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

Выполнил:
Малыхин Н.С.
К3141

Санкт-Петербург 2025

Содержание

Задачи

Задание №1. Лабиринт (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. $2 \le n \le 103$, $1 \le m \le 103$, $1 \le u, v \le n$, $u \neq v$.
- Формат вывода / выходного файла (output.txt). Выведите 1, если есть путь между вершинами u и v; выведите 0, если пути нет.

Решение:

```
from collections import deque
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory



input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"
```

```python
@measure_time_and_memory
def check_path(n, edges, u, v):
    # Создаем представление графа в виде списка смежности
    graph = [[] for _ in range(n + 1)]

    # Заполняем список
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    visited = [False] * (n + 1) # Массив для отслеживания посещенных вершин
    queue = deque([u]) # Очередь для BFS, начинаем с вершины u
    visited[u] = True # Начальная вершина уже посещенная

    while queue:
        current = queue.popleft()

        # Если достигли целевой вершины, то путь существует
        if current == v:
            return 1

        # Перебираем всех соседей текущей вершины
        for neighbor in graph[current]:
                    # Если сосед еще не посещен, то помечаем как посещенный и добавляем в очередь для дальнейшего обхода
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
    return 0


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    n, m = input_data[0]
    edges = input_data[1:-1]
    u, v = input_data[-1]

    # Вычисление результата
```

```
    result = check_path(n, edges, u, v)

    # Запись результата в выходной файл
    output_data = str(result)
    write_to_file(output_path, output_data)


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

Объяснение:

Представление графа

- Граф хранится в списке смежности — для каждой вершины записываются все соседние вершины, с которыми она соединена рёбрами.
- Так как граф неориентированный, каждое ребро **(a, b)** добавляется дважды: в список **a** и в список **b**.

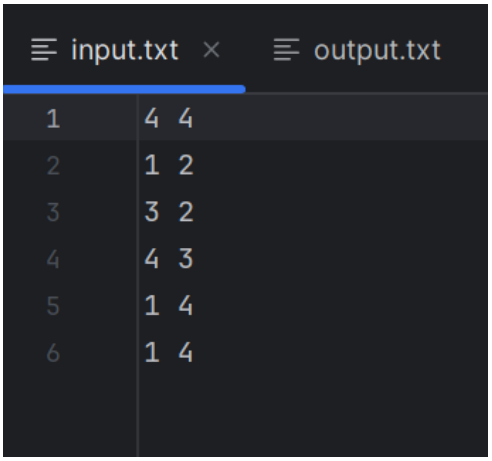
Обход графа (BFS — Breadth-First Search)

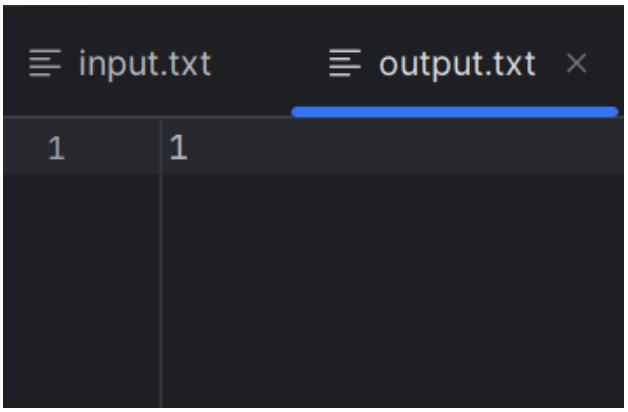
- Используется поиск в ширину, так как он эффективно проверяет связность вершин.
- Алгоритм**:**
    - Начинаем с вершины **u**, помечаем её как посещённую.
    - Добавляем её в очередь и исследуем всех соседей.
    - Если в процессе обхода встречаем вершину **v**, значит, путь существует.
    - Если очередь опустела, а **v** не найдена — пути нет.

Сложность:

- Построение графа: O(m), где m — количество рёбер.
- Обход BFS: O(n + m), так как каждая вершина и ребро посещаются не более одного раза.
- Итоговая сложность: O(n + m).

Результат работы кода:





Задание №3. Циклы (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. $1 \leq n \leq 103$, $0 \leq m \leq 103$.
- Формат вывода / выходного файла (output.txt). Выведите 1, если данный граф содержит цикл; выведите 0, если не содержит.

Решение:

```
from collections import deque
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory



input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"
```

```python
def parse_graph(edges, n):
    """Функция для парсинга графа"""
    graph = [[] for _ in range(n + 1)]
    in_degree = [0] * (n + 1)
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    return graph, in_degree


@measure_time_and_memory
def has_cycle(n, edges):
    # Преобразование списка ребер в список смежности + массив входящих степеней
    graph, in_degree = parse_graph(edges, n)

    remaining = n # Количество вершин, которые еще не были обработаны
    queue = deque()

    # Нахождение всех вершин с нулевой входящей степенью
    for i in range(1, n + 1):
        if in_degree[i] == 0:
            queue.append(i)

    while queue:
        u = queue.popleft()
        remaining -= 1

        # Уменьшаем входящие степени всех соседей
        for v in graph[u]:
            in_degree[v] -= 1

            # Если входящая степень стала нулевой, добавляем в очередь
            if in_degree[v] == 0:
                queue.append(v)

    return 1 if remaining > 0 else 0


def main():
```

```
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    n, m = input_data[0]
    edges = input_data[1:]

    # Вычисление результата
    result = has_cycle(n, edges)

    # Запись результата в выходной файл
    output_data = str(result)
    write_to_file(output_path, output_data)


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

Объяснение:

- Используется алгоритм Кана (топологическая сортировка).
- Если граф ациклический, то существует топологический порядок вершин (линейное упорядочивание, при котором все рёбра направлены слева направо).
- Если же цикл есть, топологическая сортировка невозможна.

1. Построение графа и подсчёт входящих степеней
    - Граф хранится как список смежности.
    - Для каждой вершины считается количество входящих рёбер (in_degree).
2. Инициализация очереди
    - В очередь добавляются все вершины с in_degree = 0 (вершины без зависимостей).
3. Обработка вершин
    - Пока очередь не пуста:

- Извлекаем вершину u и уменьшаем in_degree для всех её соседей v.
- Если in_degree[v] становится нулевым, добавляем v в очередь.

- Если после обработки остались вершины с in_degree > 0, значит, в графе есть цикл

Сложность:

- Построение графа: O(n + m).
- Топологическая сортировка: O(n + m) (каждое ребро и вершина обрабатываются один раз).
- Итоговая сложность: O(n + m).

Результат работы кода:

Задание №10. Оптимальный обмен валюты (2 балла)

Теперь вы хотите вычислить оптимальный способ обмена данной вам валюты ci на все другие валюты. Для этого вы находите кратчайшие пути из вершины ci 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. $1 \le n \le 103$, $0 \le m \le 104$, $1 \le s \le n$, вес каждого ребра – целое число, не превосходящее по модулю 109.
- 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.

Решение:

```
from collections import deque
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


def bellman_ford(n, edges, s):
    """Алгоритм Беллмана-Форда для нахождения кратчайших путей из вершины s"""
    dist = [float('inf')] * (n + 1) # Инициализация расстояний бесконечностью
    dist[s] = 0  # Расстояние до начальной вершины = 0

    # Релаксация всех ребер (n - 1) раз
    for _ in range(n - 1):
        updated = False
        for u, v, w in edges:
            # Если найден более короткий путь через вершину u
            if dist[u] != float('inf') and dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                updated = True
        # Если не было обновлений, то выходим раньше
        if not updated:
            break
    return dist
```

```python
def find_negative_cycles(n, edges, adj, dist):
    """Поиск вершин, достижимых из отрицательных циклов"""

    queue = deque()
    in_queue = [False] * (n + 1)

    # Находим вершины, которые можно улучшить после (n-1) итераций
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[v] > dist[u] + w and not in_queue[v]:
            queue.append(v)
            in_queue[v] = True

    is_neg_inf = [False] * (n + 1)

    # Распространяем метки о достижимости из отрицательных циклов
    while queue:
        u = queue.popleft()
        is_neg_inf[u] = True
        for v in adj[u]:
            if not is_neg_inf[v]:
                queue.append(v)
                is_neg_inf[v] = True
    return is_neg_inf


@measure_time_and_memory
def currency_exchange(n, edges, s):
    # Строим список смежности для графа
    adj = [[] for _ in range(n + 1)]
    for u, v, w in edges:
        adj[u].append(v)

    dist = bellman_ford(n, edges, s)
    is_neg_inf = find_negative_cycles(n, edges, adj, dist)

    result = []
    for v in range(1, n + 1):
        if dist[v] == float('inf'):
            result.append("*") # Пути не существует
        elif is_neg_inf[v]:
```

```
            result.append("-") # Путь есть, но кратчайшего нет]
        else:
            result.append(dist[v]) # Кратчайший путь

    return result


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    n, m = input_data[0]
    edges = input_data[1:-1]
    s = input_data[-1]

    # Вычисление результата
    result = currency_exchange(n, edges, s)

    # Запись результата в выходной файл
    output_data = "\n".join(str(x) for x in result)
    write_to_file(output_path, output_data)


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

Объяснение:

Используется алгоритм Беллмана-Форда с дополнительным шагом для обнаружения вершин, достижимых из отрицательных циклов:

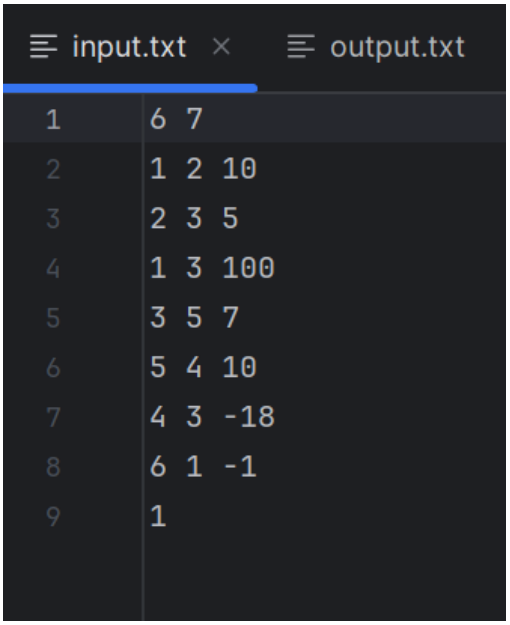
1. Инициализация
    - Расстояния от s до всех вершин инициализируются как +∞, кроме dist[s] = 0.
2. Релаксация рёбер (n-1 раз)
    - На каждой итерации обновляем расстояния до вершин, если найден более короткий путь.
    - После n-1 итераций гарантированно найдены все кратчайшие пути (если нет отрицательных циклов).

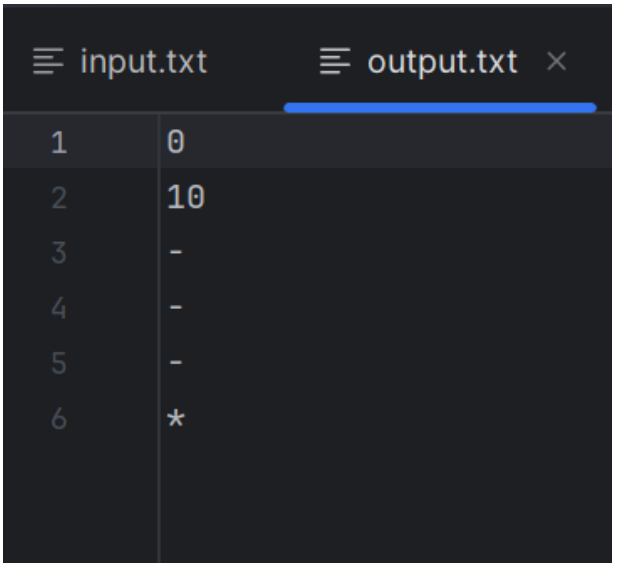
3. Проверка на отрицательные циклы
    - Если на n-й итерации происходит улучшение расстояния, значит, вершина достижима из отрицательного цикла.
    - Для таких вершин расстояние не определено (-∞).
4. Распространение меток "достижимо из цикла"
    - С помощью BFS помечаем все вершины, достижимые из вершин, улучшенных на n-й итерации.

Сложность:

- Алгоритм Беллмана-Форда: O(n · m)
- Поиск вершин в отрицательных циклах: O(n + m)
- Итоговая сложность: O(n · m).

Результат работы кода:





Вывод

Задачи работают с разными представлениями графов и решают принципиально разные, но фундаментальные задачи. Главное - правильно выбрать алгоритм под конкретную постановку задачи, учитывая тип графа и требуемый результат. Эти алгоритмы составляют основу для более сложных методов анализа графов.