Санкт-Петербургский Национальный Исследовательский
Университет Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №1
Создание программы с помощью среды разработки Visual Studio .NET

Выполнил
Стафеев И.А.

Группа
K3221

Проверил
Иванов С.Е.

Санкт-Петербург,
2025

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Цель работы: научиться реализовывать алгоритмы на графах средствами ООП на C#.

Для достижения цели были поставлены следующие задачи:

1. реализовать алгоритым поиска в глубину и ширину на графе;
2. реализовать алгоритм нахождение кратчайших путей (алгоритм Дейкстры);
3. реализовать алгоритм поиска минимального остовного дерева (алгоритм Крускала).

1 Реализация класса для графа

Для реализации графа первоначально была создана структура **Edge** для ребра графа, код которой представлен на рисунке 1. Ее полями являются номера первой и второй вершин, индицентных ребру, и ее вес. Также был переопределен метод *Tostring()* для вывода ребер в консоль.

```
Ссылок: 24
1   struct Edge {
2       public int i, j;
3       public long weight;
        Ссылок: 1
4       public override string ToString() {
5           return $"Edge(i={i}, j={j}, weight={weight})";
6       }
7   }
8
```

Рисунок 1 — Структуры Edge для ребра графа

На рисунке 2 представлены поля и конструктор класса **Graph**, реализующего граф. в качестве полей для графа хранятся его список ребер, словарь вершин с инцидентными им ребрами и матрица связей, реализованная в виде двойного словаря. Каждый из типов представления графа нужен для определенного алгоритма, где его применение наиболее оправданно. Начальным типом представления является словарь вершин с инцидентными ребрами.

```
Ссылок: 11
class Graph {
    public Dictionary<int, List<Edge>> edges_list;
    public Dictionary<int, Dictionary<int, long>> edges_dict;
    public List<Edge> edges;
    public int n;

    Ссылок: 0
    public Graph() { }

    Ссылок: 1
    public Graph(Dictionary<int, List<Edge>> edges_list) {
        this.edges_list = edges_list;
        this.n = NodesCount(edges_list);
    }
```

Рисунок 2 — Поля класса Graph и его конструктор

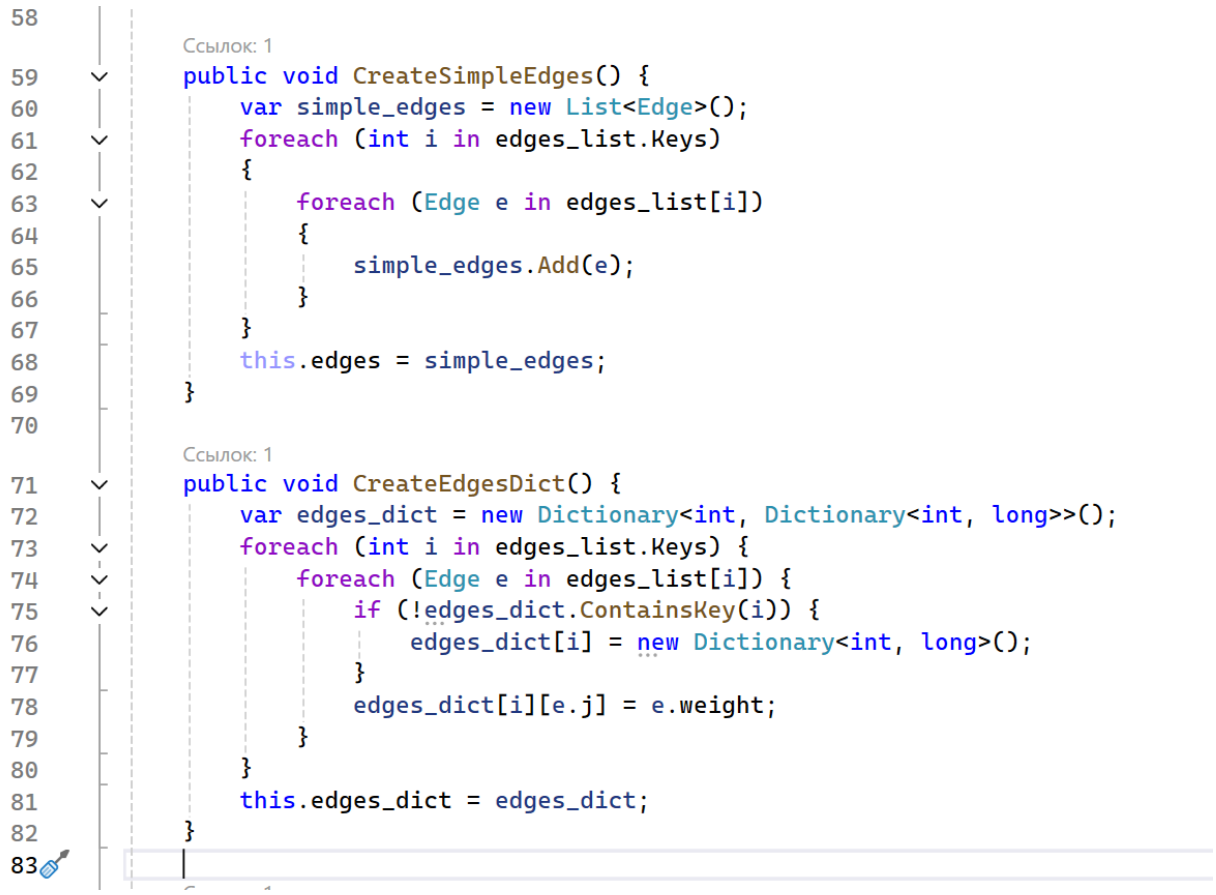
Метод для создания графа пользователем **CreateGraph** представлен на рисунке 3. У пользователя запрашивается количество ребер и наличие ориентации графа, после чего пользователь вводит ребра, укзывая инцидентные вершины и вес ребра. Если вес не указывается, он принимается равным единице. После завершения пользовательского ввода метод возвращает новый экземпляр класса графа.

```
Ссылок: 1
public static Graph CreateGraph() {
    Console.Write("Введите количество ребер: ");
    int m  = Int32.Parse(Console.ReadLine());
    var edges = new Dictionary<int, List<Edge>>();
    Console.Write("Граф ориентированный? (true/false): ");
    bool oriented = bool.Parse(Console.ReadLine());
    Console.WriteLine("Вводите ребра в формате 'i j w', где i, j - 1-я и 2-я вершины, w - вес ребра");
    for (int k = 0; k < m; k++) {
        string[] args = Console.ReadLine().Split();
        int i = Int32.Parse(args[0]);
        int j = Int32.Parse(args[1]);
        int weight;
        try
        {
            weight = Int32.Parse(args[2]);
        }
        catch (IndexOutOfRangeException) {
            weight = 1;
        }
        // добавление ребра
        if (!edges.ContainsKey(i)) {
            edges[i] = new List<Edge>();
        }
        var e = new Edge { i = i, j = j, weight = weight };
        edges[i].Add(e);
        if (oriented) { continue;  }
        if (!edges.ContainsKey(j))
        {
            edges[j] = new List<Edge>();
        }
        edges[j].Add(new Edge { i = j, j = i, weight = weight });

    }
    return new Graph(edges);
}
```

Рисунок 3 — Метод CreateGraph

Для преобразования типов представления графа были реализованы методы **CreateSimpleEdges** и **CreateEdgesDict**, которые предобразуют словарь вершин и инцидентныз ребер в простой список ребер и матрицу связей соответственно. Код этих методов можно увидеть на рисунке 4.



```
58
        Ссылок: 1
59      public void CreateSimpleEdges() {
60          var simple_edges = new List<Edge>();
61          foreach (int i in edges_list.Keys)
62          {
63              foreach (Edge e in edges_list[i])
64              {
65                  simple_edges.Add(e);
66              }
67          }
68          this.edges = simple_edges;
69      }
70
        Ссылок: 1
71      public void CreateEdgesDict() {
72          var edges_dict = new Dictionary<int, Dictionary<int, long>>();
73          foreach (int i in edges_list.Keys) {
74              foreach (Edge e in edges_list[i]) {
75                  if (!edges_dict.ContainsKey(i)) {
76                      edges_dict[i] = new Dictionary<int, long>();
77                  }
78                  edges_dict[i][e.j] = e.weight;
79              }
80          }
81          this.edges_dict = edges_dict;
82      }
83
```

Рисунок 4 — Методы CreateSimpleEdges и CreateEdgesDict

На рисунке 5 показаны методы **NodesCount** и **PrintEgdes**, которые возвращают количество вершин в графе и выводят список его ребер соответственно.

```
Ссылок: 1
84  public static int NodesCount(Dictionary<int, List<Edge>> edges_list)
85  {
86      int maxNode = 0;
87      foreach (int i in edges_list.Keys) {
88          foreach (Edge e in edges_list[i]) {
89              var args = new[] { maxNode, i, e.i, e.j };
90              maxNode = args.Max();
91          }
92      }
93      return maxNode;
94  }
95
    Ссылок: 0
96  public void PrintEdges() {
97      foreach (int i in edges_list.Keys) {
98          foreach (Edge e in edges_list[i]) {
99              Console.WriteLine(e.ToString());
100         }
101     }
102 }
```

Рисунок 5 — Методы NodesCount и PrintEdges

В рамках лабораторной работы будет использоваться граф, представленный на рисунке 6. Его создание через пользовательский ввод и вывод его ребер показан на рисунке 7.

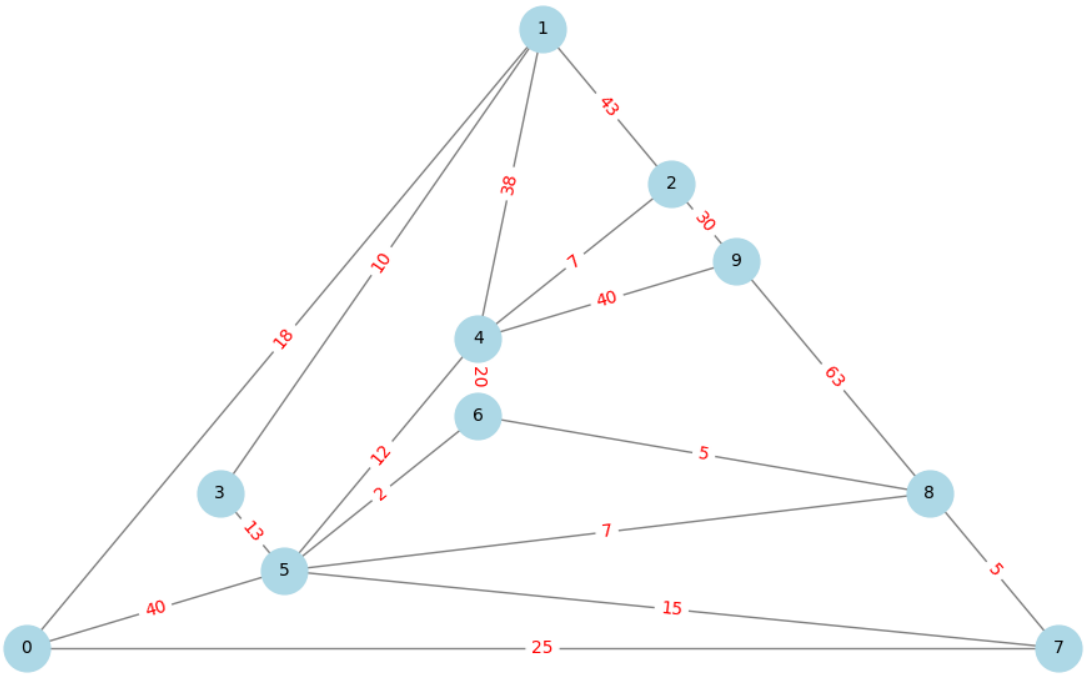


Рисунок 6 — Граф, на котором будут выполняться алгоритмы

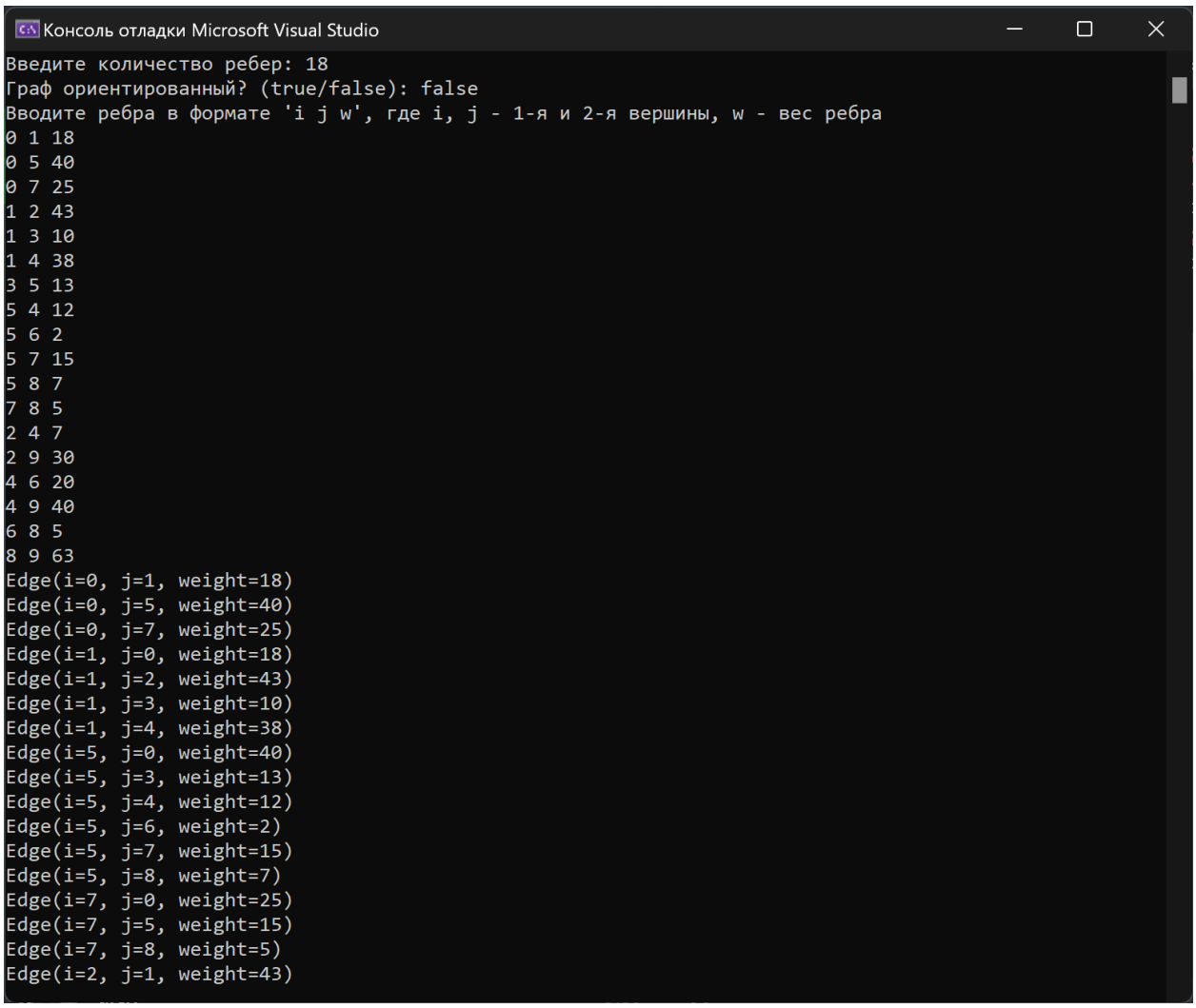


Рисунок 7 — Результат создания графа через пользовательский ввод

2 Алгоритмы обхода графа

На рисунке 8 показан код методов для обхода графа в глубину. Была реализована стандартная версия алгоритма, использующая рекурсию. В этом алгоритме используется тип представления графа в виде словаря вершин и инцидентных ребер.

```
Ссылок: 2
107  private static void _DFS(int start_node, Graph g, bool[] visited)
108  {
109      visited[start_node] = true;
110      Console.Write("{0} ", start_node);
111      var cur_edges = new List<Edge>();
112      if (!g.edges_list.TryGetValue(start_node, out cur_edges)) { return; }
113      foreach (Edge e in cur_edges)
114      {
115          if (g.edges_list.ContainsKey(e.j) && !visited[e.j])
116          {
117              _DFS(e.j, g, visited);
118          }
119      }
120  }
     Ссылок: 0
121  public static void DFS(int start_node, Graph g, bool[] visited) {
122      _DFS(start_node, g, visited);
123      Console.WriteLine();
124  }
125
```

Рисунок 8 — Алгоритм DFS обхода графа в глубину

Результат выполнения алгоритма для графа 6 показан на рисунке 9. Порядок обхода вершин показан на рисунке 10. Обход выполнялся с нулевой вершины.

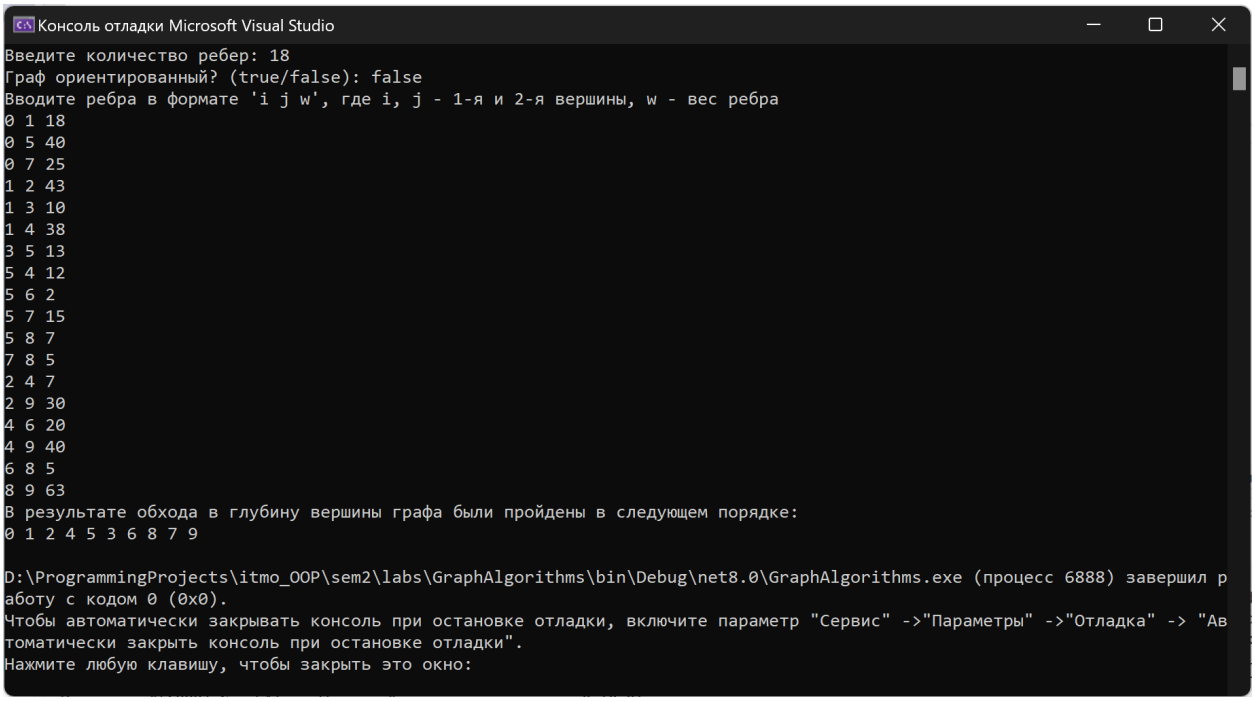


Рисунок 9 — Результат выполнения алгоритма обхода графа в глубину

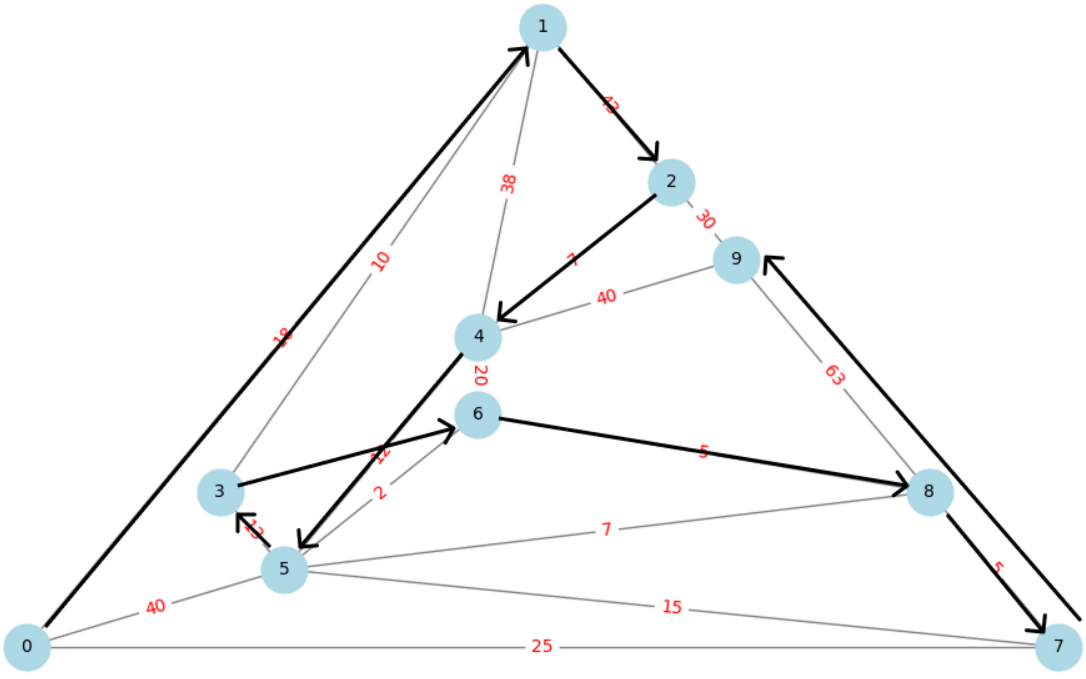


Рисунок 10 — Порядок обхода вершин в глубину

Код алгоритма обхода в ширину показан на рисунке 11. В методе используется коллекция "очередь".

```
Ссылок: 0
126     public static void BFS(int start_node, Graph g, bool[] visited)
127     {
128         var queue = new Queue<int>();
129         visited[start_node] = true;
130         queue.Enqueue(start_node);
131         while (queue.Count > 0) {
132             int cur_node = queue.Dequeue();
133             Console.Write("{0} ", cur_node);
134             var cur_edges = new List<Edge>();
135             if (!g.edges_list.TryGetValue(cur_node, out cur_edges)) { continue; }
136             foreach (Edge e in cur_edges)
137             {
138                 if (g.edges_list.ContainsKey(e.j) && !visited[e.j])
139                 {
140                     visited[e.j] = true;
141                     queue.Enqueue(e.j);
142                 }
143             }
144         }
145         Console.WriteLine();
```

Рисунок 11 — Алгоритм BFS обхода графа в ширину

Результат выполнения алгоритма для графа 6 показан на рисунке 12. Порядок обхода вершин показан на рисунке 13. Обход выполнялся с нулевой вершины.

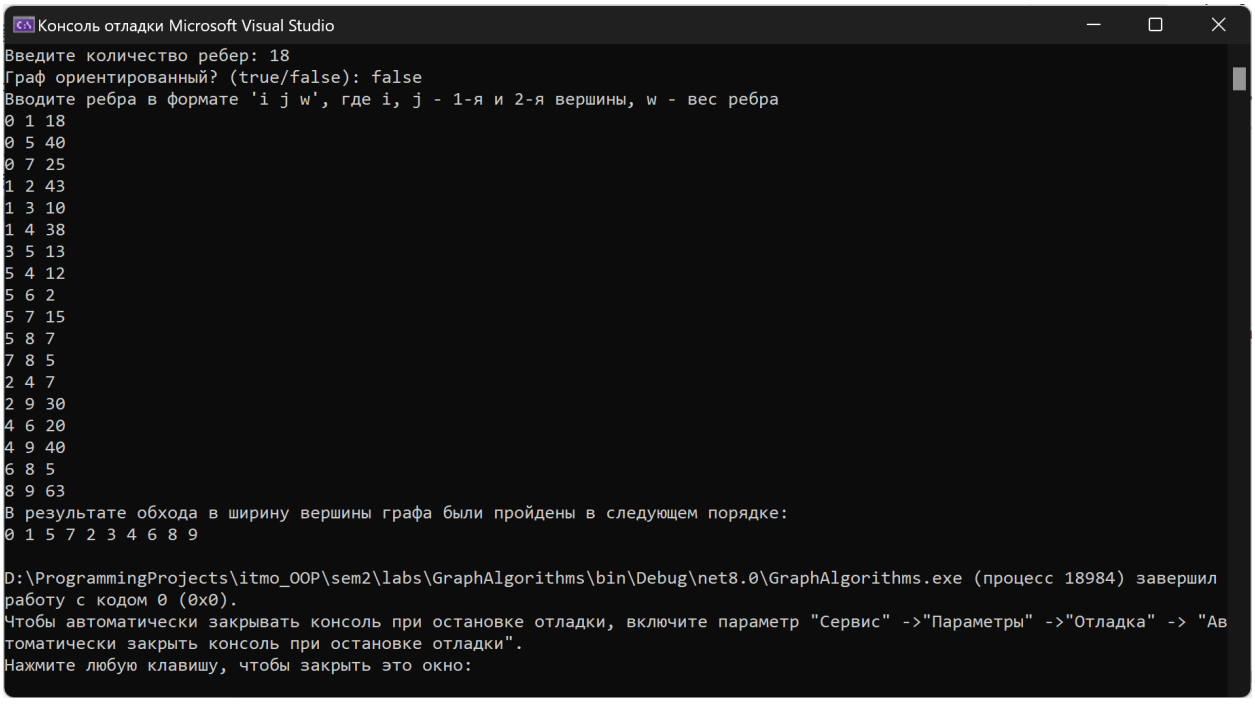


Рисунок 12 — Результат выполнения алгоритма обхода графа в ширину

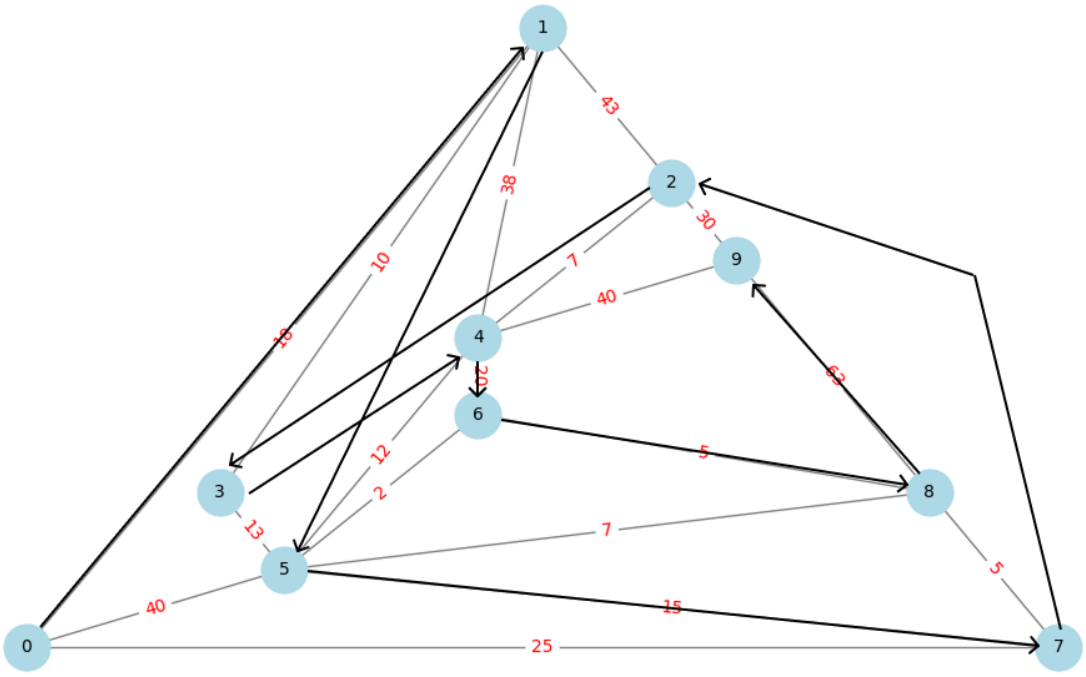


Рисунок 13 — Порядок обхода вершин в ширину

3 Алгоритм Дейкстры

Был реализован стандартнйы алгоритм Дейкстры, использующий окрашенные и неокрашенные метки, содержащие текущую минимальную длину пути от начальной вершины до текущей.

На рисунке 15 показаны вспомогательные методы алгоритма Дейкстры. Метод **GetNeighbours** возвращает список соседей вершины, то есть вершины, инцидентные данной, метки которых еще не окрашены, то есть для них еще не найден путь минимальной длины. Метод **GetMinLabelNode** возвращает вершину с минимальным значением метки, причем метка должна быть неокрашена (так выбирается следующая вершина, через которую будут обновляться длины путей).

```
177     public static List<int> GetNeighbours(int node, Graph g, bool[] marked)
178     {
179         var neighbours = new List<int>();
180         foreach (Edge e in g.edges_list[node]) {
181             if (marked[e.j]) { continue; }
182             neighbours.Add(e.j);
183         }
184         return neighbours;
185     }
186
        Ссылок: 1
187     public static int GetMinLabelNode(long[] distances, bool[] marked) {
188         long min_label = (long)Math.Pow(10, 9);
189         int argmin = 0;
190         for (int i = 0; i < marked.Length; i++) {
191             if (marked[i]) { continue; }
192             if (distances[i] < min_label) {
193                 min_label = distances[i];
194                 argmin = i;
195             }
196
197         }
198         return argmin;
199     }
200
```

Рисунок 14 — Методы GetNeighbours и GetMinLabelNode

Код алгоритма Дейкстры представлен на рисунке 16. В методе сначала вызывается создание матрицы связей у графа, для которого ищутся кратчайшие пути, а также инициализируются массивы расстояний и массив флагов окрашенности меток. Затем в цикле выбирается вершины с наименьшим значением метки, для нее находятся соседи, и для каждого соседа алгоритм пытается уменьшить длину пути с проходом через текущую

вершину. Когда метки всех верших окрашены, алгоритм заканчивает работу. Результатом работы алгоритма является список расстояний от текущей вершины для всех остальных и словарь вершин, где значением является предыдущая вершина на пути минимальной длины.

```
Ссылок: 1
150 public static void Dijkstra(int cur_node, Graph g, out long[] final_distances, out Dictionary<int, int> final_prev_nodes)
151 {
152     g.CreateEdgesDict();
153     var distances = Enumerable.Repeat((long)Math.Pow(10, 9), g.n + 1).ToArray();
154     distances[cur_node] = 0;
155     bool[] marked = new bool[g.n + 1];
156     marked[cur_node] = true;
157     int unmarked = g.n;
158     var previous_nodes = new Dictionary<int, int>();
159
160     while (unmarked > 0) {
161         var neighbours = GetNeighbours(cur_node, g, marked);
162         foreach (int neigh in neighbours)
163         {
164             long prev_dist = distances[neigh];
165             long new_dist = distances[cur_node] + g.edges_dict[cur_node][neigh];
166             if (new_dist < prev_dist) {
167                 distances[neigh] = new_dist;
168                 previous_nodes[neigh] = cur_node;
169             }
170         }
171         cur_node = GetMinLabelNode(distances, marked);
172         marked[cur_node] = true;
173         unmarked--;
174     }
175     final_distances = distances;
176     final_prev_nodes = previous_nodes;
177 }
178
```

Рисунок 15 — Код алгоритма Дейкстры Dijkstra

Метод **PrintMinCostPath**, выводящий кратчайший путь между переданными вершинами, показан на рисунке 16.

```
Ссылок: 1
246 public static void PrintMinCostPath(int start_node, int end_node, Dictionary<int, int> prevs)
247 {
248     var path = new List<int>();
249     while (end_node != start_node)
250     {
251         path.Add(end_node);
252         end_node = prevs[end_node];
253     }
254     path.Add(start_node);
255     path.Reverse();
256     for (int i = 0; i < path.Count - 1; i++)
257     {
258         Console.Write("{0} -> ", path[i]);
259     }
260     Console.Write("{0}\n", path[path.Count - 1]);
261 }
262 }
263
```

Рисунок 16 — Метод PrintMinCostPath

Результат выполнения алгоритма и вывод кратчайшего пути между вершинами 0 и 9 показан на рисунке 17. Кратчайший путь показан на рисунке 18.

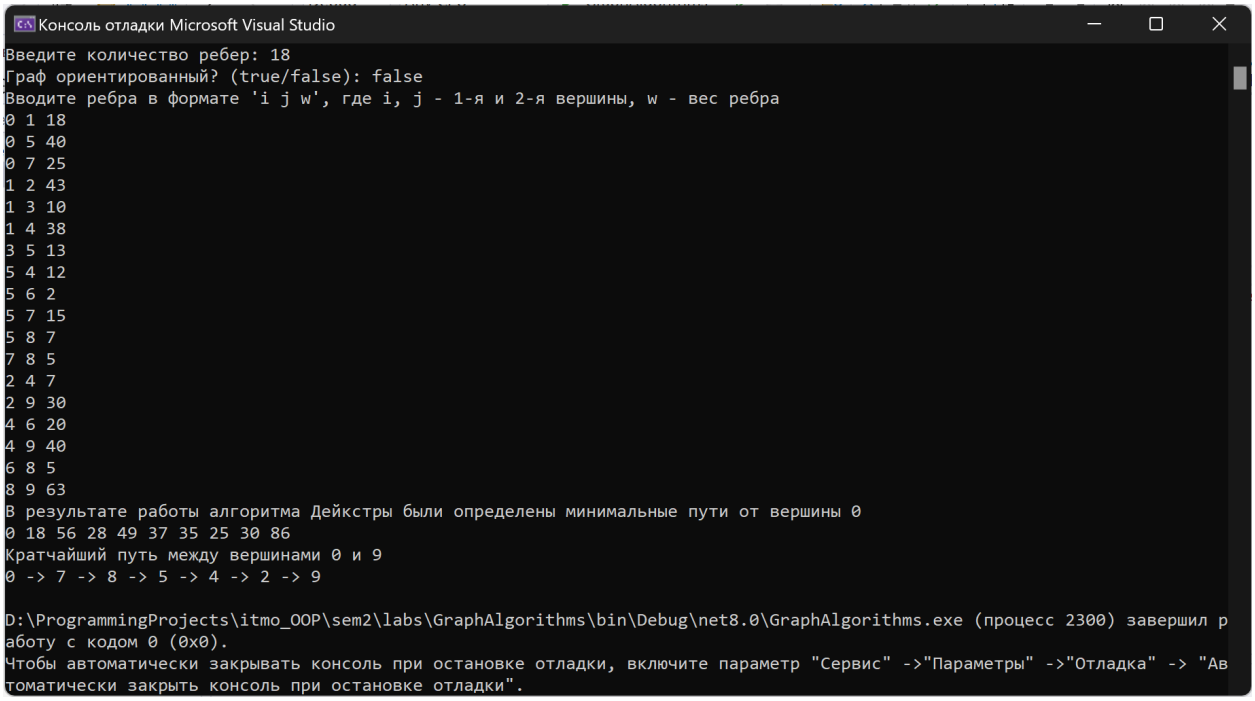


Рисунок 17 — Результат работы алгоритма Дейкстры

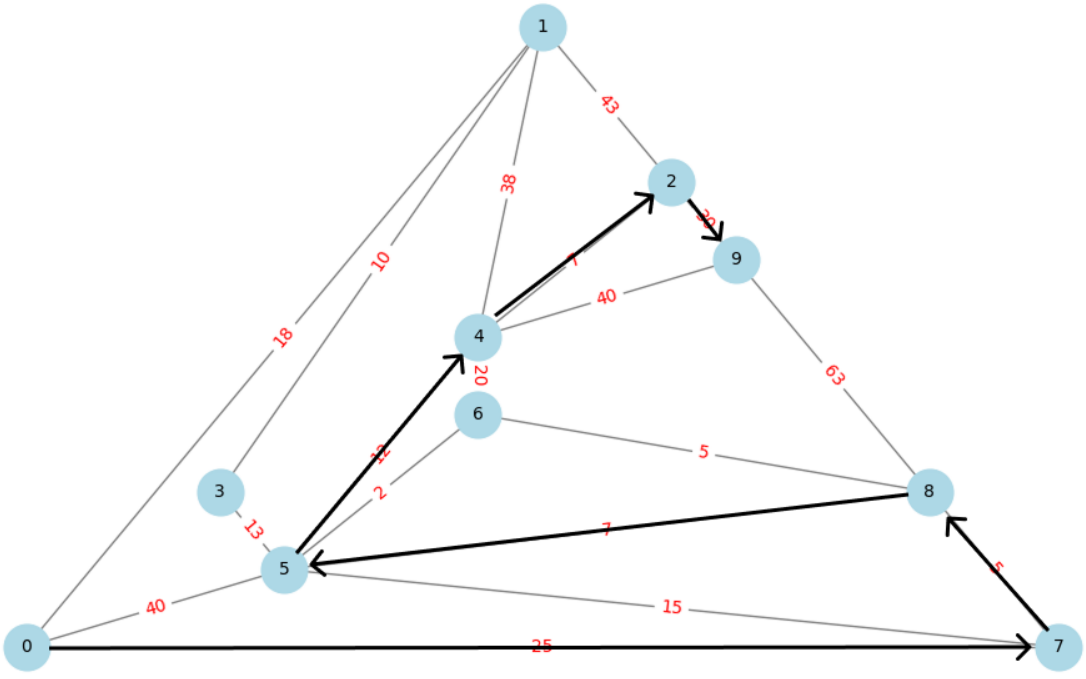


Рисунок 18 — Кратчайший путь между вершинами 0 и 9

4 Алгоритм Крускала

Код алгоритма Крускала для поиска минимального остовного дерева показан на рисунке 19. В начале для данного графа создается простой список вершин, который сортируется по возрастанию веса. Также создаются множество использованных вершин и словарь вида вершина - группа вершин минимального остовного дерева. В цикле, проходящем по всем ребрам, происходит проверка, не использованы ли уже обе вершины. Если использованы, добавление ребра между ними приведет к созданию цикла. Если хотя бы одна вершина не использована, она добавляется в группу второй вершины (если не использована и вторая, создается новая группы для этих вершин). После первого цикла будут созданы несколько изолированных групп вершин, и каждая группа уже есть минмальное остовное дерево для подграфа из вершин группы. Во втором цикле в минимальное остовное дерево добавляются ребра, соединяющие вершины из разных групп, при этом происходит объединение групп.

```
203     public static List<Edge> Kruskal(Graph g) {
204         g.CreateSimpleEdges();
205         var edges = g.edges;
206         edges = edges.OrderBy(e => e.weight).ToList();
207 
208         var used = new HashSet<int>();
209         var groups = new Dictionary<int, HashSet<int>>();
210         var MST = new List<Edge>();
211 
212         foreach (Edge e in edges) { // создаем изолированные группы
213             if (used.Contains(e.i) && used.Contains(e.j)) { continue; } // избегаем цикла
214             if (!used.Contains(e.i) && !used.Contains(e.j)) // обе вершины изолированы
215             {
216                 var group = new HashSet<int> { e.i, e.j };
217                 groups[e.i] = group;
218                 groups[e.j] = group;
219             }
220             else if (!used.Contains(e.i)) // первая изолирована
221             {
222                 groups[e.j].Add(e.i);
223                 groups[e.i] = groups[e.j];
224             }
225             else if (!used.Contains(e.j)) { // вторая изолирована
226                 groups[e.i].Add(e.j);
227                 groups[e.j] = groups[e.i];
228             }
229             MST.Add(e);
230             used.Add(e.i);
231             used.Add(e.j);
232         }
233         foreach (Edge e in edges) { // объединяем группы
234             if (!groups[e.i].Contains(e.j)) {
235                 MST.Add(e);
236                 var group = groups[e.i];
237                 groups[e.i].UnionWith(groups[e.j]);
238                 groups[e.j].UnionWith(group);
239 
240             }
241         }
242         return MST;
243     }
```

Рисунок 19 — Алгоритм Крускала Kruskal

Результат работы алгоритма показан на рисунке 20. Минимальное остовное дерево для графа 6 показано на рисунке 21.

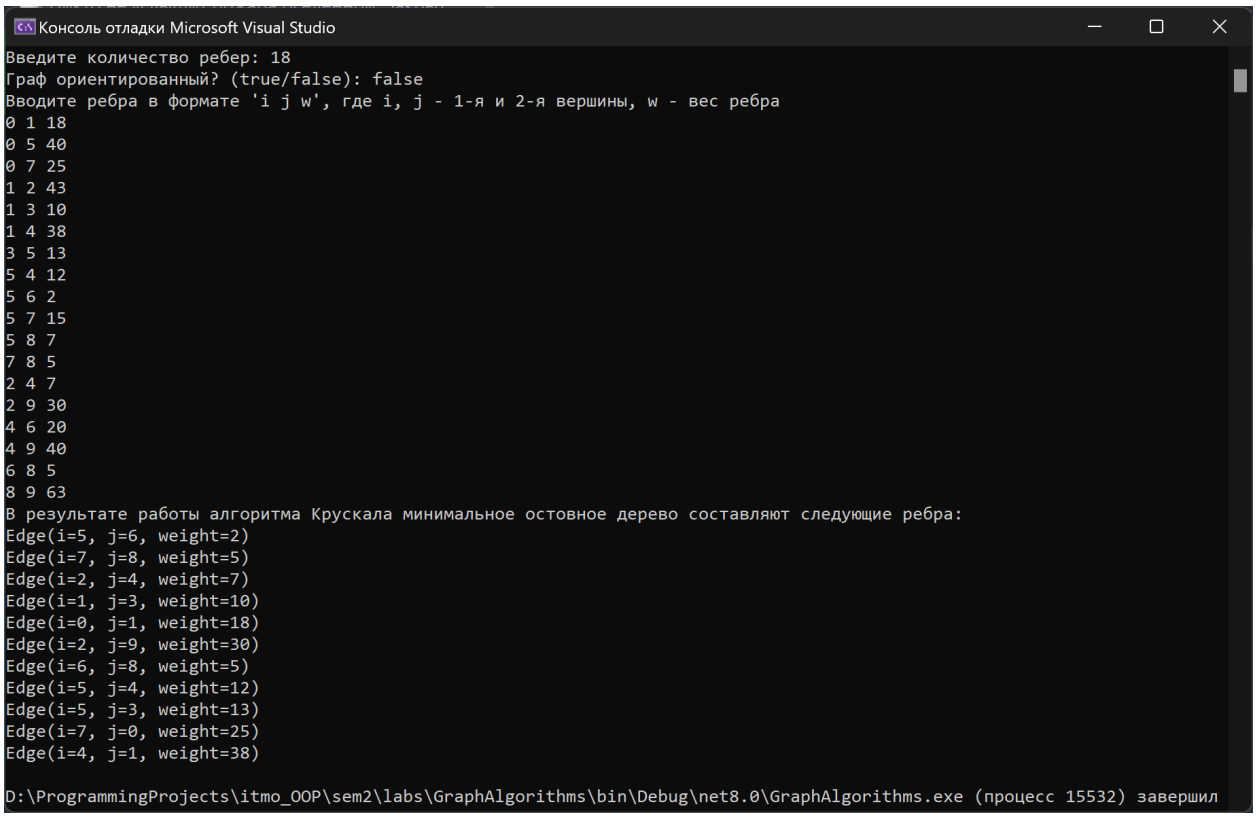


Рисунок 20 — Результат работы алгоритма Крускала

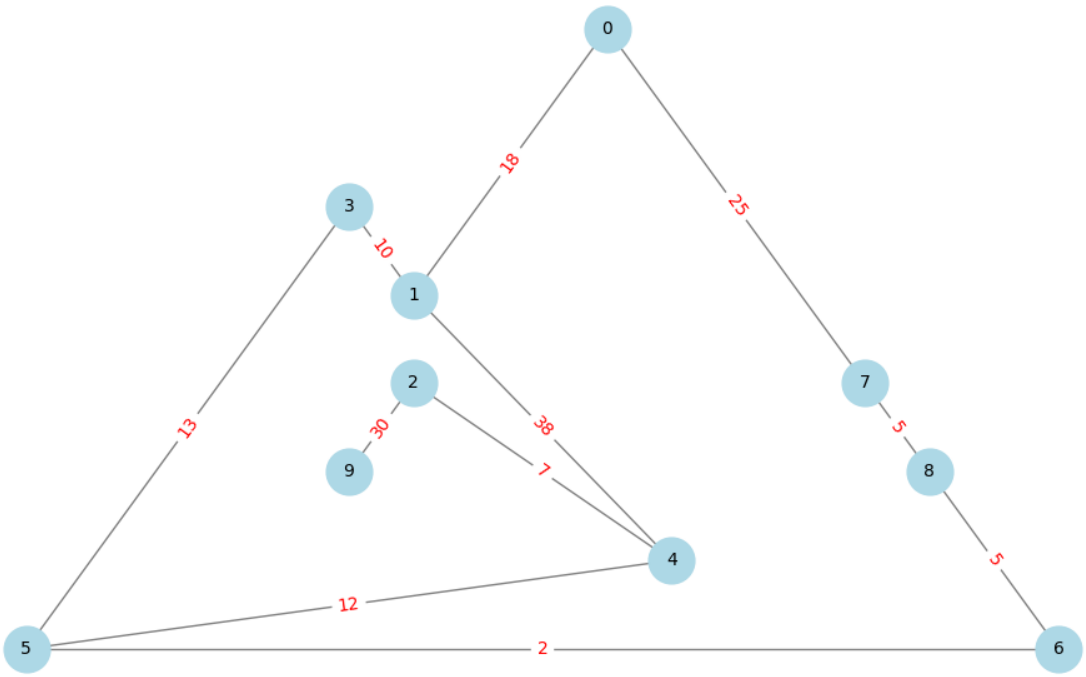


Рисунок 21 — Минимальное остовное дерево

На рисунке 22 показан метод **Main**, в котором вызываются все реализованные алгоритмы на графах.

```
263  class Program {
         Ссылок: 0
264      public static void Main() {
265          var g = Graph.CreateGraph();
266          bool[] visited = new bool[g.n + 1];
267          Console.WriteLine("В результате обхода в глубину вершины графа были пройдены в следующем порядке:");
268          Algorithms.DFS(0, g, visited);
269          visited = new bool[g.n + 1];
270          Console.WriteLine("В результате обхода в ширину вершины графа были пройдены в следующем порядке:");
271          Algorithms.BFS(0, g, visited);
272
273          var distances = new long[g.n + 1];
274          var prevs = new Dictionary<int, int>();
275          int start_node = 0;
276          int end_node = 9;
277          Algorithms.Dijkstra(start_node, g, out distances, out prevs);
278          Console.WriteLine("В результате работы алгоритма Дейкстры были определены минимальные пути от вершины {0}", start_node);
279          foreach (long dist in distances)
280          {
281              Console.Write("{0} ", dist);
282          }
283          Console.WriteLine();
284          Console.WriteLine("Кратчайший путь между вершинами {0} и {1}", start_node, end_node);
285          Algorithms.PrintMinCostPath(start_node, end_node, prevs);
286          foreach (int i in distances)
287          {
288              Console.Write("{0} ", i);
289          }
290          Console.WriteLine();
291          Console.WriteLine("В результате работы алгоритма Крускала минимальное остовное дерево составляют следующие ребра:");
292          var edges = Algorithms.Kruskal(g);
293          foreach (Edge e in edges) {
294              Console.WriteLine(e);
295          }
296      }
297  }
```

Рисунок 22 — Метод Main

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были выполнены все требуемые упражнения. Цель работы достигнута. Получены знания об алгоритмах на графах для их обхода, поиска кратчайших путей между вершинами и для поиска минимального остовного дерева графа, а также получены навыки их реализации средствами ООП на языке C#.