23. Definition of a graph, ways to store a graph in memory, DFS, BFS

1. Definition of a graph

Графом называется пара (V, E), где V — конечное множество вершин, а $E \subset V \times V$ — множество рёбер.

- Граф называется **неориентированным** (ненаправленным), если:

$$\forall (u, v) \in E: \quad (v, u) \in E.$$

  То есть каждое ребро (u, v) означает связь как от u к v, так и от v к u

2. Ways to store a graph in memory

Пусть у нас есть граф $G = (V, E)$ и $V = \{0, 1, \ldots, n-1\}$.

2.1 Матрица смежности (Adjacency Matrix)

- Матрица смежности — это матрица $n \times n$ (где $n = |V|$).
- Элемент (i, j) в этой матрице равен 1, если $(i, j) \in E$, иначе 0.

Пример (для ненаправленного графа) — матрица симметричная относительно главной диагонали.

> **Память**: $\Theta(V^2)$.
> Если V велико и E мало, матрица смежности может быть неэффективной.

2.2 Список смежности (Adjacency List)

- Список смежности — это массив (или вектор) списков, `std::vector<std::list<size_t>>` размера n.
- В i-ой ячейке хранится список соседей вершины i.

> **Память**: $\Theta(V + E)$.
>
> - Если граф полный (каждая вершина связана со всеми остальными), то список смежности может также достичь $\Theta(V^2)$.
> - Если же граф разреженный, то хранение существенно экономит память.

3. DFS (Depth-first search, поиск в глубину)

3.1 Алгоритм DFS

1. **Помечаем** начальную вершину как посещённую.
2. Для **каждого** непосещённого соседа этой вершины **рекурсивно** вызываем DFS.

Таким образом, мы «уходим» в глубину по одному из путей, пока не закончим его, и только потом рассматриваем другие пути.

3.2 Пример кода (C++)

```cpp
using Graph = std::vector<std::list<size_t>>;

void dfs(const Graph& graph, std::vector<bool>& visited, size_t vertex) {
    visited[vertex] = true;
    std::cout << vertex << '\n'; // do something with the vertex

    for (size_t neighbour : graph[vertex]) {
        if (!visited[neighbour]) {
            dfs(graph, visited, neighbour);
        }
    }
}
```

Чтобы обойти все вершины в графе (если граф несвязный), часто делают так:

```cpp
std::vector<bool> visited(graph.size(), false);

for (size_t i = 0; i < graph.size(); ++i) {
    if (!visited[i]) {
        dfs(graph, visited, i);
    }
}
```

Сложность DFS

- При использовании **списка смежности**: $O(V + E)$.
- При использовании **матрицы смежности**: $O(V^2)$.

4. BFS (Breadth-first search, поиск в ширину)

4.1 Алгоритм BFS

1. Помещаем **начальную вершину** в пустую очередь (FIFO).
2. Пока очередь не пуста:
   a. Извлекаем из очереди вершину v.
   b. Помечаем v как посещённую.
   c. Добавляем все непосещённые вершины-соседи v в очередь.

BFS также называют «**волновым алгоритмом**», так как он «распространяется» слоями от стартовой вершины.

4.2 Пример кода (C++)

```cpp
using Graph = std::vector<std::list<size_t>>;

void bfs(const Graph& graph, std::vector<bool>& visited, size_t initial) {
    std::queue<size_t> q;
    q.push(initial);

    while (!q.empty()) {
        size_t vertex = q.front();
```

```
        q.pop();

        // Если уже посещали - пропускаем
        if (visited[vertex]) {
            continue;
        }

        // Отмечаем как посещённую
        visited[vertex] = true;
        // Делаем что-нибудь с вершиной (например, вывод)
        std::cout << vertex << '\n';

        // Добавляем всех непосещённых соседей
        for (size_t neighbour : graph[vertex]) {
            if (!visited[neighbour]) {
                q.push(neighbour);
            }
        }
    }
}

// Если нужно обойти все компоненты (в случае несвязного графа)
std::vector<bool> visited(graph.size(), false);
for (size_t i = 0; i < graph.size(); ++i) {
    if (!visited[i]) {
        bfs(graph, visited, i);
    }
}
```

Сложность BFS

- При использовании **списка смежности**: $O(V + E)$.
- При использовании **матрицы смежности**: $O(V^2)$.

5. Пример BFS по шагам

Возьмём ненаправленный граф (в виде рисунка):

```
    0
   / \
  1   2
 / \   \
4   5   3
```

Шаг	Извлечённая вершина	Очередь до извлечения	Очередь после добавления соседей	Посещено, добавляем в queue
1	0	[0]	[1, 2]	{} → {0}
2	1	[1, 2]	[2, 4, 5]	{0} → {0, 1}
3	2	[2, 4, 5]	[4, 5, 3]	{0, 1} → {0, 1, 2}
4	4	[4, 5, 3]	[5, 3]	{0, 1, 2} → {0, 1, 2, 4}

Шаг	Извлечённая вершина	Очередь до извлечения	Очередь после добавления соседей	Посещено, добавляем в queue
5	5	[5, 3]	[3]	{0, 1, 2, 4} → {0, 1, 2, 4, 5}
6	3	[3]	[]	{0, 1, 2, 4, 5} → {0, 1, 2, 3, 4, 5}

1. Изначально в очереди только вершина `0`.
2. Извлекаем `0`, отмечаем посещённой. Добавляем в очередь её соседей — `1` и `2`.
3. Извлекаем `1`, помечаем, добавляем её соседей — `4`, `5` (а `0` уже посещена).
4. Извлекаем `2`, помечаем, добавляем её соседа `3`. (Сосед `0` уже посещён.)
5. Дальше берём `4`, потом `5`, потом `3`. В итоге все посещены.

Очередь опустела — BFS завершён, все достижимые из `0` вершины посещены.

6. Итоговая сводка

1. **Граф** (V, E): множество вершин V и рёбер E.
2. **Хранение**:
   - **Матрица смежности**: $\Theta(V^2)$ памяти, удобна для плотных графов и быстрой проверки смежности.
   - **Список смежности**: $\Theta(V + E)$ памяти, удобен для разреженных графов.
3. **DFS** (поиск в глубину):
   - Использует рекурсию или стек (LIFO).
   - Сложность: $O(V + E)$ (со списком), $O(V^2)$ (с матрицей).
4. **BFS** (поиск в ширину):
   - Использует очередь (FIFO).
   - «Волновой» алгоритм, идёт слоями.
   - Сложность: $O(V + E)$ (со списком), $O(V^2)$ (с матрицей).