

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

ОТЧЕТ

По лабораторной работе №10
«Поиск расстояний во взвешенном графе»
По дисциплине «Л и ОА в ИЗ»

Выполнили: ст. гр. 22ВВ4
Жуков Илья
Чумаев Сабит

Приняли: Юрова О.В.
Акифьев И.В.

Цель работы:

Написать код программы, выполнив следующие задания:

По заданию 1:

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного взвешенного графа G . Выведите матрицу на экран.
2. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки C++.
- 3.* Сгенерируйте (используя генератор случайных чисел) матрицу смежности для ориентированного взвешенного графа G . Выведите матрицу на экран и осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием.

По заданию 2:

1. Для каждого из вариантов сгенерированных графов (ориентированного и не ориентированного) определите радиус и диаметр.
2. Определите подмножества периферийных и центральных вершин.

По заданию 3:

1. Модернизируйте программу так, чтобы получить возможность запуска программы с параметрами командной строки (см. описание ниже). В качестве параметра должны указываться тип графа (взвешенный или нет) и наличие ориентации его ребер (есть ориентация или нет).

Ход работы:

Описание кода программы по заданию 1.1 - 1.2:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LR10

{
```

```

internal class Program
{
    static void Main(string[] args)
    {
        Console.Write("Введите размер матрицы(неориентированной):");
        int size = Convert.ToInt32(Console.ReadLine());

        int[,] adjacencyMatrix = GenerateAdjacencyMatrix(size);

        Console.WriteLine("Матрица смежности для
неориентированного графа:");
        PrintMatrix(adjacencyMatrix);

        Console.Write("Введите вершину, с которой начать обход:");
        int startVertex = Convert.ToInt32(Console.ReadLine());

        int[] distances = CalculateDistances(adjacencyMatrix,
startVertex);

        Console.WriteLine("Расстояния от исходной вершины:");
        for (int i = 0; i < distances.Length; i++)
        {
            if (distances[i] > 0)
            {
                Console.WriteLine("Исходная -> " + i + ": " +
distances[i]);
            }
            else
            {
                Console.WriteLine("Исходная -> " + i + ": ");
            }
        }
    }
}

```

```

    }
}
}

```

//метод генерирует случайную матрицу смежности

```

private static int[,] GenerateAdjacencyMatrix(int size)
{

```

```

    Random r = new Random();

```

//Создается двумерный массив matrix размером size x size, представляющий матрицу смежности для графа.

//Все элементы массива инициализируются значением 0.

```

    int[,] matrix = new int[size, size];

```

//Запускается первый цикл for, который итерируется по каждой строке матрицы (вершине).

```

    for (int i = 0; i < size; i++)
    {

```

//Внутри первого цикла запускается второй цикл for, который итерируется по каждому столбцу матрицы, начиная с индекса, следующего за текущим (i+1). Это делается для того, чтобы избежать повторения и дублирования ребер.

```

        for (int j = i+1; j < size; j++)
        {

```

//Внутри второго цикла проверяется, если текущая строка (i) не равна текущему столбцу (j), то выполняются следующие действия

```

            if (i != j)
            {

```

//Генерируется случайное весовое значение ребра от 1 до 10 с помощью метода r.Next(1, 10).

```

                matrix[i, j] = r.Next(1,10);
                matrix[j, i] = matrix[i, j];
            }
        }
    }
}

```

```

        }
    }
}

return matrix;
}

//выводит матрицу на экран
static void PrintMatrix(int[,] matrix)
{
    int size = matrix.GetLength(0);

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            Console.Write(matrix[i, j] + " ");
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}

//алгоритм поиска кратчайших расстояний в взвешенном
неориентированном графе.

private static int[] CalculateDistances(int[,]
adjacencyMatrix, int startVertex)
{
    int size = adjacencyMatrix.GetLength(0);
    int[] distances = new int[size];

    //Затем инициализируется массив `distances` размером
`size`, в котором будет храниться результат - кратчайшие

    //расстояния от `startVertex` до остальных вершин.
Изначально все элементы массива устанавливаются в -1.

```

```

    for (int i = 0; i < size; i++)
    {
        distances[i] = -1;
    }

    //Далее, расстояние от `startVertex` до самого себя
    устанавливается равным 0 в массиве `distances`, а `startVertex`
    добавляется в очередь `queue` с помощью метода `Enqueue`.

    distances[startVertex] = 0;

    Queue<int> queue = new Queue<int>();
    queue.Enqueue(startVertex);

    //Пока очередь `queue` не станет пустой
    while (queue.Count > 0)
    {
        //извлекаем текущую вершину из очереди с помощью
        метода `Dequeue`.

        int currentVertex = queue.Dequeue();

        for (int i = 0; i < size; i++)
        {
            //Если между текущей вершиной `currentVertex` и
            вершиной `i` существует ребро (значение в
            `adjacencyMatrix[currentVertex, i]` не равно 0)

            //и расстояние до вершины `i` еще не было
            установлено (значение в `distances[i]` равно -1), то вершина `i`
            добавляется в очередь `queue`

            //и устанавливается кратчайшее расстояние до нее,
            равное сумме расстояния до текущей вершины `currentVertex` и веса
            ребра между ними `adjacencyMatrix[currentVertex, i]`.

            if (adjacencyMatrix[currentVertex, i] != 0 &&
                distances[i] == -1)
            {

```

```

        queue.Enqueue(i);

        distances[i] = distances[currentVertex] +
adjacencyMatrix[currentVertex, i];
    }
}
}
return distances;
}
}
}

```

Описание кода программы по заданию 1.3:

```

using System;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _1._3
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите размер матрицы(ориентированной):
");

            int size = Convert.ToInt32(Console.ReadLine());

            int[][] graph = GenerateWeightedGraph(size);

            Console.WriteLine("Матрица смежности для ориентированного
графа:");

            PrintMatrix(graph);

```

```

        Console.WriteLine();
        Console.Write("Введите вершину, с которой начать обход:
");

        int startVertex = Convert.ToInt32(Console.ReadLine());
        int[] distances = FindDistances(graph, startVertex);
        Console.WriteLine("Расстояния от исходной вершины:");

        //С помощью цикла for выводятся расстояния от заданной
        //вершины до каждой вершины графа. Выводится номер вершины и ее
        //расстояние от заданной вершины.
        for (int i = 0; i < distances.Length; i++)
        {
            Console.WriteLine("Исходная -> " + i + ": " +
distances[i]);
        }
    }

    //Данный код генерирует случайный взвешенный граф и возвращает
    //его в виде матрицы смежности.
    static int[][] GenerateWeightedGraph(int vertices)
    {
        Random random = new Random();

        //Создается двумерный массив graph размером vertices x
        //vertices.
        int[][] graph = new int[vertices][];

        //Запускается цикл for, который итерируется по каждой
        //вершине графа.
        for (int i = 0; i < vertices; i++)
        {
            //Внутри первого цикла создается одномерный массив
            //размером vertices,
            //который является строкой матрицы смежности для
            //текущей вершины.

```


//Таким образом, каждая вершина имеет свою строку в матрице смежности.

```
graph[i] = new int[vertices];
```

//Запускается второй цикл for, который итерируется по каждому столбцу матрицы (вершине).

```
for (int j = 0; j < vertices; j++)
```

```
{
```

//Внутри второго цикла проверяется, если текущая вершина (i) равна текущему столбцу (j), то весовое значение устанавливается равным 0. Это означает, что между вершиной и самой собой нет ребра.

```
if (i == j)
```

```
graph[i][j] = 0; // Нет петли
```

//Иначе, генерируется случайное весовое значение от 1 до 10 с помощью метода random.Next(1, 10), и это значение присваивается весовому значению ребра между вершинами i и j.

```
else
```

graph[i][j] = random.Next(1, 10); // Случайное весовое значение от 1 до 10

```
}
```

```
}
```

```
return graph;
```

```
}
```

//вывод матрицы на экран

```
static void PrintMatrix(int[][] matrix)
```

```
{
```

```
for (int i = 0; i < matrix.Length; i++)
```

```
{
```

```
for (int j = 0; j < matrix[i].Length; j++)
```

```
{
```

```
Console.Write(matrix[i][j] + " ");
```

```

        }

        Console.WriteLine();
    }
}

//метод реализует алгоритм поиска расстояний от заданной
вершины до всех остальных вершин графа.
static int[] FindDistances(int[][] graph, int startVertex)
{
    //Создаются переменные vertices для хранения количества
вершин графа
    int vertices = graph.Length;

    //distances для хранения расстояний от стартовой вершины
до остальных вершин
    int[] distances = new int[vertices];

    //visited для отслеживания посещенных вершин.
    bool[] visited = new bool[vertices];

    //С помощью цикла for инициализируются значения в массивах
distances и visited.

    //Для всех вершин значения расстояний устанавливаются
равными -1 (что означает недостижимость),
    //а значения в массиве visited устанавливаются равными
false (вершины не посещены).
    for (int i = 0; i < vertices; i++)
    {
        distances[i] = -1;
        visited[i] = false;
    }

    //Значение расстояния от стартовой вершины до самой себя
устанавливается равным 0, а сама вершина помечается как посещенная.

```

```

distances[startVertex] = 0;
visited[startVertex] = true;

//Создается очередь queue (тип Queue<int>), в которую
помещается стартовая вершина с помощью метода Enqueue.

Queue<int> queue = new Queue<int>();
queue.Enqueue(startVertex);

//запускается цикл while, который выполняется, пока
очередь не пуста.
while (queue.Count > 0)
{
    //Внутри цикла извлекается вершина из начала очереди с
помощью метода Dequeue и сохраняется в переменной currentVertex.
    int currentVertex = queue.Dequeue();

    //С помощью цикла for проверяются все вершины графа
    for (int i = 0; i < vertices; i++)
    {
        //Если существует ребро от currentVertex до
вершины i и вершина i не была посещена ранее, то вершина i добавляется
в очередь,

        //помечается как посещенная, а значение расстояния
до нее равно сумме расстояния до currentVertex и веса ребра между
этими вершинами.

        if (graph[currentVertex][i] > 0 && !visited[i])
        {
            queue.Enqueue(i);
            visited[i] = true;
            distances[i] = distances[currentVertex] +
graph[currentVertex][i];
        }
    }
}

```

```

        }
        return distances;
    }
}

```

Описание кода программы по заданию 2.1-2.2:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _3._1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите размер матрицы(ориентированной):");
            int size = Convert.ToInt32(Console.ReadLine());

            int[][] graph = GenerateWeightedGraph(size);

            Console.WriteLine("Матрица смежности для ориентированного графа:");
            PrintMatrix(graph);

            Console.WriteLine();
        }
    }
}

```

```

        Console.WriteLine("Введите вершину, с которой начать обход:");
    });

    int startVertex = Convert.ToInt32(Console.ReadLine());

    int[] distances = FindDistances(graph, startVertex); //
Starting vertex is 0

    Console.WriteLine("Расстояния от исходной вершины:");
    for (int i = 0; i < distances.Length; i++)
    {
        if (distances[i] > 0)
        {
            Console.WriteLine("Исходная -> " + i + ": " +
distances[i]);
        }
        else
        {
            Console.WriteLine("Исходная -> " + i + ": ");
        }
    }

    int radius = FindRadius(distances);
    int diameter = FindDiameter(distances);

    List<int> peripheralVertices =
FindPeripheralVertices(distances, diameter);

    List<int> centralVertices = FindCentralVertices(distances,
radius);

    Console.WriteLine();

    Console.WriteLine("Радиус: " + radius);
    Console.WriteLine("Диаметр: " + diameter);

    Console.WriteLine("Периферийные вершины: " +
String.Join(", ", peripheralVertices));

```

```

        Console.WriteLine("Центральные вершины: " + String.Join(",
", centralVertices));

        Console.Write("\n" + "Введите размер
матрицы(неориентированной): ");

        int size2 = Convert.ToInt32(Console.ReadLine());

        int[,] adjacencyMatrix = GenerateAdjacencyMatrix2(size2);

        Console.WriteLine("Матрица смежности для
неориентированного графа:");

        PrintMatrix2(adjacencyMatrix);

        Console.Write("Введите вершину, с которой начать обход:
");

        int sourceVertex = Convert.ToInt32(Console.ReadLine());

        int[] distances2 = DistanceSearch(adjacencyMatrix,
sourceVertex);

        Console.WriteLine("Расстояния от исходной вершины:");
        for (int i = 0; i < distances2.Length; i++)
        {
            if (distances2[i] > 0)
            {
                Console.WriteLine("Исходная -> " + i + ": " +
distances2[i]);
            }
            else
            {
                Console.WriteLine("Исходная -> " + i + ": ");
            }
        }

```

```

    }

    int radius2 = FindRadius(distances2);
    int diameter2 = FindDiameter(distances2);

    List<int> peripheralVertices2 =
FindPeripheralVertices(distances2, diameter2);

    List<int> centralVertices2 =
FindCentralVertices(distances2, radius2);

    Console.WriteLine();
    Console.WriteLine($"Радиус: {radius2}" + "\n");
    Console.WriteLine($"Диаметр: {diameter2}" + "\n");

    Console.WriteLine("Периферийные вершины: " + string.Join(",",
peripheralVertices2) + "\n");

    Console.WriteLine("Центральные вершины: " + string.Join(",",
centralVertices2) + "\n");
}

//генерирует случайный взвешенный граф с заданным количеством
вершин.

static int[][] GenerateWeightedGraph(int vertices)
{
    Random random = new Random();

    //объявляет двумерный целочисленный массив с именем graph,
с числом строк, равным параметру vertices.

    int[][] graph = new int[vertices][];

    for (int i = 0; i < vertices; i++)
    {
        //инициализирует новый целочисленный массив длиной
vertices в индексе i массива graph.

```

```

graph[i] = new int[vertices];

for (int j = 0; j < vertices; j++)
{
    if (i == j)
        graph[i][j] = 0; // Нет петли

    else
        graph[i][j] = random.Next(1, 10); // Случайное
        весовое значение от 1 до 10
}
}

return graph;
}

//метод генерирует случайную матрицу смежности
private static int[,] GenerateAdjacencyMatrix2(int size)
{
    Random r = new Random();

    //Создается двумерный массив matrix размером size x size,
    представляющий матрицу смежности для графа.

    //Все элементы массива инициализируются значением 0.

    int[,] matrix = new int[size, size];

    //Запускается первый цикл for, который итерируется по
    каждой строке матрицы (вершине).

    for (int i = 0; i < size; i++)
    {
        //Внутри первого цикла запускается второй цикл for,
        который итерируется по каждому столбцу матрицы, начиная с индекса,

```


следующего за текущим (i+1). Это делается для того, чтобы избежать повторения и дублирования ребер.

```
        for (int j = 0; j < size; j++)
        {
            //Внутри второго цикла проверяется, если текущая
строка (i) не равна текущему столбцу (j), то выполняются следующие
действия

            if (i != j)
            {
                ////Генерируется случайное весовое значение
ребра от 1 до 10 с помощью метода r.Next(1, 10).

                matrix[i, j] = r.Next(0, 10);
                matrix[j, i] = matrix[i, j];
            }
        }
    }
    return matrix;
}

//выводит матрицу на экран
static void PrintMatrix(int[][] matrix)
{
    for (int i = 0; i < matrix.Length; i++)
    {
        for (int j = 0; j < matrix[i].Length; j++)
        {
            Console.Write(matrix[i][j] + " ");
        }

        Console.WriteLine();
    }
}

//выводит матрицу на экран
```

```

static void PrintMatrix2(int[,] matrix)
{
    int size = matrix.GetLength(0);

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            Console.Write(matrix[i, j] + " ");
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}

//метод реализует алгоритм поиска расстояний от заданной
вершины до всех остальных вершин графа.

static int[] FindDistances(int[][] graph, int startVertex)
{
    //Создаются переменные vertices для хранения количества
вершин графа

    int vertices = graph.Length;

    //distances для хранения расстояний от стартовой вершины
до остальных вершин

    int[] distances = new int[vertices];

    //visited для отслеживания посещенных вершин.

    bool[] visited = new bool[vertices];

    //С помощью цикла for инициализируются значения в массивах
distances и visited.

```

//Для всех вершин значения расстояний устанавливаются равными -1 (что означает недостижимость),

//а значения в массиве visited устанавливаются равными false (вершины не посещены).

```
for (int i = 0; i < vertices; i++)
{
    distances[i] = -1;
    visited[i] = false;
}
```

//Значение расстояния от стартовой вершины до самой себя устанавливается равным 0, а сама вершина помечается как посещенная.

```
distances[startVertex] = 0;
visited[startVertex] = true;
```

//Создается очередь queue (тип Queue<int>), в которую помещается стартовая вершина с помощью метода Enqueue.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(startVertex);
```

//запускается цикл while, который выполняется, пока очередь не пуста.

```
while (queue.Count > 0)
{
```

//Внутри цикла извлекается вершина из начала очереди с помощью метода Dequeue и сохраняется в переменной currentVertex.

```
int currentVertex = queue.Dequeue();
```

//С помощью цикла for проверяются все вершины графа

```
for (int i = 0; i < vertices; i++)
{
```

//Если существует ребро от currentVertex до вершины i и вершина i не была посещена ранее, то вершина i добавляется в очередь,

//помечается как посещенная, а значение расстояния до нее равно сумме расстояния до currentVertex и веса ребра между этими вершинами.

```
        if (graph[currentVertex][i] > 0 && !visited[i])
        {
            queue.Enqueue(i);
            visited[i] = true;
            distances[i] = distances[currentVertex] +
graph[currentVertex][i];
        }
    }
}
```

```
    return distances;
```

```
}
```

//Метод FindRadius принимает на вход массив расстояний и возвращает наименьшее положительное значение из этого массива.

```
static int FindRadius(int[] distances)
```

```
{
```

//переменная radius, которая изначально устанавливается в максимально возможное значение типа int.

```
    int radius = int.MaxValue;
```

//Затем происходит итерация по каждому элементу входного массива.

```
    foreach (int distance in distances)
```

```
    {
```

//Если текущее расстояние больше 0 и меньше текущего значения переменной radius, оно присваивается переменной radius.

```
        if (distance > 0 && distance < radius)
```

```
        {
```

```
            radius = distance;
```

```
        }
```

```

    }

    return radius;
}

//Метод FindDiameter также принимает на вход массив расстояний
и возвращает наибольшее значение из этого массива.

static int FindDiameter(int[] distances)
{
    //переменная diameter, которая изначально устанавливается
    в 0

    int diameter = 0;

    //происходит итерация по каждому элементу входного
    массива.

    foreach (int distance in distances)
    {
        //Если текущее расстояние больше текущего значения
        переменной diameter, оно присваивается переменной diameter

        if (distance > diameter)
        {
            diameter = distance;
        }
    }

    return diameter;
}

//Метод FindPeripheralVertices принимает массив расстояний
distances и значение радиуса radius, а затем возвращает список вершин,
которые имеют расстояние равное указанному радиусу.

static List<int> FindPeripheralVertices(int[] distances, int
radius)
{
    //переменная peripheralVertices, которая инициализируется
    пустым списком List<int>

    List<int> peripheralVertices = new List<int>();

```

```

        for (int i = 0; i < distances.Length; i++)
        {
            //Если текущее значение расстояния distances[i] равно
            //указанному радиусу, то индекс i добавляется в список
            //peripheralVertices.

            if (distances[i] == radius)
            {
                peripheralVertices.Add(i);
            }
        }

        return peripheralVertices;
    }

    //Метод FindCentralVertices также принимает массив расстояний
    //distances и значение диаметра diameter, а затем возвращает список
    //вершин, которые имеют расстояние равное указанному диаметру.

    static List<int> FindCentralVertices(int[] distances, int
    diameter)
    {
        //переменная centralVertices, которая инициализируется
        //пустым списком List<int>.

        List<int> centralVertices = new List<int>();

        for (int i = 0; i < distances.Length; i++)
        {
            //Если текущее значение расстояния distances[i] равно
            //указанному диаметру, то индекс i добавляется в список centralVertices.

            if (distances[i] == diameter)
            {
                centralVertices.Add(i);
            }
        }

        return centralVertices;
    }

```

//метод выполняет поиск расстояний между вершинами в графе на основе матрицы смежности.

```
static int[] DistanceSearch(int[,] adjacencyMatrix, int sourceVertex)
```

```
{
```

```
    int size = adjacencyMatrix.GetLength(0);
```

//Сначала в методе определяется размерность матрицы смежности и инициализируются массивы distances и visited.

```
    int[] distances = new int[size];
```

```
    bool[] visited = new bool[size];
```

```
    Queue<int> queue = new Queue<int>();
```

//Массив distances инициализируется значением -1 для всех вершин, чтобы отметить, что расстояния еще не были вычислены.

```
    for (int i = 0; i < size; i++)
```

```
    {
```

```
        distances[i] = -1;
```

//Массив visited инициализируется значением false для всех вершин, чтобы отметить, что ни одна вершина еще не была посещена.

```
        visited[i] = false;
```

```
    }
```

//Затем расстояние от исходной вершины до себя устанавливается равным 0, вершина помечается как посещенная и добавляется в очередь.

```
    distances[sourceVertex] = 0;
```

```
    visited[sourceVertex] = true;
```

```
    queue.Enqueue(sourceVertex);
```

//Далее происходит обход графа в ширину с помощью очереди. Пока очередь не пуста, извлекается текущая вершина из очереди и производится обход всех смежных вершин.

```
    while (queue.Count > 0)
```

```
    {
```

```
        int currentVertex = queue.Dequeue();
```

```

        for (int i = 0; i < size; i++)
        {
            //Если для смежной вершины i расстояние от текущей
            //вершины currentVertex до i больше 0 и вершина i не была посещена,
            //то обновляется расстояние до вершины
            i(добавляется значение adjacencyMatrix[currentVertex, i]
            //к расстоянию до текущей вершины) и вершина i
            помечается как посещенная. После этого вершина i добавляется в
            очередь.

            if (adjacencyMatrix[currentVertex, i] > 0
            && !visited[i])
            {
                distances[i] = distances[currentVertex] +
                adjacencyMatrix[currentVertex, i]; // Обновляем расстояние
                visited[i] = true; // Помечаем вершину как
                посещенную
                queue.Enqueue(i); // Добавляем вершину в
                очередь
            }
        }

        return distances;
    }
}
}

```

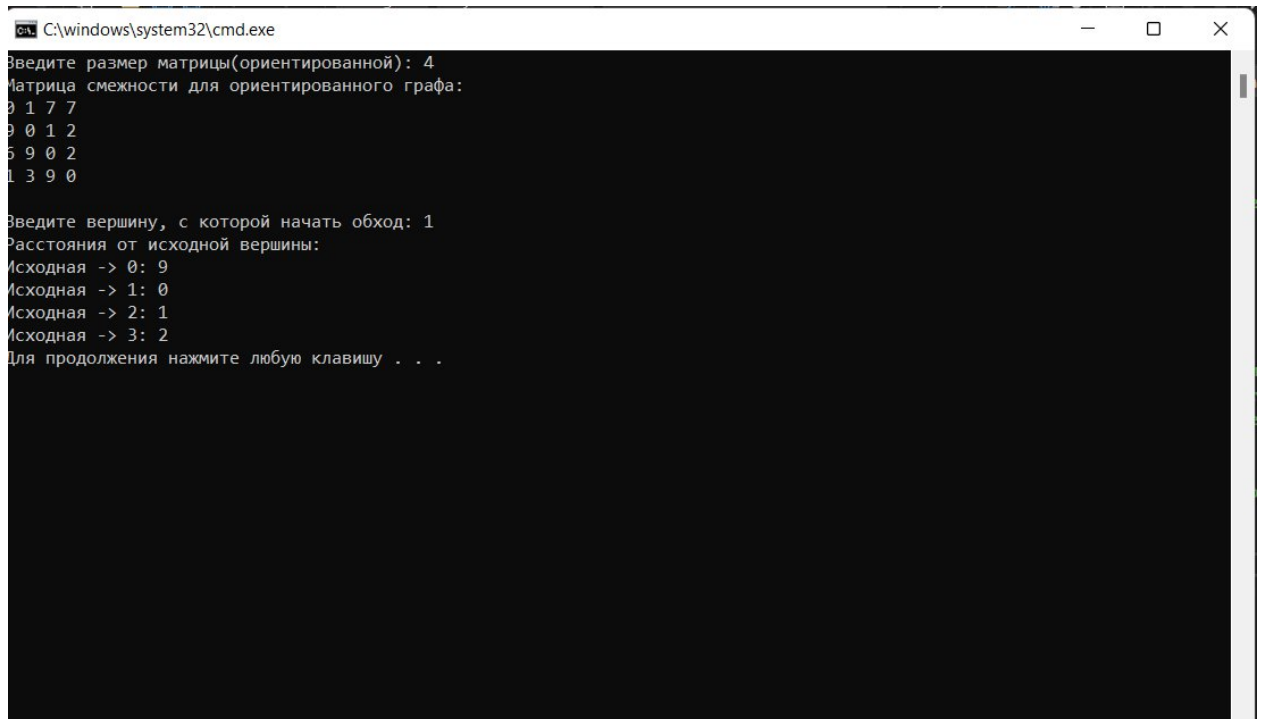

Результат работы программы 1.1-1.2:



```
C:\windows\system32\cmd.exe
Введите размер матрицы(неориентированной): 4
Матрица смежности для неориентированного графа:
0 1 2 4
1 0 1 9
2 1 0 2
4 9 2 0

Введите вершину, с которой начать обход: 1
Расстояния от исходной вершины:
Исходная -> 0: 1
Исходная -> 1:
Исходная -> 2: 1
Исходная -> 3: 9
Для продолжения нажмите любую клавишу . . .
```

Результат работы программы 1.3:



```
C:\windows\system32\cmd.exe
Введите размер матрицы(ориентированной): 4
Матрица смежности для ориентированного графа:
0 1 7 7
9 0 1 2
5 9 0 2
1 3 9 0

Введите вершину, с которой начать обход: 1
Расстояния от исходной вершины:
Исходная -> 0: 9
Исходная -> 1: 0
Исходная -> 2: 1
Исходная -> 3: 2
Для продолжения нажмите любую клавишу . . .
```

Результат работы программы 2.1-2.2:

```
C:\windows\system32\cmd.exe
Введите размер матрицы(ориентированной): 4
Матрица смежности для ориентированного графа:
0 6 7 2
4 0 7 1
9 9 0 2
4 3 9 0

Введите вершину, с которой начать обход: 1
Расстояния от исходной вершины:
Исходная -> 0: 4
Исходная -> 1:
Исходная -> 2: 7
Исходная -> 3: 1

Радиус: 1
Диаметр: 7
Периферийные вершины: 2
Центральные вершины: 3

Введите размер матрицы(неориентированной): 4
Матрица смежности для неориентированного графа:
0 3 7 4
3 0 0 6
7 0 0 8
4 6 8 0

Введите вершину, с которой начать обход: 1
Расстояния от исходной вершины:
Исходная -> 0: 3
Исходная -> 1:
Исходная -> 2: 10
Исходная -> 3: 6

Радиус: 3
Диаметр: 10
Периферийные вершины: 2
Центральные вершины: 0
Для продолжения нажмите любую клавишу . . . █
```

Вывод: в ходе лабораторной работы мы научились осуществлять поиск расстояния во взвешенном графе (ориентированном и неориентированном). А также определять радиус, диаметр, подмножества периферийных и центральных вершин графа.