Министерство образования и науки Российской Федерации  
Федеральное‌ ‌государственное‌ ‌бюджетное‌ ‌образовательное‌ ‌учреждение‌

высшего‌ ‌образования‌

**«Пермский национальный исследовательский  
политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЁТ**

**по лабораторной работе №6**

Дисциплина: «Дискретная математика и математическая

логика»

Тема: Получение остова графа

Выполнил работу

студент группы ИВТ-22-2б

Казанцев А.В.

Проверила

Старший преподаватель кафедры ИТАС

Рустамханова Г.И.

Пермь 2024

**Цель работы**

1. Создание консольного приложения «Получение остова графа»

2. Изучение теории дискретной математики об свойствах графов, алгоритмов Краскала и Прима.

3. Изучение и отработка навыков определения компонент связности, работы с матрицами достижимости и матрицей смежности.

**Постановка задачи**

1. Программа должен иметь возможность задавать систему функций через матрицу смежности, полученную из файла.
2. Программа должна выполнять следующие функции:
   1. Создание и вывод на экран матрицы дорог графа.
   2. Выполнять алгоритм Прима.
   3. Выполнять алгоритм Краскала.

**Ход работы**

Программа имеет возможность открыть файл с подходящий имеем.  
Программа считывает файл в класс MatrixGraf. Строки нормализируются для корректной работы. Файл закрывается.

Программа позволяет выводить матрицы смежности и достижимости.   
Позволяет узнать количество компонент связности графа.  
Позволяет вывести на экран алгоритм вычисления компонент связности и вершины этих компонент.  
Реализует поиск остова связанного графа через алгоритм красскала и алгоритм Прима.

Методы, используемые в классе MatrixGraf:

ReadMatrix (string path)– метод открывает и считывает файл в статический двумерный массив.

WriteMatrix (int[,] matrix) выводит на квадратную матрицу в консоль.

int[,] SumMatrix(int[,] matrix1, int[,] matrix2) – возвращает двумерный массив суммирующий две матрицы по особому правилу «1+1=1».

int[,] DegreeMatrix(int[,] matrix0) – возвращает двумерный массив, умноженный на статическую матрицу класса.

bool EqualsMatrix(int[,] matrix1, int[,] matrix2) возвращает истину если матрицы равны.

int[,] CopyMatrix(int[,] matrix) ) – возвращает копию двумерного массива.

int[,] CreateReachabilityMatrix()) – возвращает матрицу достижимости от статической матрицы класса.

Start()- запускает демонстрационную программу.

string[] IntoStr(int[,] matrix0) – возвращает массив переводит строки целочисленной матрицы в строковый массив.

void FindComp(int[,] matrix)- вычисление компонент связности по матрице достижимости и вывод ответа на экран.

Методы, используемые в классе Graph:

Graph Graph(int[,] matrix) – считывает из двумерного массива в объект класса Graph.

Sort() – сортирует хранилище рёбер хранимых в объекте по возрастанию веса ребра.  
Add() добавляет в граф класса ребро\ или ребра другого графа.

int Lenght() – возвращает вес графа.

Print() – выводит ребра графа в консоль.

Graph FindMinimumKraskala() – возвращает новый граф содержащий остов полученный алгоритмом Краскала.

Graph FindMinimumPrim(int starter = 1)– возвращает новый граф содержащий остов полученный алгоритмом Прима из начальной вершины

Методы, используемые в классе SetGraph:

Union(SetGraph set, Rib ribConect) – объединяет ребром 2 множества графов.

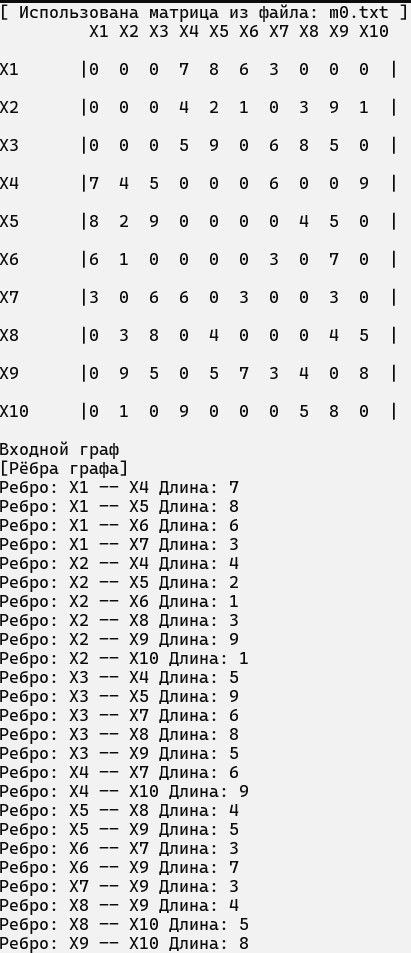
bool Contains(int vertex) – возвращает истину если хранилище вершин содержит некую вершину.

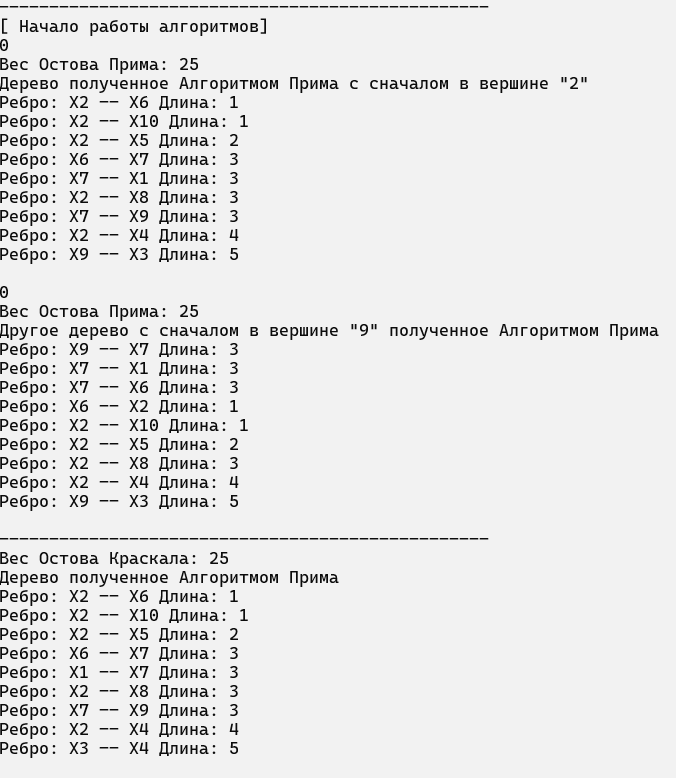
Методы, используемые в прочих классах:

SetGraph Find(int vertex) – возвращает множество содержащее искомую вершиной.

void Print() выводит в консоль объекта класса Rib.

Тесты





**Код программы**

public class Graph : IEnumerable<Rib>

{

public List<Rib> ribs;

public List<List<Rib>> grafs;

public Graph(int capacity)

{

grafs = new List<List<Rib>>();

for (int i = 0; i < capacity + 1; i++)

{

grafs.Add(new List<Rib>());

}

ribs = new List<Rib>();

}

public Graph(Graph graf)

{

this.ribs = graf.ribs;

this.grafs = graf.grafs;

}

public Graph(Rib val, int n = 10)

{

grafs = new List<List<Rib>>();

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

{

grafs.Add(new List<Rib>());

}

grafs[val.A].Add(val);

grafs[val.B].Add(val);

ribs = new List<Rib>() { val };

}

public Graph(int[,] matrix)

{

ribs = new List<Rib>();

grafs = new List<List<Rib>>();

for (int i = 0; i < matrix.GetLength(0) + 1; i++)

{

grafs.Add(new List<Rib>());

}

for (int i = 0; i < matrix.GetLength(0); i++)

{

for (int j = i + 1; j < matrix.GetLength(1); j++)

{

if (matrix[i, j] != 0)

{

grafs[i].Add(new Rib(i, j, matrix[i, j]));

grafs[j].Add(new Rib(j, i, matrix[i, j]));

ribs.Add(new Rib(i, j, matrix[i, j]));

}

}

grafs[i].Reverse();

}

}

public IEnumerator<Rib> Enumerator => ribs.GetEnumerator();

IEnumerator IEnumerable.GetEnumerator() => ribs.GetEnumerator();

public void Sort(IComparer<Rib> comparer)

{

ribs.Sort(comparer);

}

public void Sort()=>Sort(new RibCompareW());

public void Add(Graph graph)

{

foreach (Rib edge in graph)

{

ribs.Add(edge);

}

}

public void Add(List<Rib> graph)

{

foreach (Rib edge in graph)

{

ribs.Add(edge);

}

}

public int Lenght()

{

int weight = 0;

foreach (Rib rib in ribs)

{

weight += rib.weight;

}

return weight;

}

public void Remove(Rib edge)

{

ribs.Remove(edge);

grafs[edge.A].Remove(edge);

grafs[edge.B].Remove(edge);

}

public void Add(Rib edge)

{

ribs.Add(edge);

grafs[edge.A].Add(edge);

grafs[edge.B].Add(edge);

}

public void Print()

{

foreach (var item in ribs)

{

item.Print();

}

Console.WriteLine();

}

public Graph FindMinimumKraskala()

{

Sort();

var disjointSets = new SetCollection();

foreach (Rib r in this.ribs)

{

disjointSets.Add(r);

}

return disjointSets.setGraph.First().setGraph;

}

public Graph FindMinimumPrim(int starter = 1)

{

if (starter <= 0 || starter >= grafs.Count - 1)

{

Console.WriteLine( "Такой вершины нет\nВыбрана:Вершина 1. " );

starter = 1;

}

bool[] selected = new bool[grafs.Count - 1];

selected[starter - 1] = true;

grafs[starter - 1].Sort(new RibCompareW());

selected[grafs[starter - 1].First().B] = true;

var outGraph = new Graph(grafs.Count - 1);

var inGraph = new Graph(grafs[starter - 1].First(), grafs.Count - 1);

outGraph.Add(grafs[starter - 1]);

outGraph.Add(grafs[grafs[starter - 1].First().B]);

int k = 0;

while (inGraph.ribs.Count < 9)

{

outGraph.Sort();

foreach (var item in outGraph)

{

if (!selected[item.B])

{

inGraph.Add(item);

outGraph.Add(grafs[item.B]);

selected[item.B] = true;

break;

}

}

}

Console.WriteLine(k);

return inGraph;

}

public IEnumerator<Rib> GetEnumerator()

{

return ((IEnumerable<Rib>)ribs).GetEnumerator();

}

public class RibCompareW : IComparer<Rib>

{

int IComparer<Rib>.Compare(Rib? x, Rib? y)

{

if (y is null

&& x is null)

return 0;

if (y is null)

return 1;

if (x is null)

return -1;

if (x.weight == y.weight)

return x!.B.CompareTo(y!.B);

return x!.weight.CompareTo(y!.weight);

}

}

public class RibCompareA : IComparer<Rib>

{

int IComparer<Rib>.Compare(Rib? x, Rib? y) => x!.A > y!.A ? 1 : x.A == y.A ? 0 : -1;

}

public class RibCompareB : IComparer<Rib>

{

int IComparer<Rib>.Compare(Rib? x, Rib? y) => x!.B > y!.B ? 1 : x.B == y.B ? 0 : -1;

}

}

public class SetGraph

{

public Graph setGraph;

public List<int> Vertices;

public SetGraph(Rib rib)

{

setGraph = new Graph(rib);

Vertices = new List<int>();

Vertices.Add(rib.A);

Vertices.Add(rib.B);

}

public void Union(SetGraph set, Rib ribConect)

{

setGraph.Add(set.setGraph);

Vertices.AddRange(set.Vertices);

setGraph.Add(ribConect);

}

public void Add(Rib rib)

{

setGraph.Add(rib);

Vertices.Add(rib.A);

Vertices.Add(rib.B);

}

public bool Contains(int vertex)

{

return Vertices.Contains(vertex);

}

}

public class SetCollection

{

public List<SetGraph> setGraph;

public SetCollection()

{

setGraph = new List<SetGraph>();

}

public SetGraph Find(int vertex)

{

foreach (SetGraph set in setGraph)

{

if (set.Contains(vertex))

return set;

}

return null!;

}

public void Add(Rib rib)

{

SetGraph setA = Find(rib.A);

SetGraph setB = Find(rib.B);

if (setA != null && setB == null)

{

setA.Add(rib);

}

else if (setA == null && setB != null)

{

setB.Add(rib);

}

else if (setA == null && setB == null)

{

SetGraph set = new SetGraph(rib);

setGraph.Add(set);

}

else if (setA != null && setB != null)

{

if (setA != setB)

{

setA.Union(setB, rib);

setGraph.Remove(setB);

}

}

}

}

public class Rib

{

public int A;

public int B;

public int weight;

public Rib(int A, int B, int len)

{

this.A = A; this.B = B; this.weight = len;

}

public void Print()

{

Console.WriteLine($"Ребро: X{A + 1} -- X{B + 1} Длина: {weight}");

}

public int CompareTo(Rib other)

{

if (other == null)

return 1;

return weight.CompareTo(other.weight);

}

public Rib Reverse()

{

A = A + B;

B = A - B;

A = A - B;

return this;

}

}

public class MatrixGraf

{

public static int[,] matrix;

public MatrixGraf(string path)

{

string[] lines = File.ReadAllLines(path);

matrix = new int[lines.GetLength(0), (lines[0].Length + 1) / 2];

for (int i = 0; i < matrix.GetLength(0); i++)

{

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

for (int j = 0; j < matrix.GetLength(1); j++)

{

matrix[i, j] = int.Parse(buf[j]);

}

}

}

public static int[,] CopyMatrix(int[,] matrix)

{

int[,] res = new int[matrix.GetLength(0), matrix.GetLength(1)];

for (int i = 0; i < matrix.GetLength(0); i++)

{

for (int j = 0; j < matrix.GetLength(1); j++)

{

res[i, j] = matrix[i, j];

}

}

return res;

}

public static int[,] CreateReachabilityMatrix()

{

int[,] buf = new int[matrix.GetLength(0), matrix.GetLength(1)];

int[,] buf1 = new int[matrix.GetLength(0), matrix.GetLength(1)];

int[,] attainMat = new int[matrix.GetLength(0), matrix.GetLength(1)];

attainMat = SumMatrix(attainMat, matrix);

buf1 = CopyMatrix(matrix);

buf = CopyMatrix(attainMat);

int k = 0;

while (true)

{

buf1 = DegreeMatrix(buf1);

attainMat = SumMatrix(attainMat, buf1);

if (EqualsMatrix(buf, attainMat))

k++;

else

k = 0;

if (k == 2)

break;

buf = CopyMatrix(attainMat);

}

return attainMat;

}

public static int[,] DegreeMatrix(int[,] matrix0)

{

int[,] res = new int[matrix.GetLength(0), matrix.GetLength(1)];

for (int i = 0; i < matrix.GetLength(0); i++)

{

for (int j = 0; j < matrix.GetLength(1); j++)

{

for (int k = 0; k < matrix.GetLength(0); k++)

{

res[i, j] += matrix0[i, k] \* matrix[k, j];

}

if (res[i, j] > 0)

res[i, j] = 1;

}

}

return res;

}

public static bool EqualsMatrix(int[,] matrix1, int[,] matrix2)

{

bool isCheck = true;

isCheck = matrix1.GetLength(0) == matrix2.GetLength(0)

&& matrix1.GetLength(1) == matrix2.GetLength(1);

if (!isCheck)

return isCheck;

for (int i = 0; i < matrix1.GetLength(0); i++)

for (int j = 0; j < matrix2.GetLength(1) && isCheck; j++)

isCheck = matrix1[i, j] == matrix2[i, j];

return isCheck;

}

public void Exit()

{

Console.WriteLine("\n[ Нажмите любую кнопку для продолжения... ]\n[ Для выхода нажмите esc ]");

if (Console.ReadKey(true).Key == ConsoleKey.Escape)

Environment.Exit(0);

}

public static void FindComp(int[,] matrix)

{

string[] matrixStr = IntoStr(matrix);

Dictionary<string, List<string>> pairs = new Dictionary<string, List<string>>();

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

{

if (pairs.Keys.Contains(matrixStr[i]))

continue;

pairs[matrixStr[i]] = new List<string>() { "X" + (i + 1) };

for (int j = i + 1; j < matrixStr[i].Length; j++)

if (matrixStr[i] == matrixStr[j])

pairs[matrixStr[i]].Add("X" + (j + 1));

}

Console.WriteLine("Количество компонент связности: " + pairs.Keys.Count);

int r = 1;

foreach (var item in pairs)

{

Console.Write("Компонента " + r++ + ": \t");

foreach (var item1 in item.Value)

Console.Write(item1 + "\t");

Console.WriteLine();

}

}

public int CountComp()

{

int[,] matrix = CreateReachabilityMatrix();

string[] matrixStr = IntoStr(matrix);

Dictionary<string, List<string>> pairs = new Dictionary<string, List<string>>();

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

{

if (pairs.Keys.Contains(matrixStr[i]))

continue;

pairs[matrixStr[i]] = new List<string>() { "X" + (i + 1) };

for (int j = i + 1; j < matrixStr[i].Length; j++)

if (matrixStr[i] == matrixStr[j])

pairs[matrixStr[i]].Add("X" + (j + 1));

}

return pairs.Keys.Count();

}

public static string[] IntoStr(int[,] matrix0)

{

string[] res = new string[matrix0.GetLength(1)];

for (int i = 0; i < matrix0.GetLength(0); i++)

{

string str = "";

for (int j = 0; j < matrix0.GetLength(1); j++)

str += matrix0[j, i];

res[i] = str;

}

return res;

}

public static void ReadMatrix(string path)

{

string[] lines = File.ReadAllLines(path);

matrix = new int[lines.GetLength(0), (lines[0].Length + 1) / 2];

for (int i = 0; i < matrix.GetLength(0); i++)

{

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

for (int j = 0; j < matrix.GetLength(1); j++)

matrix[i, j] = int.Parse(buf[j]);

}

}

public void Run(string path)

{

Console.BackgroundColor = ConsoleColor.White;

Console.ForegroundColor = ConsoleColor.Black;

Console.Clear();

Console.WriteLine("Теория Графов");

ReadMatrix(path);

Console.WriteLine(path);

Console.WriteLine("[ Матрица Смежности ]");

WriteMatrix();

matrix = CreateReachabilityMatrix();

Console.WriteLine("[ Матрица Достижимости ]");

WriteMatrix();

Console.WriteLine("[ Количество компонент связности и их вершины ]");

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

FindComp(matrix);

}

public static int[,] SumMatrix(int[,] matrix1, int[,] matrix2)

{

int[,] res = new int[matrix.GetLength(0), matrix.GetLength(1)];

for (int i = 0; i < matrix.GetLength(0); i++)

{

for (int j = 0; j < matrix2.GetLength(1); j++)

{

res[i, j] = matrix2[i, j] + matrix1[i, j];

if (res[i, j] > 0)

res[i, j] = 1;

}

}

return res;

}

public void WriteMatrix()

{

if (matrix == null) { Console.WriteLine("Матрица пуста. "); return; }

Console.Write("\t ");

for (int i = 0; i < matrix.GetLength(0); i++)

Console.Write("X" + (i + 1) + " ");

Console.WriteLine("\n");

for (int i = 0; i < matrix.GetLength(0); i++)

{

Console.Write("X" + (i + 1) + "\t|");

for (int j = 0; j < matrix.GetLength(1); j++)

Console.Write(matrix[i, j] + " ");

Console.WriteLine("|\v");

}

}

public void Start()

{

Console.BackgroundColor = ConsoleColor.White;

Console.ForegroundColor = ConsoleColor.Black;

Console.Clear();

Console.WriteLine("Теория Графов");

string path;

for (int i = 1; i < 5; i++)

{

path = "g1" + i + ".txt";

Run(path);

Exit();

}

Console.WriteLine("\n[ Матрицы закончились] ");

}

}

public partial class Program

{

static void Main(string[] args)

{

Console.BackgroundColor = ConsoleColor.White;

Console.ForegroundColor = ConsoleColor.Black;

Console.Clear();

MatrixGraf mat = new MatrixGraf("m0.txt");

Console.WriteLine("[ Использована матрица из файла: m0.txt ]");

mat.WriteMatrix();

if (mat.CountComp() > 1)

{

Console.WriteLine("Граф не является связанным.");

mat.Exit();

}

Graph graf1 = new Graph(MatrixGraf.matrix);

Console.WriteLine("Входной граф\n[Рёбра графа]");

graf1.Print();

Console.WriteLine("-------------------------------------------------\n[ Начало работы алгоритмов]");

Graph grafr = graf1.FindMinimumPrim(2);

Console.WriteLine("Вес Остова Прима: " + grafr.Lenght()+ "\nДерево полученное Алгоритмом Прима с сначалом в вершине \"2\"");

grafr.Print();

grafr = graf1.FindMinimumPrim(9);

Console.WriteLine("Вес Остова Прима: " + grafr.Lenght() + "\nДругое дерево с сначалом в вершине \"9\" полученное Алгоритмом Прима");

grafr.Print();

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

grafr = graf1.FindMinimumKraskala();

Console.WriteLine("Вес Остова Краскала: "+grafr.Lenght()+ "\nДерево полученное Алгоритмом Прима");

grafr.Print();

}

}