

ГРАФЫ

Способы представления графов

1. Матрица смежности

Таблица, где столбцы и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот).

Это наиболее удобный способ представления плотных неизменяемых графов.

Недостатки:

- если граф разрежён, то большая часть памяти будет напрасно тратиться на хранение нулей;
- изменение размера графа влечёт создание новой матрицы смежности;
- большие затраты памяти, прямо пропорциональные квадрату количества вершин.

Достоинства:

- простота и скорость поиска смежных вершин;
- простота и скорость модификации множества рёбер.

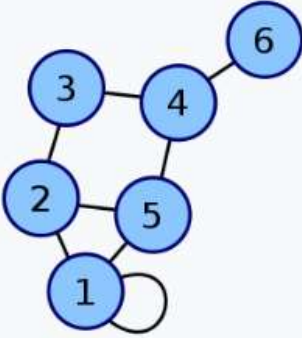
Граф	Матрица смежности
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

Рис. 1. Неориентированный граф и его матрица смежности.

Матрица смежности простого графа (не содержащего петель и кратных рёбер) является бинарной матрицей и содержит нули на главной диагонали.

Матрица смежности неориентированного графа симметрична.

Петли записываются в главную диагональ. Наличие кратных рёбер требует создания матрицы списков рёбер.

Ниже показаны матрицы смежности орграфа (рис. 2, а) и графа (рис. 2, б), а также представление их в программе.

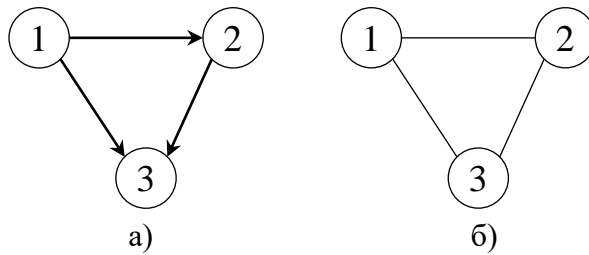


Рис. 2. Орграф (а) и граф (б)

Матрица смежности орграфа (рис. 2, а)

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

Представление в программе:

```
int[,] orgraph = { { 0, 1, 1 }, { 0, 0, 1 }, { 0, 0, 0 } };
```

Матрица смежности графа (рис. 2, б)

Граф б)	Вершина 1	Вершина 2	Вершина 3
Вершина 1	0	1	1
Вершина 2	1	0	1
Вершина 3	1	1	0

Представление в программе:

```
int[,] graph = { { 0, 1, 1 }, { 1, 0, 1 }, { 1, 1, 0 } };
```

Если граф взвешенный, то дуги/рёбра содержат веса.

Ниже показаны матрицы смежности взвешенных орграфа (рис. 3, а) и графа (рис. 3, б), а также представление их в программе.

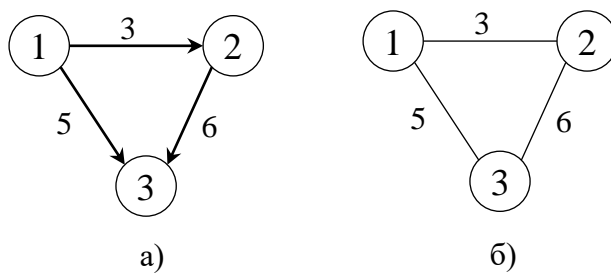


Рис. 3. Взвешенный орграф (а) и граф (б)

Матрица смежности взвешенного орграфа (рис. 3, а)

Граф а)	Вершина 1	Вершина 2	Вершина 3
Вершина 1	0	3	5
Вершина 2	0	0	6
Вершина 3	0	0	0

Представление в программе:

```
int[,] orgraph = { { 0, 3, 5 }, { 0, 0, 6 }, { 0, 0, 0 } };
```

Матрица смежности графа (рис. 3, б)

Граф б)	Вершина 1	Вершина 2	Вершина 3
Вершина 1	0	3	5
Вершина 2	3	0	6
Вершина 3	5	6	0

Представление в программе:

```
int[,] graph = { { 0, 3, 5 }, { 3, 0, 6 }, { 5, 6, 0 } };
```

2. Матрица инцидентности

Таблица, где строки соответствуют вершинам графа, а столбцы соответствуют связям (рёбрам) графа. В ячейку матрицы на пересечении строки i со столбцом j записывается

- 1 – в случае, если связь j «выходит» из вершины i ;
- -1 – если связь «входит» в вершину;
- 0 – во всех остальных случаях (то есть если связь является петлёй или связь не инцидентна вершине).

В каждом столбце обязательно должны стоять не более двух единиц (если это ребро представляет собой петлю, то единица ставится напротив вершины, которой инцидентна петля). В случае ориентированного графа в столбце должны стоять 1 и -1.

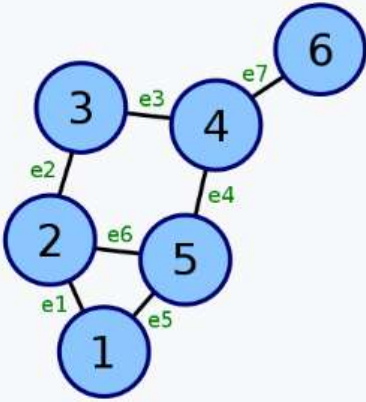
Граф	Матрица инцидентности
	$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$

Рис. 4. Неориентированный граф и его матрица инцидентности.

Недостатки:

- большие затраты памяти, пропорциональные $|V| \times |E|$;
- большая часть памяти напрасно тратится на хранение нулей;
- изменение размера графа влечёт создание новой матрицы;

Достоинства:

- быстрое нахождение циклов в графе;
- может использоваться для представления гиперграфов (в этом случае столбец может содержать больше двух единиц)

3. Коллекция списков смежности

Коллекция, где каждой вершине графа соответствует список смежных вершин (рис. 5). Такая структура данных представляет собой коллекцию списков.

Размер занимаемой памяти $O(|V| + |E|)$.

Это наиболее удобный способ для представления разреженных графов. Часто используется при реализации базовых алгоритмов обхода графа в ширину или глубину, где нужно быстро получать соседей текущей просматриваемой вершины.

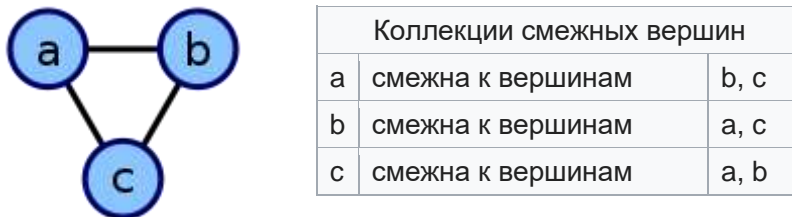


Рис. 5. Граф и его список смежных вершин

Недостатки:

- большое время и размер кода при модификации графа – как размера графа, так и множества рёбер.

Достоинства:

- экономное расходование памяти.

Способы реализации коллекции списков смежности:

3.1. Список списков смежных вершин. Структура `m` содержит вершину с именем `Name` и список смежных к ней вершин `Neigs`. Переменная `graph` содержит список структур `m`. Также в этом примере приведён способ построения пути из вершины `a`.

```
struct M
{
    public char Name;
    public List<char> Neigs;
}
private static void Main(string[] args)
{
    List<M> graph = new List<M>();
    graph.Add(new M { Name = 'a', Neigs = new List<char>() { 'b', 'c' } });
    // список вершин, смежных к A
    graph.Add(new M { Name = 'b', Neigs = new List<char>() { 'c' } }); //
    // список вершин, смежных к B
    graph.Add(new M { Name = 'c', Neigs = new List<char>() { } }); // список
    // вершин, смежных к C

    M x = graph.Find(z => z.Name == 'a'); // поиск вершины x1, смежной к A
    char x1 = x.Neigs[0]; // имя вершины x1
    M y = graph.Find(z => z.Name == x1); // поиск вершины y1, смежной к x1
    char y1 = y.Neigs[0]; // имя вершины y1

    Console.WriteLine("a -> {0} -> {1}", x1, y1); // вывод пути из A длиной 2
    Console.ReadKey();
}
```

Результат: a -> b -> c

3.2. Хэш-таблица для ассоциации каждой вершины со списком смежных вершин. Нет явного представления рёбер в этой структуре.

Хэш-таблица		
h(a)	→	b, c
h(b)	→	a, c
h(c)	→	a, b

3.3. Массив списков смежных вершин. Вершины графа являются числовыми индексами массива. Для этого способа необходимо вершины графа представлять числами, так как они служат индексами массива. Либо вершину следует представлять структурой, в которой одним из полей является уникальный числовой идентификатор.

Индекс (номер вершины)	1	2	3
Список смежных вершин	2, 3	1, 3	1, 2

В этом примере представлен граф (рис. 2 а) и способ нахождения пути длиной 2 из вершины 1.

```
List<int>[] m = new List<int>[4];  
m[1] = new List<int>() { 2, 3 };  
m[2] = new List<int>() { 3 };  
m[3] = new List<int>() { };
```

```
int x = m[1][0]; // получаем номер вершины x, смежной к первой  
int y = m[x][0]; // получаем номер вершины y, смежной к x  
Console.WriteLine("1 -> {0} -> {1}",x,y); // вывод пути из вершины 1 длиной 2  
Console.ReadKey();
```

Результат: 1 -> 2 -> 3

3.4. Объектно-ориентированный список вершин-объектов со ссылками между ними

Вершины – экземпляры класса `Node<T>`, дуги – ссылки на эти экземпляры.

Достоинства:

- возможность быстрой модификации как множества вершин, так и рёбер.
- такое представление годится для графов любого типа – псевдографов, мультиграфов и т.п.

Класс `Node<T>` описывает вершины графа. Вершина графа имеет идентификатор типа `T`, список ссылок на смежные вершины `Neighbors`, а также конструктор вершины.

```
public class Node<T>  
{  
  
    public T Name; // Идентификатор вершины Value  
    // Список идентификаторов смежных вершин (если список пустой, то null)  
    public LinkedList<Node<T>> Neighbors = new LinkedList<Node<T>>();  
    // Конструктор новой вершины  
    public Node(T name) { Name = name; }  
}
```

Класс `Graph<T>` хранит граф в виде списка вершин `LLN (LinkedListNode)`. Этот класс позволяет добавлять новую уникальную вершину с помощью метода

AddNode(T name), а также добавлять одно или несколько дуг с помощью методов AddEdge(T FromName, T ToName) и AddEdges(T FromName, T[] ToNames) соответственно. Кроме этого имеется метод вывода на экран PrintNeighbors(T name) всех смежных вершин для заданной по имени вершины name.

```
class Graph<T>
{
    public LinkedList<Node<T>> LLN;
    public Graph() { LLN = new LinkedList<Node<T>>(); }
    public void AddNode(T name) // добавление уникальной вершины
    {
        foreach (Node<T> n in LLN) if (n.Name.Equals(name)) return;
        var node = new Node<T>(name);
        LLN.AddLast(node);
    }
    public void AddEdge(T FromName, T ToName)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (Node<T> b in LLN)
                {
                    if (b.Name.Equals(ToName)) a.Neighbors.AddLast(b);
                }
                return;
            }
        }
    }
    public void AddEdges(T FromName, T[] ToNames)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (T ToName in ToNames)
                {
                    foreach (Node<T> b in LLN)
                    {
                        if (b.Name.Equals(ToName)) a.Neighbors.AddLast(b);
                    }
                }
                return;
            }
        }
    }
    public void PrintNeighbors(T name)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(name))
            {
                foreach (Node<T> m in a.Neighbors)
                {
                    Console.Write(m.Name + ", ");
                }
                return;
            }
        }
    }
}
```

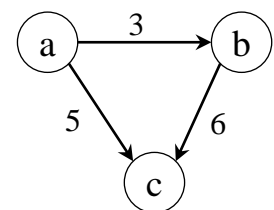


Рис. 6. Опрграф

Пример построения орграфа с тремя вершинами (рис. 6). Вначале создаются все три вершины, потом они соединяются дугами. В конце примера на экран выводится имя вершины, добавленной первой, потом все смежные вершины для вершины 'a', а также путь длиной 2 из вершины, которая была добавлена первой, по первым ссылкам списков смежных вершин.

```
var G = new Graph<char>();
G.AddNode('a');
G.AddNode('b');
G.AddNode('c');
G.AddEdges('a', new char[] { 'b', 'c' });
G.AddEdge('b', 'c');

Console.WriteLine(G.LLN.First.Value.Name);
G.PrintNeighbors('a');
Console.WriteLine();

Node<char> A = G.LLN.First.Value;
Console.Write(A.Name + " -> ");
Node<char> B = A.Neighbors.First.Value;
Console.Write(B.Name + " -> ");
Node<char> C = B.Neighbors.First.Value;
Console.WriteLine(C.Name);
```

```
Console.ReadKey();
```

С целью упрощения в этом примере не выполняется проверка пустоты списка смежных вершин.

Результат:

```
a
b, c,
a -> b -> c
```

3.5. Объектно-ориентированный список смежности содержит специальные классы вершин и рёбер. Каждый объект вершины содержит ссылку на коллекцию инцидентных рёбер. Каждый объект ребра содержит ссылки на исходящую и входящую вершины.

4. Список рёбер

Список, где каждому ребру графа соответствует кортеж/структура двух вершин, инцидентных ребру.

Размер занимаемой памяти $O(|E|)$.

Недостатки:

- трудность поиска смежных вершин;
- для представления неориентированных графов нужно либо удваивать список рёбер, либо делать функцию транзитивного замыкания, что увеличит время поиска смежных вершин.

Достоинство:

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

В следующем коде представлена структура `Edge<T>`, описывающая дугу, где `T` – тип данных имени вершины.

```
struct Edge<T>
{
    public T NodeA;
    public T NodeB;
}
```

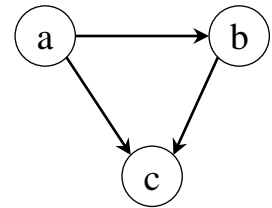


Рис. 7. Орграф

Пример описания орграфа (рис. 7) в виде списка дуг, а также способа поиска пути длиной 2 из вершины `a`.

```
List<Edge<char>> graph = new List<Edge<char>>();
graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'b' }); // дуга a-b
graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'c' }); // дуга a-c
graph.Add(new Edge<char> { NodeA = 'b', NodeB = 'c' }); // дуга b-c
```

```
Edge<char> e1 = graph.Find(x => x.NodeA == 'a'); // поиск начала дуги e1,
инцидентного вершине A
char x1 = e1.NodeB; // имя вершины, инцидентной концу дуги e1
Edge<char> e2 = graph.Find(x => x.NodeA == x1); // поиск начала дуги e2,
инцидентного вершине x1
char y1 = e2.NodeB; // имя вершины, инцидентной концу дуги e2
```

```
Console.WriteLine("a -> {0} -> {1}", x1, y1);
Console.ReadKey();
```

Результат: `a -> b -> c`

Пример функции транзитивного замыкания `TransClose` при использовании неориентированного графа (рис. 8), описанного списком дуг, а также способа поиска пути длиной 2 из вершины `c`. Транзитивное замыкание восстанавливает ребро имея одну дугу.

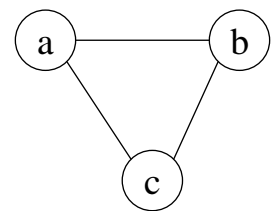


Рис. 8.

Неориентированный граф

```
struct Edge<T>
{
    public T NodeA;
    public T NodeB;
}

private static char TransClose(List<Edge<char>> graph, char node)
{
    // получение ребра, инцидентного вершине node
    Edge<char> e = graph.Find(x => (x.NodeA == node | x.NodeB == node));
    // поиск вершины, инцидентной вершине node
    if (e.NodeA == node) return e.NodeB;
    else return e.NodeA;
}

private static void Main(string[] args)
{
    List<Edge<char>> graph = new List<Edge<char>>();
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'b' }); // дуга a-b
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'c' }); // дуга a-c
    graph.Add(new Edge<char> { NodeA = 'b', NodeB = 'c' }); // дуга b-c

    char x = TransClose(graph, 'c'); // получение вершины, смежной к 'c'
    char y = TransClose(graph, x); // получение вершины, смежной к x
```



```

        Console.WriteLine("c -> {0} -> {1}", x, y);
        Console.ReadKey();
    }
    Результат: c -> a -> b

```

5. Граф на координатной сетке с подвижными вершинами

Вершины графа на координатной сетке (рис. 9) описываются классом с двумя дополнительными полями для координат X и Y:

```

public class Node<T>
{
    // Идентификатор вершины Value
    public T Name;
    public int X; // Координата x
    public int Y; // Координата y
    // Список идентификаторов смежных вершин (если список пустой, то null)
    public LinkedList<Node<T>> Neighbors = new LinkedList<Node<T>>();
    // Конструктор новой вершины со значениями Value
    public Node(T name, int x, int y) { Name = name; X = x; Y = y; }
}

```

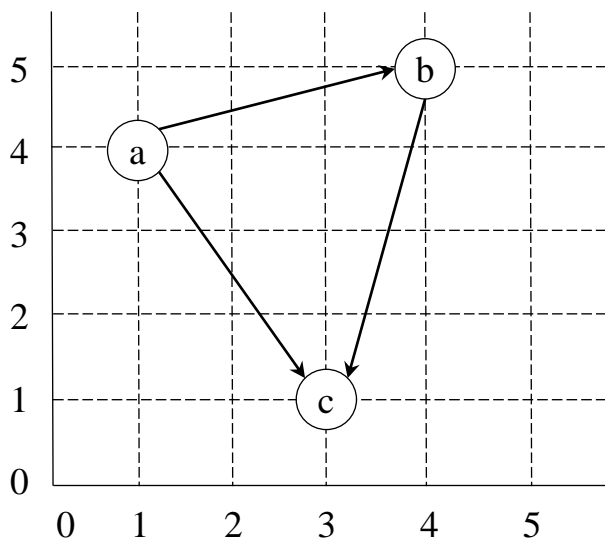


Рис. 9. Орграф на координатной сетке

Класс `Graph<T>`, также как и в п.3.4, хранит граф в виде списка вершин `LLN` (`LinkedListNode`). Этот класс позволяет добавлять новую уникальную вершину с помощью метода `AddNode(T name, int x, int y)`, а также добавлять одно или несколько дуг с помощью методов `AddEdge(T FromName, T ToName)` и `AddEdges(T FromName, T[] ToNames)` соответственно. Метод `MoveNode(T Name, int x, int y)` перемещает вершину на новые координаты. Кроме этого имеется метод вывода на экран `PrintNeighbors(T name)` всех смежных вершин для вершины, заданной по имени `name`.

```

class Graph<T>
{
    public LinkedList<Node<T>> LLN;
    public Graph() { LLN = new LinkedList<Node<T>>(); }
    public void AddNode(T name, int x, int y) // добавление уникальной вершины
    {

```

```

        foreach (Node<T> n in LLN) if (n.Name.Equals(name)) return;
        var node = new Node<T>(name, x, y);
        LLN.AddLast(node);
    }
    public void AddEdge(T FromName, T ToName)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (Node<T> b in LLN)
                {
                    if (b.Name.Equals(ToName)) a.Neighbors.AddLast(b);
                }
                return;
            }
        }
    }
    public void AddEdges(T FromName, T[] ToNames)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (T ToName in ToNames)
                {
                    foreach (Node<T> b in LLN)
                    {
                        if (b.Name.Equals(ToName)) a.Neighbors.AddLast(b);
                    }
                }
                return;
            }
        }
    }
    public void MoveNode(T name, int x, int y)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(name)) { a.X = x; a.Y = y; }
        }
    }
    public void PrintNeighbors(T name)
    {
        foreach (Node<T> a in LLN)
        {
            if (a.Name.Equals(name))
            {
                foreach (Node<T> m in a.Neighbors)
                {
                    Console.Write(m.Name + ", ");
                }
                return;
            }
        }
    }
}

```

Пример построения графа изображённого на рис. 9:

```

var G = new Graph<char>();
G.AddNode('a', 1, 4);
G.AddNode('b', 4, 5);
G.AddNode('c', 3, 1);
G.AddEdges('a', new char[] { 'b', 'c' });

```

```
G.AddEdge('b', 'c');
```

```
Console.WriteLine("Первая вершина графа: " + G.LLN.First.Value.Name);  
Console.Write("Её соседи: ");  
G.PrintNeighbors(G.LLN.First.Value.Name);
```

```
Console.ReadKey();
```

Результат:

Первая вершина графа: a
Её соседи: b, c,

6. Ad-hoc сети

Ad-hoc сети – подвижные вершины с ограничением на расстояние между вершинами. При превышении – связь теряется и граф разделяется на несвязанные компоненты. Используются для моделирования сети беспроводной мобильной связи.

Задачи ЛР:

1. Написать функцию определения расстояния между двумя вершинами графа, заданного на координатной сетке (п. 5).
2. Написать метод вывода на экран информации обо всех вершинах ориентированного графа: имя, координаты, смежные вершины, в которые можно перейти из данной вершины (п. 5).
3. Написать метод вывода на экран информации обо всех вершинах неориентированного графа: имя, координаты, смежные вершины (п. 5).

КР:

- 1.1. Разработать модель ad-hoc сети, содержащей n вершин, и работающей в границах прямоугольника с размерами a и b . Перемещая случайным образом вершины, определить зависимость числа компонент связности сети от параметра n .
- 1.2. Разработать алгоритм «разумного» равномерного перемещения вершин ad-hoc сети, при котором направление движения каждый раз выбирается в зависимости от текущего направления движения. Величина угла изменения направления движения определяется зависимостью: чем больше угол, тем меньше вероятность его появления, и наоборот. Можно использовать закон Пуассона.

Хранение графов на внешних носителях