| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |
| --- |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-2 «Прикладные информационные технологии»

**ОТЧЕТ   
о выполнении лабораторной работы №2**

**«Реализация абстрактного типа данных «Граф»»**

**по дисциплине   
«Технологии и методы программирования»**

**Вариант № 85**

Выполнил: студент 2 курса

группы БИСО-03-20

шифр 20Б0885

Турбина Виктория Александровна  
*(фио студента)*

Проверил:

*Филатов В.В.*

Москва 2021 г.

**Задание**

Реализовать в виде программы абстрактный тип данных «Граф» согласно варианту (***Номер варианта*** – **две последние цифры шифра студента, номера зачетной книжки**) с учетом заданного представления графа. Операторы (операции) АТД «Граф» *функционально* должны выполнять следующие операции (названия операций – *примерные*) ***(1 балл из 5)***:

1. FIRST(*v*) - возвращает ***индекс*** первой вершины, смежной с вершиной *v.* Если вершина *v* не имеет смежных вершин, то возвращается "нулевая" вершина Λ.
2. NEXT(*v*, *i*)- возвращает ***индекс*** вершины, смежной с вершиной *v*, следующий за индексом *i*. Если *i —* это индекс последней вершины, смежной с вершиной *v*, то возвращается Λ.
3. VERTEX(*v*, *i*) - возвращает ***вершину*** с индексом *i* из множества вершин, смежных с *v.*
4. ADD\_V(<имя>,<метка, mark>) - добавить УЗЕЛ
5. ADD\_Е(v, w, c) - добавить ДУГУ (здесь c — вес, цена дуги (v,w))
6. DEL\_V(<имя>) - удалить УЗЕЛ
7. DEL\_Е(v, w) – удалить ДУГУ
8. EDIT\_V(<имя>, <новое значение метки или маркировки>) - изменить метку (маркировку) УЗЛА

EDIT\_Е(v, w, <новый вес дуги>) - изменить вес ДУГИ

Алгоритм: Определить величину минимального разреза сети.

Способ представления графа: Список смежности

**Теоретическое введение**

**Графом** называется система объектов произвольной природы (вершин) и связок (ребер), соединяющих некоторые пары этих объектов.

**Список смежности** - один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответствует список, состоящий из «соседей» этой вершины.

**Минимальный разрез сети** - разрез с минимальной пропускной способностью**.**

**Листинг программы с расчетами**

using System;

using System.Collections.Generic;

namespace Graphs

{

class Program

{

public class Line

{

public Node Connected1 { get; }

public Node Connected2 { get; }

public List<Node> connects { get; }

public int weight { get; set; }

//создание конструктора линий

public Line(Node v, Node w, int c)

{

this.Connected1 = v; //первый узел

this.Connected2 = w; //второй узел

//дуга, состоящая из 2 узлов v и w

connects = new List<Node> { v, w };

//вес дуги

weight = c;

}

public override string ToString() => connects[0].Name + "-" + connects[1].Name;

};

//создание конструктора узла

public class Node

{

public string Name { get; set; }

public List<Line> Lines { get; }

public Node(string Name)

{

this.Name = Name;

Lines = new List<Line>();

}

public Node() { }

public void ADD\_E(Line newLine)

{

Lines.Add(newLine);

}

//конструктор добавления дуги

public void ADD\_E(Node newNode, int weight)

{

Lines.Add(new Line(this, newNode, weight));

}

//удаление дуги

public void DEL\_E(string v, string w)

{

if (Name == v || Name == w)

{

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

{

if (Lines[i].connects[0].Name == w || Lines[i].connects[1].Name == w)

{

Lines.RemoveAt(i);

}

}

}

}

}

//создание конструктора графа (создание узлов и дуг) ---/--- логическое соединение

public class Graph

{

public List<Node> Nodes { get; }

public List<Line> Lines { get; }

public Node B { get; set; }

//из чего состоит граф (узлы, линии)

public Graph()

{

Nodes = new List<Node>();

Lines = new List<Line>();

B = new Node();

}

//добавление узла

public void AddNode(string name)

{

Nodes.Add(new Node(name));

}

public void setEnd()

{

B = Nodes[Nodes.Count - 1]; //Nodes.Count - 1

}

//конструктор поиска узлов

public Node FindNode(string name)

{

foreach (var d in Nodes)

{

if (d.Name.Equals(name))

{

return d;

}

}

return null;

}

// Добавление дуги в графе (должно указываться: первая вершина, вторая, вес дуги)

public void ADD\_E(string Node1, string Node2, int weight)

{

var d1 = FindNode(Node1);

var d2 = FindNode(Node2);

if (d1 != null && d2 != null)

{

d1.ADD\_E(d2, weight);

d2.ADD\_E(d1, weight);

Lines.Add(new Line(d1, d2, weight));

}

}

//возвращает индекс первой вершины, смежной с вершиной "X"(параметр matchName)

public int FIRST(string hisName)

{

foreach (var line in Lines)

{

if (line.connects[0].Name == hisName)

{

return Nodes.IndexOf(line.connects[1]);

}

}

return 0;

}

//удаление дуги

public void DEL\_E(string v, string w)

{

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

{

if (Nodes[j].Name == v || Nodes[j].Name == w)

{

for (int i = 0; i < Nodes[j].Lines.Count; i++)

{

if (Nodes[j].Lines[i].connects[0].Name == v && Nodes[j].Lines[i].connects[1].Name == w)

{

Nodes[j].Lines.RemoveAt(i);

}

}

}

}

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

{

if (Lines[i].connects[0].Name == v && Lines[i].connects[1].Name == w)

{

Lines.Remove(Lines[i]);

break;

}

}

}

//рассчет факториала для формулы сочетания

public static int Factorial(int N)

{

int output = 1;

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

{

output \*= i;

}

return output;

}

//формула сочетания

public static int C(int n, int k)

{

return Factorial(n) / Factorial(k) / Factorial(n - k);

}

//вывод графа

public void PrintGrapgh()

{

List<Line> Liines = new List<Line>();

string[,] incident = new string[Nodes.Count + 1, Lines.Count + 1]; //дополнительный двумерный массив для узла и дуги

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

{

for (int j = 0; j < Lines.Count + 1; j++)

{

if (i != 0 && j != 0)

{

incident[i, j] = " 0";

}

else

{

incident[i, j] = " V";

}

}

}

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

{

incident[i, 0] = Nodes[i - 1].Name;

}

for (int j = 1; j < Lines.Count + 1; j++)

{

incident[0, j] = Lines[j - 1].connects[0].Name + "-" + Lines[j - 1].connects[1].Name;

}

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

{

for (int j = 1; j < Lines.Count + 1; j++)

{

if (Lines[j - 1].connects[0] == Nodes[i - 1] || Lines[j - 1].connects[1] == Nodes[i - 1])

{

incident[i, j] = Lines[j - 1].connects[0].Name;

//incident[i, j] = " 1";

}

}

}

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

{

for (int j = 0; j < Lines.Count + 1; j++)

{

Console.Write(incident[i, j] + "\t");

}

Console.WriteLine("\n");

}

}

//функция поиска различных комбинаций ребер графа

public static List<Line> Combination(int index, int t, List<Line> A)

{

List<int> result = new List<int>() { 0 };

List<Line> mainResult = new List<Line>();

int n = A.Count;

int s = 0;

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

{

int j = result[i - 1] + 1;

while ((j < (n - t + i)) && ((s + C(n - j, t - i)) <= index))

{

s += C(n - j, t - i);

j++;

}

result.Add(j);

mainResult.Add(A[j - 1]);

}

result.RemoveAt(0);

return mainResult;

}

public bool[] ExistingWay(Node u, bool[] used = null)

{

if (used == null)

{

used = new bool[Nodes.Count];

}

used[Nodes.IndexOf(u)] = true;

Queue<Node> q = new Queue<Node>();

q.Enqueue(u);

while (q.Count > 0)

{

u = q.Peek();

q.Dequeue();

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

{

Node v = u.Lines[i].connects[1];

if (!used[Nodes.IndexOf(v)])

{

used[Nodes.IndexOf(v)] = true;

q.Enqueue(v);

}

}

}

return used;

}

public void minCut()

{

Graph temporary = new Graph();

List<Line> MST = new List<Line>();

List<Line> E = new List<Line>();

List<Line> Combine = new List<Line>();

setEnd();

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

{

for (int i = 1; i <= C(Lines.Count, j); i++)

{

Graph temp = new Graph();

foreach (var Node in Nodes)

{

temp.Nodes.Add(new Node(Node.Name));

}

foreach (var line in Lines)

{

string v = line.connects[0].Name;

string w = line.connects[1].Name;

int weight = line.weight;

temp.ADD\_E(v, w, weight);

}

Combine = Combination(i, j, Lines);

foreach (Line line in Combine)

{

temp.DEL\_E(line.connects[0].Name, line.connects[1].Name);

}

bool[] result = temp.ExistingWay(temp.Nodes[0]);

if (!result[Nodes.Count - 1])

{

for (int z = 0; z < Combine.Count; z++)

{

Console.Write(Combine[z]);

if (z != Combine.Count - 1)

{

Console.Write(", ");

}

}

Console.WriteLine();

}

}

}

}

}

static void Main(string[] args)

{

Graph graph = new Graph();

graph.AddNode("X1");

graph.AddNode("X2");

graph.AddNode("X3");

graph.AddNode("X4");

//graph.AddNode("X5");

graph.ADD\_E("X1", "X2", 12);

graph.ADD\_E("X1", "X3", 2);

graph.ADD\_E("X1", "X4", 5);

graph.ADD\_E("X3", "X2", 6);

graph.ADD\_E("X3", "X4", 1);

graph.ADD\_E("X4", "X2", 3);

//graph.ADD\_E("X1", "X2", 10);

//graph.ADD\_E("X1", "X4", 30);

//graph.ADD\_E("X1", "X5", 100);

//graph.ADD\_E("X2", "X3", 50);

//graph.ADD\_E("X3", "X5", 10);

//graph.ADD\_E("X4", "X3", 20);

//graph.ADD\_E("X4", "X5", 60);

graph.PrintGrapgh();

graph.minCut();

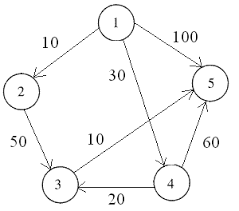
}

}

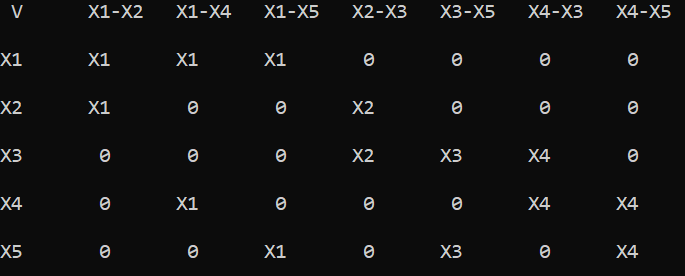
}

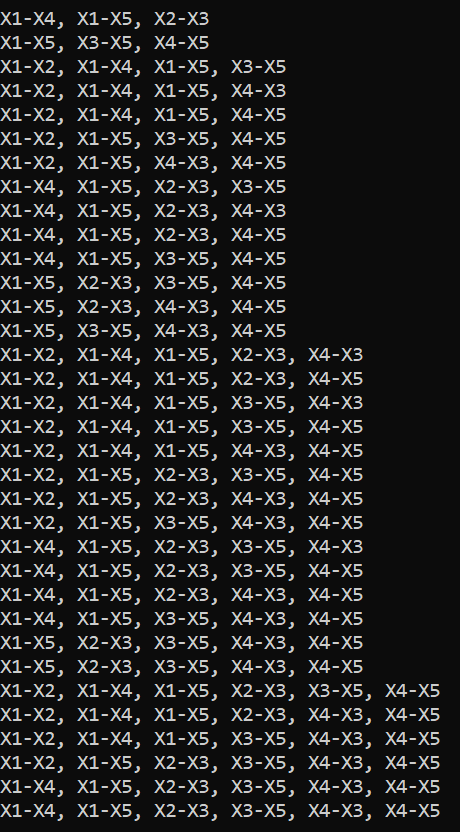
**Контрольные примеры**

1)

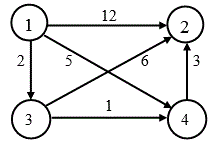


Вывод программы:

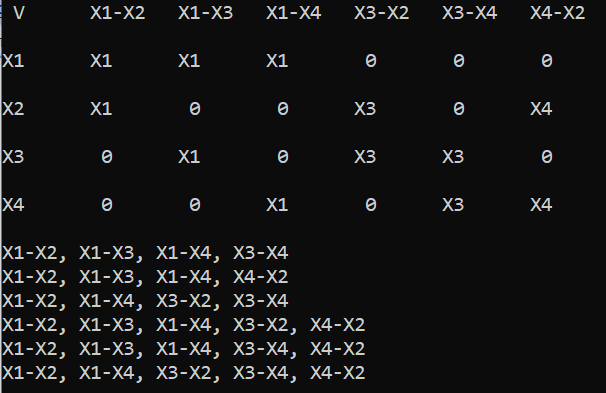




2)



Вывод программы:



**Выводы**

Был изучен абстрактный тип данных «Граф», понятие «Минимальный разрез сети» и реализован алгоритм поиска минимальных разрезов сети на языке c#.