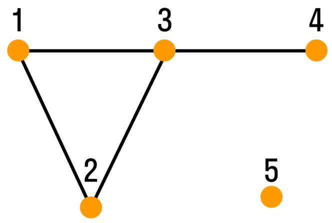
Лекция - структура данных ГРАФЫ.

1. Введение. Основные понятия.

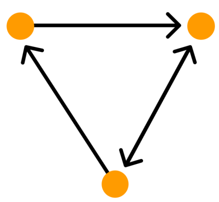
Графы являются одной из основных структур данных в компьютерных науках, используемой для моделирования и решения различных проблем, связанных с сетями, логистикой, маршрутами и многим другим.

- **Граф:**
    - **Неориентированный граф (Undirected Graph)**: Граф, в котором ребра не имеют направления. Например, связь между узлами A и B одинакова как в сторону A-B, так и B-A.

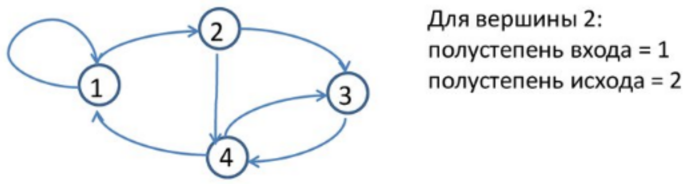


Неориентированный граф

    - **Ориентированный граф (Directed Graph или Digraph)**: Граф, в котором ребра имеют направление. Например, A → B означает, что связь существует от A к B, но не обязательно от B к A.



- **Узлы (Nodes или Vertices)**: Основные элементы графа, представляющие точки или вершины.
- **Ребра (Edges или Arcs)**: Связи между узлами. В неориентированных графах ребра соединяют узлы в обе стороны, в ориентированных — только в одном направлении.
- **Степень узла (Degree of a Node)**:
    - **Входящая степень (In-degree)**: Количество ребер, входящих в узел (для ориентированных графов).
    - **Исходящая степень (Out-degree)**: Количество ребер, исходящих из узла (для ориентированных графов).
    - **Общая степень (Degree)**: Количество ребер, связанных с узлом (для неориентированных графов).

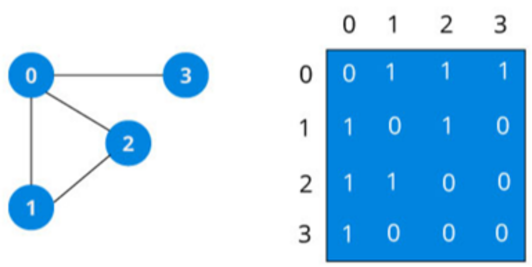
- 

- **Путь (Path):** Последовательность узлов, соединенных ребрами. Путь может быть:
    - **Простой (Simple Path)**: Путь, в котором все узлы различны.
    - **Цикл (Cycle или ПЕТЛЯ)**: Путь, который начинается и заканчивается в одном и том же узле, при этом все промежуточные узлы различны.
- **Связный граф (Connected Graph):** Граф, в котором существует путь между любыми двумя узлами (для неориентированных графов).
    - **Сильно связный граф (Strongly Connected Graph)**: Для ориентированных графов, если существует путь между любыми двумя узлами в обоих направлениях.

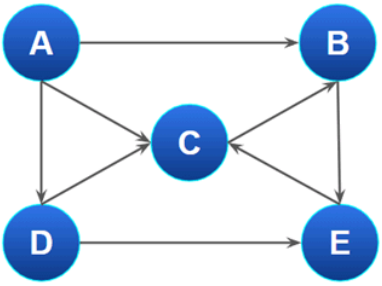
2. Представления графов.

Графы можно представлять несколькими способами, каждый из которых имеет свои преимущества и недостатки:

- **Матрица смежности (Adjacency Matrix):**
    - Двумерный массив, где ячейка (i, j) содержит 1 (или вес ребра) если есть ребро между узлами i и j, и 0 в противном случае.
    - Преимущества: Быстрый доступ для проверки наличия ребра между двумя узлами.
    - Недостатки: Затратно по памяти, особенно для разреженных графов.



- **Список смежности (Adjacency List):**
    - Для каждого узла хранится список соседних узлов.
    - Преимущества: Эффективное использование памяти для разреженных графов.
    - Недостатки: Проверка наличия конкретного ребра может быть медленнее по сравнению с матрицей смежности.



A	B, C, D
B	E
C	B
D	C, E
E	C

Пример:

```
class Graph {
  private:
     int V; // Количество узлов
     std::vector<std::list<int>> adj; // Список смежности
};
```

Пример (узлы с весом):

```
std::vector<std::list<std::pair<int, int>>> adj;
// Список смежности (узел, вес)
```

Пример (создание объекта класса Графа):

```
// Создание графа с 5 узлами
Graph g(5);
```

Пример (конструктор графа):

```
// Конструктор
Graph(int V) {
  this->V = V;
  adj.resize(V);
}
```

Пример (добавление ребра в граф - ориент.):

```
// Добавление ребра в ориентированный граф
void addEdge(int v, int w) {
  adj[v].push_back(w);
}
```

Пример (добавление ребра в граф - неориент.):

```
// Добавление ребра в неориентированный граф
void addEdge(int v, int w) {
  adj[v].push_back(w);
  adj[w].push_back(v);
}
```

Пример (удаление ребра из графа):

```
// Удаление ребра из графа
void removeEdge(int v, int w) {
  adj[v].remove(w);
  adj[w].remove(v);
}
```

Пример (добавление узла):

```
// Добавление узла в граф
void addVertex() {
  adj.push_back(std::list<int>());
  V++;
}
```

Пример (удаление узла):

```
// Удаление узла из графа
void removeVertex(int v) {
  // Удаляем все ребра, связанные с этой вершиной
  for (auto& neighbors : adj) {
     neighbors.remove(v);
  }

  // Удаляем саму вершину
  adj.erase(adj.begin() + v);
  V--;

  // Уменьшаем все вершины больше v на 1
  for (auto& neighbors : adj) {
```

```
    for (auto& neighbor : neighbors) {
      if (neighbor > v) {
        neighbor--;
      }
    }
  }
}
```

Пример (вывод графа в консоли):

```
// Вывод графа
void printGraph() {
  for (int v = 0; v < V; ++v) {
    std::cout << "Вершина " << v << ":";
    for (const auto& x : adj[v])
      std::cout << " -> " << x;
    std::cout << std::endl;
  }
}
```

Метод проверки наличия петли:

- цикл по элементам например списка смежности
- проверка что текущий item = v или нет (да - true, нет - false)

```
bool hasLoop(int v) {


}
```

3. Алгоритмы графов.

- **Поиск в глубину (Depth-First Search, DFS)**:
    - Исследует как можно дальше вдоль каждой ветви перед возвратом назад.
    - Используется для проверки связности графа, поиска циклов, топологической сортировки и т.д.

      Шаги алгоритма DFS

        - **Инициализация**:

        - массив visited размером с количество вершин и установка на все элементы false.
    - **Рекурсивная функция DFS**:
        - Если текущий узел не был посещен:
            - Пометьте его как посещенный.
            - Обработайте (или распечатайте) текущий узел.
            - Для каждого смежного узла, если он не был посещен, рекурсивно вызовите функцию DFS для него.
    - **Запуск DFS**:
        - Вызов рекурсивной функции DFS для каждого узла, если он еще не был посещен (полный обход всех компонент связности графа).

- **Поиск в ширину (Breadth-First Search, BFS)**:
    - Исследует все соседние узлы перед переходом на следующий уровень узлов.
    - Используется для нахождения кратчайшего пути в неориентированных графах, проверки связности и т.д.

    Шаги алгоритма BFS

    - **Инициализация**:
        - Создайте очередь **queue**.
        - Создайте массив **visited** размером с количество вершин и установите все элементы в false.
    - **Запуск BFS**:
        - Пометьте начальный узел как посещенный и добавьте его в очередь.
        - Пока очередь не пуста:
            - Удалите узел из очереди и обработайте (или распечатайте) его.
            - Для каждого смежного узла, если он не был посещен:
                - Пометьте его как посещенный и добавьте в очередь.
    - **Пример:**

```
void BFS(int s) {
  std::vector<bool> visited(V, false);
  std::queue<int> queue;
  visited[s] = true;
  queue.push(s);
  while (!queue.empty()) {
```

```
    s = queue.front();
    std::cout << s << " ";
    queue.pop();
    for (const auto& i : adj[s]) {
      if (!visited[i]) {
        visited[i] = true;
        queue.push(i);
      }
    }
  }
}
```

- **Алгоритм Дейкстры (Dijkstra's Algorithm)**:
  - Находит кратчайшие пути от одного узла до всех остальных в графе с неотрицательными весами ребер.
  - Применяется в задачах маршрутизации и планирования путей.

  Шаги алгоритма Дейкстры

    - **Инициализация**:
      - Создайте массив dist размером с количество вершин и установите все элементы в infinity, кроме начального узла (установите его в 0).
      - Создайте множество (или очередь с приоритетом) priority_queue, содержащее начальный узел с расстоянием 0.
    - **Поиск кратчайших путей**:
      - Пока очередь не пуста:
        - Извлеките узел с минимальным расстоянием из очереди.
        - Для каждого смежного узла обновите расстояния, если найден более короткий путь через текущий узел.
        - Если расстояние обновлено, добавьте (или обновите) узел в очереди с новым расстоянием.

```
// Метод для запуска алгоритма Дейкстры
void dijkstra(int src) {
  std::vector<int> dist(V, INT_MAX);
```

```
  dist[src] = 0;
  std::set<std::pair<int, int>> queue;
  queue.insert(std::make_pair(0, src));
  while (!queue.empty()) {
     int u = queue.begin()->second;
     queue.erase(queue.begin());
     for (const auto& i : adj[u]) {
        int v = i.first;
        int weight = i.second;
        if (dist[u] + weight < dist[v]) {
           if (dist[v] != INT_MAX) {
              queue.erase(queue.find(std::make_pair(dist[v], v)));
           }
           dist[v] = dist[u] + weight;
           queue.insert(std::make_pair(dist[v], v));
        }
     }
  }
  std::cout << "Расстояния от вершины " << src << ":\n";
  for (int i = 0; i < V; ++i) {
     std::cout << i << " \t\t " << dist[i] << "\n";
  }
}
```

- **Алгоритм Флойда-Уоршалла (Floyd-Warshall Algorithm)**:
  - Находит кратчайшие пути между всеми парами узлов.
  - Применяется в задачах, где нужно знать кратчайшие пути между всеми узлами.
- **Алгоритм Крускала (Kruskal's Algorithm)** и **Алгоритм Прима (Prim's Algorithm)**:
  - Используются для нахождения минимального остовного дерева (Minimum Spanning Tree, MST), который соединяет все узлы в графе с минимальной суммой весов ребер.