ПРИЛОЖЕНИЕ А
Код класса графов

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel.Design;
using System.Globalization;
using System.IO;
using System.Security;


namespace graph_1_1
{
    class Graph
    {
        private GraphData graph;
        class GraphData
        {
            public bool[,] adjacencyMatrix;
            public int[,] weightMatrix;
            public bool[] isChecked;
            public int[,] pathVertices;
            public GraphData(bool[,] adjacencyMatrix, int[,]
             ↪  weightMatrix)
            {
                adjacencyMatrix = new
                 ↪  bool[adjacencyMatrix.GetLength(0),
                 ↪  adjacencyMatrix.GetLength(0)];
                isChecked = new bool[VerticesCount];
                pathVertices = new int[VerticesCount,
                 ↪  VerticesCount];

                this.weightMatrix = new int[VerticesCount,
                 ↪  VerticesCount];
```

```
    for (int i = 0; i < VerticesCount; i++)
    {
        for (int j = 0; j < VerticesCount; j++)
        {
            this.adjacencyMatrix[i, j] =
             ↪  adjacencyMatrix[i, j];
            pathVertices[i, j] = -1;
            if (!adjacencyMatrix[i, j])
            {
                this.weightMatrix[i, j] = i == j ? 0 :
                 ↪  int.MaxValue;
            }
            else
            {
                this.weightMatrix[i, j] =
                 ↪  weightMatrix[i, j];
            }
        }
    }
}
public int VerticesCount
{
    get
    {
        return adjacencyMatrix.GetLength(0);
    }
}
public void ResetChecking()
{
    for (int i = 0; i < VerticesCount; i++)
    {
        isChecked[i] = false;
    }
}
```

```
//DFS
public void DepthSearch(in int currentIndex)
{
    isChecked[currentIndex] = true;
    Console.WriteLine(currentIndex);
    for (int i = 0; i < VerticesCount; i++)
    {
        if (isChecked[i] == false &&
         ↪  adjacencyMatrix[i, currentIndex])
        {
            DepthSearch(i);
        }
    }
}
//BFS
public void BreadthSearch(in int currentIndex,
 ↪  Queue<int> queue)
{
    queue.Enqueue(currentIndex);
    while (queue.Count != 0)
    {
        isChecked[currentIndex] = true;
        for (int i = 0; i < VerticesCount; i++)
        {
            if (isChecked[i] == false &&
             ↪  adjacencyMatrix[i, currentIndex])
            {
                queue.Enqueue(currentIndex);
            }
        }
        Console.WriteLine(queue.Dequeue());
    }
}
//Dijkstra
```

```
public long[] DijkstraSearch(int startSource)
{
    long[] minimalLength = new long[VerticesCount];
     ↪  //d
    for (int i = 0; i < VerticesCount; i++)
    {
        minimalLength[i] = int.MaxValue;
    }
    minimalLength[startSource] = 0;
    long newDistance;
    for (int i = 0; i < VerticesCount; i++)
    {
        int currentClosest = -1;
        for (int j = 0; j < VerticesCount; j++)
        {
            if (isChecked[j] == false &&
             ↪  (currentClosest == -1 ||
             ↪  minimalLength[j] <
             ↪  minimalLength[currentClosest]))
            {
                currentClosest = j;
            }
        }
        if (currentClosest == -1)
        {
            break;
        }
        isChecked[currentClosest] = true;
        for (int j = 0; j < VerticesCount; j++)
        {
            if (adjacencyMatrix[currentClosest, j])
            {
```

```
                newDistance =
                 ↪  minimalLength[currentClosest] +
                 ↪  weightMatrix[currentClosest, j];
                if (newDistance < minimalLength[j])
                {
                    minimalLength[j] = newDistance;
                    pathVertices[startSource, j] =
                     ↪  currentClosest;
                }
            }
        }
    }
    return minimalLength;
}
//Floyd
public long[,] FloydSearch()
{
    long[,] minimalDistance =
     ↪  (long[,])weightMatrix.Clone();
    long newDistance;
    for (int k = 0; k < VerticesCount; k++)
    {
        for (int i = 0; i < VerticesCount; i++)
        {
            for (int j = 0; j < VerticesCount; j++)
            {
                if (minimalDistance[i, k] !=
                 ↪  int.MaxValue && minimalDistance[k,
                 ↪  j] != int.MaxValue)
                {
                    newDistance = minimalDistance[i,
                     ↪  k] + minimalDistance[k, j];
                    if (newDistance <
                     ↪  minimalDistance[i, j])
```

```
                    {
                        minimalDistance[i, j] =
                         ↪  newDistance;
                        pathVertices[i, j] =
                         ↪  pathVertices[k, j];
                    }
                }
            }
        }
    }
    return minimalDistance;
}
//WayBack
public void WayBack(int startIndex, int targetIndex,
 ↪  ref Stack<int> path)
{
    if (targetIndex == startIndex)
    {
        path.Push(targetIndex);
        return;
    }
    path.Push(targetIndex);
    WayBack(startIndex, pathVertices[startIndex,
     ↪  targetIndex], ref path);
}
// Function that returns reverse (or transpose) of
 ↪  this graph
private GraphData Transpose(GraphData graphData)
{
    GraphData temp = new
     ↪  GraphData(graphData.adjacencyMatrix,
     ↪  graphData.weightMatrix);
    for (int i = 0; i < temp.VerticesCount; i++)
    {
```

```
        for (int j = 0; j < temp.VerticesCount; j++)
        {
            if (temp.adjacencyMatrix[i, j] == true)
            {
                temp.adjacencyMatrix[j, i] =
                 ↪  temp.adjacencyMatrix[i, j];
                temp.weightMatrix[j, i] =
                 ↪  temp.weightMatrix[i, j];
                temp.adjacencyMatrix[i, j] = false;
                temp.weightMatrix[i, j] = i == j ? 0 :
                 ↪  int.MaxValue;
            }
        }
    }
    return temp;
}
private void DFS1(int v, bool[] visited, Stack<int>
 ↪  stack)
{
    visited[v] = true;
    for (int i = 0; i < VerticesCount; i++)
    {
        if (!visited[i] && adjacencyMatrix[v, i])
            DFS1(i, visited, stack);
    }
    stack.Push(v);
}
private void DFS2(int v, bool[] visited, List<int>
 ↪  components)
{
    visited[v] = true;
    components.Add(v);
    for (int i = 0; i < VerticesCount; i++)
    {
```

```
            if (!visited[i] && adjacencyMatrix[v, i])
            {
                DFS2(i, visited, components);
            }
        }
    }
    //KosarajuSearch
    public List<List<int>> KosarajuSearch()
    {
        Stack<int> stack = new Stack<int>();
        bool[] visited = new bool[VerticesCount];
        for (int i = 0; i < VerticesCount; i++)
            visited[i] = false;
        for (int i = 0; i < VerticesCount; i++)
        {
            if (!visited[i])
                DFS1(i, visited, stack);
        }
        GraphData transposedGraph = Transpose(this);
        ResetChecking();
        List<List<int>> components = new
         ↪  List<List<int>>();
        while (stack.Count != 0)
        {
            int v = stack.Pop();
            int j = 0;
            if (!visited[v])
            {
                List<int> component = new List<int>();
                transposedGraph.DFS2(v, visited,
                 ↪  components[j]);
                Console.WriteLine(string.Join(", ",
                 ↪  component));
            }
```

```
    }
    return components;
}
//BoruvkiSearch
public GraphData BoruvkiSearch()
{
    List<(int, int)> edgesH = new List<(int, int)>();
    List<(int, (int, int))> edgesG = new List<(int,
     ↪  (int, int))>();
    int[] edgeID = new int[VerticesCount];
    for (int i = 0; i < VerticesCount; i++)
    {
        for (int j = i; j < VerticesCount; j++)
        {
            if (adjacencyMatrix[i, j])
            {
                edgesG.Add((weightMatrix[i, j], (i,
                 ↪  j)));
            }
        }
    }
    edgesG.Sort();
    GraphData result = new GraphData(adjacencyMatrix,
     ↪  weightMatrix);
    for (int i = 0; i < VerticesCount; i++)
    {
        for (int j = 0; j < VerticesCount; j++)
        {
            result.adjacencyMatrix[i, j] = false;
            weightMatrix[i, j] = i == j ? 0 :
             ↪  int.MaxValue;
        }
    }
    foreach (var edge in edgesG)
```

```
    {
        if (edgeID[edge.Item2.Item1] !=
         ↪  edgeID[edge.Item2.Item2])
        {
            result.adjacencyMatrix[edge.Item2.Item1,
             ↪  edge.Item2.Item1] = true;
            result.weightMatrix[edge.Item2.Item1,
             ↪  edge.Item2.Item1] = edge.Item1;
            int newID = edgeID[edge.Item2.Item1];
            int oldID = edgeID[edge.Item2.Item2];
            for (int i = 0; i < VerticesCount; i++)
            {
                if (edgeID[i] == oldID)
                    edgeID[i] = newID;
            }
        }
    }
    return result;
}
//EulerSearch
public void SearchEuler(int currentVertice, ref
 ↪  bool[,] adjacencyMatrix, ref Stack<int>
 ↪  eulerVertices)
{
    for (int i = 0; i < adjacencyMatrix.GetLength(0);
     ↪  i++)
    {
        if (adjacencyMatrix[currentVertice, i] ==
         ↪  true)
        {
            adjacencyMatrix[currentVertice, i] =
             ↪  false;
            adjacencyMatrix[i, currentVertice] =
             ↪  false;
```

```
                SearchEuler(i, ref adjacencyMatrix, ref
                 ↪  eulerVertices);
            }
        }
        eulerVertices.Push(currentVertice);
    }
}
public Graph(in bool[,] adjacencyMatrix)
{
    int[,] weightMatrix = new int[adjacencyMatrix.Length,
     ↪  adjacencyMatrix.Length];
    for (int i = 0; i < weightMatrix.Length; i++)
    {
        for(int j = 0; j < weightMatrix.Length; j++)
        {
            if (adjacencyMatrix[i, j])
            {
                weightMatrix[i, j] = 1;
            }
            else
            {
                weightMatrix[i, j] = int.MaxValue;
            }
        }
    }
    graph = new GraphData(adjacencyMatrix, weightMatrix);
}
public Graph(bool[,] adjacencyMatrix, int[,] weightMatrix)
{
    graph = new GraphData(adjacencyMatrix, weightMatrix);
}
public int Size
{
    get { return graph.VerticesCount; }
```

```
}
public void DepthSearch(in int startIndex)
{
    graph.DepthSearch(startIndex);
    graph.ResetChecking();
}
public void BreadthSearch(in int startIndex)
{
    Queue<int> queue = new Queue<int>();
    graph.BreadthSearch(startIndex, queue);
    graph.ResetChecking();
}
public long[] DijkstraSearch(in int startIndex)
{
    long[] result = graph.DijkstraSearch(startIndex);
    graph.ResetChecking();
    return result;
}
public long[,] FloydSearch()
{
    long[,] result = graph.FloydSearch();
    graph.ResetChecking();
    return result;
}
public List<List<int>> KosarajuSearch()
{
    return graph.KosarajuSearch();
}
public Graph BoruvkiSearch()
{
    GraphData temp = graph.BoruvkiSearch();
    return new Graph(temp.adjacencyMatrix,
     ↪  temp.weightMatrix);
}
```

```
        public Stack<int> EulerSearch()
        {
            bool[,] a = new bool[graph.VerticesCount,
             ↪  graph.VerticesCount];
            for (int i = 0; i < graph.VerticesCount; i++)
            {
                for (int j = 0; j < graph.VerticesCount; j++)
                {
                    a[i, j] = graph.adjacencyMatrix[i, j];
                }
            }
            Stack<int> path = new Stack<int>();
            graph.SearchEuler(0, ref a, ref path);
            return path;
        }
        public Stack<int> WayBack(int startIndex, int targetIndex)
        {
            Stack<int> path = new Stack<int>();
            if (startIndex == targetIndex)
            {
                path.Push(startIndex);
                return path;
            }
            if (graph.pathVertices[startIndex, targetIndex] == -1)
            {
                DijkstraSearch(startIndex);
            }
            graph.WayBack(startIndex, targetIndex, ref path);
            return path;
        }
    }
}
```