Структуры данных: графы.

Способы представления графа.

Алгоритмы обхода графа.

Эйлеровы и гамильтоновы пути.

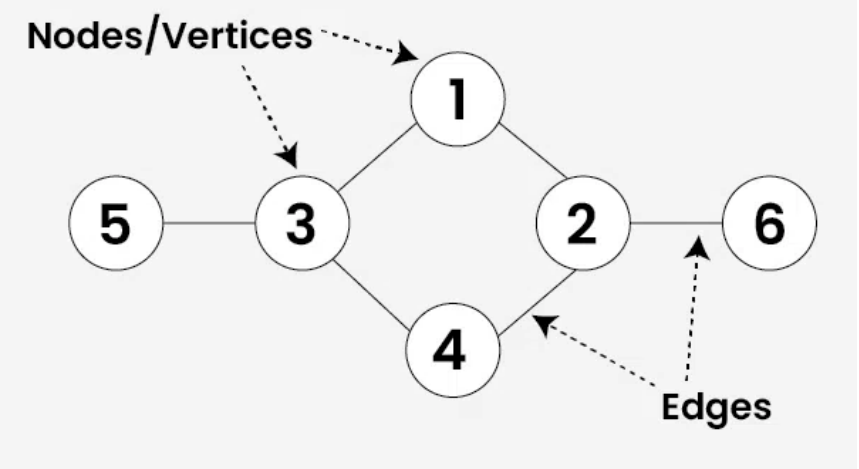
Структуры данных: графы

Граф - это нелинейная структура данных, состоящая из вершин и ребер.

Компоненты структуры данных графа:

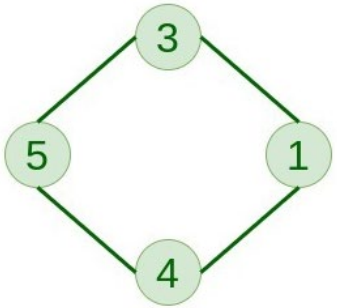
Узлы/Вершины (Nodes/Vertices): являются основными элементами графа. Каждый узел / вершина может быть помечен или не помечен.

Ребра/Дуги (Edges): Ребра рисуются или используются для соединения двух узлов графа. Это может быть упорядоченная пара вершин в ориентированном графе. Ребра могут соединять любые две вершины любым возможным способом. Правил соединения нет. Каждое ребро может быть помечено /не помечено.

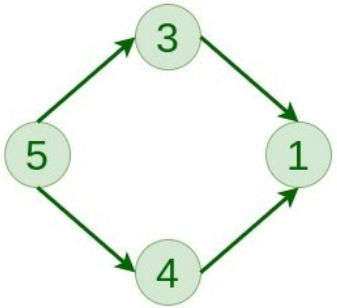


Более формально граф состоит из множества вершин (V) и множества ребер (E). Граф обозначается через G(V, E). Эта сеть связей - именно то, что представляет собой графическая структура данных.

Типы графов в структурах данных



Undirected Graph



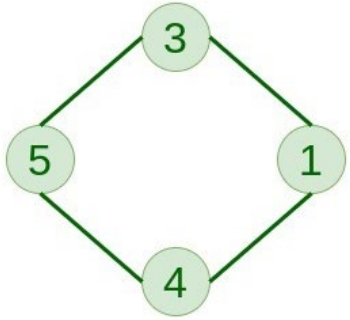
Directed Graph

Неориентированный граф (Undirected Graph)
Граф, в котором ребра не имеют никакого направления. Узлы являются неупорядоченными парами в определении каждого ребра.

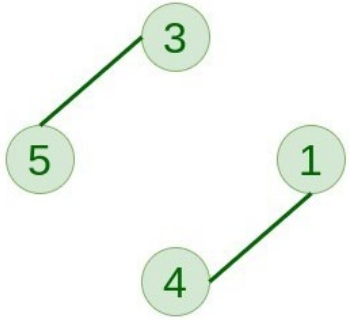
Ориентированный граф (Directed Graph)
Граф, в котором ребро имеет направление. Узлы являются упорядоченными парами в определении каждого ребра.

Связный граф (Connected Graph)
Граф, в котором из одного узла можно посетить любой другой узел.

Несвязанный граф (Disconnected Graph)
Граф, в котором хотя бы один узел недостижим из другого узла.



Connected Graph



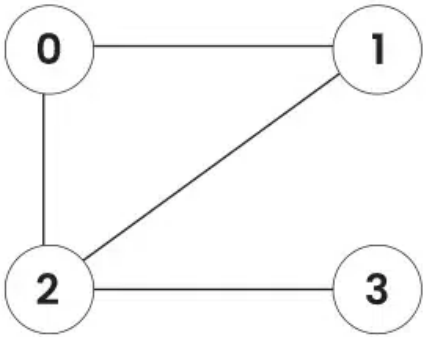
Disconnected Graph

Способы представления графа

Наиболее распространённые представления структуры данных графа:

Матрица смежности:

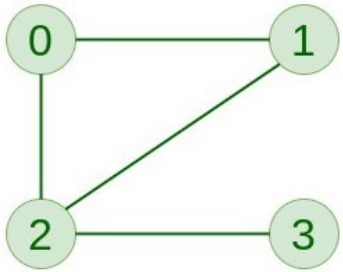
Граф хранится в виде двумерной матрицы, где строки и столбцы обозначают вершины. Каждый элемент матрицы представляет собой вес ребра между этими вершинами.

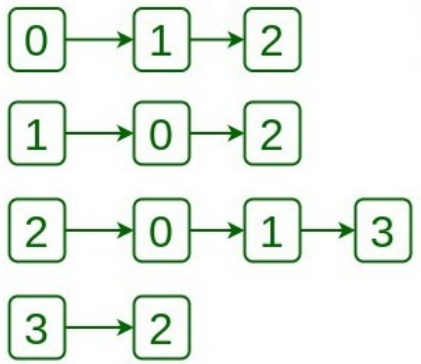


	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Список смежности:

Этот граф представлен в виде набора связанных списков. Имеется массив указателей, указывающих на рёбра, соединённые с данной вершиной.

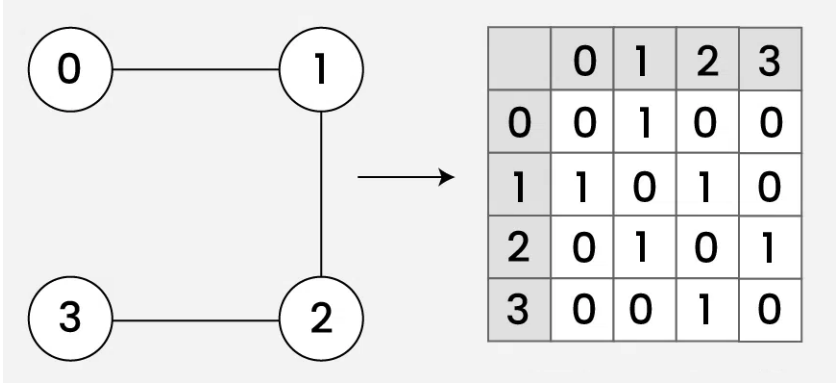




Матрица смежности

Матрица смежности — это квадратная матрица, используемая для представления конечного графа путём хранения связей между узлами в соответствующих ячейках. Элементы матрицы указывают, являются ли пары вершин смежными в графе. Для графа с V вершинами матрица смежности A представляет собой матрицу V×V.

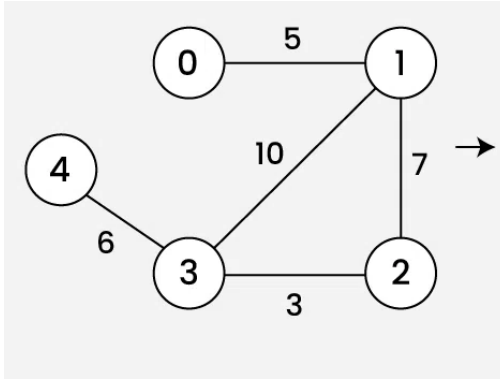
1. Неориентированный и невзвешенный граф



	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

A[i][j] = 1, между вершинами i и j есть ребро.
A[i][j] = 0, между вершинами i и j нет ребра.

2. Неориентированный и взвешенный граф

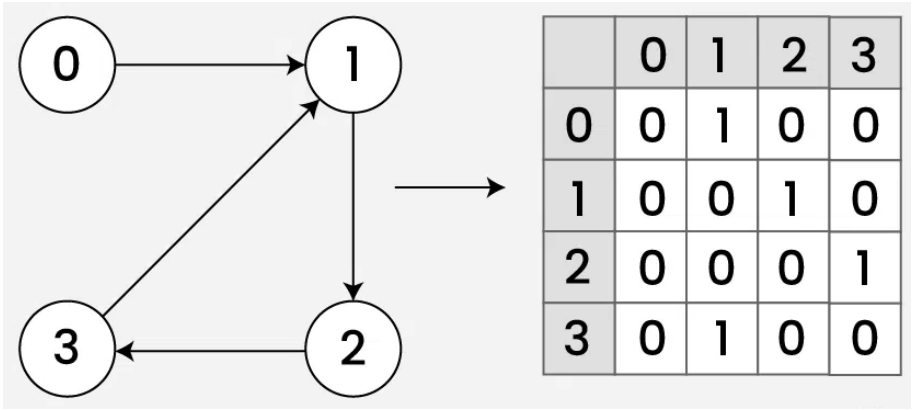


	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

A[i][j] = INF, между вершинами i и j нет ребра.
A[i][j] = w, между вершинами i и j ребро с весом = w.

Матрица смежности

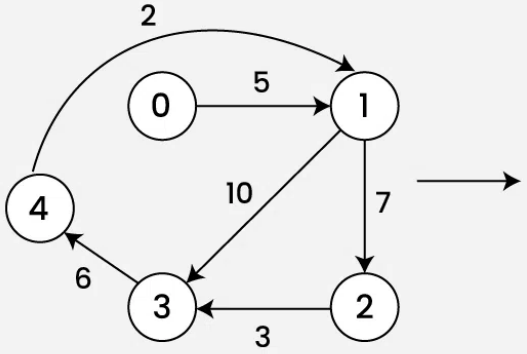
3. Ориентированный и невзвешенный граф



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

A[i][j] = 1, есть ребро из вершины i в вершину j.
A[i][j] = 0, нет ребра из вершины i в вершину j.

4. Ориентированный и взвешенный граф



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	INF	INF	7	10	INF
2	INF	INF	INF	3	INF
3	INF	INF	INF	INF	6
4	INF	2	INF	INF	INF

A[i][j] = INF, нет ребра из вершины i в j
A[i][j] = w, есть ребро из вершины i в j с весом = w.

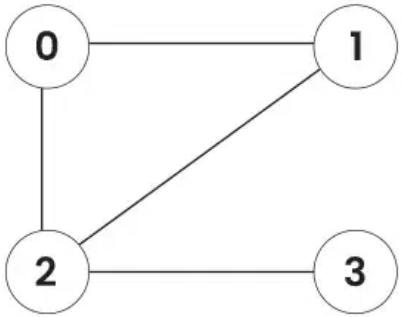
Свойства матрицы смежности:

- **Диагональные элементы** A[i][j] обычно равны 0 (для невзвешенной матрицы) и INF в случае взвешенной матрицы, при условии отсутствия петель.
- **Для неориентированных графов** матрица смежности симметрична. Это означает, что A[i][j] = A[j][i] для всех i и j.

Матрица смежности

```cpp
void addEdge(vector<vector<int>>& mat, int i, int j)
{
    mat[i][j] = 1;
    mat[j][i] = 1; // для неориентированного графа
}

void displayMatrix(vector<vector<int>>& mat)
{
    int V = mat.size();
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            cout << mat[i][j] << " ";
        cout << endl;
    }
}
```

```cpp
int main()
{
    setlocale(LC_ALL, "ru");
    int V = 4; // Создаём граф с 4 вершинами
    // Все значения инициализируем как 0
    vector<vector<int>> mat(V, vector<int>(V, 0));

    // Добавляем ребра.
    addEdge(mat, 0, 1); addEdge(mat, 0, 2);
    addEdge(mat, 1, 2); addEdge(mat, 2, 3);

    cout << "Представление матрицы смежности" << endl;
    displayMatrix(mat);

    return 0;
}
```



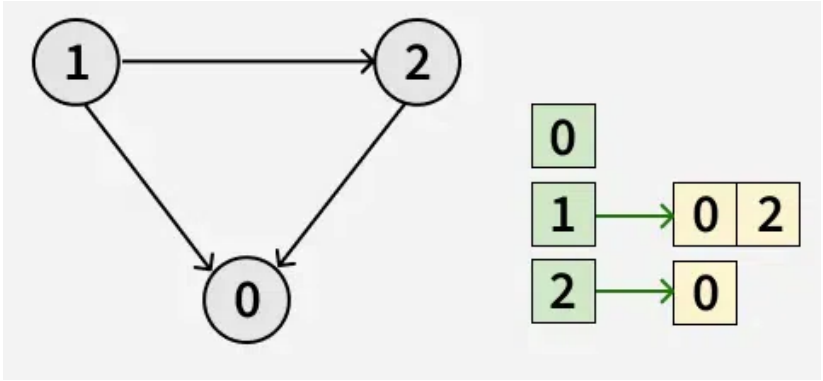
	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

```
Консоль отладки Microsoft Visual Studio
Представление матрицы смежности
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

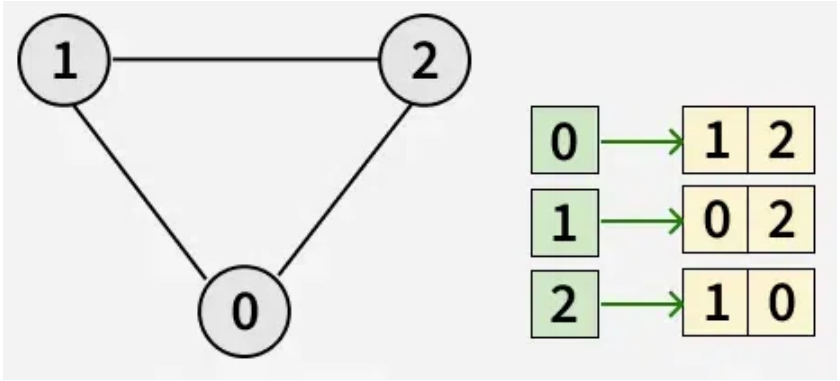
Список смежности

Список смежности — структура данных, в которой каждый узел графа хранит список своих соседних вершин.

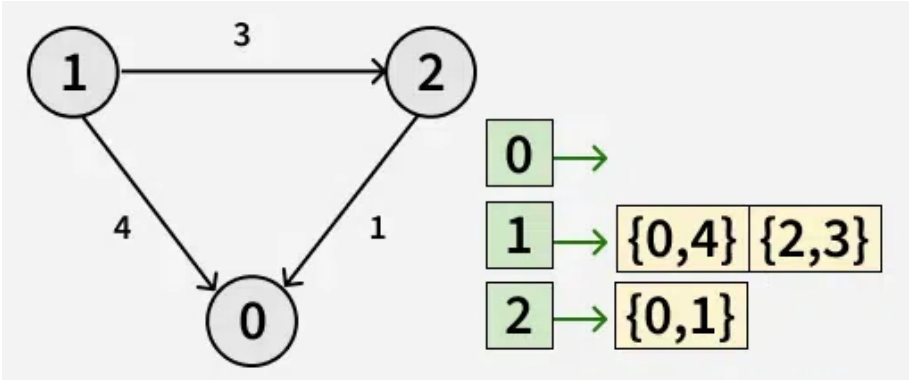
1. Ориентированный и невзвешенный граф



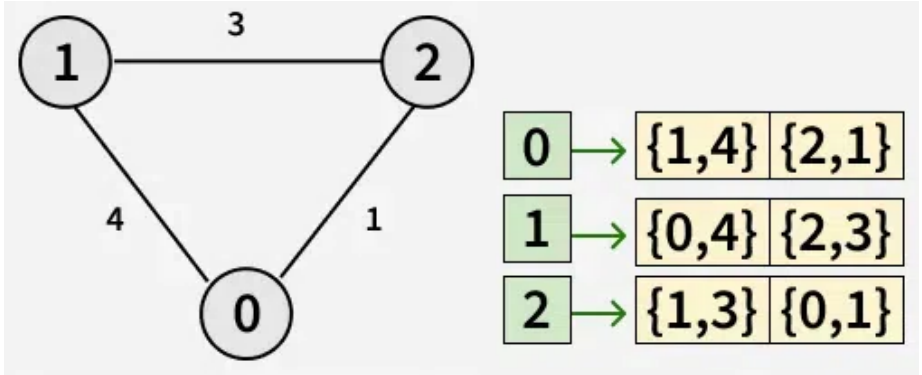
2. Неориентированный и невзвешенный граф



3. Ориентированный и взвешенный граф

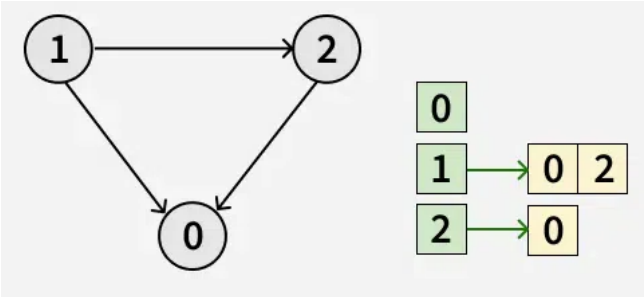


4. Неориентированный и взвешенный граф



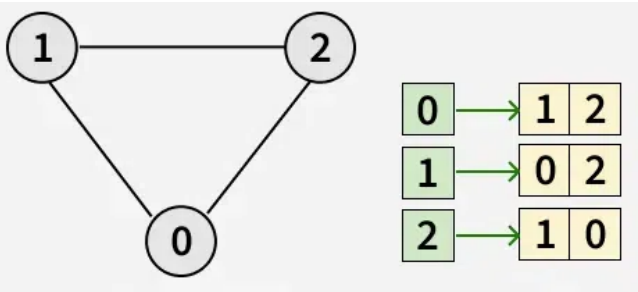
Список смежности

```cpp
//Для невзвешенного
void addEdge(vector<vector<int>>& adj, int u, int v)
{
    adj[u].push_back(v); //Для ориентированного графа
    adj[v].push_back(u); //Для неориентированного графа (оба p_b(v/u))
}

void displayAdjList(const vector<vector<int>>& adj) {
    for (int i = 0; i < adj.size(); i++) {
        cout << i << ": ";
        for (int j : adj[i]) {
            cout << j << " ";
        }
        cout << endl;
    }
}
```




```
Консоль отладки Microsoft Visual Studio
Представление списка смежности для ориентированного графа :
0:
1: 0 2
2: 0
```

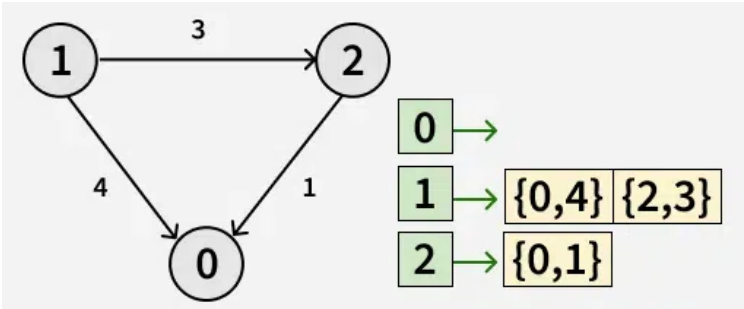
```cpp
int main() {

    setlocale(LC_ALL, "ru");
    int V = 3; // Создаём граф с 3 вершинами и 3 ребрами
    vector<vector<int>> adj(V);

    // Добавляем ребра.
    addEdge(adj, 1, 0);
    addEdge(adj, 1, 2);
    addEdge(adj, 2, 0);

    cout << "Представление списка смежности "
         << "для неориентированного графа: " << endl;
    displayAdjList(adj);

    return 0;
}
```



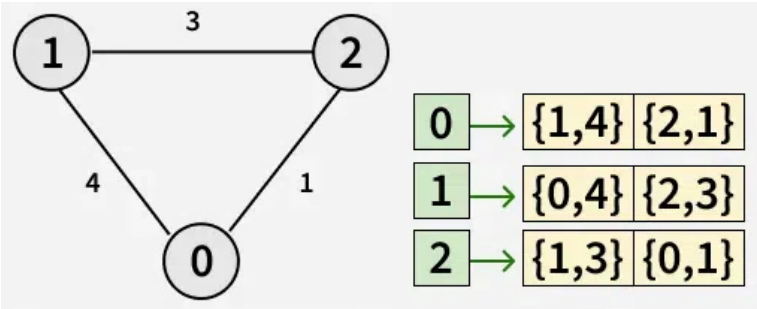

```
Консоль отладки Microsoft Visual Studio
Представление списка смежности для неориентированного графа :
0: 1 2
1: 0 2
2: 1 0
```

Список смежности

```cpp
//Для взвешенного
void addEdge(vector<vector<pair<int, int>>>& adj, int u, int v, int w) {
    adj[u].push_back({ v,w }); //Для ориентированного графа
    adj[v].push_back({ u,w }); //Для неориентированного графа (оба p_b(v/u)
}

void displayAdjList(vector<vector<pair<int, int>>>& adj) {
    for (int i = 0; i < adj.size(); i++) {
        cout << i << ": ";
        for (auto& j : adj[i]) {
            cout << "{" << j.first << ", " << j.second << "} ";
        }
        cout << endl;
    }
}
```




```cpp
int main() {
    setlocale(LC_ALL, "ru");
    int V = 3;
    vector<vector<pair<int, int>>> adj(V);

    addEdge(adj, 1, 0, 4);
    addEdge(adj, 1, 2, 3);
    addEdge(adj, 2, 0, 1);

    cout << "Представление списка смежности "
         << "для ориентированного графа : " << endl;
    displayAdjList(adj);

    return 0;
}
```



Консоль отладки Microsoft Visual Studio

```
Представление списка смежности для ориентированного графа :
0:
1: {0, 4} {2, 3}
2: {0, 1}
```

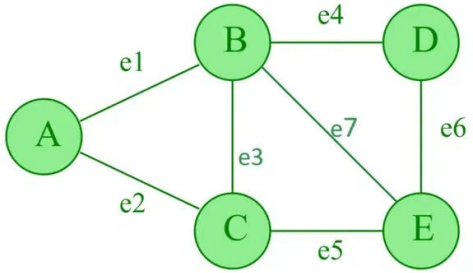
Консоль отладки Microsoft Visual Studio

```
Представление списка смежности для неориентированного графа :
0: {1, 4} {2, 1}
1: {0, 4} {2, 3}
2: {1, 3} {0, 1}
```

Операции	Матрица смежности	Список смежности
Место для хранения	В этом представлении используется матрица VxV, поэтому в худшем случае потребуется **O(|V|^2)**.	Для каждой вершины мы храним информацию о её соседях. В худшем случае, для хранения информации о вершине требуется O(V), а для хранения информации о соседях, требуется O(E). Пространственная сложность **O(|V|+|E|)**.
Добавление вершины	Необходимо увеличить объем памяти до (|V|+1)^2. Для этого нужно скопировать всю матрицу. Сложность составляет **O(|V|^2)**.	В списке смежности есть два указателя: первый указывает на первый узел, а второй — на последний. Добавление вершины может быть выполнено за время **O(1)**.
Добавление ребра	Чтобы добавить ребро из i в j, matrix[i][j] = 1, что требует **O(1)** времени.	Аналогично добавлению вершины, здесь также используются два указателя, указывающие на начало и конец списка. **O(1)**.
Удаление вершины	Чтобы удалить вершину из матрицы V*V, необходимо уменьшить объём памяти с (|V|+1)^2 до |V|^2. Для этого необходимо скопировать всю матрицу. Следовательно, сложность составляет **O(|V|^2)**.	Необходимо выполнить поиск вершины, что в худшем случае займёт O(|V|) времени, после чего необходимо обойти рёбра, что в худшем случае займёт O(|E|) времени. **O(|V|+|E|)**.
Удаление ребра	Чтобы удалить ребро, из i в j, matrix[i][j] = 0, что требует **O(1)** времени.	Для удаления ребра требуется обход всех ребер, а в худшем случае — обход всех ребер. Временная сложность **O(|E|)**.
Запрос	Чтобы найти существующее ребро, необходимо проверить содержимое матрицы. Для двух вершин, скажем, i и j, проверка matrix[i][j] может быть выполнена за время **O(1)**.	Чтобы проверить наличие ребра, необходимо проверить вершины, смежные с заданной. Вершина может иметь не более O(|V|) соседей, и в худшем случае придётся проверить каждую смежную вершину. Следовательно, временная сложность **O(|V|)**.

Матрица инцидентности и список рёбер

Матрица инцидентности — это двумерный массив, в котором строки представляют вершины, а столбцы — рёбра. Каждый элемент матрицы указывает, инцидентна ли вершина ребру. В отличие от матрицы смежности, которая показывает взаимосвязи между вершинами графа (есть ли ребро между двумя вершинами), матрица инцидентности отображает взаимосвязь между вершинами и рёбрами графа (т.е. какие вершины каким рёбром соединены).



Узел/Ребра	e1	e2	e3	e4	e5	e6	e7
A	1	1	0	0	0	0	0
B	1	0	1	1	0	0	1
C	0	1	1	0	1	0	0
D	0	0	0	1	0	1	0
E	0	0	0	0	1	1	1

$$B_{ij} = \begin{cases} -1 & \text{if edge } e_j \text{ leaves vertex } v_i, \\ 1 & \text{if edge } e_j \text{ enters vertex } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

или наоборот:
−1, если связь «входит» в вершину,
1, в случае, если связь «выходит» из вершины,
0, если связь является петлёй или не инцидентна

$\pm$w – вес, для взвешенных графов (вместо 1)

Размер занимаемой памяти: O(V*E).

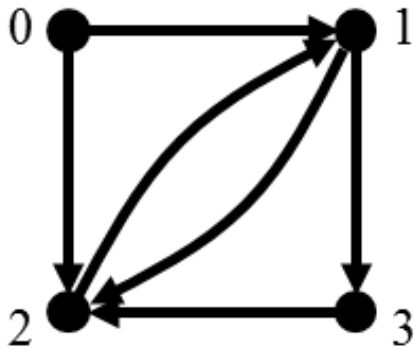
Матрица инцидентности и список рёбер

Список рёбер — это просто список (или массив) рёбер, где каждое ребро представлено парой вершин. Каждое ребро обычно представлено в виде пары **(u, v)** для неориентированного графа, где **u** и **v** — вершины, соединённые ребром.
В ориентированном графе каждое ребро представлено ориентированной парой **(u, v)**, где **u** — начальная вершина, а **v** — конечная вершина.

Размер занимаемой памяти: O(E).

Это наиболее компактный способ представления графов, поэтому часто применяется для внешнего хранения или обмена данными.

Граф
N = 4, M = 6



Список рёбер

(0; 1)
(0; 2)
(1; 2)
(1; 3)
(2; 1)
(3; 2)

Поиск в ширину или BFS (Breadth First Search) для графа

Поиск в ширину (BFS) — это фундаментальный алгоритм обхода графа. Он начинается с узла, затем обходит все его соседние узлы. После посещения всех соседних узлов обходят и его соседние узлы.

Алгоритм начинается с заданного источника и исследует все достижимые вершины из заданного источника. Он аналогичен обходу дерева в ширину. Как и в случае дерева, начинаем с заданного источника (в дереве — с корня) и обходим вершины уровень за уровнем, используя структуру данных «очередь». Единственная загвоздка заключается в том, что, в отличие от деревьев, графы могут содержать циклы. Чтобы избежать повторной обработки узла, используется массив посещённых вершин.

1. **Инициализация:** Поместить заданную исходную вершину в очередь и отметить её как посещённую.
2. **Исследование:** Пока очередь не пуста:
   - Исключить узел из очереди и посетить его (например, вывести его значение).

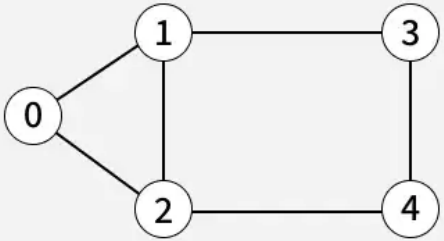
2.5. Для каждого непосещённого соседа исключённого из очереди узла:

- Поместить соседа в очередь.
- Отметить соседа как посещённого.

3. **Завершение**: Повторять шаг 2, пока очередь не опустеет.

Временная сложность: O(V + E), где V и E - количество вершин и ребер в графе.

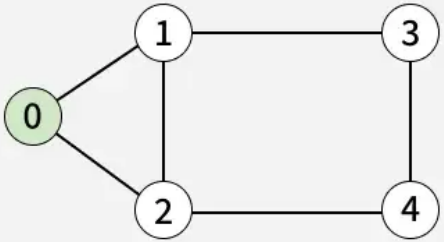
Пространственная сложность: O(V).



visited[]=

res[]=

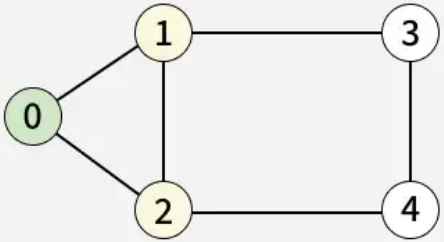
queue:



visited[]= T

res[]= 0

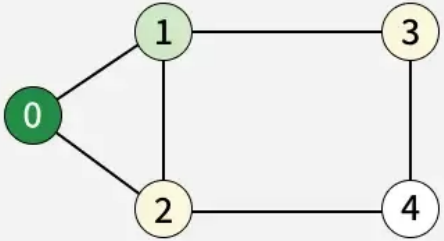
queue: 0



visited[]= T T T

res[]= 0

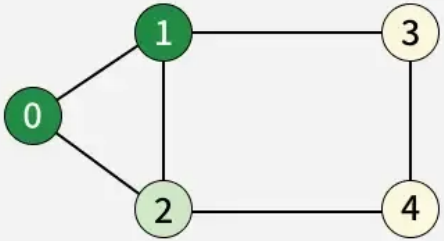
queue: 1 2



visited[]= T T T T

res[]= 0 1

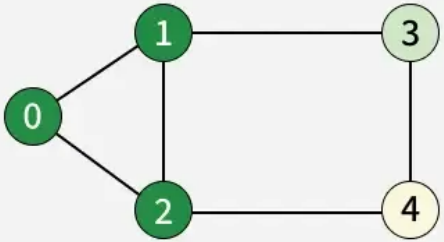
queue: 2 3



visited[]= T T T T T

res[]= 0 1 2

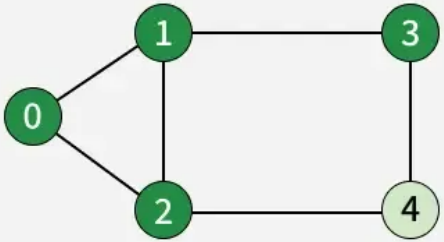
queue: 3 4



visited[]= T T T T T

res[]= 0 1 2 3

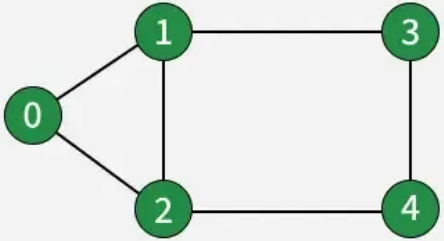
queue: 4



visited[]= T T T T T

res[]= 0 1 2 3 4

queue:



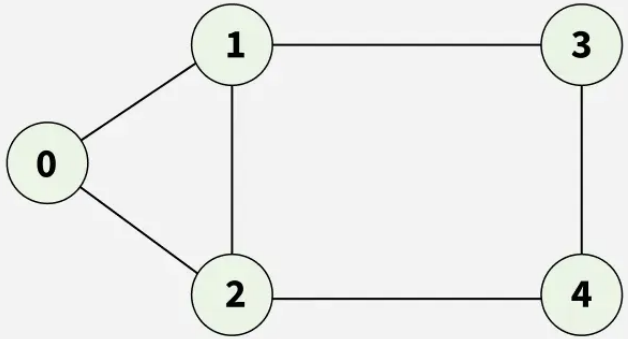
visited[]= T T T T T

res[]= 0 1 2 3 4

queue:

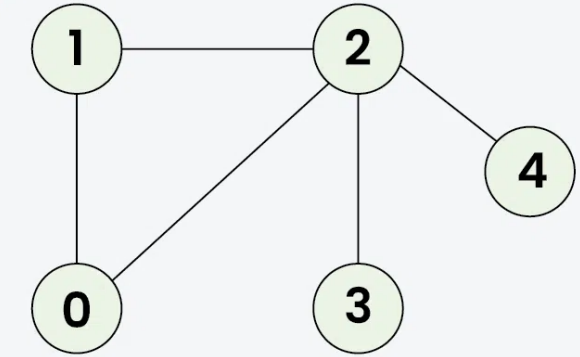
Поиск в ширину или BFS (Breadth First Search) для графа

Вход: *adj[][] = [[1,2], [0,2,3], [0,1,4], [1,4], [2,3]]*




```
Консоль отладки M
0: 1 2
1: 0 2 3
2: 0 1 4
3: 1 4
4: 2 3

BFS: 0 1 2 3 4
```

Вход: *adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]*

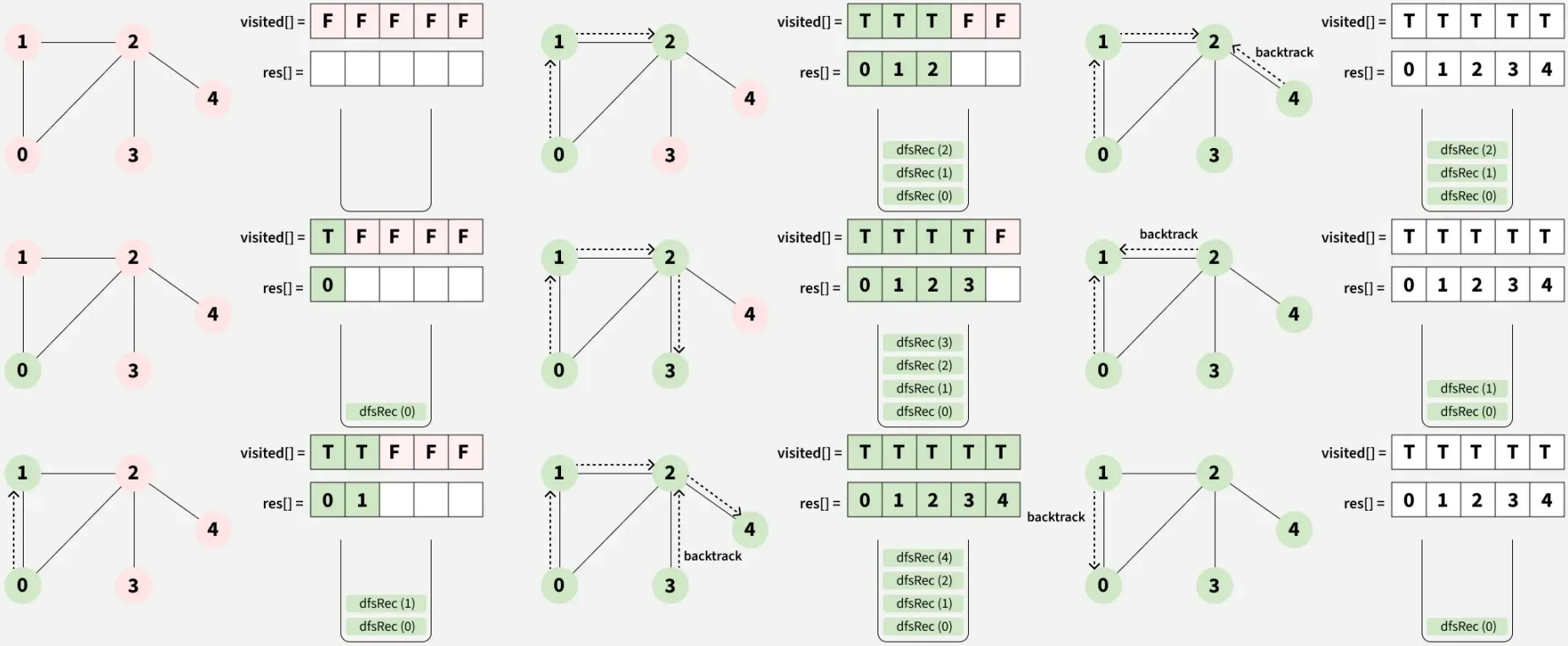



```
Консоль отладки Mic
0: 1 2
1: 0 2
2: 0 1 3 4
3: 2
4: 2

BFS: 0 1 2 3 4
```

```cpp
vector<int> bfs(vector<vector<int>>& adj) {
    int V = adj.size();
    int s = 0; // Исходный узел
    vector<int> res; // Создаём массив для хранения результатов обхода
    queue<int> q; // Создаём очередь для BFS
    vector<bool> visited(V, false); // Помечаем все вершины как не посещенные
    visited[s] = true; // 1.Исходный узел как посещенный и ставим его в очередь
    q.push(s);

    while (!q.empty()) { // Выполняем итерации по очереди
        // 2.Извлекаем вершину из очереди и сохраняем ее
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // Получаем все смежные вершины, удаленные из очереди
        // 2.5.Просматриваем вершины, если соседняя не была
        // посещена, отмечаем ее посещенной и ставим в очередь
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
    return res;
}
int main() {
    //vector<vector<int>> adj = { {1,2}, {0,2,3}, {0,1,4}, {1,4}, {2,3} };
    vector<vector<int>> adj = { {1,2}, {0,2}, {0,1,3,4}, {2}, {2} };
    displayAdjList(adj);
    vector<int> ans = bfs(adj); cout << "BFS: ";
    for (auto i : ans) {
        cout << i << " ";
    }
    return 0;
}
```

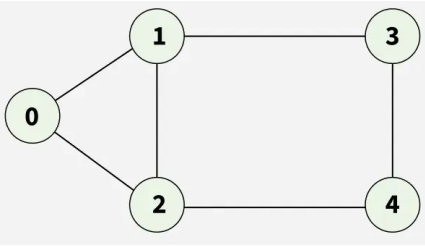
Поиск в глубину или DFS (Depth First Search) для графа

Алгоритм начинается с заданного источника и исследует все достижимые вершины из него. Он похож на прямой обход (NLR) дерева, где мы посещаем корень, а затем рекурсивно обрабатываем его дочерние вершины. **Временная сложность:** O(V + E). **Пространственная сложность:** O(V + E).



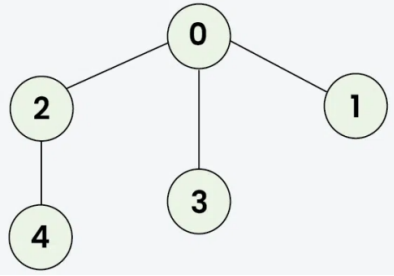
Поиск в глубину или DFS (Depth First Search) для графа

Вход: *[[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]*




```
Консоль отладки Microsoft Visual Studio
0: 1 2
1: 0 2
2: 0 1 3 4
3: 2
4: 2

DFS_Stack: 0 1 2 3 4
```

Вход: *[[2,3,1], [0], [0,4], [0], [2]]*

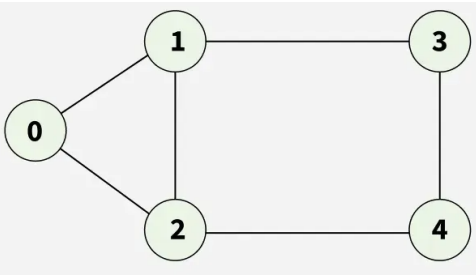



```
Консоль отладки Microsoft Visual Studio
0: 2 3 1
1: 0
2: 0 4
3: 0
4: 2

DFS_Stack: 0 2 4 3 1
```

```cpp
vector<int> DFS_Stack(vector<vector<int>>& adj)
{
    int V = adj.size();
    vector<bool> visited(V, false);
    vector<int> res;
    stack<int> st;
    // Начинаем с вершины 0
    st.push(0);
    visited[0] = true;

    while (!st.empty()) {
        int current = st.top();
        st.pop();
        res.push_back(current);

        // Добавляем соседей в стек в обратном порядке
        // для сохранения порядка обхода как в рекурсивной версии
        for (auto it = adj[current].rbegin(); it != adj[current].rend(); ++it) {
            int neighbor = *it;
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                st.push(neighbor);
            }
        }
    }

    return res;
}
```

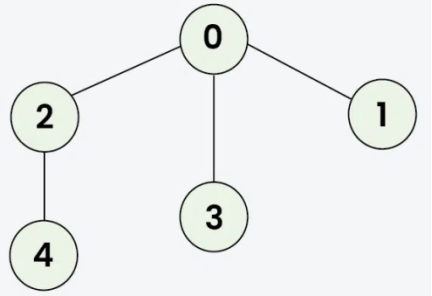
Поиск в глубину или DFS (Depth First Search) для графа

Вход: *[[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]*




```
Консоль отладки Microsoft Vis
0: 1 2
1: 0 2
2: 0 1 3 4
3: 2
4: 2

DFS: 0 1 2 3 4
```

Вход: *[[2,3,1], [0], [0,4], [0], [2]]*



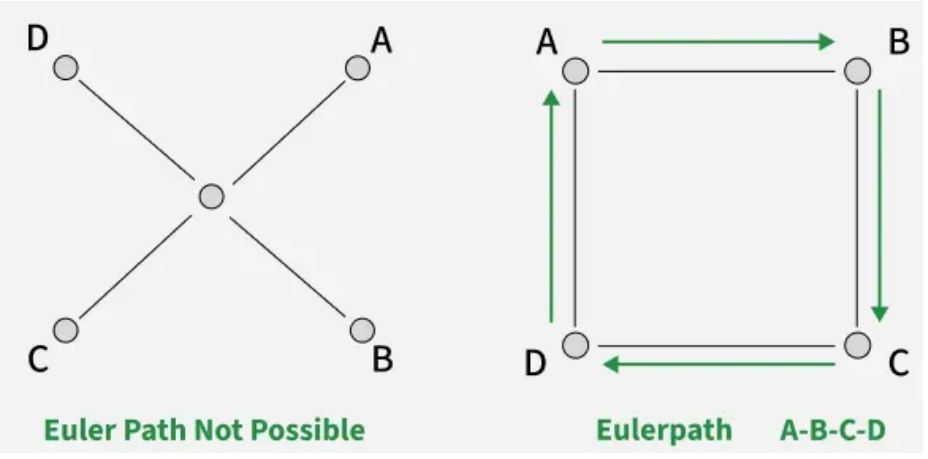

```
Консоль отладки Microsoft Visual
0: 2 3 1
1: 0
2: 0 4
3: 0
4: 2

DFS: 0 2 4 3 1
```

```cpp
// Рекурсивная функция для обхода DFS
void dfsRec(vector<vector<int>>& adj, vector<bool>& visited, int s, vector<int>& res)
{
    visited[s] = true;
    res.push_back(s);
    // Рекурсивно посещает все соседние вершины которые еще не посещены
    for (int i : adj[s])
        if (visited[i] == false)
            dfsRec(adj, visited, i, res);
}

// Основная функция DFS, которая инициализирует посещенный массив и вызывает DFSRec
vector<int> DFS(vector<vector<int>>& adj)
{
    vector<bool> visited(adj.size(), false);
    vector<int> res;
    dfsRec(adj, visited, 0, res);
    return res;
}
int main()
{
    int V = 5;
    //vector<vector<int>> adj = { {1,2}, {0,2}, {0,1,3,4}, {2}, {2} };
    vector<vector<int>> adj = { {2, 3, 1}, {0}, {0,4}, {0}, {2} };
    vector<int> res = DFS(adj);
    for (int i = 0; i < V; i++)
        cout << res[i] << " ";
}
```

----------	**BFS**	**DFS**
Структура данных	Использует «**очередь**» для поиска кратчайшего пути.	Использует «**стек**».
Определение	BFS — это метод обхода, при котором сначала **просматриваем все узлы на одном уровне**, а затем **переходим на следующий уровень**.	DFS — обход **начинается с корневого узла** и продолжается до тех пор, пока не дойдём до узла, рядом с которым нет непосещённых узлов.
Используемый подход	Он работает по принципу **FIFO (первым пришёл — первым ушёл)**.	Он работает по принципу **LIFO (последним пришёл — первым ушёл)**.
Подходит для	Поиска вершин, расположенных ближе к заданному источнику.	Для случаев, когда решения находятся вдали от источника.
Приложения	BFS используется в различных приложениях, таких как двудольные графы, поиск кратчайших. Если вес каждого ребра одинаков, то BFS находит кратчайший путь от источника до любой другой вершины.	DFS используется в различных приложениях, таких как построение ациклических графов, поиск сильно связанных компонент и т. д.

Пути и контуры (циклы) Эйлера

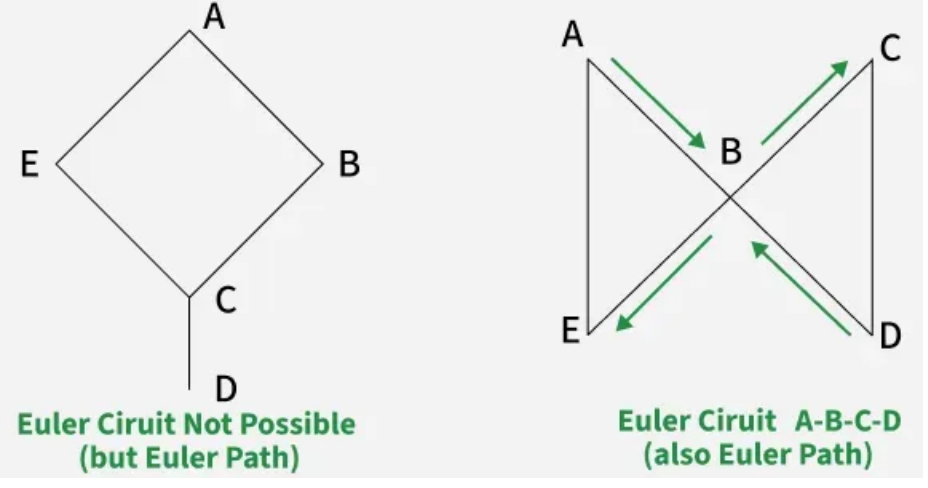
Эйлеров путь — это путь, который проходит **по каждому ребру ровно один раз**. Если путь начинается и заканчивается в одной и той же вершине, то он называется **эйлеровым контуром или циклом**.



Условия для эйлеровых путей и циклов

Эйлеров путь: Связный граф имеет эйлеров путь тогда и только тогда, когда в нём ровно ноль или две вершины нечётной степени.

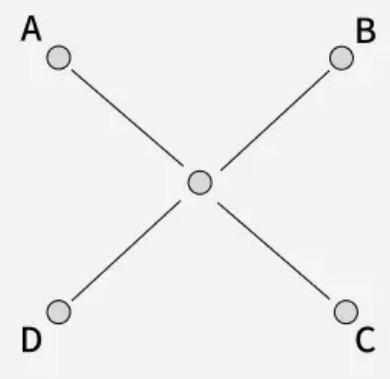
Эйлеров цикл: Связный граф имеет эйлеров цикл тогда и только тогда, когда каждая вершина имеет чётную степень.



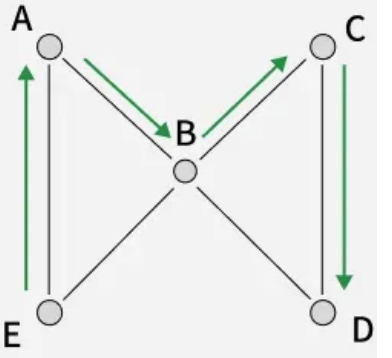
Гамильтоновы пути и контуры (циклы)

Гамильтонов путь — это путь, который проходит **через каждую вершину ровно один раз**. Если путь начинается и заканчивается в одной и той же вершине, то он называется **гамильтоновым циклом или контуром**.

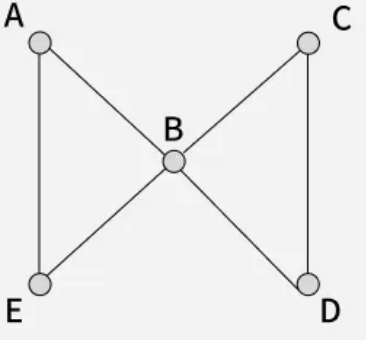
В отличие от эйлеровых путей и циклов, не существует простых необходимых и достаточных критериев для определения наличия гамильтоновых путей или циклов в графе. Однако существуют определённые критерии, исключающие существование гамильтонова цикла в графе, например, если в графе есть вершина степени один, то гамильтонов цикл в нём невозможен.



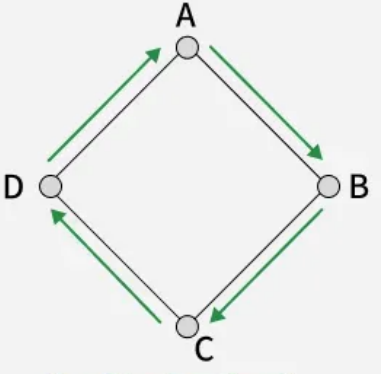
Hamiltonian Path
Not Possible



Hamiltonian path
E-A-B-C-D

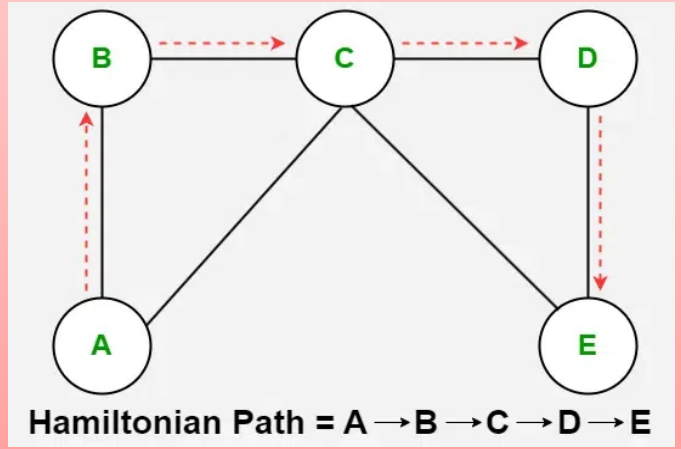
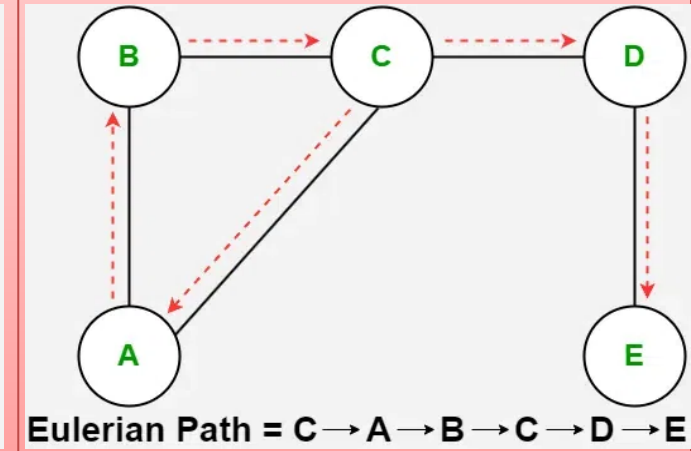


Hamiltonian Ciruit
Not Possible



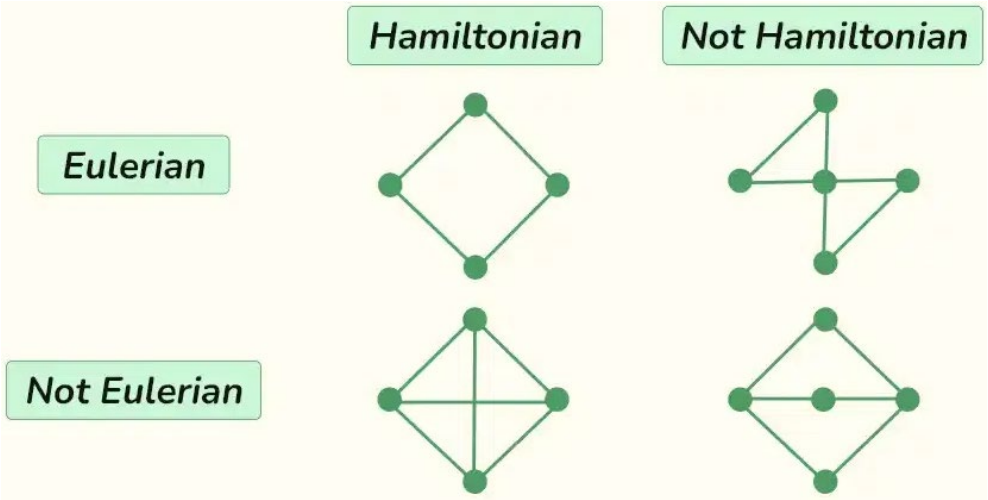
Hamiltonian Circuit
A-B-C-D-A

Разница между гамильтоновым путём и эйлеровым путём

Характеристики	Гамильтонов путь	Эйлеров Путь
Определение	Посещает каждую вершину ровно один раз	Посещает каждое ребро ровно один раз
Условие существования	Отсутствие простой характеристики	Существует, если ровно 0 или 2 вершины имеют нечётную степень
Гамильтонов цикл	Начинается и заканчивается в одной и той же вершине	Начинается и заканчивается в одной и той же вершине
Сложность проблемы	NP-полная задача	Полиномиальное время
Примеры Типов графиков	Задача коммивояжёра (TSP) и поиск пути	Кёнигсбергские мосты и почтовые маршруты
Пример		

Гамильтонов граф. Эйлеров граф

Гамильтонов граф содержит гамильтонов цикл, где каждая вершина посещается ровно один раз, прежде чем вернуться в исходную точку.
Эйлеров граф содержит эйлеров цикл, который проходит по каждому ребру ровно один раз, прежде чем вернуться в исходную точку.



<u>Применение Эйлеровых и гамильтоновых путей:</u>

-**Проектирование сетей**: гамильтоновы циклы оптимизируют маршрутизацию и минимизируют сетевые затраты (например, оптоволоконные сети).
-**Проектирование схем**: эйлеровы пути сокращают количество трассировок на печатных платах (например, эффективная топология схем).
-**Робототехника:** эйлеровы пути планируют маршруты для покрытия областей без повторных проходов.

Задача 1. В классе 30 человек. Может ли быть так, что 9 из них имеют по 3 друга, 11 - по 4 друга, а 10 - по 5 друзей?

Пусть каждый ученик класса будет вершиной, а взаимоотношения между учениками — ребром, степень вершины (число смежных вершин) соответствует количеству друзей у данного ученика.

Тогда:

- $n_3 = 9$ — число учеников, имеющих по 3 друга,
- $n_4 = 11$ — число учеников, имеющих по 4 друга,
- $n_5 = 10$ — число учеников, имеющих по 5 друзей.

Общее число учеников в классе $n = n_3 + n_4 + n_5 = 9 + 11 + 10 = 30.$

Каждое ребро соответствует дружбе между двумя учениками. Сумма степеней всех вершин графа равна удвоенному числу рёбер, так как каждое ребро увеличивает степень двух вершин на 1. Таким образом, сумма степеней всех вершин равна: $3n_3 + 4n_4 + 5n_5$ = 27 + 44 + 50 = 121

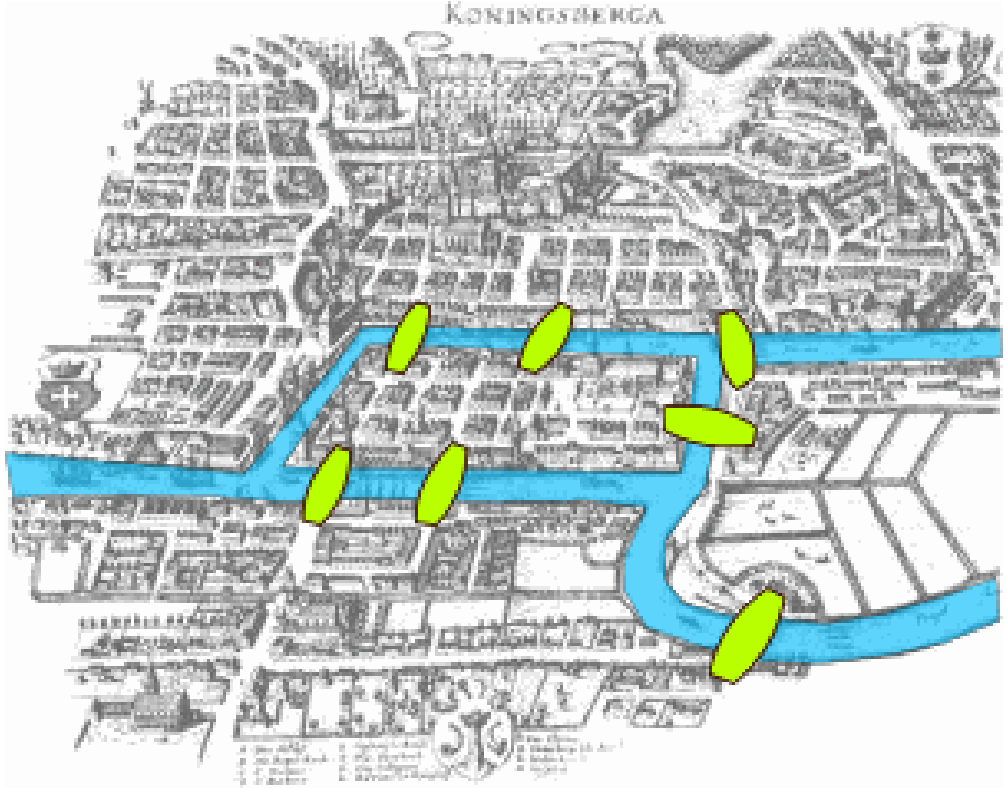
Так как сумма степеней всех вершин равна удвоенному числу рёбер, то:

$$2E = 121 \implies E = \frac{121}{2}.$$

Однако число рёбер E должно быть целым числом, а $\frac{121}{2} = 60.5$ — нецелое число. Это противоречит условию задачи, так как число рёбер графа не может быть дробным, такая ситуация невозможна. **Сумма степеней вершин в любом графе чётна (равна удвоенному числу рёбер).**

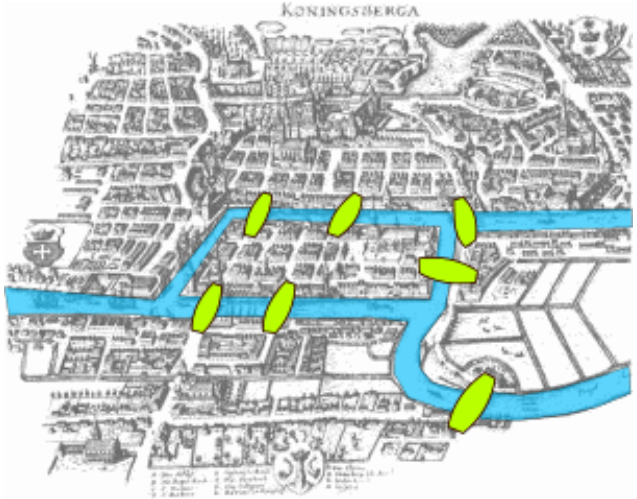
Задача 2. Задача о кёнигсбергских мостах.

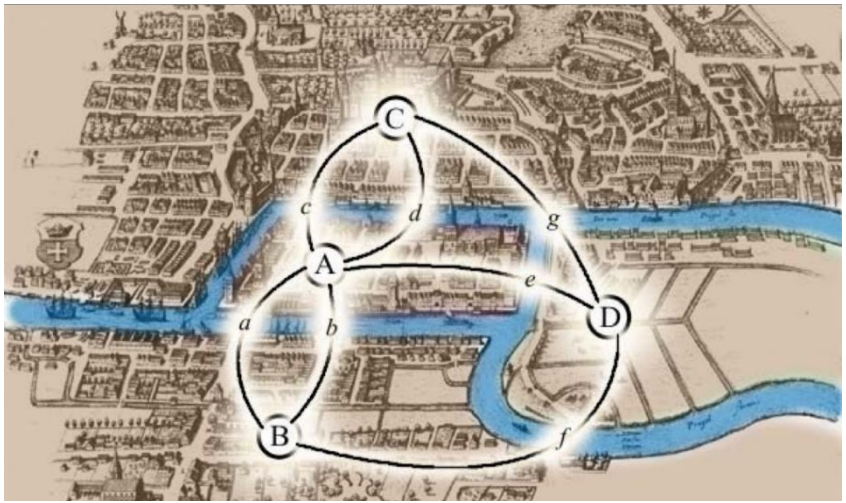
Зада́ча о кёнигсбе́ргских моста́х, или **зада́ча Э́йлера** — старинная математическая задача, которая формулировалась слудующим образом, как можно пройти по всем семи мостам центра старого Кёнигсберга, не проходя ни по одному из них дважды.



Задача 2. Задача о кёнигсбергских мостах.

Зада́ча о кёнигсбе́ргских моста́х, или **зада́ча Э́йлера** — старинная математическая задача, которая формулировалась слудующим образом, как можно пройти по всем семи мостам центра старого Кёнигсберга, не проходя ни по одному из них дважды.



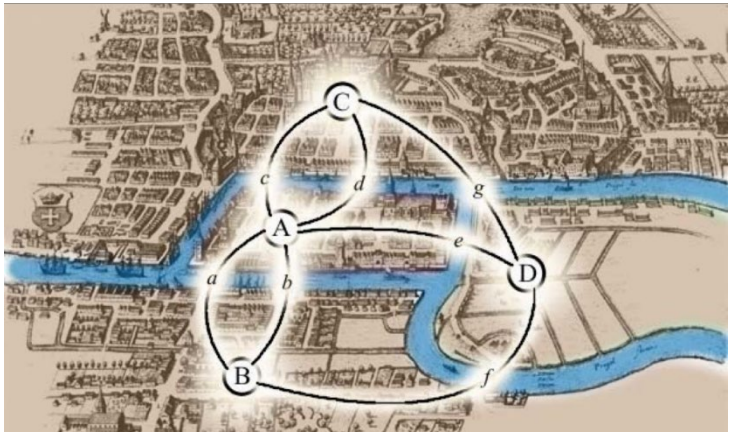


вершины — это четыре части центра Кёнигсберга;
рёбра — это семь мостов в центре города;

Задачу о кёнигсбергских мостах можно переформулировать в терминах теории графов следующим образом: существует ли в графе задачи о кёнигсбергских мостах эйлеров цикл или эйлерова цепь?

Задача 2. Задача о кёнигсбергских мостах.

Вершина A (остров): Степень = **5** (нечётная).
Вершина B (берег): Степень = **3** (нечётная).
Вершина C (берег): Степень = **3** (нечётная).
Вершина D (остров): Степень = **3** (нечётная).



Вывод: В графе Кёнигсбергских мостов **все четыре вершины имеют нечётную степень**.

Простой итог. Правило Эйлера звучит так:

- Если всех "нечётных" точек 0 — маршрут сделать МОЖНО, и он будет замкнутым (возврат в начало).
- Если "нечётных" точек ровно 2 — маршрут сделать МОЖНО, но начнётся он в одной из них, а закончится в другой.
- Если "нечётных" точек больше 2 (1, 3, 4, 5...) — маршрут сделать НЕЛЬЗЯ.

В Кёнигсберге 4. Ответ: нельзя.

Задача 2. Задача о кёнигсбергских мостах.

Суть открытия Эйлера

Эйлер догадался, что дело не в длине пути и не в форме города, а в чётности — то есть, в том, чётное или нечётное количество мостов ведёт к каждой части суши (берегу или острову).

Ключевое правило

Каждый раз, когда покидается один из берегов по мосту, то потом нужно вернуться на него по другому мосту, чтобы в итоге оказаться там же.

Если мостов чётное число (2, 4, 6...):

Выйти → вернуться → выйти → вернуться.

Если мостов нечётное число (1, 3, 5...):

выйти → вернуться → выйти. Последний раз выйти, но вернуться уже не нельзя, потому что мосты закончились. Значит, эта точка должна быть либо началом, либо концом маршрута.