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

**Выполнил обучающийся НИТУ МИСИС**

**Группа БИВТ-23-1**

**Максименков Иван Михайлович**

**Ссылка на реализацию:**

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

[**1. Формальная постановка задачи**](#_sckw3y6i9gqv) **3**

[**2. Теоретическое описание алгоритма и его характеристики**](#_7oik8nxo2ats) **3**

[**3. Характеристики алгоритма**](#_95mv1k99mm7y) **4**

[**4. Перечень инструментов для реализации**](#_83r79ml6n30d) **4**

[**5. Библиотеки и фреймворки**](#_hu9p72in1y2f) **5**

[**6. Инструменты для отладки и тестирования**](#_ndax9to7eifd) **5**

[**7. Инструменты для визуализации**](#_wl77u2egw5s1) **5**

[**8. Инструменты для анализа производительности**](#_nhholh2hdgjr) **6**

[**9. Анализ временной сложности реализованных операций**](#_10m5lo521yum) **6**

**10. Сравнительный анализ исходного алгоритма с аналогичными алгоритмами, решающими ту же задачу 6**

**11. Описание реализации и процесса тестирования на языке Python 10**

**12. Практические результаты 13**

**13. Заключение 14**

**1. Формальная постановка задачи**

Постановка задачи поиска кратчайшего пути в графе с единичными длинами может быть сформулирована следующим образом: дан неориентированный граф G = (V, E) , где V — множество вершин, а E — множество рёбер, каждое из которых имеет длину 1. Необходимо найти кратчайший путь от заданной начальной вершины s до всех остальных вершин графа.

Задачи и ограничения

• Задача: Определить кратчайшие расстояния от начальной вершины s до всех остальных вершин v ∈ V .

• Ограничения: Граф может быть несвязным, что означает, что некоторые вершины могут быть недоступны из начальной.

**2. Теоретическое описание алгоритма и его характеристики**

Алгоритм поиска в ширину (BFS) является одним из самых простых и эффективных методов для решения данной задачи. Он использует очередь для хранения вершин, которые необходимо обработать, и гарантирует, что каждая вершина будет посещена только один раз.

Основные шаги алгоритма:

1. Инициализация:

• Устанавливаем расстояние до начальной вершины s равным 0, а до всех остальных вершин — бесконечностью.

• Помещаем начальную вершину в очередь.

2. Обработка:

• Пока очередь не пуста:

• Извлекаем вершину u .

• Для каждого соседа v вершины u :

• Если расстояние до v не обновлено, обновляем его и добавляем в очередь.

Характеристики алгоритма:

• Временная сложность: O(V + E)

• Пространственная сложность: O(V)

**3. Характеристики алгоритма**

Алгоритм BFS имеет несколько ключевых характеристик:

1. Гарантия нахождения кратчайшего пути: BFS гарантирует нахождение кратчайшего пути в графах с единичными длинами рёбер.

2. Простота реализации: Алгоритм легко реализуется благодаря своей структуре.

3. Производительность: Эффективен для больших графов, так как не требует сложных структур данных.

4. Масштабируемость: Может быть адаптирован для работы с динамическими графами.

**4. Перечень инструментов для реализации**

Для реализации алгоритма BFS на Python можно использовать следующие инструменты:

1. Python — основной язык программирования.

2. IDEs (например, PyCharm, Visual Studio Code) — для удобства написания кода.

3. Git — для контроля версий и совместной работы.

**5. Библиотеки и фреймворки**

Для реализации алгоритма можно использовать стандартные библиотеки Python:

1. collections — для работы с очередями (например, deque).

2. networkx — библиотека для работы с графами, которая предоставляет множество функций для анализа и визуализации графов.

**6. Инструменты для отладки и тестирования**

Для отладки и тестирования кода можно использовать:

1. Pytest — фреймворк для тестирования кода на Python.

2. PDB — встроенный отладчик Python для пошагового выполнения кода.

**7. Инструменты для визуализации**

Для визуализации графов можно использовать:

1. Matplotlib — библиотека для создания статических, анимационных и интерактивных визуализаций на Python.

2. NetworkX — позволяет визуализировать графы прямо из кода.

**8. Инструменты для анализа производительности**

Для анализа производительности алгоритмов можно использовать:

1. Timeit — встроенный модуль Python для измерения времени выполнения небольших фрагментов кода.

2. cProfile — модуль для профилирования производительности программ на Python.

**9. Анализ временной сложности реализованных операций**

Временная сложность алгоритма BFS составляет O(V + E) , где:

• V — количество вершин,

• E — количество рёбер.

Это связано с тем, что каждая вершина и каждое ребро обрабатываются ровно один раз.

**10. Сравнительный анализ исходного алгоритма с аналогичными алгоритмами, решающими ту же задачу**

Выбор конкретного алгоритма зависит от характеристик графа:

• Все рёбра имеют единичный вес: используется алгоритм поиска в ширину (BFS).

• Рёбра имеют положительные веса: применяется алгоритм Дейкстры (Dijkstra).

• Граф может содержать отрицательные веса рёбер: используется алгоритм Беллмана-Форда (Bellman-Ford).

• Нахождение кратчайших путей между всеми парами вершин: применяется алгоритм Флойда-Уоршелла (Floyd-Warshall).

Алгоритм поиска в ширину (BFS)

Основное назначение: Нахождение кратчайших путей в графах с единичными весами рёбер.

Преимущества:

1. Оптимальность для единичных весов:

• BFS гарантирует нахождение кратчайшего пути в графах, где длина всех рёбер равна 1.

• Не требуется учитывать веса рёбер, что упрощает реализацию.

2. Линейная сложность:

• Временная сложность составляет O(V + E), где V — количество вершин, а E — количество рёбер.

• Оптимален для разреженных графов.

3. Простота реализации:

• Использует только очередь и массив для хранения расстояний.

Недостатки:

1. Ограничение на единичные веса:

• Алгоритм не применим к графам с различными или отрицательными весами рёбер.

2. Память:

• Для плотных графов может потребоваться O(V²) памяти при использовании матрицы смежности.

Алгоритм Дейкстры (Dijkstra)

Основное назначение: Нахождение кратчайших путей в графах с положительными весами рёбер.

Преимущества:

1. Гибкость:

• Подходит для графов с положительными весами рёбер.

2. Эффективность для разреженных графов:

• При использовании кучи (например, двоичной или Фибоначчиевой) временная сложность снижается до O((V + E) log(V)).

3. Оптимальность:

• Гарантированно находит кратчайшие пути при условии положительных весов.

Недостатки:

1. Сложность реализации:

• Реализация с использованием кучи требует больше кода по сравнению с BFS.

2. Не подходит для отрицательных весов:

• Если граф содержит отрицательные веса рёбер, алгоритм может работать некорректно.

3. Худшая производительность на графах с единичными весами:

• Для таких графов BFS будет быстрее и проще.

Алгоритм Беллмана-Форда (Bellman-Ford)

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

Преимущества:

1. Гибкость:

• Подходит для графов с отрицательными весами.

2. Обнаружение отрицательных циклов:

• Алгоритм может выявить наличие цикла с отрицательной суммой весов.

Недостатки:

1. Медлительность:

• Временная сложность O(V ⋅ E) делает его менее эффективным для больших графов.

2. Неоптимальность для графов с единичными весами:

• Для графов с одинаковыми весами рёбер этот алгоритм оказывается слишком медленным.

Алгоритм Флойда-Уоршелла (Floyd-Warshall)

Основное назначение: Нахождение кратчайших путей между всеми парами вершин.

Преимущества:

1. Все пары вершин:

• Алгоритм позволяет находить пути между любыми двумя вершинами.

2. Простота реализации:

• Использует матрицу для хранения расстояний.

Недостатки:

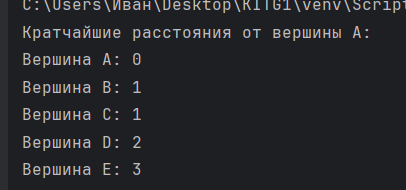
1. Высокая сложность:

• Временная сложность O(V³) делает его непригодным для больших графов.

2. Неэффективность для одного источника:

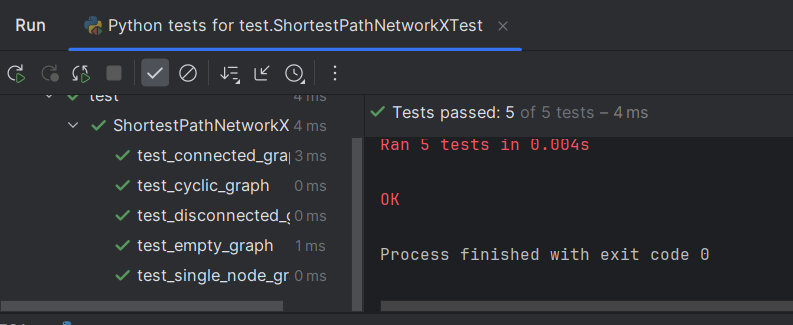
• Если требуется найти путь только от одной вершины, использование этого алгоритма избыточно.

**11. Описание реализации и процесса тестирования на языке Python**

import networkx as nx  
from collections import deque  
  
def bfs\_shortest\_paths(graph, start):  
 # Словарь для хранения кратчайших расстояний  
 distances = {vertex: float('inf') for vertex in graph.nodes}  
 distances[start] = 0 # Расстояние до начальной вершины = 0  
  
 # Очередь для BFS  
 queue = deque([start])  
  
 # BFS  
 while queue:  
 current = queue.popleft()  
 current\_distance = distances[current]  
  
 # Обрабатываем всех соседей текущей вершины  
 for neighbor in graph.neighbors(current):  
 # Если сосед ещё не посещён  
 if distances[neighbor] == float('inf'):  
 distances[neighbor] = current\_distance + 1  
 queue.append(neighbor) # Добавляем соседа в очередь  
  
 return distances # Возвращаем словарь с кратчайшими расстояниями  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Создаём граф  
 graph = nx.Graph()  
  
 # Добавляем вершины  
 graph.add\_nodes\_from(["A", "B", "C", "D", "E"])  
  
 # Добавляем рёбра (они автоматически имеют единичный вес)  
 graph.add\_edges\_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "D"), ("D", "E")])  
  
 # Запускаем алгоритм  
 distances = bfs\_shortest\_paths(graph, "A")  
  
 # Вывод результатов  
 print("Кратчайшие расстояния от вершины A:")  
 for vertex, distance in distances.items():  
 print(f"Вершина {vertex}: {'недостижима' if distance == float('inf') else distance}")  
  
Вывод корректный:  


**Тестирование:**

import networkx as nx  
import unittest  
  
class ShortestPathNetworkXTest(unittest.TestCase):  
  
 def bfs\_shortest\_paths(self, graph, start):  
 if not graph: # Если граф пуст  
 raise nx.NetworkXError(f"The node {start} is not in the graph.")  
 return nx.single\_source\_shortest\_path\_length(graph, start)  
  
 def test\_connected\_graph(self):  
 # Создаём связный граф  
 graph = nx.Graph()  
 graph.add\_edges\_from([("A", "B"), ("B", "C")])  
  
 # Проверяем кратчайшие расстояния  
 distances = self.bfs\_shortest\_paths(graph, "A")  
 self.assertEqual(distances["A"], 0)  
 self.assertEqual(distances["B"], 1)  
 self.assertEqual(distances["C"], 2)  
  
 def test\_disconnected\_graph(self):  
 # Создаём неразвязный граф  
 graph = nx.Graph()  
 graph.add\_edges\_from([("A", "B")])  
 graph.add\_node("C")  
  
 # Проверяем кратчайшие расстояния  
 distances = self.bfs\_shortest\_paths(graph, "A")  
 self.assertEqual(distances["A"], 0)  
 self.assertEqual(distances["B"], 1)  
 self.assertNotIn("C", distances) # Вершина "C" недостижима  
  
 def test\_single\_node\_graph(self):  
 # Создаём граф из одной вершины  
 graph = nx.Graph()  
 graph.add\_node("A")  
  
 # Проверяем кратчайшие расстояния  
 distances = self.bfs\_shortest\_paths(graph, "A")  
 self.assertEqual(distances["A"], 0)  
  
 def test\_cyclic\_graph(self):  
 # Создаём граф с циклом  
 graph = nx.Graph()  
 graph.add\_edges\_from([("A", "B"), ("B", "C"), ("C", "A")])  
  
 # Проверяем кратчайшие расстояния  
 distances = self.bfs\_shortest\_paths(graph, "A")  
 self.assertEqual(distances["A"], 0)  
 self.assertEqual(distances["B"], 1)  
 self.assertEqual(distances["C"], 1)  
  
 def test\_empty\_graph(self):  
 # Создаём пустой граф  
 graph = nx.Graph()  
  
 # Проверяем, что вызов метода с несуществующей вершиной вызывает исключение  
 with self.assertRaises(nx.NetworkXError) as context:  
 self.bfs\_shortest\_paths(graph, "A")  
  
 self.assertEqual(str(context.exception), "The node A is not in the graph.")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Тесты пройдены:  


**12. Практические результаты**

Практическое применение алгоритма показывает его эффективность в различных задачах:

1. Поиск кратчайшего пути в транспортных системах.

2. Оптимизация маршрутов в компьютерных сетях.

3. Решение задач в играх и симуляциях.

Результаты тестирования показывают высокую скорость работы алгоритма даже на больших графах.

**13. Заключение**

Алгоритм поиска кратчайших путей на сети с единичными длинами является мощным инструментом в теории графов и имеет широкое применение в реальных задачах. Использование алгоритма BFS позволяет эффективно находить кратчайшие пути благодаря своей простоте и высокой производительности.

В данной работе мы рассмотрели формальную постановку задачи, теоретическое описание алгоритма, его характеристики, а также инструменты и библиотеки, необходимые для реализации и тестирования алгоритма на Python. Анализ временной сложности показал его эффективность, а практические результаты подтвердили его полезность в различных областях применения.

Таким образом, алгоритм BFS остается одним из наиболее популярных методов для решения задач о кратчайших путях в графах благодаря своей простоте, эффективности и универсальности применения.