**Доклад: “** **Алгоритм построения кратчайших путей на сети с единичными длинами ”.**

**Алгоритмы решений.**

**по предмету Комбинаторика и теория графов**

**Выполнил обучающийся НИТУ МИСИС**

**Группа БИВТ-23-1**

**Зуйков Арсений Николаевич**

**Ссылка на реализацию:**

**[GitHub](** **https://github.com/jlobsterino/BFS)**

**Оглавление**

[**Формальная постановка задачи** 3](#_Toc183908347)

[**Описание задачи** 3](#_Toc183908348)

[**Входные данные** 3](#_Toc183908349)

[**Выходные данные** 3](#_Toc183908350)

[**Требования к решению** 3](#_Toc183908351)

[**Ограничения** 4](#_Toc183908352)

[**Теоретическое описание алгоритма** 4](#_Toc183908353)

[**Идея алгоритма** 4](#_Toc183908354)

[**Основные элементы алгоритма** 4](#_Toc183908355)

[**Шаги алгоритма** 5](#_Toc183908356)

[**Применимость** 5](#_Toc183908357)

[**Характеристика алгоритма** 5](#_Toc183908358)

[**Временная сложность** 5](#_Toc183908359)

[**Пространственная сложность** 6](#_Toc183908360)

[**Сравнительный анализ алгоритма с аналогичными** 6](#_Toc183908361)

[**Алгоритмы для решения задачи кратчайших путей** 6](#_Toc183908362)

[**Алгоритм поиска в ширину (BFS)** 7](#_Toc183908363)

[**Алгоритм Дейкстры (Dijkstra)** 7](#_Toc183908364)

[**Алгоритм Беллмана-Форда (Bellman-Ford)** 8](#_Toc183908365)

[**Алгоритм Флойда-Уоршелла (Floyd-Warshall)** 8](#_Toc183908366)

[**Перечень инструментов для реализации** 8](#_Toc183908367)

[**Языки программирования** 8](#_Toc183908368)

[**Библиотеки и фреймворки** 9](#_Toc183908369)

[**Инструменты для тестирования и отладки** 10](#_Toc183908370)

[**Инструменты для визуализации** 10](#_Toc183908371)

[**Описание реализации и процесса тестирования на языке Java** 11](#_Toc183908372)

[**Код алгоритма:** 11](#_Toc183908373)

[**Результат работы программы:** 13](#_Toc183908374)

[**Тестирование алгоритма:** 13](#_Toc183908375)

[**Результаты тестов:** 15](#_Toc183908376)

[**Заключение** 15](#_Toc183908377)

# **Формальная постановка задачи**

## **Описание задачи**

Имеется граф G=(V,E), где:

* V — множество вершин (узлов) графа.
* E — множество рёбер графа.
* Каждое ребро e∈E имеет одинаковую длину, равную 1.

Необходимо найти минимальное количество рёбер, через которые нужно пройти, чтобы попасть из начальной вершины s∈V до каждой другой вершины графа v∈V.

## **Входные данные**

**Граф G=(V,E):**

Представляется в виде:

* **Списка смежности:** массив, где graph[i] содержит список вершин, смежных с вершиной i.
* **Матрицы смежности:** квадратная матрица n×n, где n=∣V∣, а matrix[i][j]=1, если вершины i и j соединены ребром, иначе matrix[i][j]=0.

Граф может быть:

* **Ориентированным:** Ребро имеет направление (например, из u в v).
* **Неориентированным:** Ребро двустороннее (например, из u в v и из v в u).

**Начальная вершина s:**

* Вершина s∈V, с которой начинается обход графа.

## **Выходные данные**

Результат алгоритма — массив d, где:

* d[i] — кратчайшее расстояние (в рёбрах) от начальной вершины s до вершины i.
* Если вершина i недостижима из s, то d[i]=∞ (или некоторый условный маркер, например, Integer.MAX\_VALUE.

## **Требования к решению**

1. **Корректность:**

* Алгоритм должен корректно находить кратчайшие расстояния для всех вершин, достижимых из начальной вершины s.
* Недостижимые вершины должны быть помечены как недостижимые (∞).

1. **Оптимальность:**

* Алгоритм должен использовать минимальное количество операций для выполнения поиска.

1. **Эффективность:**

* Решение должно иметь временную сложность O(V+E), где V — количество вершин, E — количество рёбер.
* Пространственная сложность O(V) для хранения расстояний и очереди.

## **Ограничения**

* Количество вершин ∣V∣: 1≤∣V∣≤10^5.
* Количество рёбер ∣E∣: 0≤∣E∣≤10^6.
* Граф может быть как связным (все вершины достижимы друг из друга), так и несвязным (некоторые вершины недостижимы).
* Вес каждого ребра равен 1.

# **Теоретическое описание алгоритма**

## **Идея алгоритма**

Алгоритм построения кратчайших путей на сети с единичными длинами основан на методе **поиска в ширину (Breadth-First Search, BFS)**. BFS — это метод обхода графа, при котором все вершины на одном уровне от начальной вершины исследуются до перехода к следующему уровню.

В графе с одинаковыми весами рёбер w=1, BFS позволяет гарантированно найти минимальное количество шагов (рёбер) от начальной вершины s до всех остальных вершин, так как:

* BFS сначала обрабатывает ближайшие вершины, а затем те, что находятся дальше.
* Алгоритм завершает обработку вершины, как только она впервые посещается, и на этом этапе расстояние до неё минимально.

## **Основные элементы алгоритма**

1. **Граф:**

* Представляется в виде:
  + Списка смежности: для каждой вершины хранится список всех вершин, с которыми она соединена рёбрами.
  + Матрицы смежности: квадратная матрица, где matrix[i][j]=1, если вершины i и j соединены, иначе 0.

1. **Начальная вершина (s):**

* Вершина, с которой начинается поиск.

1. **Очередь:**

* BFS использует очередь (FIFO — "первый пришёл, первый вышел"), чтобы обрабатывать вершины уровня за уровнем.

1. **Массив расстояний (d):**

* Хранит минимальное количество шагов от начальной вершины s до каждой вершины графа.
* Инициализируется значением ∞ (или очень большим числом), кроме начальной вершины s, для которой d[s]=0.

## **Шаги алгоритма**

1. **Инициализация:**

* Устанавливаем d[s]=0 (расстояние до начальной вершины равно нулю).
* Для всех остальных вершин d[i]=∞ (расстояние пока неизвестно).
* Создаём пустую очередь и добавляем начальную вершину s.

1. **Цикл обработки:**

* Пока очередь не пуста:
  + 1. Извлекаем текущую вершину u из очереди.
    2. Для каждого соседа v вершины u:
    - Если вершина v ещё не посещена (d[v]=∞):
      * Обновляем расстояние d[v]=d[u]+1.
      * Добавляем v в очередь.

1. **Завершение:**

* Когда очередь пуста, все достижимые вершины обработаны, и в массиве d находятся минимальные расстояния от начальной вершины s до каждой другой вершины.

## **Применимость**

Алгоритм подходит для задач, где:

* Все рёбра имеют одинаковую длину.
* Требуется найти минимальное количество шагов для достижения цели.

Примеры:

* Навигация в городах, где расстояния между пунктами равны.
* Анализ социальных сетей, например, минимальное количество связей между пользователями.
* Лабиринты, где каждая клетка имеет одинаковую "стоимость" перехода.

# **Характеристика алгоритма**

## **Временная сложность**

**Анализ шагов алгоритма**

1. **Инициализация:**

* Создание массива расстояний d, который хранит минимальное количество рёбер от начальной вершины до каждой другой: O(V)
* Создание очереди для обхода вершин: O(1)

1. **Обход графа:**

* Каждая вершина добавляется в очередь и обрабатывается ровно один раз: O(V)
* Для каждой вершины проверяются все её соседи, то есть каждое ребро обрабатывается один раз: O(E)

**Общая временная сложность**

Общая временная сложность равна O(V+E) где:

* V — количество вершин
* E — количество рёбер.

**Особенности временной сложности**

* Линейная сложность делает алгоритм пригодным для больших графов.
* В худшем случае (полный граф) E≈V^2E, что приводит к сложности O(V^2). Однако в реальных задачах графы часто разреженные (E<<V^2), что даёт близкую к O(V) производительность.

## **Пространственная сложность**

**Анализ используемой памяти**

1. **Массив расстояний** (d):

* Требуется O(V) памяти для хранения минимальных расстояний от начальной вершины до каждой другой вершины.

1. **Очередь**:

* В худшем случае, если все вершины находятся на одном уровне, очередь может содержать до O(V) элементов.

1. **Список смежности (или матрица)**:

* Список смежности занимает O(V+E) памяти (хранение всех рёбер).
* Матрица смежности занимает O(V^2) (но используется редко, так как она менее эффективна для разреженных графов.

**Общая пространственная сложность**

**Список смежности**:

* O(V+E)

**Матрица смежности**:

* O(V^2)

# **Сравнительный анализ алгоритма с аналогичными**

## **Алгоритмы для решения задачи кратчайших путей**

Существует несколько популярных алгоритмов для нахождения кратчайших путей в графе. Их выбор зависит от свойств графа:

* Все рёбра имеют единичный вес: **BFS**.
* Рёбра имеют положительные веса: **Алгоритм Дейкстры (Dijkstra)**.
* Граф может содержать отрицательные веса рёбер: **Алгоритм Беллмана-Форда (Bellman-Ford)**.
* Нахождение кратчайших путей между всеми парами вершин: **Алгоритм Флойда-Уоршелла**.

## **Алгоритм поиска в ширину (BFS)**

**Основное назначение:** Построение кратчайших путей в графе с единичными весами рёбер.

**Преимущества:**

1. **Оптимальность для единичных весов:**

* BFS гарантирует нахождение кратчайшего пути в графе, где длина всех рёбер равна 1.
* Нет необходимости учитывать веса рёбер, что упрощает реализацию.

1. **Линейная сложность:**

* Временная сложность O(V+E), где V — количество вершин, E — количество рёбер.
* Оптимально для разреженных графов.

1. **Простота реализации:**

* Использует только очередь и массив для хранения расстояний.

**Недостатки:**

1. **Ограничение на единичные веса:**

* Алгоритм не работает для графов с различными или отрицательными весами рёбер.

1. **Память:**

* Для плотных графов требуется O(V^2) памяти при использовании матрицы смежности.

## **Алгоритм Дейкстры (Dijkstra)**

**Основное назначение:** Построение кратчайших путей в графе с положительными весами рёбер.

**Преимущества:**

1. **Гибкость:**

* Работает для графов с положительными весами рёбер.

1. **Эффективность для разреженных графов:**

* При использовании кучи (например, двоичной или Фибоначчиевой) сложность уменьшается до O((V+E)log(V)).

1. **Оптимальность:**

* Гарантированно находит кратчайшие пути при условии положительных весов.

**Недостатки:**

1. **Сложность реализации:**

* Реализация с использованием кучи требует больше кода, чем BFS.

1. **Не подходит для отрицательных весов:**

* Если граф содержит отрицательные веса рёбер, алгоритм может не работать корректно.

1. **Худшая производительность на графах с единичными весами:**

* Для таких графов BFS быстрее и проще.

## **Алгоритм Беллмана-Форда (Bellman-Ford)**

**Основное назначение:** Построение кратчайших путей в графах с отрицательными весами рёбер.

**Преимущества:**

1. **Гибкость:**

* Подходит для графов с отрицательными весами.

1. **Обнаружение отрицательных циклов:**

* Если граф содержит цикл с отрицательной суммой весов, алгоритм может его выявить.

**Недостатки:**

1. **Медлительность:**

* Сложность O(V⋅E) делает его менее эффективным для больших графов.

1. **Неоптимальность для графов с единичными весами:**

* Для графов с одинаковыми весами рёбер этот алгоритм слишком медленный.

## **Алгоритм Флойда-Уоршелла (Floyd-Warshall)**

**Основное назначение:** Нахождение кратчайших путей между всеми парами вершин.

**Преимущества:**

1. **Все пары вершин:**

* Алгоритм находит пути между любыми двумя вершинами.

1. **Простота реализации:**

* Использует матрицу для хранения расстояний.

**Недостатки:**

1. **Высокая сложность:**

* Сложность O(V3), что делает его непригодным для больших графов.

1. **Неэффективность для одного источника:**

* Если нужен путь только от одной вершины, избыточен.

# **Перечень инструментов для реализации**

## **Языки программирования**

**1. Java**

**Преимущества:**

* Надежность: строгая типизация предотвращает множество ошибок.
* Удобные стандартные коллекции (например, Queue, List).
* Отлично подходит для крупных проектов благодаря объектно-ориентированному подходу.
* Богатая экосистема для тестирования, отладки и анализа.

**Недостатки:**

* Производительность ниже, чем у C++.
* Более многословный код по сравнению с Python.

**2. Python**

**Преимущества:**

* Простота синтаксиса: минимальное количество кода для реализации алгоритма.
* Обширная стандартная библиотека (например, collections для очереди).
* Быстрое прототипирование и удобство для обучения.

**Недостатки:**

* Производительность ниже, чем у Java и C++, особенно на больших графах.
* Отсутствие строгой типизации может привести к ошибкам в сложных системах.

**3. C++**

**Преимущества:**

* Высокая производительность благодаря низкоуровневому управлению памятью.
* STL (Standard Template Library) предоставляет эффективные структуры данных и алгоритмы.
* Идеально для задач, где требуется максимальная оптимизация.

**Недостатки:**

* Сложность реализации из-за управления памятью.
* Более сложный синтаксис по сравнению с Java и Python.

## **Библиотеки и фреймворки**

**1. Java**

* **Java Collections Framework**:
  + Встроенная поддержка Queue (например, LinkedList) для реализацииBFS.
  + ArrayList для представления графа как списка смежности.
* **JGraphT**:
  + Библиотека для работы с графами.
  + Поддерживает представление графов, обходы и визуализацию.

**2. Python**

* **collections**:
* Модуль стандартной библиотеки, предоставляющий deque для очереди в BFS.
* **NetworkX**:
* Библиотека для работы с графами.
* Поддерживает построение графов, поиск кратчайших путей (включая BFS), визуализацию.
* **igraph**:
* Высокопроизводительная библиотека для анализа больших графов.

**3. C++**

* **Standard Template Library (STL)**:
* queue для реализации BFS.
* vector для хранения графа как списка смежности.
* **Boost Graph Library**:
* Расширенная библиотека для работы с графами.
* Поддерживает поиск кратчайших путей, включая BFS.

## **Инструменты для тестирования и отладки**

**1. Java**

* **JUnit**:
* Фреймворк для модульного тестирования.
* Подходит для проверки корректности алгоритма и сценариев на граничных условиях.
* **IntelliJ IDEA Debugger**:
* Встроенный отладчик, поддерживающий точки останова, просмотр стека вызовов и локальных переменных.

**2. Python**

* **unittest**:
* Встроенный модуль для написания тестов.
* **pytest**:
* Более мощный инструмент, поддерживающий сложные сценарии тестирования.
* **PDB (Python Debugger)**:
* Интерактивный отладчик, встроенный в стандартную библиотеку Python.

**3. C++**

* **Google Test (GTest)**:
* Популярный фреймворк для модульного тестирования.
* **GDB (GNU Debugger)**:
* Мощный отладчик для пошагового выполнения кода.
* Подходит для выявления ошибок в логике алгоритма.

## **Инструменты для визуализации**

**1. Java**

* **JGraphT**:
* Поддерживает визуализацию графов через интеграцию с библиотеками визуализации, например, Graphviz.

**2. Python**

* **NetworkX + Matplotlib**:
* Визуализация графов через NetworkX и построение графиков через Matplotlib.

**3. C++**

* **Boost Graph Library**:
* Поддерживает интеграцию с Graphviz для визуализации графов.
* **Graphviz**:
* Генерация графов через экспорт в формат DOT.

# **Описание реализации и процесса тестирования на языке Java**

## **Код алгоритма:**

import org.jgrapht.Graph;  
import org.jgrapht.graph.DefaultEdge;  
  
import java.util.\*;  
  
public class ShortestPathJGraphT {  
 public static <V> Map<V, Integer> bfsShortestPaths(Graph<V, DefaultEdge> graph, V start) {  
 // Карта для хранения кратчайших расстояний  
 Map<V, Integer> distances = new HashMap<>();  
 for (V vertex : graph.vertexSet()) {  
 distances.put(vertex, Integer.*MAX\_VALUE*); // Инициализируем расстояния как бесконечные  
 }  
 distances.put(start, 0); // Расстояние до начальной вершины = 0  
  
 // Очередь для BFS  
 Queue<V> queue = new LinkedList<>();  
 queue.add(start);  
  
 // BFS  
 while (!queue.isEmpty()) {  
 V current = queue.poll();  
 int currentDistance = distances.get(current);  
  
 // Обрабатываем всех соседей текущей вершины  
 for (V neighbor : graph.edgesOf(current).stream()  
 .map(edge -> graph.getEdgeTarget(edge).equals(current) ?  
 graph.getEdgeSource(edge) : graph.getEdgeTarget(edge))  
 .toList()) {  
 // Если сосед ещё не посещён  
 if (distances.get(neighbor) == Integer.*MAX\_VALUE*) {  
 distances.put(neighbor, currentDistance + 1);  
 queue.add(neighbor); // Добавляем соседа в очередь  
 }  
 }  
 }  
  
 return distances; // Возвращаем карту с кратчайшими расстояниями  
 }  
}

import org.jgrapht.Graph;  
import org.jgrapht.graph.DefaultEdge;  
import org.jgrapht.graph.SimpleGraph;  
  
import java.util.\*;  
  
public class Runner {  
 public static void main(String[] args) {  
 // Создаём граф  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
  
 // Добавляем вершины  
 graph.addVertex("A");  
 graph.addVertex("B");  
 graph.addVertex("C");  
 graph.addVertex("D");  
 graph.addVertex("E");  
  
 // Добавляем рёбра (они автоматически имеют единичный вес)  
 graph.addEdge("A", "B");  
 graph.addEdge("A", "C");  
 graph.addEdge("B", "D");  
 graph.addEdge("C", "D");  
 graph.addEdge("D", "E");  
  
 // Запускаем алгоритм  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
  
 // Вывод результатов  
 System.*out*.println("Кратчайшие расстояния от вершины A:");  
 for (Map.Entry<String, Integer> entry : distances.entrySet()) {  
 System.*out*.println("Вершина " + entry.getKey() + ": " +  
 (entry.getValue() == Integer.*MAX\_VALUE* ? "недостижима" : entry.getValue()));  
 }  
 }  
}

### **Результат работы программы:**

Кратчайшие расстояния от вершины A:

Вершина A: 0

Вершина B: 1

Вершина C: 1

Вершина D: 2

Вершина E: 3

Process finished with exit code 0

## **Тестирование алгоритма:**

import org.jgrapht.Graph;  
import org.jgrapht.graph.DefaultEdge;  
import org.jgrapht.graph.SimpleGraph;  
import org.junit.jupiter.api.Test;  
  
import java.util.Map;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class ShortestPathJGraphTTest {  
  
 @Test  
 public void testConnectedGraph() {  
 // Создаём связный граф  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
 graph.addVertex("A");  
 graph.addVertex("B");  
 graph.addVertex("C");  
 graph.addEdge("A", "B");  
 graph.addEdge("B", "C");  
  
 // Проверяем кратчайшие расстояния  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
 *assertEquals*(0, distances.get("A"));  
 *assertEquals*(1, distances.get("B"));  
 *assertEquals*(2, distances.get("C"));  
 }  
  
 @Test  
 public void testDisconnectedGraph() {  
 // Создаём неразвязный граф  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
 graph.addVertex("A");  
 graph.addVertex("B");  
 graph.addVertex("C");  
 graph.addEdge("A", "B");  
  
 // Проверяем кратчайшие расстояния  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
 *assertEquals*(0, distances.get("A"));  
 *assertEquals*(1, distances.get("B"));  
 *assertEquals*(Integer.*MAX\_VALUE*, distances.get("C")); // Вершина "C" недостижима  
 }  
  
 @Test  
 public void testSingleNodeGraph() {  
 // Создаём граф из одной вершины  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
 graph.addVertex("A");  
  
 // Проверяем кратчайшие расстояния  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
 *assertEquals*(0, distances.get("A"));  
 }  
  
 @Test  
 public void testCyclicGraph() {  
 // Создаём граф с циклом  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
 graph.addVertex("A");  
 graph.addVertex("B");  
 graph.addVertex("C");  
 graph.addEdge("A", "B");  
 graph.addEdge("B", "C");  
 graph.addEdge("C", "A");  
  
 // Проверяем кратчайшие расстояния  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
 *assertEquals*(0, distances.get("A"));  
 *assertEquals*(1, distances.get("B"));  
 *assertEquals*(1, distances.get("C"));  
 }  
  
 @Test  
 public void testEmptyGraph() {  
 // Создаём пустой граф  
 Graph<String, DefaultEdge> graph = new SimpleGraph<>(DefaultEdge.class);  
  
 try {  
 // Пытаемся выполнить BFS на пустом графе  
 Map<String, Integer> distances = ShortestPathJGraphT.*bfsShortestPaths*(graph, "A");  
 *fail*("Ожидалось исключение IllegalArgumentException из-за отсутствия вершины в графе");  
 } catch (IllegalArgumentException e) {  
 *assertEquals*("no such vertex in graph: A", e.getMessage());  
 }  
 }  
}

### **Результаты тестов:**

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Все тесты завершились успешно

# **Заключение**

Алгоритм построения кратчайших путей на сети с единичными длинами, основанный на **поиске в ширину (BFS)**, является одним из самых эффективных решений для задач, где рёбра графа имеют одинаковую длину. Этот алгоритм обеспечивает оптимальность и простоту реализации при решении задач нахождения минимальных путей между вершинами, так как он всегда находит кратчайший путь в графах с единичными весами рёбер.

Применение алгоритма ограничивается графами с равными весами рёбер, что делает его особенно полезным для задач, таких как маршрутизация в сетях, анализ социальных связей, навигация в лабиринтах, и многих других приложениях, где все шаги (рёбра) имеют одинаковую "стоимость".

**Преимущества алгоритма заключаются в:**

* Простоте реализации: Алгоритм легко реализуется с использованием стандартных структур данных, таких как очередь и массив для хранения расстояний.
* Время работы: Алгоритм работает за время O(V+E), что делает его подходящим для больших графов, особенно для разреженных графов, где количество рёбер значительно меньше квадрата количества вершин.
* Корректность: Благодаря своей природе, BFS всегда находит кратчайший путь в графах с единичными длинами рёбер.

**Недостатками являются:**

* Ограниченность в применении только к графам с одинаковыми весами рёбер. Для графов с разными весами рёбер или графов, содержащих отрицательные веса, требуется использовать другие алгоритмы, такие как алгоритм Дейкстры или Беллмана-Форда.
* Память: В случае использования матрицы смежности для представления графа, память будет занимать O(V^2), что может быть проблемой для очень крупных графов.