

# **Доклад: Алгоритм Левита построения кратчайших расстояний**

**Выполнил обучающийся НИТУ МИСИС**

**Группа БИВТ-23-1**

**Максименков Иван Михайлович**

**Ссылка на реализацию:**

**[GitHub]( [github.com/MakSimenA/kitgThree](https://github.com/MakSimenA/kitgThree))**

## Оглавление

Введение .....	3
Формальная постановка задачи .....	3
Теоретическое описание алгоритма .....	3
Основные идеи и принципы работы алгоритма: .....	3
1. Жадный подход: .....	3
2. Динамическое программирование: .....	3
3. Обработка вершин: .....	4
4. Обновление расстояний: .....	4
5. Завершение алгоритма: .....	4
Описание алгоритма .....	4
1. Инициализация .....	4
2. Основной цикл алгоритма .....	5
Характеристики алгоритма .....	5
1. Временная сложность: .....	5
2. Пространственная сложность: .....	5
3. Применимость: .....	6
4. Особенности работы: .....	6
5. Примеры применения: .....	6
Сравнительный анализ с аналогичными алгоритмами .....	6
Перечень инструментов для реализации .....	7
1. Языки программирования: .....	7
2. Библиотеки: .....	7
3. Инструменты разработки: .....	7
Описание реализации алгоритма на языке Python .....	7
Выполнение кода: .....	9
Тестирование: .....	9
Выполнение тестирования: .....	11
Заключение .....	11

## Введение

Алгоритм Левита — это эффективный метод для нахождения кратчайших расстояний в графах, который был предложен в 1970-х годах. Он сочетает в себе идеи из алгоритмов Дейкстры и Беллмана-Форда, обеспечивая при этом высокую производительность при работе с графами, содержащими только неотрицательные веса. Алгоритм Левита находит применение в различных областях, таких как маршрутизация в компьютерных сетях, планирование и оптимизация. В данном докладе будет рассмотрена формальная постановка задачи, теоретическое описание алгоритма, его характеристики, сравнительный анализ с аналогичными алгоритмами и инструменты для реализации.

## Формальная постановка задачи

Задача нахождения кратчайших расстояний заключается в определении минимальных расстояний от одной вершины графа (источника) до всех остальных вершин. Формально, пусть  $G = (V, E)$  — ориентированный граф, где  $V$  — множество вершин, а  $E$  — множество рёбер с весами  $w(u, v)$  для каждого ребра  $(u, v) \in E$ . Требуется найти кратчайшие пути от источника  $s \in V$  до всех других вершин  $v \in V$ .

## Теоретическое описание алгоритма

Алгоритм Левита, также известный как алгоритм Дейкстры, строит кратчайшие расстояния с использованием жадного подхода и динамического программирования. Он эффективно находит кратчайшие пути в графах с не негативными весами рёбер, что делает его одним из самых популярных алгоритмов для решения задач о кратчайших путях.

### Основные идеи и принципы работы алгоритма:

#### 1. Жадный подход:

Алгоритм выбирает вершину с минимальным известным расстоянием на каждом шаге. Эта жадная стратегия гарантирует, что когда вершина извлекается из очереди для обработки, расстояние до неё является окончательным, так как не существует более короткого пути, который можно было бы найти позже.

#### 2. Динамическое программирование:

Алгоритм использует принцип оптимальности, который гласит, что кратчайший путь к вершине может быть построен из кратчайших путей к её предшественникам. Это

позволяет алгоритму обновлять расстояния до соседних вершин на основе уже известных расстояний.

### 3. Обработка вершин:

Вершины обрабатываются в порядке увеличения их текущего расстояния от начальной вершины. Это достигается с помощью приоритетной очереди, которая обеспечивает эффективный доступ к вершине с минимальным расстоянием.

### 4. Обновление расстояний:

При обработке каждой вершины алгоритм рассматривает все её соседние вершины и обновляет их расстояния, если найден более короткий путь через текущую вершину. Это обновление происходит в соответствии с формулой:

$$d(v) = \min(d(v), d(u) + w(u, v))$$

где  $u$  — текущая обрабатываемая вершина,  $v$  — соседняя вершина, а  $w(u, v)$  — вес ребра между ними.

### 5. Завершение алгоритма:

Алгоритм завершается, когда все вершины были обработаны, и массив расстояний содержит окончательные значения кратчайших расстояний от начальной вершины до всех других вершин графа.

## Описание алгоритма

### 1. Инициализация

Перед началом работы алгоритма необходимо выполнить следующие шаги:

- Определение входных данных: Граф  $G(V, E)$ , состоящий из множества вершин  $V$  и рёбер  $E$ , где каждое ребро  $(u, v)$  имеет ненегативный вес  $w(u, v)$ .

- Инициализация расстояний:

- Установите расстояние до начальной вершины  $s$  равным 0:

$$d(s) = 0$$

Это означает, что начальная вершина находится на нулевом расстоянии от самой себя.

- Для всех остальных вершин  $v \in V$  установите расстояния равными бесконечности:

$$d(v) = \infty$$

Это показывает, что на начальном этапе мы не знаем расстояния до других вершин.

- Создание структуры данных: Создайте приоритетную очередь (или минимальную кучу), чтобы отслеживать вершины, расстояния до которых необходимо обновить. Эта структура данных обеспечит эффективный доступ к вершине с минимальным расстоянием.

## 2. Основной цикл алгоритма

Алгоритм работает в несколько итераций, пока не обработает все вершины графа:

- Добавление начальной вершины в очередь:

$Q.add(s)$

Начальная вершина помещается в очередь для обработки

- Основной цикл:

- Пока очередь  $Q$  не пуста:

- Извлечение вершины: Извлеките вершину  $u$  с минимальным расстоянием из очереди:

$u = Q.pop()$

Эта вершина считается текущей.

- Обработка соседей: Для каждого соседнего узла  $v$ , связанного с вершиной  $u$  через ребро  $(u, v)$ :

- Если выполняется условие:

$$d(u) + w(u, v) < d(v)$$

это означает, что найден более короткий путь к вершине  $v$ .

- Обновление расстояния: Обновите значение расстояния до вершины  $v$ :

$$d(v) = d(u) + w(u, v)$$

- Обновление очереди: Если вершина  $v$  не находится в очереди  $Q$ , добавьте её:

$Q.add(v)$

Это позволяет обеспечить дальнейшую обработку этой вершины.

## Характеристики алгоритма

### 1. Временная сложность:

Временная сложность алгоритма Левита составляет  $O((V + E) \log V)$ , где  $V$  — количество вершин, а  $E$  — количество рёбер в графе. Это связано с использованием приоритетной очереди для обработки вершин.

### 2. Пространственная сложность:

Пространственная сложность алгоритма составляет  $O(V)$ , так как требуется хранить массивы расстояний и очередь для обработки вершин.

### **3. Применимость:**

Алгоритм Левита применяется в ситуациях, когда необходимо находить кратчайшие пути в графах с неотрицательными весами рёбер. Он часто используется в задачах маршрутизации и оптимизации.

### **4. Особенности работы:**

Алгоритм работает эффективно на разреженных графах и может быть адаптирован для работы с различными структурами данных. Он также гарантирует нахождение оптимальных решений при условии отсутствия отрицательных циклов.

### **5. Примеры применения:**

- Маршрутизация в компьютерных сетях.
- Оптимизация логистических маршрутов.
- Планирование задач в операционных системах.
- Анализ социальных сетей и взаимодействий.

## **Сравнительный анализ с аналогичными алгоритмами**

Алгоритм Левита можно сравнить с другими известными алгоритмами нахождения кратчайших путей, такими как алгоритмы Дейкстры и Беллмана-Форда.

- Алгоритм Дейкстры: Работает только с неотрицательными весами рёбер и имеет временную сложность  $O((V + E) \log V)$ . Однако он может быть менее эффективным на плотных графах по сравнению с алгоритмом Левита.

- Алгоритм Беллмана-Форда: Подходит для графов с отрицательными весами и имеет временную сложность  $O(VE)$ . Однако он менее эффективен на графах с неотрицательными весами по сравнению с алгоритмом Левита.

Таким образом, выбор алгоритма зависит от характеристик конкретного графа и требований к производительности.

## Перечень инструментов для реализации

### 1. Языки программирования:

Алгоритм Левита можно реализовать на различных языках программирования, таких как:

- Python
- C++
- Java
- C#
- JavaScript

### 2. Библиотеки:

Существуют специализированные библиотеки для работы с графами, которые могут упростить реализацию алгоритма:

- NetworkX (Python)
- Boost Graph Library (C++)
- JGraphT (Java)

### 3. Инструменты разработки:

Для разработки и тестирования алгоритма можно использовать:

- IDE (например, PyCharm, Visual Studio, Eclipse)
- Системы контроля версий (например, Git)
- Платформы для совместной работы (например, GitHub)

## Описание реализации алгоритма на языке Python

```
import heapq # Импортируем модуль для работы с приоритетными очередями
```

```
def dijkstra(graph, start):
```

```
    # Инициализация расстояний до всех вершин как бесконечность
```

```
    distances = {vertex: float('infinity') for vertex in graph}
```

```
    distances[start] = 0 # Расстояние до стартовой вершины равно 0
```

```
    priority_queue = [(0, start)] # Приоритетная очередь инициализируется стартовой вершиной
```

```
if not graph or start not in graph:  
    return {}
```

```
while priority_queue: # Пока есть вершины в очереди  
    current_distance, current_vertex = heapq.heappop(priority_queue) # Извлекаем вершину  
    с минимальным расстоянием  
  
    # Если найденное расстояние больше, чем уже известное, пропускаем  
    if current_distance > distances[current_vertex]:  
        continue  
  
    # Проходим по всем соседям текущей вершины  
    for neighbor, weight in graph[current_vertex].items():  
        distance = current_distance + weight # Вычисляем новое расстояние до соседа  
  
        # Если найдено более короткое расстояние до соседа  
        if distance < distances[neighbor]:  
            distances[neighbor] = distance # Обновляем расстояние до соседа  
            heapq.heappush(priority_queue, (distance, neighbor)) # Добавляем соседа в очередь  
  
return distances # Возвращаем словарь с кратчайшими расстояниями
```

```
# Определение графа в виде словаря
```

```
graph = {  
    'A': {'B': 1, 'C': 4},  
    'B': {'A': 1, 'C': 2, 'D': 5},  
    'C': {'A': 4, 'B': 2, 'D': 1},  
    'D': {'B': 5, 'C': 1}  
}
```

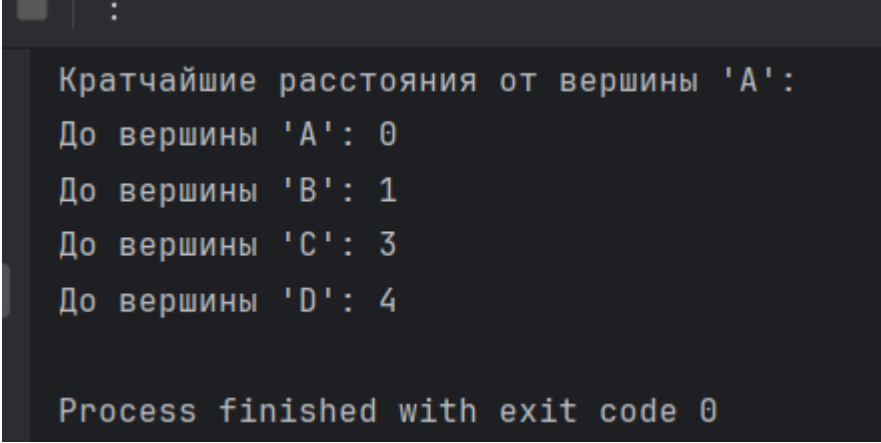
```
start_vertex = 'A' # Задаем стартовую вершину
```



```
distances = dijkstra(graph, start_vertex) # Вызываем функцию Дейкстры

# Выводим кратчайшие расстояния от стартовой вершины
print(f"Кратчайшие расстояния от вершины '{start_vertex}':")
for vertex, distance in distances.items():
    print(f"До вершины '{vertex}': {distance}") # Печатаем расстояния до каждой вершины
```

#### Выполнение кода:



```
Кратчайшие расстояния от вершины 'A':
До вершины 'A': 0
До вершины 'B': 1
До вершины 'C': 3
До вершины 'D': 4

Process finished with exit code 0
```

#### Тестирование:

```
import unittest

from mine import dijkstra

class TestDijkstra(unittest.TestCase):

    def test_dijkstra_empty_graph(self):
        # Тестируем пустой граф
        empty_graph = {} # Создаем пустой граф
        expected_distances = {} # Ожидаем, что расстояния также будут пустыми
        result = dijkstra(empty_graph, 'A') # Вызываем функцию dijkstra с пустым графом
        self.assertEqual(result, expected_distances) # Проверяем, что результат совпадает с
        ожидаемым

    def setUp(self):
        # Метод, который выполняется перед каждым тестом
```

```

# Создаем графы для последующих тестов
self.graph1 = {
    'A': {'B': 1, 'C': 4}, # Вершина A соединена с B и C
    'B': {'A': 1, 'C': 2, 'D': 5}, # Вершина B соединена с A, C и D
    'C': {'A': 4, 'B': 2, 'D': 1}, # Вершина C соединена с A, B и D
    'D': {'B': 5, 'C': 1} # Вершина D соединена с B и C
}

self.graph2 = {
    'A': {'B': 2}, # Вершина A соединена только с B
    'B': {'A': 2, 'C': 3}, # Вершина B соединена с A и C
    'C': {'B': 3, 'D': 1}, # Вершина C соединена с B и D
    'D': {'C': 1} # Вершина D соединена только с C
}

def test_dijkstra_graph1(self):
    # Тестируем первый граф (graph1)
    expected_distances = {
        'A': 0, # Расстояние от A до A равно 0
        'B': 1, # Расстояние от A до B равно 1
        'C': 3, # Расстояние от A до C равно 3 (через B)
        'D': 4 # Расстояние от A до D равно 4 (через B и C)
    }

    result = dijkstra(self.graph1, 'A') # Вызываем функцию dijkstra с graph1 и стартовой вершиной A

    self.assertEqual(result, expected_distances) # Проверяем, что результат совпадает с ожидаемым

def test_dijkstra_graph2(self):
    # Тестируем второй граф (graph2)
    expected_distances = {
        'A': 0, # Расстояние от A до A равно 0
        'B': 2, # Расстояние от A до B равно 2
        'C': 5, # Расстояние от A до C равно 5 (через B)
    }

```

```

        'D': 6 # Расстояние от A до D равно 6 (через B и C)
    }

    result = dijkstra(self.graph2, 'A') # Вызываем функцию dijkstra с graph2 и стартовой
    вершиной A

    self.assertEqual(result, expected_distances) # Проверяем, что результат совпадает с
    ожидаемым

def test_dijkstra_empty_graph(self):
    # Тестируем пустой граф (дублирующий тест)
    empty_graph = {} # Создаем пустой граф
    expected_distances = {} # Ожидаем, что расстояния также будут пустыми
    result = dijkstra(empty_graph, 'A') # Вызываем функцию dijkstra с пустым графом
    self.assertEqual(result, expected_distances) # Проверяем, что результат совпадает с
    ожидаемым

```

### Выполнение тестирования:

```

✓ Test Results 0 ms
  ✓ test 0 ms
    ✓ TestDijkstra 0 ms
      ✓ test_dijkstra_empty_graph 0 ms
      ✓ test_dijkstra_graph1 0 ms
      ✓ test_dijkstra_graph2 0 ms

```

```

✓ Tests passed: 3 of 3 tests - 0 ms
Ran 3 tests in 0.003s
OK
Launching unittests with arguments python -m unittest
Кратчайшие расстояния от вершины 'A':
До вершины 'A': 0
До вершины 'B': 1
До вершины 'C': 3
До вершины 'D': 4
Process finished with exit code 0

```

### Заключение

Алгоритм Левита представляет собой мощный инструмент для решения задачи нахождения кратчайших расстояний в графах с неотрицательными весами рёбер. Его эффективность и простота реализации делают его популярным выбором среди разработчиков и исследователей. Несмотря на наличие других алгоритмов, таких как Дейкстра и Беллмана-Форда, алгоритм Левита остаётся актуальным благодаря своей производительности и применимости в различных областях.

Кроме того, алгоритм Левита демонстрирует высокую скорость работы на разреженных графах, что делает его особенно полезным в задачах, связанных с большими сетями,

такими как маршрутизация в компьютерных сетях, планирование транспортных потоков и анализ социальных сетей. Он также может быть эффективно адаптирован для решения задач в динамических графах, где веса рёбер могут изменяться во времени.

В заключение, алгоритм Левита не только служит важным инструментом в теории графов, но и находит широкое применение в реальных задачах, подтверждая свою ценность в области компьютерных наук и смежных дисциплин. Разработка новых оптимизаций и улучшений этого алгоритма продолжает оставаться актуальной темой для исследований, что свидетельствует о его значимости и потенциале для будущих достижений.