# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4
# ГРАФЫ.DSF.BSF

# Оглавление

Описание задачи. ..... 3

Описание метода/модели. ..... 4

Выполнение задачи. ..... 5

Заключение. ..... 18

# Описание задачи.

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

- Максимальное/Минимальное количество генерируемых вершин
- Максимальное/Минимальное количество генерируемых ребер
- Максимальное количество ребер связанных с одной вершины
- Генерируется ли направленный граф
- Максимальное количество входящи и выходящих ребер

Сгенерированный граф должен быть описан в рамках одного класса(этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.

- Выдача матрицы смежности
- Выдача матрицы инцидентности
- Выдача список смежности
- Выдача списка ребер

# Описание метода/модели.

Под графом в математике понимается абстракция реальной системы объектов безотносительно их природы, обладающих парными связями.

Вершина графа – это некоторая точка, связанная с другими точками

Ребро графа – это линия, соединяющая две точки и олицетворяющая связь между ними

Граф – это множество вершин, соединённых друг с другом произвольным образом множеством ребер

Что описывает граф:

- Взаимоотношение между людьми (Социальные связи)
- Иерархические отношения (Подчиненность людей, подразделений и прочего)
- Пути перемещения в любой местности (Карта метро, сеть дорог)
- Взаимозависимости поставщиков услуг или товаров (Поставщики для сборки одного автомобиля)
- Распределенные системы (Любая микросервисная архитектура)

## Ориентированный граф

Связанную с каким либо ребром вершину называются инцидентной, это такая вершина которая каким либо образом принадлежит ребру.

Что описывает ориентированный граф:

- Пути распространения информации между людьми (Социальные связи)
- Пути решения и эскалации проблемы в системах ведения задач
- Пути перемещения в любой местности(Карта метро, сеть дорог)
- Предоставления товаров и услуг различным контрагентам (Поставщики для сборки одного автомобиля)
- Распространение ошибки в сложных системах
- Пути распространения расчета в нейронных сетях

## Описание графа

Алгоритм удаления выглядит следующим образом:

- Матрица смежности, это двумерная таблица, для которой столбцы и строки соответствуют вершинам, а значения в таблицы соответствуют ребрам, для невзвешенного графа они могут быть просто 1 если связь есть и идет в нужном направлении и 0 если ее нет, а для взвешенного графа будут стоять конкретные значения.
- Матрица инциндетности, это матрица, в которой строки соответствуют вершинам, а столбцы соответствуют связям, и ячейки ставиться 1 если связь выходит из вершины, -1 если входит и 0 во всех остальных случаях.
- Список смежности, это список списков, содержащий все вершины, а внутренние списки для каждой вершины содержат все смежные ей.
- Список ребер, это список строк в которых хранятся все ребра вершины, а внутренние значение содержит две вершины к которым присоединено это ребро.

## Обход графа

Обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Выделяют два вариант обхода, обход в глубину(или поиск в глубину, DFS) и обход в ширину(или поиск в ширину, BFS)

**DFS (Deep first search)** следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.

**DFS (Deep first search)** следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.

## Поиск в глубину

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2. Запускаем процедуру dfs(u)
3. Помечаем вершину u как пройденную
4. Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v)
5. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

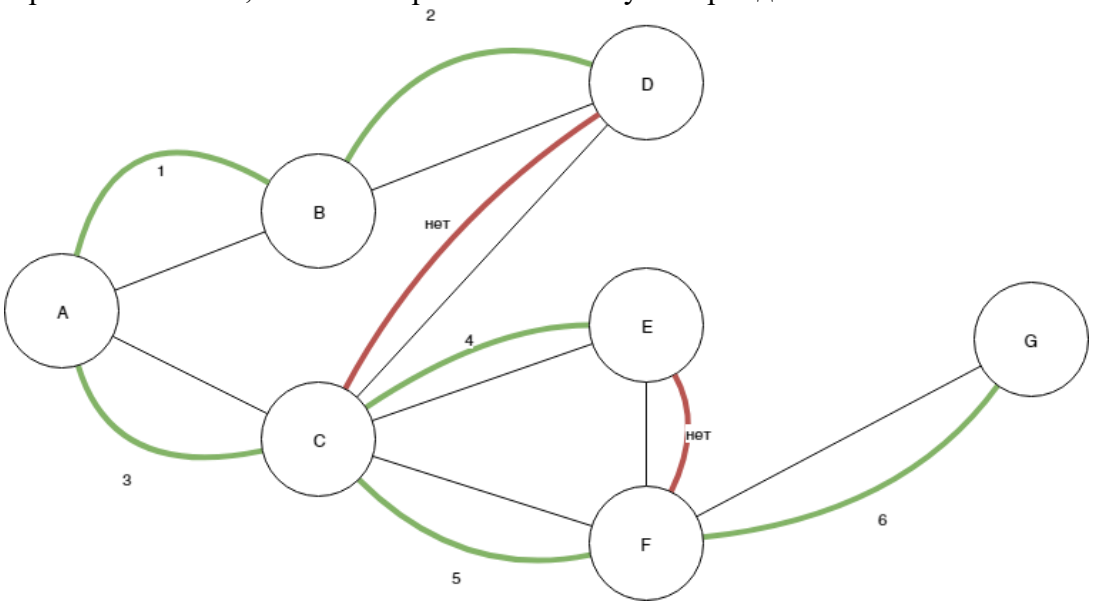


Рис.1 Поиск в глубину.

Сложность алгоритма определяется количеством вершин и количеством ребер в графе, вся процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

## Поиск в ширину

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развёрнутый.
3. Если узел u является целевым узлом, то завершить поиск с результатом «успех».
4. В противном случае, в конец очереди добавляются все преемники узла u, которые ещё не развёрнуты и не находятся в очереди.
5. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
6. Вернуться к п. 2.

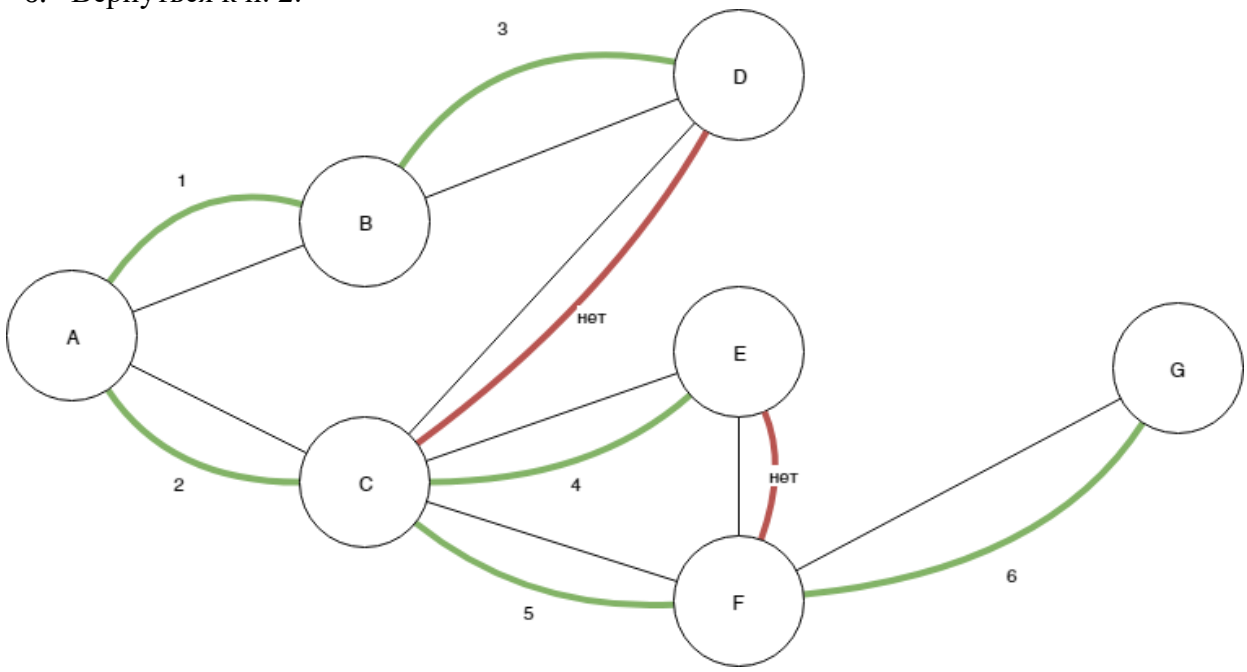


Рис.2 Поиск в ширину.

Сложность алгоритма определяется количеством вершин и количеством ребер в графе, вся процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

## Поиск в ширину: применение

- Волновой алгоритм поиска пути в лабиринте
- Волновая трассировка печатных плат
- Поиск компонент связности в графе
- Поиск кратчайшего пути между двумя узлами невзвешенного графа
- Поиск в пространстве состояний: нахождение решения задачи с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа
- Нахождение кратчайшего цикла в ориентированном невзвешенном графе
- Нахождение всех вершин и рёбер, лежащих на каком-либо кратчайшем пути между двумя вершинами a и b
- Поиск увеличивающего пути в алгоритме Форда-Фалкерсона (алгоритм Эдмондса-Карпа)

## Различия двух алгоритмов

Обход графа в ширину и в глубину имеет общие цели, но разные характеристики, вот несколько из них:

- Поиск в ширину и в глубину — это и есть обход графа.
- Поиск в глубину — это поиск по ребрам графа туда-обратно, а поиск в ширину — это плавный «обход по соседям».
- В DFS главное — стек, а в BFS главное — очередь.
- Результат алгоритма поиска в глубину — это некий маршрут, который открывается от стартовой вершины и до искомой. А в алгоритме поиска в ширину маршрут не всегда является результатом.
- DFS является рекурсивным алгоритмом, а BFS — нет.

## Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

Код:

```
using System.Diagnostics;
using System.Text;
const int AMMOUNT = 10;//кол-во тстов
StreamWriter f = new StreamWriter("test1.txt");
StreamWriter f2 = new StreamWriter("test2.txt");
//тест
for (int i = 0; i < AMMOUNT; i++)
{
    Random rnd = new Random();
    var graph = RandomGraphGen((i+1)*100*10, (i + 1) * 100 *8, (i + 1) * 100 *14, (i + 1) *
100 *12, (i + 1) * 100 *4, true, (i + 1) * 100 * 5, (i + 1) * 100 * 4);
    //graph.PrintMatrix();
    //graph.PrintIncMatrix();
    //graph.PrintAdjList();
    //graph.PrintEdgeList();

        //Console.WriteLine("\nВыберите начальную точку для проверки A: ");
        Vertex a = new Vertex(rnd.Next(0, graph.vertex_count));
        //Console.WriteLine("\nВыберите начальную точку для проверки B: ");
        Vertex b = new Vertex(rnd.Next(0, graph.vertex_count));
        Stopwatch stopwatch1 = new Stopwatch();
        //замеры времени, потраченного на поиск DFS
        stopwatch1.Start();
        graph.FindShortFS(graph.GetAdjList(), a, b, 1);
        stopwatch1.Stop();
        f.WriteLine("ВРЕМЯ : " + stopwatch1.ElapsedMilliseconds);
        Stopwatch stopwatch2 = new Stopwatch();
        //замеры времени, потраченного на поиск BFS
        stopwatch2.Start();
        graph.FindShortFS(graph.GetAdjList(), a, b, 2);
        stopwatch2.Stop();
        f2.WriteLine("ВРЕМЯ 2: " + stopwatch2.ElapsedMilliseconds);
}
f.Close();
f2.Close();
/// <summary>
/// Метод, возвращающий рандомно-сгенерированный граф
/// </summary>
/// <param name="max_vertex">Макс. кол-во вершин</param>
/// <param name="min_vertex">Мин. кол-во вершин</param>
/// <param name="max_edge">Макс. кол-во рёбер</param>
/// <param name="min_edge">Мин. кол-во рёбер</param>
/// <param name="max_one_edge">Макс. кол-во рёбер у одной вершины</param>
/// <param name="is_oriented">Ориентированность графа</param>
/// <param name="max_in_edge">Макс. кол-во входящих рёбер</param>
/// <param name="max_out_edge">Макс. кол-во выходящих рёбер</param>
/// <returns>Рандомно-сгенерированный граф</returns>
/// <exception cref="Exception"></exception>
Graph RandomGraphGen(int max_vertex, int min_vertex, int max_edge, int min_edge, int
max_one_edge, bool is_oriented, int max_in_edge, int max_out_edge)
{
    var graph = new Graph(is_oriented);
    Random rnd = new Random();
    int rand_vert = rnd.Next(min_vertex, max_vertex + 1);//Выбор кол-ва вершин у графа в
диапазоне [max_vertex,  min_vertex]
    int rand_edge = rnd.Next(min_edge, max_edge + 1);//Выбор кол-ва рёбер у графа в диапазоне
[max_edge,  min_edge]
    int E = rand_vert * (rand_vert - 1) / 2;//Макс возможное кол-во ребер, при данном кол-ве
вершин
    if (rand_edge <= E)//Проверка данных параметров
    {
        Vertex[] vert_arr = new Vertex[rand_vert];
```

```
        //Создание вершин
        for (int i = 0; i < rand_vert; i++)
        {
            var v = new Vertex(i + 1);
            graph.AddVert(v);
            vert_arr[i] = new Vertex(i + 1);
        }

        vert_arr = vert_arr.OrderBy(x => rnd.Next()).ToArray();//Перемешивание элементов графа
        int count_edge = 0;
        if (is_oriented)//Если ориентированный
        {
            for (int i = 0; i < vert_arr.Length; i++)
            {
                int cur_in_edge = rnd.Next(0, max_in_edge);//Задание рандомного кол-ва
входящих вершин на шаге
                int cur_out_edge = rnd.Next(0, max_out_edge);//Задание рандомного кол-ва
выходящих вершин на шаге
                if (i == 0) cur_out_edge = max_out_edge;//Задание макс.кол-ва выход вершин для
первого шага
                if (i == 1) cur_in_edge = max_in_edge;//Задание макс.кол-ва вход вершин для
первого шага
                for (int j = 0; j < cur_out_edge && count_edge <= rand_edge;
j++)//Инициализация вершины с максимальным кол-вом выход элементов
                {
                    if (i + j + 1 < vert_arr.Length)
                    {
                        graph.AddEdge(vert_arr[i], vert_arr[i + j + 1]);
                        count_edge++;
                    }
                }
                for (int j = 0; j < cur_in_edge - 1 && count_edge <= rand_edge;
j++)//Инициализация вершины с максимальным кол-вом вход элементов
                {
                    if (i + j + 1 < vert_arr.Length && i < vert_arr.Length)
graph.AddEdge(vert_arr[i + j + 1], vert_arr[i]);
                    count_edge++;
                }
            }
        }
        else//если неориентированный
        {
            for (int i = 0; i < vert_arr.Length; i++)
            {
                int cur_in_edge = rnd.Next(0, max_one_edge - 1);
                if (i == 0) cur_in_edge = max_one_edge;
                for (int j = 0; j < cur_in_edge && count_edge < rand_edge; j++)//Инициализация
вершины с максимальным кол-вом выход элементов
                {
                    if (i + j + 1 < vert_arr.Length && i < vert_arr.Length)
                    {
                        graph.AddEdge(vert_arr[i], vert_arr[i + j + 1]);
                    }
                }
            }
        }
        return graph;
    }
    else
    {
        throw new Exception("Количество ребер должно быть меньше");
    }
}
/// <summary>
/// Класс:Вершина
/// </summary>
class Vertex
{
    public int number { get; set; }
```

```
    public bool visited = false;
    public Vertex(int number)
    {
        this.number = number;
    }
}
/// <summary>
/// Класс:Рёбра
/// </summary>
class Edge
{
    public Vertex from { get; set; }
    public Vertex to { get; set; }
    public int weight { get; set; }
    public Edge(Vertex from, Vertex to, int weight=1)
    {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
}

/// <summary>
/// Класс:Граф
/// </summary>
class Graph
{
    public List<Edge> edges=new List<Edge>();
    public List<Vertex> vertexes=new List<Vertex>();
    public bool oriented;
    public int vertex_count => vertexes.Count;
    public int edge_count => edges.Count;
    public Graph(bool oriented)
    {
        this.oriented = oriented;
    }
    /// <summary>
    /// Метод, возвращающий вес ребра
    /// </summary>
    /// <param name="from">Вершина От</param>
    /// <param name="to">Вершина Куда</param>
    /// <returns>Вес ребра</returns>
    public int FindWeightEdge(Vertex from, Vertex to)
    {
        int weight = 0;
        foreach(Edge e in edges)
        {
            if (e.from.number == from.number && e.to.number == to.number) weight = e.weight;
        }
        return weight;
    }
    /// <summary>
    /// Метод добавления Вершины в Граф
    /// </summary>
    /// <param name="vertex"></param>
    public void AddVert(Vertex vertex)
    {
        vertexes.Add(vertex);
    }
    /// <summary>
    /// Метод добавления Ребра в Граф
    /// </summary>
    /// <param name="from">Вершина От</param>
    /// <param name="to">Вершина Куда</param>
    public void AddEdge(Vertex from, Vertex to)
    {
        Edge edge = new Edge(from, to);
        edges.Add(edge);
    }
```

```
/// <summary>
/// Метод, возвращающий матрицу смежности
/// </summary>
/// <returns>Матрица смежности</returns>
public int[,] GetAdjMatrix()
{
    var matrix = new int[vertexes.Count, vertexes.Count];
    foreach(Edge edge in edges)
    {
        var row=edge.from.number - 1;
        var colm = edge.to.number - 1;
        matrix[row, colm] = edge.weight;
        if (oriented!=true) matrix[colm, row] = edge.weight;
    }
    return matrix;
}
/// <summary>
/// Метод, возвращающий матрицу инцидентности
/// </summary>
/// <returns>Матрица инцидентности</returns>
public int[,] GetIncMatrix()
{
    var matrix = new int[vertexes.Count, edges.Count];

    for (int i = 0; i < vertex_count; i++)
    {
        for (int j = 0; j < edge_count; j++)
        {
            if(this.edges[j].from.number == i+1)
            {
                matrix[i, j] = 1;
            }
            else if(this.edges[j].to.number == i+1)
            {
                matrix[i, j] = -1;
            }
        }
    }
    return matrix;
}
/// <summary>
///Метод, возвращающий список смежности
/// </summary>
/// <returns>Список смежности</returns>
public List<List<Vertex>> GetAdjList()
{
    var list = new List<List<Vertex>>();

    for (int i = 0; i < vertex_count; i++)
    {
        var list_vert = new List<Vertex>();
        foreach(var edge in edges)
        {
            if (edge.from.number == i + 1)
            {
                list_vert.Add(edge.to);
                edge.to.visited = false;

            }
            edge.from.visited = false;
        }
        list.Add(list_vert);
    }
    return list;
}
/// <summary>
/// Метод, возвращающий список рёбер
/// </summary>
/// <returns>Список рёбер</returns>
```

```
    public List<Edge> GetEdgeList()
    {
        return edges;
    }
    /// <summary>
    /// Метод, ищущий кратчайший путь от A до B
    /// </summary>
    /// <param name="adj_list">Список смежности</param>
    /// <param name="A">Точка A</param>
    /// <param name="B">Точка B</param>
    /// <param name="choice">Метод поиска</param>
    public void FindShortFS(List<List<Vertex>> adj_list, Vertex A, Vertex B, int choice)
    {
        int sum_weight = 0;
        string string_temp = "";
        Dictionary<int, string> all_path=new Dictionary<int, string>();
        for(int i = 0; i < adj_list.Count; i++)
        {
            for(int j=0; j < adj_list[i].Count; j++)
            {
                if (adj_list[i][j].number==A.number) adj_list[i][j].visited=true;
            }
        }
        if (choice == 1)
        {
            Console.WriteLine("Поиск по DFS");
            if (DFS(adj_list, A, B, ref all_path, ref sum_weight, ref string_temp))
            {
                string result = A.number + all_path[all_path.Keys.Min()];
                Console.WriteLine(result);
            }
            else Console.WriteLine("нет пути");
        }
        else
        {
            Console.WriteLine("Поиск по BFS");
            if (BFS(adj_list, A, B, ref all_path, ref sum_weight, ref string_temp))
            {
                string result = A.number + all_path[all_path.Keys.Min()];
                //Console.WriteLine("Путь есть");
                Console.WriteLine(result);
            }
            else Console.WriteLine("нет пути");
        }
    }

    /// <summary>
    /// Метод поиска DFS
    /// </summary>
    /// <param name="adj_list">Список смежности</param>
    /// <param name="A">Точка A</param>
    /// <param name="B">Точка B</param>
    /// <param name="path">Словарь для пути</param>
    /// <param name="sum_weight">Вес</param>
    /// <param name="string_temp">Строка для содержания пути</param>
    /// <returns>bool</returns>
    public bool DFS(List<List<Vertex>> adj_list, Vertex A,Vertex B, ref Dictionary<int,
string> path, ref int sum_weight, ref string string_temp)
    {
        if (A.number == B.number)
        {
            if(!path.ContainsKey(sum_weight)) path.Add(sum_weight,string_temp);
            return true;
        }
        if (A.visited) {return false; };
        A.visited = true;
        bool temp_bool=false;
        foreach(var adj in adj_list[A.number-1])
        {
```

```
            if (!adj.visited)
            {
                sum_weight += FindWeightEdge(A, adj);
                string_temp += adj.number.ToString();

                var reach=DFS(adj_list, adj, B, ref path, ref sum_weight, ref string_temp);
                if (reach) temp_bool=reach;
                sum_weight -= FindWeightEdge(A, adj);
                if (string_temp.Length - 1 >= 0) string_temp = string_temp.Substring(0,
string_temp.Length - 1);
            }
        }
        if (temp_bool) return true;
        return false;
    }
    /// <summary>
    /// Метод поиска BFS
    /// </summary>
    /// <param name="adj_list">Список смежности</param>
    /// <param name="A">Точка A</param>
    /// <param name="B">Точка B</param>
    /// <param name="path">Словарь для пути</param>
    /// <param name="sum_weight">Вес</param>
    /// <param name="string_temp">Строка для содержания пути</param>
    /// <returns></returns>
    public bool BFS(List<List<Vertex>> adj_list, Vertex A, Vertex B, ref Dictionary<int,
string> path, ref int sum_weight, ref string string_temp)
    {
        Queue<Vertex> queue=new Queue<Vertex>();
        queue.Enqueue(A);
        A.visited = true;
        while(queue.Count > 0)
        {
                var v = queue.Dequeue();
                foreach (var adj in adj_list[v.number - 1])
                {
                    if (!adj.visited)
                    {
                        sum_weight += FindWeightEdge(A, adj);
                        string_temp += adj.number.ToString();
                        queue.Enqueue(adj);
                        adj.visited = true;
                    if (adj.number == B.number) {
                        if (!path.ContainsKey(sum_weight)) path.Add(sum_weight, string_temp);
                        return true;
                    }
                        sum_weight -= FindWeightEdge(A, adj);
                        if (string_temp.Length - 1 >= 0) string_temp =
string_temp.Substring(0, string_temp.Length - 1);
                    }
                }
        }
        return false;
    }
    /// <summary>
    /// Меод вывода матрицы смежности
    /// </summary>
    public void PrintMatrix()
    {
        Console.WriteLine("\nМатрица смежности: ");
        var matrix = this.GetAdjMatrix();

        for (int i = 0; i < this.vertex_count; i++)
        {
            Console.Write("{0,2}|", i + 1);
            for (int j = 0; j < this.vertex_count; j++)
            {
                Console.Write("{0,3}", matrix[i, j] + " ");
            }
```

```
            Console.WriteLine();
        }
        Console.Write("   ");
        for (int j = 0; j < this.vertex_count; j++)
        {
            Console.Write("{0,2}|", j + 1);
        }
    }
    /// <summary>
    /// Метод вывода матрицы инцидентности
    /// </summary>
    public void PrintIncMatrix()
    {
        Console.WriteLine("\n\nМатрица инцидентности: ");
        var matrix_inc = this.GetIncMatrix();
        for (int i = 0; i < this.vertex_count; i++)
        {
            Console.Write("{0,2}|", i + 1);
            for (int j = 0; j < this.edge_count; j++)
            {
                Console.Write("{0,5}", matrix_inc[i, j] + " ");
            }
            Console.WriteLine();
        }
        Console.Write("   ");
        for (int j = 0; j < this.edge_count; j++)
        {
            Console.Write("{0,4}|", j + 1);
        }
        Console.WriteLine();
    }
    /// <summary>
    /// Метод вывода списка смежности
    /// </summary>
    public void PrintAdjList()
    {
        Console.WriteLine("\n\nСмежный список: ");
        var list = this.GetAdjList();
        for (int i = 0; i < this.vertex_count; i++)
        {
            var text = $"{i + 1}: ";
            for (int j = 0; j < list[i].Count; j++)
            {
                text+=/*"{0,5}", */list[i][j].number + ", ";
            }

            Console.Write("{0,5}|", text);
        }
        Console.Write("   ");
    }
    /// <summary>
    /// Метод вывода списка рёбер
    /// </summary>
    public void PrintEdgeList()
    {
        Console.WriteLine("\n\nСписок рёбер: ");
        var list = this.GetEdgeList();
        for (int i = 0; i < this.edge_count; i++)
        {
            Console.Write("{0}: {1}-{2}|",i+1, list[i].from.number, list[i].to.number);
        }
        Console.Write("   ");
    }


}
```

## Реализация:

1)Данный Граф реализован по схеме списка рёбер. Создаю класс Графа и внутри него класс Vertex и Edge, которые будут в данной реализации играть роль Вершин и Рёбер соответственно. Vertex содержит поле number, в котором будет хранится имя Вершины и visited(для отметки в обходах, посещали ли данную вершину), а Edge содержит поля from(для начальной точки) и to(для конечной), также weight(для замера веса ребра). В классе graph есть такие методы как edges(список всех ребер), vertexes(список всех вершин), oriented(делегат ориентированности графа) и vertex_count с edge_count, хранящие кол-во вершин и рёбер соответственно.

2) Создаю метод AddVert (отвечает за добавление новых элементов в список вершин). Создаю метод AddEdge(отвечает за добавление новых элементов в список рёбер).

3)Создаю методы для выдачи:

- GetAdjMatrix() Матрицы смежности (двумерная таблица для которой столбцы и строки соответствуют вершинам, а значения в таблицы соответствуют ребрам, для невзвешенного графа они могут быть просто 1 если связь есть и идет в нужном направлении и 0 если ее нет, а для взвешенного графа будут стоять конкретные значения.)
- GetIncMatrix() Матрицы инцидентности (матрица в которой строки соответствуют вершинам, а столбцы соответствуют связям, и ячейки ставиться 1 если связь выходит из вершины, -1 если входит и 0 во всех остальных случаях.
- GetAdjList() Списка смежности (список списков, содержащий все вершины, а внутренние списки для каждой вершины содержат все смежные ей.)
- GetEdgeList() Списка рёбер(Список ребер, это список строк в которых хранятся все ребра вершины, а внутренние значение содержит две вершины к которым присоединено это ребро.)

4)И также создаю методы для их вывода PrintMatrix(), PrintIncMatrix(), PrintAdjList(), PrintEdgeList().

5)Создаю метод DFS(обхода в глубину) по данному в лекции алгоритму рекурсивным методом

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2. Запускаем процедуру dfs(u)
3. Помечаем вершину u как пройденную
4. Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v)
5. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

6) Создаю метод BFS(обхода в ширину) по данному в лекции алгоритму методом

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развёрнутый.
3. Если узел u является целевым узлом, то завершить поиск с результатом «успех».
4. В противном случае, в конец очереди добавляются все преемники узла u, которые ещё не развёрнуты и не находятся в очереди.
5. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
6. Вернуться к п. 2.

6)Создаю в функции RandomGraphGen генератор случайных графок, который принимает в качестве параметров: Максимальное/Минимальное количество генерируемых вершин, Максимальное/Минимальное количество генерируемых ребер, Максимальное количество ребер связанных с одной вершины, Генерируется ли направленный граф, Максимальное количество входящи и выходящих ребер. Генерирует из них, кол-во ребер и вершин, по которым создает веришны и ребра. Далее в функции проверяет, возможно ли с таким кол-вом вершин создать столько ребер, если да, то он их создает, перемешивает, чтобы уже в цикле, в случайном порядке распределить для первых попавшихся вершин макс. кол-во входящий ребер и выходящих для второго, а для остальных это будет определяться случайным числом в диапазоне от 0 до максимального кол-ва входящих, выходящих минус 1 элемент.

7)В функции main я замеряю время, потраченное на поиск обоими методами и записываю выводы в файлы test1.txt и test2.txt

## Результаты:

1)Демонстрация работы методов на небольших значениях при RandomGraphGen(10, 8, 14, 12, 4, true, 5, 4)

Матрица смежности:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Матрица инцидентности:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | -1 | 0 | 0 | 0 | 1 | 1 | 1 | -1 | -1 | -1 | -1 | 0 | 0 | 0 |
| 3 | 0 | 0 | -1 | 0 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | -1 | 0 | 1 |
| 4 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | -1 | 0 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | -1 | 0 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | -1 | -1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Смежный список:

1:
2: 5, 3, 7
3: 2, 7
4: 2, 5, 3, 7
5: 2, 3, 7
6:
7: 2
8: 2
9:

Список рёбер:

1: 4-2
2: 4-5
3: 4-3
4: 4-7
5: 2-5
6: 2-3
7: 2-7

8: 5-2
9: 3-2
10: 7-2
11: 8-2
12: 5-3
13: 5-7
14: 3-7

2)Демонстрация работы поиска

Тест 1:
Выберите начальную точку для проверки A:1
Выберите начальную точку для проверки B:8
Поиск по DFS: нет пути
Поиск по BFS: нет пути

Тест 2:
Выберите начальную точку для проверки A:3
Выберите начальную точку для проверки B:5
Поиск по DFS: 325
Поиск по BFS:35

Тест 3:
Выберите начальную точку для проверки A: 4
Выберите начальную точку для проверки B:7
Поиск по DFS:47
Поиск по BFS:47

3) Замерил время исполнения поиска по BFS и DFS с возрастающими в каждом тесте кол-ве Вершин и Рёбер графа.

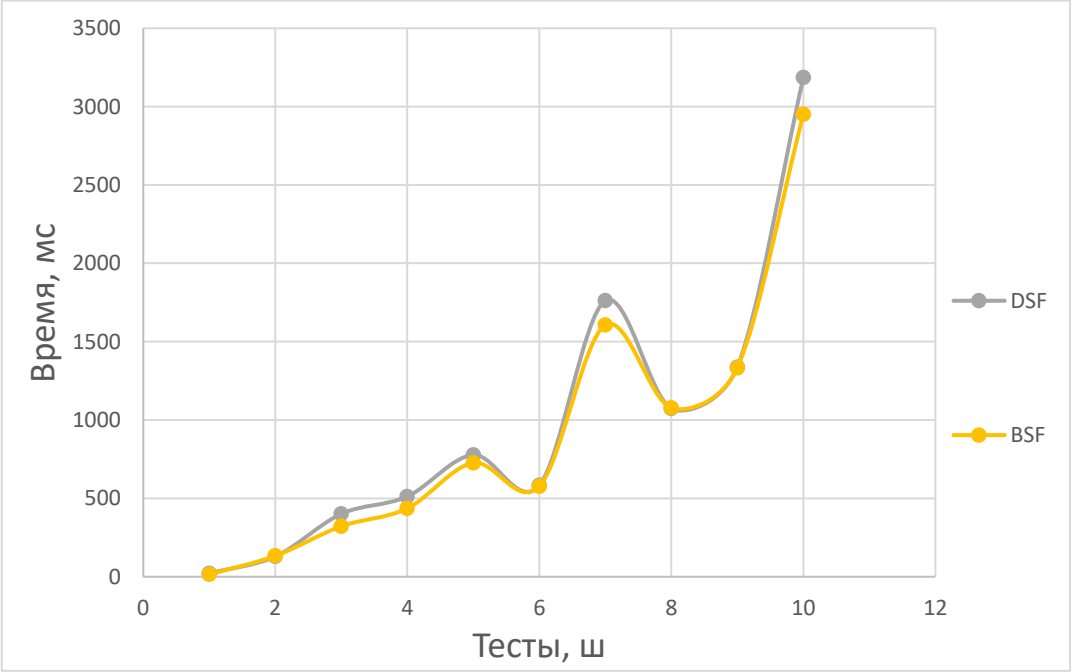


Рис.3. График изменения времени исполнения поиска у графа.

На графике видно, что методы с увеличением кол-ва Вершин и Рёбер почти одинаково возрастают, при этом обычно DSF обходится дольше.

## Заключение.

В реализации графа самым непростым мне показалась создание случайного генератора для него. В плане эффективности видно, что DSF немного уступает BSF, так как обходит весь граф, что можно немного ускорить, дав ему выйти сразу же после первого нахождения пути, но тогда нельзя будет

отсечь кратчайший путь до конечной точки. Также DSF позволяет точно отслеживать путь до конечной точки.