МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСИС»

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК (ИКН)

Курсовая работа на тему «Алгоритм Левита, построение кратчайших расстояний»

Выполнила: студентка группы БИВТ-23-4

Галеницкая Валерия Александровна

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритм Левита — это метод поиска кратчайших путей от одной вершины до всех остальных в графе с взвешенными рёбрами, включая рёбра с отрицательным весом (без циклов). Он использует три множества вершин, чтобы эффективно обновлять расстояния, перемещая вершины между группами в зависимости от их статуса.

Алгоритм применяется для оптимизации маршрутов, моделирования сетей и других задач, связанных с анализом графов.

Алгоритм Левита предназначен для решения задачи поиска кратчайших путей от одной заданной вершины до всех остальных вершин в ориентированном или неориентированном графе с неотрицательными и отрицательными весами рёбер (при условии отсутствия отрицательных циклов).

Задача может звучать так:

Дан взвешенный граф $G= (V, E)$, в котором нет циклов отрицательного веса, и стартовая вершина $s \in V$. Требуется найти кратчайшие расстояния от s до всех остальных вершин графа.

Псевдокод:

- M_0 — хеш-таблица,
- M_1 — основная и срочная очереди,
- M_2 — хеш-таблица.

```
for u : u ∈ V
    d[u] = ∞
d[s] = 0
M1'.push(s)
for u : u ≠ s and u ∈ V
    M2.add(u)
while M1' ≠ ∅ and M1'' ≠ ∅
    u = (M1'' = ∅ ? M1'.pop() : M1''.pop())
    for v : uv ∈ E
        if v ∈ M2
            M1'.push(v)
            M2.remove(v)
            d[v] = min(d[v], d[u] + w_uv)
        else if v ∈ M1
            d[v] = min(d[v], d[u] + w_uv)
        else if v ∈ M0 and d[v] > d[u] + w_uv
            M1''.push(v)
            M0.remove(v)
            d[v] = d[u] + w_uv
    M0.add(u)
```

Рисунок 1 - Псевдокод

ОСНОВНАЯ ЧАСТЬ

Теория графов

В теории графов выделяют 3 основных термина (на примере рис.2.1):

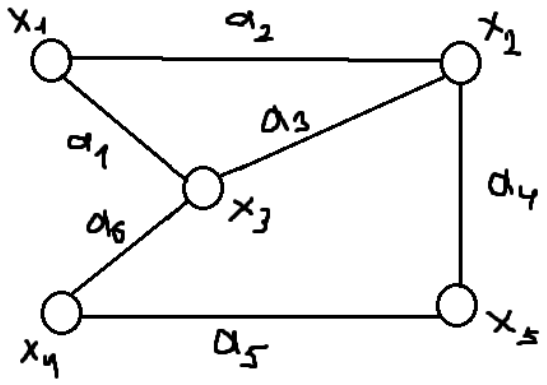

Рис.2.1 - Граф G(X,A)

Граф - это математическая структура, которая используется для описания множества объектов и связей между ними. Граф определяется как G=(X,A), где:

- X - множество объектов, называемых вершинами.
- A - множество связей между вершинами, называемых рёбрами.

Граф применяется для моделирования различных систем, например, дорог между городами, социальных сетей, компьютерных сетей и т.д.

Вершина (узел) графа - это элемент множества X, представляющий объект. Они могут быть обозначены числами, буквами или другими символами.

Ребро - это связь между двумя вершинами. Оно определяется как пара вершин:

- В ориентированном графе связь имеет направление и представляется упорядоченной парой (xi,xj), где xi→xj.
- В неориентированном графе направление не задаётся, и связь представляется как {xi,xj}.

Ребро может иметь вес — численное значение, которое может обозначать, например, расстояние, стоимость или время.

Если ребра из множества A ориентированы, что обычно показывается стрелкой, то они называются дугами, и граф с такими ребрами называется ориентированным графом. Если ребра не имеют ориентации, то граф называется неориентированным. Алгоритм Левита может применяться как к ориентированным, так и к неориентированным графам.

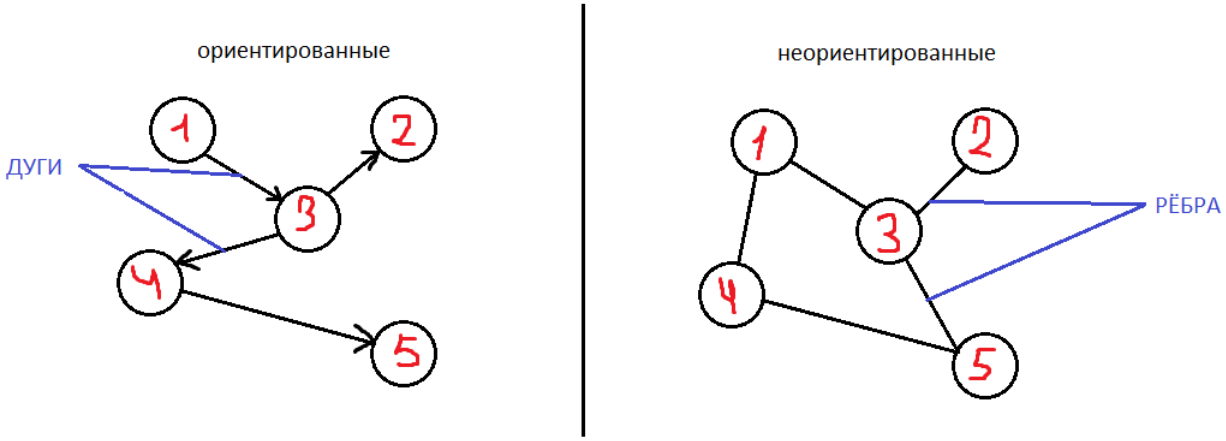

Рис.2.2 – Виды графов

Для понимания алгоритма Левита его применения полезно знать следующие термины из теории графов:

1. Вес ребра

Числовое значение, ассоциированное с ребром. Может быть положительным или отрицательным (если нет отрицательных циклов).

2. Кратчайший путь

Последовательность рёбер, соединяющая две вершины, с минимальной суммой весов. Цель алгоритма — найти такие пути.

3. Релаксация

Процесс улучшения оценки кратчайшего расстояния до вершины, если найден путь меньшей стоимости через другую вершину. Это ключевая операция в алгоритме.

4. Множества вершин в алгоритме Левита

- Необработанные вершины: вершины, до которых ещё не найдены кратчайшие пути.

- Текущие вершины: вершины, которые сейчас обрабатываются.
- Обработанные вершины: вершины, для которых уже определены кратчайшие пути.

5. Отрицательный цикл

Цикл, сумма весов рёбер которого отрицательна. Если граф содержит такой цикл, алгоритм сигнализирует, что корректный кратчайший путь невозможно определить.

6. Очередь

Структура данных, используемая в алгоритме для обработки вершин. В Левите применяются две структуры:

- Обычная очередь для вершин текущего уровня обработки.
- Дек (двусторонняя очередь) для оптимизации обработки соседних вершин.

7. Предшественник вершины

Вершина, из которой был достигнут кратчайший путь к текущей вершине.

Способы задания графа

Далее будут представлены способы представления графов, на примере графа G :

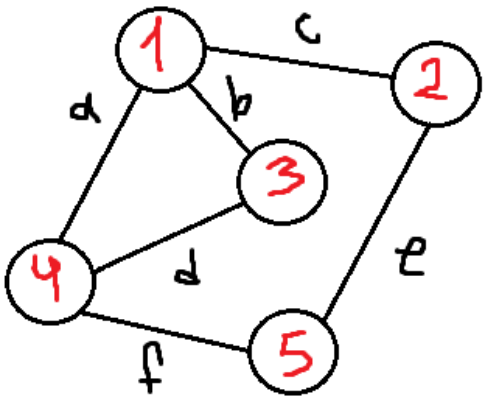

Рис.3 – Граф G

Матрица инцидентности.

Описание: матрица инцидентности — это прямоугольная матрица, где строки представляют вершины, а столбцы — рёбра. Если вершина инцидентна ребру (то есть, соединяется этим ребром с другой вершиной), то в соответствующей ячейке ставится 1, иначе — 0.

Как задавать:

- Перечисляем вершины и рёбра графа.
- Создаём матрицу, в которой для каждой вершины и каждого ребра указываем их инцидентность (1, если инцидентны, иначе 0).

Пример:

Таблица 1.1 - Матрица инцидентности для графа G

		рёбра					
		a	b	c	d	e	f
вершины	1	1	1	1	0	0	0
	2	0	0	1	0	1	0
	3	0	1	0	1	0	0
	4	1	0	0	1	0	1
	5	0	0	0	0	1	1

Матрица смежности.

Описание: матрица смежности — это квадратная матрица A размером V × V, где V - количество вершин графа. Если между вершинами i и j есть ребро, то A[i][j] = 1; если нет A[i][j] = 0.

Как задавать:

- Нумеруем вершины от 1 до V.
- Создаём двумерную матрицу и для каждой пары вершин проверяем наличие рёбра.
- Если ребро есть, записываем 1 (или вес ребра для взвешенных графов), если нет - 0.

Пример:

Таблица 1.2 - Матрица смежности для графа G

		вершины				
		1	2	3	4	5
вершины	1	0	1	1	1	0
	2	1	0	0	0	1
	3	1	0	0	1	0
	4	1	0	1	0	1
	5	0	1	0	1	0

На главной диагонали матрицы смежности обыкновенного графа всегда стоят нули (нет петель) и эта матрица симметрична относительно главной диагонали (граф неориентированный).

Список рёбер.

Описание: представляет граф как набор всех его рёбер. Каждый элемент списка — это пара (или тройка, если граф взвешенный), где указаны соединяемые вершины (и вес ребра).

Как задавать:

- Для каждого ребра указываем пары соединённых вершин.
- Если граф взвешенный, добавляем вес для каждой пары.

Пример:

Таблица 1.3 - Список рёбер для графа G

	начало	конец	вес
1	1	2	c
2	1	3	b
3	1	4	a
4	2	5	e
5	3	4	d
6	4	5	f

Список смежности.

Описание: представление графа в виде списка, где каждой вершине соответствует список всех её смежных вершин (то есть вершин, с которыми она соединена рёбрами).

Как задавать:

- Каждой вершине сопоставляется список (массив) всех вершин, смежных с ней.
- Записываем все смежные вершины для каждой вершины.

Пример:

```
graph = [
 [2, 5, 6], # Вершина 1 соединена с вершинами 2, 5 и 6
 [3, 1], # Вершина 2 соединена с вершиной 3 и 1
 [2, 4, 5], # Вершина 3 соединена с вершинами 2, 4 и 5
 [3], # Вершина 4 соединена с вершиной 3
 [1, 3, 6, 7], # Вершина 5 соединена с вершинами 1, 3, 6 и 7
 [1, 5, 7], # Вершина 6 соединена с вершинами 1, 5 и 7
 [5, 6], # Вершина 7 соединена с вершинами 5 и 6
]
```

В этом способе пример представлен на основе этого графа.

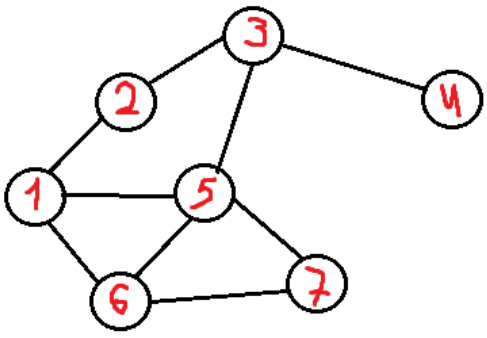

Рис.4 - Граф G1

Для реализации алгоритма Левита граф обычно задаётся в виде списка смежности, так как этот способ наиболее эффективен для работы с разреженными графами, где число рёбер значительно меньше квадрата числа вершин (E≪V^2). Поэтому в дальнейшем алгоритмы будут работать с именно этим способом.

Сравнительный анализ

Алгоритм Левита решает задачу поиска кратчайших путей от одной вершины до всех остальных в графе, аналогично таким алгоритмам, как Дейкстра и Беллман-Форд. Однако каждый из них имеет свои особенности, которые определяют их преимущества и ограничения в разных условиях.

Таблица 2.1 - Сравнение 1

Критерий	Левит	Дейкстра	Беллман-Форд
Графы	Ориентированные/неориентированные; поддерживает отрицательные веса (без циклов).	Ориентированные/неориентированные; только положительн ые веса.	Ориентированные/неориентированные; поддерживает отрицательные веса (включая проверку циклов).
Сложность	O(V+E) для разреженных графов (с использованием дека).	O((V+E)·logV) с приоритетной очередью.	O(V·E)
Эффективность	Быстрее Беллмана-Форда для разреженных графов; конкурирует с Дейкстрой на плотных графах.	Быстрее для графов с положительными весами; оптимален для разреженных графов.	Медленнее в среднем; полезен для графов с отрицательными рёбрами.
Поддержка отрицательных весов	Да (но не поддерживает отрицательные циклы).	Нет	Да (и определяет наличие отрицательных циклов).

Структура данных	Список смежности + дек.	Список смежности + очередь с приоритетом.	Список рёбер или смежности.
Применение	Оптимизация маршрутов в сетях, где могут быть отрицательные веса.	Навигационные системы, задачи маршрутизации (без отрицательных весов).	Анализ финансовых моделей, транспортные задачи с ограничениями.

Таблица 2.2 - Сравнение 2

Алгоритм	Преимущества	Недостатки
Левит	Универсальность; высокая эффективность на разреженных графах; поддержка отрицательных весов.	Сложнее реализовать, чем Дейкстру; не определяет отрицательные циклы.
Дейкстра	Высокая скорость для графов с положительными весами; широко используется в практике.	Не работает с отрицательными весами.
Беллман-Форд	Простая реализация; работает с отрицательными весами; обнаруживает отрицательные циклы.	Низкая скорость работы на больших графах из-за O(V·E).

Алгоритм Левита

Алгоритм Левита — это способ найти кратчайший путь в графе от одной выбранной вершины до всех остальных. Представьте, что вы планируете маршруты доставки, где дороги имеют разную стоимость проезда, включая возможность скидок (отрицательные веса).

Алгоритм Левита позволяет эффективно определить самый короткий путь, даже если граф содержит как положительные, так и отрицательные веса рёбер (но без циклов с отрицательной длиной).

Метод работает, группируя вершины в три множества: необработанные, текущие и обработанные. Он постепенно изучает текущие вершины и перемещает их в другие группы, обновляя минимальные расстояния до соседей. Это похоже на процесс, где вы уточняете стоимость проезда, изучая все доступные маршруты, и улучшаете данные по мере обнаружения более выгодных вариантов.

Особенность алгоритма Левита — использование очереди (или дека), которая позволяет эффективно управлять вершинами и ускоряет обработку графов, особенно если они разреженные.

Ключевые моменты алгоритма:

1. Поддержка отрицательных весов:

В отличие от Дейкстры, алгоритм Левита может обрабатывать графы с отрицательными весами рёбер, если нет отрицательных циклов.

2. Оптимизация обработки графов:

Использование очереди позволяет избежать лишних проверок и улучшить производительность на разреженных графах.

3. Три множества вершин:

- Необработанные: те, до которых ещё не определены кратчайшие пути.
- Текущие: вершины, которые активно обрабатываются.
- Обработанные: вершины, для которых уже найдены оптимальные пути.

Шаги реализации алгоритма Левита:

1. Инициализация**:**

- Задайте начальную вершину S. Установите её расстояние равным 0, а для всех остальных — бесконечность.
- Добавьте начальную вершину в очередь.

2. Работа с очередями:

Пока очередь не пуста:

- Извлеките вершину U из очереди.

- Для каждого соседа V вершины U:
  - Рассчитайте новое потенциальное расстояние до V через U.
  - Если расстояние до V улучшилось, обновите его:
  - Если V находится в необработанном множестве, добавьте её в конец очереди.
  - Если V уже в текущем множестве, ничего не меняйте.
  - Если V была обработана, добавьте её в начало очереди.

3. Завершение**:**

- Когда очередь опустеет, кратчайшие расстояния до всех достижимых вершин будут определены.
- Если требуется, восстановите пути с помощью массива предшественников.

Анализ алгоритма

Корректность алгоритма:

Алгоритм Левита работает корректно для ориентированных и неориентированных графов, в которых отсутствуют отрицательные циклы. Он основан на принципе релаксации рёбер и организует обработку вершин с помощью трёх множеств:

- **M0** — множество необработанных вершин (ещё не достигнуты);
- **M1** — множество текущих (активных) вершин, находящихся в очереди;
- **M2** — множество уже обработанных вершин (их кратчайшее расстояние определено).

На каждом шаге из M1 извлекается вершина u, и для каждого соседа v пересчитывается возможное минимальное расстояние d[v] через u. Если найден более короткий путь, расстояние обновляется, и вершина перемещается в нужное множество (при необходимости — снова в очередь).

Доказательство корректности строится на индукции по числу обработанных вершин:

1. Изначально только начальная вершина S имеет расстояние d[S]=0, все остальные — ∞.
2. Пусть для первых k обработанных вершин кратчайшие расстояния найдены корректно.
3. При извлечении k+1-й вершины из очереди, она выбирается с минимальным текущим значением d[v], которое уже не может быть улучшено через другие вершины (аналогично алгоритму Дейкстры).
4. Таким образом, для каждой вершины, добавляемой в M2, расстояние до неё действительно кратчайшее.

Отсюда следует, что по завершении работы алгоритма значения в массиве расстояний соответствуют кратчайшим путям от стартовой вершины S.

Оценка временной сложности:

Обозначим:

- V — количество вершин,
- E — количество рёбер.

Алгоритм использует двустороннюю очередь (дек) для управления активными вершинами. Каждая вершина:

- добавляется в очередь не более двух раз (первоначально и при возможном улучшении пути);
- каждое ребро проверяется при обработке смежной вершины.

Основные операции:

- Инициализация расстояний и очереди: O(V),
- Обработка всех рёбер: O(E),
- Перемещения между множествами и в очереди: O(V).

Общая временная сложность алгоритма:

O(V+E), что делает его особенно эффективным на разреженных графах.

Оценка используемой памяти:

Алгоритм использует следующие структуры данных:

- Массив расстояний d[v]: O(V),
- Массив предшественников p[v]: O(V),
- Три множества вершин (M0, M1, M2): O(V),
- Очередь (дек): O(V),
- Список смежности для хранения графа: O(V+E).

Итоговая пространственная сложность: O(V+E)

Реализация

Рассмотрим реализацию алгоритма данного графа:

Пример реализации вручную с помощью библиотек networkx + matplotlib, то есть с визуализацией (1):

Изначально все вершины имеют расстояние ∞, кроме начальной вершины S, для которой расстояние равно 0. Массив предшественников для всех вершин также инициализируется значением null. Начальная вершина S добавляется в очередь.

Таблица 3.1 - Начальные данные

Вершина	Расстояние	Предшественник
S	0	-
A	∞	-
B	∞	-
C	∞	-

Очередь: [S].

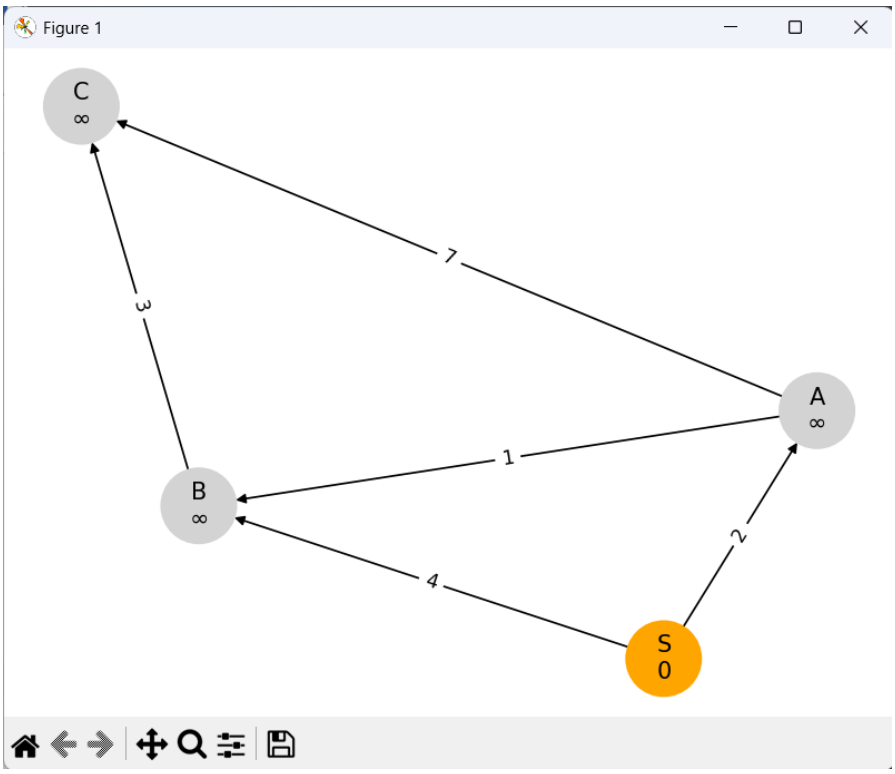

Рис. 5.1 – Обработка вершины S

Шаг 1: Обработка вершины S

- Извлекаем S из очереди.
- Обновляем расстояния до соседей A и B:
  - d(A)= 0+2=2, предшественник A=S.
  - d(B)= 0+4=4, предшественник B=S .
- Добавляем вершины A и B в очередь.

Таблица 3.2 - Обработка вершины S

Вершина	Расстояние	Предшественник
S	0	-
A	2	S
B	4	S
C	∞	-

Очередь: [A,B].

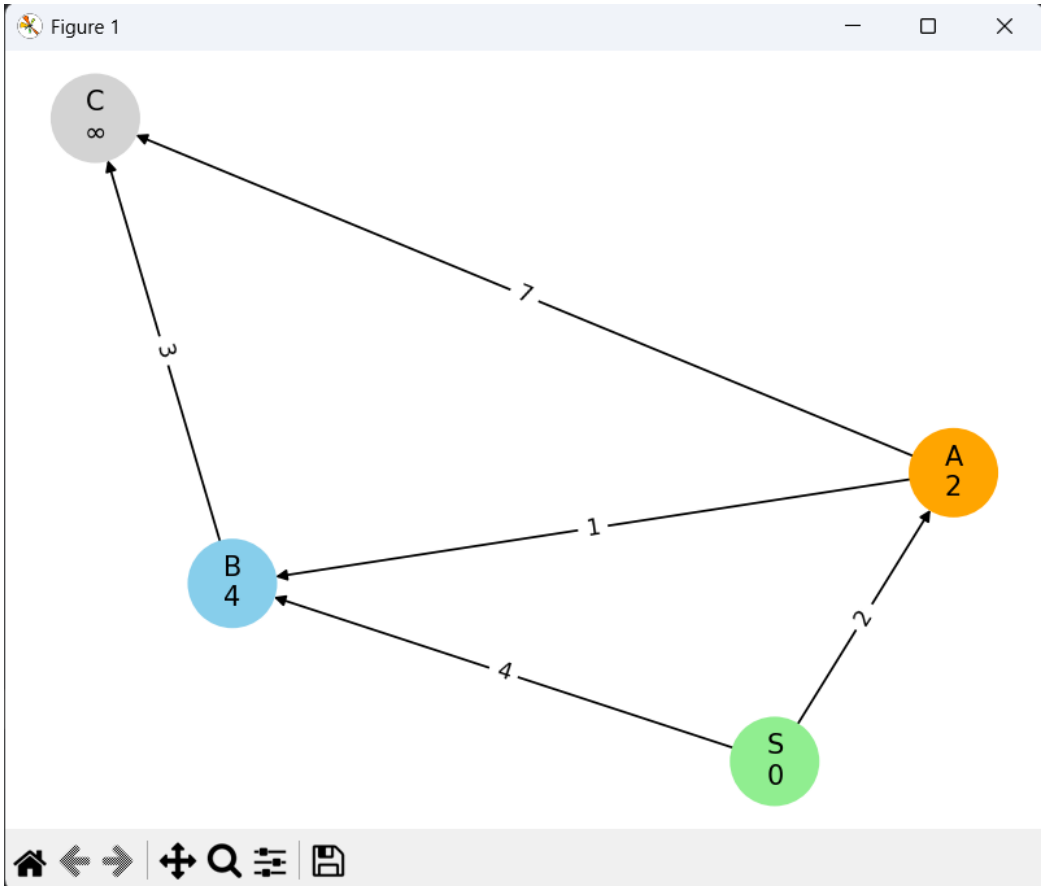

Рис. 5.2 – Обработка вершины A

Шаг 2: Обработка вершины A

- Извлекаем A из очереди.
- Рассматриваем соседей B и C:
  - B: d(B)= 2+1=3.
  - C: d(C)= 2+7=9, обновляем расстояние, предшественник C=A.
- Добавляем вершину C в очередь.

Таблица 3.3 - Обработка вершины A

Вершина	Расстояние	Предшественник
S	0	-
A	2	S
B	3	S
C	9	A

Очередь: [B,C].

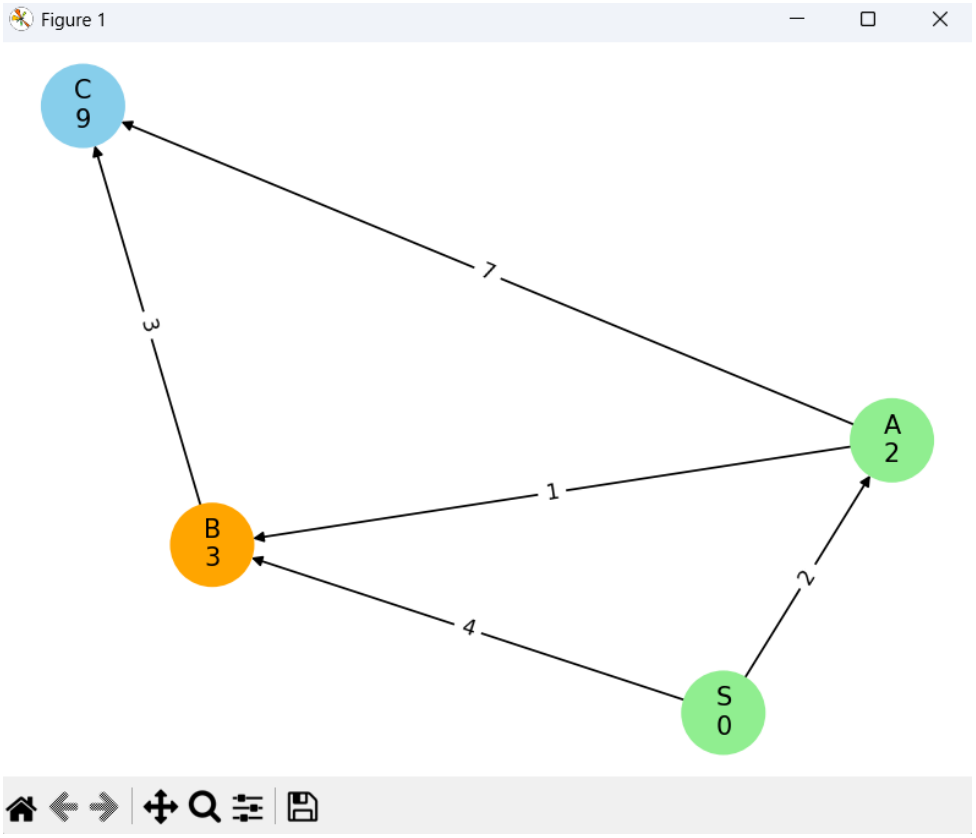

Рис. 5.3 – Обработка вершины B

Шаг 3: Обработка вершины B

- Извлекаем B из очереди.
- Рассматриваем соседа C:
  - C: d(C)=3+3=6, обновляем расстояние, предшественник C=B.

Таблица 3.4 - Обработка вершины B

Вершина	Расстояние	Предшественник
S	0	-
A	2	S
B	3	S
C	6	B

Очередь: [C].

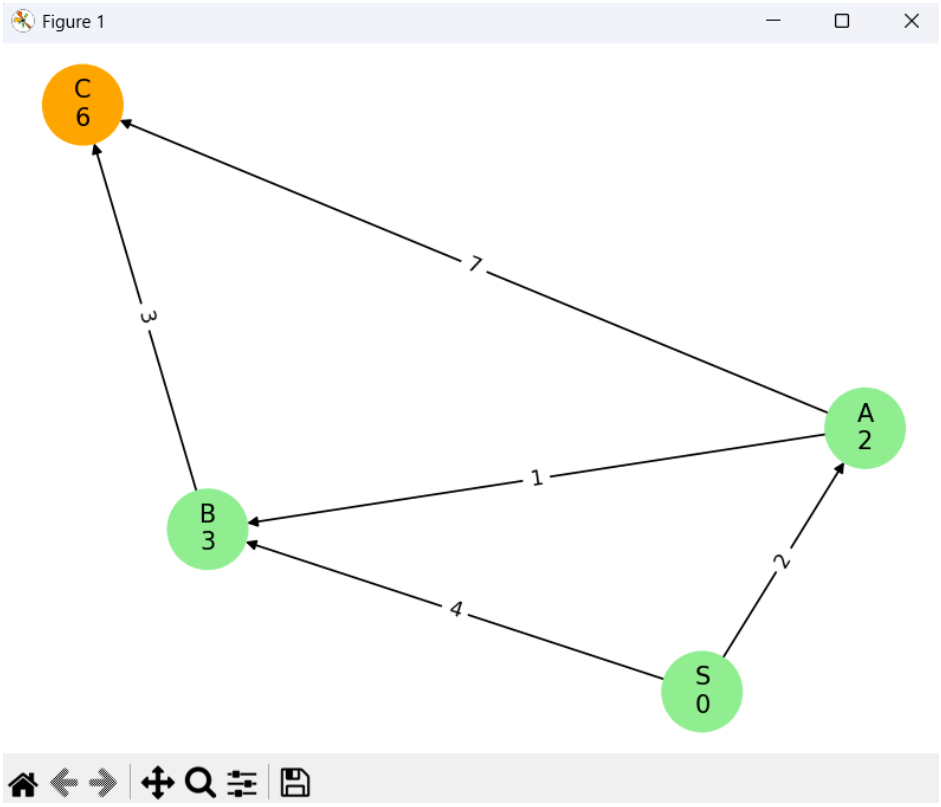

Рис. 5.4 – Обработка вершины C

Шаг 4: Обработка вершины C

- Извлекаем C из очереди.
- У вершины C нет соседей, поэтому ничего не меняем.
- Очередь пуста, алгоритм завершается.

Таблица 3.5 - Обработка вершины C

Вершина	Расстояние	Предшественник
S	0	-
A	2	S
B	3	S
C	6	B

Очередь: [].

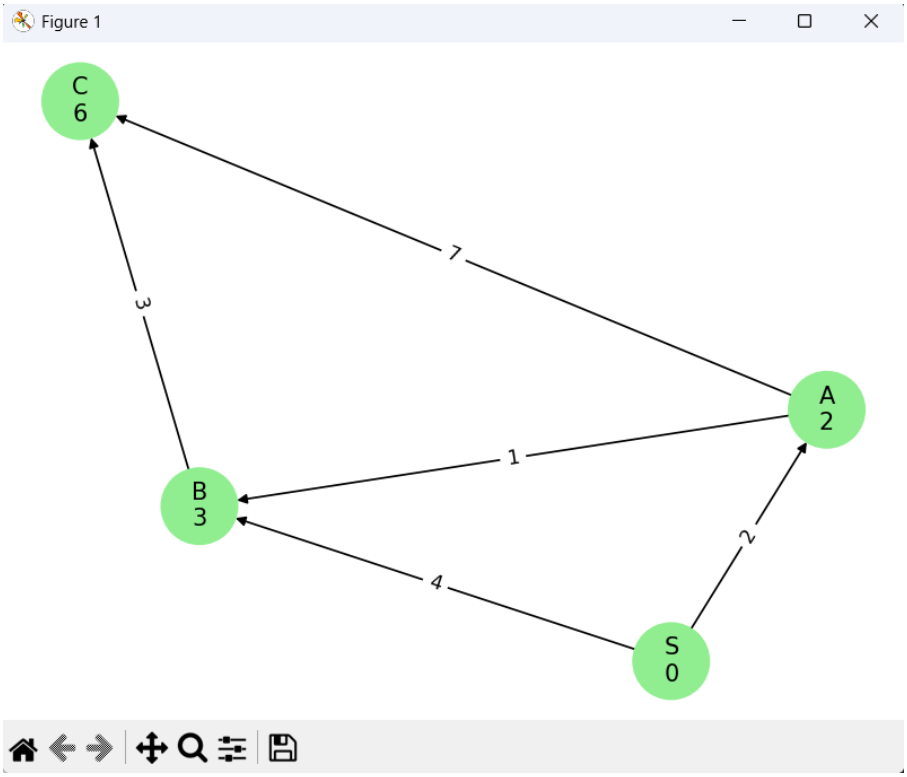

Рис. 5.5 – Итог

Пример реализации на Python с набором тестовых данных (2):

Напишем реализацию на питоне на основе предыдущего примера:

```
from collections import deque

def levit_shortest_path(graph, start, vertices):
    INF = float('inf')
    dist = [INF] * vertices
    pred = [-1] * vertices

    dist[start] = 0
    dq = deque()
    dq.append(start)

    while dq:
        u = dq.popleft()

        for v, weight in graph.get(u, []):
            if dist[v] > dist[u] + weight:
                dist[v] = dist[u] + weight
                pred[v] = u
```

Рис. 6.1 – Код 1

```
            if v not in dq:
                if dq and dist[v] < dist[dq[0]]:
                    dq.appendleft(v)
                else:
                    dq.append(v)

    vertex_names = ["S", "A", "B", "C"]

    print("Вершина | Расстояние | Предшественник")
    for i in range(vertices):
        pred_name = "null" if pred[i] == -1 else vertex_names[pred[i]]
        print(f"{vertex_names[i]:^7} | {dist[i]:^10} | {pred_name}")
```

Рис. 6.2 – Код 2

```
if __name__ == "__main__":
    graph = {
        0: [(1, 2), (2, 4)],   # S
        1: [(2, 1), (3, 7)],   # A
        2: [(3, 3)],           # B
        3: []                  # C
    }

    levit_shortest_path(graph, start=0, vertices=4)
```

Рис. 6.3 – Код 3

Программа реализует алгоритм Левита на языке Python. Она принимает на вход граф, заданный в виде словаря смежности, где каждой вершине сопоставлен список кортежей из смежных вершин и соответствующих весов рёбер. Также задаются номер стартовой вершины и общее количество вершин в графе.

На этапе инициализации создаются два массива: dist — для хранения кратчайших расстояний от стартовой вершины до остальных, и pred — для хранения предшественников, то есть вершин, из которых был получен кратчайший путь. Расстояния изначально принимают значение бесконечности, за исключением стартовой вершины, для которой оно равно нулю. Предшественники инициализируются значением -1.

Далее создаётся очередь (deque) для обработки вершин в порядке их приоритетов. Алгоритм последовательно извлекает вершины из очереди и проверяет все их смежные вершины. Если найден путь с меньшим расстоянием, чем ранее зафиксированное, происходит обновление массива расстояний и массива предшественников. После этого вершина возвращается в очередь: либо в начало, если у неё высокий приоритет, либо в конец — если более низкий. Этот механизм ускоряет работу алгоритма по сравнению с классическим методом Беллмана-Форда.

После завершения основного цикла программа выводит таблицу, в которой указана каждая вершина, её кратчайшее расстояние от стартовой, а также вершина-предшественник, по которой был достигнут кратчайший путь.

Результат:

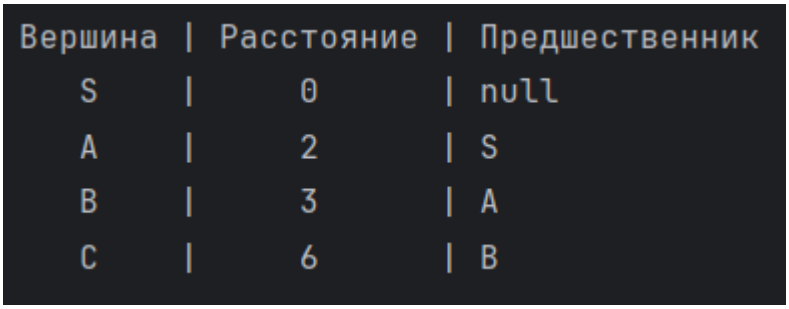

Рис. 7 – Результат

Эти данные полностью совпадают с тем, что можно получить при ручном анализе графа (как раз для этого и рассматривали, чтобы убедиться). Прямой путь от S до B весит 4, но путь через A даёт сумму 2 + 1 = 3, что выгоднее. Аналогично, путь от S до C через A составляет 9, но через B — 3 + 3 = 6, что тоже оптимальнее. Алгоритм корректно учитывает такие случаи и выбирает наилучшие маршруты.

ЗАКЛЮЧЕНИЕ

Алгоритм Левита представляет собой эффективное решение для поиска кратчайших путей в графах с положительными весами рёбер, предоставляя оптимальную альтернативу алгоритму Дейкстры в случаях, когда необходимо учитывать очередность обработки вершин для получения наилучшего пути. Благодаря использованию приоритетной очереди и концепции двустороннего поиска, алгоритм Левита демонстрирует высокую эффективность и скорость работы. Он идеально подходит для графов с большим количеством рёбер и вершин, где другие алгоритмы могут оказаться менее эффективными. Левит быстро находит решения для задач с ограниченными временными ресурсами и способен работать в реальном времени. Несмотря на свою сложность в реализации, алгоритм Левита является ценным инструментом в области обработки графов, особенно в приложениях, требующих скоростных и точных решений, таких как навигационные системы, маршрутизация в сетях и оптимизация логистики.

ИСТОЧНИКИ

1. https://руни.рф/Алгоритм_Левита
2. https://nerc.itmo.ru/wiki/index.php?title=Алгоритм_Левита#.D0.9F.D1.81.D0.B5.D0.B2.D0.B4.D0.BE.D0.BA.D0.BE.D0.B4

https://github.com/Kusiksu/Kusa.git - ссылка на репозиторий с докладом и кодом реализации.