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 24

Выполнил:
Чан Тхи Лиен
К3140

Проверил:
Петросян.А.М

Санкт-Петербург
2025 г.

Содержание отчета

Цель работы

В данной лабораторной работе изучаются графы, поиск в глубину, поиск в ширину, алгоритм Дейкстры, алгоритм Беллмана-Форда.

Задачи по варианту

Задание 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.

1. Функция **topological_sort**:

```python
def topological_sort(n, edges):  7 usages
    indegree = [0] * (n + 1)
    graph = [[] for _ in range(n + 1)]
    # Заполнение данных
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    # Очередь для вершин с нулевой степенью входа
    queue = deque()

    # Заполнение очереди
    for i in range(1, n + 1):
        if indegree[i] == 0:
            queue.append(i)
    result = []
    # Основной цикл
    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return result if len(result) == n else []
```

- **indegree[i]** — массив входных степеней (**in-degree**) вершин.
- **indegree[i]** — сколько рёбер входит в вершину **i**.
- **graph** — список смежности для хранения графа.

❖ Заполнение списка смежности и входных степеней:

- Добавляем ребро **u** → **v** в **graph**.
- Увеличиваем входную степень вершины **v**, так как в неё ведёт ребро.

❖ Очередь вершин с **in-degree == 0:**

- В очередь (**queue**) добавляем вершины, у которых нет входных рёбер (**in-degree == 0**).
- Они могут быть началом топологического порядка.

❖ Основной цикл алгоритма Кана:

- Берем вершину из очереди, добавляем в **result**.
- Удаляем её из графа, уменьшая **in-degree** у соседей.
- Если у соседа **in-degree** стал **0**, добавляем его в очередь.
- Повторяем процесс, пока очередь не пуста.
- Если в **result** ровно **n** элементов, то топологическая сортировка возможна.
- Если меньше (**len(result) < n**), значит, в графе есть цикл, и топологическая сортировка невозможна → возвращаем **[]**.

2. Функция для выполнения задачи.

```
def task4():  1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n, m = map(int, data[0].split())
    edges = [tuple(map(int, data[i+1].split())) for i in range(m)]

    result = topological_sort(n, edges)


    write_file(PATH_OUTPUT, ' '.join(map(str, result)) + '\n')

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task4()
```

3. Проведенные тесты.

- input.txt:

```
5 7
2 1
3 2
3 1
4 3
4 1
5 2
5 3
```

- output.txt:

```
4 5 3 2 1
```

```
Время работы: 0.017637100012507 29 секунд
память использовать: 0.011719 MB
```

4. Запись теста.

❖ Когда использовать этот алгоритм?

- Построение зависимостей (например, порядок выполнения задач, компиляция кода).
- Анализ графов без циклов.
- Не работает для циклических графов! Если граф содержит цикл, вернётся **[]**.

❖ **O(V + E)** — линейная сложность (проходим по всем вершинам и рёбрам).

❖ Очень эффективен для DAG.

Задание 10. Оптимальный обмен валюты

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.

1. Функция **bellman_ford**:

```
def bellman_ford(n, edges, s):  7 usages
    INF = float('inf')
    dist = [INF] * (n + 1)
    dist[s] = 0
    reachable = [False] * (n + 1)
    reachable[s] = True

    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
                reachable[v] = True

    in_negative_cycle = [False] * (n + 1)
    for _ in range(n):
        for u, v, w in edges:
            if dist[u] < INF and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                in_negative_cycle[v] = True
                reachable[v] = True
```

```
    for _ in range(n):
        for u, v, w in edges:
            if in_negative_cycle[u]:
                in_negative_cycle[v] = True

    result = []
    for i in range(1, n + 1):
        if not reachable[i]:
            result.append('*')
        elif in_negative_cycle[i]:
            result.append('-')
        else:
            result.append(str(dist[i]) if dist[i] != INF else '*')

    return result
```

- **dist[i]** — массив расстояний от **s** до **i**.
- Все расстояния инициализируются бесконечностью (**INF**), кроме **s** (начальная вершина, её расстояние **0**).

- **reachable[i]** — массив, который отмечает, достижима ли вершина **i** из **s**.

❖ Основной алгоритм Беллмана-Форда:

- Запускаем **n - 1** итераций (по свойству, что путь в худшем случае может содержать **n - 1** рёбер).
- Если **dist[u]** не бесконечность, обновляем **dist[v] = dist[u] + w**, если нашли более короткий путь.
- Отмечаем **v** как достижимую вершину.

❖ Обнаружение отрицательных циклов:

- Запускаем ещё n итераций.
- Если после **n - 1** итераций ещё можно улучшить путь, значит, есть отрицательный цикл → помечаем **v** как принадлежащую отрицательному циклу (**in_negative_cycle[v] = True**).

❖ Распространение информации об отрицательных циклах:

- Если вершина **u** принадлежит отрицательному циклу, все её потомки (**v**) тоже оказываются в отрицательном цикле.

❖ Формирование результата:

- Если вершина **недостижима**, добавляем **'*'**.
- Если вершина **в отрицательном цикле**, добавляем **'-'**.
- Иначе, добавляем её расстояние от **s**.

2. Функция для выполнения задачи.

```
def task10():
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n, m = map(int, data[0].split())
    s = int(data[-1])

    edges = [tuple(map(int, data[i+1].split())) for i in range(m)]

    result = bellman_ford(n, edges, s)


    write_file(PATH_OUTPUT, "\n".join(result) + "\n")

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task10()
```

3. Проведенные тесты.

- input.txt:

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

- output.txt:

```
-
-
-
0
*
```

```
Время работы: 0.011125899996841326 секунд
память использовать: 0.023438 MB
```

4. Запись теста.

❖ Когда использовать этот алгоритм?

- Граф может содержать рёбра с отрицательными весами.
- Нужно проверить наличие отрицательных циклов.
- Если нет отрицательных весов, лучше использовать Дейкстру (**O((V + E) log V)**), так как Беллман-Форд медленный.

❖ **O(V × E)** — из-за **n-1 + n + n** итераций по рёбрам.

❖ Медленный для больших графов, но незаменим, если есть отрицательные веса или циклы.

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***).

1. Функция

```
def min_time_to_reach_villages(N, d, v, R, trips):  8 usages
    # Инициализируем граф
    graph = {i: [] for i in range(1, N + 1)}
    # Заполняем граф автобусными рейсами
    for start_village, start_time, end_village, end_time in trips:
        graph[start_village].append((start_time, end_village, end_time))
    # Приоритетная очередь для Dijkstra
    pq = []
    heapq.heappush(pq, (0, d))  # (время, деревня)
    # Время минимального достижения для каждой деревни
    min_time = [float('inf')] * (N + 1)
    min_time[d] = 0
    while pq:
        current_time, current_village = heapq.heappop(pq)

        # Если уже лучшее время
        if current_time > min_time[current_village]:
            continue
        # Обрабатываем все автобусные рейсы из текущей деревни
        for start_time, end_village, end_time in graph[current_village]:
            # Если мы можем сесть на автобус
            if current_time <= start_time:
                if end_time < min_time[end_village]:
                    min_time[end_village] = end_time
                    heapq.heappush(pq, (end_time, end_village))
    # Проверяем время достижения деревни v
    return min_time[v] if min_time[v] != float('inf') else -1
```

- Граф представлен в виде списка смежности, где для каждой деревни хранятся доступные автобусные рейсы.
- Каждый рейс хранится как **(start_time, end_village, end_time)**.
- Используется приоритетная очередь **(heap) pq** для обработки деревень по времени.
- **min_time[i]** — минимальное время прибытия в деревню **i** (изначально бесконечность, кроме **d**, где **0**).

❖ Основной цикл Дейкстры:

- Обрабатываем деревни в порядке минимального времени прибытия.
- Если **current_time** хуже уже найденного **min_time[current_village]**, пропускаем итерацию.

❖ Обработка доступных автобусных рейсов:

- Проверяем все доступные рейсы из текущей деревни.
- Если текущее время (**current_time**) позволяет успеть на автобус (**current_time <= start_time**), проверяем, можно ли улучшить **min_time[end_village]**.
- Если **да**, обновляем **min_time[end_village]** и добавляем в очередь.
- Если **min_time[v] == ∞**, значит, нет пути → возвращаем **-1**.
- Иначе возвращаем минимальное время прибытия в деревню **v**.

2. Функция для выполнения задачи.

```
def task14():
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    N = int(data[0].strip())
    d, v = map(int, data[1].strip().split())
    R = int(data[2].strip())
    trips = [tuple(map(int, data[i+3].strip().split())) for i in range(R)]

    result = min_time_to_reach_villages(N, d, v, R, trips)

    write_file(PATH_OUTPUT, str(result) + '\n')

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task14()
```

3. Проведенные тесты.

- 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
```

```
Время работы: 0.010427100001834333 секунд
память использовать: 0.027344 MB
```

4. Запись теста.

❖ Когда использовать этот алгоритм?

- Есть временные ограничения (время отправления автобусов).
- Нужен кратчайший путь по времени, а не по количеству остановок.
- Не учитывает стоимость проезда.
- Не применим для поиска кратчайшего пути по количеству пересадок.

❖ **O(R log R)**, так как обрабатываем **R** рейсов в приоритетной очереди.

❖ В худшем случае (если **R ≈ N²**), сложность **O(N² log N)**.

Дополнительные задачи

Задание 3. Циклы

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.

1. Функция **has_cycle**:

```python
def has_cycle(n, edges):  11 usages
    # Состояния: 0 = не посещена, 1 = в процессе посещения, 2 = полностью обработана
    state = [0] * (n + 1)
    graph = [[] for _ in range(n + 1)]

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

    def dfs(node):
        if state[node] == 1:  # цикл найден
            return True
        if state[node] == 2:  # уже обработана
            return False

        state[node] = 1  # помечаем как в процессе посещения
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True

        state[node] = 2  # помечаем как полностью обработан
        return False

    # Проверяем все вершины, так как граф может быть не связным
    for i in range(1, n + 1):
        if state[i] == 0:  # если не посещена
            if dfs(i):
                return 1  # найден цикл

    return 0
```

- **state[i]** — массив, который отслеживает состояние каждой вершины.
- **graph** — список смежности для представления графа.

❖ Для каждой пары **(u, v)** добавляем **v** в список соседей **u**, так как граф ориентированный.

❖ Функция **DFS** для поиска цикла:

- Если **state[node] == 1**, значит, мы снова встретили вершину в процессе обработки → есть цикл.

- Если **state[node] == 2**, значит, вершина уже проверена и в ней нет цикла.
- Рекурсивно вызываем **dfs** для всех соседей **node**.
- Если все соседи пройдены и цикла не найдено, то **node** помечается как **2** (полностью обработана).

❖ Запускаем **dfs** для каждой непосещённой вершины, так как граф может быть несвязным (содержать несколько компонент).

- Если **dfs(i)** находит цикл, сразу возвращаем **1**.
- Если проверены все вершины и цикла нет, возвращаем **0**.

2. Функция для выполнения задачи.

```
def task3():  1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n, m = map(int, data[0].split())
    edges = [tuple(map(int, data[i+1].split())) for i in range(m)]

    result =has_cycle(n, edges)



    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task3()
```

3. Проведенные тесты.

- input.txt:

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

- output.txt:

```
1
```

```
Время работы: 0.010025800002040341 секунд
память использовать: 0.015625 MB
```

4. Запись теста.

Этот метод эффективен для обнаружения циклов в ориентированном графе:

- Позволяет быстро выявлять циклы без избыточного хранения информации.
- Работает за **O(V + E)**, где **V** — вершины, **E** — рёбра.
- Используется в алгоритме **Тарьяна** для поиска компонент сильной связности.

Задание 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.

1. Функция **bfs**:

```
def bfs(sx, sy, queen_pos, N, M, L, garden):  1 usage
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    queue = deque([(sx, sy, 0)])
    visited = set()
    visited.add((sx, sy))

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

        if (x, y) == queen_pos:
            return time

        if time < L:
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if (1 <= nx < N - 1) and (1 <= ny < M - 1) and (nx, ny) not in visited and garden[nx][ny] == '0':
                    visited.add((nx, ny))
                    queue.append((nx, ny, time + 1))

    return float('inf')
```

- Определяем 4 направления движения: вверх, вниз, влево, вправо.
- Очередь **queue** содержит кортежи **(координаты, текущее время)**.
- **visited** хранит уже посещённые клетки, чтобы избежать повторного прохода.

❖ Основной цикл BFS:

- Извлекаем текущую позицию **(x, y)** и **time** (шаги до сюда).
- Если дошли до **queen_pos**, возвращаем **time** (оптимальное время).
- Двигаемся в 4 направлениях.
- Проверяем:

- Клетка в границах сада (**1 ≤ nx < N - 1, 1 ≤ ny < M - 1**).
- Не посещали (**(nx, ny) not in visited**).
- Проходимая клетка (**garden[nx][ny] == '0'**).

- Если все условия выполняются, добавляем клетку в очередь с увеличенным временем **time + 1**.
- Если вышли из цикла, значит, невозможно добраться до королевы за **L** шагов → возвращаем ∞.

2. Функция для выполнения задачи.

```
def task4():  1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    N, M = map(int, data[0].split())
    garden = [data[i+1].strip() for i in range(N)]
    Qx, Qy, L = map(int, data[N+1].strip().split())
    queen_pos = (Qx-1, Qy-1)

    total_pendants = 0

    for j in range(4):
        Ax, Ay, Pa = map(int, data[N+j+2].strip().split())
        time_to_reach = bfs(Ax-1, Ay-1, queen_pos, N, M, L, garden)
        if time_to_reach <= L:
            total_pendants += Pa

    write_file(PATH_OUTPUT, str(total_pendants) + '\n')

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task4()
```

3. Проведенные тесты.

- input.txt:

```
5 5
11111
10001
10111
10101
11111
4 4 10
2 2 1
2 2 2
2 2 3
2 2 4
```

- output.txt:

```
0
```

```
Время работы: 0.00964390000444837 секунд
память использовать: 0.027344 MB
```

4. Запись теста.

❖ Когда использовать BFS?

- Гарантированно найдёт кратчайший путь в невзвешенных графах.
- Хорошо подходит для поиска в решётках (**N** × **M**).
- Не подходит для поиска кратчайшего пути с разными весами рёбер (лучше Дейкстра).
- Потребляет много памяти, если **N, M** очень велики.

❖ **O(N × M)** в худшем случае (если сад полностью свободен, алгоритм проверяет все **N** × **M** клетки).

❖ Обычно быстрее, если есть препятствия, так как они сокращают количество проверяемых путей.

Вывод:

- Выполнение всех задач.
- Работа с алгоритмами: **Тарьяна, Дейкстра, Беллман-Форд**.