не является двудольным. Чтобы убедиться в этом, предположим, что вершина 1 окрашена в белый цвет. Тогда вершины 2 и 3 нужно покрасить в черный цвет, так как граф содержит ребра 1, 2 и 1, 3. Но тогда ребро 2, 3 имеет оба конца одного цвета.

- Пример 2:

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



Этот граф двудольный: вершины 4 и 5 покрасим в белый цвет, все остальные вершины – в черный цвет.

- Что делать? Адаптируйте поиск в ширину (BFS), чтобы решить эту проблему.

```
import os
from collections import defaultdict, deque

def is_bipartite(n, edges):
    graph = defaultdict(list)

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

    color = {}

    def bfs(start):
        queue = deque([start])
        color[start] = 0

        while queue:
            node = queue.popleft()
            for neighbor in graph[node]:
                if neighbor not in color:
                    color[neighbor] = 1 - color[node]
                    queue.append(neighbor)
                elif color[neighbor] == color[node]:
                    return False
        return True

    for vertex in range(1, n + 1):
        if vertex not in color:
            if not bfs(vertex):
                return 0

    return 1

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as f:
        n, m = map(int, f.readline().strip().split())
        edges = [tuple(map(int, f.readline().strip().split())) for _ in range(m)]

    result = is_bipartite(n, edges)

    with open(output_path, 'w') as f:
```

```
        f.write(f"{result}\n")

    return str(result)

if __name__ == "__main__":
    task_scheduler(input_file='input.txt', output_file='output.txt')
```

input.txt:

```
4 4
1 2
4 1
2 3
3 1
```

output.txt:

```
0
```

Объяснение кода задания о двудольном графе:
- В данной задаче необходимо проверить, является ли неориентированный граф двудольным. Это можно сделать с помощью алгоритма обхода в ширину (BFS), раскрашивая вершины графа в два цвета и проверяя, что соседние вершины имеют разные цвета.

Структура кода:

1. **Импорт библиотек**:
   - Импортируем `os` для работы с файловой системой и `defaultdict`, `deque` из модуля `collections`, чтобы создать граф и реализовать очередь для BFS.
2. **Функция `is_bipartite(n, edges)`**:
   - Основная функция, которая принимает количество вершин `n` и список рёбер `edges`, чтобы проверить, является ли граф двудольным.
   - **Создание графа**:
     - Создается словарь `graph`, где ключом является вершина, а значениями — список соседних вершин на основе введенных рёбер.
3. **Распределение цветов**:
   - Создается словарь `color`, который будет хранить цвет каждой вершины (0 или 1).
4. **Функция `bfs(start)`**:
   - Реализует обход в ширину начиная с вершины `start`.
   - Вершина `start` получает цвет 0, и добавляется в очередь `queue`.
   - Пока очередь не пуста, происходит следующий процесс:
     - Извлекается вершина `node` из очереди.
     - Для каждого соседа `neighbor` проверяется:
       - Если сосед еще не окрашен, он получает цвет, противоположный цвету `node`.
       - Если сосед уже окрашен и имеет тот же цвет, что и `node`, это означает, что граф не является двудольным, и функция возвращает `False`.
   - Если все соседи окрасились правильно, возвращается `True`.
5. **Главный цикл проверки**:
   - Для каждой вершины от 1 до n проверяется, была ли она уже окрашена. Если нет, запускается BFS из этой вершины.

- Если в какой-то момент BFS возвращает `False`, то граф не двудольный, и функция `is_bipartite` возвращает `0`.

6. **Запись результата в файл**:
   - Результат (1, если граф двудольный, и 0 в противном случае) записывается в файл `output.txt`.

Примеры работы:

1. **Пример 1**:
   - Граф: 1 соединяется с 2 и 3.
   - Невозможно раскрасить 2 и 3 разными цветами, если 1 окрашена в один из них. Значит, граф не является двудольным.
   - Результат: `0`.
2. **Пример 2**:
   - Например, 4 и 5 окрашены в белый, остальные в черный, и все ребра соединяют вершины разных цветов.
   - Результат: `1`.

Заключение:

- Код реализует эффективный способ проверки свойств двудольности графа с использованием алгоритма BFS и цвета вершин. Это решение работает в пределах заданных ограничений по времени и памяти, что делает его подходящим для больших графов.

Задание 13 : Грядки

Прямоугольный садовый участок шириной N и длиной M метров разбит на квадраты со стороной 1 метр. На этом участке вскопаны грядки. Грядкой называется совокупность квадратов, удовлетворяющая таким условиям:

- из любого квадрата этой грядки можно попасть в любой другой квадрат этой же грядки, последовательно переходя по грядке из квадрата в квадрат через их общую сторону;
- никакие две грядки не пересекаются и не касаются друг друга ни по вертикальной, ни по горизонтальной сторонам квадратов (касание грядок углами квадратов допускается).

Подсчитайте количество грядок на садовом участке.

- **Формат входных данных (input.txt) и ограничения.** В первой строке входного файла INPUT.TXT находятся числа N и M через пробел, далее идут N строк по M символов. Символ # обозначает территорию грядки, точка соответствует незанятой территории. Других символов в исходном файле нет ($1 \le N, M \le 200$).
- **Формат выходных данных (output.txt).** В выходной файл OUTPUT.TXT выведите количество грядок на садовом участке.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.
- Примеры:

input.txt	output.txt	input.txt	output.txt
5 10 ##.....#. .#..#...#. .###....#. ..##....#. ....#.	3	5 10 ##..#####. .#.#.#.... ###..##.#. ..##.....# .###.#####	5

- Проверяем **обязательно** – на acmp.

```python
import os
from collections import deque

def count_gardens():
    base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    input_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_path = os.path.join(base_dir, 'txtf', 'output.txt')

    with open(input_path, 'r') as f:
        lines = [line.strip() for line in f.readlines() if line.strip()]

    if not lines:
        with open(output_path, 'w') as f:
            f.write('0')
        return

    N, M = map(int, lines[0].split())
    grid = [list(line.replace(' ', '')) for line in lines[1:N + 1]]

    visited = [[False for _ in range(M)] for _ in range(N)]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    count = 0

    for i in range(N):
        for j in range(M):
            if grid[i][j] == '#' and not visited[i][j]:
                queue = deque()
                queue.append((i, j))
                visited[i][j] = True

                while queue:
                    x, y = queue.popleft()

                    for dx, dy in directions:
                        nx, ny = x + dx, y + dy
                        if 0 <= nx < N and 0 <= ny < M:
                            if grid[nx][ny] == '#' and not visited[nx][ny]:
                                visited[nx][ny] = True
                                queue.append((nx, ny))
                count += 1

    with open(output_path, 'w') as f:
        f.write(str(count))

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

input.txt:

```
5 10
# # . . . . . . # .
. # . . # . . . # .
. # # # . . . . # .
. . # # . . . . # .
. . . . . . . . # .
```

output.txt:

```
3
```

Объяснение кода задания о грядках:
- В данной задаче необходимо подсчитать количество "грядок" на садовом участке, представимом в виде двумерной сетки, где грядки обозначены символом `#`, а незанятая территория — символом `.`. Грядка — это совокупность связанных по сторонам квадратов с символом `#`.

Структура кода:

1. **Импорт библиотек**:
   - Импортируем `os` для работы с файловой системой и `deque` из модуля `collections` для реализации очереди, которая будет использоваться в алгоритме обхода в ширину (BFS).
2. **Функция `count_gardens()`**:
   - Основная функция, отвечающая за чтение входных данных, поиск грядок и запись результата в выходной файл.
3. **Чтение входных данных**:
   - Открываем файл `input.txt` и считываем все строки.
   - Первая строка содержит размеры сетки `N` (количество строк) и `M` (количество столбцов).
   - Оставшиеся строки представляют собой саму сетку, где каждая строка разбивается на символы.
4. **Инициализация переменных**:
   - Создается двумерный массив `visited` для отслеживания уже посещенных квадратов, инициализированный значениями `False`.
   - Определяем список `directions` для перемещения по сетке (вверх, вниз, влево, вправо).
   - Создаем переменную `count` для подсчета количества грядок.
5. **Поиск грядок с помощью BFS**:
   - Двойной цикл проходит по всем клеткам сетки. Если клетка содержит `#` и не была посещена, начинается BFS:
     - Добавляем клетку в очередь и отмечаем её как посещённую.
     - Извлекаем клетку из очереди и проверяем всех её соседей в указанных направлениях.
     - Если соседняя клетка тоже содержит `#` и не была посещена, добавляем её в очередь и отмечаем как посещённую.
   - После завершения обработки всех соседей увеличиваем счётчик `count` на 1, что означает обнаруженную грядку.
6. **Запись результата**:
   - Результат, представляющий количество найденных грядок, записывается в файл `output.txt`.

Пример работы:
- Для входных данных:

```
5 10
.#....##..
#..###..#.
..#....#..
..##..#...
.......#..
```

- Сетка имеет 5 строк и 10 столбцов.

- Грядки:
    1. Грядка 1: клетки (0, 1), (1, 1) и (1, 2) образуют первую грядку.
    2. Грядка 2: составляют (1, 3), (1, 4), (1, 5) и (2, 3).
    3. Грядка 3: составляет (3, 3), (3, 4) и (1, 7).
    4. Отдельные участки на границе могут образовывать новые грядки.

- Таким образом, код считает количество грядок и выводит результат, например, `3` для примера выше.

Заключение:

- Код использует алгоритм обхода в ширину (BFS) для эффективного поиска всех связанных частей (грядок) в двумерной сетке, что соответствует условиям задачи. Решение работает в пределах заданных ограничений по времени и памяти.

Задание 14 : Автобусы

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 ($1 \le N \le 100$), номера деревень d и v, затем количество автобусных рейсов R ($0 \le R \le 10000$). 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.
- Пример:

input.txt	output.txt
3 1 3 4 1 0 2 5 1 1 2 3 2 3 3 5 1 1 3 10	5

- Проверяем **обязательно** – на acmp.

```python
import os
import heapq

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as f:
        N = int(f.readline().strip())
        d, v = map(int, f.readline().strip().split())
        R = int(f.readline().strip())

        bus_routes = []
        for _ in range(R):
            start, dep_time, end, arr_time = map(int, f.readline().strip().split())
            bus_routes.append((start, dep_time, end, arr_time))

    graph = {i: [] for i in range(1, N + 1)}
    for start, dep_time, end, arr_time in bus_routes:
        graph[start].append((dep_time, end, arr_time))

    min_time = {i: float('inf') for i in range(1, N + 1)}
    min_time[d] = 0
    priority_queue = [(0, d)]

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

        if current_time > min_time[current_village]:
            continue

        for dep_time, next_village, arr_time in graph[current_village]:
            if current_time <= dep_time:
                if arr_time < min_time[next_village]:
                    min_time[next_village] = arr_time
                    heapq.heappush(priority_queue, (arr_time, next_village))

    result = min_time[v]
    with open(output_path, 'w') as f:
        f.write(str(result) if result != float('inf') else '-1')

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

input.txt:

```
3
1 3
4
1 0 2 5
1 1 2 3
2 3 3 5
1 1 3 10
```

output.txt:

```
5
```

Объяснение кода задания о автобусах:

- В данной задаче необходимо найти минимальное время, когда Мария Ивановна может добраться из одной деревни в другую, используя автобусные рейсы. Для этого используется алгоритм, основанный на приоритетной очереди (минимальной куче), похожий на алгоритм Дейкстры, который позволяет находить кратчайшие пути в графе.

Структура кода:

1. **Импорт библиотек**:
   - Импортируем `os` для работы с файловой системой и `heapq` для реализации приоритетной очереди.
2. **Функция `task_scheduler()`**:
   - Основная функция, отвечающая за чтение входных данных, обработку автобусных маршрутов и поиск минимального времени.
3. **Чтение входных данных**:
   - Открываем файл `input.txt` и считываем количество деревень `N`, номера начальной `d` и конечной `v` деревней, а также количество автобусных рейсов `R`.
   - Затем считываются описания автобусных рейсов и сохраняются в список `bus_routes`.
4. **Структура графа**:
   - Создается граф в виде словаря, где ключ — это номер деревни, а значение — список маршрутов, исходящих из этой деревни. Каждый маршрут представлен кортежем `(время отправления, конечная деревня, время прибытия)`.
5. **Инициализация времени**:
   - Создается словарь `min_time`, который будет хранить минимальное время, необходимое для достижения каждой деревни, и инициализируется значением бесконечности для всех деревень, кроме стартовой, которая инициализируется нулем.
   - Инициализируется приоритетная очередь `priority_queue`, в которую добавляется стартовая деревня с временем 0.
6. **Поиск минимального времени**:
   - Запускается цикл, который работает, пока очередь не станет пустой.
   - Извлекается текущая деревня и текущее время из очереди. Если текущее время больше, чем уже известное минимальное время для этой деревни, продолжаем цикл.
   - Проходим по всем маршрутам, доступным из текущей деревни. Если текущее время меньше или равно времени отправления, проверяем, если время прибытия в следующую деревню меньше, чем уже известное, обновляем значение и добавляем новый маршрут в очередь.
7. **Запись результата**:
   - После завершения поиска выводится минимальное время для достижения деревни `v` или `-1`, если достигнуть её невозможно.

Пример работы:

- Для входных данных:

```
3
1 3
4
1 0 2 5
```

```
1 1 2 3
2 3 3 5
1 1 3 10
```

- 3 деревни: 1, 2 и 3.
- Нужно добраться из деревни 1 в 3.
- Множество маршрутов, включая:
  - Из 1 в 2 с отправлением в 0 и прибытием в 5.
  - Из 1 в 3 с отправлением в 1 и прибытием в 10.
  - Из 2 в 3 с отправлением в 3 и прибытием в 5.

- Мария Ивановна может воспользоваться маршрутом из 1 в 2 и потом из 2 в 3, что даст ей минимальное время.
- В результате программа вычисляет минимальное время и выдает его в выходной файл.

Заключение:

- Код реализует эффективный метод поиска кратчайшего пути с учетом времени на основе автобусных маршрутов, что позволяет решить задачу в заданных рамках по времени и памяти.

Задание 17 : Слабая K-связность

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.

```python
import os
from collections import deque

def find_min_k():
    base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    input_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_path = os.path.join(base_dir, 'txtf', 'output.txt')

    with open(input_path, 'r') as f:
        lines = f.readlines()

    N, M = map(int, lines[0].split())
    adj = [[] for _ in range(N + 1)]
    reverse_adj = [[] for _ in range(N + 1)]

    for line in lines[1:M + 1]:
        u, v = map(int, line.split())
        adj[u].append(v)
        reverse_adj[v].append(u)

    def is_weakly_k_connected(K):
        for start in range(1, N + 1):
            visited = [False] * (N + 1)
            q = deque()
            q.append((start, 0))
            visited[start] = True

            while q:
                node, violations = q.popleft()
                if violations > K:
                    continue

                for neighbor in adj[node]:
                    if not visited[neighbor]:
                        visited[neighbor] = True
                        q.append((neighbor, violations))

                for neighbor in reverse_adj[node]:
                    if not visited[neighbor] and violations + 1 <= K:
                        visited[neighbor] = True
                        q.append((neighbor, violations + 1))

            for i in range(1, N + 1):
                if i != start and not visited[i]:
                    return False
        return True

    low = 0
    high = N

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

    with open(output_path, 'w') as f:
        f.write(str(low))

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

input.txt:

```
3 2
1 2
1 3
```

output.txt:

```
1
```

Объяснение кода задания о слабой K-связности:

- В данной задаче нужно определить минимальное значение K для заданного графа, где граф представляет собой набор городов и односторонних дорог между ними. Граф считается слабо K-связным, если для любых двух различных городов можно добраться из одного в другой, нарушая правила проезда по дорогам не более K раз.

Структура кода:

1. **Импорт библиотек**:
   - Импортируем `os` для работы с файловой системой и `deque` из модуля `collections` для реализации очереди, которая поможет при обходе графа.
2. **Функция `find_min_k()`**:
   - Основная функция, которая выполняет все необходимые действия для определения минимального K.
3. **Чтение входных данных**:
   - Открываем файл `input.txt` и считываем количество городов `N` и количество дорог `M`.
   - Создаем списки смежности для хранения дорог в нормальном и обратном направлении (`adj` и `reverse_adj`).
4. **Заполнение списков смежности**:
   - Для каждой дороги, прочитанной из файла, добавляем направленное ребро в список смежности `adj` и обратное ребро в `reverse_adj`.
5. **Функция `is_weakly_k_connected(K)`**:
   - Проверяет, является ли граф слабо K-связным для данного значения K.
   - Для каждого города (начальной точки) выполняется обход в ширину (BFS):
     - Используется очередь `q` для хранения узлов и текущего количества нарушений (`violations`).
     - Если количество нарушений превышает K, дальнейшие нарушения не допускаются.
     - Проверяются как прямые соседи (по нормальным ребрам), так и обратные (по обратным ребрам).
     - Если после обхода остаются непосещенные города, функция возвращает `False`, иначе — `True`.
6. **Использование бинарного поиска для нахождения K**:
   - Настраиваем границы бинарного поиска: `low` равен 0 (ноль нарушений), `high` равен `N` (максимальное возможное количество нарушений).
   - В цикле бинарного поиска вычисляется среднее значение `mid`, проверяется, является ли граф слабо K-связным для этого значения.
   - Если да, уменьшаем верхнюю границу `high`, если нет — увеличиваем нижнюю `low`.
7. **Запись результата**:

- В выходной файл `output.txt` записывается минимальное значение K.

Пример работы:

- Для входных данных:

```
4 4
2 4
1 3
4 1
3 2
```

- Имеется 4 города и 4 дороги.
- Граф можно представить следующим образом:
  - Из 2 в 4
  - Из 1 в 3
  - Из 4 в 1
  - Из 3 в 2

- В результате, необходимо выяснить, сколько раз можно нарушать правила, чтобы обеспечить возможность проезда между любыми двумя городами.

Заключение:

- Код применяет алгоритм BFS для проверки достижимости между городами с учётом количества нарушений и использует бинарный поиск для нахождения минимального K. Это решение эффективно работает в заданных ограничениях по времени и памяти, что делает его подходящим для решения данной задачи.