Лекция 19. Кратчайшие пути: обход в ширину (BFS), Дейкстра, Форд-Беллман, Флойд, Джонсон.

1. Введение в задачу поиска кратчайших путей

Задача. Дан граф $G = (V, E)$ с весами рёбер w: $E \to R$. Требуется найти кратчайший путь от вершины s до вершины t, где длина пути равна сумме весов рёбер.

Типы задач:

- **Единичные веса.** BFS – сложность $O(n + m)$.
- **Неотрицательные веса.** Дейкстра – сложность $O(m \cdot \log n)$.
- **Произвольные веса.** Форд-Беллман – сложность $O(n \cdot m)$.
- **Все пары вершин.** Флойд – сложность $O(n^3)$, Джонсон – $O(n \cdot m \cdot \log n)$.

2. Обход в ширину (BFS) – единичные веса

Идея. Посещаем вершины в порядке увеличения расстояния от стартовой.

Алгоритм:

1. Инициализируем массив расстояний dist значениями -1 (непосещённые).
2. Очередь queue начинается с стартовой вершины s, dist[s] = 0.
3. Пока очередь не пуста:
- Извлекаем вершину u.
- Для каждого соседа v. Если dist[v] == -1, обновляем dist[v] = dist[u] + 1 и добавляем v в очередь.

Реализация:

```
from collections import deque


def bfs_shortest_path(graph: list[list[int]], start: int) -> list[int]:
    # graph - список смежности (для каждой вершины список смежных вершин)
    # start - начальная вершина, от которой ищем расстояния

    n = len(graph)  # количество вершин в графе
    dist = [-1] * n  # массив расстояний, -1 означает "еще не посещена"
    dist[start] = 0  # расстояние от стартовой вершины до самой себя = 0
    queue = deque([start])  # очередь для BFS, начинаем со стартовой вершины

    while queue:  # пока очередь не пуста
        u = queue.popleft()  # извлекаем первую вершину из очереди
        for v in graph[u]:  # перебираем всех соседей вершины u
            if dist[v] == -1:  # если сосед еще не посещен
                dist[v] = dist[u] + 1  # расстояние = расстояние до u + 1
                queue.append(v)  # добавляем соседа в очередь для обработки
    return dist  # возвращаем массив расстояний от start до всех вершин
```

Сложность: $O(n + m)$.

3. Алгоритм Дейкстры – неотрицательные веса

Идея. Жадно выбираем вершину с минимальным текущим расстоянием.

Алгоритм:

1. Инициализируем dist бесконечностями, кроме dist[s] = 0.
2. Используем мин-кучу (distance, vertex).
3. Пока куча не пуста:

• Извлекаем вершину u с минимальным dist.

• Для каждого соседа v с весом w. Если dist[u] + w < dist[v], обновляем dist[v] и добавляем в кучу.

Реализация:

```
import heapq


def dijkstra(graph: list[list[tuple[int, int]]], start: int) -> list[int]:
    # graph - список смежности: для каждой вершины список кортежей (сосед,
вес)
    # start - начальная вершина

    n = len(graph)
    dist = [float('inf')] * n  # инициализируем расстояния бесконечностями
    dist[start] = 0  # расстояние до стартовой вершины = 0
    heap = [(0, start)]  # мин-куча: хранит пары (расстояние, вершина)

    while heap:  # пока куча не пуста
        d, u = heapq.heappop(heap)  # извлекаем вершину с минимальным
расстоянием
        if d != dist[u]:  # если расстояние в куче устарело - пропускаем
            continue
        for v, w in graph[u]:  # перебираем всех соседей u
            new_dist = dist[u] + w  # пробуем дойти до v через u
            if new_dist < dist[v]:  # если нашли более короткий путь
                dist[v] = new_dist  # обновляем расстояние
                heapq.heappush(heap, (new_dist, v))  # добавляем в кучу
    return dist
```

Сложность: $O(m \cdot \log n)$.

4. Алгоритм Форда-Беллмана – произвольные веса

Идея. Релаксация всех рёбер $n - 1$ раз.

Алгоритм:

1. Инициализируем dist бесконечностями, кроме dist[s] = 0.
2. Повторяем $n - 1$ раз:

• Для каждого ребра (u, v, w). Если dist[u] + w < dist[v], обновляем dist[v].

3. Проверяем на отрицательные циклы.

Реализация:

```
def ford_bellman(edges: list[tuple[int, int, int]], n: int, start: int) ->
list[int]:
    # edges - список рёбер: каждый элемент (u, v, w) - из u в v с весом w
    # n - количество вершин
    # start - начальная вершина

    dist = [float('inf')] * n  # инициализируем расстояния бесконечностями
    dist[start] = 0  # расстояние до стартовой вершины = 0

    # Основная часть: релаксация всех рёбер n-1 раз
    for _ in range(n - 1):  # максимальная длина пути без циклов = n-1 рёбер
        for u, v, w in edges:  # перебираем все рёбра
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                # Если можем улучшить расстояние до v через u - улучшаем
                dist[v] = dist[u] + w

    # Проверка на отрицательные циклы
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            # Если после n-1 итерации ещё можно улучшить - есть отрицательный
цикл
            raise ValueError("Граф содержит отрицательный цикл!")

    return dist
```

Сложность: $O(n \cdot m)$.

5. Алгоритм Флойда-Уоршелла – все пары вершин

Идея. Динамическое программирование: $dp[k][i][j]$ – кратчайший путь из i в j через вершины $\{0, 1, \ldots, k\}$.

Алгоритм:

1. Инициализируем dist матрицу смежности.
2. Для каждой вершины k:

• Для каждой пары (i, j):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).

Реализация:

```
def floyd_warshall(graph: list[list[int]]) -> list[list[int]]:
    # graph - матрица смежности n×n, graph[i][j] = вес ребра i→j
    # float('inf') означает, что ребра нет

    n = len(graph)
    dist = [row[:] for row in graph]  # создаём копию матрицы

    # Основной алгоритм: перебираем все промежуточные вершины
    for k in range(n):  # k - промежуточная вершина
        for i in range(n):  # i - начальная вершина
            for j in range(n):  # j - конечная вершина
                # Пробуем пройти i→k→j вместо i→j
```

```
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist  # возвращаем матрицу кратчайших расстояний между всеми
парами
```

Сложность: $O(n^3)$.

6. Алгоритм Джонсона – все пары вершин

Идея:

1. Добавляем фиктивную вершину q с рёбрами веса 0 ко всем вершинам.
2. Запускаем Форда-Беллмана из q для получения потенциалов h.
3. Перевзвешиваем рёбра: $w'(u, v) = w(u, v) + h[u] - h[v]$.
4. Запускаем Дейкстру из каждой вершины.

Реализация:

```
def johnson(graph: list[list[tuple[int, int]]], n: int) -> list[list[int]]:
    # graph - список смежности: для каждой вершины список (сосед, вес)
    # n - количество вершин

    # Шаг 1: собираем все рёбра и добавляем фиктивную вершину
    edges = []
    for u in range(n):
        for v, w in graph[u]:
            edges.append((u, v, w))
    # Добавляем рёбра из фиктивной вершины n ко всем остальным с весом 0
    for v in range(n):
        edges.append((n, v, 0))

    # Шаг 2: запускаем Форда-Беллмана из фиктивной вершины
    # Получаем потенциалы h (расстояния от фиктивной вершины)
    h = ford_bellman(edges, n + 1, n)

    # Шаг 3: перевзвешиваем рёбра для устранения отрицательных весов
    new_graph = [[] for _ in range(n)]
    for u in range(n):
        for v, w in graph[u]:
            new_w = w + h[u] - h[v]  # новое неотрицательное веса
            new_graph[u].append((v, new_w))

    # Шаг 4: запускаем Дейкстру из каждой вершины
    dist = [[float('inf')] * n for _ in range(n)]
    for u in range(n):
        dist[u] = dijkstra(new_graph, u)  # получаем расстояния в
перевзвешенном графе
        # Корректируем расстояния обратно к исходным весам
        for v in range(n):
            if dist[u][v] != float('inf'):
                dist[u][v] -= h[u] - h[v]

    return dist  # матрица кратчайших расстояний между всеми парами
```

Сложность: $O(n \cdot m \cdot \log n)$.

7. Сравнение алгоритмов

Алгоритм	**Сложность**	**Условия**	**Применение**
BFS	$O(n+m)$	Единичные веса	Кратчайшие пути в невзвешенных графах
Дейкстра	$O(m \cdot \log n)$	Неотрицательные веса	Маршрутизация, карты
Форд-Беллман	$O(n \cdot m)$	Произвольные веса	Обнаружение отрицательных циклов
Флойд-Уоршелл	$O(n^3)$	Все пары вершин	Маленькие графы, замыкания
Джонсон	$O(n \cdot m \cdot \log n)$	Все пары вершин	Большие разреженные графы