Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №6

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Унарные и бинарные операции над графами»

Выполнили:

студенты группы 21ВВ1.2

Брянцев Артём  
Сущёв Максим  
Тюрин Владислав

Приняли:

Митрохин М. А.

Юрова О. В.

Пенза 2022

**Название**

Унарные и бинарные операции над графами.

**Лабораторное задание.**

**Задание 1.**

1. Сгенерируйте (используя генератор случайных чисел) две матрицы M1, М2 смежности неориентированных помеченных графов G1, G2. Выведите сгенерированные матрицы на экран.
2. \* Для указанных графов преобразуйте представление матриц смежности в списки смежности. Выведите полученные списки на экран.

**Задание 2.**

1. 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.

1. \* 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.

**Листинг**

lb5.1.cpp

#include <iostream>

using namespace std;

struct node

{

int vertex; //номер вершины

struct node\* next; //ссылка на след. вершину

};

struct node\* createNode(int); //создание вершины

struct Graph

{

int numVertices; //кол-во вершин

struct node\*\* adjLists; //массив на указатели, которые хранят адрес вершин

};

struct Graph\* createGraph(int vertices); //созднание пустого графа

void addEdge(struct Graph\* graph, int src, int dest); //добавить ребро

void printGraph(struct Graph\* graph); //вывод графа в виде списка смежности

void fillGraph(struct Graph\* graph, int\*\* g, int n); //заполнение списка смежности

void createMatrix(int\*\* g, int n); //создание матрицы

void printMatrix(int\*\* g, int n); //вывод матрицы

void freedom(int\*\* g, int n); //освобждение памяти для двумерного массива (матрицы)

int main()

{

setlocale(LC\_ALL, "");

srand(time(NULL));

int n, \*\* g1, \*\* g2;

cout << "Введите количество вершин для графов: ";

cin >> n;

cout << endl;

g1 = (int\*\*)malloc(n \* sizeof(int\*)); //выделяем память под столбец указателей

g2 = (int\*\*)malloc(n \* sizeof(int\*));

createMatrix(g1, n);

createMatrix(g2, n);

printMatrix(g1, n);

printMatrix(g2, n);

struct Graph\* graph1 = createGraph(n);

struct Graph\* graph2 = createGraph(n);

fillGraph(graph1, g1, n);

fillGraph(graph2, g2, n);

printGraph(graph1);

cout << endl;

printGraph(graph2);

freedom(g1, n);

freedom(g2, n);

}

struct node\* createNode(int v)

{

struct node\* newNode = (node\*)malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices)

{

struct Graph\* graph = (Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (node\*\*)malloc(vertices \* sizeof(struct node\*));

int i;

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

graph->adjLists[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest)

{

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

//newNode = createNode(src);

//newNode->next = graph->adjLists[dest];

//graph->adjLists[dest] = newNode;

}

void printGraph(struct Graph\* graph)

{

for (int v = 0; v < graph->numVertices; v++)

{

struct node\* temp = graph->adjLists[v];

cout << endl << "Список смежности вершин " << v << endl;

while (temp)

{

if (temp->next == NULL) {

cout << temp->vertex;

}

else {

cout << temp->vertex << " -> ";

}

temp = temp->next;

}

cout << endl;

}

}

void fillGraph(struct Graph\* graph, int\*\* g, int n) {

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = 0; j < n; j++) // цикл по столбцам строки

{

if (\*(\*(g + i) + j) == 1) {

addEdge(graph, i, j);

}

}

}

}

void createMatrix(int\*\* g, int n) {

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

\*(g + i) = (int\*)malloc(n \* sizeof(int)); //выделяем память под значения столбцов

}

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = i; j < n; j++) // цикл по столбцам строки

{

if (i == j) {

\*(\*(g + i) + j) = 0;

}

else {

\*(\*(g + i) + j) = rand() % 2;

\*(\*(g + j) + i) = \*(\*(g + i) + j);

}

}

}

}

void printMatrix(int\*\* g, int n) {

cout << " ";

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

cout.width(3);

cout << i;

}

cout << endl << endl;

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

cout << i << " ";

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

{

cout.width(3);

cout << \*(\*(g + i) + j);

}

cout << endl;

}

cout << endl << endl;

}

void freedom(int\*\* g, int n) {

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

free(\*(g + i)); //освобождаем память

}

free(g); //освобождаем память

}

lb5.2.cpp

#include <iostream>

using namespace std;

struct node

{

int vertex; //номер вершины

struct node\* next; //ссылка на след. вершину

};

struct node\* createNode(int); //создание вершины

struct Graph

{

int numVertices; //кол-во вершин

struct node\*\* adjLists; //массив на указатели, которые хранят адрес вершин

};

struct Graph\* createGraph(int vertices); //созднание пустого графа

int checker(struct node\* temp1, struct node\* temp2); //проверка на повтор

void freedom(int\*\* g, int n); //освобждение памяти для двумерного массива (матрицы)

void createMatrix(int\*\* g, int n); //создание матрицы

void printMatrix(int\*\* g, int n); //вывод матрицы

int\*\* matchVertex(int\*\* g\_old, int n); //отождествление вершин матрицы

int\*\* constrictEdge(int\*\* g\_old, int n); //стягивание ребра матрицы

int\*\* splitVertex(int\*\* g\_old, int n); //расщепление вершины матрицы

void freedom(struct Graph\* graph); //освобждение памяти для двумерного массива (графа)

void addEdge(struct Graph\* graph, int src, int dest); //добавить ребро

void printGraph(struct Graph\* graph); //вывод графа в виде списка смежности

void fillGraph(struct Graph\* graph, int\*\* g, int n); //заполнение списка смежности

struct Graph\* matchVertex(struct Graph\* graph); //отождествление вершин графа

struct Graph\* constrictEdge(struct Graph\* graph); //стягивание ребра графа

struct Graph\* splitVertex(struct Graph\* graph); //расщепление вершины графа

void constrictEdge(); //стягивание ребра графа

void splitVertex(); //расщепления вершины графа

int main()

{

setlocale(LC\_ALL, "");

srand(time(NULL));

int n1, \*\* g1;

cout << "Введите количество вершин для графа: ";

cin >> n1;

cout << endl;

g1 = (int\*\*)malloc(n1 \* sizeof(int\*)); //выделяем память под столбец указателей

createMatrix(g1, n1);

printMatrix(g1, n1);

struct Graph\* graph1 = createGraph(n1);

fillGraph(graph1, g1, graph1->numVertices);

printGraph(graph1);

printMatrix(g1, n1);

g1 = matchVertex(g1, \_msize(g1) / sizeof(int\*));

printMatrix(g1, \_msize(g1) / sizeof(int\*));

g1 = constrictEdge(g1, \_msize(g1) / sizeof(int\*));

printMatrix(g1, \_msize(g1) / sizeof(int\*));

g1 = splitVertex(g1, \_msize(g1) / sizeof(int\*));

printMatrix(g1, \_msize(g1) / sizeof(int\*));

graph1 = matchVertex(graph1);

printGraph(graph1);

graph1 = constrictEdge(graph1);

printGraph(graph1);

graph1 = splitVertex(graph1);

printGraph(graph1);

freedom(g1, \_msize(g1) / sizeof(int\*));

freedom(graph1);

}

int checker(struct node\* temp1, int E)

{

if (temp1 == 0) return 0;

return (E == temp1->vertex) + checker(temp1->next, E);

}

struct node\* createNode(int v)

{

struct node\* newNode = (node\*)malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices)

{

struct Graph\* graph = (Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (node\*\*)malloc(vertices \* sizeof(struct node\*));

int i;

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

graph->adjLists[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest)

{

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

}

void printGraph(struct Graph\* graph)

{

for (int v = 0; v < graph->numVertices; v++)

{

struct node\* temp = graph->adjLists[v];

cout << endl << "Список смежности вершин " << v << endl;

while (temp)

{

if (temp->next == NULL) {

cout << temp->vertex;

}

else {

cout << temp->vertex << " -> ";

}

temp = temp->next;

}

cout << endl;

}

cout << endl;

}

void fillGraph(struct Graph\* graph, int\*\* g, int n) {

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = 0; j < n; j++) // цикл по столбцам строки

{

if (\*(\*(g + i) + j) == 1) {

addEdge(graph, i, j);

}

}

}

}

struct Graph\* matchVertex(struct Graph\* graph) {

int v1, v2, k;

struct node\* temp1, \* temp2;

cout << "Введите номера вершин для отождествления:" << endl;

cin >> v1 >> v2;

for (int i = 0; i < graph->numVertices; i++) { //исправление нумерации с учётом отождествляемых вершин

temp1 = graph->adjLists[i];

while (temp1)

{

if (temp1->vertex > v2) {

temp1->vertex--;

}

else if (temp1->vertex == v2) {

temp1->vertex = v1;

}

temp1 = temp1->next;

}

}

temp1 = graph->adjLists[v1];

temp2 = graph->adjLists[v2];

while (temp2)

{

if (!checker(temp1, temp2->vertex)) {

addEdge(graph, v1, temp2->vertex);

}

temp2 = temp2->next;

}

for (int i = 0; i < graph->numVertices; i++) { //удаление повторений

k = 0;

temp1 = graph->adjLists[i];

while (temp1)

{

if (k != 0) {

if (temp1->vertex == v1) {

temp2->next = temp1->next;

free(temp1);

break;

}

}

temp2 = temp1;

if (temp1->vertex == v1) {

k++;

}

temp1 = temp1->next;

}

}

//копирование графа

struct Graph\* graph1 = createGraph(graph->numVertices-1);

k = 0;

for (int i = 0; i < graph->numVertices; i++) {

if (i != v2) {

temp1 = graph->adjLists[i];

while (temp1)

{

addEdge(graph1, k, temp1->vertex);

temp1 = temp1->next;

}

k++;

}

}

freedom(graph);

return graph1;

}

struct Graph\* constrictEdge(struct Graph\* graph) {

int v1, v2, k;

struct node\* temp1, \* temp2;

cout << "Введите номера вершин для стягивания ребра между ними:" << endl;

cin >> v1 >> v2;

temp1 = graph->adjLists[v1];

while (temp1) //проверка наличие ребра

{

if (temp1->vertex == v2) {

break;

}

else if (temp1->next == 0) {

cout << "Ребро отсутствует, стягивания не произошло" << endl;

return graph;

}

temp1 = temp1->next;

}

for (int i = 0; i < graph->numVertices; i++) { //замена номера второй вершины на первую и уменьшение номеров на 1, которые иду после второй вершины

temp1 = graph->adjLists[i];

while (temp1)

{

if (temp1->vertex > v2) {

temp1->vertex--;

}

else if (temp1->vertex == v2) {

temp1->vertex = v1;

}

temp1 = temp1->next;

}

}

temp1 = graph->adjLists[v1];

temp2 = graph->adjLists[v2];

while (temp2) //перенос рёбер из первой во вторую вершину с проверкой на наличие повторов

{

if (!checker(temp1, temp2->vertex)) {

addEdge(graph, v1, temp2->vertex);

}

temp2 = temp2->next;

}

for (int i = 0; i < graph->numVertices; i++) { //удаление повторов появившихся в результате замены

k = 0;

temp1 = graph->adjLists[i];

while (temp1)

{

if (k != 0) {

if (temp1->vertex == v1) {

temp2->next = temp1->next;

free(temp1);

break;

}

}

temp2 = temp1;

if (temp1->vertex == v1) {

k++;

}

temp1 = temp1->next;

}

}

temp1 = graph->adjLists[v1];

temp2 = graph->adjLists[v1];

while (temp1) //удаление петли

{

temp2 = temp1->next;

if (temp1->vertex == v1) {

free(temp1);

graph->adjLists[v1] = temp2;

break;

}

else if (temp2->vertex == v1) {

temp1->next = temp2->next;

free(temp2);

break;

}

temp1 = temp1->next;

}

struct Graph\* graph1 = createGraph(graph->numVertices - 1);

k = 0;

for (int i = 0; i < graph->numVertices; i++) {

if (i != v2) {

temp1 = graph->adjLists[i];

while (temp1)

{

addEdge(graph1, k, temp1->vertex);

temp1 = temp1->next;

}

k++;

}

}

freedom(graph);

return graph1;

}

struct Graph\* splitVertex(struct Graph\* graph) {

int v1, v2, k;

struct node\* temp1, \* temp2;

cout << "Введите номер вершины для расщепления:" << endl;

cin >> v1;

v2 = v1 + 1;

struct Graph\* graph1 = createGraph(graph->numVertices + 1);

k = 0;

for (int i = 0; i < graph1->numVertices; i++) {

temp1 = graph->adjLists[k];

if (i == v1) {

while (temp1)

{

addEdge(graph1, i, temp1->vertex);

temp1 = temp1->next;

}

}

else {

while (temp1)

{

addEdge(graph1, i, temp1->vertex);

temp1 = temp1->next;

}

k++;

}

}

freedom(graph);

for (int i = 0; i < graph1->numVertices; i++) { //добавение номера второй вершины туда где есть первая и увеличение номеров на 1, которые иду после второй вершины

temp1 = graph1->adjLists[i];

while (temp1)

{

if (temp1->vertex > v1) {

temp1->vertex++;

}

if (temp1->vertex == v1) {

addEdge(graph1, i, v2);

}

temp1 = temp1->next;

}

}

if (!checker(graph1->adjLists[v1], v2)) {

addEdge(graph1, v1, v2);

addEdge(graph1, v2, v1);

}

return graph1;

}

void createMatrix(int\*\* g, int n) {

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

\*(g + i) = (int\*)malloc(n \* sizeof(int)); //выделяем память под указатели на строки

}

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = i; j < n; j++) // цикл по столбцам строки

{

if (i == j) {

\*(\*(g + i) + j) = 0;

}

else {

\*(\*(g + i) + j) = rand() % 2;

\*(\*(g + j) + i) = \*(\*(g + i) + j);

}

}

}

}

void printMatrix(int\*\* g, int n) {

cout << " ";

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

cout.width(3);

cout << i;

}

cout << endl << endl;

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

cout << i << " ";

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

{

cout.width(3);

cout << \*(\*(g + i) + j);

}

cout << endl;

}

cout << endl << endl;

}

int\*\* matchVertex(int\*\* g\_old, int n) {

int m = n - 1, v1, v2, k1, k2;

cout << "Введите номера вершин для отождествления:" << endl;

cin >> v1 >> v2;

int\*\* g = (int\*\*)malloc(m \* sizeof(int\*));

for (int i = 0; i < m; i++) {

\*(g + i) = (int\*)malloc(m \* sizeof(int));

}

for (int j = 0; j < n; j++) //записываем соединения двух вершин в одну (наименьшую по номеру)

{

\*(\*(g\_old + v1) + j) = \*(\*(g\_old + v1) + j) | \*(\*(g\_old + v2) + j);

\*(\*(g\_old + j) + v1) = \*(\*(g\_old + v1) + j);

}

if (\*(\*(g\_old + v1) + v2) == 1) { //если есть петля на одной из вершин

\*(\*(g\_old + v1) + v1) = 1;

}

k1 = 0;

for (int i = 0; i < n; i++) // цикл по строкам

{

k2 = 0;

if (i != v2) { //пропускаем строку наибольшой вершины из тех которые отождествляем

for (int j = 0; j < n; j++) // цикл по столбцам строки

{

if (j != v2) { //пропускаем столбец наибольшой вершины из тех которые отождествляем

\*(\*(g + k1) + k2) = \*(\*(g\_old + i) + j);

k2++;

}

}

k1++;

}

}

freedom(g\_old, n);

return g;

}

int\*\* constrictEdge(int\*\* g\_old, int n) {

int m = n - 1, v1, v2, k1, k2;

cout << "Введите номера вершин для стягивания ребра между ними:" << endl;

cin >> v1 >> v2;

if (\*(\*(g\_old + v1) + v2) == 0) {

cout << "Ребро отсутствует, стягивания не произошло" << endl;

return g\_old;

}

else if (v1 == v2) {

\*(\*(g\_old + v1) + v2) = 0;

return g\_old;

}

else {

int\*\* g = (int\*\*)malloc(m \* sizeof(int\*));

for (int i = 0; i < m; i++) {

\*(g + i) = (int\*)malloc(m \* sizeof(int));

}

for (int j = 0; j < n; j++) //записываем соединения двух вершин в одну (наименьшую по номеру)

{

\*(\*(g\_old + v1) + j) = \*(\*(g\_old + v1) + j) | \*(\*(g\_old + v2) + j);

\*(\*(g\_old + j) + v1) = \*(\*(g\_old + v1) + j);

}

\*(\*(g\_old + v1) + v1) = 0;

k1 = 0;

for (int i = 0; i < n; i++) // цикл по строкам

{

k2 = 0;

if (i != v2) { //пропускаем строку наибольшой вершины из тех которые отождествляем

for (int j = 0; j < n; j++) // цикл по столбцам строки

{

if (j != v2) { //пропускаем столбец наибольшой вершины из тех которые отождествляем

\*(\*(g + k1) + k2) = \*(\*(g\_old + i) + j);

k2++;

}

}

k1++;

}

}

freedom(g\_old, n);

return g;

}

}

int\*\* splitVertex(int\*\* g\_old, int n) {

int m = n + 1, v1, v2, k1, k2;

cout << "Введите номер вершины для расщепления:" << endl;

cin >> v1;

v2 = v1 + 1;

int\*\* g = (int\*\*)malloc(m \* sizeof(int\*));

for (int i = 0; i < m; i++) {

\*(g + i) = (int\*)malloc(m \* sizeof(int));

}

k1 = 0;

for (int i = 0; i < m; i++) // цикл по строкам

{

k2 = 0;

if (i != v1) { //заполняем новую строку значениями из предыдущей если это она

for (int j = 0; j < m; j++) // цикл по столбцам строки

{

if (j != v1) { //заполняем новый столбец значение из предыдущего если это он

\*(\*(g + i) + j) = \*(\*(g\_old + k1) + k2);

k2++;

}

else {

\*(\*(g + i) + j) = \*(\*(g\_old + k1) + k2);

}

}

k1++;

}

else {

for (int j = 0; j < m; j++) // цикл по столбцам строки

{

if (j != v1) { //заполняем новый столбец значение из предыдущего если это он

\*(\*(g + i) + j) = \*(\*(g\_old + k1) + k2);

k2++;

}

else {

\*(\*(g + i) + j) = \*(\*(g\_old + k1) + k2);

}

}

}

}

\*(\*(g + v1) + v2) = 1;

\*(\*(g + v2) + v1) = 1;

freedom(g\_old, n);

return g;

}

void freedom(int\*\* g, int n) {

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

free(\*(g + i)); //освобождаем память

}

free(g); //освобождаем память

}

void freedom(struct Graph\* graph) {

struct node\* temp1, \* temp2;

for (int i = 0; i < graph->numVertices; i++) {

temp1 = graph->adjLists[i];

temp2 = graph->adjLists[i];

while (temp2)

{

temp1 = temp1->next;

free(temp2);

temp2 = temp1;

}

}

free(graph); //освобождаем память

}

lb5.3.cpp

#include <iostream>

using namespace std;

void createMatrix(int\*\* g, int n); //создание матрицы

void printMatrix(int\*\* g, int n); //вывод матрицы

int\*\* mergeMatrix(int\*\* g1, int\*\* g2, int n1, int n2); //объединение графов

int\*\* crossMatrix(int\*\* g1, int\*\* g2, int n1, int n2); //пересечение графов

int\*\* ringSumMatrix(int\*\* g1, int\*\* g2, int n1, int n2); //кольцевая сумма графов

void freedom(int\*\* g, int n); //освобждение памяти для двумерного массива (матрицы)

int main()

{

setlocale(LC\_ALL, "");

srand(time(NULL));

int n1, n2, \*\* g1, \*\* g2, \*\*merger, \*\*crossing, \*\*ringSum;

cout << "Введите количество вершин для перевого графа: ";

cin >> n1;

cout << "Введите количество вершин для второго графа: ";

cin >> n2;

cout << endl;

g1 = (int\*\*)malloc(n1 \* sizeof(int\*)); //выделяем память под столбец указателей

g2 = (int\*\*)malloc(n2 \* sizeof(int\*));

createMatrix(g1, n1);

createMatrix(g2, n2);

printMatrix(g1, n1);

printMatrix(g2, n2);

merger = mergeMatrix(g1, g2, n1, n2);

cout << "Объединение графов" << endl;

printMatrix(merger, \_msize(merger) / sizeof(int\*));

crossing = crossMatrix(g1, g2, n1, n2);

cout << "Пересечение графов" << endl;

printMatrix(crossing, \_msize(crossing) / sizeof(int\*));

ringSum = ringSumMatrix(g1, g2, n1, n2);

cout << "Кольцевая сумма графов" << endl;

printMatrix(ringSum, \_msize(ringSum) / sizeof(int\*));

freedom(g1, n1);

freedom(g2, n2);

freedom(merger, \_msize(merger) / sizeof(int\*));

freedom(crossing, \_msize(crossing) / sizeof(int\*));

freedom(ringSum, \_msize(ringSum) / sizeof(int\*));

}

void createMatrix(int\*\* g, int n) {

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

\*(g + i) = (int\*)malloc(n \* sizeof(int)); //выделяем память под значения столбцов

}

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = i; j < n; j++) // цикл по столбцам строки

{

if (i == j) {

\*(\*(g + i) + j) = 0;

}

else {

\*(\*(g + i) + j) = rand() % 2;

\*(\*(g + j) + i) = \*(\*(g + i) + j);

}

}

}

}

void printMatrix(int\*\* g, int n) {

cout << " ";

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

cout.width(3);

cout << i;

}

cout << endl << endl;

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

cout << i << " ";

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

{

cout.width(3);

cout << \*(\*(g + i) + j);

}

cout << endl;

}

cout << endl << endl;

}

int\*\* mergeMatrix(int\*\* g1, int\*\* g2, int n1, int n2) {

if (n1 > n2) { //если первая матрица больше второй

int\*\* temp2 = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp2 + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

if (i >= n2 || j >= n2) {

\*(\*(temp2 + i) + j) = 0;

}

else {

\*(\*(temp2 + i) + j) = \*(\*(g2 + i) + j);

}

}

}

for (int i = 0; i < n1; i++) // цикл по строкам

{

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp2 + i) + j) = \*(\*(temp2 + i) + j) | \*(\*(g1 + i) + j);

}

}

return temp2;

}

else if (n1 < n2) { //если вторая матрица больше первой

int\*\* temp1 = (int\*\*)malloc(n2 \* sizeof(int\*));

for (int i = 0; i < n2; i++) // цикл по строкам

{

\*(temp1 + i) = (int\*)malloc(n2 \* sizeof(int));

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

if (i >= n1 || j >= n1) {

\*(\*(temp1 + i) + j) = 0;

}

else {

\*(\*(temp1 + i) + j) = \*(\*(g1 + i) + j);

}

}

}

for (int i = 0; i < n2; i++) // цикл по строкам

{

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

\*(\*(temp1 + i) + j) = \*(\*(temp1 + i) + j) | \*(\*(g2 + i) + j);

}

}

return temp1;

}

else {

int\*\* temp = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp + i) + j) = \*(\*(g1 + i) + j) | \*(\*(g2 + i) + j);

}

}

return temp;

}

}

int\*\* crossMatrix(int\*\* g1, int\*\* g2, int n1, int n2) {

if (n1 > n2) { //если первая матрица больше второй

int\*\* temp2 = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp2 + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

if (i >= n2 || j >= n2) {

\*(\*(temp2 + i) + j) = 0;

}

else {

\*(\*(temp2 + i) + j) = \*(\*(g2 + i) + j);

}

}

}

for (int i = 0; i < n1; i++) // цикл по строкам

{

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp2 + i) + j) = \*(\*(temp2 + i) + j) & \*(\*(g1 + i) + j);

}

}

return temp2;

}

else if (n1 < n2) { //если вторая матрица больше первой

int\*\* temp1 = (int\*\*)malloc(n2 \* sizeof(int\*));

for (int i = 0; i < n2; i++) // цикл по строкам

{

\*(temp1 + i) = (int\*)malloc(n2 \* sizeof(int));

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

if (i >= n1 || j >= n1) {

\*(\*(temp1 + i) + j) = 0;

}

else {

\*(\*(temp1 + i) + j) = \*(\*(g1 + i) + j);

}

}

}

for (int i = 0; i < n2; i++) // цикл по строкам

{

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

\*(\*(temp1 + i) + j) = \*(\*(temp1 + i) + j) & \*(\*(g2 + i) + j);

}

}

return temp1;

}

else {

int\*\* temp = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp + i) + j) = \*(\*(g1 + i) + j) & \*(\*(g2 + i) + j);

}

}

return temp;

}

}

int\*\* ringSumMatrix(int\*\* g1, int\*\* g2, int n1, int n2) {

if (n1 > n2) { //если первая матрица больше второй

int\*\* temp2 = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp2 + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

if (i >= n2 || j >= n2) {

\*(\*(temp2 + i) + j) = 0;

}

else {

\*(\*(temp2 + i) + j) = \*(\*(g2 + i) + j);

}

}

}

for (int i = 0; i < n1; i++) // цикл по строкам

{

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp2 + i) + j) = \*(\*(temp2 + i) + j) ^ \*(\*(g1 + i) + j);

}

}

return temp2;

}

else if (n1 < n2) { //если вторая матрица больше первой

int\*\* temp1 = (int\*\*)malloc(n2 \* sizeof(int\*));

for (int i = 0; i < n2; i++) // цикл по строкам

{

\*(temp1 + i) = (int\*)malloc(n2 \* sizeof(int));

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

if (i >= n1 || j >= n1) {

\*(\*(temp1 + i) + j) = 0;

}

else {

\*(\*(temp1 + i) + j) = \*(\*(g1 + i) + j);

}

}

}

for (int i = 0; i < n2; i++) // цикл по строкам

{

for (int j = 0; j < n2; j++) // цикл по столбцам строки

{

\*(\*(temp1 + i) + j) = \*(\*(temp1 + i) + j) ^ \*(\*(g2 + i) + j);

}

}

return temp1;

}

else {

int\*\* temp = (int\*\*)malloc(n1 \* sizeof(int\*));

for (int i = 0; i < n1; i++) // цикл по строкам

{

\*(temp + i) = (int\*)malloc(n1 \* sizeof(int));

for (int j = 0; j < n1; j++) // цикл по столбцам строки

{

\*(\*(temp + i) + j) = \*(\*(g1 + i) + j) ^ \*(\*(g2 + i) + j);

}

}

return temp;

}

}

void freedom(int\*\* g, int n) {

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

free(\*(g + i)); //освобождаем память

}

free(g); //освобождаем память

}

lb5.4.cpp

#include <iostream>

using namespace std;

void createMatrix(int\*\* g, int n); //создание матрицы

void printMatrix(int\*\* g, int n); //вывод матрицы

void decart(int\*\* g1, int\*\* g2, int\*\* g3, int n1, int n2); //декартовое произведение

void freedom(int\*\* g, int n); //освобждение памяти для двумерного массива (матрицы)

int main()

{

setlocale(LC\_ALL, "");

srand(time(NULL));

int n1, n2, \*\* g1, \*\* g2, \*\*g3;

cout << "Введите количество вершин для перевого графа: ";

cin >> n1;

cout << "Введите количество вершин для второго графа: ";

cin >> n2;

cout << endl;

g1 = (int\*\*)malloc(n1 \* sizeof(int\*)); //выделяем память под столбец указателей

g2 = (int\*\*)malloc(n2 \* sizeof(int\*));

g3 = (int\*\*)malloc(n1 \* n2 \* sizeof(int\*));

createMatrix(g1, n1);

createMatrix(g2, n2);

createMatrix(g3, n1 \* n2);

printMatrix(g1, n1);

printMatrix(g2, n2);

decart(g1, g2, g3, n1, n2);

printMatrix(g3, n1 \* n2);

//printMatrix(merger, \_msize(merger) / sizeof(int\*));

freedom(g1, n1);

freedom(g2, n2);

freedom(g3, n1 \* n2);

}

void createMatrix(int\*\* g, int n) {

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

\*(g + i) = (int\*)malloc(n \* sizeof(int)); //выделяем память под значения столбцов

}

for (int i = 0; i < n; i++) // цикл по строкам

{

for (int j = i; j < n; j++) // цикл по столбцам строки

{

if (i == j) {

\*(\*(g + i) + j) = 0;

}

else {

\*(\*(g + i) + j) = rand() % 2;

\*(\*(g + j) + i) = \*(\*(g + i) + j);

}

}

}

}

void printMatrix(int\*\* g, int n) {

cout << " ";

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

cout.width(3);

cout << i;

}

cout << endl << endl;

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

cout << i << " ";

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

{

cout.width(3);

cout << \*(\*(g + i) + j);

}

cout << endl;

}

cout << endl << endl;

}

void decart(int\*\* g1, int\*\* g2, int\*\* g3, int n1, int n2) {

int x = -1, y = -1;

for (int i = 0; i < n1 \* n2; i++) {

g3[i] = (int\*)malloc(n1 \* n2 \* sizeof(int)); //выделяем память под значения столбцов

for (int j = 0; j < n1 \* n2; j++) {

g3[i][j] = 0;

}

}

for (int i = 0; i < n1; i++) {

for (int j = 0; j < n2; j++) {

x++;

y = -1;

for (int k = 0; k < n1; k++) {

for (int l = 0; l < n2; l++) {

y++;

if (i == k) g3[x][y] = g2[j][l];

if (j == l) g3[x][y] = g1[i][k];

}

}

}

}

}

void freedom(int\*\* g, int n) {

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

free(\*(g + i)); //освобождаем память

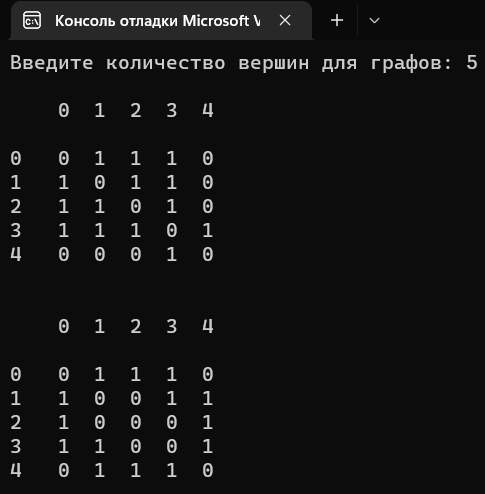
}

free(g); //освобождаем память

}

**Результат работы программы**

Результаты работы программы (задание 1) показаны на рисунке 1, 2, 3.

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

**Рисунок 1, 2, 3 — Результаты работы программы**

Результаты работы программы (задание 2) показаны на рисунке 4, 5.

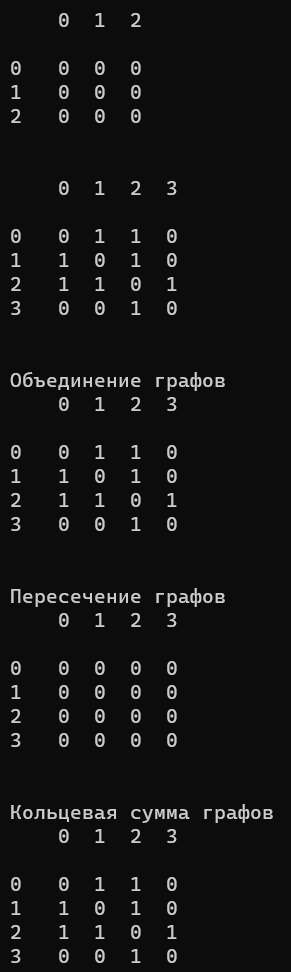
Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

**Рисунок 4, 5 — Результаты работы программы**

Результаты работы программы (задание 3, 4) показаны на рисунке 6, 7.

 Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 6, 7 — Результаты работы программы**

### Выводы

Граф – фигура, состоящая из заданных точек (вершин), соединенных отрезками, которые, называются ребрами (дугами) графа.

Отождествление – операция объединения двух вершин и их рёбер в одну.

Стягивание – операция удаления дуги или ребра и отождествление его концевых вершин.

Расщепление – операция деления одной вершины на две (рёбра у этих вершин одинаковые) и между этими вершинами будет ребро.

Объединение – операция сложения двух графов в один.

Пересечение – операция вычленения одинаковых рёбер в двух графах (результат записывается в новый граф).

Кольцевая сумма – операция вычленения не повторяющихся рёбер в двух графах (результат записывается в новый граф).

Декартово произведение – операция перемножения двух графов.