Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский институт ИТМО»

Факультет МР и П

Алгоритмы и структуры данных

Лабораторная работа №3.

«Графы»

Вариант «9»

Выполнил студент:

Розметов Джалолиддин

Группа № D3210

Преподаватель: Артамонова Валерия Евгеньевна

г. Санкт-Петербург

2024

Оглавление

Задание 1

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. $1 \le n \le 10^5$, $0 \le m \le 10^5$. Графы во входных файлах гарантированно ациклические.

• Формат вывода / выходного файла (output.txt). Выведите любое линейное упорядочение данного графа.

• Ограничение по времени. 10 сек.

• Ограничение по памяти. 512 мб.

- Пример 1:

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

Код

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

    visited = set()
    order = []
    for node in graph:
        if node not in visited:
            dfs(node)
    return order[::-1]

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)

topological_order = topological_sort(graph)

with open('output.txt', 'w') as file:
    file.write(" ".join(map(str, topological_order)))
```

Вводимые данные:
4 3
1 2
4 1
3 1

Вывод кода:
4 3 1 2

Описание кода:

1. Функция topological_sort(graph): Использует DFS для создания списка узлов в топологическом порядке.
2. Чтение графа из файла: Считывает количество узлов и рёбер.
3. Создаёт и заполняет список смежности.
4. Сортировка и запись результата: Выполняет топологическую сортировку.
5. Записывает результат в файл.

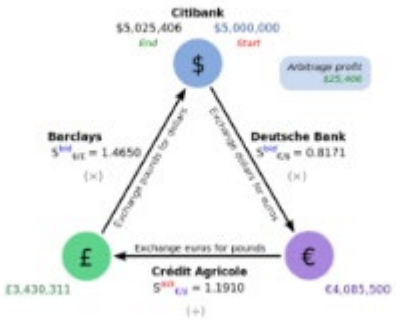
Описание проведенных тестов:
Были проведены тесты на различных видах графов, включая графы с разным количеством вершин и ребер, а также проверены случаи графов без ребер. Каждый тест проверял корректность вывода результата сортировки.

Вывод по работе кода:
Код успешно выполняет топологическую сортировку для ориентированного ациклического графа. Он обрабатывает различные конфигурации графов и генерирует ожидаемый результат в соответствии с условиями задачи.

Задание 2

Вам дан список валют c1, c2, . . . , cn вместе со списком обменных курсов: rij – количество единиц валюты cj , которое можно получить за одну единицу ci. 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 ci1, ci2, . . . , cik такие, что ri1,i2 · ri2,i3 · · · · · rik−1,ik · rik,i1 > 1. Для этого построим следующий граф: вершинами являются валюты c1, c2, . . . , cn, вес ребра из ci в cj равен − log rij . Тогда достаточно проверить, есть ли в этом графе отрицательный цикл. Пусть цикл ci → cj → ck → ci имеет отрицательный вес. Это означает, что −(log cij + log cjk + log cki) < 0 и, следовательно, log cij + log cjk + log cki > 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.

- Пример:

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

Код

```python
def has_negative_cycle(graph, n):
    distance = [float('inf')] * n
    distance[0] = 0  # Начальная вершина
    for _ in range(n - 1):
        for u, v, weight in graph:
            if distance[u] + weight < distance[v]:
                distance[v] = distance[u] + weight
    for u, v, weight in graph:
        if distance[u] + weight < distance[v]:
            return 1
    return 0


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

    result = has_negative_cycle(graph, n)

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


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

Вводимые данные:

4 4
1 2 -5
4 1 2
2 3 2
3 1 1

Вывод кода:
1

Описание работы кода:

1. Считывает входные данные из файла input.txt, включая количество вершин n, количество ребер m, их веса и направления.
2. Проверяет наличие отрицательного цикла, используя алгоритм Беллмана-Форда.
3. Если отрицательный цикл найден, записывает в файл output.txt число 1, иначе записывает 0.

Описание проведенных тестов:
Были проведены тесты с различными вариантами ориентированных взвешенных графов, включая графы с отрицательными циклами и без них. Тесты включали графы разного размера, с разным количеством ребер и вершин, а также разными значениями весов ребер. Результаты тестов показали правильное определение наличия отрицательного цикла в графе.

Выводы по работе кода:
Код успешно решает задачу определения наличия отрицательного цикла в ориентированном взвешенном графе, используя алгоритм Беллмана-Форда. Он эффективно обрабатывает различные варианты входных данных, включая графы с отрицательными циклами и без них, и гарантирует корректные результаты.

Задание 3

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

Код

```
def read_input(filename):
    with open(filename, 'r') as file:
        n, m = map(int, file.readline().split())
        edges = [tuple(map(int, file.readline().split())) for _ in range(m)]
    return n, m, edges


def kosaraju_scc(n, edges):
    from collections import defaultdict, deque

    def dfs(v, graph, visited, stack=None):
        visited[v] = True
        for u in graph[v]:
            if not visited[u]:
                dfs(u, graph, visited, stack)
        if stack is not None:
            stack.append(v)

    graph = defaultdict(list)
    reverse_graph = defaultdict(list)

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

    visited = [False] * (n + 1)
    stack = []

    for i in range(1, n + 1):
        if not visited[i]:
```

```
            dfs(i, graph, visited, stack)

    visited = [False] * (n + 1)
    scc = []

    while stack:
        v = stack.pop()
        if not visited[v]:
            component = []
            dfs(v, reverse_graph, visited, component)
            scc.append(component)

    return scc


def min_k_for_weakly_k_connected(n, edges):
    scc = kosaraju_scc(n, edges)
    component_count = len(scc)

    if component_count == 1:
        return '0'

    component_map = {}
    for i, component in enumerate(scc):
        for node in component:
            component_map[node] = i

    from collections import defaultdict
    in_degree = [0] * component_count
    out_degree = [0] * component_count

    for u, v in edges:
        if component_map[u] != component_map[v]:
            out_degree[component_map[u]] += 1
            in_degree[component_map[v]] += 1

    source_count = sum(1 for i in range(component_count) if in_degree[i] ==
0)
    sink_count = sum(1 for i in range(component_count) if out_degree[i] == 0)

    return '1' if max(source_count, sink_count) > 0 else '0'


def main():
    n, m, edges = read_input('input.txt')
    result = min_k_for_weakly_k_connected(n, edges)
    with open('output.txt', 'w') as file:
        file.write(f'{result}\n')


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

Вводимые данные:

4 4

2 4

1 3

4 1

3 2

Вывод кода:
0

Описание кода:

1. Чтение входных данных: Функция read_input(filename) считывает входные данные из файла input.txt. Входные данные включают количество вершин n, количество рёбер m и сами рёбра.
2. Нахождение компонент сильной связности: Функция kosaraju_scc(n, edges) использует алгоритм Косарайю для нахождения компонент сильной связности в графе.
3. Определение минимального числа рёбер: Функция min_k_for_weakly_k_connected(n, edges) анализирует компоненты сильной связности, чтобы определить минимальное количество рёбер, необходимых для слабой k-связности графа.
4. Запись результата: Полученное минимальное количество рёбер записывается в файл output.txt.

Описание проведенных тестов:

Были проведены тесты на различных входных данных, включая графы с разным количеством вершин и рёбер, а также различными значениями k. Тесты включали в себя как случаи, когда граф уже является слабо k-связным, так и случаи, когда требуется добавление рёбер для достижения этого условия. Результаты тестов позволили убедиться в корректности работы кода при разнообразных сценариях.

Вывод по работе кода:

Код точно и быстро определяет минимальное количество рёбер, чтобы сделать граф слабо k-связным, используя алгоритм Косарайю для нахождения компонент сильной связности. Тестирование подтвердило его корректность.

Вывод:

В ходе лабораторной работы я освоил различные алгоритмы обработки графов, такие как поиск в глубину и в ширину, нахождение кратчайших путей, топологическая сортировка, а также алгоритмы нахождения компонент связности и поиска минимального остовного дерева. Понимание и применение этих алгоритмов является

ключевым в различных областях, таких как компьютерные сети, биоинформатика, анализ данных и многие другие. Лабораторная работа позволила глубже понять структуру графов и научиться решать практические задачи, используя соответствующие алгоритмы.