МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
доцент, к. ф.-м. н.
_____ М. В. Огнева

ОТЧЕТ О ПРАКТИКЕ

студентки 3 курса 311 группы факультета компьютерных наук и информационных технологий
Никитенко Яны Валерьевны

вид практики: учебная
кафедра: информатики и программирования
курс: 3
семестр: 5
продолжительность: 2 нед., с 01.09.2025 г. по 31.12.2025 г.

Руководитель практики от университета,

с.п, кафедры информатики и программирования М. С. Портенко

Руководитель практики от организации (учреждения, предприятия),

доцент, к. ф.-м. н. _____ С. В. Миронов

Тема практики:«Теория графов в 2025»

СОДЕРЖАНИЕ

1 Класс Граф

1.1 Задание

Для решения всех задач курса необходимо создать класс (или иерархию классов - на усмотрение разработчика), содержащий:

1. Структуру для хранения списка смежности графа (не работать с графом через матрицы смежности, если в некоторых алгоритмах удобнее использовать список ребер - реализовать метод, создающий список рёбер на основе списка смежности);

2. Конструкторы (не менее 3-х):

конструктор по умолчанию, создающий пустой граф конструктор, заполняющий данные графа из файла конструктор-копию (аккуратно, не все сразу делают именно копию) специфические конструкторы для удобства тестирования

3. Методы:

- добавляющие вершину,
- добавляющие ребро (дугу),
- удаляющие вершину,
- удаляющие ребро (дугу),
- выводящие список смежности в файл (в том числе в пригодном для чтения конструктором формате).

Не выполняйте некорректные операции, сообщайте об ошибках.

4. Должны поддерживаться как ориентированные, так и неориентированные графы. Заранее предусмотрите возможность добавления меток и\или весов для дуг. Поддержка мультиграфа не требуется.

5. Добавьте минималистичный консольный интерфейс пользователя (не смешивая его с реализацией!), позволяющий добавлять и удалять вершины и рёбра (дуги) и просматривать текущий список смежности графа.

6. Сгенерируйте не менее 4 входных файлов с разными типами графов (балансируйте на комбинации ориентированность-взвешенность) для тестирования класса в этом и последующих заданиях. Графы должны содержать не менее 7-10 вершин, в том числе петли и изолированные вершины.

1.2 Реализация

1. Реализация класса Graph1 и его основных методов:

Структура класса - приватные поля:

Dictionary<int, List<Edge» adjacencyList - список смежности

bool IsDirected - флаг ориентированности графа

bool IsWeighted - флаг взвешенности графа

2. Конструкторы:

По умолчанию (создает пустой граф с указанной направленностью и взвешенностью)

Из файла (загружает граф из указанного файла)

Копирования (создает копию существующего графа)

Из списка рёбер (для тестирования)

Из матрицы смежности (для тестирования)

3. Методы проверки:

ContainsVertex() - проверяет существование вершины

ContainsEdge() - проверяет существование ребра

4. Методы модификации графа:

AddVertex() - добавляет новую вершину

AddEdge() - добавляет новое ребро (с весом или без)

RemoveVertex() - удаляет вершину и все связанные с ней рёбра

RemoveEdge() - удаляет указанное ребро

5. Методы работы с файлами и вывода:

SaveToFile() - сохраняет граф в файл в формате:

text

directed/undirected weighted/unweighted

вершина1: смежная вершина вес...

вершина 2: ...

Конструктор Graph(string filename) - загружает граф из файла аналогичного формата

GetEdgeList() - возвращает список всех рёбер графа в виде кортежей (from, to, weight)

GetAdjacencyListString() - возвращает строковое представление списка смежности

6. Дополнительные методы:

GetVertices() - возвращает список всех вершин

GetAdjacentVertices() - возвращает список смежных вершин для указанной вершины

VertexCount - свойство, возвращающее количество вершин

EdgeCount - свойство, возвращающее количество рёбер

7. Вложенный класс Edge:

Target - целевая вершина ребра

Weight - вес ребра (по умолчанию 1.0)

Особенности реализации:

- Поддержка как ориентированных, так и неориентированных графов
- Поддержка как взвешенных, так и невзвешенных графов
- Автоматическое добавление обратных рёбер для неориентированных графов
- Проверка на существование вершин и рёбер перед операциями
- Обработка исключительных ситуаций при работе с файлами
- Запрет петель в неориентированных графах

Ограничение: граф не является мультиграфом (ребра уникальны)

Формат файла для сохранения/загрузки:

text

[тип графа] [взвешенность]

вершина1: смежная вершина вес...

вершина 2: ...

Пример:

text

undirected weighted

23: 27 7, 79 10

27: 23 7, 80 5

36: 80 1

57: 80 2

70: 79 1, 80 3

79: 23 10, 70 1

80: 27 5, 36 1, 57 2, 70 3

Все задания данной практики выполнены на языке программирования C#. Далее представлен программный код класса Graph1.

```
1 using System;
```

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Json.Net;


// Тут все для графа

namespace Graph1
{

    // Клаааааас для графа
    public class Graph
    {
        private Dictionary<int, List<Edge>> adjacencyList;
        public bool IsDirected { get; private set; }
        public bool IsWeighted { get; private set; }

        // Конструктор по умолчанию
        public Graph(bool directed = false, bool weighted = false)
        {
            adjacencyList = new Dictionary<int, List<Edge>>();
            IsDirected = directed;
            IsWeighted = weighted;
        }

        // Конструктор из файла

        // Конструктор из файла
        public Graph(string filename)
        {
            adjacencyList = new Dictionary<int, List<Edge>>();

            try
            {
                string[] lines = File.ReadAllLines(filename);

                if (lines.Length < 2)
                    throw new ArgumentException("Файл должен содержать как минимум 2
                    ↪  строки");

```

```
42              // Первая строка - параметры графа
43              string[] parameters = lines[0].Split(' ');
44              if (parameters.Length < 2)
45                  throw new ArgumentException("Неверный формат параметров графа");
46
47              IsDirected = parameters[0] == "directed";
48              IsWeighted = parameters[1] == "weighted";
49
50              // Сначала собираем все вершины
51              var allVertices = new HashSet<int>();
52
53              for (int i = 1; i < lines.Length; i++)
54              {
55                  string[] parts = lines[i].Split(':');
56                  if (parts.Length != 2)
57                      throw new ArgumentException($"Неверный формат строки {i + 1}");
58
59                  int vertex = int.Parse(parts[0].Trim());
60                  allVertices.Add(vertex);
61
62                  if (!string.IsNullOrWhiteSpace(parts[1]))
63                  {
64                      string[] edges = parts[1].Split(',');
65                      foreach (string edgeStr in edges)
66                      {
67                          string[] edgeParts = edgeStr.Trim().Split(' ');
68                          if (edgeParts.Length == 0) continue;
69
70                          int target = int.Parse(edgeParts[0]);
71                          allVertices.Add(target);
72                      }
73                  }
74              }
75
76              // Создаем все вершины
77              foreach (int vertex in allVertices.OrderBy(v => v))
78              {
79                  if (!adjacencyList.ContainsKey(vertex))
80                  {
81                      adjacencyList[vertex] = new List<Edge>();
82                  }
```

```
83              }
84
85              // Добавляем все рёбра без ограничений
86              for (int i = 1; i < lines.Length; i++)
87              {
88                  string[] parts = lines[i].Split(':');
89                  int vertex = int.Parse(parts[0].Trim());
90
91                  if (!string.IsNullOrWhiteSpace(parts[1]))
92                  {
93                      string[] edges = parts[1].Split(',');
94                      foreach (string edgeStr in edges)
95                      {
96                          string[] edgeParts = edgeStr.Trim().Split(' ');
97                          if (edgeParts.Length == 0) continue;
98
99                          int target = int.Parse(edgeParts[0]);
100                         double weight = IsWeighted && edgeParts.Length > 1 ?
101                                 double.Parse(edgeParts[1]) : 1.0;
102
103                         // Убрано условие vertex <= target и suppressReverse
104                         try
105                         {
106                             AddEdge(vertex, target, weight);
107                         }
108                         catch (ArgumentException ex)
109                         {
110                             // Игнорируем только ошибку "ребро уже существует"
111                             if (!ex.Message.Contains("уже существует"))
112                             {
113                                 throw;
114                             }
115                         }
116                     }
117                 }
118             }
119         }
120         catch (Exception ex)
121         {
122             throw new ArgumentException($"Ошибка чтения файла: {ex.Message}");
123         }
```

```
124         }
125
126
127         // Конструктор копирования
128         public Graph(Graph other)
129         {
130             if (other == null)
131                 throw new ArgumentNullException(nameof(other));
132
133             IsDirected = other.IsDirected;
134             IsWeighted = other.IsWeighted;
135             adjacencyList = new Dictionary<int, List<Edge>>();
136
137             foreach (var vertex in other.adjacencyList)
138             {
139                 adjacencyList[vertex.Key] = new List<Edge>();
140                 foreach (var edge in vertex.Value)
141                 {
142                     adjacencyList[vertex.Key].Add(new Edge(edge.Target, edge.Weight));
143                 }
144             }
145         }
146         //
147
148         // Конструктор для тестирования (из списка рёбер)
149         public Graph(List<Tuple<int, int>> edges, bool directed = false, bool weighted = false)
150         {
151             adjacencyList = new Dictionary<int, List<Edge>>();
152             IsDirected = directed;
153             IsWeighted = weighted;
154
155             foreach (var edge in edges)
156             {
157                 AddVertex(edge.Item1);
158                 AddVertex(edge.Item2);
159                 AddEdge(edge.Item1, edge.Item2);
160             }
161         }
162         //
163
164         // Конструктор для тестирования (из матрицы смежности)
```

```
165         public Graph(int[,] matrix, bool directed = false, bool weighted = false)
166         {
167             adjacencyList = new Dictionary<int, List<Edge>>();
168             IsDirected = directed;
169             IsWeighted = weighted;
170 
171             int size = matrix.GetLength(0);
172             for (int i = 0; i < size; i++)
173             {
174                 AddVertex(i);
175                 for (int j = 0; j < size; j++)
176                 {
177                     if (matrix[i, j] != 0)
178                     {
179                         AddEdge(i, j, matrix[i, j]);
180                     }
181                 }
182             }
183         }
184         //
185 
186         // Работа с вершинами //
187 
188         // Добавление вершины
189         public void AddVertex(int vertex)
190         {
191             if (adjacencyList.ContainsKey(vertex))
192                 throw new ArgumentException($"Вершина {vertex} уже существует");
193 
194             adjacencyList[vertex] = new List<Edge>();
195         }
196         //
197 
198         // Удаление вершины
199         public void RemoveVertex(int vertex)
200         {
201             if (!adjacencyList.ContainsKey(vertex))
202                 throw new ArgumentException($"Вершина {vertex} не существует");
203 
204             // Удаляем все рёбра, ведущие к этой вершине
205             foreach (var v in adjacencyList.Keys.ToList())
```

```
206             {
207                 if (v != vertex)
208                 {
209                     adjacencyList[v].RemoveAll(e => e.Target == vertex);
210                 }
211             }
212 
213             adjacencyList.Remove(vertex);
214         }
215         //
216         //
217 
218         // Работа с ребрами/дугами
219 
220         // Добавление ребра/дуги
221         // Добавление ребра/дуги
222         public void AddEdge(int from, int to, double weight = 1.0, bool suppressReverse = false)
223         {
224             if (!adjacencyList.ContainsKey(from))
225                 throw new ArgumentException($"Вершина {from} не существует");
226             if (!adjacencyList.ContainsKey(to))
227                 throw new ArgumentException($"Вершина {to} не существует");
228             if (from == to && !IsDirected)
229                 throw new ArgumentException("Петли разрешены только в ориентированных
                    ↪ графах");
230 
231             // Проверка на существующее ребро (не мультиграф)
232             if (adjacencyList[from].Any(e => e.Target == to))
233                 throw new ArgumentException($"Ребро из {from} в {to} уже существует");
234 
235             adjacencyList[from].Add(new Edge(to, weight));
236 
237             // Для неориентированного графа добавляем обратное ребро
238             if (!IsDirected && from != to && !suppressReverse)
239             {
240                 // Не проверяем существование обратного ребра, чтобы избежать рекурсии
241                 adjacencyList[to].Add(new Edge(from, weight));
242             }
243         }
244         //
245 
```

```
246         // Удаление ребра/дуги
247         public void RemoveEdge(int from, int to)
248         {
249             if (!adjacencyList.ContainsKey(from))
250                 throw new ArgumentException($"Вершина {from} не существует");
251             if (!adjacencyList.ContainsKey(to))
252                 throw new ArgumentException($"Вершина {to} не существует");
253 
254             int removedCount = adjacencyList[from].RemoveAll(e => e.Target == to);
255 
256             if (removedCount == 0)
257                 throw new ArgumentException($"Ребро из {from} в {to} не существует");
258 
259             // Для неориентированного графа удаляем обратное ребро
260             if (!IsDirected && from != to)
261             {
262                 adjacencyList[to].RemoveAll(e => e.Target == from);
263             }
264         }
265         //
266         //
267 
268         // Получение списка смежности в виде строки
269         public string GetAdjacencyListString()
270         {
271             var result = new List<string>();
272             foreach (var vertex in adjacencyList.OrderBy(v => v.Key))
273             {
274                 string edges = string.Join(", ", vertex.Value
275                     .OrderBy(e => e.Target)
276                     .Select(e => IsWeighted ? $"{e.Target} {e.Weight}" : e.Target.ToString()));
277 
278                 result.Add($"{vertex.Key}: {edges}");
279             }
280             return string.Join(Environment.NewLine, result);
281         }
282         //
283 
284         // Запись в файл
285         public void SaveToFile(string filename)
286         {
```

```
287             try
288             {
289                 string output = $"{(IsDirected ? "directed" : "undirected")} {(IsWeighted ?
                    ↪ "weighted" : "unweighted")}\n";
290                 output += GetAdjacencyListString();
291 
292                 File.WriteAllText(filename, output);
293             }
294             catch (UnauthorizedAccessException ex)
295             {
296                 throw new ArgumentException($"Нет прав доступа для записи в файл:
                    ↪ {filename}", ex);
297             }
298             catch (DirectoryNotFoundException ex)
299             {
300                 throw new ArgumentException($"Директория не найдена:
                    ↪ {Path.GetDirectoryName(filename)}", ex);
301             }
302             catch (IOException ex)
303             {
304                 throw new ArgumentException($"Ошибка ввода-вывода при записи файла:
                    ↪ {ex.Message}", ex);
305             }
306             catch (ArgumentException ex)
307             {
308                 throw;
309             }
310             catch (Exception ex)
311             {
312                 throw new ArgumentException($"Неожиданная ошибка при записи в файл:
                    ↪ {ex.Message}", ex);
313             }
314         }
315         //
316 
317         // Получение списка рёбер
318         public List<Tuple<int, int, double>> GetEdgeList()
319         {
320             var edges = new List<Tuple<int, int, double>>();
321             var processed = new HashSet<string>();
322 
```

```
323             foreach (var vertex in adjacencyList)
324             {
325                 foreach (var edge in vertex.Value)
326                 {
327                     string key = IsDirected ? $"{vertex.Key}-{edge.Target}" :
328                         string.Compare(vertex.Key.ToString(), edge.Target.ToString()) < 0 ?
329                         $"{vertex.Key}-{edge.Target}" : $"{edge.Target}-{vertex.Key}";
330 
331                     if (!processed.Contains(key))
332                     {
333                         edges.Add(Tuple.Create(vertex.Key, edge.Target, edge.Weight));
334                         processed.Add(key);
335                     }
336                 }
337             }
338 
339             return edges;
340         }
341         //
342 
343         // Получение вершин
344         public List<int> GetVertices()
345         {
346             return adjacencyList.Keys.OrderBy(v => v).ToList();
347         }
348         //
349 
350         // Получение смежных вершин
351         public List<Edge> GetAdjacentVertices(int vertex)
352         {
353             if (!adjacencyList.ContainsKey(vertex))
354                 throw new ArgumentException($"Вершина {vertex} не существует");
355 
356             return new List<Edge>(adjacencyList[vertex]);
357         }
358         //
359 
360         // Проверка существования вершины
361         public bool ContainsVertex(int vertex)
362         {
363             return adjacencyList.ContainsKey(vertex);
```

```
364         }
365         //
366
367         // Проверка существования ребра
368         public bool ContainsEdge(int from, int to)
369         {
370             return adjacencyList.ContainsKey(from) &&
371                    adjacencyList[from].Any(e => e.Target == to);
372         }
373         //
374
375         // Количество вершин
376         public int VertexCount => adjacencyList.Count;
377         //
378
379         // Количество рёбер
380         public int EdgeCount
381         {
382             get
383             {
384                 int count = adjacencyList.Sum(v => v.Value.Count);
385                 return IsDirected ? count : count / 2;
386             }
387         }
388         //
389
390         // Вложенный класс для хранения рёбер
391         public class Edge
392         {
393             public int Target { get; set; }
394             public double Weight { get; set; }
395
396             public Edge(int target, double weight = 1.0)
397             {
398                 Target = target;
399                 Weight = weight;
400             }
401         }
402         //
403     }
404 }
```

405

2 Консольный интерфейс

Консольный интерфейс реализует взаимодействие пользователя с графом через меню с 9 основными операциями: добавление/удаление вершин и рёбер, вывод графа, сохранение и загрузка из файла, показ информации о графе. Программа работает в бесконечном цикле до выбора опции выхода.

Интерфейс использует объект класса Graph1 для всех операций, обрабатывает пользовательский ввод через консоль и включает защиту от ошибок с помощью try-catch блоков. При работе с рёбрами поддерживается возможность задания веса, а при работе с файлами пользователь может указать имя файла для сохранения или загрузки данных.

Далее представлен программный код создания консольного интерфейса.

```
using Graf1;
using Graph1;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Тут интерфейс
namespace Graph1
{
    class Program
    {
        private static Graph graph;

        // Меню
        static void Main(string[] args)
        {
            Console.WriteLine("=== РАБОТА С ГРАФАМИ ===");
            InitializeGraph();

            bool running = true;
            while (running)
            {
                ShowMenu();
                string choice = Console.ReadLine();

                switch (choice)
```

```
29              {
30                  case "1": AddVertex(); break;
31                  case "2": AddEdge(); break;
32                  case "3": RemoveVertex(); break;
33                  case "4": RemoveEdge(); break;
34                  case "5": ShowAdjacencyList(); break;
35                  case "6": SaveToFile(); break;
36                  case "7": LoadFromFile(); break;
37                  case "8": ShowEdgeList(); break;
38                  case "9": ShowGraphInfo(); break;
39                  case "0": running = false; break;
40                  default: Console.WriteLine("Неверный выбор!"); break;
41              }
42
43              if (running)
44              {
45                  Console.WriteLine("\nНажмите любую клавишу для продолжения...");
46                  Console.ReadKey();
47              }
48          }
49      }
50      //
51
52      //
53      static void InitializeGraph()
54      {
55          Console.WriteLine("0 - Если у вас есть файл txt с графом, вы можете загрузить
            ↪ его из файла");
56          Console.WriteLine("Выберите тип графа:");
57          Console.WriteLine("1 - Неориентированный невзвешенный");
58          Console.WriteLine("2 - Ориентированный невзвешенный");
59          Console.WriteLine("3 - Неориентированный взвешенный");
60          Console.WriteLine("4 - Ориентированный взвешенный");
61          string choice = Console.ReadLine();
62          switch (choice)
63          {
64              case "0":
65                  Console.Write("Введите имя файла: ");
66                  string filename = Console.ReadLine();
67                  try
68                  {
```

```
69                  graph = new Graph(filename);
70                  Console.WriteLine($"Граф загружен из файла {filename}");
71                  return; // Выходим, так как граф уже создан
72              }
73              catch (Exception ex)
74              {
75                  Console.WriteLine($"Ошибка загрузки из файла: {ex.Message}");
76                  Console.WriteLine("Будет создан граф по умолчанию
                    ↪  (неориентированный невзвешенный)");
77                  graph = new Graph(false, false);
78              }
79              break;
80          case "1": graph = new Graph(false, false); break;
81          case "2": graph = new Graph(true, false); break;
82          case "3": graph = new Graph(false, true); break;
83          case "4": graph = new Graph(true, true); break;
84          //case "5": LoadFromFile(); break;
85          default:
86              Console.WriteLine("Создан граф по умолчанию (неориентированный
                ↪  невзвешенный)");
87              graph = new Graph(false, false);
88              break;
89      }
90
91      Console.WriteLine("Граф создан успешно!");
92  }
93  //
94
95  //
96  static void ShowMenu()
97  {
98
99      Console.WriteLine("=== МЕНЮ УПРАВЛЕНИЯ ГРАФОМ ===");
100
101     Console.WriteLine($"Тип графа: {(graph.IsDirected ? "Ориентированный" :
        ↪  "Неориентированный")}, " +
102                 $"{(graph.IsWeighted ? "Взвешенный" : "Невзвешенный")}");
103     Console.WriteLine($"Вершин: {graph.VertexCount}, Рёбер: {graph.EdgeCount}");
104     Console.WriteLine();
105     Console.WriteLine("1. Добавить вершину");
106     Console.WriteLine("2. Добавить ребро/дугу");
```

```
107         Console.WriteLine("3. Удалить вершину");
108         Console.WriteLine("4. Удалить ребро/дугу");
109         Console.WriteLine("5. Показать список смежности");
110         Console.WriteLine("6. Сохранить в файл");
111         Console.WriteLine("7. Загрузить из файла");
112         Console.WriteLine("8. Показать список рёбер");
113         Console.WriteLine("9. Информация о графе");
114 
115         Console.WriteLine("0. Выход");
116         Console.Write("Выберите действие: ");
117     }
118     //
119 
120     //
121     static void AddVertex()
122     {
123         Console.Write("Введите номер вершины: ");
124         if (int.TryParse(Console.ReadLine(), out int vertex))
125         {
126             try
127             {
128                 graph.AddVertex(vertex);
129                 Console.WriteLine($"Вершина {vertex} добавлена успешно!");
130             }
131             catch (Exception ex)
132             {
133                 Console.WriteLine($"Ошибка: {ex.Message}");
134             }
135         }
136         else
137         {
138             Console.WriteLine("Неверный формат числа!");
139         }
140     }
141     //
142 
143     //
144     static void AddEdge()
145     {
146         Console.Write("Введите начальную вершину: ");
147         if (!int.TryParse(Console.ReadLine(), out int from))
```

```
148         {
149             Console.WriteLine("Неверный формат числа!");
150             return;
151         }
152 
153         Console.Write("Введите конечную вершину: ");
154         if (!int.TryParse(Console.ReadLine(), out int to))
155         {
156             Console.WriteLine("Неверный формат числа!");
157             return;
158         }
159 
160         double weight = 1.0;
161         if (graph.IsWeighted)
162         {
163             Console.Write("Введите вес ребра: ");
164             if (!double.TryParse(Console.ReadLine(), out weight))
165             {
166                 Console.WriteLine("Неверный формат веса! Используется вес по умолчанию
                    ↪  1.0");
167                 weight = 1.0;
168             }
169         }
170 
171         try
172         {
173             graph.AddEdge(from, to, weight);
174             Console.WriteLine($"Ребро из {from} в {to} добавлено успешно!");
175         }
176         catch (Exception ex)
177         {
178             Console.WriteLine($"Ошибка: {ex.Message}");
179         }
180     }
181     //
182 
183     //
184     static void RemoveVertex()
185     {
186         Console.Write("Введите номер вершины для удаления: ");
187         if (int.TryParse(Console.ReadLine(), out int vertex))
```

```
188         {
189             try
190             {
191                 graph.RemoveVertex(vertex);
192                 Console.WriteLine($"Вершина {vertex} удалена успешно!");
193             }
194             catch (Exception ex)
195             {
196                 Console.WriteLine($"Ошибка: {ex.Message}");
197             }
198         }
199         else
200         {
201             Console.WriteLine("Неверный формат числа!");
202         }
203     }
204     //
205 
206     //
207     static void RemoveEdge()
208     {
209         Console.Write("Введите начальную вершину: ");
210         if (!int.TryParse(Console.ReadLine(), out int from))
211         {
212             Console.WriteLine("Неверный формат числа!");
213             return;
214         }
215 
216         Console.Write("Введите конечную вершину: ");
217         if (!int.TryParse(Console.ReadLine(), out int to))
218         {
219             Console.WriteLine("Неверный формат числа!");
220             return;
221         }
222 
223         try
224         {
225             graph.RemoveEdge(from, to);
226             Console.WriteLine($"Ребро из {from} в {to} удалено успешно!");
227         }
228         catch (Exception ex)
```

```
229             {
230                 Console.WriteLine($"Ошибка: {ex.Message}");
231             }
232         }
233         //
234 
235         //
236         static void ShowAdjacencyList()
237         {
238             Console.WriteLine("Список смежности:");
239             Console.WriteLine(graph.GetAdjacencyListString());
240         }
241         //
242 
243         //
244         static void SaveToFile()
245         {
246             Console.Write("Введите имя файла: ");
247             string filename = Console.ReadLine();
248 
249             try
250             {
251                 graph.SaveToFile(filename);
252                 Console.WriteLine($"Граф сохранён в файл {filename}");
253             }
254             catch (Exception ex)
255             {
256                 Console.WriteLine($"Ошибка: {ex.Message}");
257             }
258         }
259         //
260 
261         //
262         static void LoadFromFile()
263         {
264             Console.Write("Введите имя файла: ");
265             string filename = Console.ReadLine();
266 
267             try
268             {
269                 graph = new Graph(filename);
```

```
270             Console.WriteLine($"Граф загружен из файла {filename}");
271         }
272         catch (Exception ex)
273         {
274             Console.WriteLine($"Ошибка: {ex.Message}");
275         }
276     }
277     //
278 
279     //
280     static void ShowEdgeList()
281     {
282         var edges = graph.GetEdgeList();
283         Console.WriteLine("Список рёбер:");
284         foreach (var edge in edges)
285         {
286             if (graph.IsWeighted)
287                 Console.WriteLine($"{edge.Item1} -> {edge.Item2} (вес: {edge.Item3})");
288             else
289                 Console.WriteLine($"{edge.Item1} -> {edge.Item2}");
290         }
291     }
292     //
293 
294 
295     static void ShowGraphInfo()
296     {
297         Console.WriteLine("=== ИНФОРМАЦИЯ О ГРАФЕ ===");
298         Console.WriteLine($"Тип: {(graph.IsDirected ? "Ориентированный" :
            ↪  "Неориентированный")}");
299         Console.WriteLine($"Взвешенный: {(graph.IsWeighted ? "Да" : "Нет")}");
300         Console.WriteLine($"Количество вершин: {graph.VertexCount}");
301         Console.WriteLine($"Количество рёбер: {graph.EdgeCount}");
302 
303         var vertices = graph.GetVertices();
304         Console.WriteLine($"Вершины: {string.Join(", ", vertices)}");
305 
306         List<int> isolated = new List<int>();
307 
308         if (!graph.IsDirected)
309         {
```

```
310             // Для неориентированного графа: вершина без смежных вершин
311             isolated = vertices.Where(v => graph.GetAdjacentVertices(v).Count ==
                ↪  0).ToList();
312         }
313         else
314         {
315             // Для ориентированного графа: нет ни входящих, ни исходящих рёбер
316             foreach (int vertex in vertices)
317             {
318                 int outDegree = graph.GetAdjacentVertices(vertex).Count;
319                 int inDegree = CalculateInDegree(vertex); // Нужен вспомогательный метод
320 
321                 if (outDegree == 0 && inDegree == 0)
322                 {
323                     isolated.Add(vertex);
324                 }
325             }
326         }
327 
328         if (isolated.Count > 0)
329             Console.WriteLine($"Изолированные вершины: {string.Join(", ", isolated)}");
330 
331         //
332     }
333 
334     // Вспомогательный метод для расчета входящей степени вершины
335     private static int CalculateInDegree(int vertex)
336     {
337         if (!graph.ContainsVertex(vertex))
338             return 0;
339 
340         int inDegree = 0;
341         foreach (int otherVertex in graph.GetVertices())
342         {
343             if (graph.ContainsEdge(otherVertex, vertex))
344             {
345                 inDegree++;
346             }
347         }
348         return inDegree;
349     }
```

```
350
351
352         }
353     }
```

На рисунках 1, 2 представлены примеры работы с графом через консольный интерфейс.

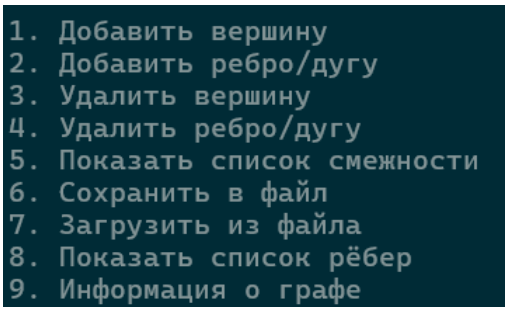

Рисунок 1 – Консольный интерфейс

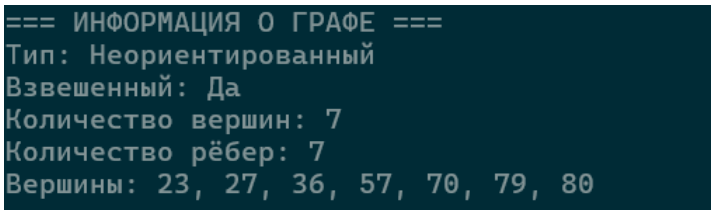

Рисунок 2 – Информация о графе

3 Список смежности

3.1 Задания

1. Для каждой вершины графа вывести её степень.
2. Вывести все висячие вершины графа (степени 1).
3. Построить орграф, полученный из исходного удалением изолированных вершин.

3.2 Реализация

Задание 1

Класс DegreeCalculator предназначен для расчёта степеней вершин в графе, представленном через структуру Graph1. Он поддерживает как неориентированные, так и ориентированные графы.

Основные методы:

— CalculateAndPrintDegrees() - Вычисляет и выводит в консоль степени (или полустепени) всех вершин графа. Для неориентированных графов выводит степени вершин, для ориентированных - исходящую, входящую и общую степень.

— GetDegreeDictionary() - Возвращает словарь, где ключами являются вершины, а значениями - их степени (или суммарные степени для ориентированного графа).

Класс DegreeCalculator предоставляет удобный интерфейс для анализа структуры графа через степени его вершин, учитывая особенности ориентированных и неориентированных графов.

Далее представлен программный код всех методов.

```
1
2       public DegreeCalculator(Graph graph)
3       {
4           this.graph = graph ?? throw new ArgumentNullException(nameof(graph));
5       }
6
7       // Метод для вычисления и вывода степени каждой вершины
8       public void CalculateAndPrintDegrees()
9       {
10          if (graph.VertexCount == 0)
11          {
12              Console.WriteLine("Граф пуст.");
```

```
13              return;
14          }
15
16          var vertices = graph.GetVertices();
17
18          if (!graph.IsDirected)
19          {
20              // Для неориентированного графа степень вершины равна количеству
                ↪ смежных рёбер
21              Console.WriteLine("=== СТЕПЕНИ ВЕРШИН (неориентированный граф)
                ↪ ===");
22              foreach (var vertex in vertices)
23              {
24                  int degree = graph.GetAdjacentVertices(vertex).Count;
25                  Console.WriteLine($"Вершина {vertex}: степень {degree}");
26              }
27          }
28          else
29          {
30              // Для ориентированного графа считаем полустепени исхода и захода
31              Console.WriteLine("=== СТЕПЕНИ ВЕРШИН (ориентированный граф)
                ↪ ===");
32              Console.WriteLine("(Полустепень исхода / Полустепень захода / Суммарная
                ↪ степень)");
33
34              foreach (var vertex in vertices)
35              {
36                  int outDegree = graph.GetAdjacentVertices(vertex).Count; // смежные
                    ↪ вершины - это исходящие рёбра
37                  int inDegree = CalculateInDegree(vertex); // нужно посчитать входящие
                    ↪ рёбра
38                  int totalDegree = outDegree + inDegree;
39
40                 Console.WriteLine($"Вершина {vertex}: исходящая = {outDegree}, входящая
                    ↪ = {inDegree}, общая = {totalDegree}");
41              }
42          }
43      }
44
45      // Вспомогательный метод для вычисления полустепени захода в ориентированном
        ↪ графе
```

```
46      private int CalculateInDegree(int vertex)
47      {
48          if (!graph.ContainsVertex(vertex))
49              return 0;
50
51          int inDegree = 0;
52          var allVertices = graph.GetVertices();
53
54          foreach (var v in allVertices)
55          {
56              if (v != vertex)
57              {
58                  // Проверяем, есть ли ребро из v в vertex
59                  if (graph.ContainsEdge(v, vertex))
60                      inDegree++;
61              }
62              else
63              {
64                  // Петля учитывается и как входящее, и как исходящее ребро
65                  if (graph.ContainsEdge(vertex, vertex))
66                      inDegree++;
67              }
68          }
69
70          return inDegree;
71      }
72
73      // Метод для получения словаря степеней (может пригодиться для других
        ↪ операций)
74      public Dictionary<int, int> GetDegreeDictionary()
75      {
76          var degreeDict = new Dictionary<int, int>();
77          var vertices = graph.GetVertices();
78
79          if (!graph.IsDirected)
80          {
81              foreach (var vertex in vertices)
82              {
83                  degreeDict[vertex] = graph.GetAdjacentVertices(vertex).Count;
84              }
85          }
```

```
86              else
87              {
88                  foreach (var vertex in vertices)
89                  {
90                      int outDegree = graph.GetAdjacentVertices(vertex).Count;
91                      int inDegree = CalculateInDegree(vertex);
92                      degreeDict[vertex] = outDegree + inDegree;
93                  }
94              }
95
96              return degreeDict;
97          }
98
99          // Статический метод для быстрого использования без создания экземпляра класса
100         public static void CalculateDegrees(Graph graph)
101         {
102             var calculator = new DegreeCalculator(graph);
103             calculator.CalculateAndPrintDegrees();
104         }
105     }
106
107 }
```

Задание 2

Класс PendantVertices предназначен для обнаружения и анализа висячих вершин в графе (вершин степени 1). Он корректно обрабатывает как неориентированные, так и ориентированные графы. Используемые методы из класса Graph1:

Основные методы

- FindAndPrintPendantVertices() - Основной метод для поиска и вывода всех висячих вершин в графе.
- GetPendantVertices() - Возвращает упорядоченный список всех висячих вершин графа.
- IsPendantVertex(int vertex) Проверяет, является ли заданная вершина висячей (степени 1).
- GetTotalDegree(int vertex) - Вычисляет общую степень вершины.
- GetInDegree(int vertex) - Вычисляет количество входящих рёбер для вершины в ориентированном графе.
- PrintVertexInfo(int vertex) - Выводит детальную информацию о висячей

вершине.

Класс PendantVertices обеспечивает полный анализ висячих вершин, включая детализацию связей и статистику, что делает его удобным инструментом для изучения структуры графа.

Далее представлен код метода FindAndPrintPendantVertices

```
1
2       // Основной метод: найти и вывести все висячие вершины
3       public void FindAndPrintPendantVertices()
4       {
5           Console.WriteLine("=== ПОИСК ВИСЯЧИХ ВЕРШИН (СТЕПЕНИ 1) ===");
6
7           if (graph == null || graph.VertexCount == 0)
8           {
9               Console.WriteLine("Граф пуст или не создан.");
10              return;
11          }
12
13          // Получаем список всех висячих вершин
14          List<int> pendantVertices = GetPendantVertices();
15
16          // Выводим результат
17          if (pendantVertices.Count == 0)
18          {
19              Console.WriteLine("В графе нет висячих вершин (степени 1).");
20          }
21          else
22          {
23              Console.WriteLine($"Найдено {pendantVertices.Count} висячих вершин:");
24              foreach (int vertex in pendantVertices)
25              {
26                  PrintVertexInfo(vertex);
27              }
28          }
29
30          // Дополнительная статистика
31          PrintStatistics(pendantVertices);
32      }
```

Задание 3

Класс IsolatedVerticesRemover предназначен для создания нового графа,

в котором удалены все изолированные вершины (вершины без инцидентных рёбер). Класс корректно обрабатывает как неориентированные, так и ориентированные графы.

Основные методы

- public Graph RemoveIsolatedVertices() - Создаёт и возвращает новый граф без изолированных вершин. Вызывает поиск изолированных вершин и копирует остальные вершины и рёбра.
- public List<int> GetIsolatedVertices() - Возвращает список всех изолированных вершин в исходном графе. Перебирает все вершины и проверяет каждую через IsIsolatedVertex.
- public bool IsIsolatedVertex(int vertex) -Проверяет, является ли указанная вершина изолированной. Для неориентированных графов: степень = 0. Для ориентированных графов: полустепени исхода и захода = 0.

Класс IsolatedVerticesRemover предоставляет безопасный и эффективный способ очистки графа от изолированных вершин, сохраняя при этом все остальные свойства и связи.

Далее представлен код метода IsIsolatedVertex, который позволяет проверить изолирована ли вершина или нет.

```
1
2       // Проверка, является ли вершина изолированной
3       public bool IsIsolatedVertex(int vertex)
4       {
5           if (!originalGraph.ContainsVertex(vertex))
6               return false;
7
8           if (!originalGraph.IsDirected)
9           {
10              // Для неориентированного графа: нет инцидентных ребер
11              return originalGraph.GetAdjacentVertices(vertex).Count == 0;
12          }
13          else
14          {
15              // Для ориентированного графа: нет входящих и исходящих ребер
16              int outDegree = originalGraph.GetAdjacentVertices(vertex).Count;
17              int inDegree = CalculateInDegree(vertex);
18              return outDegree == 0 && inDegree == 0;
19          }
20      }
```

3.3 Примеры

```
=== СТЕПЕНИ ВЕРШИН (неориентированный граф) ===
Вершина 23: степень 2
Вершина 27: степень 2
Вершина 36: степень 1
Вершина 57: степень 1
Вершина 70: степень 2
Вершина 79: степень 2
Вершина 80: степень 4
```

Рисунок 3 – Задание 1

```
=== СТАТИСТИКА ===
Всего вершин в графе: 7
Висячих вершин: 2
Процент висячих вершин: 28,6%
Список всех висячих вершин: 36, 57
```

Рисунок 4 – Задание 2

```
=== УДАЛЕНИЕ ИЗОЛИРОВАННЫХ ВЕРШИН ===

Исходный граф:
Тип: Неориентированный
Вершин: 7, Рёбер: 7
Изолированных вершин нет.
Найдено изолированных вершин: 0

Новый граф (без изолированных вершин):
Тип: Неориентированный
Вершин: 7, Рёбер: 7

Список смежности нового графа:
23: 27 7, 79 10
27: 23 7, 80 5
36: 80 1
57: 80 2
70: 79 1, 80 3
79: 23 10, 70 1
80: 27 5, 36 1, 57 2, 70 3

Сравнение:
Удалено вершин: 0
Удалено рёбер: 0
```

Рисунок 5 – Задание 3

4 Обходы графа

4.1 Задания

1. Корнем ацикличного орграфа называется такая вершина u, что из неё существуют пути в каждую из остальных вершин орграфа. Определить,

имеет ли данный ацикличный орграф корень.

2. Проверить, можно ли из графа удалить какую-либо вершину так, чтобы получилось дерево.

4.2 Реализация

Задание 1

Вспомогательный статический класс для работы с ациклическими ориентированными графами. Содержит методы для поиска корневой вершины (вершины, из которой достижимы все остальные) и проверки графа на ацикличность.

Основные методы класса

— FindRootInDAG(Graph graph) - Основной метод поиска корня в ациклическом орграфе. Проверяет ориентированность и отсутствие циклов, затем ищет вершины, из которых достижимы все остальные.

— IsRoot(Graph graph, int startVertex) - Проверяет, является ли вершина startVertex корнем, выполняя обход в глубину (DFS) и сравнивая число посещённых вершин с общим числом вершин.

— DFS(Graph graph, int vertex, HashSet<int> visited) - Рекурсивный обход в глубину для посещения всех достижимых вершин.

— IsAcyclic(Graph graph) - Проверяет граф на ацикличность с помощью DFS с отслеживанием вершин в стеке рекурсии.

— HasCycleDFS(...) - Вспомогательный рекурсивный метод для обнаружения циклов в графе.

— FindRootAlternative(Graph graph) - Альтернативный метод поиска корня через вершины с нулевой полустепенью захода. Выводит степени захода всех вершин и проверяет кандидатов на роль корня.

Далее представлен код метода FindRootInDAG

```
1 // Основной метод для проверки наличия корня в ацикличном орграфе
2     public static void FindRootInDAG(Graph graph)
3     {
4         if (graph == null)
5         {
6             Console.WriteLine("Граф не создан!");
7             return;
8         }
9
```

```
10          // Проверяем, что граф ориентированный
11          if (!graph.IsDirected)
12          {
13              Console.WriteLine("Граф должен быть ориентированным!");
14              return;
15          }
16
17          // Проверяем, что граф ацикличен
18          if (!IsAcyclic(graph))
19          {
20              Console.WriteLine("Граф содержит циклы! Алгоритм работает только для
                ↪ ациклических графов.");
21              return;
22          }
23
24          // Получаем список вершин
25          var vertices = graph.GetVertices();
26
27          // Для каждой вершины проверяем, является ли она корнем
28          var potentialRoots = new List<int>();
29
30          foreach (var vertex in vertices)
31          {
32              if (IsRoot(graph, vertex))
33              {
34                  potentialRoots.Add(vertex);
35              }
36          }
37
38          // Выводим результат
39          if (potentialRoots.Count == 0)
40          {
41              Console.WriteLine("В орграфе нет корня.");
42          }
43          else if (potentialRoots.Count == 1)
44          {
45              Console.WriteLine($"Корень найден: вершина {potentialRoots[0]}");
46          }
47          else
48          {
49              Console.WriteLine($"Найдено несколько вершин-корней: {string.Join(", ",
                ↪ potentialRoots)}");
```

```
        }
    }
```

Задание 2

Статический класс для проверки возможности преобразования графа в дерево путём удаления одной вершины. Поддерживает ориентированные и неориентированные графы

Основные методы

- CheckVertexRemoveForTree - выводит характеристики графа и запускает соответствующую проверку.
- CheckForDirectedGraph - Проверяет для ориентированного графа возможность получения ориентированного дерева.
- IsTree - Проверяет, является ли неориентированный граф деревом (связный + без циклов).
- IsDirectedTree - Проверяет, является ли ориентированный граф ориентированным деревом.
- FindBestVertexToRemove - Выбирает лучшую вершину для удаления (сохраняет максимум вершин).

Далее представлен код метода CheckVertexRemoveForTree

```
// Основной метод проверки
    public static void CheckVertexRemoveForTree(Graph graph)
    {
        if (graph == null)
        {
            Console.WriteLine("Граф не создан!");
            return;
        }

        Console.WriteLine("=== ПРОВЕРКА ===");

        var vertices = graph.GetVertices();

        // Вычисляем характеристики исходного графа
        Console.WriteLine($"Характеристики исходного графа:");
        Console.WriteLine($"Количество вершин: {graph.VertexCount}");
```

```
17          Console.WriteLine($"Количество рёбер: {graph.EdgeCount}");
18          Console.WriteLine($"Ориентированный: {(graph.IsDirected ? "да" : "нет")}");
19          Console.WriteLine($"Взвешенный: {(graph.IsWeighted ? "да" : "нет")}");
20
21          // Для неориентированных графов
22          if (!graph.IsDirected)
23          {
24              CheckForUndirectedGraph(graph);
25          }
26          else
27          {
28              CheckForDirectedGraph(graph);
29          }
30      }
```

4.3 Примеры

```
=== МЕНЮ УПРАВЛЕНИЯ ГРАФОМ ===
Тип графа: Ориентированный, Взвешенный
Вершин: 7, Рёбер: 6

1. Добавить вершину
2. Добавить ребро/дугу
3. Удалить вершину
4. Удалить ребро/дугу
5. Показать список смежности
6. Сохранить в файл
7. Загрузить из файла
8. Показать список рёбер
9. Информация о графе
10. Вывести степени вершин
11. Найти весячие вершины (степени 1)
12. Удалить изолированные вершины и создать новый граф
13. Найти корень в ацикличном орграфе
14. Проверить возможность получения дерева удалением вершины
15. Найти минимальный остовный каркас (алгоритм Краскала)
16. Найти кратчайшие пути для всех пар вершин (алгоритм Дейкстры)
17. Найти кратчайшие пути (Беллмана-Форда)
18. Найти кратчайшие пути для всех пар вершин (алгоритм Флойда-Уоршелла)
19. Найти максимальный поток (алгоритм Форда-Фалкерсона)
0. Выход
Выберите действие: 13
Корень найден: вершина 23
```

Рисунок 6 – Задание 1

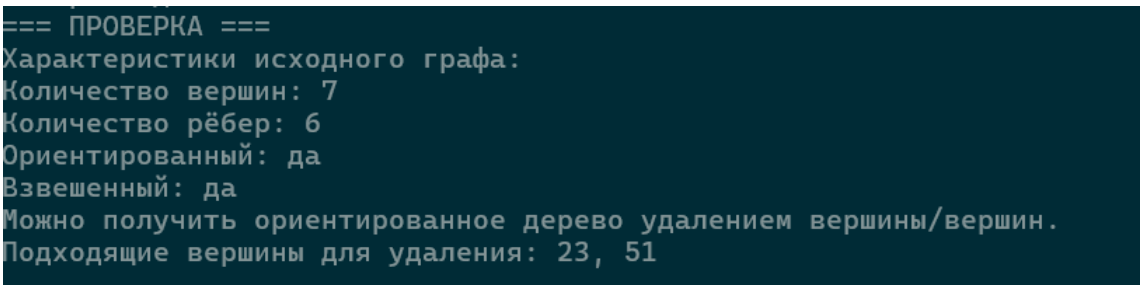

Рисунок 7 – Задание 2

5 Построение минимального остовного дерева

5.1 Задания

1. Дан взвешенный неориентированный граф из N вершин и M ребер. Требуется найти в нем каркас минимального веса. Алгоритм Краскала.

5.2 Реализация

Статический класс для нахождения минимального остовного дерева (МОД) невзвешенного неориентированного графа с использованием алгоритма Краскала.

Основной метод

FindMinimumSpanningTree(Graph graph)

Находит минимальное остовное дерево алгоритмом Краскала:

1. Проверяет, что граф неориентированный и взвешенный.
2. Собирает все рёбра, удаляет дубликаты (From < To).
3. Сортирует рёбра по весу.
4. Использует DSU для последовательного добавления рёбер, не образующих циклов.
5. Строит результирующий граф-дерево, выводит список рёбер, общий вес и список смежности.
6. Возвращает граф, представляющий минимальное остовное дерево (или лес, если граф несвязный).

Код метода FindMinimumSpanningTree

```
1
2       /// <param name="graph">Исходный граф</param>
3       /// <returns>Новый граф, представляющий минимальное остовное
        ↪  дерево</returns>
4       public static Graph FindMinimumSpanningTree(Graph graph)
5       {
```

```
6           // Проверка, что граф неориентированный и взвешенный
7           if (graph.IsDirected)
8           {
9               throw new ArgumentException("Алгоритм Краскала работает только с
                ↪ неориентированными графами");
10          }
11
12          if (!graph.IsWeighted)
13          {
14              throw new ArgumentException("Граф должен быть взвешенным для
                ↪ поиска минимального остовного дерева");
15          }
16
17          // Получение всех вершин
18          var vertices = graph.GetVertices();
19          if (vertices.Count == 0)
20          {
21              throw new ArgumentException("Граф пустой");
22          }
23
24          // Создания словарь для отображения номеров вершин в индексы от 0 до
            ↪ N-1
25          Dictionary<int, int> vertexToIndex = new Dictionary<int, int>();
26          Dictionary<int, int> indexToVertex = new Dictionary<int, int>();
27
28          for (int i = 0; i < vertices.Count; i++)
29          {
30              vertexToIndex[vertices[i]] = i;
31              indexToVertex[i] = vertices[i];
32          }
33
34          // Собираем все ребра
35          List<EdgeInfo> allEdges = new List<EdgeInfo>();
36
37          foreach (var vertex in vertices)
38          {
39              var adjacentEdges = graph.GetAdjacentVertices(vertex);
40              foreach (var edge in adjacentEdges)
41              {
42                  // Добавляем каждое ребро только один раз (From < To)
43                  if (vertex < edge.Target)
```

```
44              {
45                  allEdges.Add(new EdgeInfo(vertex, edge.Target, edge.Weight));
46              }
47          }
48      }
49
50      // Сортируем ребра по весу
51      allEdges.Sort();
52
53      // Инициализируем DSU
54      DisjointSetUnion dsu = new DisjointSetUnion(vertices.Count);
55
56      // Создаем новый граф для остовного дерева
57      Graph mstGraph = new Graph(false, true);
58
59      // Добавляем все вершины в новый граф
60      foreach (var vertex in vertices)
61      {
62          mstGraph.AddVertex(vertex);
63      }
64
65      double totalWeight = 0;
66      int edgesAdded = 0;
67      List<EdgeInfo> mstEdges = new List<EdgeInfo>();
68
69      // Проходим по всем ребрам в порядке возрастания веса
70      foreach (var edge in allEdges)
71      {
72          int indexFrom = vertexToIndex[edge.From];
73          int indexTo = vertexToIndex[edge.To];
74
75          // Если ребро соединяет разные компоненты связности
76          if (dsu.Union(indexFrom, indexTo))
77          {
78              // Добавляем ребро в остовное дерево
79              mstGraph.AddEdge(edge.From, edge.To, edge.Weight);
80              mstEdges.Add(edge);
81              totalWeight += edge.Weight;
82              edgesAdded++;
83
84              // Если добавили N-1 ребро, остовное дерево построено
```

```
85                  if (edgesAdded == vertices.Count - 1)
86                      break;
87              }
88          }
89
90          // Проверка, что граф связный (для связного графа должно быть ровно
            ↪ N-1 ребро в MST)
91          if (edgesAdded < vertices.Count - 1)
92          {
93              Console.WriteLine("Внимание: Граф не является связным. Построен
                ↪ минимальный остовный лес.");
94          }
95
96          // Вывод
97          Console.WriteLine("=== РЕЗУЛЬТАТЫ АЛГОРИТМА КРАСКАЛА
            ↪ ===");
98          Console.WriteLine($"Всего вершин: {vertices.Count}");
99          Console.WriteLine($"Всего ребер в исходном графе: {graph.EdgeCount}");
100         Console.WriteLine($"Ребер в минимальном остовном дереве:
            ↪ {edgesAdded}");
101         Console.WriteLine($"Общий вес остовного дерева: {totalWeight:F2}");
102         Console.WriteLine();
103         Console.WriteLine("Ребра минимального остовного дерева:");
104
105         foreach (var edge in mstEdges.OrderBy(e => e.From).ThenBy(e => e.To))
106         {
107             Console.WriteLine($"{edge.From} -- {edge.To} (вес: {edge.Weight:F2})");
108         }
109
110         Console.WriteLine();
111         Console.WriteLine("Список смежности минимального остовного дерева:");
112         Console.WriteLine(mstGraph.GetAdjacencyListString());
113
114         return mstGraph;
115     }
```

5.3 Примеры

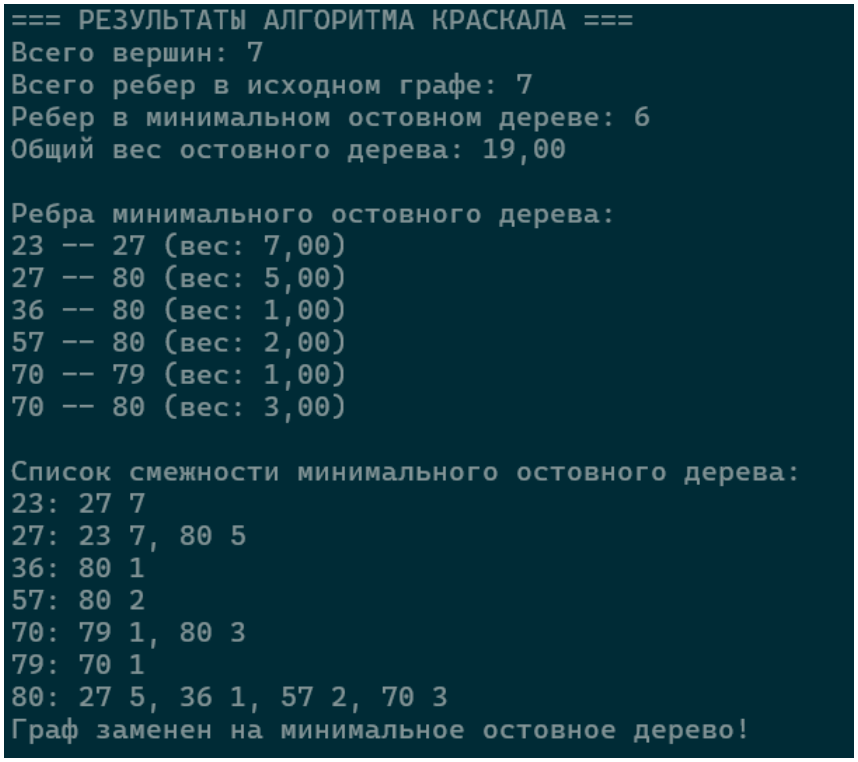

Рисунок 8 – Задание

6 Взвешенный граф

6.1 Задания

1. Вывести длины кратчайших путей для всех пар вершин. Алгоритм Дейкстры.
2. Вывести длины кратчайших путей от u до всех остальных вершин. Алгоритм Беллман-Форд
3. Вывести кратчайшие пути до вершины u из всех остальных вершин. Алгоритм Флойда.

6.2 Реализация

Задание 1

Статический класс для нахождения кратчайших путей во взвешенном неориентированном графе с неотрицательными весами рёбер с использованием алгоритма Дейкстры.

Вложенный класс:

DijkstraResult

Содержит результаты алгоритма Дейкстры для одной исходной вершины:

— SourceVertex - исходная вершина
— Distances - словарь кратчайших расстояний до всех вершин
— PreviousVertices - словарь предыдущих вершин для восстановления путей

Основные методы:

1. FindShortestPathsFromVertex(Graph graph, int source)

   Находит кратчайшие пути от исходной вершины source до всех остальных:
   — Проверяет наличие отрицательных весов
   — Использует приоритетную очередь для выбора следующей вершины
   — Возвращает объект DijkstraResult
2. FindAllPairsShortestPaths(Graph graph)

   Находит кратчайшие пути между всеми парами вершин:
   — Запускает алгоритм Дейкстры из каждой вершины
   — Выводит матрицу расстояний
   — Анализирует связность графа
   — Возвращает словарь результатов для всех исходных вершин
3. ReconstructPath(DijkstraResult result, int target)

   Восстанавливает кратчайший путь от исходной вершины (из result) до целевой target:
   — Использует словарь PreviousVertices
   — Возвращает список вершин в порядке следования или пустой список, если путь не существует

Код метода FindShortestPathsFromVertex

```
1
2   public static DijkstraResult FindShortestPathsFromVertex(Graph graph, int source)
3   {
4       if (graph == null)
5           throw new ArgumentNullException(nameof(graph));
6
7       if (!graph.ContainsVertex(source))
8           throw new ArgumentException($"Вершина {source} не существует в графе");
9
10      // Проверка на отрицательные веса (хотя по условию их нет)
11      foreach (var vertex in graph.GetVertices())
12      {
13          foreach (var edge in graph.GetAdjacentVertices(vertex))
14          {
15              if (edge.Weight < 0)
```

```
16                  throw new ArgumentException("Граф содержит ребра с отрицательным
                    ↪ весом. Алгоритм Дейкстры не применим.");
17              }
18          }
19
20          var vertices = graph.GetVertices();
21          var result = new DijkstraResult(source);
22
23          // Инициализация
24          var unvisitedVertices = new HashSet<int>(vertices);
25          var priorityQueue = new SortedSet<Tuple<double, int>>();
26
27          // Установка начальные расстояния
28          foreach (var vertex in vertices)
29          {
30              if (vertex == source)
31              {
32                  result.Distances[vertex] = 0;
33                  priorityQueue.Add(Tuple.Create(0.0, vertex));
34              }
35              else
36              {
37                  result.Distances[vertex] = double.PositiveInfinity;
38                  priorityQueue.Add(Tuple.Create(double.PositiveInfinity, vertex));
39              }
40              result.PreviousVertices[vertex] = -1;
41          }
42
43          // Основной цикл алгоритма
44          while (unvisitedVertices.Count > 0 && priorityQueue.Count > 0)
45          {
46              // Извлечение вершины с минимальным расстоянием
47              var currentTuple = priorityQueue.Min;
48              priorityQueue.Remove(currentTuple);
49
50              double currentDistance = currentTuple.Item1;
51              int currentVertex = currentTuple.Item2;
52
53              // Если вершина уже посещена, пропускаем
54              if (!unvisitedVertices.Contains(currentVertex))
55                  continue;
```

```
56
57              unvisitedVertices.Remove(currentVertex);
58
59              // Если расстояние до текущей вершины бесконечно, значит остальные
                ↪  вершины недостижимы
60              if (currentDistance == double.PositiveInfinity)
61                  break;
62
63              // Рассматриваем всех соседей текущей вершины
64              foreach (var edge in graph.GetAdjacentVertices(currentVertex))
65              {
66                  int neighbor = edge.Target;
67
68                  // Если сосед еще не посещен
69                  if (unvisitedVertices.Contains(neighbor))
70                  {
71                      double newDistance = currentDistance + edge.Weight;
72
73                      if (newDistance < result.Distances[neighbor])
74                      {
75                          // Обновляем расстояние
76                       priorityQueue.Remove(Tuple.Create(result.Distances[neighbor], neighbor));
77                          result.Distances[neighbor] = newDistance;
78                          priorityQueue.Add(Tuple.Create(newDistance, neighbor));
79                          result.PreviousVertices[neighbor] = currentVertex;
80                      }
81                  }
82              }
83          }
84
85          return result;
86      }
87
88
```

Задание 2

Класс BellmanFordAlgorithm Статический класс для реализации алгоритма Беллмана-Форда нахождения кратчайших путей во взвешенном графе с возможностью обнаружения циклов отрицательного веса. Работает с ориентированными и неориентированными графами.

Вспомогательный класс BellmanFordResult

Хранит результаты выполнения алгоритма:

- StartVertex - исходная вершина
- Distances - словарь кратчайших расстояний до всех вершин
- Predecessors - словарь предыдущих вершин для восстановления путей

Внутренний класс EdgeData Представляет ребро графа с полями From, To, Weight.

Основные методы

1. FindShortestPaths(Graph graph, int startVertex)
   Основной метод алгоритма Беллмана-Форда:
   - Проверяет корректность входных данных
   - Инициализирует расстояния и предшественников
   - Выполняет релаксацию рёбер (V-1) раз
   - Обнаруживает циклы отрицательного веса
   - Возвращает BellmanFordResult или null при наличии цикла отрицательного веса
2. GetAllEdges(Graph graph)
   Вспомогательный метод: собирает список всех уникальных рёбер графа, учитывая ориентацию.
3. CheckNegativeCycle(Graph graph, List<EdgeData> edges, Dictionary<int, double> distances)
   Проверяет наличие циклов отрицательного веса после выполнения основной части алгоритма.
4. DisplayResults(Graph graph, int startVertex)
   Выводит результаты алгоритма в консоль в табличном формате: расстояния и пути до всех вершин.
5. ReconstructPath(BellmanFordResult result, int targetVertex)
   Восстанавливает путь от начальной вершины до целевой в виде строки.
6. GetPath(BellmanFordResult result, int targetVertex)
   Восстанавливает путь от начальной вершины до целевой в виде списка вершин.

Код метода FindShortestPaths

```
1
2       public static BellmanFordResult FindShortestPaths(Graph graph, int startVertex)
3       {
4           if (graph == null)
```

```
5           throw new ArgumentNullException(nameof(graph));
6
7       if (!graph.ContainsVertex(startVertex))
8           throw new ArgumentException($"Вершина {startVertex} не существует в
            ↪ графе");
9
10      if (!graph.IsWeighted)
11          throw new ArgumentException("Алгоритм Беллмана-Форда требует
            ↪ взвешенный граф");
12
13      var vertices = graph.GetVertices();
14      int n = vertices.Count;
15
16      // Инициализация расстояний
17      var distances = new Dictionary<int, double>();
18      var predecessors = new Dictionary<int, int?>();
19
20      foreach (int vertex in vertices)
21      {
22          distances[vertex] = double.PositiveInfinity;
23          predecessors[vertex] = null;
24      }
25      distances[startVertex] = 0;
26
27      // Получаем список всех рёбер
28      var edges = GetAllEdges(graph);
29
30      // Основная часть алгоритма: релаксация рёбер (n-1) раз
31      for (int i = 0; i < n - 1; i++)
32      {
33          bool anyDistanceChanged = false;
34
35          foreach (var edge in edges)
36          {
37              int u = edge.From;
38              int v = edge.To;
39              double weight = edge.Weight;
40
41              if (distances[u] + weight < distances[v])
42              {
43                  distances[v] = distances[u] + weight;
```

```
44                      predecessors[v] = u;
45                      anyDistanceChanged = true;
46                  }
47
48                  // Для неориентированных графов проверяем также обратное направление
49                  if (!graph.IsDirected)
50                  {
51                      if (distances[v] + weight < distances[u])
52                      {
53                          distances[u] = distances[v] + weight;
54                          predecessors[u] = v;
55                          anyDistanceChanged = true;
56                      }
57                  }
58              }
59
60              // Если на данной итерации не изменилось ни одно расстояние, можно
                ↪ остановиться
61              if (!anyDistanceChanged)
62                  break;
63          }
64
65          // Проверка на наличие циклов отрицательного веса
66          bool hasNegativeCycle = CheckNegativeCycle(graph, edges, distances);
67
68          if (hasNegativeCycle)
69          {
70              return null; // Цикл отрицательного веса обнаружен
71          }
72
73          return new BellmanFordResult(startVertex, distances, predecessors);
74      }
```

Задание 3

Класс FloydAlgorithm

Статический класс для реализации алгоритма Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин во взвешенном графе (ориентированном или неориентированном). Обнаруживает циклы отрицательного веса.

Вложенный класс FloydResult

Содержит результаты выполнения алгоритма:

— Distances - матрица кратчайших расстояний между всеми парами вершин
— Predecessors - матрица предшественников для восстановления путей
— HasNegativeCycle - флаг наличия цикла отрицательного веса
— NegativeCycle - список вершин, образующих цикл отрицательного веса (если есть)

Основные методы

1. ExecuteFloydWarshall(Graph graph)
   Выполняет алгоритм Флойда-Уоршелла:
   — Проверяет, что граф взвешенный
   — Инициализирует матрицы расстояний и предшественников
   — Основной тройной цикл для нахождения кратчайших путей через промежуточные вершины
   — Обнаруживает циклы отрицательного веса
   — Возвращает объект FloydResult
2. FindNegativeCycle(FloydResult result, List<int> vertices, int startIndex)
   Вспомогательный метод для обнаружения цикла отрицательного веса по результатам алгоритма.
3. ReconstructPath(FloydResult result, List<int> vertices, int fromIndex, int toIndex)
   Восстанавливает кратчайший путь между двумя вершинами по матрице предшественников.
4. DisplayResults(Graph graph, int startVertex)
   Основной метод вывода результатов: показывает кратчайшие пути из заданной вершины, матрицу расстояний, пути между всеми парами, предупреждение о циклах отрицательного веса.
5. DisplayDistanceMatrix(FloydResult result, List<int> vertices)
   Выводит матрицу расстояний в читаемом формате.
6. DisplayAllPairsPaths(FloydResult result, List<int> vertices)
   Выводит кратчайшие пути между всеми парами вершин.
7. HasNegativeWeightCycle(Graph graph)
   Проверяет граф на наличие циклов отрицательного веса.
8. FindShortestPath(Graph graph, int from, int to)
   Находит кратчайший путь между двумя конкретными вершинами.

Возвращает кортеж (путь, расстояние).

Код метода ExecuteFloydWarshall

```
1
2   public static FloydResult ExecuteFloydWarshall(Graph graph)
3   {
4       if (graph == null)
5           throw new ArgumentNullException(nameof(graph));
6
7       if (!graph.IsWeighted)
8           throw new ArgumentException("Алгоритм Флойда-Уоршелла работает только
            ↪  со взвешенными графами");
9
10      // Получаем отсортированный список вершин
11      List<int> vertices = graph.GetVertices();
12      int n = vertices.Count;
13
14      // Создаем матрицы расстояний и предшественников
15      FloydResult result = new FloydResult(n);
16
17      // Инициализация матриц
18      for (int i = 0; i < n; i++)
19      {
20          for (int j = 0; j < n; j++)
21          {
22              if (i == j)
23              {
24                  result.Distances[i, j] = 0;
25                  result.Predecessors[i, j] = i;
26              }
27              else
28              {
29                  result.Distances[i, j] = double.PositiveInfinity;
30                  result.Predecessors[i, j] = -1;
31              }
32          }
33      }
34
35      // Заполняем матрицу начальными расстояниями (прямыми рёбрами)
36      for (int i = 0; i < n; i++)
37      {
38          int vertexFrom = vertices[i];
```

```
39              var adjacentEdges = graph.GetAdjacentVertices(vertexFrom);
40
41              foreach (var edge in adjacentEdges)
42              {
43                  int j = vertices.IndexOf(edge.Target);
44                  result.Distances[i, j] = edge.Weight;
45                  result.Predecessors[i, j] = i;
46              }
47          }
48
49          // Основная часть алгоритма Флойда-Уоршелла
50          for (int k = 0; k < n; k++)
51          {
52              for (int i = 0; i < n; i++)
53              {
54                  for (int j = 0; j < n; j++)
55                  {
56                      // Проверяем, можно ли улучшить путь через вершину k
57                      if (result.Distances[i, k] != double.PositiveInfinity &&
58                          result.Distances[k, j] != double.PositiveInfinity)
59                      {
60                          double newDistance = result.Distances[i, k] + result.Distances[k, j];
61
62                          if (newDistance < result.Distances[i, j])
63                          {
64                              result.Distances[i, j] = newDistance;
65                              result.Predecessors[i, j] = result.Predecessors[k, j];
66                          }
67                      }
68                  }
69              }
70          }
71
72          // Проверка на наличие циклов отрицательного веса
73          for (int i = 0; i < n; i++)
74          {
75              if (result.Distances[i, i] < 0)
76              {
77                  result.HasNegativeCycle = true;
78                  result.NegativeCycle = FindNegativeCycle(result, vertices, i);
79                  break;
```

```
80              }
81          }
82
83          return result;
84      }
```

6.3 Примеры

```
=== АЛГОРИТМ ДЕЙКСТРЫ ДЛЯ ВСЕХ ПАР ВЕРШИН ===
Всего вершин: 7

Из вершины 23:
  -> 27: 7,00
  -> 36: 13,00
  -> 57: 14,00
  -> 70: 11,00
  -> 79: 10,00
  -> 80: 12,00

Из вершины 27:
  -> 23: 7,00
  -> 36: 6,00
  -> 57: 7,00
  -> 70: 8,00
  -> 79: 9,00
  -> 80: 5,00

Из вершины 36:
  -> 23: 13,00
  -> 27: 6,00
  -> 57: 3,00
  -> 70: 4,00
  -> 79: 5,00
  -> 80: 1,00

Из вершины 57:
  -> 23: 14,00
  -> 27: 7,00
  -> 36: 3,00
  -> 70: 5,00
  -> 79: 6,00
  -> 80: 2,00

Из вершины 70:
  -> 23: 11,00
  -> 27: 8,00
  -> 36: 4,00
  -> 57: 5,00
  -> 79: 1,00
  -> 80: 3,00

Из вершины 79:
  -> 23: 10,00
  -> 27: 9,00
  -> 36: 5,00
  -> 57: 6,00
  -> 70: 1,00
  -> 80: 4,00

Из вершины 80:
  -> 23: 12,00
  -> 27: 5,00
  -> 36: 1,00
  -> 57: 2,00
  -> 70: 3,00
  -> 79: 4,00

=== МАТРИЦА РАССТОЯНИЙ ===
        23     27     36     57     70     79     80
 23:     -   7,00  13,00  14,00  11,00  10,00  12,00
 27:  7,00      -   6,00   7,00   8,00   9,00   5,00
 36: 13,00   6,00      -   3,00   4,00   5,00   1,00
 57: 14,00   7,00   3,00      -   5,00   6,00   2,00
 70: 11,00   8,00   4,00   5,00      -   1,00   3,00
 79: 10,00   9,00   5,00   6,00   1,00      -   4,00
 80: 12,00   5,00   1,00   2,00   3,00   4,00      -

=== АНАЛИЗ СВЯЗНОСТИ ===
Граф является связным.

Алгоритм Дейкстры успешно выполнен!
```

Рисунок 9 – Задание 1

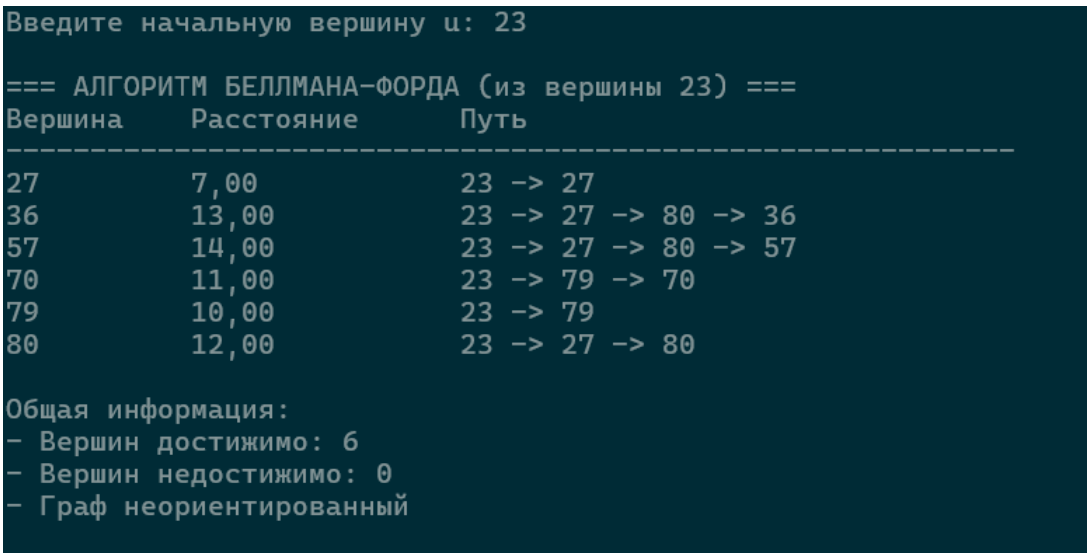

Рисунок 10 – Задание 2

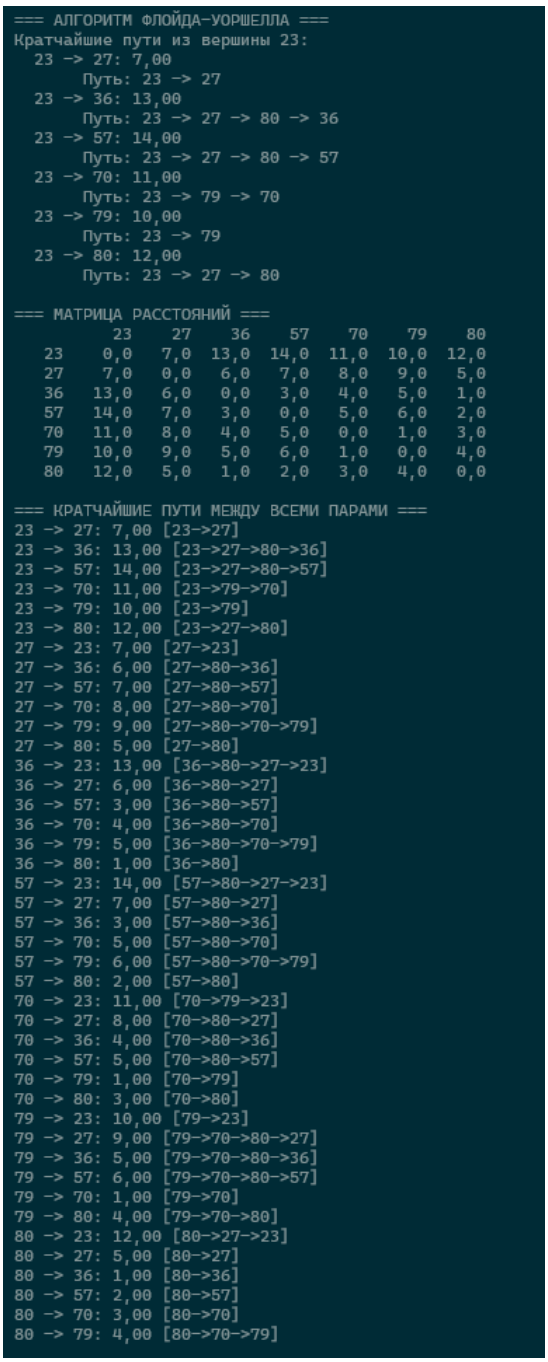

Рисунок 11 – Задание 3

7 Потоки

7.1 Задание

Решить задачу на нахождение максимального потока любым алгоритмом. Подготовить примеры, демонстрирующие работу алгоритма в разных случаях.

7.2 Реализация

Класс FordFulkersonAlgorithm

Класс для реализации алгоритма Форда-Фалкерсона нахождения

максимального потока в транспортной сети. Работает с ориентированными взвешенными графами, где веса рёбер представляют пропускные способности.

Внутренний класс FlowEdge Представляет ребро в остаточной сети:

- To - целевая вершина
- Capacity - остаточная пропускная способность
- Reverse - ссылка на обратное ребро

Основные методы

1. BFS(Dictionary<int, List<FlowEdge» residualGraph, int source, int sink, Dictionary<int, int> parent, Dictionary<int, FlowEdge> parentEdge)
   Поиск в ширину для нахождения увеличивающего пути в остаточной сети.
2. FindMaxFlow(Graph graph, int source, int sink, out Dictionary<Tuple<int, int>, double> flow)
   Основной метод алгоритма Форда-Фалкерсона:
   - Проверяет корректность входных данных
   - Строит остаточную сеть с прямыми и обратными рёбрами
   - Итеративно находит увеличивающие пути с помощью BFS
   - Обновляет потоки и остаточные пропускные способности
   - Возвращает значение максимального потока и распределение потоков по рёбрам
3. Demonstrate(Graph graph)
   Демонстрационный метод: автоматически выбирает исходную и стоковую вершины, вычисляет и выводит максимальный поток.
4. GetSourceVertex(Graph graph)
   Вспомогательный метод: находит вершину с минимальной входящей степенью (кандидат на источник).
5. GetSinkVertex(Graph graph, int source)
   Вспомогательный метод: находит вершину с минимальной исходящей степенью, отличную от источника (кандидат на сток).
6. CalculateInDegree(Graph graph, int vertex)
   Вспомогательный метод: вычисляет входящую степень вершины в ориентированном графе.
7. FindMaxFlowInteractive(Graph graph)
   Интерактивный метод: позволяет пользователю выбрать исходную и стоковую вершины, затем вычисляет и выводит максимальный поток с дета-

лизацией по рёбрам.

Код метода FindMaxFlow

```
public static double FindMaxFlow(Graph graph, int source, int sink,
                                 out Dictionary<Tuple<int, int>, double> flow)
    {
        // Проверка входных данных
        if (graph == null)
            throw new ArgumentException("Граф не задан");

        if (!graph.ContainsVertex(source))
            throw new ArgumentException($"Исходная вершина {source} не существует");

        if (!graph.ContainsVertex(sink))
            throw new ArgumentException($"Стоковая вершина {sink} не существует");

        if (source == sink)
            throw new ArgumentException("Исходная и стоковая вершины не могут
            ↪ совпадать");

        if (!graph.IsDirected)
            throw new ArgumentException("Алгоритм Форда-Фалкерсона работает только
            ↪ с ориентированными графами");

        if (!graph.IsWeighted)
            throw new ArgumentException("Алгоритм Форда-Фалкерсона работает только
            ↪ со взвешенными графами");

        // Инициализация остаточной сети
        Dictionary<int, List<FlowEdge>> residualGraph = new Dictionary<int,
        ↪ List<FlowEdge>>();

        // Копируем вершины
        foreach (var vertex in graph.GetVertices())
        {
            residualGraph[vertex] = new List<FlowEdge>();
        }

        // Копируем рёбра как рёбра с пропускной способностью
        foreach (var vertex in graph.GetVertices())
        {
            foreach (var edge in graph.GetAdjacentVertices(vertex))
```

```
36              {
37                  // Создаём прямое ребро с заданной пропускной способностью
38                  FlowEdge forwardEdge = new FlowEdge(edge.Target, edge.Weight);
39                  // Создаём обратное ребро с нулевой пропускной способностью
40                  FlowEdge reverseEdge = new FlowEdge(vertex, 0);
41
42                  // Связываем рёбра
43                  forwardEdge.Reverse = reverseEdge;
44                  reverseEdge.Reverse = forwardEdge;
45
46                  // Добавляем рёбра в остаточную сеть
47                  residualGraph[vertex].Add(forwardEdge);
48                  residualGraph[edge.Target].Add(reverseEdge);
49              }
50          }
51
52          flow = new Dictionary<Tuple<int, int>, double>();
53          double maxFlow = 0;
54
55          // Инициализируем поток нулями
56          foreach (var vertex in graph.GetVertices())
57          {
58              foreach (var edge in graph.GetAdjacentVertices(vertex))
59              {
60                  flow[Tuple.Create(vertex, edge.Target)] = 0;
61              }
62          }
63
64          // Основной цикл алгоритма
65          Dictionary<int, int> parent = new Dictionary<int, int>();
66          Dictionary<int, FlowEdge> parentEdge = new Dictionary<int, FlowEdge>();
67
68          while (BFS(residualGraph, source, sink, parent, parentEdge))
69          {
70              // Находим минимальную пропускную способность на найденном пути
71              double pathFlow = double.MaxValue;
72
73              for (int v = sink; v != source; v = parent[v])
74              {
75                  FlowEdge edge = parentEdge[v];
76                  pathFlow = Math.Min(pathFlow, edge.Capacity);
```

```
77              }
78
79              // Обновляем остаточные пропускные способности
80              for (int v = sink; v != source; v = parent[v])
81              {
82                  FlowEdge edge = parentEdge[v];
83                  FlowEdge reverseEdge = edge.Reverse;
84
85                  // Уменьшаем пропускную способность прямого ребра
86                  edge.Capacity -= pathFlow;
87                  // Увеличиваем пропускную способность обратного ребра
88                  reverseEdge.Capacity += pathFlow;
89
90                  // Обновляем поток
91                  int u = parent[v];
92                  Tuple<int, int> edgeKey = Tuple.Create(u, v);
93
94                  // Проверяем, существует ли ребро в исходном графе
95                  bool isOriginalEdge = graph.ContainsEdge(u, v);
96
97                  if (isOriginalEdge)
98                  {
99                      // Обновляем поток по прямому ребру
100                     if (flow.ContainsKey(edgeKey))
101                     {
102                         flow[edgeKey] += pathFlow;
103                     }
104                     else
105                     {
106                         flow[edgeKey] = pathFlow;
107                     }
108
109                     // Обновляем поток по обратному ребру (если оно существует в
                        ↪  исходном графе)
110                     Tuple<int, int> reverseEdgeKey = Tuple.Create(v, u);
111                     if (flow.ContainsKey(reverseEdgeKey))
112                     {
113                         flow[reverseEdgeKey] -= pathFlow;
114                     }
115                 }
116             }
```

```
117
118             // Увеличиваем общий поток
119             maxFlow += pathFlow;
120
121             // Очищаем для следующей итерации
122             parent.Clear();
123             parentEdge.Clear();
124         }
125
126         return maxFlow;
127     }
```

7.3 Примеры

```
=== АЛГОРИТМ ФОРДА-ФАЛКЕРСОНА ===
Выберите действие:
1. Найти максимальный поток в текущем графе. Автомтический(без выбора)
2. Найти максимальный поток в текущем графе. С выбором
2
=== НАХОЖДЕНИЕ МАКСИМАЛЬНОГО ПОТОКА ===
Доступные вершины: 23, 35, 51, 57, 73, 78, 89
Введите исходную вершину (source): 57
Введите стоковую вершину (sink): 89

Максимальный поток из 57 в 89: 1,00

Потоки по рёбрам (положительные значения):
  35 -> 78: 1,00
  57 -> 35: 1,00
  78 -> 89: 1,00

Рёбра с нулевым потоком:
  23 -> 57
  73 -> 51
  89 -> 73
```

Рисунок 12 – Задание 3