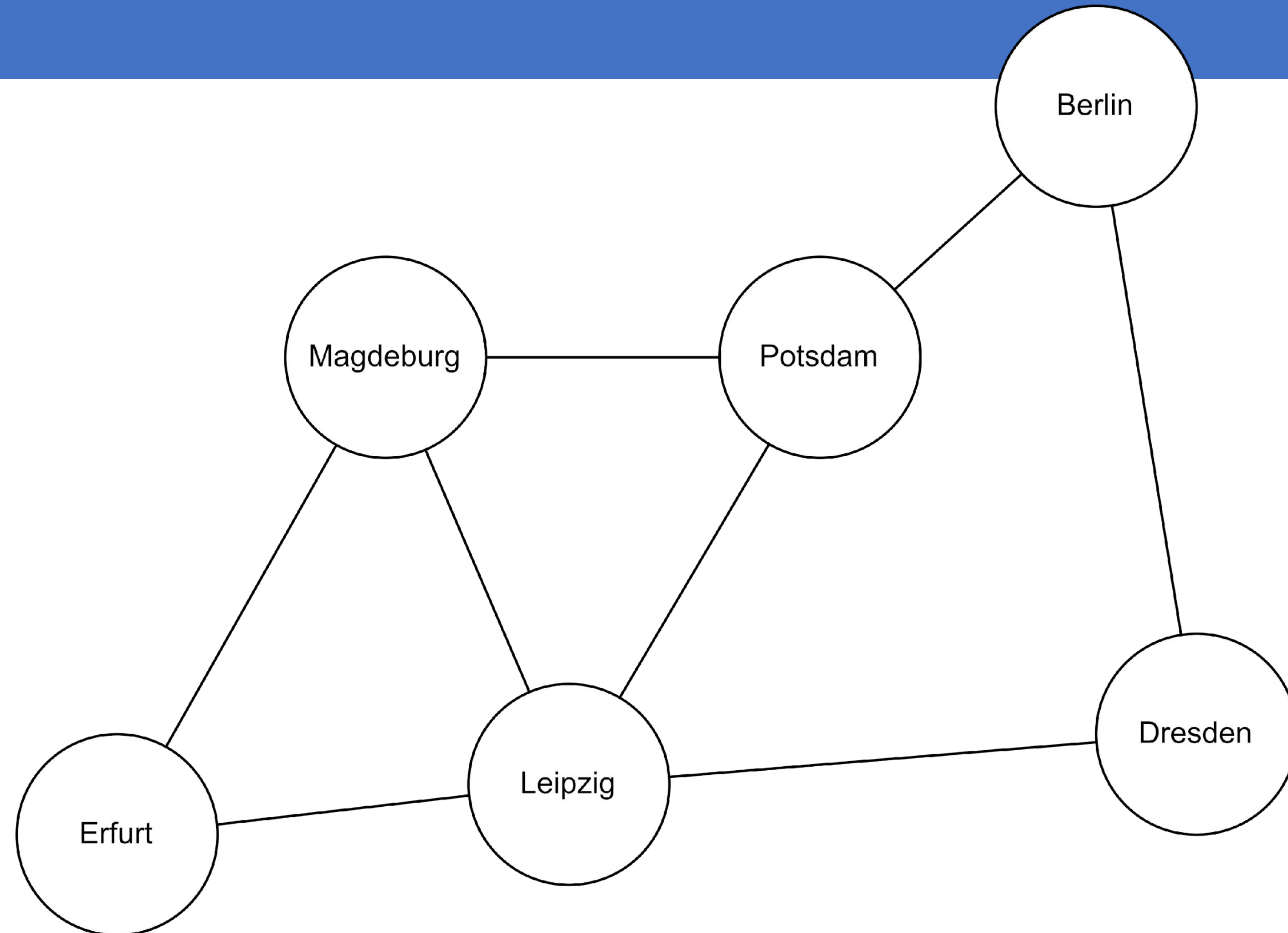
Теория графов

# Теория графов

- Графом называется множество узлов и связей между ними.
- Каждый узел называется *вершиной*, а каждая связь *ребром*.
- Каждое ребро соединяет только две вершины.

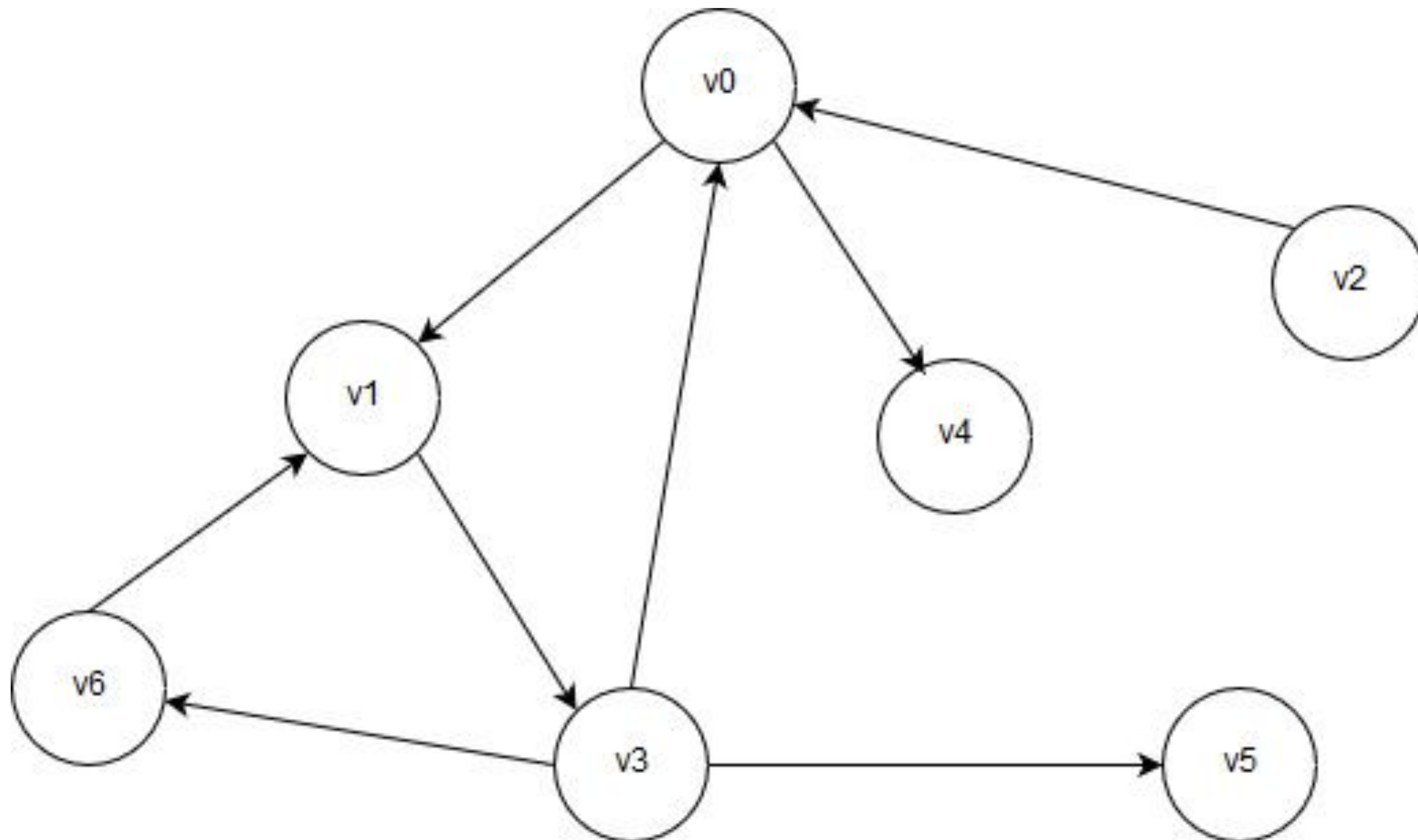
# Пример графа



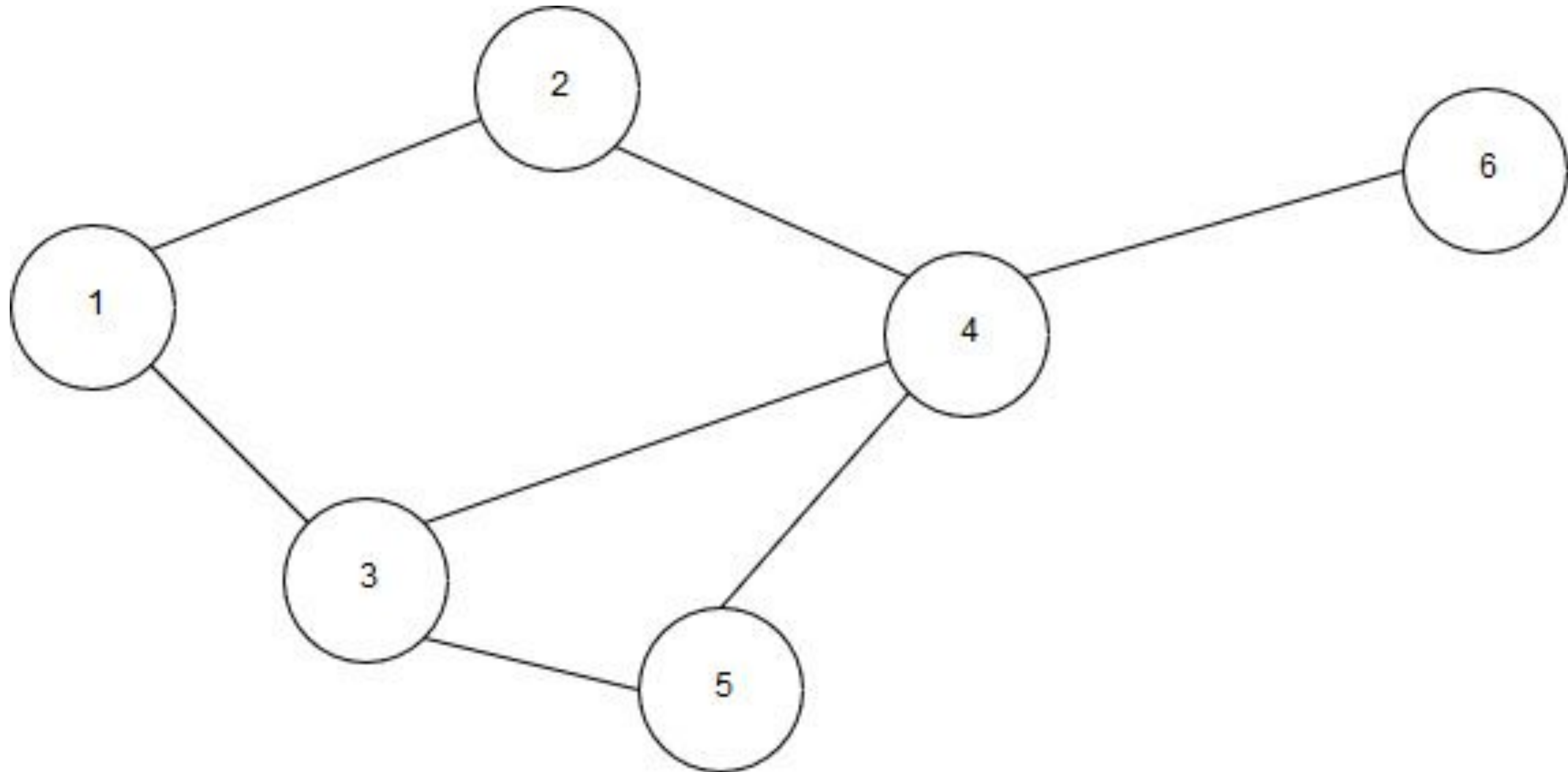
# Классификация графов

- Графы делятся на ***ориентированные*** и ***неориентированные***.
- *Ориентированный граф* – такой граф, в котором можно двигаться от вершины к вершине только в одном направлении
- *В неориентированном* графе можно передвигаться свободно от вершины к вершине в любую сторону.
- Граф, в котором присутствуют как ориентированные ребра, так и неориентированные, называется

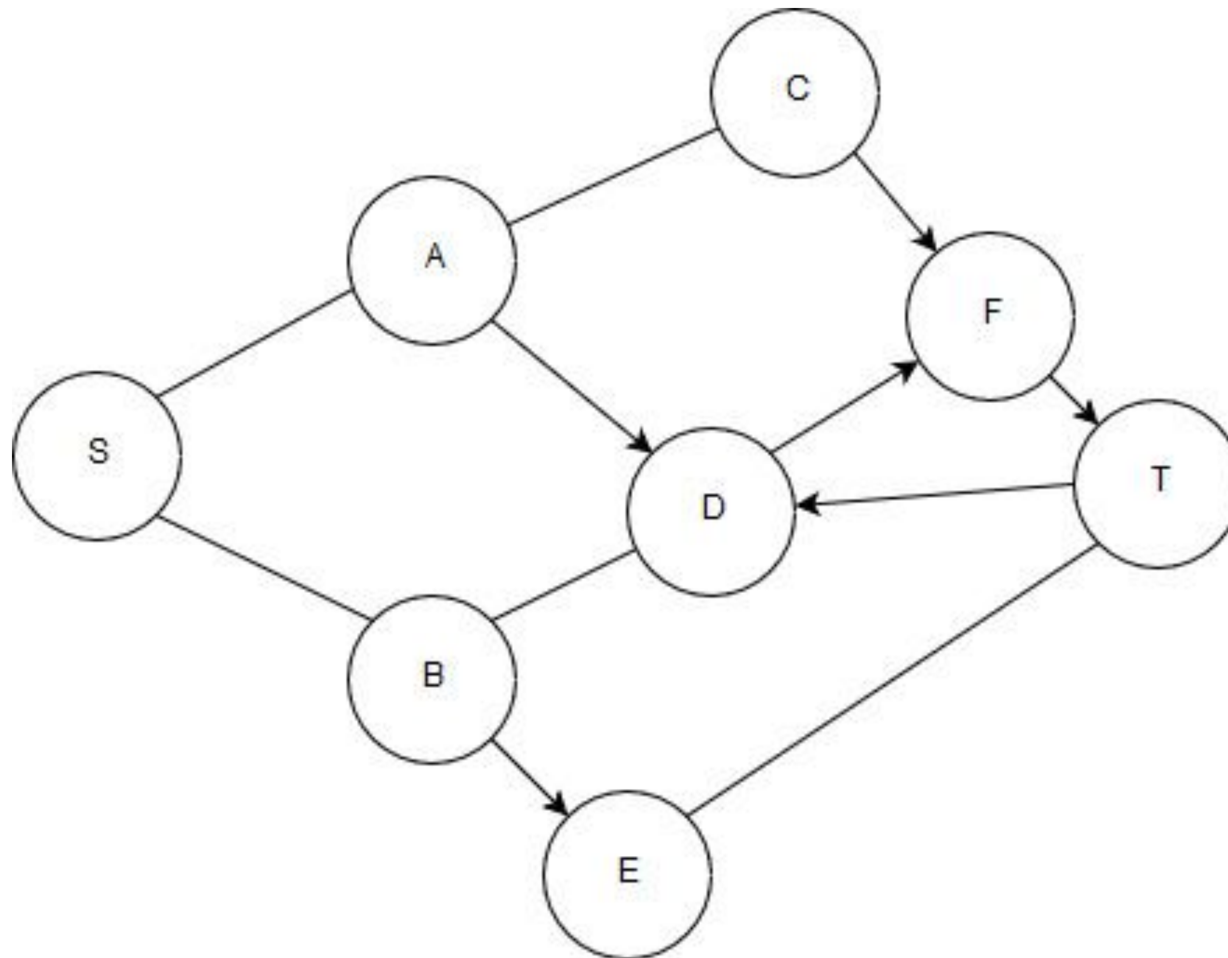
# Пример ориентированного графа



# Пример неориентированного графа



# Пример смешанного графа

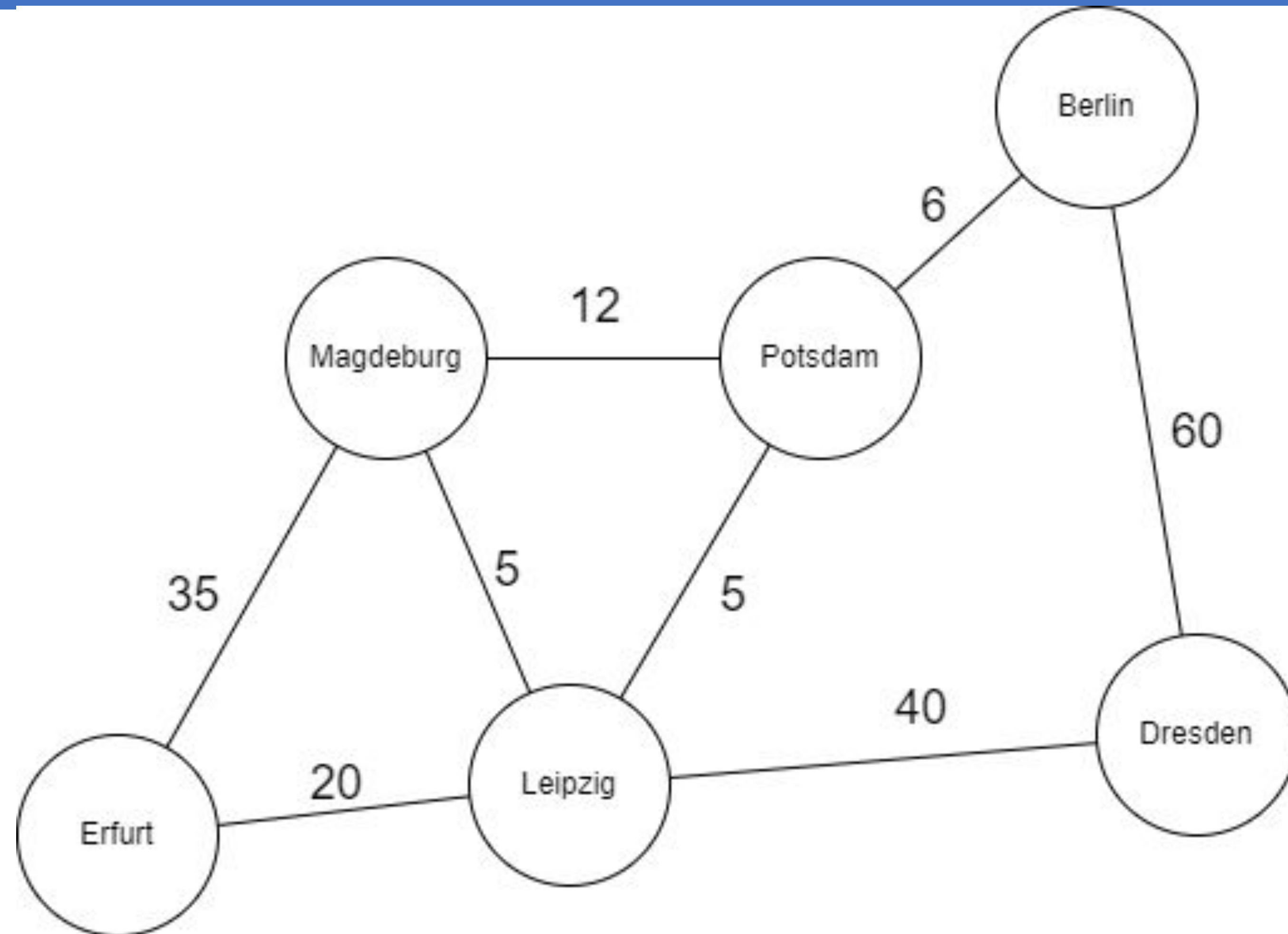


# Классификация графов

- Графы также делятся на ***взвешенные*** и ***невзвешенные***.
- Граф, в котором *каждому ребру* в соответствие поставлено некоторое *числовое значение – вес*, называется *взвешенным* графом.
- Если *никакого числового значения* рёбрам не поставлено, то граф называется *невзвешенным*.

- До этого мы рассматривали только невзвешенные графы
- В названии графа указывают его ориентированность/неориентированность и взвешенность/невзвешенность

# Взвешенный неориентированный граф

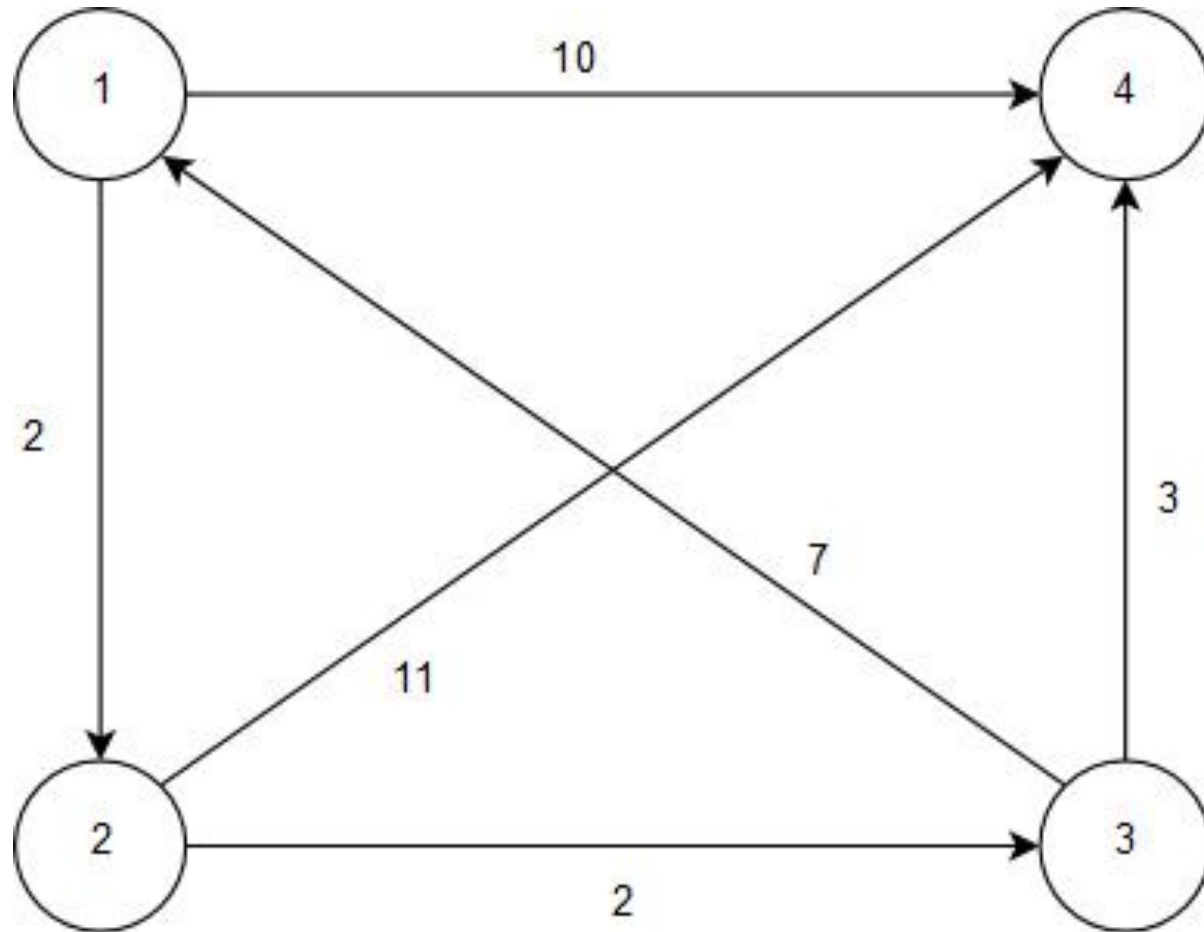


## Классификация графов

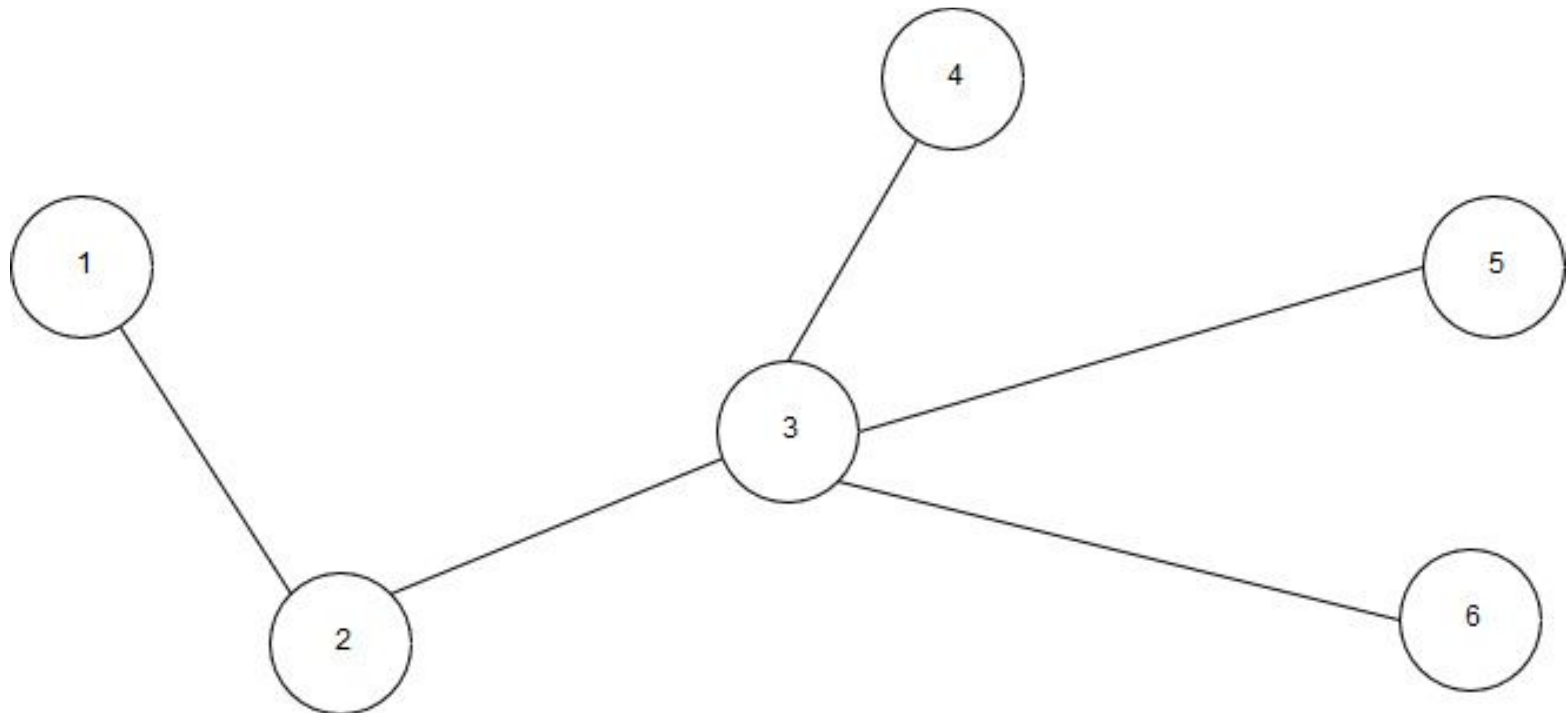
- Граф, в котором между любой парой вершин существует, как минимум, один путь, называется связным
- Если в графе существует хотя бы одна вершина, не связанная с другими, он называется несвязным.

- Зачастую в названии графа связность/несвязность опускается.

# Взвешенный связанный ориентированный граф



# Невзвешенный несвязанный неориентированный граф



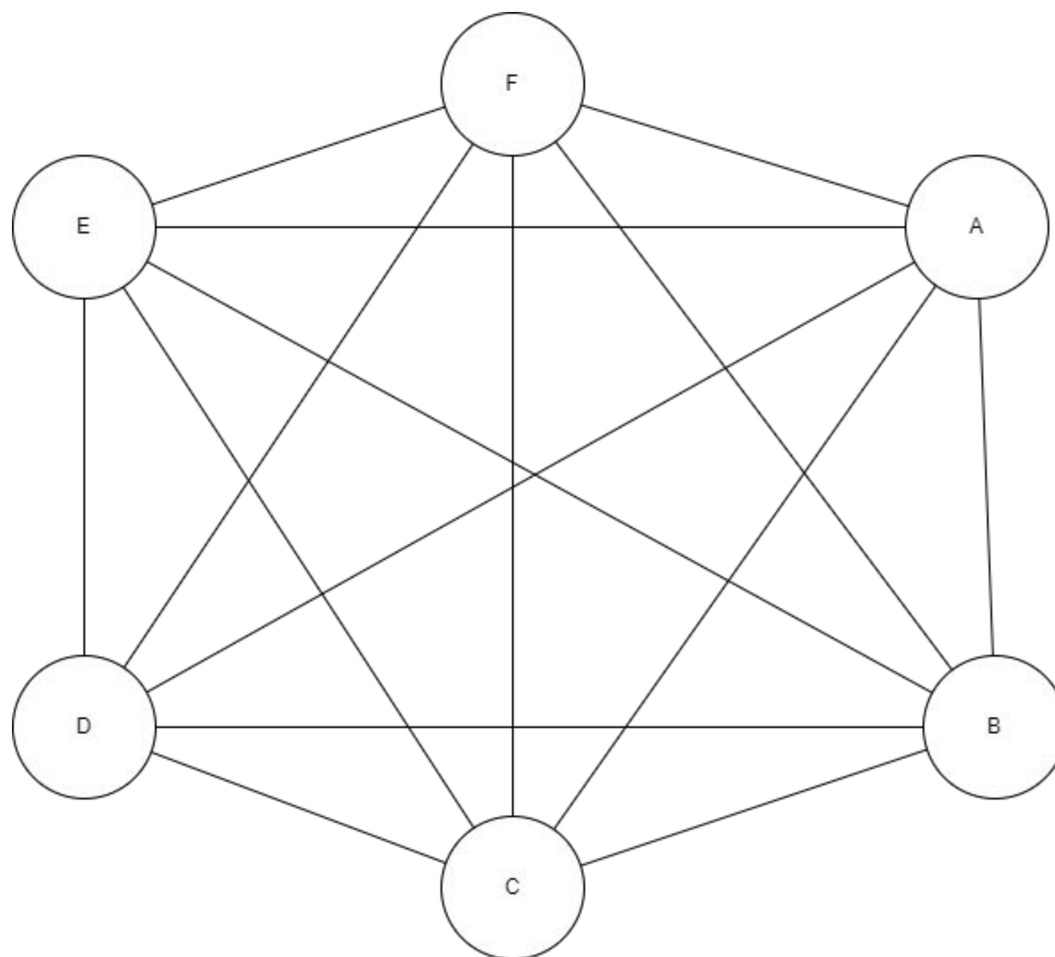
# Классификация графов

- Граф, в котором число рёбер близко к максимальному (когда каждая вершина графа связана с любой другой вершиной графа рёбрами), называется ***плотным графом***.

- Граф с противоположным свойством, имеющий малое число рёбер, называется ***разреженным графом*****.**

- Любой связный граф является плотным, но не каждый плотный является связным

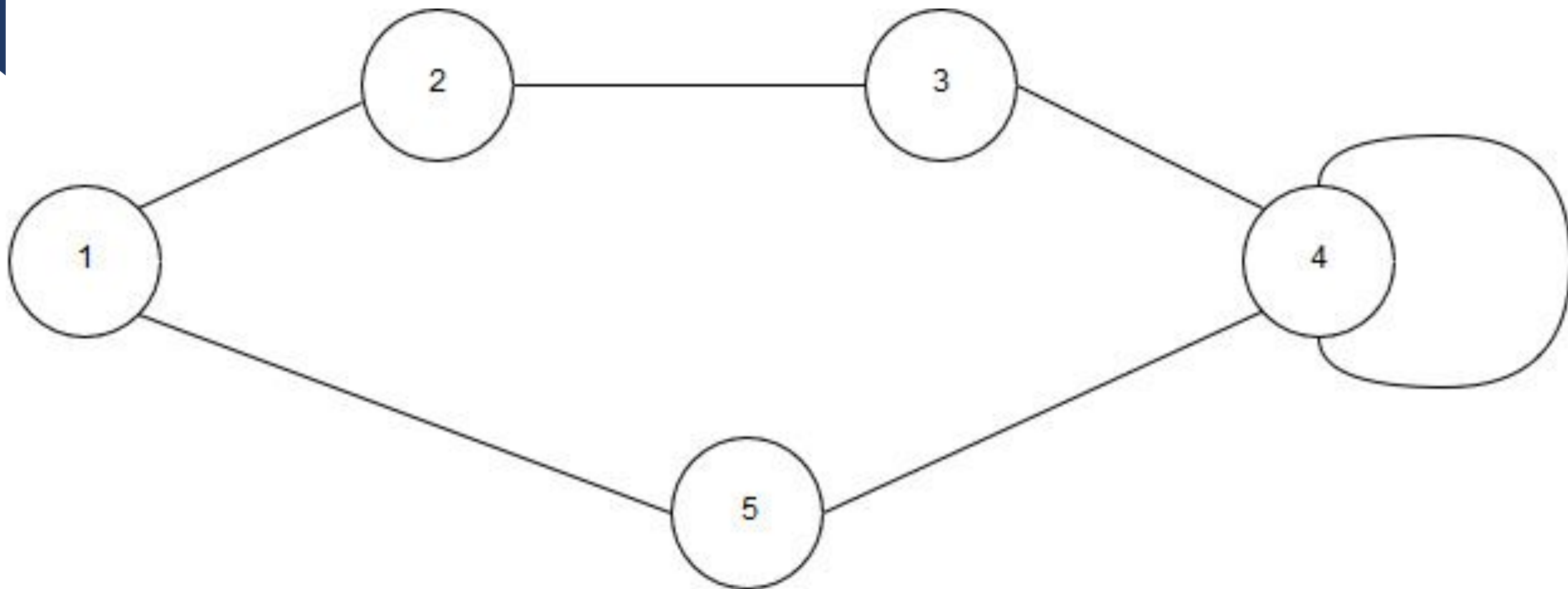
# Плотный граф



# Петля

- ***Петлёй*** называется ребро, которое соединяет вершины v1 и v2, причём v1 и v2 совпадают. Иными словами, петля – это ребро, которое начинается и заканчивается в одной вершине.

# Граф с петлей

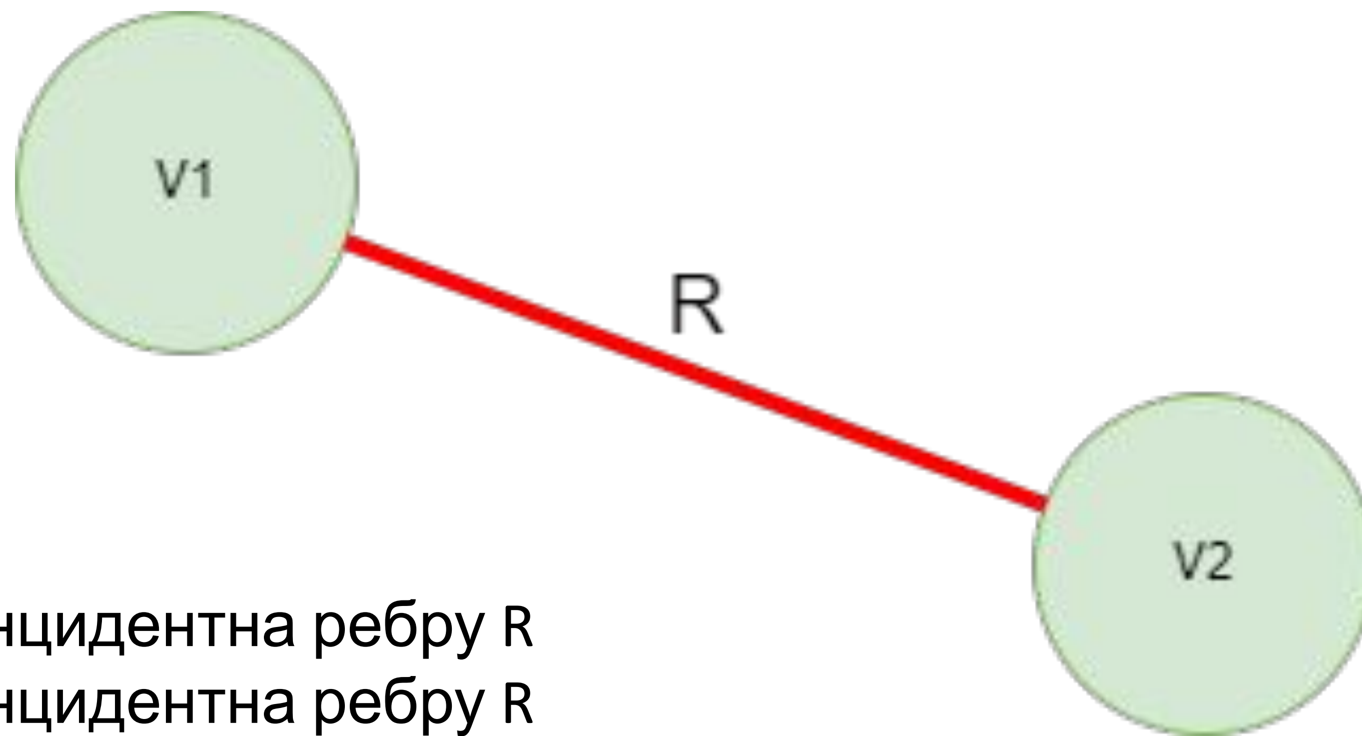


# Инцидентность

**Инцидентность** – понятие, используемое только в отношении ребра и вершины. Если v1, v2 - вершины, а R = (v1, v2) - соединяющее их ребро, тогда вершина v1 и ребро R инцидентны, вершина v2 и ребро R тоже инцидентны.

Две вершины (или два ребра) инцидентными быть не могут.

# Инцидентность




Вершина V1 – инцидентна ребру R
Вершина V2 – инцидентна ребру R

# Смежность

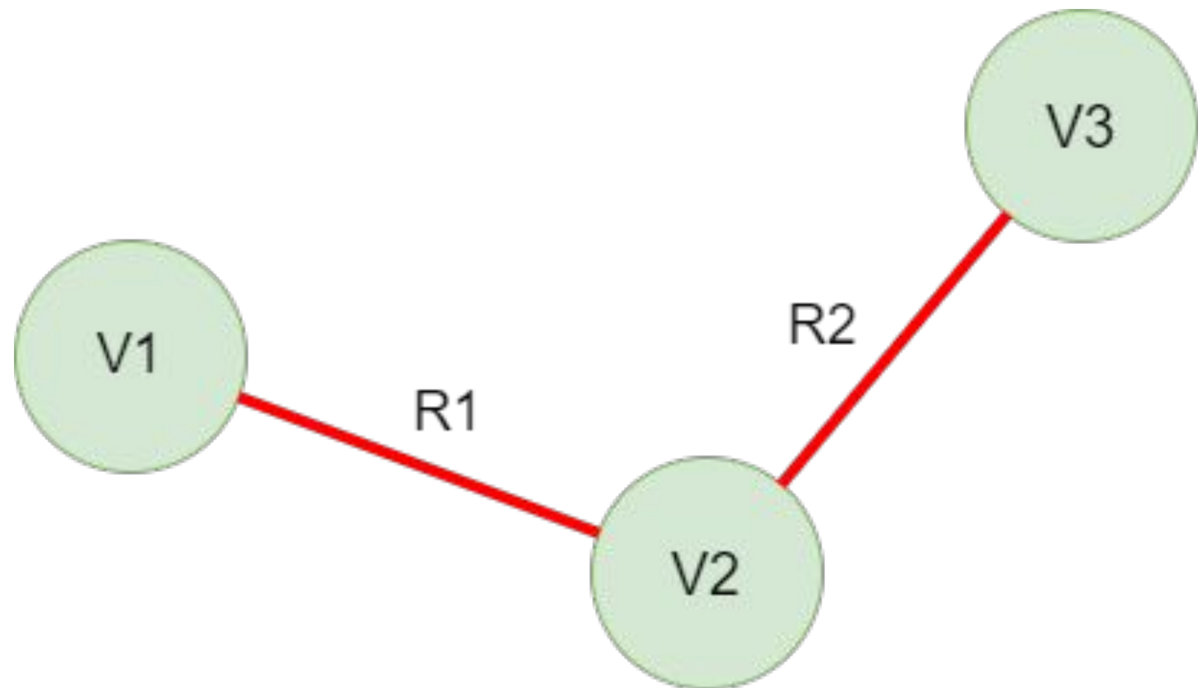
- ***Смежность*** – понятие, используемое в отношении только двух рёбер, либо только двух вершин: два ребра, инцидентные одной вершине, называются смежными; две вершины, инцидентные одному ребру, также называются смежными.

# Смежность

Ребра R1 и R2 – смежные т.к инцидентны одной вершине V2

Вершины V1 и V2 смежные т.к инцидентны одному ребру R1

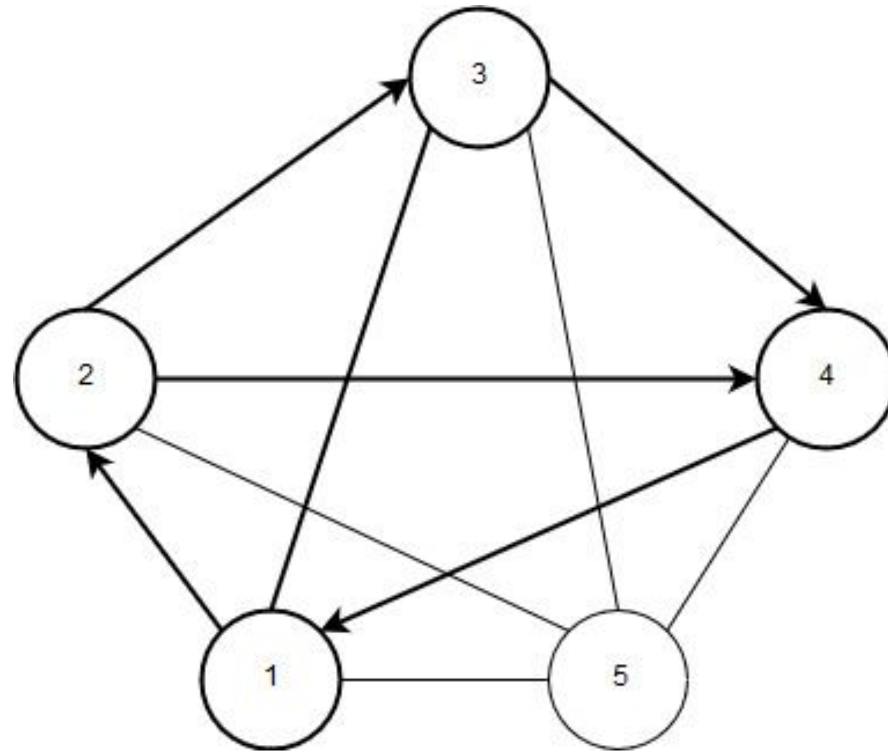
Вершины V2 и V3 смежные т.к. инцидентны одному ребру R2



# Маршрут

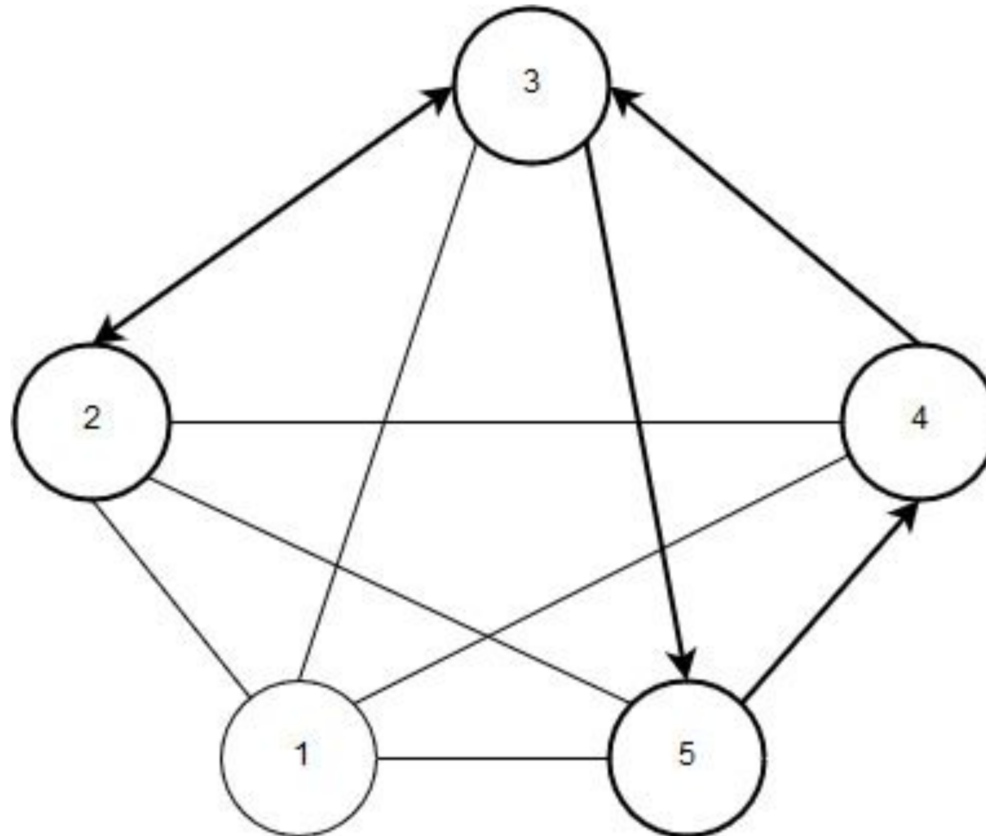
- ***Маршрут*** – это проход по графу через заданную последовательность вершин.
- Если $v_0 = v_k$, начальная и конечная вершины последовательности совпадают, то маршрут *замкнут,* иначе маршрут *открыт.*

# Открытый маршрут




2-4-1-2-3-4-1

# Замкнутый маршрут



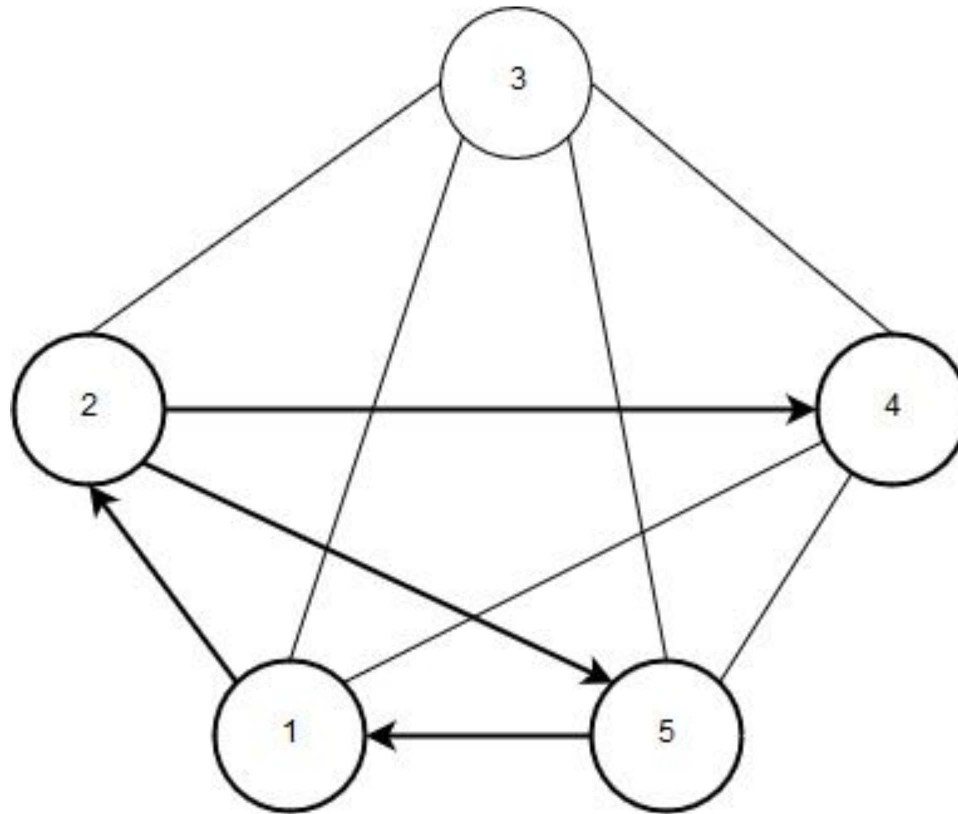

2-3-5-4-3-2

## Цепь

- ***Цепь*** – маршрут, в котором любое ребро графа входит не более одного раза. Если все вершины такого маршрута не повторяются, то цепь называется *открытой* (рисунок 10).

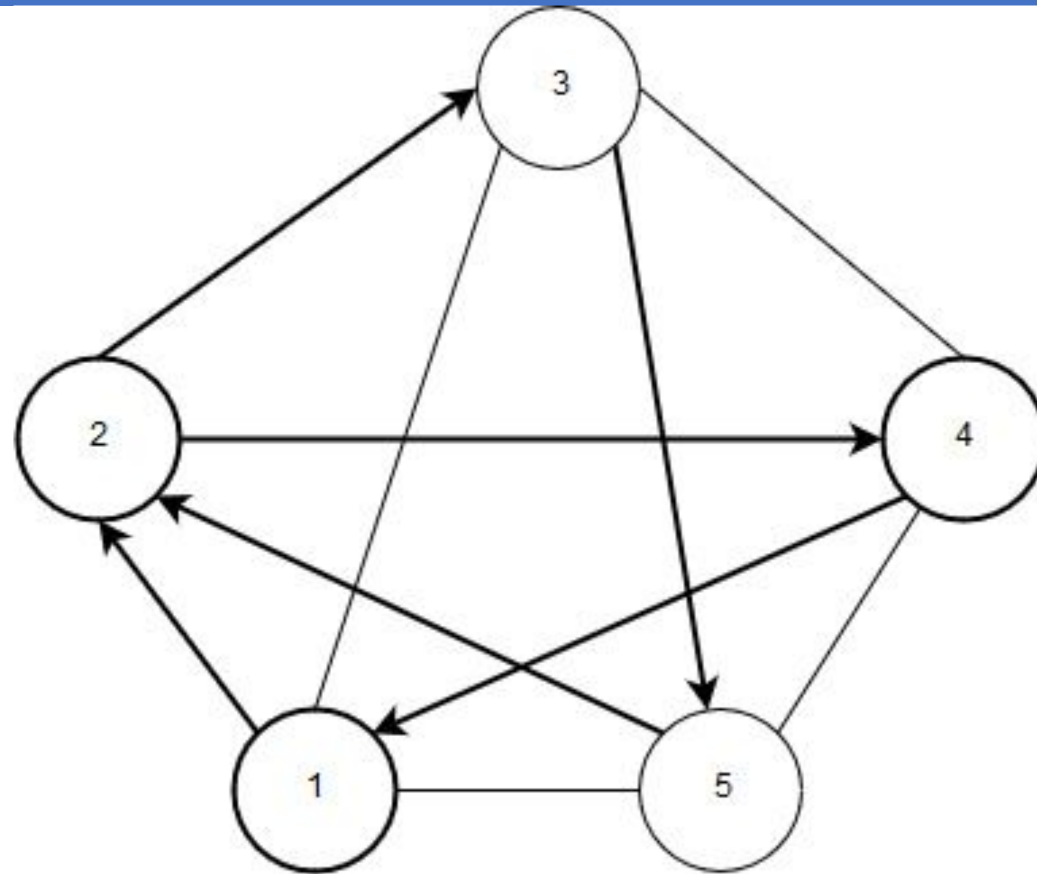
- ***Циклом*** называется цепь в которой начальная и конечная точки маршрута являются одной вершиной. На рисунке 11 вершина 2 является началом и концом циклического маршрута.

# Открытая цепь




2-5-1-2-4

# Замкнутая цепь

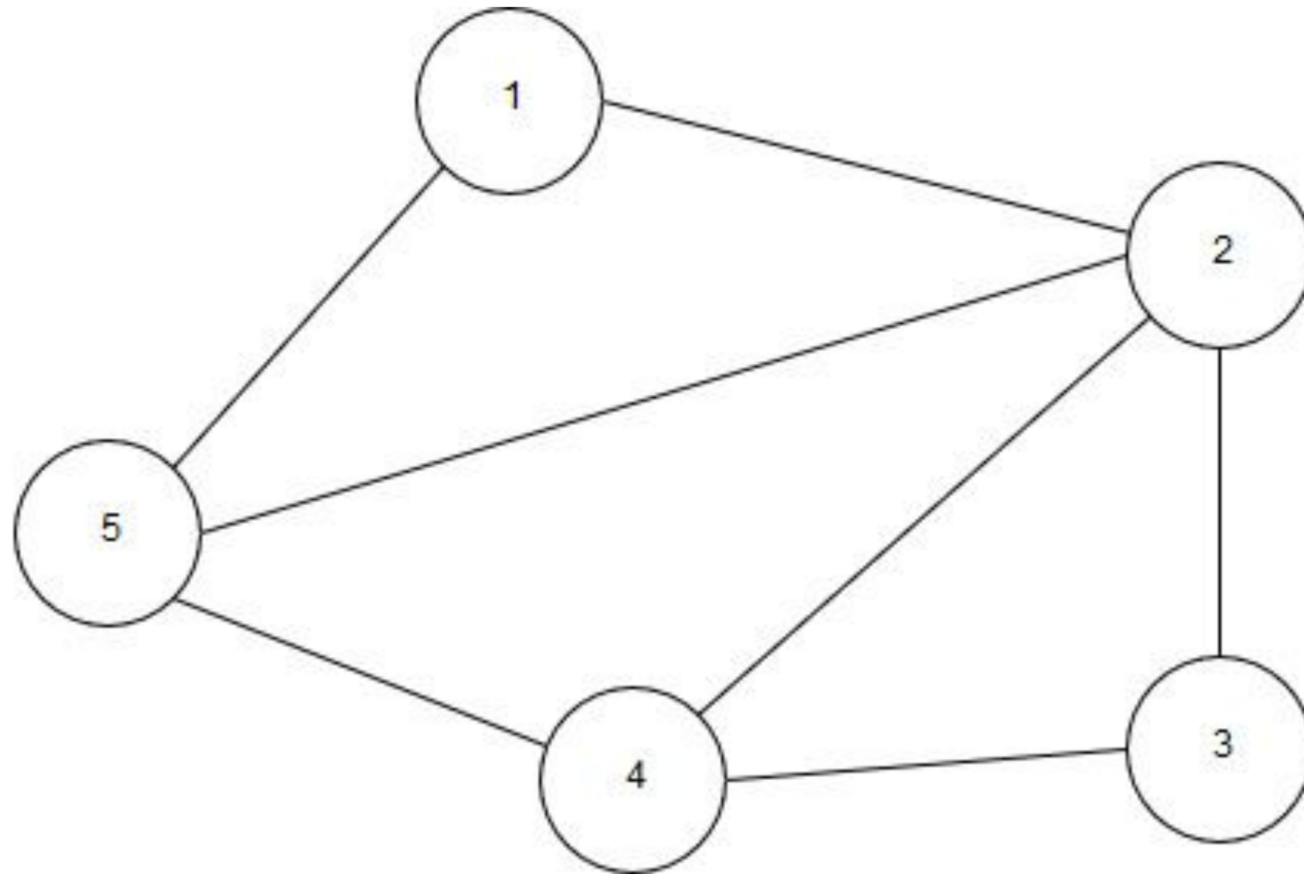



2-4-1-2-3-5-2

# Представление графов. Матрица смежности.

- ***Матрица смежности графа*** – это способ представления графа в виде квадратной матрицы, в которой каждый элемент принимает одно из двух значений: 0 или 1 для невзвешенного графа. Значения 1 и 0 отображают существование ребра между вершинами, которые характеризуются номерами строк и столбцов.
- Если граф взвешенный, то элементами матрицы смежности будут числа, соответствующие весам ребер, соединяющих вершины, на пересечении которых в таблице стоит элемент.

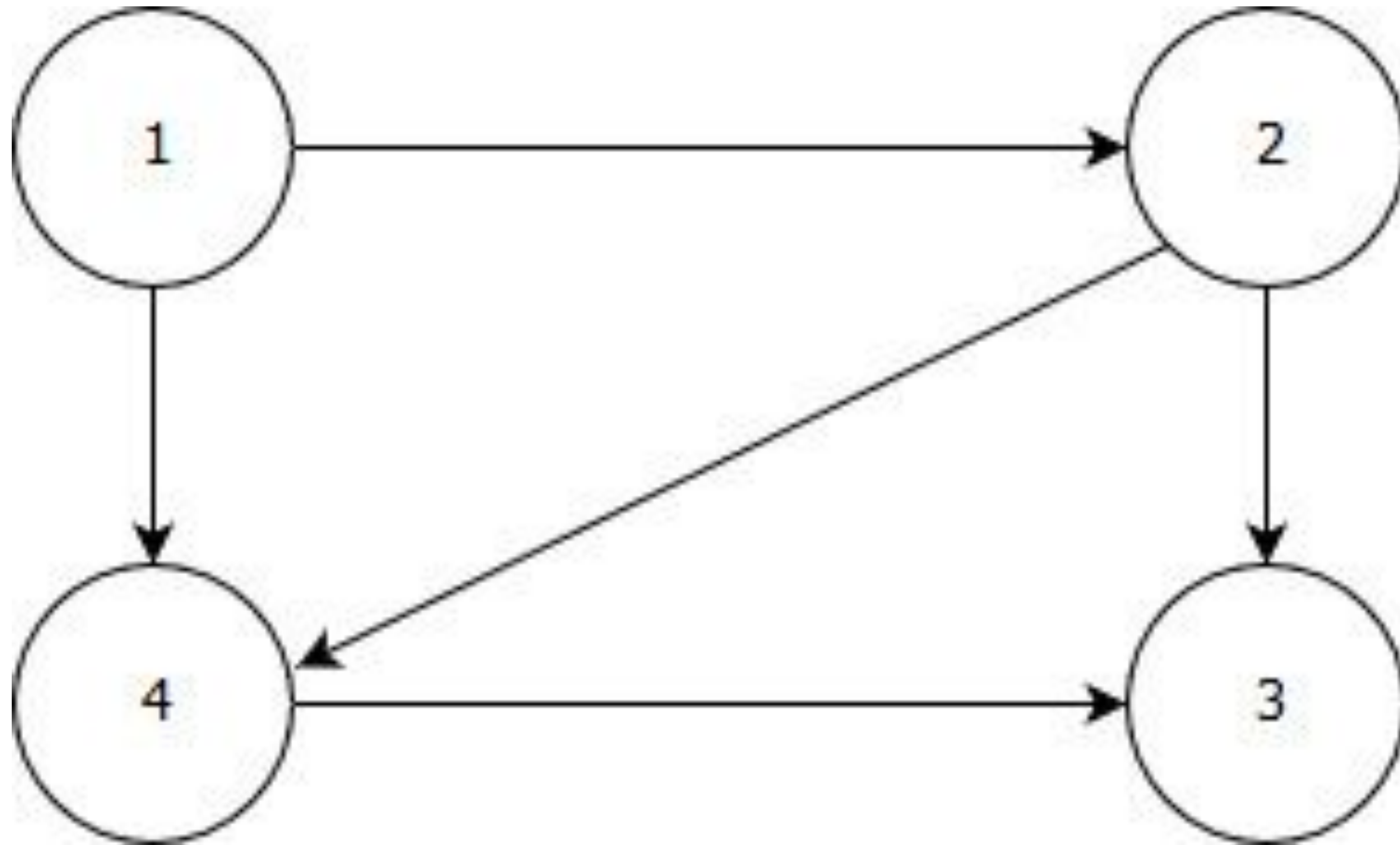
# Невзвешенный неориентированный граф



# Матрица смежности

| № вершины | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

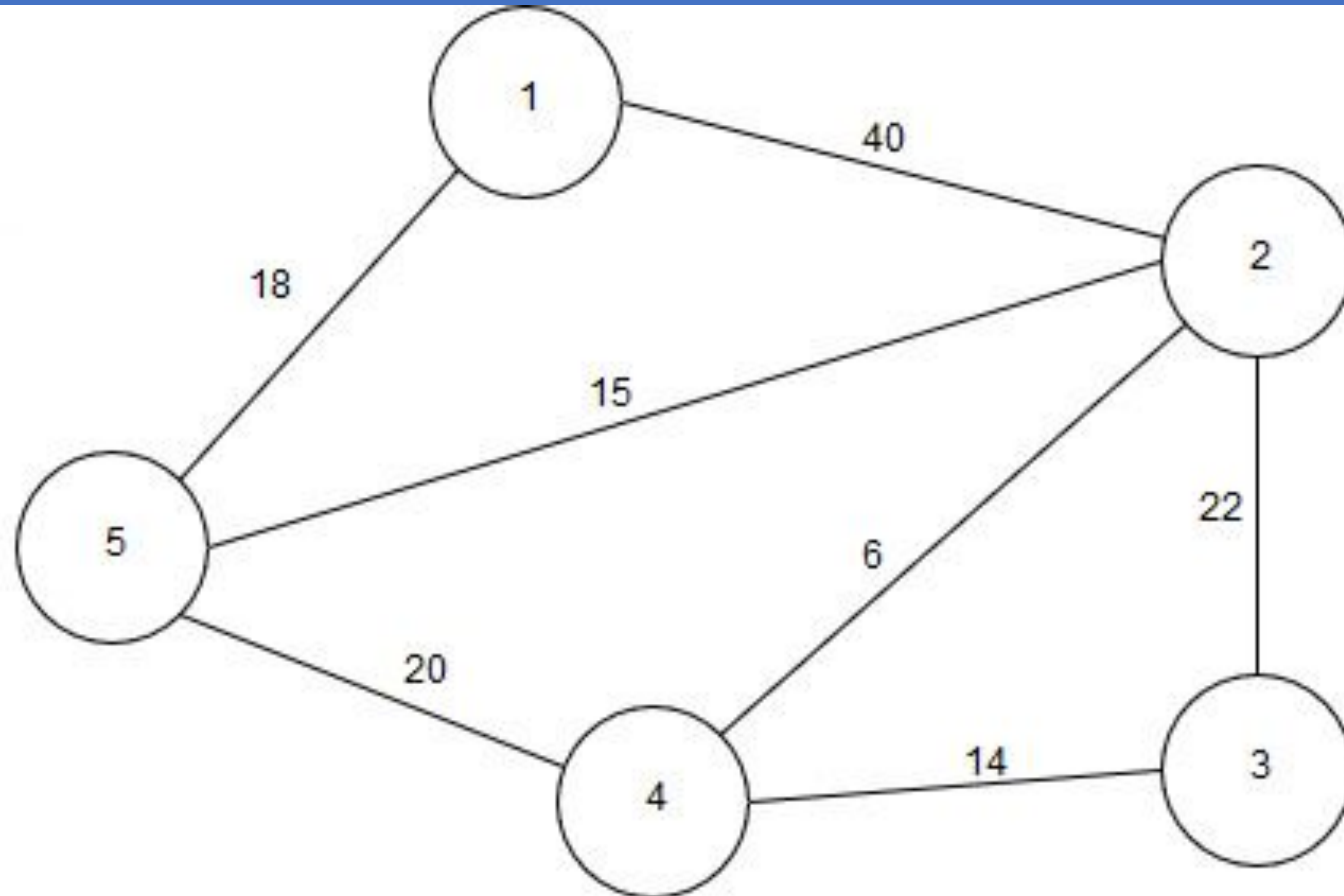
# Невзвешенный ориентированный граф



# Матрица смежности

| № вершины | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 |

# Взвешенный неориентированный граф

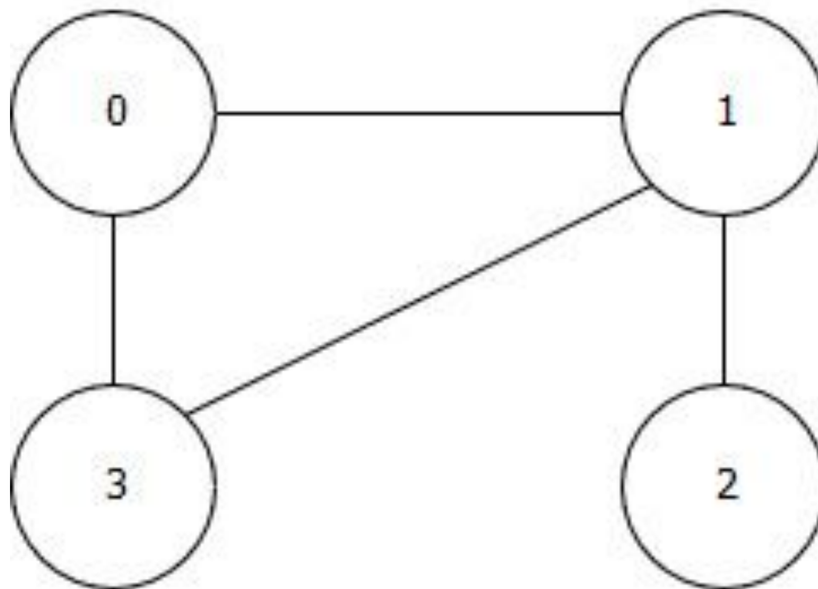


# Матрица смежности

| № вершины | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 40 | 0 | 0 | 18 |
| 2 | 40 | 0 | 22 | 6 | 15 |
| 3 | 0 | 22 | 0 | 14 | 0 |
| 4 | 0 | 6 | 14 | 0 | 20 |
| 5 | 18 | 15 | 0 | 20 | 0 |

# вектор смежности

Граф может быть представлен с помощью ***вектора смежности***. В векторе смежности для каждой вершины хранятся номера смежных с ней вершин.

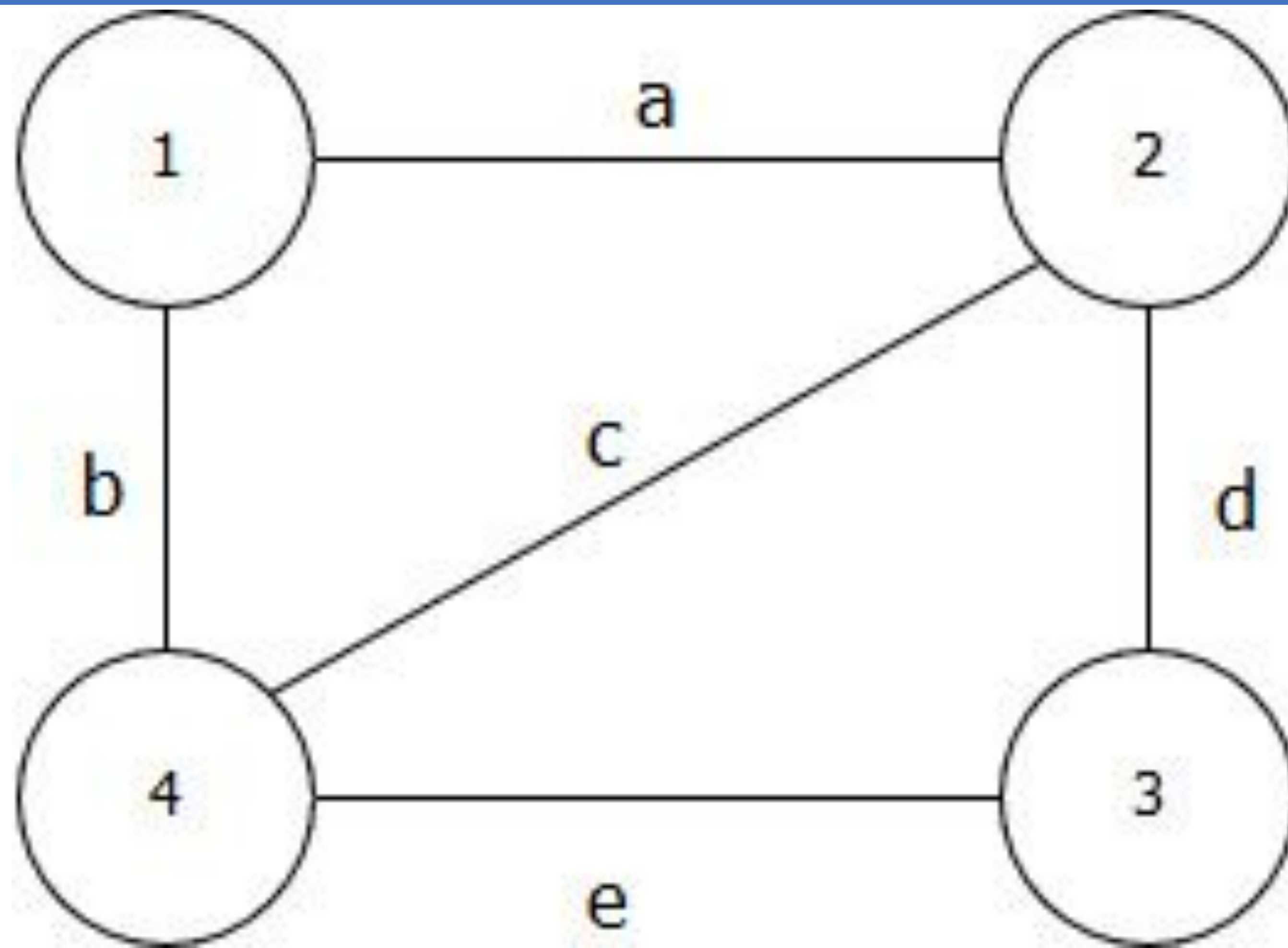



| № вершины | Вектор смежности |
|---|---|
| 0 | 1 3 |
| 1 | 0 2 3 |
| 2 | 1 |
| 3 | 0 1 |

# Матрица инцидентности

- Еще одной формой представления графа является ***матрица инцидентности***, в которой указываются связи между инцидентными элементами графа. Столбцы матрицы соответствуют ребрам, строки – вершинам.
- В каждом элементе матрицы инцидентности неориентированного графа стоит 0, если вершина не инцидентна ребру, или 1, если вершина инцидентна ребру.
- В случае ориентированного графа в матрицу вносятся 1, если вершина инцидентна ребру и является его началом, или 0, если вершина не инцидентна ребру, или -1, если вершина инцидентна ребру и

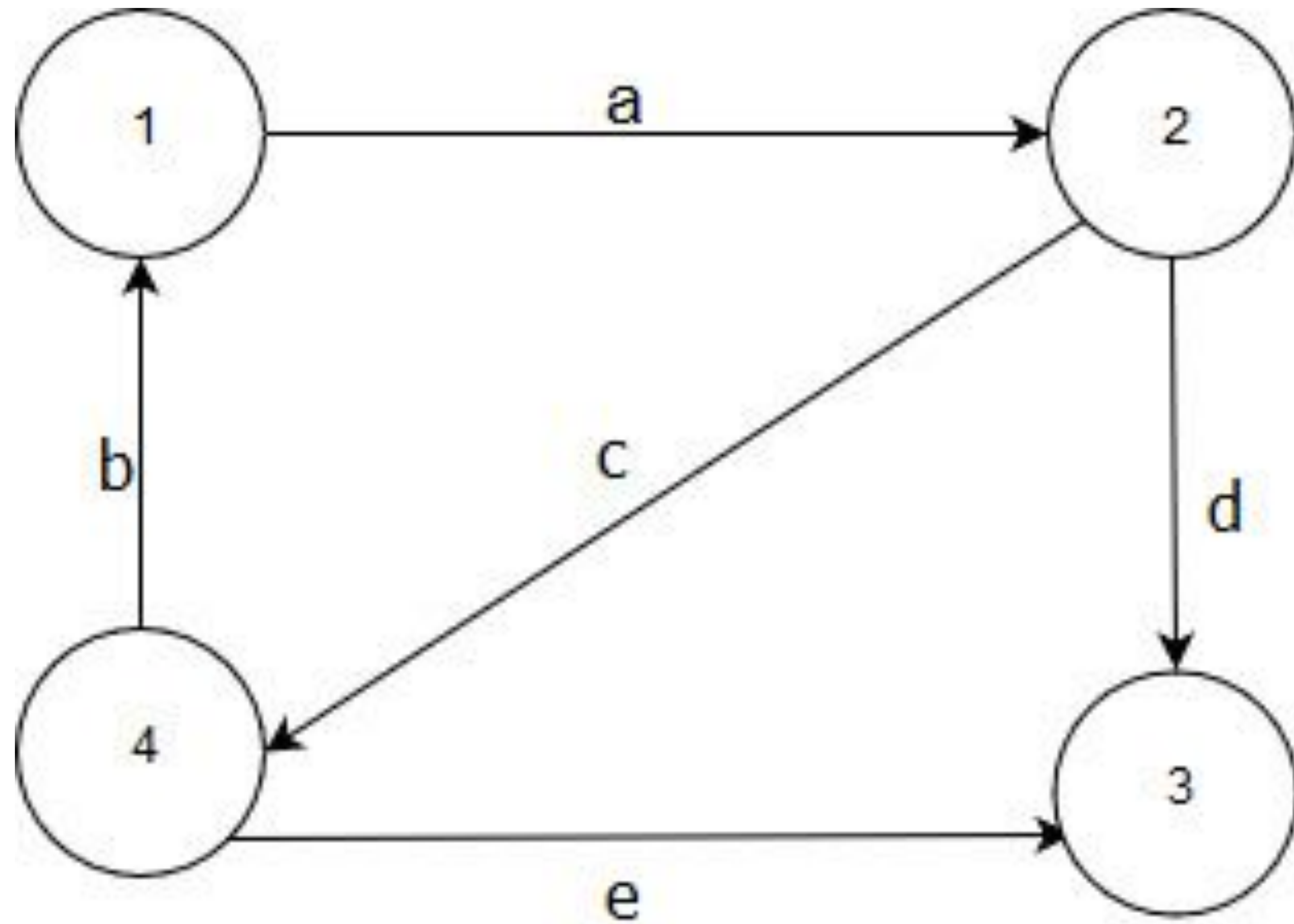
# Невзвешенный неориентированный граф



# Матрица инцидентности

| ребро<br>вершина | a | b | c | d | e |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 1 |

# Невзвешенный ориентированный граф



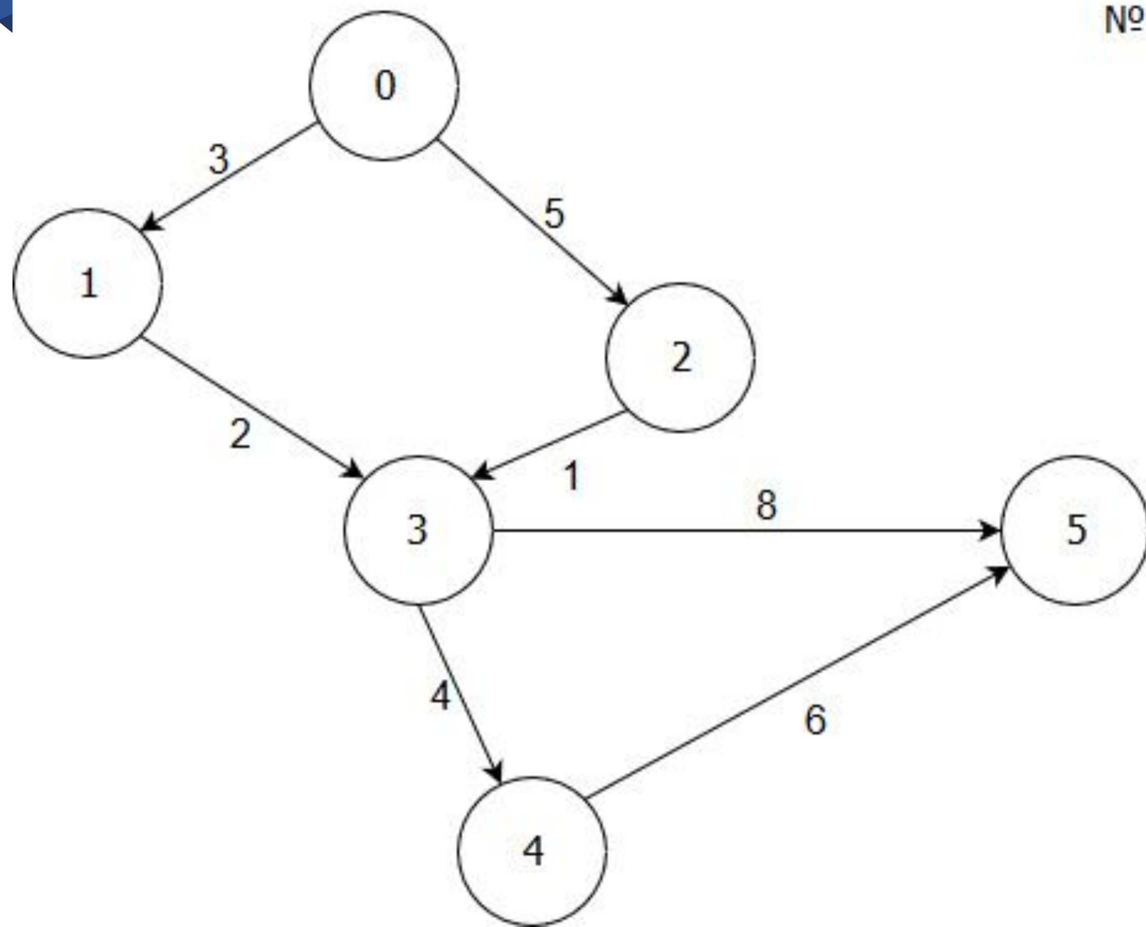
# Матрица инцидентности

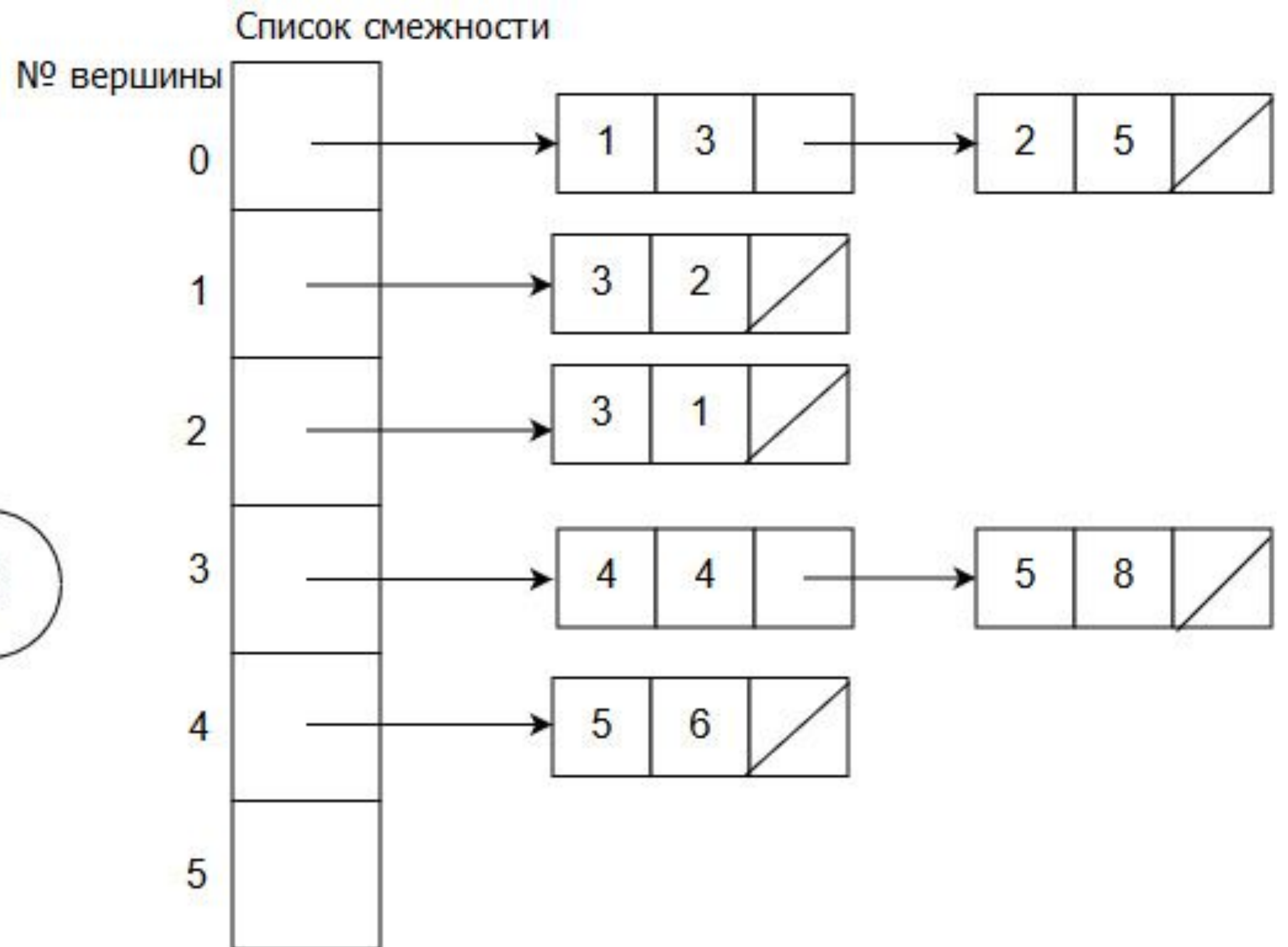
| ребро<br>вершина | a | b | c | d | e |
|---|---|---|---|---|---|
| 1 | 1 | -1 | 0 | 0 | 0 |
| 2 | -1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | -1 | -1 |
| 4 | 0 | 1 | -1 | 0 | 1 |

## Список смежности

- ***Список смежности*** – ещё один из способов представления графа в виде коллекции списков вершин.
- Каждой вершине графа соответствует список, состоящий из "соседей" этой вершины. Если граф взвешенный, то рядом с номером вершины-соседа также указывается длина ребра до этого соседа.

# Список смежности



# Проектирование графов на алгоритмическом языке C++

- Создается класс Graph, он будет шаблонным, чтобы узел графа мог содержать данные разного типа (например, строки или числа).
- Граф будет представлен следующим образом: все вершины графа заносятся в вектор вершин где индекс каждой вершины соответствует ее индексу в матрице смежности.
- Например, если в векторе вершин у одной вершины индекс *i*, а у другой *j*, то наличие или отсутствие ребра между вершинами в матрице смежности определяется значением по индексу [i][j].
- В приватном поле класса Graph хранится

# Свойства класса

Вектор вершин

$$\begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$$

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Матрица смежности

$a_{ij}$ - Веса ребер, инцидентных i-ой и j-й вершине. В случае невзвешенных графов принимают два значения – 0 и 1

*n* – размер матрицы смежности

# Создание класса. Свойства класса

```
template <class T>
class Graph
{
private:
    //Вектор вершин
    vector<T> vertexList;

    //Матрица смежности
    vector<vector<int>> adjMatrix;

    //Размер вектора вершин и матрицы смежности
    int maxSize;
```

# Создание класса. Конструктор

```
template<class T>
inline Graph<T>::Graph(const int& size)
{
    this->maxSize = size;
    this->adjMatrix = vector<vector<T>>(size, vector<T>(size));
    for (int i = 0; i < this->maxSize; ++i)
    {
        for (int j = 0; j < this->maxSize; ++j)
        {
            this->adjMatrix[i][j] = 0;
        };
    };
};
```

# Методы проверки на заполненность

```
template<class T>
inline bool Graph<T>::isFull()
{
    return this->vertexList.size() == this->maxSize;
};

template<class T>
inline bool Graph<T>::isEmpty()
{
    return this->vertexList.size() == 0;
};
```

# Вставка вершины

```
template<class T>
inline void Graph<T>::insertVertex(const T& vert)
{
    if (this->isFull())
    {
        cout << "Невозможно добвать вершину." << endl;
        return;
    };
    this->vertexList.push_back(vert);
};
```

# Получение индекса вершины

```
template<class T>
inline int Graph<T>::GetVertPos(const T& v)
{
    for (int i = 0; i < this->vertexList.size(); ++i)
    {
        if (this->vertexList[i] == v)
        {
            return i;
        }
    }
    return -1;
};
```

# Получение количества вершин

```
template<class T>
inline int Graph<T>::GetAmountVerts()
{
    return this->vertexList.size();
};
```

# Получение веса между вершинами

```
template<class T>
inline int Graph<T>::GetWeight(const T& v1, const T& v2)
{
    if (this->isEmpty())
    {
        return 0;
    };

    int v1_p = this->GetVertPos(v1);
    int v2_p = this->GetVertPos(v2);

    if (v1_p == -1 || v2_p == -1)
    {
        cout << "Одного из узлов в графе не существует." << endl;
        return 0;
    };

    return this->adjMatrix[v1_p][v2_p];
};
```

# Получение вектора соседей

```
template<class T>
std::vector<T> Graph<T>::GetNbrs(const T& vertex) {
    std::vector<T> nbrsList; // создание списка соседей
    int pos = this->GetVertPos(vertex); /* вычисление позиции vertex в матрице смежности */
    if (pos != -1)
    { /* проверка, что vertex есть в матрице смежности */
        for (int i = 0; i < this->vertexList.size(); ++i)
        {
            if (this->adjMatrix[pos][i] != 0)
            {
                nbrsList.push_back(this->vertexList[i]);
            }
        }
    }
    return nbrsList; // возврат списка соседей
};
```

# Вставка ребра для ориентированного графа

```
template<class T>
void Graph<T>::InsertEdge(const T& vertex1, const T& vertex2, int weight = 1) {
    if (GetVertPos(vertex1) != (-1) && this->GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);

        if (this->adjMatrix[vertPos1][vertPos2] != 0 && this->adjMatrix[vertPos2][vertPos1] != 0)
        {
            cout << "Ребро между вершинами уже есть" << endl;
            return;
        }
        else
        {
            this->adjMatrix[vertPos1][vertPos2] = weight;
        }
    }
    else
    {
        cout << "Обеих вершин (или одной из них) нет в графе " << endl;
        return;
    }
};
```

# Вставка ребра для неориентированного графа

```
template<class T>
void Graph<T>::InsertEdge(const T& vertex1, const T& vertex2, int weight = 1) {
    if (GetVertPos(vertex1) != (-1) && this->GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);

        if (this->adjMatrix[vertPos1][vertPos2] != 0 && this->adjMatrix[vertPos2][vertPos1] != 0)
        {
            cout << "Ребро между вершинами уже есть" << endl;
            return;
        }
        else
        {
            this->adjMatrix[vertPos1][vertPos2] = weight;
            this->adjMatrix[vertPos2][vertPos1] = weight;
        }
    }
    else
    {
        cout << "Обеих вершин (или одной из них) нет в графе " << endl;
        return;
    }
};
```

# Печать матрицы смежности графа

```
template<class T>
void Graph<T>::Print() {
    if (!this->isEmpty())
    {
        cout << "Матрица смежности графа: " << endl;

        cout << "-  ";
        for (int i = 0; i < vertexList.size(); ++i)
        {
            cout << vertexList[i] << "  ";
        }
        cout << endl;

        for (int i = 0; i < this->vertexList.size(); ++i) {
            cout << this->vertexList[i] << " ";
            for (int j = 0; j < this->vertexList.size(); ++j) {
                cout << " " << this->adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
    else {
        cout << "Граф пуст " << endl;
    }
}
```

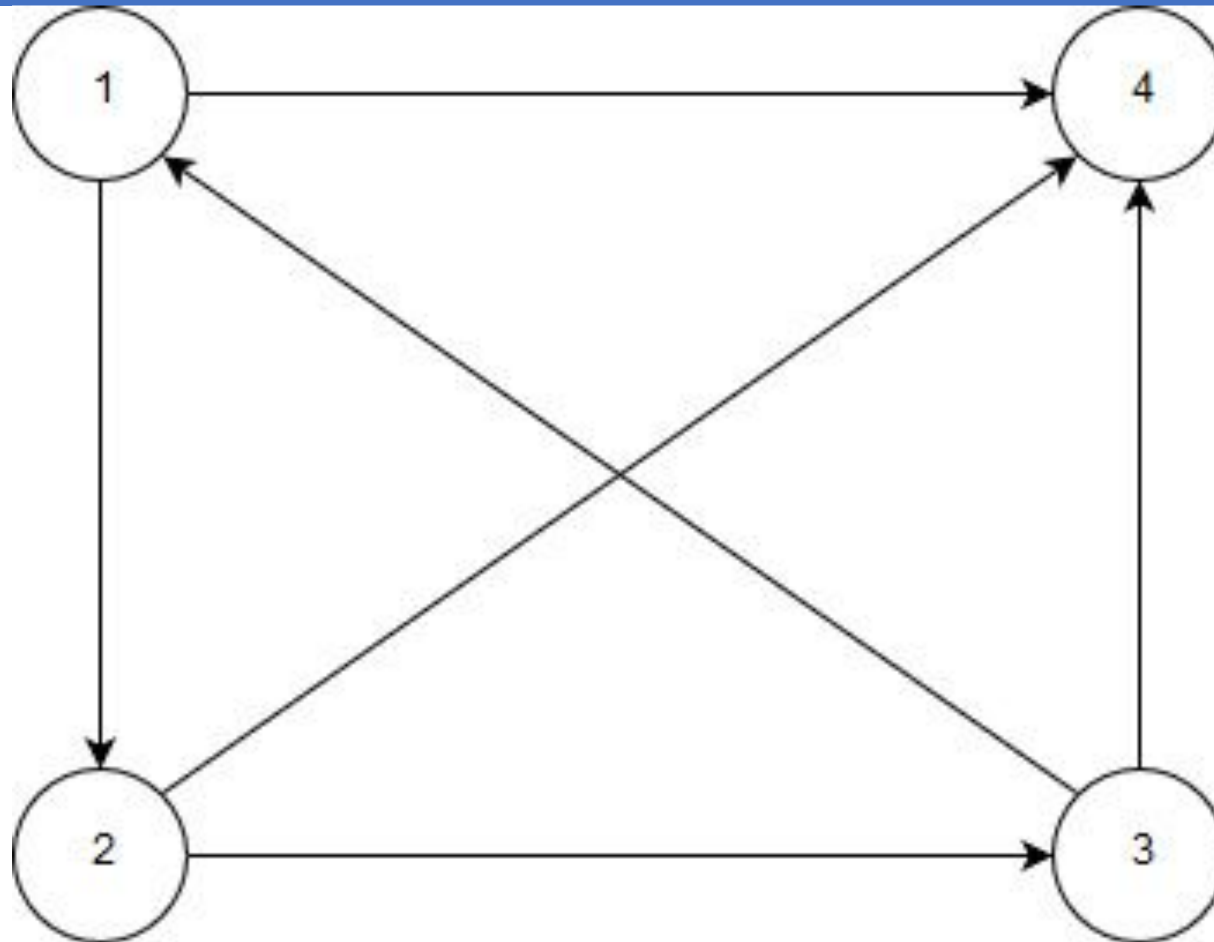
# Получение количества ребер для ориентированного графа

```
template<class T>
int Graph<T>::GetAmountEdges()
{
    int amount = 0;
    if (!this->IsEmpty())
    {
        for (int i = 0; i < this->vertList.size(); ++i)
        {
            for (int j = 0; j < this->vertList.size(); ++j)
            {
                if (this->adjMatrix[i][j] != 0)
                {
                    amount++;
                }
            }
        }
    }
    return amount;
}
```

# Получение количества ребер для неориентированного графа

```
template<class T>
int Graph<T>::GetAmountEdges()
{
    int amount = 0;
    if (!this->IsEmpty())
    {
        for (int i = 0; i < this->vertList.size(); ++i)
        {
            for (int j = 0; j < this->vertList.size(); ++j)
            {
                if (this->adjMatrix[i][j] != 0)
                {
                    amount++;
                }
            }
        }
    }
    return amount / 2;
};
```

# ример работы с графом. (Ориентированным, невзвешенным)

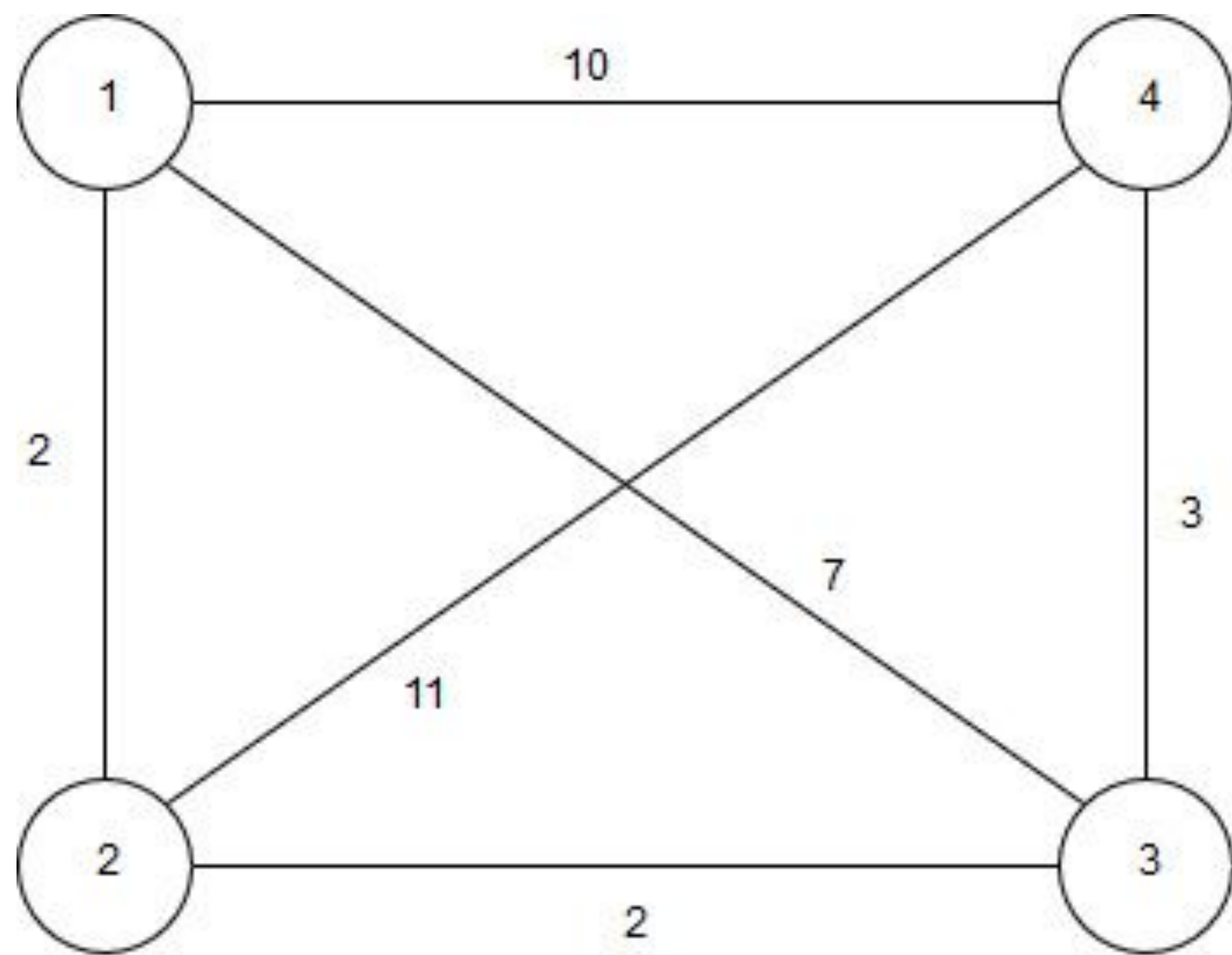


```cpp
int main() {
    setlocale(LC_ALL, "Russian");
    {
        Graph<int> graph; // создание графа, содержащего вершины с номерами целого типа
        int amountVerts, amountEdges, vertex, sourceVertex, targetVertex;
        cout << "Введите количество вершин графа: "; cin >> amountVerts; cout << endl;
        cout << "Введите количество ребер графа: "; cin >> amountEdges; cout << endl;

        for (int i = 0; i < amountVerts; ++i)
        {
            cout << "Вершина: "; cin >> vertex;
            graph.insertVertex(vertex);
            cout << endl;
        }
        for (int i = 0; i < amountEdges; ++i) {
            cout << "Исходная вершина: "; cin >> sourceVertex; cout << endl;
            cout << "Конечная вершина: "; cin >> targetVertex; cout << endl;

            int* targetVertPtr = &targetVertex;

            graph.InsertEdge(sourceVertex, targetVertex);
        }
        cout << endl;
        graph.Print();
    }
    _getch();
    return 0;
}
```

```
C:\Users\ASUS\OneDrive\Документы\ПНИПУ\Первый кур
Введите количество вершин графа: 4

Введите количество ребер графа: 6

Вершина: 1

Вершина: 2

Вершина: 3

Вершина: 4
```

```
Исходная вершина: 1

Конечная вершина: 2

Исходная вершина: 1

Конечная вершина: 4

Исходная вершина: 2

Конечная вершина: 4

Исходная вершина: 2

Конечная вершина: 3

Исходная вершина: 3

Конечная вершина: 4

Исходная вершина: 3

Конечная вершина: 1
```

```
Матрица смежности графа:
-   1   2   3   4
1   0   1   0   1
2   0   0   1   1
3   1   0   0   1
4   0   0   0   0
```



10
1
4
2
3
7
11
2
3
2

```cpp
int main() {
    setlocale(LC_ALL, "Russian");
    {
        Graph<int> graph;
        int amountVerts, amountEdges, vertex, sourceVertex, targetVertex, edgeWeight;
        cout << "Введите количество вершин графа: "; cin >> amountVerts; cout << endl;
        cout << "Введите количество ребер графа: "; cin >> amountEdges; cout << endl;
        for (int i = 0; i < amountVerts; ++i) {
            cout << "Вершина: "; cin >> vertex;
            graph.insertVertex(vertex);
                cout << endl;
        }
        for (int i = 0; i < amountEdges; ++i) {
            cout << "Исходная вершина: "; cin >> sourceVertex; cout << endl;
            cout << "Конечная вершина: "; cin >> targetVertex; cout << endl;
            cout << "Вес ребра: "; cin >> edgeWeight; cout << endl;
            graph.InsertEdge(sourceVertex, targetVertex, edgeWeight);
        }
        cout << endl;
        graph.Print();
    }
    _getch();
    return 0;
}
```

```
Введите количество вершин графа: 4

Введите количество ребер графа: 6

Вершина: 1

Вершина: 2

Вершина: 3

Вершина: 4
```

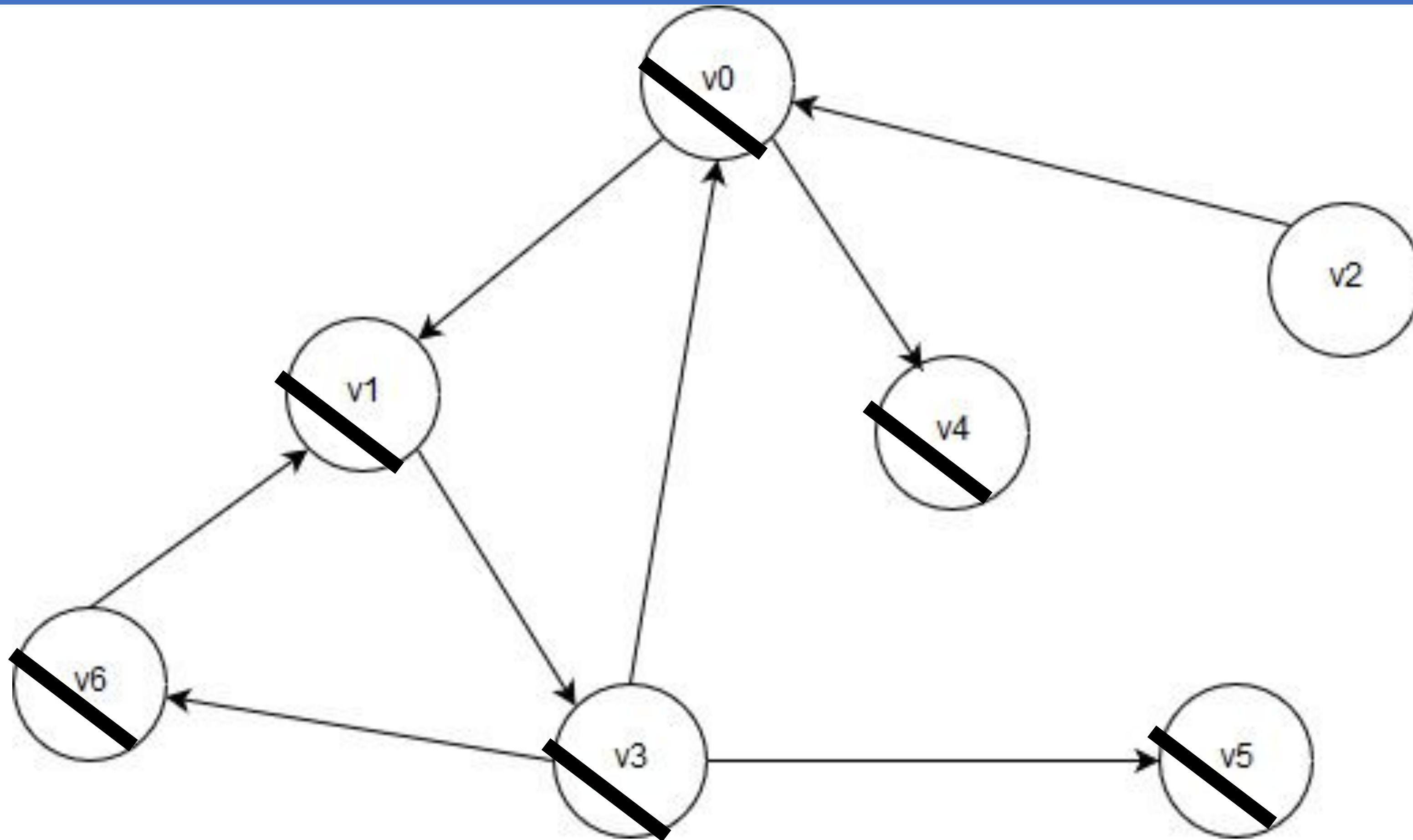
```
Исходная вершина: 1

Конечная вершина: 2

Вес ребра: 2

Исходная вершина: 1

Конечная вершина: 4

Вес ребра: 10

Исходная вершина: 2

Конечная вершина: 3

Вес ребра: 2

Исходная вершина: 3

Конечная вершина: 4

Вес ребра: 3

Исходная вершина: 3

Конечная вершина: 1

Вес ребра:
7

Исходная вершина: 2

Конечная вершина: 4

Вес ребра: 11
```

```
Матрица смежности графа:
-   1   2   3   4
1   0   2   7   10
2   2   0   2   11
3   7   2   0   3
4   10  11  3   0
```

## Методы обхода графов. Обход в глубину или правило левой руки

- Алгоритм обхода графа в глубину начинает выполнение с одной из вершин графа - начальной вершины, фиксирует информацию о посещении этой вершины, и, перемещаясь по ребру, посещает соседние вершины.
- Кроме того, правило левой руки при прохождении лабиринта (идти, ведя левой рукой по стенке) также является обходом в глубину. По завершении обхода все вершины окажутся пройдёнными - обработанными. Если при обходе встречается вершина, которая уже была пройдена, то повторной обработки делать не нужно.

# Обход в глубину



# Обход в глубину

```
template<class T>
void Graph<T>::DFS(T& startVertex, bool* visitedVerts)
{
    cout << "Вершина " << startVertex << " посещена" << endl;
    visitedVerts[this->GetVertPos(startVertex)] = true;
    std::vector<T> neighbors = this->GetNbrs(startVertex);
    for (int i = 0; i < neighbors.size(); ++i)
    {
        if (!visitedVerts[this->GetVertPos(neighbors[i])])
        {
            this->DFS(neighbors[i], visitedVerts);
        };
    };
}
```

# Обход в глубину

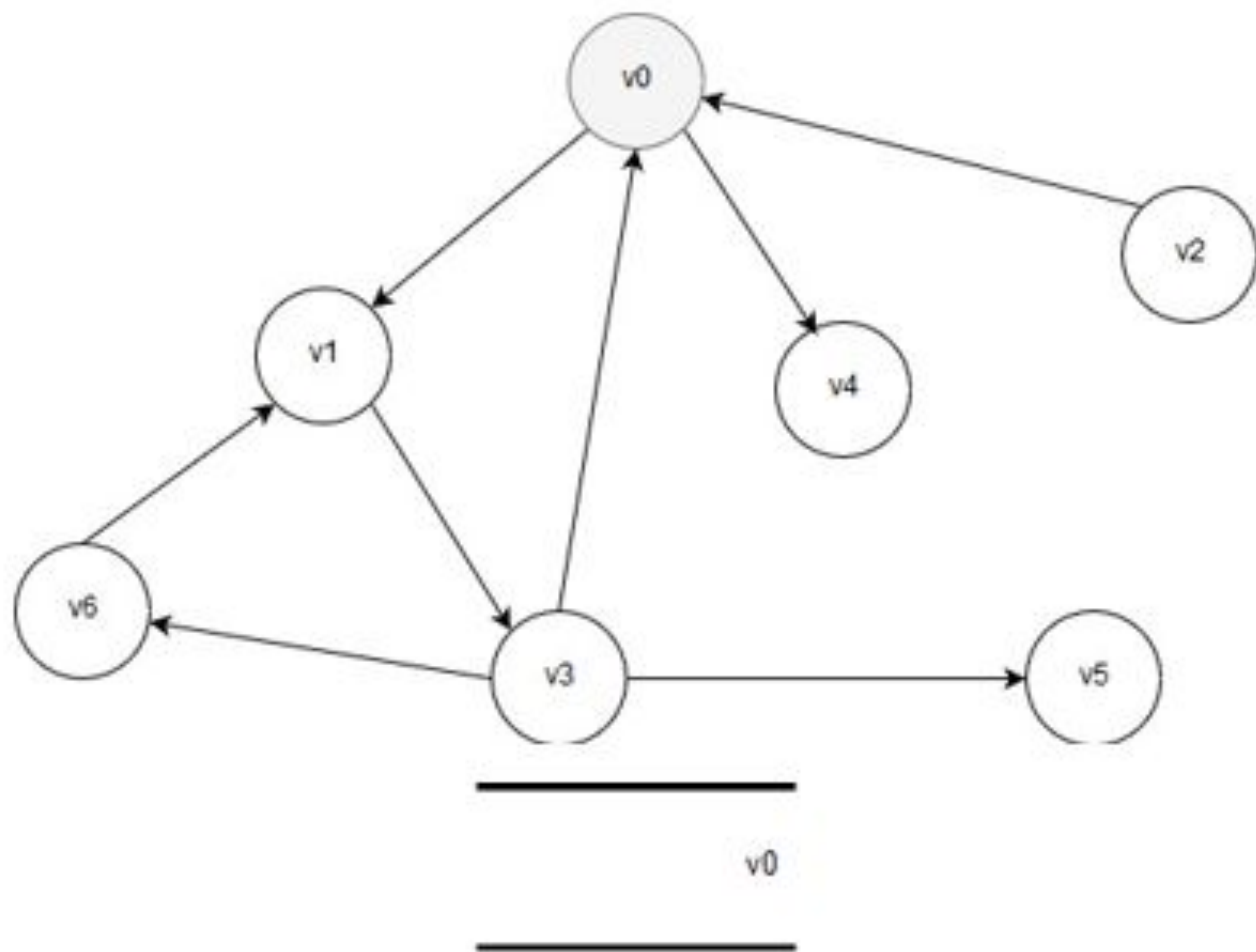
```
int main() {
    setlocale(LC_ALL, "Russian");
    bool* visitedVerts = new bool[20];
    fill(visitedVerts, visitedVerts + 20, false);

    Graph<int> graph;
    int amountVerts, amountEdges, vertex, sourceVertex, targetVertex;
    cout << "Введите количество вершин графа: "; cin >> amountVerts; cout << endl;
    cout << "Введите количество ребер графа: "; cin >> amountEdges; cout << endl;
    for (int i = 0; i < amountVerts; ++i) {
        cout << "Вершина: "; cin >> vertex;
        graph.insertVertex(vertex);
        cout << endl;
    }
    for (int i = 0; i < amountEdges; ++i)
    {
        cout << "Исходная вершина: "; cin >> sourceVertex; cout << endl;
        cout << "Конечная вершина: "; cin >> targetVertex; cout << endl;
        graph.InsertEdge(sourceVertex, targetVertex);
        cout << endl;
    }
        graph.Print();
        cout << "Введите вершину, с которой начать обход: "; cin >> vertex; cout << endl;
        graph.DFS(vertex, visitedVerts);
}
```

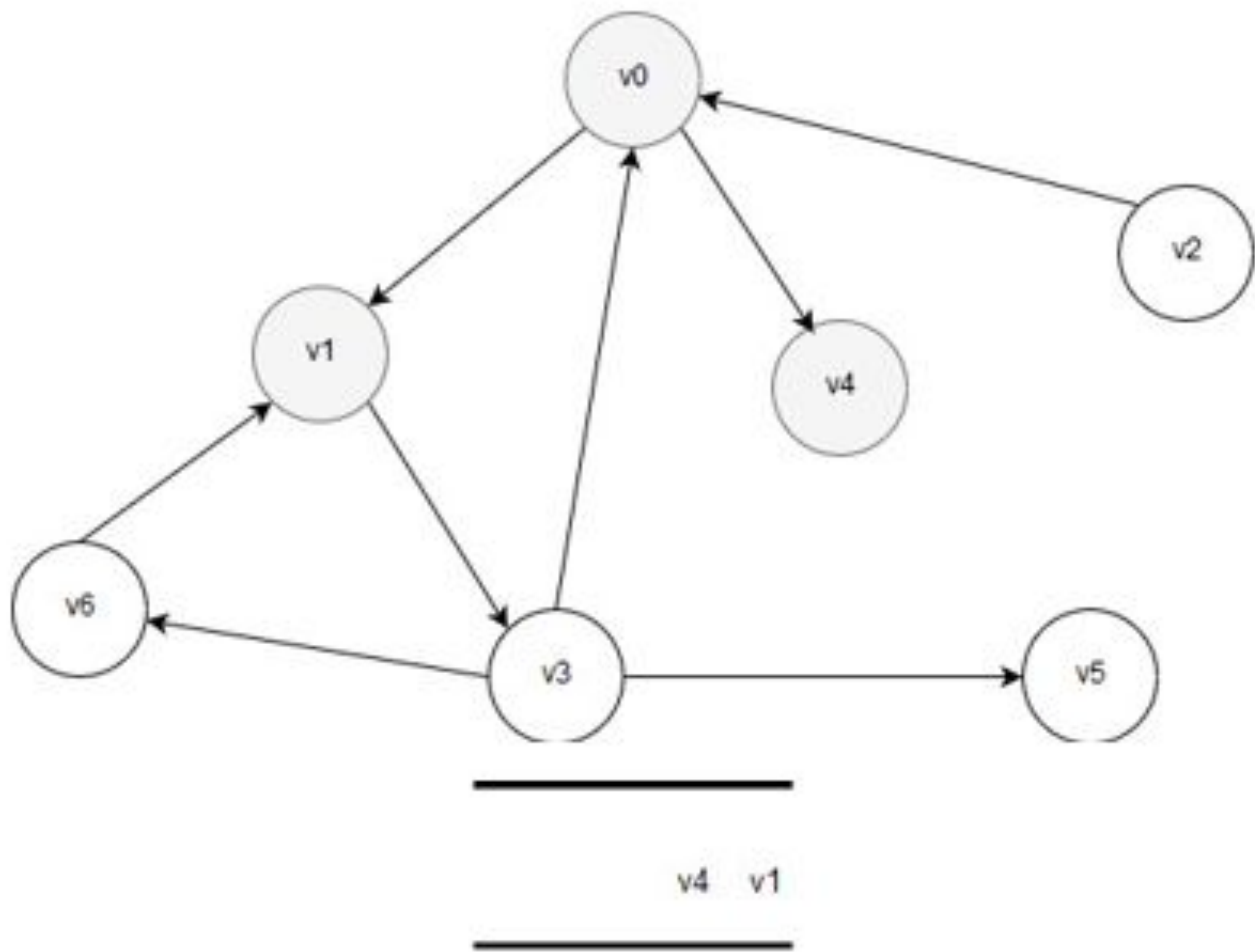
# Обход в глубину

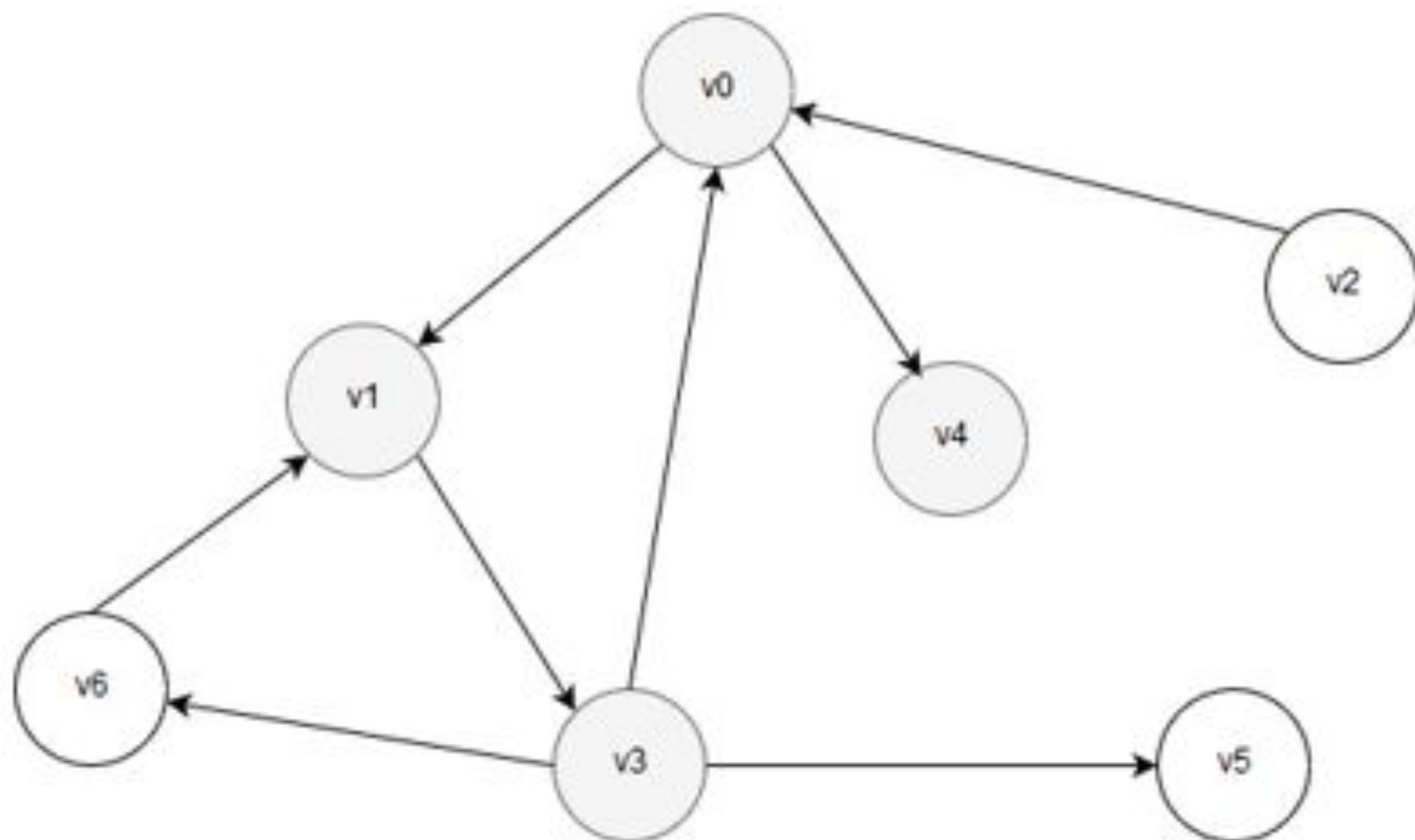
```
Матрица смежности графа:
-   0   1   2   3   4   5   6
0   0   1   0   0   1   0   0
1   0   0   0   1   0   0   0
2   1   0   0   0   0   0   0
3   1   0   0   0   0   1   1
4   0   0   0   0   0   0   0
5   0   0   0   0   0   0   0
6   0   1   0   0   0   0   0
Введите вершину, с которой начать обход: 0

Вершина 0 посещена
Вершина 1 посещена
Вершина 3 посещена
Вершина 5 посещена
Вершина 6 посещена
Вершина 4 посещена
```

# Методы обхода графов. Обход в ширину

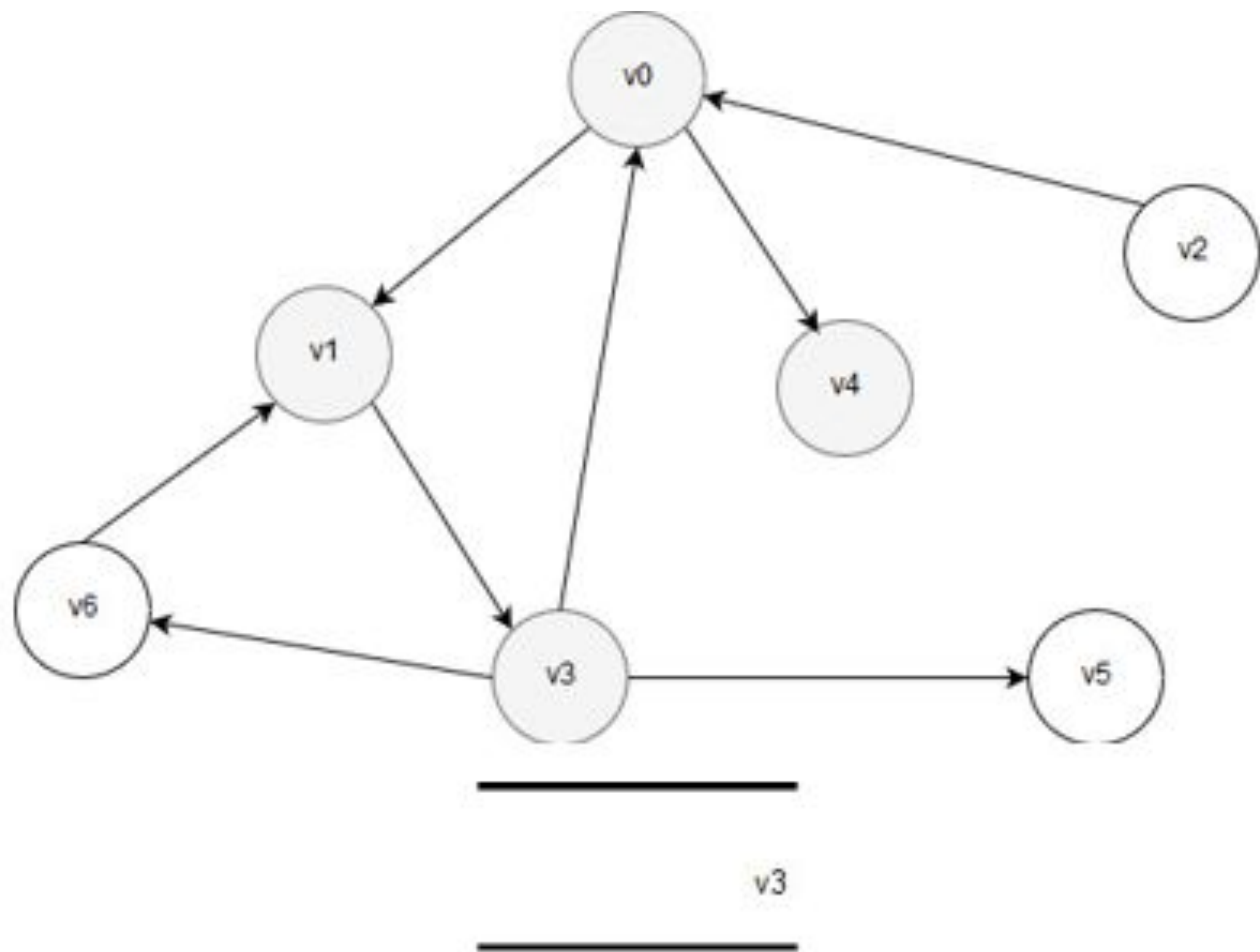
- Поиск начинается с начальной вершины, которая обрабатывается, маркируется и помещается в очередь.
- Основой алгоритма является циклический процесс, в котором обработанная вершина удаляется из очереди, а в очередь помещаются соседствующие с обработанной вершины. Таким образом, тело цикла состоит из двух основных шагов:
- Удалить вершину v из головы очереди
- Для каждой непомеченной вершины u, соседней по отношению к вершине v, обработать вершину u, маркировать ее и поместить в очередь (вершина u может

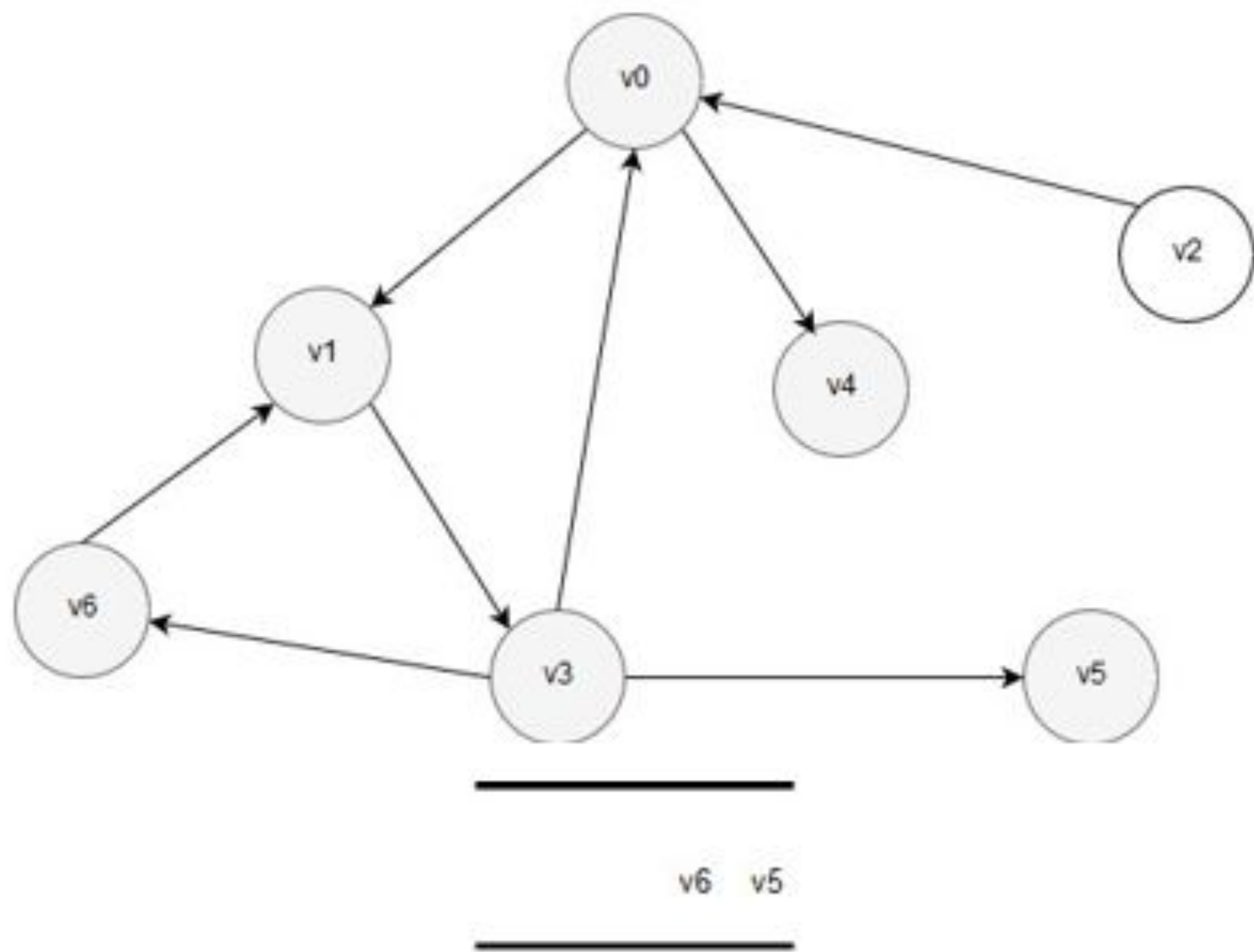

v0
v1
v2
v3
v4
v5
v6
v0


v0
v2
v1
v4
v6
v3
v5
v4 v1




v3 v4


v0
v2
v1
v4
v6
v3
v5
v3


v0
v2
v1
v4
v6
v3
v5
v6 v5

# Обход в ширину

```
template<class T>
void Graph<T>::BFS(T& startVertex, bool* visitedVerts) {
    //Условие истино только при первом вызове функции
    if (visitedVerts[this->GetVertPos(startVertex)] == false) {
        this->VertsQueue.push(startVertex);
        cout << "Вершина " << startVertex << " обработана" << endl;
        visitedVerts[this->GetVertPos(startVertex)] = true;
    }
    std::vector<T> neighbors = this->GetNbrs(startVertex);
    this->VertsQueue.pop();
    for (int i = 0; i < neighbors.size(); ++i) {
        if (!visitedVerts[this->GetVertPos(neighbors[i])])
        {
            this->VertsQueue.push(neighbors[i]);

            visitedVerts[this->GetVertPos(neighbors[i])] = true;
            cout << "Вершина " << neighbors[i] << " обработана" << endl;
        }
    }
    if (this->VertsQueue.empty())
    {
        return;
    };
    BFS(VertsQueue.front(),visitedVerts);
}
```

# Обход в ширину

```
Матрица смежности графа:
-   0   1   2   3   4   5   6
0   0   1   0   0   1   0   0
1   0   0   0   1   0   0   0
2   1   0   0   0   0   0   0
3   1   0   0   0   0   1   1
4   0   0   0   0   0   0   0
5   0   0   0   0   0   0   0
6   0   1   0   0   0   0   0
Введите вершину, с которой начать обход: 0

Вершина 0 обработана
Вершина 1 обработана
Вершина 4 обработана
Вершина 3 обработана
Вершина 5 обработана
Вершина 6 обработана
```