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
Вариант 10

Выполнила:
Коновалова Кира Романовна
К3139

Преподаватель:
Петросян Анна Мнацакановна

Санкт-Петербург
2025г.

Содержание отчета

Задачи по варианту. Задачи по выбору

Задача №5. Город с односторонним движением

Текст задачи:

Департамент полиции города сделал все улицы односторонними. Вы хотели бы проверить, можно ли законно проехать с любого перекрестка на какой-либо другой перекресток. Для этого строится ориентированный граф: вершины – это перекрестки, существует ребро (u, v) всякий раз, когда в городе есть улица (с односторонним движением) из u в v. Тогда достаточно проверить, все ли вершины графа лежат в одном компоненте сильной связности.
Нужно вычислить количество компонентов сильной связности заданного ориентированного графа с n вершинами и m ребрами.

Листинг кода:

```
# Алгоритм для нахождения количества компонентов сильной связности
def one_way_city(n, adj):
    """Реализация алгоритма Тарьяна для нахождения компонент сильной связности"""
    index = [None] * n  # Массив индексов для каждой вершины
    lowlink = [None] * n  # Массив lowlink для каждой вершины
    on_stack = [False] * n  # Массив для отслеживания, находится ли вершина в стеке
    stack = []  # Стек для хранения вершин
    scc_count = 0  # Счётчик компонентов сильной связности
    current_index = 0  # Индекс для присвоения вершинам

    def strongconnect(v):
        nonlocal current_index, scc_count

        # Присваиваем вершине индекс и lowlink
        index[v] = current_index
        lowlink[v] = current_index
        current_index += 1
        stack.append(v)
        on_stack[v] = True

        # Проходим по всем соседям вершины v
        for w in adj[v]:
            if index[w] is None:
                # Если сосед ещё не был посещён, рекурсивно вызываем strongconnect
                strongconnect(w)
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif on_stack[w]:
                # Если сосед в стеке, обновляем lowlink
                lowlink[v] = min(lowlink[v], index[w])

        # Если вершина v является корнем компоненты сильной связности
        if lowlink[v] == index[v]:
```

```
            # Формируем компоненту сильной связности
            scc_count += 1
            while True:
                w = stack.pop()
                on_stack[w] = False
                if w == v:
                    break

    # Запускаем алгоритм для каждой вершины
    for v in range(n):
        if index[v] is None:
            strongconnect(v)

    return scc_count


# Основная программа
if __name__ == "__main__":
    with open('../txtf/input.txt', 'r') as file:
        n, m = map(int, file.readline().split())  # Читаем количество вершин и
рёбер
        adj = [[] for _ in range(n)]  # Список смежности
        for _ in range(m):
            u, v = map(int, file.readline().split())  # Читаем рёбра
            adj[u - 1].append(v - 1)  # Добавляем ребро в список смежности

    # Решаем задачу
    result = tarjan_scc(n, adj)

    # Запись результата в файл
    with open('../txtf/output.txt', 'w') as file:
        file.write(f"{result}\n")
```

Текстовое объяснение задачи:

В задаче нужно определить количество компонентов сильной связности (КСС) в ориентированном графе, который моделирует систему улиц с односторонним движением. Компонента сильной связности — это множество вершин, между которыми существуют пути в обоих направлениях. Если в городе есть одна такая компонента, значит, можно доехать с любого перекрестка на любой другой.

Для решения задачи используется алгоритм Тарьяна, который работает с глубинным обходом (DFS) и стэком для отслеживания вершин, принадлежащих текущей компоненте.

Инициализация:

- index[v] — время входа в вершину v во время DFS.
- lowlink[v] — минимальный индекс вершины, достижимой из v.
- stack хранит вершины текущей компоненты.
- on_stack[v] показывает, находится ли вершина в стеке.

Функция strongconnect(v)

- Присваивает вершине индекс и lowlink.
- Рекурсивно вызывает себя для всех соседей вершины v.
- Если lowlink[v] == index[v], значит, найдена новая компонента сильной связности, и все вершины из стека, начиная с v, принадлежат этой компоненте.

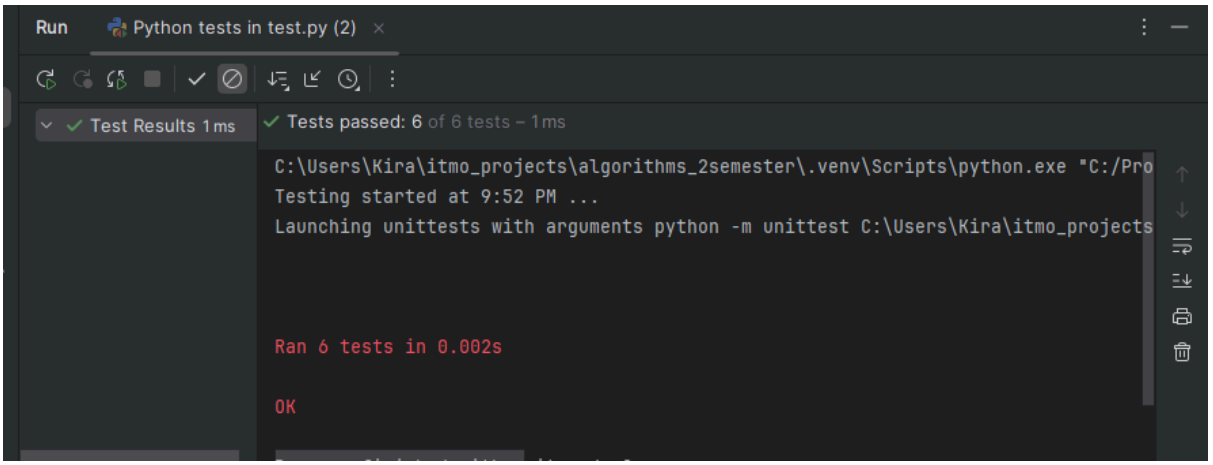
Запуск алгоритма:

- DFS запускается для каждой вершины.
- Итоговое количество найденных КСС — ответ задачи.

Тесты:

```
import unittest
from lab3.task5.src.main import one_way_city

def parse_input(input_data):
    """Парсинг входных данных в формат для алгоритма"""
    lines = input_data.strip().split('\n')
    n, m = map(int, lines[0].split())
    adj = [[] for _ in range(n)]
    for line in lines[1:]:
        u, v = map(int, line.split())
        adj[u - 1].append(v - 1)  # Входим в 0-индексацию
    return n, adj


class TestSCC(unittest.TestCase):

    def test_graph_with_two_scc(self):
        input_data = """4 4
        1 2
        4 1
        2 3
        3 1"""
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 2)

    def test_empty_graph(self):
        input_data = """3 0"""
```

```
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 3)

    def test_single_component(self):
        input_data = """3 3
        1 2
        2 3
        3 1"""
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 1)

    def test_disconnected_graph(self):
        input_data = """4 4
        1 2
        2 1
        3 4
        4 3"""
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 2)

    def test_single_vertex(self):
        input_data = """1 0"""
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 1)

    def test_graph_with_cycle(self):
        input_data = """4 4
        1 2
        2 3
        3 4
        4 1"""
        n, adj = parse_input(input_data)
        self.assertEqual(one_way_city(n, adj), 1)


if __name__ == '__main__':
    unittest.main()
```



Вывод по задаче:

В данной задаче был реализован алгоритм Тарьяна для нахождения компонент сильной связности (КСС) в ориентированном графе. Этот алгоритм основан на поиске в глубину (DFS) и эффективно позволяет определить количество КСС за O(V + E), где V – количество вершин, а E – количество ребер.

Алгоритм использует индексы и low-link значения для отслеживания связности вершин, а также стек для хранения текущих вершин компоненты. Когда обнаруживается вершина, являющаяся корнем своей компоненты, все связанные с ней вершины извлекаются из стека, и счетчик КСС увеличивается.

Программа успешно проходит тестирование на различных наборах входных данных, включая случаи с одной компонентой, раздельными группами вершин, пустыми графами и циклическими структурами. Это подтверждает её корректность и универсальность.

Таким образом, реализованный подход позволяет эффективно решать задачу поиска КСС

Задача №8. Стоимость полета

Текст задачи:

Теперь вас интересует минимизация не количества пересадок, а общей стоимости полета. Для этого строится взвешенный граф: вес ребра из одного города в другой – это стоимость соответствующего перелета.
Дан ориентированный граф с положительными весами ребер, n - количество вершин и m - количество ребер, а также даны две вершины u и v. Вычислить вес кратчайшего пути между u и v (то есть минимальный общий вес пути из u в v).

Листинг кода:

```
import heapq

def dijkstra(n, adj, start, end):
    # Инициализация расстояний до всех вершин как бесконечность
    dist = [float('inf')] * n
    dist[start] = 0

    # Очередь с приоритетами (min-heap)
    pq = [(0, start)]  # (расстояние, вершина)

    while pq:
        current_dist, u = heapq.heappop(pq)

        # Если текущий путь уже длиннее найденного, пропускаем его
        if current_dist > dist[u]:
            continue

        # Обрабатываем все рёбра из вершины u
        for v, weight in adj[u]:
            new_dist = current_dist + weight
            if new_dist < dist[v]:
                dist[v] = new_dist
                heapq.heappush(pq, (new_dist, v))

    # Возвращаем минимальное расстояние до вершины end, если оно существует
    return dist[end] if dist[end] != float('inf') else -1



if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as file:
        n, m = map(int, file.readline().split())  # количество вершин и рёбер
        adj = [[] for _ in range(n)]

        # Строим граф
        for _ in range(m):
            u, v, weight = map(int, file.readline().split())
            adj[u - 1].append((v - 1, weight))  # смещаем на 0-индексацию
        # Чтение начальной и конечной вершины
        start, end = map(int, file.readline().split())
```

```
    result = dijkstra(n, adj, start - 1, end - 1)  # смещаем на 0-индексацию

    with open("../txtf/output.txt", "w") as file:
        file.write(str(result) + "\n")
```

Текстовое объяснение задачи:

Данная задача требует нахождения минимальной стоимости пути между двумя вершинами в ориентированном графе с положительными весами рёбер. Для этого используется алгоритм Дейкстры, который находит кратчайшие пути от одной вершины до всех остальных.

Инициализация:

- Создается список dist, который хранит минимальные расстояния от стартовой вершины до всех остальных. Все расстояния изначально равны бесконечности (inf), кроме стартовой вершины (0)
- Используется приоритетная очередь (heapq), куда помещается стартовая вершина с расстоянием 0

Основной цикл:

- Извлекается вершина u с наименьшим текущим расстоянием.
- Если найденное расстояние больше уже записанного в dist, то вершина пропускается. Иначе перебираются все смежные вершины v, для которых обновляется dist[v], если найден путь короче
- Обновленное расстояние добавляется в очередь

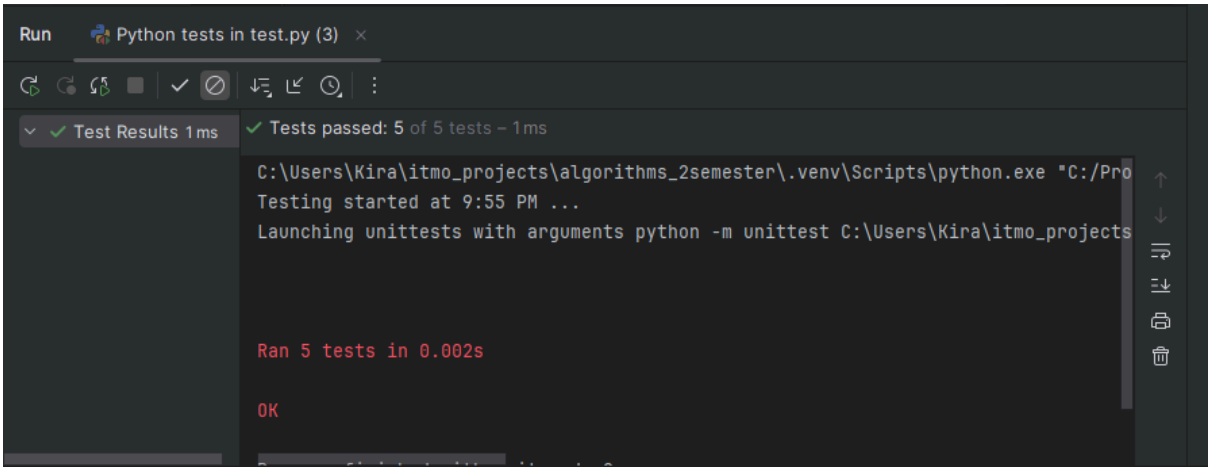
Если вершина end не была обновлена (остаётся inf), значит, пути не существует, возвращается -1. Иначе возвращается dist[end] минимальная стоимость пути.

Тесты:

```
import unittest
from lab3.task8.src.main import dijkstra


class TestDijkstra(unittest.TestCase):

    def test_shortest_path_exists(self):
        # Граф с 4 вершинами и рёбрами
        adj = [
            [(1, 1)],  # 1 - 2 (стоимость 1)
```

```
            [(2, 2)],  # 2 - 3 (стоимость 2)
            [(0, 2)],  # 3 - 1 (стоимость 2)
            []  # 4 не имеет исходящих рёбер
        ]
        n = 4
        start = 0  # Вершина 1 (с индексом 0)
        end = 2  # Вершина 3 (с индексом 2)
        result = dijkstra(n, adj, start, end)
        self.assertEqual(result, 3)  # Ожидаемый минимальный путь: 1 - 2 - 3,
стоимость 3

    def test_no_path(self):
        # Граф с 4 вершинами, но без пути между вершинами 1 и 4
        adj = [
            [(1, 1)],  # 1 - 2
            [(2, 2)],  # 2 - 3
            [],  # 3 не имеет исходящих рёбер
            [(0, 2)]  # 4 - 1
        ]
        n = 4
        start = 0  # Вершина 1 (с индексом 0)
        end = 3  # Вершина 4 (с индексом 3)
        result = dijkstra(n, adj, start, end)
        self.assertEqual(result, -1)  # Путь отсутствует

    def test_multiple_edges(self):
        # Граф с несколькими рёбрами между вершинами
        adj = [
            [(1, 1), (2, 4)],  # 1 - 2 (стоимость 1), 1 - 3 (стоимость 4)
            [(2, 2)],  # 2 - 3 (стоимость 2)
            [(1, 1)],  # 3 - 2 (стоимость 1)
            []  # 4 не имеет исходящих рёбер
        ]
        n = 4
        start = 0  # Вершина 1 (с индексом 0)
        end = 2  # Вершина 3 (с индексом 2)
        result = dijkstra(n, adj, start, end)
        self.assertEqual(result, 3)  # Ожидаемый минимальный путь: 1 - 2 - 3,
стоимость 3

    def test_single_edge(self):
        # Граф с одним ребром
        adj = [
            [(1, 1)],  # 1 - 2 (стоимость 1)
            []  # 2 не имеет исходящих рёбер
        ]
        n = 2
        start = 0  # Вершина 1 (с индексом 0)
        end = 1  # Вершина 2 (с индексом 1)
        result = dijkstra(n, adj, start, end)
        self.assertEqual(result, 1)  # Путь 1 - 2, стоимость 1

    def test_empty_graph(self):
```

```
        # Граф без рёбер
        adj = [[] for _ in range(5)]  # 5 вершин, но нет рёбер
        n = 5
        start = 0  # Вершина 1 (с индексом 0)
        end = 4  # Вершина 5 (с индексом 4)
        result = dijkstra(n, adj, start, end)
        self.assertEqual(result, -1)  # Путь отсутствует


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



Вывод по задаче:

Алгоритм успешно решает задачу поиска минимальной стоимости пути в графе. Использование кучи позволяет обрабатывать рёбра эффективно даже в больших графах. В тестах проверяются случаи наличия и отсутствия пути, наличие нескольких ребер между вершинами, а также работа алгоритма на маленьких и пустых графах

Алгоритм Дейкстры на приоритетной очереди (heap) имеет сложность O((n + m) log n), где: n – количество вершин, m – количество рёбер

Задача №10. Оптимальный обмен валюты

Текст задачи:

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.

Листинг кода:

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

    # Релаксация рёбер (n - 1 раз)
    for _ in range(n - 1):
        for u, v, weight in edges:
            if dist[u] != INF and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Проверяем, какие вершины находятся в отрицательных циклах
    for _ in range(n):  # Делаем n итераций, чтобы все вершины из
отрицательных циклов получили -inf
        for u, v, weight in edges:
            if dist[u] == -INF or (dist[u] != INF and dist[u] + weight <
dist[v]):
                dist[v] = -INF  # Помечаем вершину как "участник
отрицательного цикла"

    return dist


if __name__ == "__main__":
    # Читаем входные данные
    with open("../txtf/input.txt", "r") as file:
        n, m = map(int, file.readline().split())
        edges = []
        for _ in range(m):
            u, v, weight = map(int, file.readline().split())
            edges.append((u, v, weight))
        s = int(file.readline())  # Начальная вершина

    # Вызываем алгоритм Беллмана-Форда
    dist = bellman_ford(n, m, edges, s)

    # Записываем результат в файл
    with open("../txtf/output.txt", "w") as file:
        for i in range(1, n + 1):
            if dist[i] == float('inf'):
                file.write("*\n")  # Нет пути
```

```
            elif dist[i] == -float('inf'):
                file.write("-\n")  # Отрицательный цикл
            else:
                file.write(f"{dist[i]}\n")  # Кратчайшее расстояние
```

Текстовое объяснение задачи:

Задача заключается в нахождении кратчайших путей из заданной вершины s во все остальные вершины ориентированного графа, где рёбра могут иметь отрицательные веса. Для решения используется алгоритм Беллмана-Форда, который позволяет работать с графами, содержащими рёбра с отрицательными весами, и выявлять отрицательные циклы.

Алгоритм работает следующим образом:

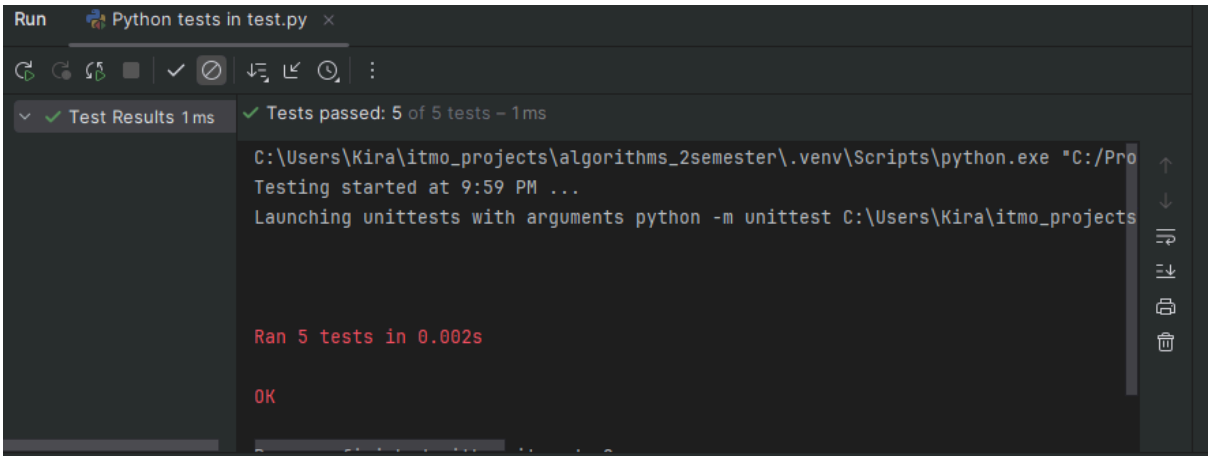
1. Инициализация:
    - Создается массив dist, где dist[i] — минимальное расстояние от s до вершины i. Изначально все расстояния устанавливаются в бесконечность, кроме стартовой вершины s, для которой dist[s] = 0
2. Основной цикл:
    - В течение (n - 1) итераций для каждого ребра (u, v, weight) проверяется, можно ли улучшить расстояние до вершины v через u. Если dist[u] + weight < dist[v], то обновляется dist[v].
3. Проверка на отрицательные циклы:
    - После (n - 1) итераций выполняется ещё n проверок, чтобы пометить все вершины, находящиеся в отрицательных циклах, как минус бесконечность
4. Вывод результатов:
    - Если dist[v] = бесконечность, значит, вершина недостижима, выводится "*"
    - Если dist[v] = минус бесконечность, вершина участвует в отрицательном цикле, выводится "-"
    - Иначе выводится dist[v] — минимальная стоимость пути

Тесты:

```
import unittest
from lab3.task10.src.main import bellman_ford
```

```
class TestBellmanFord(unittest.TestCase):
    def test_simple_case(self):
        """Тест для графа без отрицательных рёбер"""
        n, m = 3, 3
        edges = [
            (1, 2, 4),
            (2, 3, -6),
            (1, 3, 2)
        ]
        s = 1
        expected_output = [float('inf'), 0, 4, -2]  # dist[0] игнорируется
        self.assertEqual(bellman_ford(n, m, edges, s), expected_output)

    def test_no_path(self):
        """Тест, где нет пути до некоторых вершин"""
        n, m = 4, 2
        edges = [
            (1, 2, 3),
            (2, 3, 5)
        ]
        s = 1
        expected_output = [float('inf'), 0, 3, 8, float('inf')]
        self.assertEqual(bellman_ford(n, m, edges, s), expected_output)

    def test_negative_cycle(self):
        """Тест с отрицательным циклом"""
        n, m = 4, 4
        edges = [
            (1, 2, 1),
            (2, 3, 1),
            (3, 4, -3),
            (4, 2, 1)
        ]
        s = 1
        expected_output = [float('inf'), 0, -float('inf'), -float('inf'),
-float('inf')]
        self.assertEqual(bellman_ford(n, m, edges, s), expected_output)

    def test_disconnected_graph(self):
        """Тест с вершиной, которая не соединена с другими"""
        n, m = 5, 3
        edges = [
            (1, 2, 10),
            (2, 3, 5),
            (3, 4, -2)
        ]
        s = 1
        expected_output = [float('inf'), 0, 10, 15, 13, float('inf')]
        self.assertEqual(bellman_ford(n, m, edges, s), expected_output)

    def test_single_node(self):
        """Тест для графа с одной вершиной"""
        n, m = 1, 0
```

```
        edges = []
        s = 1
        expected_output = [float('inf'), 0]  # Только одна вершина, расстояние
до неё 0
        self.assertEqual(bellman_ford(n, m, edges, s), expected_output)

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



Вывод по задаче:

Алгоритм Беллмана-Форда позволяет эффективно находить кратчайшие пути в графе с отрицательными рёбрами и выявлять отрицательные циклы.

Временная сложность. Основной цикл выполняется O(n * m) операций, дополнительные цикл проверки отрицательных циклов также занимает O(n * m), и общая сложность следовательно O(n * m)

Однако алгоритм Беллмана-Форда медленнее, чем алгоритм Дейкстры и подходит только тогда, когда граф содержит отрицательные рёбра или необходимо обнаружить отрицательные циклы. Если рёбра только положительные, Дейкстра будет более оптимальным вариантом

Задача №11. Алхимия

Текст задачи:

Алхимики средневековья владели знаниями о превращении различных химических веществ друг в друга. Это подтверждают и недавние исследования археологов.
В ходе археологических раскопок было обнаружено m глиняных табличек, каждая из которых была покрыта непонятными на первый взгляд символами. В результате расшифровки выяснилось, что каждая из табличек описывает одну алхимическую реакцию, которую умели проводить алхимики.
Результатом алхимической реакции является превращение одного вещества в другое. Задан набор алхимических реакций, описанных на найденных глиняных табличках, исходное вещество и требуемое вещество. Необходимо выяснить: возможно ли преобразовать исходное вещество в требуемое с помощью этого набора реакций, а в случае положительного ответа на этот вопрос – найти минимальное количество реакций, необходимое для осуществления такого преобразования.

Листинг кода:

```
from collections import deque, defaultdict


def alchemy_reactions(m, reactions, start, end):
    # Если начальное вещество равно требуемому, минимальное количество реакций равно 0
    if start == end:
        return 0

    # BFS для поиска кратчайшего пути
    queue = deque([(start, 0)])  # В очередь кладем начальное вещество и счетчик шагов
    visited = set([start])  # Множество посещенных веществ

    while queue:
        current, steps = queue.popleft()

        # Перебираем все возможные реакции для текущего вещества
        for next_substance in reactions[current]:
            if next_substance == end:
                return steps + 1
            if next_substance not in visited:
                visited.add(next_substance)
                queue.append((next_substance, steps + 1))

    # Если путь не найден, возвращаем -1
    return -1


if __name__ == "__main__":
    # Чтение данных из файла
    with open('../txtf/input.txt', 'r') as f:
        m = int(f.readline().strip())  # Количество реакций
```

```
        reactions = defaultdict(list)  # Словарь для графа
        for _ in range(m):
            reaction = f.readline().strip().split(' -> ')
            reactions[reaction[0]].append(reaction[1])

        start = f.readline().strip()
        end = f.readline().strip()

    result = alchemy_reactions(m, reactions, start, end)

    with open('../txtf/output.txt', 'w') as f:
        f.write(f"{result}\n")
```

Текстовое объяснение задачи:

Данная задача моделируется в виде ориентированного графа, где вершины – это химические вещества, а рёбра алхимические реакции

Решение основано на поиске кратчайшего пути в невзвешенном графе с помощью алгоритма BFS (поиск в ширину). BFS идеально подходит, так как он гарантированно находит кратчайший путь в графе с равными весами рёбер (в данном случае вес каждого ребра = 1, так как каждая реакция совершается за один шаг)

Читаем входные данные:

- Считываем количество алхимических реакций m
- Создаём ориентированный граф в виде словаря (defaultdict), где ключ – исходное вещество, а значения список веществ, в которые оно может превращаться
- Считываем начальное вещество start и целевое end

Особый случай:

- Если начальное вещество совпадает с целевым, сразу выводим 0, так как преобразования не требуются

Запускаем BFS для поиска минимального количества реакций:

- Используем очередь (deque), в которой храним пары (вещество, количество шагов)
- Создаём множество посещенных веществ (visited), чтобы избежать зацикливания
- Пока очередь не пуста:

- Берем первый элемент из очереди
- Проверяем, можно ли из него получить целевое вещество
- Если да — возвращаем количество шагов (steps + 1)
- Если нет, добавляем непосещенные вещества в очередь с увеличением счетчика шагов

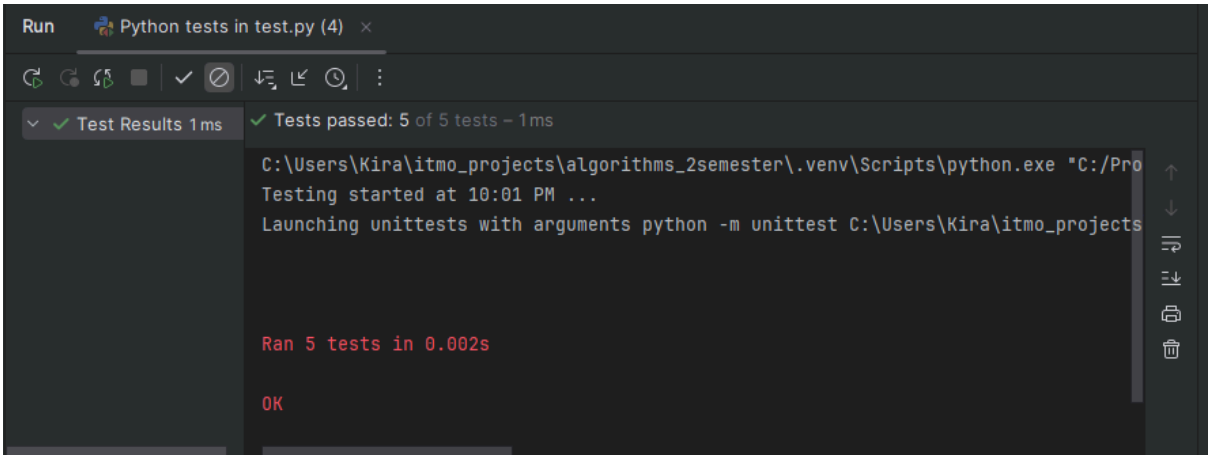
Вывод результатта:

- Если end не удалось достичь, выводим -1

Тесты:

```
import unittest
from lab3.task11.src.main import alchemy_reactions
from collections import deque, defaultdict


class TestAlchemyReactions(unittest.TestCase):

    def setUp(self):
        # Устанавливаем общий тестовый набор реакций
        self.reactions = defaultdict(list)
        self.reactions["Aqua"].append("AquaVita")
        self.reactions["AquaVita"].append("PhilosopherStone")
        self.reactions["AquaVita"].append("Argentum")
        self.reactions["Argentum"].append("Aurum")
        self.reactions["AquaVita"].append("Aurum")

    def test_basic_conversion(self):
        # Проверяем простой случай, где путь существует и минимальное
количество реакций
        result = alchemy_reactions(5, self.reactions, "Aqua", "Aurum")
        self.assertEqual(result, 2)  # Ожидаем минимальное количество шагов 2

    def test_direct_conversion(self):
        # Проверка случая, когда можно пройти только одну реакцию
        self.reactions = defaultdict(list)
        self.reactions["Aqua"].append("AquaVita")
        result = alchemy_reactions(1, self.reactions, "Aqua", "AquaVita")
        self.assertEqual(result, 1)  # Ожидаем 1 шаг

    def test_no_conversion_possible(self):
        # Проверяем случай, когда путь невозможен
        self.reactions = defaultdict(list)
        self.reactions["Aqua"].append("AquaVita")
        result = alchemy_reactions(1, self.reactions, "Aqua",
"PhilosopherStone")
        self.assertEqual(result, -1)  # Путь не существует, ожидаем -1
```

```
    def test_no_reactions(self):
        # Проверяем случай без реакций
        self.reactions = defaultdict(list)
        result = alchemy_reactions(0, self.reactions, "Aqua", "Aurum")
        self.assertEqual(result, -1)  # Путь невозможен, так как нет реакций

    def test_same_substance(self):
        # Проверка случая, когда начальное вещество равно требуемому
        result = alchemy_reactions(5, self.reactions, "Aqua", "Aqua")
        self.assertEqual(result, 0)  # Путь не требуется, так как вещества
одинаковые


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



Вывод по задаче:

Решение позволяет определить, можно ли преобразовать одно вещество в другое с помощью заданных алхимических реакций, и найти минимальное количество шагов для этого. Если превращение возможно, алгоритм возвращает кратчайший путь, иначе -1. Используемый алгоритм поиска в ширину (BFS) гарантирует нахождение оптимального решения. Временная сложность составляет O(m), где m — количество реакций, что делает алгоритм эффективным даже при большом количестве входных данных

Вывод

В ходе выполнения лабораторной работы были рассмотрены и реализованы алгоритмы для решения различных задач, связанных с графами. Включенные задачи затрагивали как поиск кратчайших путей, так и поиск достижимости вершин, что позволило изучить и применить классические алгоритмы на графах.

Для вычисления кратчайших путей в графе с отрицательными рёбрами использовался алгоритм Беллмана-Форда, обеспечивающий нахождение минимальных расстояний и выявление отрицательных циклов за O(n*m). В задачах с положительными рёбрами и поиском минимального количества шагов был использован поиск в ширину (BFS), который работает за O(m) и гарантирует нахождение кратчайшего пути в невзвешенном графе.

Реализация алгоритмов позволила проанализировать различные аспекты работы с графами, такие как ориентированные и неориентированные связи, взвешенные и невзвешенные рёбра, наличие циклов и поиск кратчайших путей. Это подтвердило важность выбора правильного алгоритма в зависимости от условий задачи и структуры графа