**Отчет по практике по дисциплине «Теория графов»**

Подготовил студент

341 группы

факультета КНиИТ

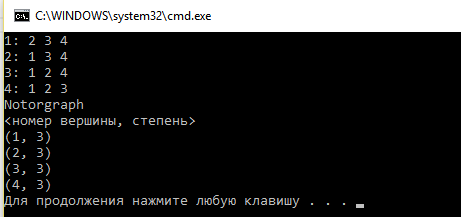
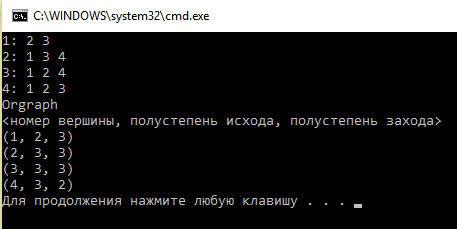
Акимов Артем

Саратов 2015

# I а Список смежности

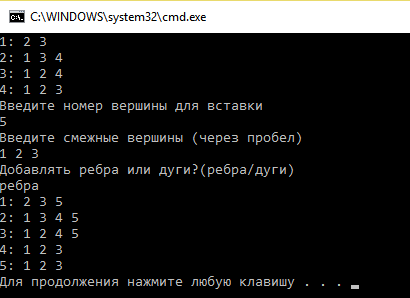
## 1.Для каждой вершины графа подсчитать её степень.

При помощи метода VariantOfGraph() определим вариант входного графа. Отсюда следует, что нужно Degree() – полустепени орграфа(OutsideDegree() и InsideDegree()) или степени вершин неорграфа.



## 23.Добавить в граф новую вершину.

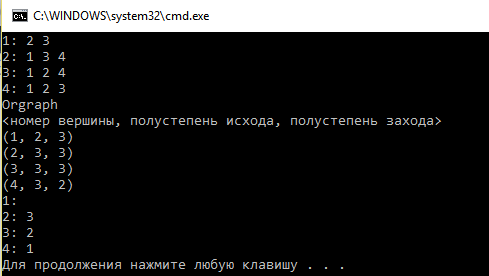
При помощи метода AddVertex() произведем добавление вершины и смежных ребер(дуг) с другими старыми вершинами.



# I б Список смежности (несколько графов)

## 10.Вывести список смежности подграфа данного орграфа, получающегося удалением дуг, соединяющих вершины разной чётности.

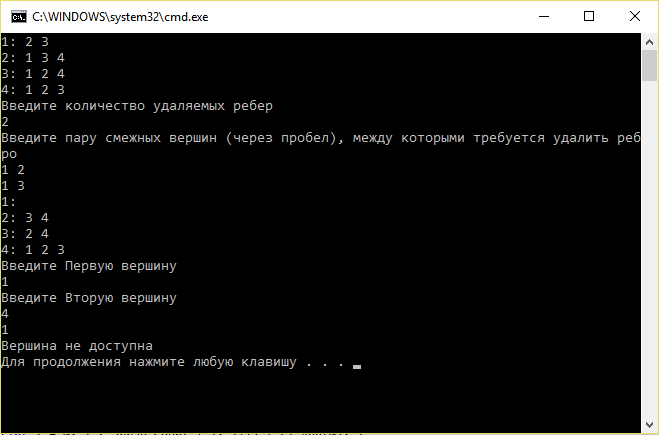
Для выведения подграфа был создан класс GraphCopy (при помощи метода Copy() производится копирование исходного графа), в котором производится удаление дуг при помощи метода DeleteEdges(), который вызывается в методе Parity()(ищет вершины разной четности).



# II Обходы (список смежности + обходы в глубину и ширину)

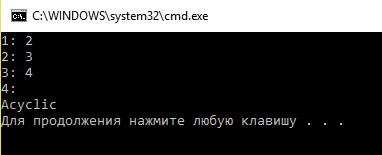
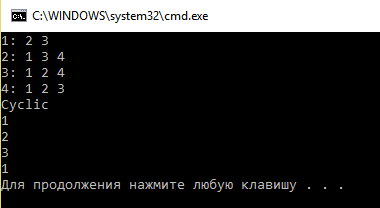
## 10. Определить, можно ли добиться того, чтобы из вершины *u* нельзя было попасть в вершину *v*, закрыв заданные *k* рёбер.

Метод Way(), который запускает процесс проверки и запрашивает вершины u и м, число k. В GraphCopy, в котором производится удаление дуг при помощи метода DeleteEdges(edges)(edges получаем при вызове метода Action(), который запрашивает количество дуг и сами дуги). Существует ли путь из требуемых вершин друг в друга определяется при помощи метода DfsForWay().



## 18. Проверить граф на ацикличность.

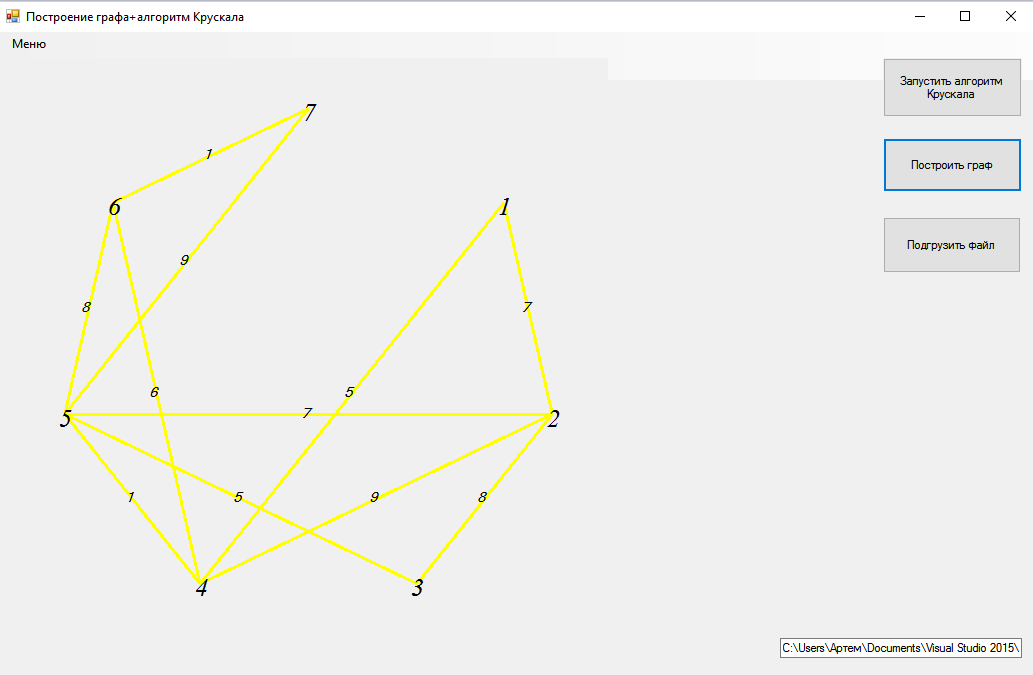
Метод Acycle() запускает поиск цикла при помощи метода DfsForAcycle(), если он существует, то выведет его в консоль.



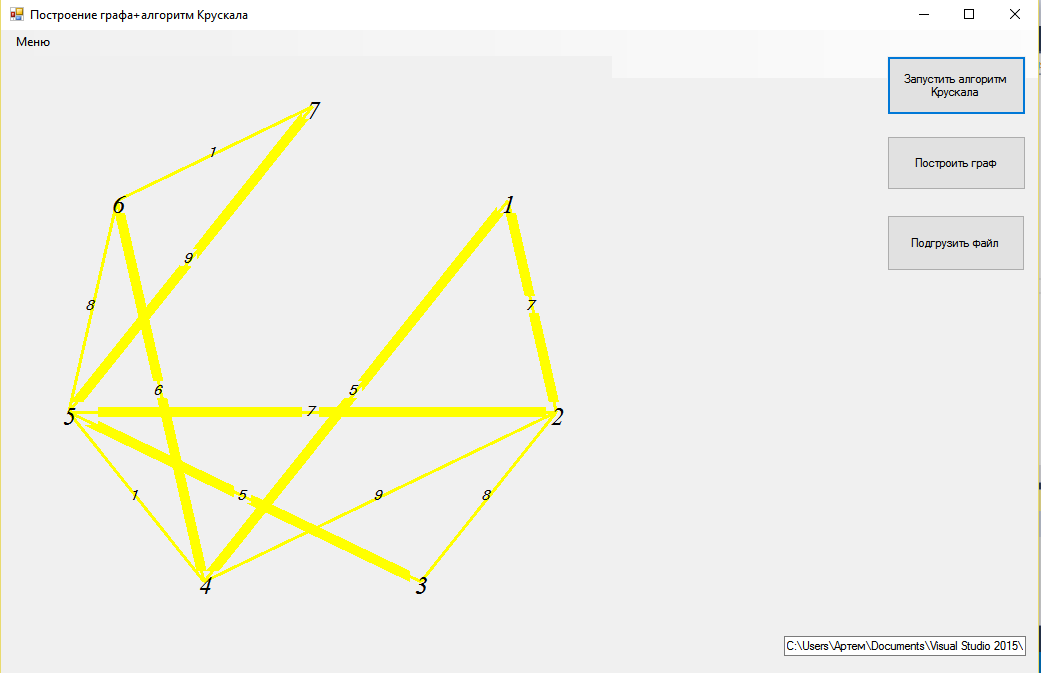
# III Каркас (Краскал)+Визуализация

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

Файл, с алгоритмом приложен в письме.Нажимаем последовательно “Подгрузить файл”->”Построить граф”->”Запустить алгоритм Крускала”.

****В Меню описывайтся тип входных данных, инструкция, формат входного файла.

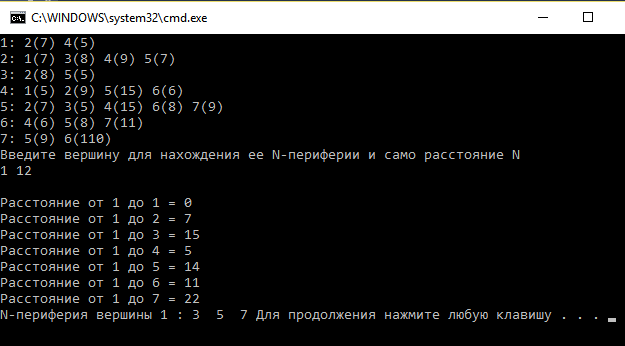
Результат работы алгоритма Крускала.

****

# IV Веса (Дейкстра)

## 7. N-периферией для вершины называется множество вершин, расстояние от которых до заданной вершины больше *N*. Определить *N*-периферию для заданной вершины графа.

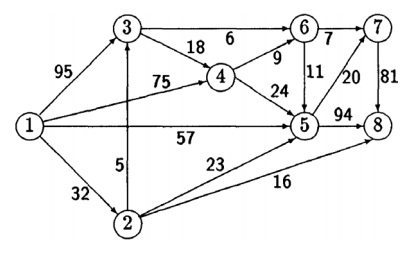
В методе Perifery () запустим алгоритм Дейкстра(метод Dijkstr()), который найдет кратчайшие пути из данной вершины до всех остальных. После это выведет все вершины, путь до которых длиннее N.



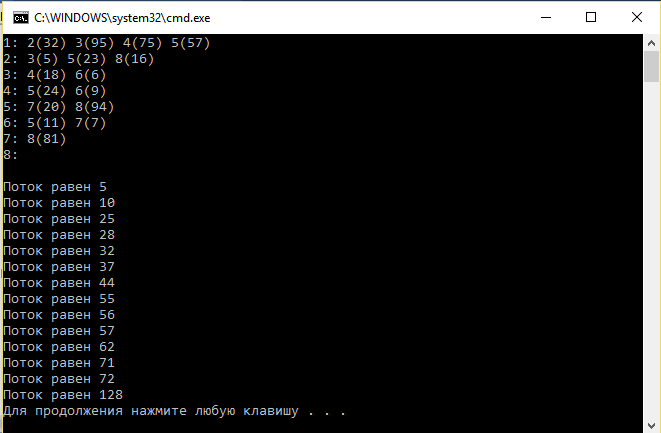
# V Потоки(Форд-Фалькерсон)

## Найти максимальный поток.

Запускаем алгоритм поиска максимального пути из вершины u в вершину v в методе MaxFlow(). В нем преобразуем список смежности с весами в матрицу смежности, так как алгоритм Форда-Фалькерсона работает на матрице смежности. Запустим метод Flow(), который будет искать поток из u в v пока путь существует(поиск пути осуществляем при помощи метода FindPath()). Если пути нет, то сумма найденных ранее потоков и есть максимальный поток



Результат работы алгоритма.



**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using QuickGraph;

using System.Diagnostics;

namespace Pract16.II

{

class Program

{

public static void Main()

{

Dictionary<string, List<int>> array = new Dictionary<string, List<int>>();

//var value = new List<int>();

Graph g = new Graph();

string key;

string needed;

array = g.Print();

}

}

}

**GraphCopy.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Pract16.II

{

class GraphCopy : Graph

{

public List<Tuple<string, string>> Action()

{

Console.WriteLine("Введите количество удаляемых ребер");

int count = int.Parse(Console.ReadLine());

Console.WriteLine("Введите пару смежных вершин (через пробел), между которыми требуется удалить ребро");

var edges = new List<Tuple<string, string>>();

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

{

string[] v = Console.ReadLine().Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

var pair = Tuple.Create(v[0], v[1]);

edges.Add(pair);

}

return edges;

}

public void CopyArray(Dictionary<string, List<int>> array1)

{

foreach (var kvp in array)

{

array1.Add(kvp.Key, kvp.Value);

}

}

public Dictionary<string, List<int>> DeleteEdges()

{

CopyArray(array1);

var edges = Action();

foreach (var item in edges)

{

array1[item.Item1].Remove(Convert.ToInt32(item.Item2));

array1[item.Item2].Remove(Convert.ToInt32(item.Item1));

}

foreach (var kvp in array)

{

Console.WriteLine("{0}: {1}",

kvp.Key, String.Join(" ", kvp.Value));

}

return array1;

}

public Dictionary<string, List<int>> DeleteEdges(List<Tuple<string, string>> edges)

{

CopyArray(array1);

foreach (var item in edges)

{

array1[item.Item1].Remove(Convert.ToInt32(item.Item2));

array1[item.Item2].Remove(Convert.ToInt32(item.Item1));

}

foreach (var kvp in array1)

{

Console.WriteLine("{0}: {1}",

kvp.Key, String.Join(" ", kvp.Value));

}

return array1;

}

public void Parity()

{

var vertex\_degrees = Degree();

List<Tuple<string, string>> edges = new List<Tuple<string, string>>();

foreach (var item in array)

{

foreach (var i in item.Value)

if ((vertex\_degrees[i-1].Item2%2) != (vertex\_degrees[Convert.ToInt32(item.Key)-1].Item2 % 2))

{

var edge = Tuple.Create(item.Key,i.ToString());

edges.Add(edge);

}

}

DeleteEdges(edges);

}

}

}

**Graph.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace Pract16.II

{

class Graph

{

public Dictionary<string, List<int>> array = new Dictionary<string, List<int>>();

public Dictionary<string, List<int>> array1 = new Dictionary<string, List<int>>();

public List<Tuple<int, Tuple<int, int>>> arrayWeighted = new List<Tuple<int, Tuple<int, int>>>();

public Dictionary<int, List<Tuple<int, int>>> array1Weighted = new Dictionary<int, List<Tuple<int, int>>>();

public string variant { get; set; }

public List<int> way = new List<int>();

public int insideDegree, outsideDegree;

public List<int> cl = new List<int>();

public List<int> p = new List<int>();

public List<int> d = new List<int>();

public List<bool> u = new List<bool>();

public int cycle\_st, cycle\_end;

int cost = 0;

int iterator = 0;

int n;

bool WeightedOrNot = false;

public Graph(Graph previousGraph)

{

array = previousGraph.array; ;

}

public Graph()

{

string[] mas = File.ReadAllLines(@"F:\Visual Studio 2013/Graph.txt");

n = int.Parse(mas[0]);

if (!mas[2].Contains(")"))

{

WeightedOrNot = true;

for (int i = 1; i < n + 1; i++)

{

string[] mas2 = mas[i].Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

List<int> matr = new List<int>();

for (int j = 1; j < mas2.Length; j++)

{

matr.Add(int.Parse(mas2[j]));

}

array.Add(mas2[0], matr);

}

}

else

{

string[] m;

for (int i = 1; i < n + 1; i++)

{

string[] mas2 = mas[i].Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

List<Tuple<int, int>> matr = new List<Tuple<int, int>>();

for (int j = 1; j < mas2.Length; j++)

{

m = mas2[j].Split(new char[] { '(', ')' }, StringSplitOptions.RemoveEmptyEntries);

Tuple<int, int> pair = Tuple.Create(int.Parse(m[0]), int.Parse(m[1]));

matr.Add(pair);

}

array1Weighted.Add(int.Parse(mas2[0]), matr);

}

foreach (var item in array1Weighted)

{

var value = item.Value;

foreach (var i in value)

{

Tuple<int, int> newPart = new Tuple<int, int>(item.Key, i.Item1);

Tuple<int, Tuple<int, int>> newItem = new Tuple<int, Tuple<int, int>>(i.Item2, newPart);

arrayWeighted.Add(newItem);

}

}

}

}

public Graph(int size, int first, int second)

{

Random rnd = new Random();

for (int i = 1; i <= size; i++)

{

int v = rnd.Next(second + 1);

int k = 0;

int sized = 0;

List<int> matr = new List<int>();

while (sized < v) //чтобы было рандомное число смежных вершин и количество ребер не превышало количество вершин и не было петель

{

k = rnd.Next(first, second + 1);

matr.Add(k);

sized++;

}

var distinct = matr.Distinct();//!повторения

matr = distinct.ToList();

matr.Sort();

if (matr.Contains(i))

{

matr.Remove(i);

}

array.Add(i.ToString(), matr);

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public void AddVertex(string key, List<int> value)

{

if (WeightedOrNot == true)

{

Console.WriteLine("Введите номер вершины для вставки");

key += int.Parse(Console.ReadLine());

Console.WriteLine("Введите смежные вершины (через пробел)");

string[] v = Console.ReadLine().Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

Console.WriteLine("Добавлять ребра или дуги?(ребра/дуги)");

string edges\_or\_not = Console.ReadLine();

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

{

value.Add(int.Parse(v[j]));

}

array.Add(key, value);

if (edges\_or\_not == "ребра" || edges\_or\_not == "Ребра" || edges\_or\_not == "РЕБРА")

{

foreach (var i in value)

{

if (value.Contains(i))

{

List<int> values = array[i.ToString()];

values.Add(Convert.ToInt32(key));

values.Sort();

array[i.ToString()] = values;

}

}

}

else

{

Console.WriteLine("Введите вершины, из которых есть ребро в новую (через пробел)");

string[] v2 = Console.ReadLine().Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries); ;

for (int j = 0; j < v2.Length; j++)

{

array[v2[j]].Add(Convert.ToInt32(key));

array[v2[j]].Sort();

}

}

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public void DeleteVertex(string vertex)

{

if (WeightedOrNot == true)

{

array.Remove(vertex);

foreach (var pair in array)

{

foreach (var item in pair.Value)

{

if (item == Convert.ToInt32(vertex))

{

pair.Value.Remove(Convert.ToInt32(vertex));

break;

}

}

}

}

else

{

array1Weighted.Remove(Convert.ToInt32(vertex));

foreach (var item in array1Weighted)

{

foreach (var i in item.Value)

{

if (i.Item1 == Convert.ToInt32(vertex))

item.Value.Remove(i);

}

}

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public List<int> DfsForWay(string key)

{

int vertex = int.Parse(key);

way.Add(vertex);

cl[vertex] = 1;

foreach (var i in array1[key])

if (cl[i] == 0)

DfsForWay(i.ToString());

cl[vertex] = 2;

return way;

}

public void Way(out string key, out string needed)

{

GraphCopy copy = new GraphCopy();

array1 = copy.DeleteEdges();

for (int i = 0; i < array.Count + 1; i++) { cl.Add(0); }

Console.WriteLine("Введите Первую вершину");

key = Console.ReadLine();

Console.WriteLine("Введите Вторую вершину");

needed = Console.ReadLine();

DfsForWay(key);

Console.WriteLine(String.Join(" ", way));

if ((way.Contains(Convert.ToInt32(key)) && (way.Contains(Convert.ToInt32(needed)))))

Console.WriteLine("Вершина доступна");

else Console.WriteLine("Вершина не доступна");

cl.Clear();

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public bool dfsForAcycle(string key)

{

List<int> cycle = new List<int>();

int vertex = int.Parse(key);

cl[vertex] = 1;

foreach (var i in array[key])

{

int to = i;

if (cl[to] == 0)

{

p[to] = vertex;

if (dfsForAcycle((to.ToString()))) return true;

}

else if ((cl[to] == 1) && (p[vertex] != to))

{

cycle\_end = vertex;

cycle\_st = to;

return true;

}

}

cl[vertex] = 2;

return false;

}

public void Acycle()

{

for (int i = 0; i < array.Keys.Count + 1; i++) { cl.Add(0); }

for (int i = 0; i < array.Keys.Count + 1; i++) { p.Add(-1); }

cycle\_st = -1;

foreach (var i in array.Keys)

if (dfsForAcycle(i))

break;

if (cycle\_st == -1)

Console.WriteLine("Acyclic");

else

{

Console.WriteLine("Cyclic");

List<int> cycle = new List<int>();

cycle.Add(cycle\_st);

for (int v = cycle\_end; v != cycle\_st; v = p[v])

cycle.Add(v);

cycle.Add(cycle\_st);

cycle.Reverse();

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

Console.WriteLine("{0}", cycle[i]);

}

p.Clear();

cl.Clear();

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public string VariantOfGraph()

{

foreach (var item in array)

{

foreach (var i in item.Value)

{

if (array[i.ToString()].Contains(Convert.ToInt32(item.Key))) variant = "Notorgraph";

else { variant = "Orgraph"; return variant; }

}

}

return variant;

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public int InsideDegree(string item)

{

insideDegree = 0;

foreach (var i in array)

{

if (i.Value.Contains(Convert.ToInt32(item))) insideDegree++;

}

return insideDegree;

}

public int OutsideDegree(string item)

{

outsideDegree = array[item].Count;

return outsideDegree;

}

public List<Tuple<string, int>> Degree()

{

List<Tuple<string, int>> listDegreesNotOr = new List<Tuple<string, int>>();

List<Tuple<string, int, int>> listDegreesOr = new List<Tuple<string, int, int>>();

foreach (var item in array)

{

if (item.Value.Count != 0)

{

foreach (var i in item.Value)

if (VariantOfGraph() == "Notorgraph")

{

var vertex\_degree = Tuple.Create(item.Key, item.Value.Count);

listDegreesNotOr.Add(vertex\_degree);

break;

}

else if (VariantOfGraph() == "Orgraph")

{

var vertex\_degree\_degree = Tuple.Create(item.Key, OutsideDegree(item.Key), InsideDegree(item.Key));

listDegreesOr.Add(vertex\_degree\_degree);

break;

}

}

else

{

var vertex\_degree\_degree = Tuple.Create(item.Key, item.Value.Count, InsideDegree(item.Key));

listDegreesOr.Add(vertex\_degree\_degree);

}

}

List<Tuple<string, int>> vertex\_degrees = new List<Tuple<string, int>>();

if (VariantOfGraph() == "Notorgraph")

{

Console.WriteLine(variant);

Console.WriteLine("<номер вершины, степень>");

foreach (var item in listDegreesNotOr)

{

Tuple<string, int> degrees = new Tuple<string, int>(item.Item1, item.Item2);

vertex\_degrees.Add(degrees);

Console.WriteLine(String.Join("", item));

}

}

else if (VariantOfGraph() == "Orgraph")

{

Console.WriteLine(variant);

Console.WriteLine("<номер вершины, полустепень исхода, полустепень захода>");

foreach (var item in listDegreesOr)

{

Tuple<string, int> degrees = new Tuple<string, int>(item.Item1, item.Item2 + item.Item3);

vertex\_degrees.Add(degrees);

Console.WriteLine(String.Join("", item));

}

}

return vertex\_degrees;

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public Dictionary<string, List<int>> Print()

{

if (WeightedOrNot == true)

foreach (var item in array)

{

Console.WriteLine("{0}: {1}",

item.Key, String.Join(" ", item.Value));

}

else

foreach (var item in array1Weighted)

{

string line = "" + item.Key + ": ";

foreach (var i in item.Value)

{

line += i.Item1.ToString() + "(" + i.Item2.ToString() + ")" + " ";

}

Console.WriteLine(line);

}

return array;

}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public void Kruskal()

{

arrayWeighted.Sort();

List<int> tree\_id = new List<int>();

List<Tuple<int, int>> res = new List<Tuple<int, int>>();

for (int i = 0; i < arrayWeighted.Count + 1; i++)

tree\_id.Add(i);

foreach (var kvp in arrayWeighted)

{

int a = kvp.Item2.Item1;

int b = kvp.Item2.Item2;

int l = kvp.Item1;

if (tree\_id[a] != tree\_id[b])

{

cost += l;

res.Add(Tuple.Create(a, b));

int old\_id = tree\_id[a], new\_id = tree\_id[b];

for (int j = 0; j < arrayWeighted.Count; ++j)

if (tree\_id[j] == old\_id)

tree\_id[j] = new\_id;

}

}

Console.WriteLine("Минимальный остов:" + "\n" + "<первая смежная вершина, вторая смежная вершина>");

foreach (var i in res)

{

Console.WriteLine("{0}", i);

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public int Flow(int[,] cap, int s, int t)

{

for (int flow = 0; ;)

{

List<int> way = new List<int>();

int df = FindPath(cap, new bool[cap.GetLength(0)], s, t, Int32.MaxValue);

Console.WriteLine("Минимум из ребер в пути равен {0}", df);

if (df == 0)

return flow;

Console.WriteLine("Поток равен {0}",flow += df);

Console.WriteLine();

}

}

public static int FindPath(int[,] cap, bool[] vis, int u, int t, int f)

{

if (u == t)

return f;

vis[u] = true;

for (int v = 0; v < cap.GetLength(0); v++)

if (!vis[v] && cap[u, v] > 0)

{

Console.WriteLine("[{0},{1}]", u + 1, v + 1);

int df = FindPath(cap, vis, v, t, Math.Min(f, cap[u, v]));

if (df > 0)

{

cap[u, v] -= df;

cap[v, u] += df;

if ((cap[u, v] != 0) && (u < v)) Console.WriteLine("[{0},{1}]ЄI i[{0},{1}]={2}", u + 1, v + 1, cap[u, v]); else if (cap[u, v] != 0) Console.WriteLine("[{0},{1}]ЄR r[{0},{1}]={2}", v + 1, u + 1, cap[v, u]);

if ((cap[v, u] != 0) && (v > u)) Console.WriteLine("[{0},{1}]ЄR r[{0},{1}]={2}", u + 1, v + 1, cap[v, u]); else if (cap[v, u] != 0) Console.WriteLine("[{0},{1}]ЄI i[{0},{1}]={2}", v + 1, u + 1, cap[u, v]);

Console.WriteLine();

return df;

}

}

return 0;

}

public void MaxFlow(int first, int second)

{

int[,] MyArray = new int[n, n];

int[] R = new int[n];

int[] I = new int[n];

foreach (var item in array1Weighted)

{

foreach (var i in item.Value)

{

MyArray[item.Key - 1, i.Item1 - 1] = i.Item2;

}

}

Console.WriteLine();

int strange = Flow(MyArray, first - 1, second - 1);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public List<int> Dijkstr(string key)

{

Console.WriteLine();

List<int> distance = new List<int>();

for (int i = 0; i < array1Weighted.Count+1; ++i)

{

distance.Add(int.MaxValue);

}

distance[Convert.ToInt32(key)] = 0;

for (int i = 0; i < array1Weighted.Count+1; ++i) { u.Add(false); }

for (int i = 0; i < array1Weighted.Count+1; ++i) { p.Add(-1); }

for (int i = 0; i < array1Weighted.Count+1;++i)

{

int v = -1;

for (int j = 0; j < n+1; ++j)

if (!u[j]&&(v == -1 || distance[j] < distance[v]))

v = j;

if (distance[v] > int.MaxValue-1)

break;

u[v] = true;

foreach (var value in array1Weighted[v])

{

int to = value.Item1;

int len = value.Item2;

if (distance[v] + len < distance[to])

{

distance[to] = distance[v] + len;

p[to] = v;

}

}

}

distance.Remove(distance[0]);

for (int i = 1; i < distance.Count+1; i++)

{

Console.WriteLine("Расстояние от "+key+" до "+i+" = "+String.Join(" ", distance[i-1]));

}

return distance;

}

public void Perifery(out string key)

{

Console.WriteLine("Введите вершину для нахождения ее N-периферии и само расстояние N");

string[] line = Console.ReadLine().Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

key = line[0];

int N = Convert.ToInt32(line[1]);

var distance = Dijkstr(key);

Console.Write("N-периферия вершины {0} :",key);

for (int i = 1; i < distance.Count + 1; i++)

{

if (distance[i - 1]>N)

Console.Write(" {0} ",i);

}

}

public void Parity()

{

GraphCopy copy = new GraphCopy();

copy.Parity();

}

}

}