Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ

Н. Г. ЧЕРНЫШЕВСКОГО»

кафедра: информатики и программирования

**ОТЧЕТ О ПРАКТИКЕ ПО ТЕОРИИ ГРАФОВ**

Выполнила:

студентка 3 курса 341 группы

направления «Математическое обеспечение

и администрирование информационных систем»

факультета Компьютерных наук и Информационных технологий

Батталова Нур Алия

Проверила:

Портенко М.С.

Саратов 2017

Оглавление

[СОЗДАНИЕ КЛАССА ГРАФ 3](#_Toc501564498)

[Минимальные требования для класса Граф 3](#_Toc501564499)

[РАБОТА С ГРАФОМ: СПИСОК СМЕЖНОСТИ 8](#_Toc501564500)

[Список смежности I А 8](#_Toc501564501)

[Список смежности I Б 9](#_Toc501564502)

[ОБХОДЫ ГРАФА 11](#_Toc501564503)

[Обходы графа II 11](#_Toc501564504)

[ПОСТРОЕНИЕ МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА 13](#_Toc501564505)

[Каркас III (Краскал) 13](#_Toc501564506)

[ВЗВЕШЕННЫЙ ГРАФ 17](#_Toc501564507)

[Веса IV А Дейкстра 17](#_Toc501564508)

[Веса IV В Форд-Беллман 21](#_Toc501564509)

[Веса IV С Флойд 22](#_Toc501564510)

[ПОТОКИ 25](#_Toc501564511)

[V Нахождение максимального потока 25](#_Toc501564512)

# СОЗДАНИЕ КЛАССА ГРАФ

## Минимальные требования для класса Граф

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, преобразующий список смежности в список ребер);

2. Конструкторы (не менее 4-х):

* конструктор по умолчанию - создающий связный граф по некоторым одним вам известным принципам - для тестирования
* конструктор, заполняющий данные графа из файла
* конструктор-копию (аккуратно, не все сразу делают именно копию)
* конструктор специфический для каждого отдельного разработчика, зависит от набора задач (сильно связный граф, цепь, кольцо, дерево)

Конструкторов может быть больше.

3. Методы:

* добавляющие вершину,
* добавляющие ребро,
* удаляющие вершину,
* удаляющие ребро.

Программы код:

Класс граф

class Graph

{

private Dictionary<int, SortedList<int, int>> graph = new Dictionary<int,

SortedList<int, int>>();

private int type; //Невзвешанный – 1, взвешанный - 0

public Graph(int type, int v) //Конструктор для создания сильно связанного графа

{

this.type = type;

if (type == 1) //Невзвешанный граф

{

for (int i = 0; i < v; i++)

{

SortedList<int, int> list = new SortedList<int, int>();

for (int j = 0; j < v; j++)

{

if (i != j)

{

list.Add(j, 1);

}

}

graph.Add(i, list);

}

}

else

{

Random rnd = new Random();

for (int i = 0; i < v; i++)

{

SortedList<int, int> list = new SortedList<int, int>();

for (int j = 0; j < v; j++)

{

if (i != j)

{

list.Add(j, rnd.Next(0, 100)); //Веса ребер от 0 до 100

}

}

graph.Add(i, list);

}

}

}

public Graph() //Конструктор, считывающий граф из файла

{

using (StreamReader fileIn = new StreamReader("Input.txt"))

{

int type = int.Parse(fileIn.ReadLine());

this.type = type;

string line = fileIn.ReadToEnd();

string[] lines = line.Split('\n');

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

{

if (lines[i] != "-1")

{

SortedList<int, int> list = new SortedList<int, int>();

string[] line2 = lines[i].Split(' ');

for (int j = 0; j < line2.Length - 1; j += 2)

{

list.Add(Convert.ToInt32(line2[j]),

Convert.ToInt32(line2[j +1]));

}

graph.Add(i, list);

}

else

{

graph.Add(i, null);

}

}

}

}

public Graph(Graph a) //Конструктор, создающий копию существующего графа

{

for (int i = 0; i < a.Size; i++)

{

SortedList<int, int> list1 = new SortedList<int, int>(a.graph[i]);

graph.Add(i, list1);

}

}

public int Size //свойство для получения размерности списка

{

get

{

return graph.Count();

}

}

public void AddNode() //Добавление вершины

{

SortedList<int, int> list = new SortedList<int, int>();

if (type == 0) //взвешанный граф

{

Random rnd = new Random();

int lastNode = graph.Keys.Last() + 1;

foreach (var node in graph.Keys)

{

int weigth = rnd.Next(0, 100);

graph[node].Add(lastNode, weigth);

list.Add(node, weigth);

}

graph.Add(lastNode, list);

}

else

{

int lastNode = graph.Keys.Last() + 1;

foreach (var node in graph.Keys)

{

graph[node].Add(lastNode, 1);

list.Add(node, 1);

}

graph.Add(lastNode, list);

}

}

public void AddEdge(int u, int v) //Добавление ребра

{

SortedList<int, int> list = new SortedList<int, int>();

if (type == 0) //взвешанный граф

{

Random rnd = new Random();

graph[u].Add(v, rnd.Next(0, 100));

graph[v].Add(u, rnd.Next(0, 100));

}

else

{

graph[u].Add(v, 1);

graph[v].Add(u, 1);

}

}

public void RemoveNode(int element) //Удаление вершины

{

foreach (KeyValuePair<int, int> item in graph[element])

{

graph[item.Key].Remove(element);

}

graph.Remove(element);

}

public void RemoveEdge(int u, int v) //Удаление ребра

{

graph[u].Remove(v);

graph[v].Remove(u);

}

//Возращает список смежных вершин с данной вершиной

public void TryGetValue(out SortedList<int, int> list1, int u)

{

graph.TryGetValue(u, out list1);

}

public int Type { get { return type; } } //Возращается тип графа

public int ValuesCount

{

get

{

int countValues = 0;

for (int i = 0; i < Size; i++)

{

countValues += graph[i].Values.Count;

}

return countValues;

}

}

public Dictionary<int, SortedList<int, int>>.KeyCollection Keys

{

get

{

return graph.Keys;

}

}

public void Show()

{

foreach (KeyValuePair<int, SortedList<int, int>> keyValue in graph)

{

Console.Write(keyValue.Key + " - ");

foreach (KeyValuePair<int, int> element in keyValue.Value)

{

Console.Write(element);

}

Console.WriteLine();

}

}

}

Класс Program

class Program

{

static void Main(string[] args)

{

//Создание графов

Graph graph\_standart1 = new Graph(1, 5); //Сильно связанный граф

Console.WriteLine("---Graph 1----");

graph\_standart1.Show();

Graph graph\_standart2 = new Graph(0, 5); //Сильно связанный взвешанный граф

Console.WriteLine("---Graph 2----");

graph\_standart2.Show();

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph 3----");

graph\_standart3.Show();

Graph graph\_standart4 = new Graph(graph\_standart3); //Скопированный граф

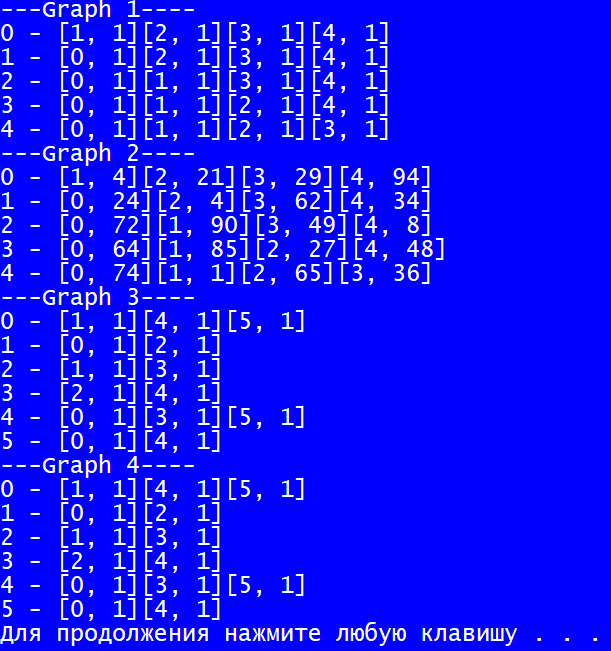
Console.WriteLine("---Graph 4----");

graph\_standart4.Show();

}

}

Пример выходных данных:



Проверим корретность выполнения операций над графом (неориентированным, для ориентированно операции представлены в главе «Кратчайшие пути в графе»).

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph 3----");

graph\_standart3.Show();

Console.WriteLine("---Remove Node----");

graph\_standart3.RemoveNode(3);

graph\_standart3.Show();

Console.WriteLine("---Remove Edge----");

graph\_standart3.RemoveEdge(1, 2);

graph\_standart3.Show();

//Проверка копирования

Graph graph\_standart4 = new Graph(graph\_standart3); //Скопированный граф

Console.WriteLine("---Graph 4----");

graph\_standart4.Show();

Console.WriteLine("---Add Edge----");

graph\_standart3.AddEdge(1, 2);

graph\_standart3.Show();

Console.WriteLine("---Add Node----");

graph\_standart3.AddNode();

graph\_standart3.Show();

Console.WriteLine("Parent");

graph\_standart4.Show();

}

}

Выходные данные:

|  |  |
| --- | --- |
|  |  |

# РАБОТА С ГРАФОМ: СПИСОК СМЕЖНОСТИ

## Список смежности I А

Задача 16. Определить, можно ли попасть из вершины ***u*** в вершину ***v*** через одну какую-либо вершину орграфа.

Класс граф используется тот же, который создавался в главе «Создание класса граф».

Программный код:

Класс Program

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph 3----");

graph\_standart3.Show();

//Задаем вершины

int u = 1; //Первая вершина

int v = 3; //Вторая вершина

if (Task1\_16(graph\_standart3, u, v))

{

Console.WriteLine("Возможно попасть из вершины " + u + " в вершину " +

v + " через одну вершину");

}

else

{

Console.WriteLine("Невозможно попасть из вершины " + u + " в вершину "

+ v + " через одну вершину");

}

}

//Метод возвращает булевского значение, если существует вершина через которую из первой

//можно попасть во вторую, то возращается true, иначе false.

public static bool Task1\_16(Graph graphOur, int u, int v)

{

SortedList<int, int> list1 = new SortedList<int, int>();

graphOur.TryGetValue(out list1, u); //Получаем все смежные вершины с первой верш

foreach (KeyValuePair<int, int> i in list1)

{

//У каждой такой вершины, проверяем есть ли в ее списке смежных вершин вторая вершина

SortedList<int, int> list2 = new SortedList<int, int>();

graphOur.TryGetValue(out list2, i.Key);

// Если в списке будет вторая вершина, то ее индекс в списке ключей будет >=0

if (list2.IndexOfKey(v) >= 0)

{

return true;

}

}

return false;

}

}

Входные данные:

Дан взвешанный ориентированный граф (type = 0), заданный списком смежности.

0

1 1 2 4

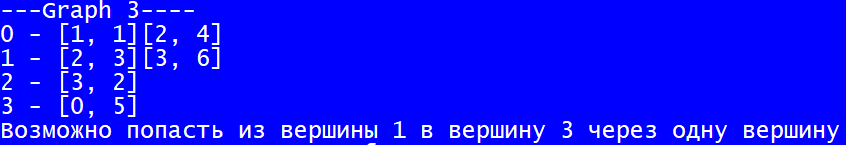
2 3 3 6

3 2

0 5

Тесты:

1. Возможно ли попать из 1 в 3 через одну вершину?



1. Возможно ли попать из 2 в 1 через одну вершину?



## Список смежности I Б

Задача 1. Вывести список смежности графа, являющегося дополнением данного графа.

Класс граф используется тот же, который создавался в главе «Создание класса граф».

Программы код:

Класс Program

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph ----");

graph\_standart3.Show();

//Задание №1

Graph additionGraph = Task2\_1(graph\_standart3);

Console.WriteLine("New Graph Addirion");

additionGraph.Show();

}

//В представленном ниже методе, для получания дополнение графа, мы создаем сильно //связанный граф с тем же количеством вершин, далее просто удаляем из него, все ребра, //которые присутствуют в исходном графе.

public static Graph Task2\_1(Graph graphOur)

{

int count = graphOur.Size;

Graph additionGraph = new Graph(graphOur.Type, count);

foreach (var node in graphOur.Keys)

{

foreach (KeyValuePair<int, int> item in graphOur.Values(node))

{

additionGraph.Values(node).Remove(item.Key);

}

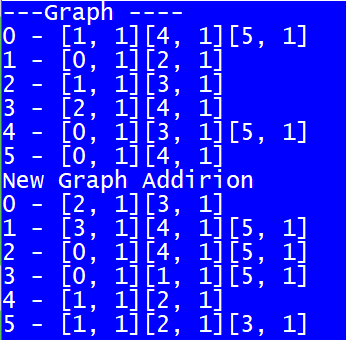
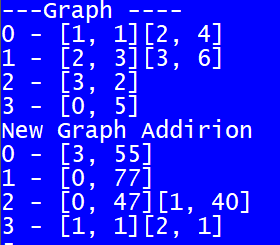
}

return additionGraph;

}

}

Примеры: Первые граф исходные, второй дополнение исходного графа

# ОБХОДЫ ГРАФА

## Обходы графа II

Задача 21. Проверить, можно ли из графа удалить какую-либо вершину так, чтобы получилось дерево.

Для решения поставленной задачи воспользуемся одним их критериев, когда граф является деревом: «Граф – дерево, когда граф связанный и количество вершин на 1 больше количества ребер»

Программный код:

Добавим графу еще одину характеристику type2, который означает ориентированный граф или нет (0 – неориентированный, 1 - ориентированный). В наш класс граф, нужно добавить обход в глубину, для проверки связанности графа и свойство для подсчета ребер в графе. Для обхода в глубину создадим стек «myStack» и одномерный массив visit, для хранения посещенных вершин.

public int ValuesCount

{

get

{

int countValues = 0;

foreach (var node in graph.Keys)

{

countValues += graph[node].Values.Count;

}

if (type2 == 0) //Если граф неориентированный, то количество ребер делим на 2,

//так как каждое ребро учитывалось при подсчете дважды.

return countValues /= 2;

else

return countValues;

}

}

public void Dfs(int v) //Обход в глубину

{

visit[v] = true;

myStack.Push(v);

int u, w;

while (myStack.Count != 0)

{

u = (int)myStack.Peek();

w = AdjD(u);

if (w >= 0)

{

visit[w] = true;

myStack.Push(w);

}

else

{

myStack.Pop();

}

}

}

public int AdjD(int node) //метод поиска смежных не просмотренные вершин

{

foreach (KeyValuePair<int, int> item in graph[node])

{

if (visit[item.Key] == false)

return item.Key;

}

return -1;

}

public void IsTreeOrNot()

{

Graph graph2 = new Graph(); //Создается новый граф, так как для подсчета

//количаства ребер нужно будет удалять подсчитанные ребра

int counterEdge = ValuesCount;

SortedList<int, int> list = new SortedList<int, int>();

int size = Size; //Количество вершин в графе

int n; //Переменная для хранения количества ребер у текущей вершины

Graph graph3;

foreach (var node in graph.Keys)

{

n = Values(node).Values.Count;

graph3 = new Graph(); //При каждой итерации создаем новый граф, для того чтобы

//проверять связанность графа после удаления текущей вершины из него.

//Но изменяя наш заданный граф

graph3.RemoveNode(node);

//Проверка свойства, является ли граф деревом

if ((((Size - 1) - (counterEdge - n)) == 1) && (graph3.IsTree(node)))

{

Console.WriteLine("При удалении вершины " + node + " граф становится

деревом");

}

}

}

public bool IsTree(int node)

{

if (node > graph.Keys.Last()) //если удаляется последняя по счету вершина в графе,

//то обход в глубину начинается с первой вершины графа

{

node = 0;

}

Else //иначе со следующей

{

node++;

}

Dfs(node);

int n = 0;

for (int i = 0; i < Size; i++)

{

if (visit[i])

n++;

}

if (n == graph.Count - 1)

{

Console.WriteLine("vertex = " + n + " edge = " + graph.Count);

return true;

}

else

return false;

}

Класс Program

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph ----");

graph\_standart3.Show();

//Задание №21

Task21(graph\_standart3);

}

public static void Task21(Graph graphOur)

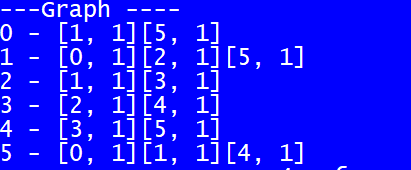
{

graphOur.IsTreeOrNot();

}

}

Входные данные



Выходные данные



# ПОСТРОЕНИЕ МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА

## Каркас III (Краскал)

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

Программный код:

Алгоритм Краскала взаимодействует с отсортированным в неубывающем порядке списком ребер для этого нужно создать класс Edge:

public class Edge

{

public int U;

public int V;

public double Weight;

public Edge()

{

}

public Edge(int U, int V, double Weight)

{

this.U = U;

this.V = V;

this.Weight = Weight;

}

}

Сам алгоритм реализован в отдельном классе.

class KruskalAlgorithm

{

private int edgesCount; //количество ребер в графе

private int nodeCount; //количества вершин в графе

private List<Edge> edges; //список ребер графа

private SortedList<int, int> tree; //результирующее дерево

private int[] sets; //массив для хранения компоненты связанность для каждой вершины

//Свойства

public List<Edge> Edges { get { return edges; } }

public int NodeCount { get { return nodeCount; } }

public double Cost { get; private set; }

public KruskalAlgorithm(Graph graph) //Конструктор. Инициализируем параметры

{

nodeCount = graph.Size;

tree = new SortedList<int, int>();

sets = new int [nodeCount];

edgesCount = graph.ValuesCount;

edges = new List<Edge>(edgesCount);

foreach (var dictionary in graph)

{

foreach (var elem in dictionary.Value)

edges.Add(new Edge

{

U = dictionary.Key,

V = elem.Key,

Weight = elem.Value

});

}

//Сначала все вершины принадлежат разным компонентам связанности

for (int i = 0; i < nodeCount; i++) sets[i] = i;

}

private void ArrangeEdges(int k) //сортировка пузырьком

{

Edge temp;

for (int i = 0; i < k; i++)

{

for (int j = k - 1; j > i ; j--)

{

if (edges[j - 1].Weight >= edges[j].Weight)

{

temp = edges[j - 1];

edges[j - 1] = edges[j];

edges[j] = temp;

}

}

}

for ( int i=0; i<k;i++)

{

Console.WriteLine(edges[i].U + " " + edges[i].V + " " + edges[i].Weight);

}

}

//метод для определения к какой компоненте связанность принадлежит текущая вершина

private int Find(int vertex)

{

return (sets[vertex]);

}

//метод объединения двух вершин в одном компоненту связанность

private void Join(int v1, int v2)

{

if (v1 < v2)

sets[v2] = v1;

else

sets[v1] = v2;

}

//Построение остовного дерева

public void BuildSpanningTree()

{

Console.WriteLine("Количество ребер = " + edgesCount);

int i;

this.ArrangeEdges(edgesCount); //Сортируем ребра

this.Cost = 0;

tree = new SortedList<int, int>();

for (i = 0; i < edgesCount; i++)

{

//Если концы ребра принадлежат разным компонента связанности

if (this.Find(edges[i].U) != this.Find(edges[i].V))

{

//И Если дерево не содержит еще такой вершины, то

if (!tree.ContainsKey(edges[i].U))

{

tree.Add(edges[i].U, edges[i].V); //добавляем ребро в дерево

//Удаляем ребро из списка ребер

edges.Remove(new Edge(edges[i].V, edges[i].U, edges[i].Weight));

this.Join(edges[i].U, edges[i].V); //в одну компоненту связанности

this.Cost += edges[i].Weight;

edgesCount--;

}

}

}

}

//Вывод ребер минимального остовного дерева

public void DisplayInfo()

{

Console.WriteLine("The Edges of the Minimum Spanning Tree are:");

foreach (var couple in tree)

{

Console.WriteLine(couple);

}

}

}

Для доступа к графу из класса KruskalAlgorithm нужен IEnumerator:

class GraphEnum : IEnumerator

{

int position = -1;

private Dictionary<int, SortedList<int, int>> graph1;

public GraphEnum(Dictionary<int, SortedList<int, int>> graph1)

{

this.graph1 = graph1;

}

public bool MoveNext()

{

position++;

return (position < graph1.Count);

}

public void Reset()

{

position = -1;

}

object IEnumerator.Current

{

get

{

return Current;

}

}

public KeyValuePair<int, SortedList<int, int>> Current

{

get

{

try

{

return graph1.ElementAt(position);

}

catch (IndexOutOfRangeException)

{

throw new InvalidOperationException();

}

}

}

}

Так же в классе Graph нужно добавить:

class Graph : IEnumerable

{

...

IEnumerator IEnumerable.GetEnumerator()

{

return (IEnumerator)GetEnumerator();

}

public GraphEnum GetEnumerator()

{

return new GraphEnum(graph);

}

}

Класс Program

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

Console.WriteLine("---Graph ----");

graph\_standart3.Show();

//Задание №4

Task4(graph\_standart3);

}

public static void Task4(Graph graphOur)

{

KruskalAlgorithm k = new KruskalAlgorithm(graphOur);

k.BuildSpanningTree();

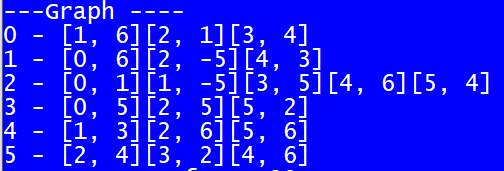
Console.WriteLine("Cost: " + k.Cost);

k.DisplayInfo();

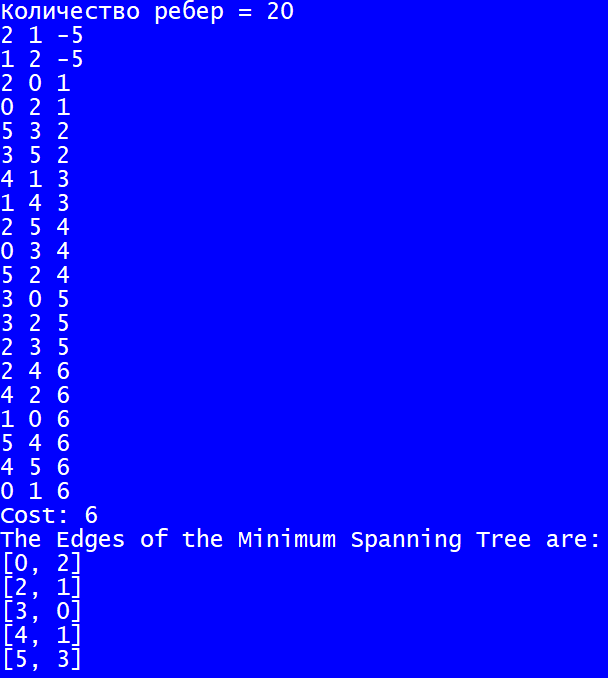
}

}

Входные данные



Выходные данные



# ВЗВЕШЕННЫЙ ГРАФ

## Веса IV А Дейкстра

В графе нет рёбер отрицательного веса.

Задача 5. Вывести длины кратчайших путей от u до всех остальных вершин.

Программный код:

Класс Program

class Program

{

static void Main(string[] args)

{

Graph\_R graph = new Graph\_R();

graph.Show();

TaskDejkstra(graph, 0);

}

static void TaskDejkstra(Graph\_R graph, int node)

{

int i = 0;

foreach (var dictionary in graph.Dejkstra(node))

{

Console.WriteLine("The shortest way from the node: " + node + " to the node: "

+ dictionary.Key + " is = " + dictionary.Value.Keys[0]);

i++;

}

}

}

Класс Graph

Добавляем метод реализующий алгоритм Дейкстры

class Graph\_R

{

...

public Dictionary<int, SortedList<int, bool>> Dejkstra(int node)

{

Dictionary<int, SortedList<int, bool>> dictionary =

new Dictionary<int, SortedList<int, bool>>();

Initialize(node, dictionary);

foreach (var i in graph)

{

long min = int.MaxValue;

int w = 0;

foreach (var j in dictionary)

{

if (j.Value.Keys[0] < min && j.Value.Values[0] == true)

{

min = j.Value.Keys[0];

w = j.Key;

}

}

SortedList<int, bool> newList = new SortedList<int, bool>(1);

newList.Add(dictionary[w].Keys[0], false);

dictionary[w] = newList;

int[] adj = new int[graph[w].Count];

if (Adj(w, adj))

{

int k = 0;

foreach (int v in adj)

{

Relax(w, v, k, dictionary);

k++;

}

}

}

return dictionary;

}

public void Initialize(int node, Dictionary<int, SortedList<int, bool>> d)

{

foreach (var str in graph)

{

SortedList<int, bool> list = new SortedList<int, bool>();

if (str.Key == node)

{

list.Add(0, true);

}

else

{

list.Add(int.MaxValue - 10, true);

}

d.Add(str.Key, list);

}

}

public void Relax(int u, int v, int k, Dictionary<int, SortedList<int, bool>> d)

{

SortedList<int, bool> newl = new SortedList<int, bool>();

if ((d[v].Values[0]) && (d[v].Keys[0] > d[u].Keys[0] + graph[u].Values[k]))

{

newl.Add(d[u].Keys[0] + graph[u].Values[k], d[v].Values[0]);

d[v] = newl;

}

}

public bool Adj(int node, int[] adj)

{

int i = 0;

if (graph[node] != null)

{

foreach (KeyValuePair<int, int> item in graph[node])

{

adj[i] = item.Key;

i++;

}

return true;

}

return false;

}

...

}

Так как алгоритм работает как для ориентированные графов, так и для неориентированных. Следует добавить новые реализации операций добавления ребра, удаление ребра. Операции с вершинами, можно оставить те же.

Как указывалось ранее, у нас есть переменные описывающие тип графа. Одна из них type2 – отвечает , что ориентированно или неориентированность графа. На основе этого добавляем методы.

Метод добавления ребра:

public void AddEdge(int u, int v)

{

SortedList<int, int> list = new SortedList<int, int>();

if (type2 == 0) //неориентированный граф

{

if (type == 0) //взвешанный граф

{

Random rnd = new Random();

graph[u].Add(v, rnd.Next(0, 100));

graph[v].Add(u, rnd.Next(0, 100));

}

else

{

graph[u].Add(v, 1);

graph[v].Add(u, 1);

}

}

else

{

if (type == 0) //взвешанный граф

{

Random rnd = new Random();

graph[u].Add(v, rnd.Next(0, 100));

}

else

{

graph[u].Add(v, 1);

}

}

Метод удаления ребра:

public void RemoveEdge(int u, int v)

{

if (type2 == 0) //неориентированный граф

{

graph[u].Remove(v);

graph[v].Remove(u);

}

else

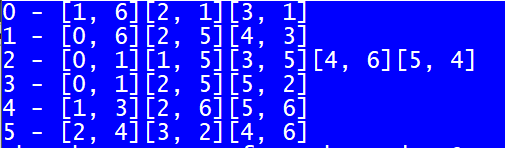
{

graph[u].Remove(v);

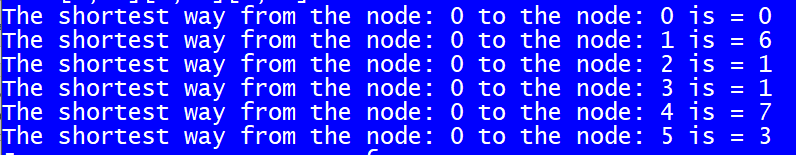
}

}

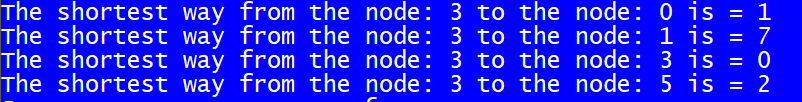
Входные данные



Выходные данные от вершины 0:



Выходные данные от вершины 3, при удалении из исходного графа вершина 2 и 4:



## Веса IV В Форд-Беллман

В графе нет циклов отрицательного веса.

Задача 14. Вывести все кратчайшие пути из вершины u.

Программный код:

Класс Program

class Program

{

static void Main(string[] args)

{

Graph\_R graph = new Graph\_R();

graph.Show();

TaskFordBellman(graph, 0);

}

static void TaskFordBellman(Graph\_R graph, int node)

{

int i = 0;

SortedList<int, int> d = new SortedList<int, int>(graph.Size);

if (graph.FordBellman(node, d))

{

foreach (KeyValuePair<int, int> distance in d)

{

Console.WriteLine("The shortest way from the node: " + node +

" to the node: " + distance.Key + " is = " + distance.Value);

i++;

}

}

else

{

Console.WriteLine("There is a cycle of negative weight");

}

}

}

В класс Graph добавляем реаоизацию алгоритма

public bool FordBellman(int node, SortedList<int, int> d)

{

SortedList<int, bool> nov = new SortedList<int, bool>();

Initialize(node, d, nov);

for (int i = 0; i < Size - 1; i++)

{

foreach (var str in graph)

{

foreach (KeyValuePair<int, int> pair in str.Value)

{

if (nov[pair.Key] && (d[pair.Key] > d[str.Key] + pair.Value))

{

d[pair.Key] = d[str.Key] + pair.Value;

}

}

}

}

foreach (KeyValuePair<int, SortedList<int, int>> p in graph)

{

if (p.Value != null)

{

foreach (KeyValuePair<int, int> pair in graph[p.Key])

{

if ((pair.Key != node) && (d[pair.Key] > d[p.Key] + pair.Value))

{

Console.WriteLine(pair.Key + " " + p.Key + ' ' + pair.Value);

return false;

}

}

}

}

return true;

}

public void Initialize(int node, SortedList<int, int> d, SortedList<int, bool> nov)

{

foreach (var str in graph)

{

if (str.Key != node)

{

d.Add(str.Key, int.MaxValue - 10);

nov[str.Key] = true;

}

else

{

d.Add(str.Key, 0);

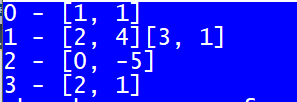
nov[str.Key] = false;

}

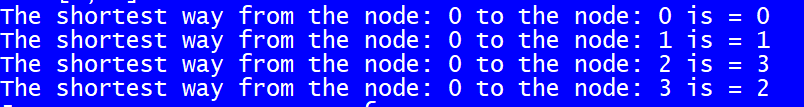
}

}

Входные данные



Выходные данные



## Веса IV С Флойд

В графе могут быть циклы отрицательного веса.

Задача 19. Найти все такие пары вершин, что между ними существует путь сколько угодно малой длины (То есть нужно найти достижимые вершины из вершин, которые лежат в отрицательном цикле).

Программы код

Класс Program

class Program

{

static void Main(string[] args)

{

Graph\_R graph = new Graph\_R();

graph.Show();

TaskFloyd(graph); //oriented - РАботает

}

static void TaskFloyd(Graph\_R graph)

{

graph.Floyd();

}

}

В класс Graph добавим реализацию алгоритма Форда

public void Floyd()

{

int[][] d = new int[Size][];

for (int i = 0; i < Size; i++)

{

d[i] = new int[Size];

for (int j = 0; j < Size; j++)

{

if (i == j)

{

d[i][j] = 0;

}

else

{

if (graph[i] != null)

{

if (graph[i].ContainsKey(j))

{

int value;

graph[i].TryGetValue(j, out value);

d[i][j] = value;

}

else

{

d[i][j] = int.MaxValue;

}

}

else

{

d[i][j] = int.MaxValue;

}

}

}

}

ShowMassive(d);

Console.WriteLine();

for (int pos = 0; pos < 2; pos++)

{

for (int j = 0; j < Size; j++)

{

for (int i = 0; i < Size; i++)

{

for (int k = 0; k < Size; k++)

{

if (d[i][k] < int.MaxValue && d[k][j] < int.MaxValue)

{

d[i][j] = Math.Min(d[i][j], d[i][k] + d[k][j]);

}

}

}

}

}

ShowMassive(d);

Console.WriteLine();

for (int i = 0; i < Size; ++i)

for (int j = 0; j < Size; ++j)

for (int t = 0; t < Size; ++t)

if (d[i][t] < int.MaxValue && d[t][j] < int.MaxValue && (d[t][t]) < 0)

{

d[i][j] = -int.MaxValue;

}

for (int i = 0; i < Size; ++i)

for (int j = 0; j < Size; ++j)

{

if (d[i][j] == -int.MaxValue)

{

Console.WriteLine("(" + i + "," + j + ")");

}

}

}

public void ShowMassive(int[][] d)

{

for (int k = 0; k < Size; k++)

{

for (int j = 0; j < Size; j++)

{

Console.Write(d[k][j] + " ");

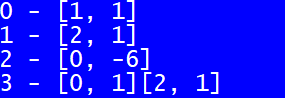
}

Console.WriteLine();

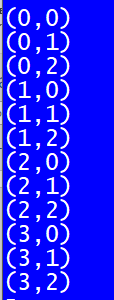
}

}

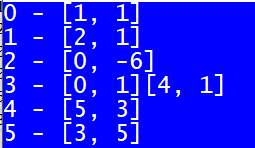
Входные данные



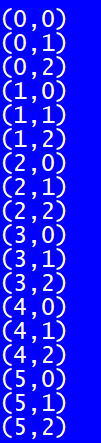
Выходные данные (пары вершин)



Входные данные



Выходные данные (пары вершин)



# ПОТОКИ

## V Нахождение максимального потока

Решить задачу на нахождение максимального потока алгоритмом «Проталкивание предпотока».

Программный код.

Организация графа немного изменила. Теперь каждая вершина имеет своей список дуг. Дуга организована в отдельном классе.

public class ExtArc : Arc

{

public ExtArc(int to) : base(to,0)

{

}

}

public class Arc

{

private int to;

private int capacity;

private int flow;

public Arc() { }

public Arc(int to)

{

this.to = to;

}

public Arc(int to, int capacity)

{

this.to = to;

this.capacity = capacity;

}

public Arc(int to, int capacity, int flow)

{

this.to = to;

this.capacity = capacity;

this.flow = flow;

}

public Arc(Arc a)

{

to = a.To;

capacity = a.Capacity;

flow = a.Flow;

}

public void changeFlow(int delta)

{

flow += delta;

}

public int To

{

get

{

return to;

}

set

{

to = value;

}

}

public int Flow

{

get

{

return flow;

}

set

{

flow = value;

}

}

public int Capacity

{

get

{

return capacity;

}

}

public void Show()

{

Console.Write("("+ to + ", " + capacity + ", " + flow + ")");

}

В класс Graph добавляем сам алгоритм.

public bool ContainsArc(int from, int to)

{

if (graph[from] != null)

{

foreach (var arc in graph[from])

if (arc.To == to)

return true;

}

return false;

}

public void ChangeFlow(int from, int to, int delta)

{

foreach (var arc in graph[from])

if (arc.To == to)

arc.Flow += delta;

}

public void MaxFlow(int source, int sink)

{

int[] excess = new int[Size]; // Избытки в вершинах

int[] height = new int[Size]; // Высоты вершин

// for (int i =0; i < Size; i++ )

foreach (var nodes in graph)

{

foreach (Arc element in graph[nodes.Key])

{

if (ContainsArc(element.To, nodes.Key) == false)

{

ExtArc arc = new ExtArc(nodes.Key);

graph[element.To].Add(arc);

}

if (nodes.Key == source)

{

// Пускаем максимальный поток по выходящим из истока дугам.

element.Flow = element.Capacity;

ChangeFlow(element.To, nodes.Key, -element.Capacity);

excess[element.To] = element.Capacity;

}

else if (element.To != source)

{

// Все остальные дуги имеют нулевой начальный поток.

element.Flow = 0;

}

}

}

// Высоты всех вершин кроме истока равны нулю.

height[source] = graph.Count;

// 2. Цикл проверки вершин

int h = 0;

bool excessFound; // Есть вершина с положительным избытком?

do

{

excessFound = false;

foreach (var nodes in graph)

{

if (nodes.Key == source || nodes.Key == sink)

{

continue;

}

if (excess[nodes.Key] > 0)

{

// Нашли вершину с положительным избытком;

// Исследуем ее соседей.

excessFound = true;

int minHeight = 2 \* graph.Count; // Минимальная из высот соседей

int maxFlowArc = 0; // Максимум потока по дугам

Arc workArc = null; // Дуга, по которой можно

//протолкнуть дополнительный поток.

foreach (Arc arc in nodes.Value)

{

int to = arc.To;

if (arc.Capacity - arc.Flow > 0)

{

if (height[to] == height[nodes.Key] - 1)

{

// Можно выполнить проталкивание

if (arc.Capacity - arc.Flow > maxFlowArc)

{

maxFlowArc = arc.Capacity - arc.Flow;

workArc = arc;

}

}

// Оцениваем высоту вершины для подъема.

if (height[to] < minHeight)

{

minHeight = height[to];

}

}

}

if (workArc != null)

{

// Выполняем проталкивание предпотока по дуге

int flow = Math.Min(excess[nodes.Key], maxFlowArc);

excess[nodes.Key] -= flow;

excess[workArc.To] += flow;

workArc.Flow += flow;

ChangeFlow(workArc.To, nodes.Key, -flow);

Console.WriteLine("Push " + nodes.Key + "-> " + workArc.To +

"(" + flow + ")");

}

else

{

// Выполняем подъём

height[nodes.Key] = minHeight + 1;

Console.WriteLine("Lift " + nodes.Key + " to " +

height[nodes.Key]);

}

}

}

}

while (excessFound);

// 3. Убираем фиктивные дуги

foreach (var nodes in graph)

{

for (int i = 0; i < nodes.Value.Count; i++) //Так как размер коллекции

//изменяется, нельзя перечисление делать

{

if (nodes.Value[i].GetType() == typeof(ExtArc))

{

nodes.Value.Remove(nodes.Value[i]);

i--;

}

}

}

// 4. Вычисление величины потока

int maxFlow = 0;

Console.WriteLine("Arc Flow: ");

foreach (var arc in graph[source])

{

Console.WriteLine(arc.To);

Console.WriteLine(arc.Flow);

maxFlow += arc.Flow;

}

Console.WriteLine("Максимальный поток = " + maxFlow);

Console.WriteLine("h = " + h);

}

}

Класс Program

class Program

{

static void Main(string[] args)

{

Graph graph\_standart3 = new Graph(); //Граф созданные из файла

graph\_standart3.Show();

graph\_standart3.Show();

int source = 0; //Исток

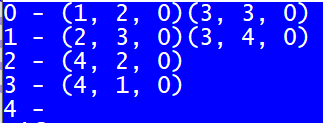
int sink = 4; // Сток

graph\_standart3.MaxFlow(source, sink);

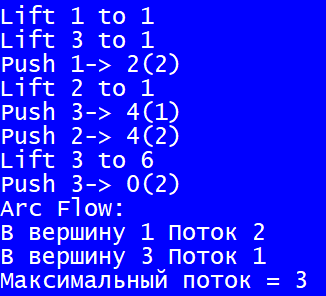
}

}

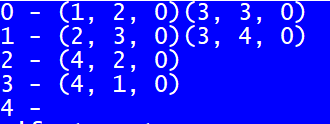
Входные данные исток 0, сток 4



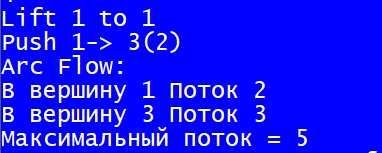
Выходные данные



Входные данные исток 0, сток 3



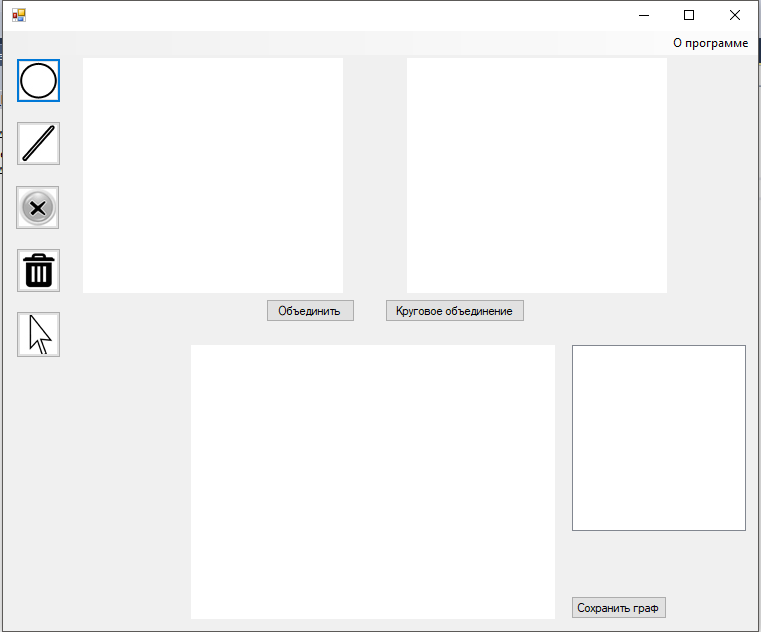
Выходные данные



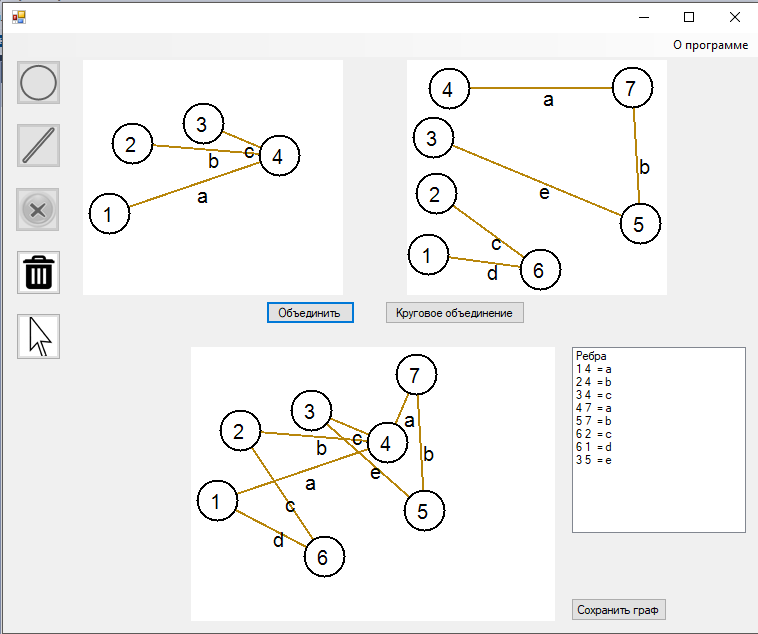
# ИТОГОВАЯ ЗАДАЧА

Дескпот приложение, позволяющее создать 2 графа и слить их в один. Так же рассмотреть подробно степени вершин и сохранить результаты в двух форматах: текст, изображение.

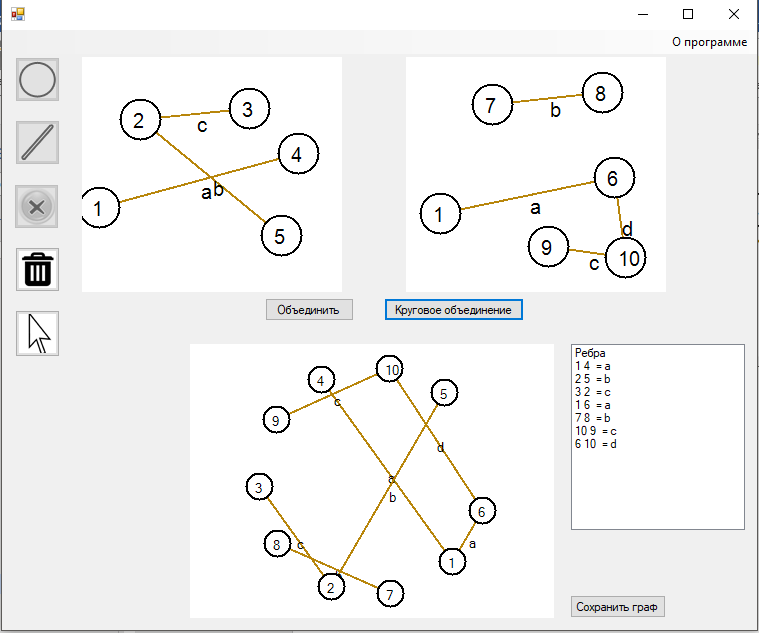
В верхних 2 окнах создаются графы. Их можно просто объединить (координаты остаются не изменными) и так, чтобы вершины расположились кругом. Следует, заметить, что максимальное число вершин в результирующем графе может быть не более 14.



Обычные объединение



Круговое обхединение



Граф хранить как список вершин и список ребер.

SortedList<int, Vertex> V1 = new SortedList<int, Vertex>();

List<Edge> E1 = new List<Edge>();

Класс Vertex

class Vertex

{

public int x, y;

public Vertex(int x, int y)

{

this.x = x;

this.y = y;

}

}

Класс Edge

class Edge

{

public int v1, v2;

public string name = null;

public Edge(int v1, int v2, int count)

{

this.v1 = v1;

this.v2 = v2;

char n= (char)('a' + count);

name += n;

}

public Edge(int v1, int v2)

{

this.v1 = v1;

this.v2 = v2;

}

public Edge(int v1, int v2, string name)

{

this.v1 = v1;

this.v2 = v2;

this.name = name;

}

}

Класс DrawGraph

class DrawGraph

{

Bitmap bitmap; //объект, используемый для работы с изображениями, определяемых

//данными о пикселях

Pen blackPen;

Pen redPen;

Pen darkGoldPen;

Graphics gr; //Инкапсулирует поверхность рисования GDI+

Font fo; //шрифт

Brush br; //заливка

PointF point; //точка

public int R = 20; //радиус окружности вершины

public DrawGraph(int width, int height)

{

bitmap = new Bitmap(width, height);

gr = Graphics.FromImage(bitmap); //Создает новый объект Graphics из указанного

//объекта Image

clearSheet();

blackPen = new Pen(Color.Black);

blackPen.Width = 2;

redPen = new Pen(Color.Red);

redPen.Width = 2;

darkGoldPen = new Pen(Color.DarkGoldenrod);

darkGoldPen.Width = 2;

fo = new Font("Arial", 15);

br = Brushes.Black;

}

public Bitmap GetBitmap()

{

return bitmap;

}

public int GetBitmapWidth()

{

return bitmap.Width;

}

public void clearSheet()

{

gr.Clear(Color.White);

}

public void drawVertex(int x, int y, string number)

{

gr.FillEllipse(Brushes.White, (x - R), (y - R), 2 \* R, 2 \* R); //заливка

gr.DrawEllipse(blackPen, (x - R), (y - R), 2 \* R, 2 \* R);

point = new PointF(x - R/2, y - R/2); //определяет точку

gr.DrawString(number, fo, br, point);

}

public void drawSelectedVertex(int x, int y)

{

gr.DrawEllipse(redPen, (x - R), (y - R), 2 \* R, 2 \* R);

}

public void drawEdge(Vertex V1, Vertex V2, Edge E)

{

if (E.v1 == E.v2)

{

gr.DrawArc(darkGoldPen, (V1.x - 2 \* R), (V1.y - 2 \* R), 2 \* R, 2 \* R,

90, 270);

point = new PointF(V1.x - (int)(2.75 \* R), V1.y - (int)(2.75 \* R));

gr.DrawString(E.name, fo, br, point);

drawVertex(V1.x, V1.y, (E.v1 + 1).ToString());

}

else

{

gr.DrawLine(darkGoldPen, V1.x, V1.y, V2.x, V2.y);

point = new PointF((V1.x + V2.x) / 2, (V1.y + V2.y) / 2);

gr.DrawString(E.name, fo, br, point);

drawVertex(V1.x, V1.y, E.v1 .ToString());

drawVertex(V2.x, V2.y, E.v2.ToString());

}

}

public void drawALLGraph(SortedList<int, Vertex> V, List<Edge> E)

{

//рисуем ребра

for (int i = 0; i < E.Count; i++)

{

if (E[i].v1 == E[i].v2)

{

gr.DrawArc(darkGoldPen, (V[E[i].v1].x - 2 \* R), (V[E[i].v1].y - 2 \* R),

2 \* R, 2 \* R, 90, 270);

point = new PointF(V[E[i].v1].x - (int)(2.75 \* R), V[E[i].v1].y –

(int)(2.75 \* R));

gr.DrawString(E[i].name, fo, br, point);

}

else

{

gr.DrawLine(darkGoldPen, V[E[i].v1].x, V[E[i].v1].y, V[E[i].v2].x,

V[E[i].v2].y);

point = new PointF((V[E[i].v1].x + V[E[i].v2].x) / 2, (V[E[i].v1].y +

V[E[i].v2].y) / 2);

gr.DrawString(E[i].name, fo, br, point);

}

}

//рисуем вершины

foreach (int node in V.Keys)

{

drawVertex(V[node].x, V[node].y, node.ToString());

}

}

public void drawALLGraphCombine(SortedList<int, Vertex> V, List<Edge> E,

double width, double heigth)

{

double alpha = alpha = 360 / (V.Count \* 2);

if (V.Count > 9)

{

this.R = 13;

fo = new Font("Arail", 10);

alpha = 360 / (V.Count \* 2 - 2);

}

else

{

this.R = 20;

fo = new Font("Arail", 15);

if (V.Count == 8)

{

alpha = 360 / (V.Count \* 2 + 2);

}

}

double RC = 80;

double[] center = { width / 2, heigth / 2 };

int k = 0;

foreach (int node in V.Keys)

{

if (node == 1)

{

V[node].x = (int)(center[0] + RC);

V[node].y = (int)(center[1] + RC);

}

else

{

//определяем координаты по часовой стрелке

V[node].x = (int)(center[0] + (V[k].x - center[0]) \* Math.Cos(alpha) –

(V[k].y - center[1]) \* Math.Sin(alpha));

V[node].y = (int)(center[1] + (V[k].x - center[0]) \* Math.Sin(alpha) +

(V[k].y - center[1]) \* Math.Cos(alpha));

}

k = node;

}

//рисуем ребра

for (int i = 0; i < E.Count; i++)

{

if (E[i].v1 == E[i].v2)

{

gr.DrawArc(darkGoldPen, (V[E[i].v1].x - 2 \* R), (V[E[i].v1].y - 2 \* R),

2 \* R, 2 \* R, 90, 270);

point = new PointF(V[E[i].v1].x - (int)(2.75 \* R), V[E[i].v1].y –

(int)(2.75 \* R));

gr.DrawString(E[i].name, fo, br, point);

}

else

{

gr.DrawLine(darkGoldPen, V[E[i].v1].x, V[E[i].v1].y, V[E[i].v2].x,

V[E[i].v2].y);

point = new PointF((V[E[i].v1].x + V[E[i].v2].x) / 2, (V[E[i].v1].y +

V[E[i].v2].y) / 2);

gr.DrawString(E[i].name, fo, br, point);

}

}

//рисуем вершины

foreach (int node in V.Keys)

{

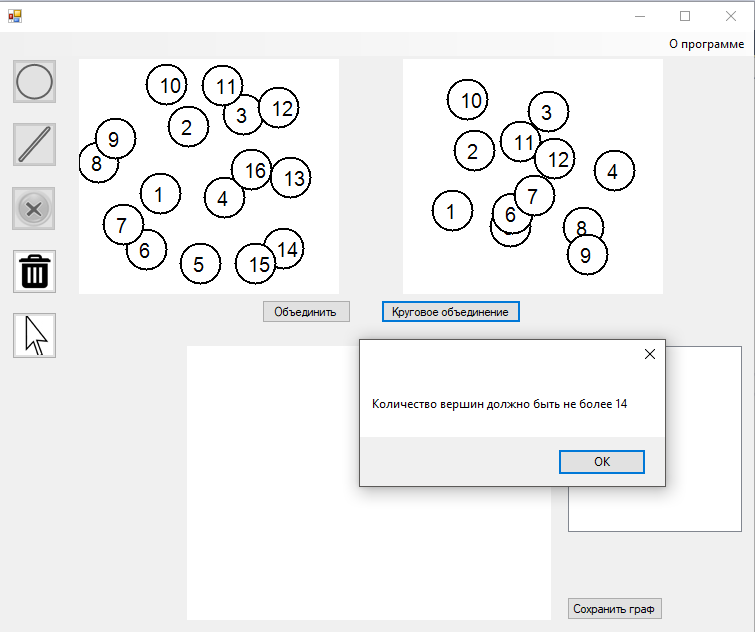
drawVertex(V[node].x, V[node].y, node.ToString());

}

}

}

Если ввести больше 14 вершин, то будет сообщение о том, что первышено количество вершин.



Пример текстового файла после сохранения в него данных.

Вершины графа 1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Ребра графа 1

Вершины графа 2

1 2 3 4 5 6 7 8 9 10 11 12

Ребра графа 2

Вершины графа 1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Ребра графа 1

Вершины графа 2

1 2 3 4 5 6 7 8 9 10 11 12

Ребра графа 2

Вершины графа 1

1 2 3 4

Ребра графа 1

4 3 a

1 4 b

2 4 c

Вершины графа 2

1 2 3 4 5 6 7 8

Ребра графа 2

8 7 a

3 4 b

5 6 c

2 3 d

1 1 e

2 8 f

Вершины графа 3

1 2 3 4 5 6 7 8

Ребра графа 3

4 3 a, b

1 4 b

2 4 c

8 7 a

5 6 c

2 3 d

1 1 e

2 8 f

Соответствующее изображение результирующего графа.

