МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Вычислительной техники»

**ОТЧЁТ**

по лабораторной работе №6

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Унарные и бинарные операции над графами»

Выполнил:

Студент группы 23ВВВ2

Монин Иван

Приняли:

доцент Юрова О.В.

доцент Митрохин М.А.

Пенза 2024

**Цель работы** – научиться создавать различные сложные динамические структуры данных.

**Лабораторное задание:**

**Задание 1**

Сгенерируйте (используя генератор случайных чисел) две матрицы M1, М2 смежности неориентированных помеченных графов G1, G2. Выведите сгенерированные матрицы на экран.

\* Для указанных графов преобразуйте представление матриц смежности в списки смежности. Выведите полученные списки на экран.

**Задание 2**

Для матричной формы представления графов выполните операцию:

а) отождествления вершин

б) стягивания ребра

в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

\* Для представления графов в виде списков смежности выполните операцию:

а) отождествления вершин

б) стягивания ребра

в) расщепления вершины

Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

**Задание 3**

Для матричной формы представления графов выполните операцию:

а) объединения G = G1 G2

б) пересечения G = G1 G2

в) кольцевой суммы G = G1  G2

Результат выполнения операции выведите на экран.

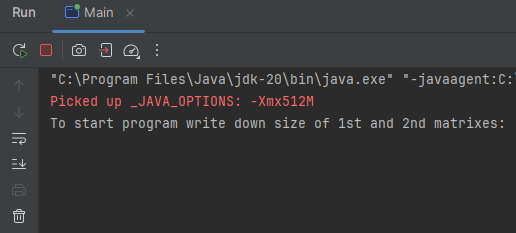
**Задание 4 \***

Для матричной формы представления графов выполните операцию декартова произведения графов G = G1 X G2.

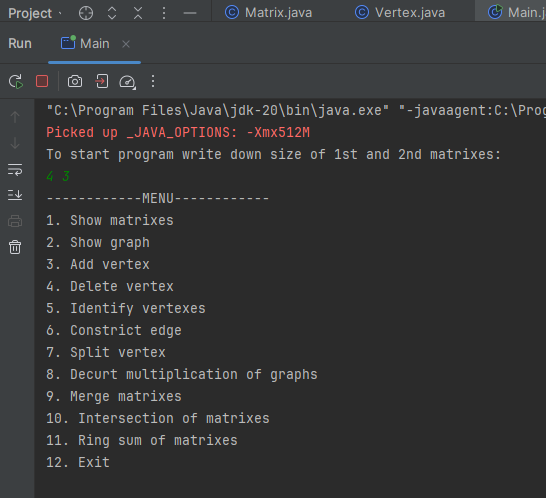
Результат выполнения операции выведите на экран.

**Задание 1**

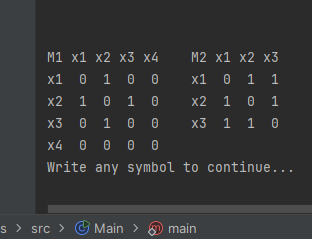
1. Для начала нас просят ввести размер 1 и 2 матриц



1. После предоставляется меню работы с программой

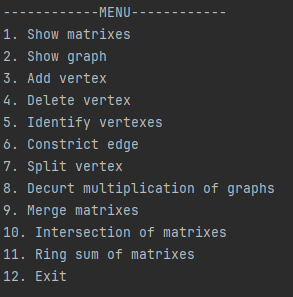


1. Для выполнения 1 задания необходимо сгенерировать 2 матрицы и создать для них списки смежности. Сразу после того, как мы ввели значения размер матриц, все было сгенерировано. Чтобы увидеть матрицы можно воспользоваться 1 пунктом меню:



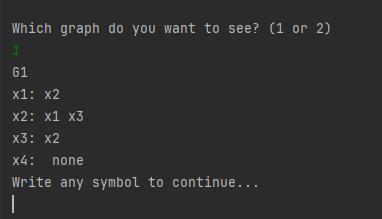
Для удобства был реализован вывод сразу двух матриц

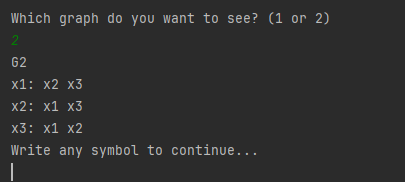
1. Чтобы вернуться в меню напишем любой символ.



Выберем 2 пункт меню чтобы отобразить список смежности для 1 и 2 матрицы

Сначала нам дают выбор: какой граф вывести на экран. Выведем сначала 1, а потом 2.

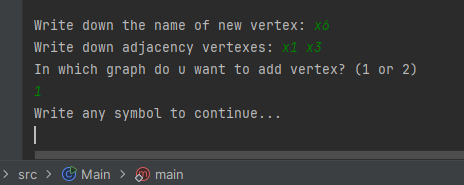




Если посмотреть на матрицы, то списки смежности действительно им соответствуют.

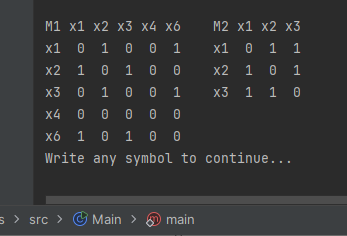
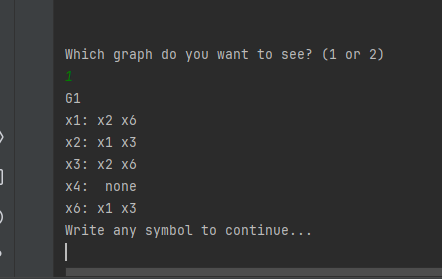
**Задание 2**

1. C помощью меню можно добавить в матрицу и одновременно в список смежности 1 вершину. Для этого выберем 3 пункт из нашего меню.



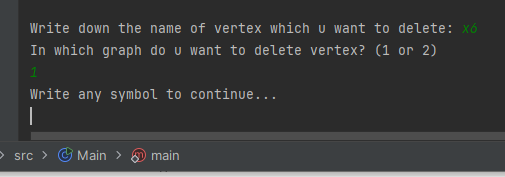
Сначала нам предлагают ввести название новой вершины(оно может быть любым). Далее предлагают ввести смежные вершины и в какой граф необходимо добавить вершину.

1. Проверим матрицу и 1 список, чтобы удостовериться в правильности добавления:

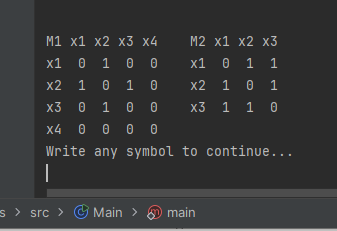
 

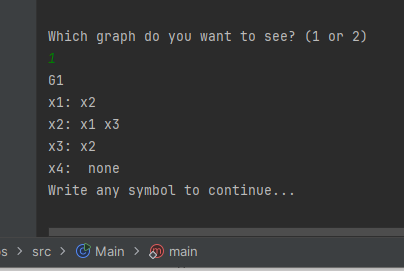
Как можем заметить вершина верно добавляется в граф.

1. Также программа позволяет удалять вершины (пункт меню 4). Удалим только что добавленную вершину.



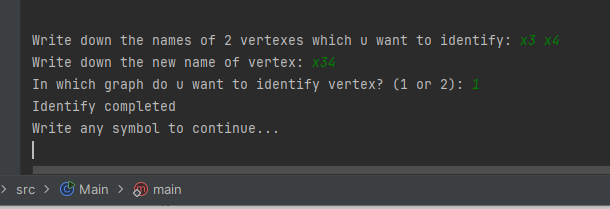
Проверим:



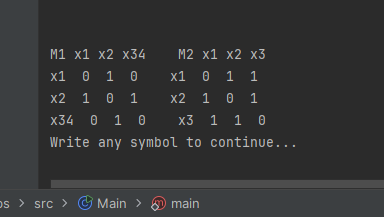


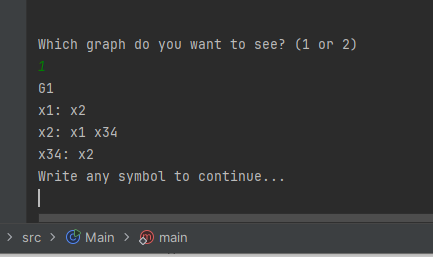
Вершина удалена!

1. Для выполнения задания необходимо было добавить эти методы добавления и удаления. Теперь можно приступать к отождествлению вершины (пункт меню 5). Чтобы объединить 2 вершины нужно сначала ввести названия 2х вершин, затем новое название вершины и выбрать граф:



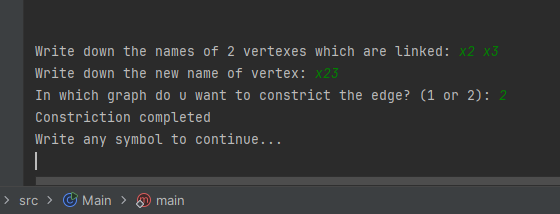
Посмотрим, что получилось

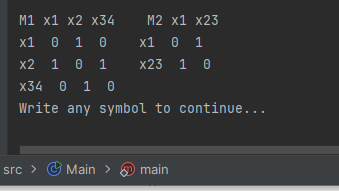


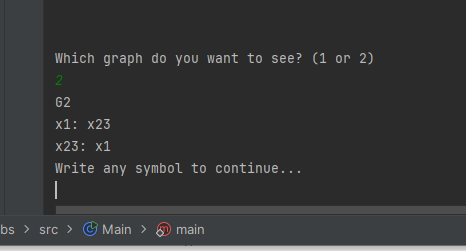


Так как раньше x3 соединялась с x2, то теперь новая вершина x34 соединяется с x2, следовательно x2 соединяется с x34.

1. Также программа позволяет осуществить стягивание ребра (т.е. возможность объединить только те вершины, которые смежные друг другу). Выберем пункт 6 из нашего меню и объединим x2 и x3 образовав новую вершину x23 в 2 графе.



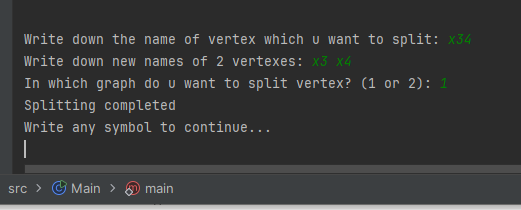




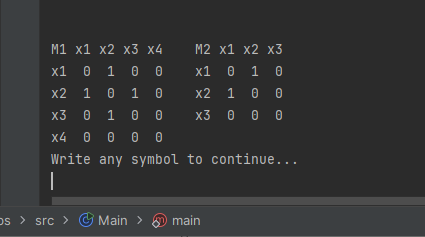
Так как любая из вершин x2 и x3 соединялась с х1, следовательно новая стала смежной х1.

1. Теперь нам необходимо разделить 1 вершину на 2. Для этого воспользуемся 7 пунктом меню:

Разъединим только что объединенные вершины в 1 и 2 графах



Сделаем то же самое для x23 в 2 графе и посмотрим на матрицы:



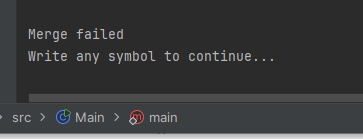
По правилу расщепления вершин: новые вершины разделяют между собой смежные вершины старой, следовательно x3 теряет ребро с x1, т.к x2 получило его от x23.

**Задание 3**

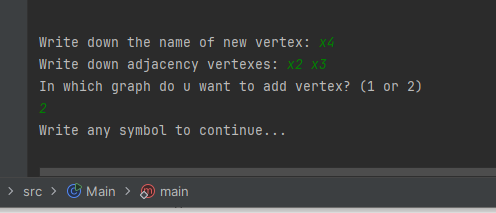
Поработаем над матрицами:

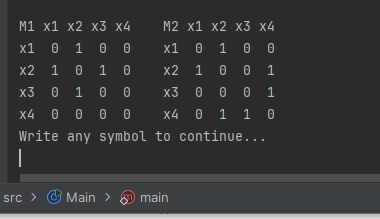
1. Реализовано объединение матриц:

Для этого выберем пункт 9.

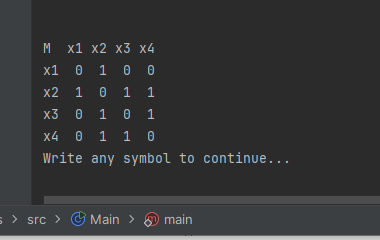


Объединение не удалось. Все потому, что у нас матрицы разных размеров или их вершины разные. Добавим вершину x4 во 2 граф, чтобы осуществить объединение



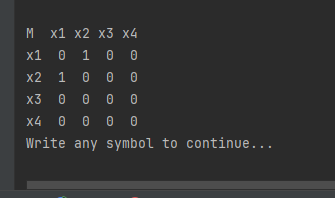


Теперь матрицы одинакового размера и с одинаковыми вершинами. Выполним объединение:

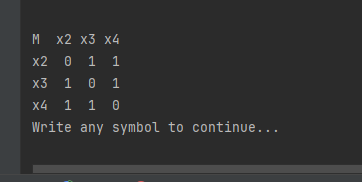


Все верно!

1. Теперь попробуем выполнить пересечение матриц (пункт меню 10)



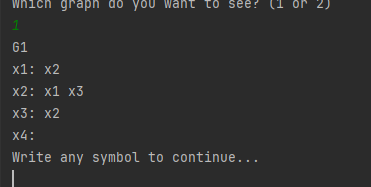
1. И наконец кольцевая сумма:

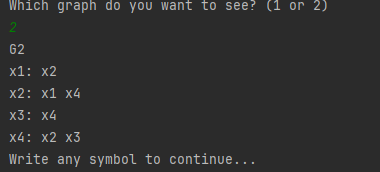


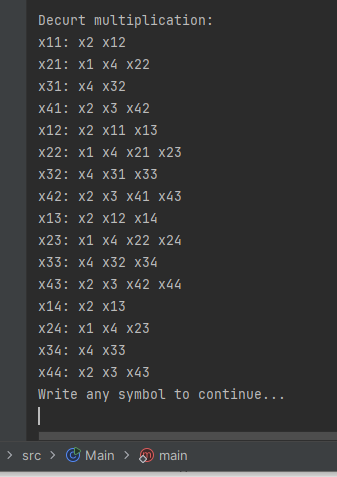
Так как новая матрица стала обладать изолированную вершину, то вершину не показывают.

**Задание 4**

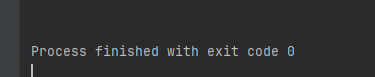
1. Выполним декартовое произведение графов (пункт меню 8):







1. Теперь, закончив работу с программой, можем выйти из нее, выбрав 12 пункт меню:



**Листинг:**

**Main.java**

import java.io.\*;  
import java.util.Scanner;  
  
public class Main{  
  
 static Scanner *sc* = new Scanner(System.*in*);  
 public static void clearConsole(){  
 for(int i=0;i<100;i++){  
 System.*out*.print("\n\n");  
 }  
 }  
 public static void main(String[] args) {  
 System.*out*.println("To start program write down size of 1st and 2nd matrixes: ");  
 int n=*sc*.nextInt();  
 Matrix M1 = new Matrix(n);  
 int b=*sc*.nextInt();  
 Matrix M2 = new Matrix(b);  
 Vertex G1 = new Vertex("start",null,null,M1);  
 Vertex G2 = new Vertex("start",null,null,M2);  
  
 while(true) {  
 System.*out*.println("------------MENU------------");  
 System.*out*.println("1. Show matrixes");  
 System.*out*.println("2. Show graph");  
 System.*out*.println("3. Add vertex");  
 System.*out*.println("4. Delete vertex");  
 System.*out*.println("5. Identify vertexes");  
 System.*out*.println("6. Constrict edge");  
 System.*out*.println("7. Split vertex");  
 System.*out*.println("8. Decurt multiplication of graphs");  
 System.*out*.println("9. Merge matrixes");  
 System.*out*.println("10. Intersection of matrixes");  
 System.*out*.println("11. Ring sum of matrixes");  
 System.*out*.println("12. Exit");  
 int choice = *sc*.nextInt();  
 *clearConsole*();  
 switch (choice) {  
 case 1:  
 Matrix.*printMatrixes*(M1, M2, G1, G2);  
 break;  
 case 2:  
 System.*out*.println("Which graph do you want to see? (1 or 2)");  
 int choice2 = *sc*.nextInt();  
 switch (choice2) {  
 case 1:  
 System.*out*.println("G1");  
 Vertex.*printGraph*(G1);  
 break;  
 case 2:  
 System.*out*.println("G2");  
 Vertex.*printGraph*(G2);  
 break;  
 default:  
 break;  
 }  
 break;  
 case 3:  
 System.*out*.print("Write down the name of new vertex: ");  
 String newName = *sc*.next();  
 System.*out*.print("Write down adjacency vertexes: ");  
 String adjList = *sc*.nextLine();  
 adjList = *sc*.nextLine();  
 System.*out*.println("In which graph do u want to add vertex? (1 or 2)");  
 int choice3 = *sc*.nextInt();  
 switch (choice3) {  
 case 1:  
 if(Vertex.*addVertex*(G1, newName, adjList)) Matrix.*addVertInMx*(M1,G1);  
 break;  
 case 2:  
 if(Vertex.*addVertex*(G2, newName, adjList)) Matrix.*addVertInMx*(M2,G2);  
 break;  
 default:  
 break;  
  
 }  
 break;  
 case 4:  
 System.*out*.print("Write down the name of vertex which u want to delete: ");  
 String newName2 = *sc*.next();  
 System.*out*.println("In which graph do u want to delete vertex? (1 or 2)");  
 int choice4 = *sc*.nextInt();  
 switch (choice4) {  
 case 1:  
 Vertex.*deleteVertex*(G1, newName2, M1);  
 break;  
 case 2:  
 Vertex.*deleteVertex*(G2, newName2, M2);  
 break;  
 default:  
 break;  
  
 }  
 break;  
 case 5:  
 System.*out*.print("Write down the names of 2 vertexes which u want to identify: ");  
 String oldName1 = *sc*.next();  
 String oldName2 = *sc*.next();  
 System.*out*.print("Write down the new name of vertex: ");  
 String newName3 = *sc*.next();  
 System.*out*.print("In which graph do u want to identify vertex? (1 or 2): ");  
 int choice5 = *sc*.nextInt();  
 switch (choice5) {  
 case 1:  
 if (Vertex.*identifyVertexes*(newName3, oldName1, oldName2, M1, G1))  
 System.*out*.println("Identify completed");  
 else System.*out*.println("Identify failed");  
 break;  
 case 2:  
 if (Vertex.*identifyVertexes*(newName3, oldName1, oldName2, M2, G2))  
 System.*out*.println("Identify completed");  
 else System.*out*.println("Identify failed");  
 break;  
 default:  
 break;  
 }  
 break;  
 case 6:  
 System.*out*.print("Write down the names of 2 vertexes which are linked: ");  
 String oldName3 = *sc*.next();  
 String oldName4 = *sc*.next();  
 System.*out*.print("Write down the new name of vertex: ");  
 String newName4 = *sc*.next();  
 System.*out*.print("In which graph do u want to constrict the edge? (1 or 2): ");  
 int choice6 = *sc*.nextInt();  
 switch (choice6) {  
 case 1:  
 if (Vertex.*constrictionEdge*(newName4, oldName3, oldName4, M1, G1))  
 System.*out*.println("Constriction completed");  
 else System.*out*.println("Constriction failed");  
 break;  
 case 2:  
 if (Vertex.*constrictionEdge*(newName4, oldName3, oldName4, M2, G2))  
 System.*out*.println("Constriction completed");  
 else System.*out*.println("Constriction failed");  
 break;  
 default:  
 break;  
 }  
 break;  
 case 7:  
 System.*out*.print("Write down the name of vertex which u want to split: ");  
 String oldName5 = *sc*.next();  
 System.*out*.print("Write down new names of 2 vertexes: ");  
 String newName5 = *sc*.next();  
 String newName6 = *sc*.next();  
 System.*out*.print("In which graph do u want to split vertex? (1 or 2): ");  
 int choice7 = *sc*.nextInt();  
 switch (choice7) {  
 case 1:  
 if (Vertex.*splittingVertex*(G1, M1, newName5, newName6, oldName5))  
 System.*out*.println("Splitting completed");  
 else System.*out*.println("Splitting failed");  
 break;  
 case 2:  
 if (Vertex.*splittingVertex*(G2, M2, newName5, newName6, oldName5))  
 System.*out*.println("Splitting completed");  
 else System.*out*.println("Splitting failed");  
 break;  
 default:  
 break;  
 }  
 break;  
 case 8:  
 Vertex decNew = Vertex.*decMulti*(G1, G2);  
 if (decNew == null) {  
 System.*out*.println("Decurt failed");  
 } else {  
 System.*out*.println("Decurt multiplication:");  
 Vertex.*printGraph*(decNew);  
 }  
 break;  
 case 9:  
 Matrix newM = Matrix.*mergeMatrixes*(M1, M2, G1, G2);  
 if (newM == null)  
 System.*out*.println("Merge failed");  
 else Matrix.*printMatrix*(newM, G1);  
 break;  
 case 10:  
 Matrix intM = Matrix.*intersectionMatrixes*(M1,M2,G1,G2);  
 if (intM == null)  
 System.*out*.println("Intersection failed");  
 else Matrix.*printMatrix*(intM, G1);  
 break;  
 case 11:  
  
 Matrix ringM = Matrix.*ringSumMatrixes*(M1,M2,G1,G2);  
 if(ringM == null)  
 System.*out*.println("Ring sum failed");  
 break;  
 case 12:  
 Runtime.*getRuntime*().exit(0);  
 break;  
 default:  
 break;  
 }  
  
 System.*out*.println("Write any symbol to continue...");  
 String tmp = *sc*.next();  
 *clearConsole*();  
 }  
 }  
}

**Vertex.java**

import java.util.Arrays;  
  
public class Vertex {  
 private String vertex;  
 private Vertex next;  
 private Vertex[] adjVertexes;  
   
 public Vertex(String vertex, Vertex next, Vertex[] adjVertexes, Matrix M) {  
 this.vertex = vertex;  
 this.next = next;  
 this.adjVertexes = adjVertexes;  
 if(vertex.equals("start") && M!=null)  
 fillGraph(M);  
 }  
 private static Vertex findVertex(Vertex G,String name){  
 Vertex tmp = G;  
 while(tmp != null){  
 if(tmp.getVertex().equals(name)) return tmp;  
 tmp = tmp.getNext();  
 }  
 return null;  
 }  
private void fillGraph(Matrix M){  
 int[][] arr = M.getMatrix();  
 Vertex[] vert = new Vertex[arr.length+1];  
 vert[0]=this;  
 for(int i=1;i<arr.length+1;i++){  
 vert[i] = new Vertex("x"+i,null,null,null);  
 }  
 for(int i=0;i<arr.length;i++){  
 vert[i].setNext(vert[i+1]);  
 }  
 for(int i = 0; i < arr.length; i++){  
 for(int j = 0; j < arr.length; j++){  
 if(arr[i][j]==1){  
 int k=1;  
 if(vert[i+1].adjVertexes!=null)  
 k=1+vert[i+1].adjVertexes.length;  
 Vertex[] tmp = new Vertex[k];  
 if(vert[i+1].adjVertexes!=null)  
 tmp= Arrays.*copyOf*(vert[i+1].adjVertexes, k);  
 tmp[tmp.length-1]=vert[j+1];  
 vert[i+1].setAdjVertexes(tmp);  
 }  
 }  
 }  
}  
 public static void printGraph(Vertex graph){  
 Vertex temp=graph;  
 temp=temp.next;  
 while(temp!=null){  
 System.*out*.print(temp.vertex+": ");  
 if(temp.adjVertexes!=null)  
 Arrays.*stream*(temp.adjVertexes).forEach(n -> System.*out*.print(n.vertex+" "));  
 else{  
 System.*out*.print(" none ");  
 }  
 System.*out*.println();  
 temp = temp.next;  
 }  
 }  
  
 public static Vertex deleteVertex(Vertex graph,String vertex,Matrix M){  
 Vertex temp=graph;  
 temp = temp.next;  
 Vertex last=null;  
 int count = 1;  
 while(temp!=null){  
 if (temp.vertex.equals(vertex)) {  
 if (last != null)  
 last.next = temp.next;  
 else graph.next = temp.next;  
 Matrix.*deleteVertInMx*(M,count);  
 }  
 else if(temp.adjVertexes!=null) {  
 for (int i = 0; i < temp.adjVertexes.length; i++) {  
 if (temp.adjVertexes[i].vertex.equals(vertex)) {  
 Vertex[] tmpVertexes;  
 if(temp.adjVertexes.length - 1!=0){  
 tmpVertexes = new Vertex[temp.adjVertexes.length - 1];  
 int k = 0;  
 for (int j = 0; j < temp.adjVertexes.length; j++) {  
 if (i != j) {  
 tmpVertexes[k] = temp.adjVertexes[j];  
 k++;  
 }  
 }  
 temp.setAdjVertexes(tmpVertexes);  
 }  
 else temp.setAdjVertexes(null);  
 break;  
 }  
  
 }  
 }  
 if(graph!=temp.next)  
 last=temp;  
 temp=temp.next;  
 count++;  
 }  
 return graph;  
 }  
 public static boolean addVertex(Vertex G, String name,String adjVert){  
 if(*findVertex*(G,name)!=null) return false;  
 Vertex temp=G;  
 String[] adj = adjVert.trim().split(" ");  
 Vertex[] tmp = new Vertex[adj.length];  
 Vertex newVert = new Vertex(name,null,null,null);  
 Vertex last=null;  
 while(temp!=null){  
 for(int i=0;i<adj.length;i++){  
 if(temp.vertex.equals(adj[i])){  
 //старая  
 int k=1;  
 if(temp.adjVertexes!=null)  
 k=1+temp.adjVertexes.length; // размер массива смежных вершин  
 Vertex[] tmp2 = new Vertex[k];  
 if(temp.adjVertexes!=null)  
 tmp2= Arrays.*copyOf*(temp.adjVertexes, k);  
 tmp2[tmp2.length-1]=newVert;  
 temp.setAdjVertexes(tmp2);  
  
 //новая  
 k=1;  
 if(newVert.adjVertexes!=null)  
 k=1+newVert.adjVertexes.length; // размер массива смежных вершин  
 tmp2 = new Vertex[k];  
 if(newVert.adjVertexes!=null)  
 tmp2= Arrays.*copyOf*(newVert.adjVertexes, k);  
 tmp2[tmp2.length-1]=temp;  
 newVert.setAdjVertexes(tmp2);  
 }  
 }  
 if(temp.next==null) last=temp;  
 temp=temp.next;  
 }  
 last.next=newVert;  
 return true;  
 }  
 public static Vertex addVertex(Vertex G, String name,Vertex adjVert[]){  
 Vertex tmp = G;  
 while(tmp.next!=null){  
 tmp=tmp.next;  
 }  
 Vertex newVert = new Vertex(name,null,adjVert,null);  
 tmp.setNext(newVert);  
 return newVert;  
 }  
 public static boolean identifyVertexes(String newName,String vert1,String vert2, Matrix M, Vertex G){  
 if(*findVertex*(G,vert1)==null || *findVertex*(G,vert2)==null){  
 return false;  
 }  
  
 Vertex[] tmp1 = *findVertex*(G,vert1).adjVertexes;  
 Vertex[] tmp2 = *findVertex*(G,vert2).adjVertexes;  
 Vertex[] newAdjVert=new Vertex[M.getSize()];  
 Vertex nwVert = *addVertex*(G,newName,newAdjVert);  
 int k=M.getSize();  
 if(tmp1!=null) {  
 newAdjVert= Arrays.*copyOf*(tmp1, k);  
 k-=tmp1.length;  
 if(tmp2!=null){  
 for (int i = 0; i < tmp2.length; i++) {  
 for (int j = 0; j < newAdjVert.length; j++) {  
 if(newAdjVert[j]!=null && (newAdjVert[j].vertex.equals(vert1) || newAdjVert[j].vertex.equals(vert2))){  
 if(Arrays.*stream*(newAdjVert).noneMatch(n-> n == nwVert)){  
 newAdjVert[j]=nwVert;  
 }  
 }  
 if ((!(tmp2[i] == newAdjVert[j]) && newAdjVert[j] == null) && !(tmp2[i].vertex.equals(vert1) || tmp2[i].vertex.equals(vert2))) {  
 newAdjVert[j] = tmp2[i];  
 k--;  
 break;  
 } else if(tmp2[i] == newAdjVert[j]) j = M.getSize();  
 }  
 }  
 }  
 }  
 else if(tmp2!=null) {  
 newAdjVert = Arrays.*copyOf*(tmp2, tmp2.length);  
 for (int j = 0; j < newAdjVert.length; j++) {  
 if (newAdjVert[j].vertex.equals(vert1) || newAdjVert[j].vertex.equals(vert2)) {  
 newAdjVert[j] = nwVert;  
 }  
 }  
 k-=tmp2.length;  
 }  
 Vertex[] finAdjVert;  
 if(k!=0){  
 finAdjVert=new Vertex[M.getSize()-k];  
 finAdjVert= Arrays.*copyOf*(newAdjVert, M.getSize()-k);  
 }  
 else{  
 finAdjVert= Arrays.*copyOf*(newAdjVert, newAdjVert.length);  
 }  
 nwVert.adjVertexes=finAdjVert;  
 *deleteVertex*(G,vert1,M);  
 *deleteVertex*(G,vert2,M);  
 for (int i = 0; i < finAdjVert.length; i++) {  
 if(!finAdjVert[i].vertex.equals(newName)) {  
 int size = 1;  
 Vertex[] newArrVert = new Vertex[size];  
 if (finAdjVert[i].adjVertexes != null) {  
 size += finAdjVert[i].adjVertexes.length;  
 newArrVert = new Vertex[size];  
 newArrVert = Arrays.*copyOf*(finAdjVert[i].adjVertexes, size);  
 }  
 newArrVert[size - 1] = nwVert;  
 finAdjVert[i].setAdjVertexes(newArrVert);  
 }  
 }  
  
 Matrix.*addVertInMx*(M,G);  
 return true;  
 }  
  
 public static boolean constrictionEdge(String newName,String vert1,String vert2, Matrix M, Vertex G){  
 Vertex v1 = *findVertex*(G,vert1);  
 Vertex v2 = *findVertex*(G,vert2);  
 if(v1==null || v2==null){  
 return false;  
 }  
 Vertex[] tmp1 = v1.adjVertexes;  
 Vertex[] tmp2 = v2.adjVertexes;  
 if(tmp1==null || Arrays.*stream*(tmp1).noneMatch(n-> n==v2)){  
 return false;  
 }  
  
 Vertex[] newAdjVert=new Vertex[M.getSize()];  
 Vertex nwVert = *addVertex*(G,newName,newAdjVert);  
 int k=M.getSize();  
 if(tmp1!=null) {  
 newAdjVert= Arrays.*copyOf*(tmp1, k);  
 k-=tmp1.length;  
 if(tmp2!=null){  
 for (int i = 0; i < tmp2.length; i++) {  
 for (int j = 0; j < newAdjVert.length; j++) {  
 if(newAdjVert[j]!=null && (newAdjVert[j].vertex.equals(vert1) || newAdjVert[j].vertex.equals(vert2))){  
 if(Arrays.*stream*(newAdjVert).noneMatch(n-> n == nwVert)){  
 newAdjVert[j]=nwVert;  
 }  
 }  
 if ((!(tmp2[i] == newAdjVert[j]) && newAdjVert[j] == null) && !(tmp2[i].vertex.equals(vert1) || tmp2[i].vertex.equals(vert2))) {  
 newAdjVert[j] = tmp2[i];  
 k--;  
 break;  
 } else if(tmp2[i] == newAdjVert[j]) j = M.getSize();  
 }  
 }  
 }  
 }  
 else if(tmp2!=null) {  
 newAdjVert = Arrays.*copyOf*(tmp2, tmp2.length);  
 for (int j = 0; j < newAdjVert.length; j++) {  
 if (newAdjVert[j].vertex.equals(vert1) || newAdjVert[j].vertex.equals(vert2)) {  
 newAdjVert[j] = nwVert;  
 }  
 }  
 k-=tmp2.length;  
 }  
 if(Arrays.*stream*(newAdjVert).anyMatch(n->n.vertex.equals(newName))){  
 Vertex[] tmp = new Vertex[newAdjVert.length-1];  
 if(tmp.length!=0) {  
 int o = 0;  
 for (int i = 0; i < newAdjVert.length; i++) {  
 if (newAdjVert[i]!=null && !newAdjVert[i].vertex.equals(newName)) {  
 tmp[o] = newAdjVert[i];  
 o++;  
 }  
 }  
 newAdjVert = new Vertex[o];  
 newAdjVert = Arrays.*copyOf*(tmp, o);  
 }  
 else{  
 newAdjVert = new Vertex[0];  
 }  
 k++;  
 }  
 Vertex[] finAdjVert;  
 if(k!=0){  
 finAdjVert=new Vertex[M.getSize()-k];  
 finAdjVert= Arrays.*copyOf*(newAdjVert, M.getSize()-k);  
 }  
 else{  
 finAdjVert= Arrays.*copyOf*(newAdjVert, newAdjVert.length);  
 }  
 nwVert.adjVertexes=finAdjVert;  
 *deleteVertex*(G,vert1,M);  
 *deleteVertex*(G,vert2,M);  
 for (int i = 0; i < finAdjVert.length; i++) {  
 int size = 1;  
 Vertex[] newArrVert = new Vertex[size];  
 if (finAdjVert[i].adjVertexes != null) {  
 size += finAdjVert[i].adjVertexes.length;  
 newArrVert = new Vertex[size];  
 newArrVert = Arrays.*copyOf*(finAdjVert[i].adjVertexes, size);  
 }  
 newArrVert[size - 1] = nwVert;  
 finAdjVert[i].setAdjVertexes(newArrVert);  
 }  
  
 Matrix.*addVertInMx*(M,G);  
 return true;  
 }  
 public static boolean splittingVertex(Vertex G, Matrix M, String name1, String name2, String oldName){  
 Vertex tmp = G;  
 while(tmp!=null){  
 if(tmp.vertex.equals(oldName)){  
 break;  
 }  
 tmp=tmp.next;  
 }  
 if(tmp==null){return false;}  
 if(Arrays.*stream*(tmp.adjVertexes).anyMatch(n->n.vertex.equals(oldName))){  
 Vertex[] newTmpAdjVert = new Vertex[tmp.adjVertexes.length-1];  
 int k=0;  
 for (int i = 0; i < tmp.adjVertexes.length; i++) {  
 if(!tmp.adjVertexes[i].vertex.equals(oldName)){  
 newTmpAdjVert[k]=tmp.adjVertexes[i];  
 k++;  
 }  
 }  
 tmp.adjVertexes=newTmpAdjVert;  
 }  
 Vertex[] adj1 = new Vertex[tmp.adjVertexes.length-tmp.adjVertexes.length/2];  
 Vertex[] adj2 = new Vertex[tmp.adjVertexes.length/2];  
 if(tmp.adjVertexes.length!=0){  
 int k=0;  
 for(int i=0;i<adj1.length;i++){  
 adj1[i]=tmp.adjVertexes[k];  
 k++;  
 }  
 for(int i=0;i<adj2.length;i++){  
 adj2[i]=tmp.adjVertexes[k];  
 k++;  
 }  
 }  
  
 Vertex newVert1 = Vertex.*addVertex*(G,name1,adj1);  
 Matrix.*addVertInMx*(M,G);  
 Vertex newVert2 = *addVertex*(G,name2,adj2);  
 Matrix.*addVertInMx*(M,G);  
  
 if(newVert1.adjVertexes!=null) {  
 for (int i = 0;i<newVert1.adjVertexes.length;i++){  
 for(int j=0;j<newVert1.adjVertexes[i].adjVertexes.length;j++){  
 if(newVert1.adjVertexes[i].adjVertexes[j].vertex.equals(oldName)){  
 newVert1.adjVertexes[i].adjVertexes[j] = newVert1;  
 }  
 }  
 }  
 }  
 if(newVert2.adjVertexes!=null) {  
 for (int i = 0;i<newVert2.adjVertexes.length;i++){  
 for(int j=0;j<newVert2.adjVertexes[i].adjVertexes.length;j++){  
 if(newVert2.adjVertexes[i].adjVertexes[j].vertex.equals(oldName)){  
 newVert2.adjVertexes[i].adjVertexes[j] = newVert2;  
 }  
 }  
 }  
 }  
 *deleteVertex*(G,oldName,M);  
 return true;  
 }  
  
 public static Vertex decMulti(Vertex G1,Vertex G2){  
 if(G1.next==null || G2.next==null){return null;}  
 int count1=0;  
 Vertex tmp = G1.next;  
 while(tmp!=null){  
 count1++;  
 tmp=tmp.next;  
 }  
 tmp=G2.next;  
 int count2=0;  
 while(tmp!=null){  
 count2++;  
 tmp=tmp.next;  
 }  
  
 Vertex newVert=new Vertex("start",null,null,null);  
 Vertex tmp2=newVert;  
 Vertex last = null;  
 for(int i=0;i<Math.*min*(count1,count2);i++) {  
 if (count1 > count2) tmp = G1.next;  
 else tmp = G2.next;  
  
 while (tmp != null) {  
 Vertex[] newAdj=tmp.adjVertexes;  
 if(last!=null) {  
 int k=1;  
 if(newAdj!=null)  
 k+=newAdj.length;  
 newAdj = new Vertex[k];  
 if(tmp.adjVertexes!=null) newAdj = Arrays.*copyOf*(tmp.adjVertexes, k);  
 newAdj[k-1] = last;  
 }  
 Vertex temp = Vertex.*addVertex*(newVert, tmp.vertex+(i+1), newAdj);  
 if(last!=null){  
 int k=1;  
 if(last.adjVertexes!=null)  
 k+=last.adjVertexes.length;  
 newAdj = new Vertex[k];  
 if(last.adjVertexes!=null) newAdj = Arrays.*copyOf*(last.adjVertexes, k);  
 newAdj[k-1] = temp;  
 last.adjVertexes = newAdj;  
 last=last.next;  
 }  
 tmp = tmp.next;  
 }  
 tmp2=tmp2.next;  
 last=tmp2;  
 while(tmp2.next!=null){  
 tmp2=tmp2.next;  
 }  
 }  
 return newVert;  
 }  
  
 public String getVertex() {  
 return vertex;  
 }  
 public Vertex[] getAdjVertexes() {  
 return adjVertexes;  
 }  
 public Vertex getNext() {  
 return next;  
 }  
 public void setAdjVertexes(Vertex[] adjVertexes) {  
 this.adjVertexes = adjVertexes;  
 }  
 public void setVertex(String vertex) {  
 this.vertex = vertex;  
 }  
 public void setNext(Vertex next) {  
 this.next = next;  
 }  
}

**Matrix.java**

import java.util.Arrays;  
  
public class Matrix {  
 private int[][] matrix;  
 private int size;  
 public Matrix(int size){  
 this.size = size;  
 matrix = new int[size][size];  
 createMatrix();  
 }  
  
 public void createMatrix(){  
 for (int i = 0; i < this.matrix.length; i++) {  
 for (int j = i; j < this.matrix.length; j++) {  
 this.matrix[i][j] = (int)(Math.*random*()\*2);  
 this.matrix[j][i] = this.matrix[i][j];  
 if(j==i) this.matrix[i][i]=0;  
 }  
 }  
 }  
 public static void printMatrixes(Matrix M1, Matrix M2,Vertex G1, Vertex G2){  
 if((M1==null && M2==null) || (M1.getMatrix().length==0 && M2.getMatrix().length==0)){  
 System.*out*.println("Matrixes are empty");  
 return;  
 }  
 System.*out*.print("M1 ");  
 Vertex tmp1=G1;  
 tmp1=tmp1.getNext();  
 while(tmp1!=null){  
 System.*out*.print(tmp1.getVertex()+" ");  
 tmp1=tmp1.getNext();  
 }  
 System.*out*.print(" M2 ");  
 tmp1=G2;  
 tmp1=tmp1.getNext();  
 while(tmp1!=null){  
 System.*out*.print(tmp1.getVertex()+" ");  
 tmp1=tmp1.getNext();  
 }  
 tmp1=G1;  
 tmp1=tmp1.getNext();  
 Vertex tmp2 =G2;  
 tmp2=tmp2.getNext();  
 for (int i = 0; i < Math.*max*(M1.getMatrix().length,M2.getMatrix().length); i++) {  
 if(i<M1.getMatrix().length){  
 System.*out*.print("\n"+tmp1.getVertex()+" ");  
 tmp1=tmp1.getNext();  
 for (int j = 0; j < M1.getMatrix().length; j++) {  
 System.*out*.print(M1.getMatrix()[i][j]+" ");  
 }  
 }  
 else{  
 System.*out*.print("\n ");  
 for (int j = 0; j < M2.getMatrix().length; j++) {  
 System.*out*.print(" ");  
 }  
 }  
 if(i<M2.getMatrix().length) {  
 System.*out*.print(" "+tmp2.getVertex()+" ");  
 tmp2=tmp2.getNext();  
 for (int j = 0; j < M2.getMatrix().length; j++) {  
 System.*out*.print(M2.getMatrix()[i][j] + " ");  
 }  
 }  
 }  
 System.*out*.println();  
 }  
  
 public static void printMatrix(Matrix M, Vertex G){  
 if(M==null || M.size<=0){  
 System.*out*.println("Matrix is empty");  
 return;  
 }  
 System.*out*.print("M ");  
 Vertex tmp1=G;  
 tmp1=tmp1.getNext();  
 while(tmp1!=null){  
 System.*out*.print(tmp1.getVertex()+" ");  
 tmp1=tmp1.getNext();  
 }  
 tmp1=G;  
 tmp1=tmp1.getNext();  
 for (int i = 0; i < M.size; i++) {  
 System.*out*.print("\n"+tmp1.getVertex()+" ");  
 tmp1=tmp1.getNext();  
 for (int j = 0; j < M.size; j++) {  
 System.*out*.print(M.matrix[i][j]+" ");  
 }  
 }  
 System.*out*.println();  
 }  
  
 public static void printMatrix(Matrix M, Vertex G,String[] exVert){  
 if(M==null || M.size<=0){  
 System.*out*.println("Matrix is empty");  
 return;  
 }  
 System.*out*.print("M ");  
 Vertex tmp1=G;  
 tmp1=tmp1.getNext();  
 while(tmp1!=null){  
 final String finStr = tmp1.getVertex();  
 if(exVert==null || Arrays.*stream*(exVert).noneMatch(n->n.equals(finStr))) {  
 System.*out*.print(tmp1.getVertex() + " ");  
 }  
 tmp1 = tmp1.getNext();  
 }  
 tmp1=G;  
 tmp1=tmp1.getNext();  
 for (int i = 0; i < M.size; i++) {  
 final String finStr = tmp1.getVertex();  
 if(exVert==null || Arrays.*stream*(exVert).noneMatch(n->n.equals(finStr))) {  
 System.*out*.print("\n" + tmp1.getVertex() + " ");  
 Vertex tmp2=G;  
 tmp2=tmp2.getNext();  
 for (int j = 0; j < M.size; j++) {  
 final String finStr2 = tmp2.getVertex();  
 if(exVert==null || Arrays.*stream*(exVert).noneMatch(n->n.equals(finStr2)))  
 System.*out*.print(M.matrix[i][j] + " ");  
 tmp2=tmp2.getNext();  
 }  
 }  
 tmp1 = tmp1.getNext();  
 }  
 System.*out*.println();  
 }  
  
 public static void deleteVertInMx(Matrix M,int value){  
 int[][] tmp = new int[M.getMatrix().length-1][M.getMatrix()[0].length-1];  
 int x=0;  
 for (int i = 0; i < M.getMatrix().length; i++) {  
 if(i!=value-1) {  
 int y=0;  
 for (int j = 0; j < M.getMatrix()[0].length; j++) {  
 if (j != value - 1) {  
 tmp[x][y] = M.getMatrix()[i][j];  
 y++;  
 }  
 }  
 x++;  
 }  
 }  
 M.size--;  
 M.setMatrix(tmp);  
 }  
  
 public static void addVertInMx(Matrix M, Vertex G){  
 Vertex tmp=G;  
 while(tmp.getNext()!=null){  
 tmp=tmp.getNext();  
 }  
  
 int k=1;  
 if(M.size!=0)  
 k=1+M.size; // размер массива смежных вершин  
 int[][] tmp2 = new int[k][k];  
 if(M.size!=0) {  
 Vertex nxt = G;  
 nxt=nxt.getNext();  
 for (int i = 0; i < k; i++) {  
 final Vertex forCheck = nxt;  
 if(i!=k-1) {  
 for (int j = 0; j < k; j++) {  
 if (j != k - 1) {  
 tmp2[i][j] = M.getMatrix()[i][j];  
 } else {  
 if(tmp.getAdjVertexes()!=null)  
 tmp2[i][j] = Arrays.*stream*(tmp.getAdjVertexes()).anyMatch(n -> n.getVertex().equals(forCheck.getVertex())) ? 1 : 0;  
 else tmp2[i][j]=0;  
 }  
 }  
 }  
 else{  
 Vertex nxt2 = G;  
 nxt2=nxt2.getNext();  
 for (int j = 0; j < k; j++) {  
 final Vertex forCheck2 = nxt2;  
 if(tmp.getAdjVertexes()!=null)  
 tmp2[i][j] = Arrays.*stream*(tmp.getAdjVertexes()).anyMatch(n -> n.getVertex().equals(forCheck2.getVertex())) ? 1 : 0;  
 else tmp2[i][j]=0;  
 nxt2=nxt2.getNext();  
 }  
 }  
 nxt = nxt.getNext();  
 }  
 }  
 M.size++;  
 M.matrix = tmp2;  
 }  
 public static Matrix mergeMatrixes(Matrix M1,Matrix M2,Vertex G1,Vertex G2){  
 if(M1.size!=M2.size) return null;  
 Vertex tmp1=G1;  
 int k=0;  
 for(int i=0;i<M1.size;i++){  
 Vertex tmp2=G2;  
 for (int j = 0; j < M1.size; j++) {  
 if(tmp1.getVertex().equals(tmp2.getVertex())){  
 k++;  
 }  
 tmp2=tmp2.getNext();  
 }  
 tmp1=tmp1.getNext();  
 }  
 if(k!=M1.size) return null;  
 Matrix newM = new Matrix(Math.*max*(M1.size,M2.size));  
 for (int i = 0; i < M1.size; i++) {  
 for (int j = 0; j < M1.size; j++) {  
 newM.matrix[i][j]=M1.matrix[i][j] | M2.matrix[i][j];  
 }  
 }  
 return newM;  
 }  
 public static Matrix intersectionMatrixes(Matrix M1,Matrix M2,Vertex G1,Vertex G2){  
 if(M1.size!=M2.size) return null;  
 Vertex tmp1=G1;  
 int k=0;  
 for(int i=0;i<M1.size;i++){  
 Vertex tmp2=G2;  
 for (int j = 0; j < M1.size; j++) {  
 if(tmp1.getVertex().equals(tmp2.getVertex())){  
 k++;  
 }  
 tmp2=tmp2.getNext();  
 }  
 tmp1=tmp1.getNext();  
 }  
 if(k!=M1.size) return null;  
 Matrix newM = new Matrix(Math.*max*(M1.size,M2.size));  
 for (int i = 0; i < M1.size; i++) {  
 for (int j = 0; j < M1.size; j++) {  
 newM.matrix[i][j]=M1.matrix[i][j] & M2.matrix[i][j];  
 }  
 }  
 return newM;  
 }  
  
 public static Matrix ringSumMatrixes(Matrix M1,Matrix M2,Vertex G1,Vertex G2){  
 if(M1.size!=M2.size) return null;  
 Vertex tmp1=G1;  
 Vertex tmp2=G2;  
 for(int i=0;i<M1.size;i++){  
 if(!tmp1.getVertex().equals(tmp2.getVertex())) return null;  
 tmp2=tmp2.getNext();  
 tmp1=tmp1.getNext();  
 }  
 Matrix newM = new Matrix(Math.*max*(M1.size,M2.size));  
 for (int i = 0; i < M1.size; i++) {  
 for (int j = 0; j < M1.size; j++) {  
 newM.matrix[i][j]=M1.matrix[i][j] ^ M2.matrix[i][j];  
 }  
 }  
 tmp1 = G1;  
 tmp1=tmp1.getNext();  
 String[] exVert=new String[newM.size];  
 int k=0;  
 for (int i = 0; i < newM.size; i++) {  
 int countZero=0;  
 for (int j = 0; j < newM.size; j++) {  
 if(newM.matrix[i][j]==0){  
 countZero++;  
 }  
 }  
 if(countZero==newM.size){  
 exVert[k]=tmp1.getVertex();  
 k++;  
 }  
 tmp1=tmp1.getNext();  
 }  
 int count=0;  
 for(int i=0;i<newM.size;i++){  
 if(exVert[i]==null){count++;}  
 }  
 String[] newExVert=new String[newM.size-count];  
 newExVert=Arrays.*copyOf*(exVert,newM.size-count);  
 *printMatrix*(newM,G1,newExVert);  
 return newM;  
 }  
 public int getSize(){  
 return size;  
 }  
 public void setSize(int size){  
 this.size = size;  
 }  
 public int[][] getMatrix(){  
 return matrix;  
 }  
 public void setMatrix(int[][] matrix){  
 this.matrix = matrix;  
 }  
}

**Вывод**

В ходе работы выполнения лабораторной работы было реализовано создание матриц и списков смежности графов. Также были реализованы основные операции над графами и матрицами, а также декартовое произведение графов.