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»

**Выполнил:**

**Студент группы 23ВВВ2**

Стрельцов А.П.

Федоров Б.М.

**Приняли:**

Юрова О. В.

Митрохин М. А.

Пенза 2024

**Цель работы:** проработать унарные и бинарные операции с графами, выполнить задания и усвоить материал.

**Лабораторное задание.**

**Задание 1**

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

**ЛИСТИНГ**

**lab6-1.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <locale.h>

// Структура для представления списка смежности

typedef struct Node {

int vertex; // Номер вершины

struct Node\* next; // Указатель на следующий элемент списка

} Node;

typedef struct {

Node\* head; // Указатель на голову списка

} AdjList;

void generate\_adjacency\_matrix(int\*\* matrix, int vertices) {

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

for (int j = i; j < vertices; j++) {

if (i == j) {

matrix[i][j] = 0; // Нет петель

}

else {

matrix[i][j] = rand() % 2; // Генерация наличия ребра (0 или 1)

matrix[j][i] = matrix[i][j]; // Симметричность для неориентированного графа

}

}

}

}

void print\_adjacency\_matrix(int\*\* matrix, int vertices) {

printf("Матрица смежности:\n");

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

void convert\_to\_adjacency\_list(int\*\* matrix, AdjList\* list, int vertices) {

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

list[i].head = NULL; // Инициализация списка

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

if (matrix[i][j] == 1) {

// Создание нового узла для списка

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->vertex = j; // Установка номера вершины

new\_node->next = list[i].head; // Указатель на предыдущую голову списка

list[i].head = new\_node; // Обновление головы списка

}

}

}

}

void print\_adjacency\_list(AdjList\* list, int vertices) {

printf("Список смежности:\n");

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

printf("Вершина %d: ", i);

Node\* current = list[i].head;

while (current != NULL) {

printf("%d -> ", current->vertex);

current = current->next;

}

printf("NULL\n");

}

}

void free\_adjacency\_list(AdjList\* list, int vertices) {

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

Node\* current = list[i].head;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

}

int main() {

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

int vertices1, vertices2;

printf("Введите количество вершин для матрицы M1: ");

scanf("%d", &vertices1);

printf("Введите количество вершин для матрицы M2: ");

scanf("%d", &vertices2);

// Проверка на допустимое количество вершин

if (vertices1 < 1 || vertices2 < 1) {

printf("Ошибка: количество вершин должно быть больше 0.\n");

return 1;

}

// Динамическое выделение памяти для матриц смежности

int\*\* M1 = (int\*\*)malloc(vertices1 \* sizeof(int\*));

int\*\* M2 = (int\*\*)malloc(vertices2 \* sizeof(int\*));

for (int i = 0; i < vertices1; i++) {

M1[i] = (int\*)malloc(vertices1 \* sizeof(int));

}

for (int i = 0; i < vertices2; i++) {

M2[i] = (int\*)malloc(vertices2 \* sizeof(int));

}

// Генерация первой матрицы смежности

generate\_adjacency\_matrix(M1, vertices1);

print\_adjacency\_matrix(M1, vertices1);

// Генерация второй матрицы смежности

generate\_adjacency\_matrix(M2, vertices2);

print\_adjacency\_matrix(M2, vertices2);

// Преобразование первой матрицы в список смежности

AdjList\* list1 = (AdjList\*)malloc(vertices1 \* sizeof(AdjList));

convert\_to\_adjacency\_list(M1, list1, vertices1);

print\_adjacency\_list(list1, vertices1);

// Преобразование второй матрицы в список смежности

AdjList\* list2 = (AdjList\*)malloc(vertices2 \* sizeof(AdjList));

convert\_to\_adjacency\_list(M2, list2, vertices2);

print\_adjacency\_list(list2, vertices2);

// Освобождение памяти

free\_adjacency\_list(list1, vertices1);

free\_adjacency\_list(list2, vertices2);

free(list1);

free(list2);

for (int i = 0; i < vertices1; i++) {

free(M1[i]);

}

for (int i = 0; i < vertices2; i++) {

free(M2[i]);

}

free(M1);

free(M2);

return 0;

}

**Результат работы программы**

На рисунке 1 показана реализация задания №1.

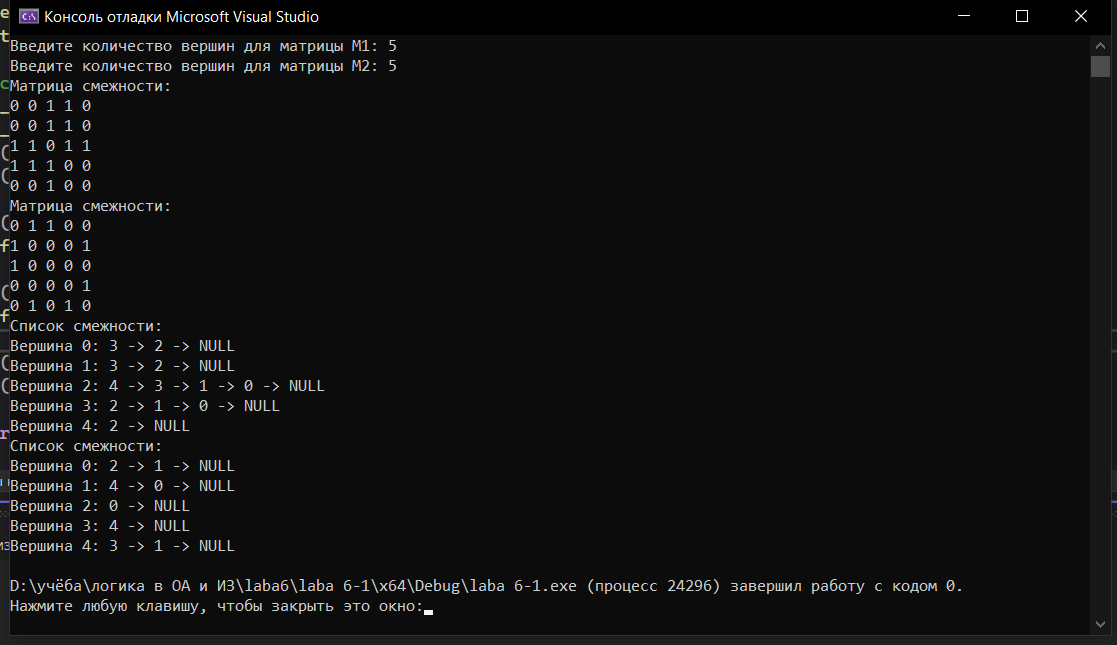


Рисунок 1 - Результат работы программы №1.

**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.

**ЛИСТИНГ**

**lab6-2.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <locale.h>

// Структура для представления списка смежности

typedef struct Node {

int vertex; // Номер вершины

struct Node\* next; // Указатель на следующий элемент списка

} Node;

typedef struct {

Node\* head; // Указатель на голову списка

} AdjList;

void generate\_adjacency\_matrix(int\*\* matrix, int vertices) {

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

for (int j = i; j < vertices; j++) {

if (i == j) {

matrix[i][j] = 0;

}

else {

matrix[i][j] = rand() % 2;

matrix[j][i] = matrix[i][j];

}

}

}

}

void print\_adjacency\_matrix(int\*\* matrix, int vertices) {

printf("Матрица смежности:\n");

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

void convert\_to\_adjacency\_list(int\*\* matrix, AdjList\* list, int vertices) {

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

list[i].head = NULL; // Инициализация списка

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

if (matrix[i][j] == 1) {

// Создание нового узла для списка

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->vertex = j; // Установка номера вершины

new\_node->next = list[i].head; // Указатель на предыдущую голову списка

list[i].head = new\_node; // Обновление головы списка

}

}

}

}

void print\_adjacency\_list(AdjList\* list, int vertices) {

printf("Список смежности:\n");

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

printf("Вершина %d: ", i);

Node\* current = list[i].head;

while (current != NULL) {

printf("%d -> ", current->vertex);

current = current->next;

}

printf("NULL\n");

}

}

void free\_adjacency\_list(AdjList\* list, int vertices) {

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

Node\* current = list[i].head;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

}

void remove\_vertex(int\*\* matrix, int\* vertices, int v) {

for (int i = v; i < \*vertices - 1; i++) {

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

matrix[i][j] = matrix[i + 1][j];

}

}

for (int i = 0; i < \*vertices - 1; i++) {

for (int j = v; j < \*vertices - 1; j++) {

matrix[i][j] = matrix[i][j + 1];

}

}

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

int\* temp = (int\*)realloc(matrix[i], (\*vertices - 1) \* sizeof(int));

if (temp == NULL) {

printf("Ошибка перераспределения памяти!\n");

exit(1);

}

matrix[i] = temp;

}

\*vertices -= 1;

}

void merge\_vertices(int\*\* matrix, int\* vertices, int v1, int v2) {

if (v1 == v2) return; // Нельзя объединить вершину с собой

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

matrix[v1][i] = matrix[v1][i] | matrix[v2][i];

matrix[i][v1] = matrix[i][v1] | matrix[i][v2];

}

remove\_vertex(matrix, vertices, v2);

}

void collapse\_edge(int\*\* matrix, int\* vertices, int v1, int v2) {

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

matrix[v1][i] = matrix[v1][i] | matrix[v2][i];

matrix[i][v1] = matrix[i][v1] | matrix[i][v2];

}

remove\_vertex(matrix, vertices, v2);

}

void split\_vertex(int\*\*\* matrix, int\* vertices, int v) {

int new\_size = \*vertices + 1;

int\*\* temp\_matrix = (int\*\*)realloc(\*matrix, new\_size \* sizeof(int\*));

if (temp\_matrix == NULL) {

printf("Ошибка перераспределения памяти!\n");

exit(1);

}

\*matrix = temp\_matrix;

for (int i = 0; i < new\_size; i++) {

int\* temp\_row = (int\*)realloc((\*matrix)[i], new\_size \* sizeof(int));

if (temp\_row == NULL) {

printf("Ошибка перераспределения памяти!\n");

exit(1);

}

(\*matrix)[i] = temp\_row;

}

for (int i = 0; i < new\_size; i++) {

(\*matrix)[new\_size - 1][i] = 0;

(\*matrix)[i][new\_size - 1] = 0;

}

// Дублируем связи для новой вершины

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

(\*matrix)[new\_size - 1][i] = (\*matrix)[v][i];

(\*matrix)[i][new\_size - 1] = (\*matrix)[i][v];

}

// Обнуляем петлю для новой вершины

(\*matrix)[new\_size - 1][new\_size - 1] = 0;

\*vertices = new\_size;

}

void free\_matrix(int\*\* matrix, int vertices) {

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

free(matrix[i]);

}

free(matrix);

}

int main() {

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

int vertices;

printf("Введите количество вершин: ");

if (scanf("%d", &vertices) != 1 || vertices < 1) {

printf("Ошибка: количество вершин должно быть больше 0.\n");

return 1;

}

int\*\* M = (int\*\*)malloc(vertices \* sizeof(int\*));

if (M == NULL) {

printf("Ошибка выделения памяти!\n");

return 1;

}

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

M[i] = (int\*)malloc(vertices \* sizeof(int));

if (M[i] == NULL) {

printf("Ошибка выделения памяти!\n");

return 1;

}

}

generate\_adjacency\_matrix(M, vertices);

print\_adjacency\_matrix(M, vertices);

AdjList\* list = (AdjList\*)malloc(vertices \* sizeof(AdjList));

convert\_to\_adjacency\_list(M, list, vertices);

print\_adjacency\_list(list, vertices);

int operation, v1, v2;

while (1) {

printf("\nВыберите операцию:\n");

printf("1. Отождествление вершин\n");

printf("2. Стягивание ребра\n");

printf("3. Расщепление вершины\n");

printf("4. Выход\n");

if (scanf("%d", &operation) != 1) {

printf("Некорректный ввод!\n");

break;

}

if (operation == 4) {

break;

}

switch (operation) {

case 1:

printf("Введите номера вершин для отождествления (0-%d): ", vertices - 1);

if (scanf("%d %d", &v1, &v2) != 2 || v1 >= vertices || v2 >= vertices || v1 < 0 || v2 < 0) {

printf("Неверные номера вершин!\n");

}

else {

merge\_vertices(M, &vertices, v1, v2);

printf("\nМатрица после отождествления вершин %d и %d:\n", v1, v2);

print\_adjacency\_matrix(M, vertices);

AdjList\* list = (AdjList\*)malloc(vertices \* sizeof(AdjList));

convert\_to\_adjacency\_list(M, list, vertices);

print\_adjacency\_list(list, vertices);

}

break;

case 2:

printf("Введите номера вершин для стягивания ребра (0-%d): ", vertices - 1);

if (scanf("%d %d", &v1, &v2) != 2 || v1 >= vertices || v2 >= vertices || v1 < 0 || v2 < 0 || v1 == v2) {

printf("Неверные номера вершин!\n");

}

if (M[v1][v2] == 0) {

printf("Ребро между вершинами %d и %d отсутствует. Выберите другое ребро.\n", v1, v2);

break;

}

else {

collapse\_edge(M, &vertices, v1, v2);

printf("\nМатрица после стягивания ребра между вершинами %d и %d:\n", v1, v2);

print\_adjacency\_matrix(M, vertices);

AdjList\* list = (AdjList\*)malloc(vertices \* sizeof(AdjList));

convert\_to\_adjacency\_list(M, list, vertices);

print\_adjacency\_list(list, vertices);

}

break;

case 3:

printf("Введите номер вершины для расщепления (0-%d): ", vertices - 1);

if (scanf("%d", &v1) != 1 || v1 >= vertices || v1 < 0) {

printf("Неверный номер вершины!\n");

}

else {

split\_vertex(&M, &vertices, v1);

printf("\nМатрица после расщепления вершины %d:\n", v1);

print\_adjacency\_matrix(M, vertices);

AdjList\* list = (AdjList\*)malloc(vertices \* sizeof(AdjList));

convert\_to\_adjacency\_list(M, list, vertices);

print\_adjacency\_list(list, vertices);

}

break;

default:

printf("Некорректный выбор операции!\n");

break;

}

}

free\_matrix(M, vertices);

return 0;

}

**Результат работы программы**

На рисунке 2-5 показана реализация задания №2.



Рисунок 2 - Результат работы программы №2.



Рисунок 3 - Результат работы программы №2.

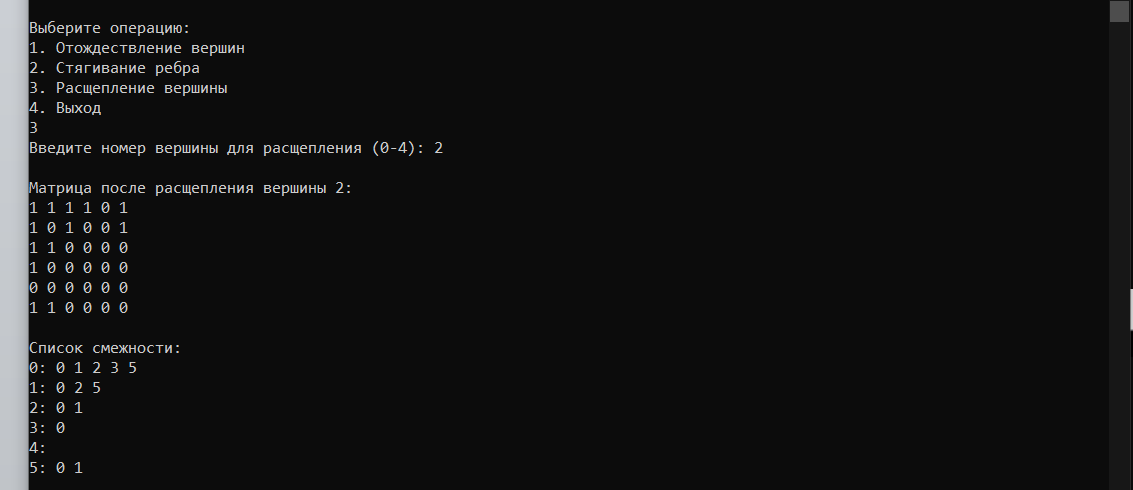


Рисунок 4 - Результат работы программы №2.

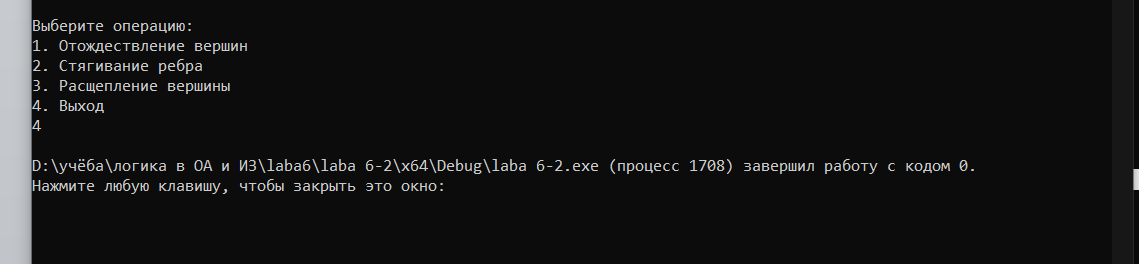


Рисунок 5 - Результат работы программы №2.

**Задание 3**

1. Для матричной формы представления графов выполните операцию:

а) объединения *G* = *G*1 ![Изображение выглядит как черный, темнота

Автоматически созданное описание](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAACcAAAAcCAYAAADiHqZbAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAAIdUAACHVAQSctJ0AAABuSURBVFhH7Y/LCcAwDEO9RPfpJl2th+7ZuJ+AL7nIcilBDwQ5JNKLCSEm4HxTQbpbciiSQ5Fcht/LVQhSOivkqJ1swatrfY55dg9LkP3Rm16aKS4R66CC6DuIODYa3DzxzuL5lDg+yuERQkyOWQORjkmxGiAtDgAAAABJRU5ErkJggg==) *G*2

б) пересечения  *G* = *G*1 ![Изображение выглядит как черный, темнота

Автоматически созданное описание](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAACcAAAAcCAYAAADiHqZbAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAAIdUAACHVAQSctJ0AAABcSURBVFhH7c9BCsAgDETRnL4n8Z5Vt1JUkokI/Q+yK+OvAfiBd+OedkeNAV9Ku9U3cp7H0iMVD6QFqoblgepB2Z46rLs6rpPFZQj/dHhggrgI4ryI88rcBnCQWQVL40PoLLwd7QAAAABJRU5ErkJggg==) *G*2

в) кольцевой суммы *G* = *G*1 ![Изображение выглядит как черный, темнота

Автоматически созданное описание](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAACcAAAAnBAMAAAB+jUwGAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAYUExURQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFY3HCoAAAAIdFJOUwD/F81D9+chTvvhKwAAAAlwSFlzAAAh1QAAIdUBBJy0nQAAAHVJREFUOMvVkd0NgDAIhLm4QG8ER3EER3AE938xtYVrTWOMT3pJCXz0h4L9TCAZfnVmZqUOgrIhllVp2YnkiSRINxCcUpcpEIKLoO4484jwGQRdb44PX2/qxKh43v9dXVobOOynRz6Pqjyf7cLMds0tpvlxmR1MEQbeirHZ/gAAAABJRU5ErkJggg==) *G*2

Результат выполнения операции выведите на экран.

**ЛИСТИНГ**

**lab6-3.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <locale.h>

// Функция для генерации матрицы смежности случайного графа

void generate\_adjacency\_matrix(int\*\* matrix, int size) {

// Инициализируем матрицу нулями

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

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

matrix[i][j] = 0;

}

}

// Случайно заполняем верхний треугольник матрицы для создания неориентированных рёбер

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

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

if (rand() % 2) { // С вероятностью 0.5 создаём ребро

matrix[i][j] = 1;

matrix[j][i] = 1; // Граф неориентированный

}

}

}

}

// Функция для вывода матрицы смежности

void print\_matrix(int\*\* matrix, int size) {

printf("Матрица смежности:\n");

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

// Функция для освобождения памяти матрицы

void free\_matrix(int\*\* matrix, int size) {

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

free(matrix[i]);

}

free(matrix);

}

// Функция для объединения двух графов

int\*\* union\_graphs(int\*\* m1, int\*\* m2, int size1, int size2) {

int max\_size = (size1 > size2) ? size1 : size2;

int\*\* union\_matrix = (int\*\*)malloc(max\_size \* sizeof(int\*));

for (int i = 0; i < max\_size; i++) {

union\_matrix[i] = (int\*)calloc(max\_size, sizeof(int));

for (int j = 0; j < max\_size; j++) {

int edge1 = (i < size1 && j < size1) ? m1[i][j] : 0;

int edge2 = (i < size2 && j < size2) ? m2[i][j] : 0;

union\_matrix[i][j] = edge1 | edge2; // Операция объединения (OR)

}

}

return union\_matrix;

}

// Функция для пересечения двух графов

int\*\* intersection\_graphs(int\*\* m1, int\*\* m2, int size1, int size2) {

int min\_size = (size1 < size2) ? size1 : size2;

int\*\* intersection\_matrix = (int\*\*)malloc(min\_size \* sizeof(int\*));

for (int i = 0; i < min\_size; i++) {

intersection\_matrix[i] = (int\*)calloc(min\_size, sizeof(int));

for (int j = 0; j < min\_size; j++) {

int edge1 = (i < size1 && j < size1) ? m1[i][j] : 0;

int edge2 = (i < size2 && j < size2) ? m2[i][j] : 0;

intersection\_matrix[i][j] = edge1 & edge2; // Операция пересечения (AND)

}

}

return intersection\_matrix;

}

// Функция для кольцевой суммы двух графов

int\*\* ring\_sum\_graphs(int\*\* m1, int\*\* m2, int size1, int size2) {

int max\_size = (size1 > size2) ? size1 : size2;

int\*\* ring\_sum\_matrix = (int\*\*)malloc(max\_size \* sizeof(int\*));

for (int i = 0; i < max\_size; i++) {

ring\_sum\_matrix[i] = (int\*)calloc(max\_size, sizeof(int));

for (int j = 0; j < max\_size; j++) {

int edge1 = (i < size1 && j < size1) ? m1[i][j] : 0;

int edge2 = (i < size2 && j < size2) ? m2[i][j] : 0;

ring\_sum\_matrix[i][j] = edge1 ^ edge2; // Операция кольцевой суммы (XOR)

}

}

return ring\_sum\_matrix;

}

int main() {

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

int size1, size2;

printf("Введите размер первого графа: ");

scanf("%d", &size1);

printf("Введите размер второго графа: ");

scanf("%d", &size2);

// Выделяем память для матриц смежности

int\*\* M1 = (int\*\*)malloc(size1 \* sizeof(int\*));

int\*\* M2 = (int\*\*)malloc(size2 \* sizeof(int\*));

for (int i = 0; i < size1; i++) {

M1[i] = (int\*)malloc(size1 \* sizeof(int));

}

for (int i = 0; i < size2; i++) {

M2[i] = (int\*)malloc(size2 \* sizeof(int));

}

// Генерируем матрицы смежности

generate\_adjacency\_matrix(M1, size1);

generate\_adjacency\_matrix(M2, size2);

// Выводим сгенерированные матрицы

printf("\nМатрица смежности графа G1:\n");

print\_matrix(M1, size1);

printf("\nМатрица смежности графа G2:\n");

print\_matrix(M2, size2);

// Выполняем операции над графами

int\*\* union\_matrix = union\_graphs(M1, M2, size1, size2);

int\*\* intersection\_matrix = intersection\_graphs(M1, M2, size1, size2);

int\*\* ring\_sum\_matrix = ring\_sum\_graphs(M1, M2, size1, size2);

// Выводим результаты операций

printf("\nМатрица объединения графов:\n");

print\_matrix(union\_matrix, (size1 > size2) ? size1 : size2);

printf("\nМатрица пересечения графов:\n");

print\_matrix(intersection\_matrix, (size1 < size2) ? size1 : size2);

printf("\nМатрица кольцевой суммы графов:\n");

print\_matrix(ring\_sum\_matrix, (size1 > size2) ? size1 : size2);

// Освобождаем выделенную память

free\_matrix(M1, size1);

free\_matrix(M2, size2);

free\_matrix(union\_matrix, (size1 > size2) ? size1 : size2);

free\_matrix(intersection\_matrix, (size1 < size2) ? size1 : size2);

free\_matrix(ring\_sum\_matrix, (size1 > size2) ? size1 : size2);

return 0;

}

**Результат работы программы**

На рисунке 6 показана реализация задания №3

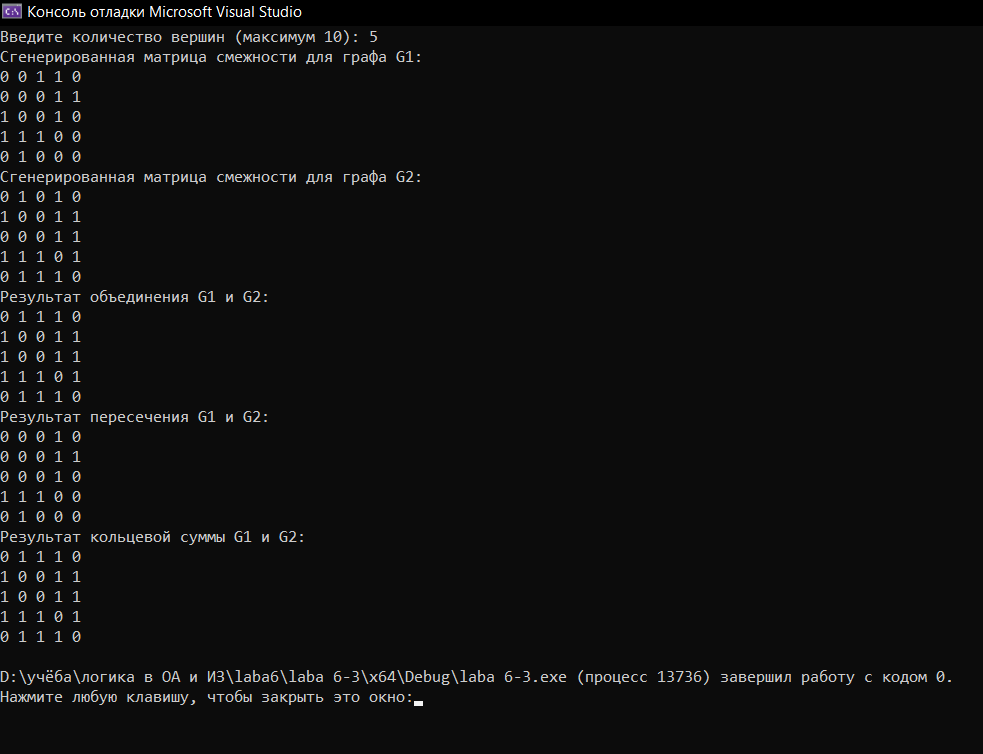


Рисунок 3 - Результат работы программы №6.

**Задание 4 \***

1. Для матричной формы представления графов выполните операцию декартова произведения графов *G = G*1X *G*2.

Результат выполнения операции выведите на экран.

**ЛИСТИНГ**

**lab6-4.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <locale.h>

// Функция для генерации матрицы смежности случайного графа

void generate\_adjacency\_matrix(int\*\* matrix, int vertices) {

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

for (int j = i; j < vertices; j++) {

if (i == j) {

matrix[i][j] = 0; // Нет петель

}

else {

matrix[i][j] = rand() % 2; // Случайное значение 0 или 1

matrix[j][i] = matrix[i][j]; // Обеспечиваем симметричность для неориентированного графа

}

}

}

}

// Функция для вывода матрицы смежности

void print\_adjacency\_matrix(int\*\* matrix, int vertices) {

printf("Матрица смежности:\n");

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

// Функция для вычисления декартова произведения графов

void cartesian\_product(int\*\* g1, int vertices1, int\*\* g2, int vertices2, int\*\*\* result, int\* result\_vertices) {

\*result\_vertices = vertices1 \* vertices2; // Общее количество вершин в графе G

\*result = (int\*\*)malloc(\*result\_vertices \* sizeof(int\*));

for (int i = 0; i < \*result\_vertices; i++) {

(\*result)[i] = (int\*)malloc(\*result\_vertices \* sizeof(int));

for (int j = 0; j < \*result\_vertices; j++) {

(\*result)[i][j] = 0; // Инициализация всех рёбер нулями

}

}

// Заполнение матрицы смежности декартова произведения

for (int i = 0; i < vertices1; i++) {

for (int j = 0; j < vertices2; j++) {

for (int k = 0; k < vertices2; k++) {

if (g2[j][k] == 1) {

(\*result)[i \* vertices2 + j][i \* vertices2 + k] = 1; // Вершины из G2 смежны

}

}

for (int k = 0; k < vertices1; k++) {

if (g1[i][k] == 1) {

(\*result)[i \* vertices2 + j][k \* vertices2 + j] = 1; // Вершины из G1 смежны

}

}

}

}

}

int main() {

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

int vertices1, vertices2;

printf("Введите количество вершин для графа G1: ");

scanf("%d", &vertices1);

printf("Введите количество вершин для графа G2: ");

scanf("%d", &vertices2);

// Выделение памяти для матриц смежности

int\*\* G1 = (int\*\*)malloc(vertices1 \* sizeof(int\*));

int\*\* G2 = (int\*\*)malloc(vertices2 \* sizeof(int\*));

for (int i = 0; i < vertices1; i++) {

G1[i] = (int\*)malloc(vertices1 \* sizeof(int));

}

for (int i = 0; i < vertices2; i++) {

G2[i] = (int\*)malloc(vertices2 \* sizeof(int));

}

// Генерация матриц смежности

generate\_adjacency\_matrix(G1, vertices1);

generate\_adjacency\_matrix(G2, vertices2);

// Вывод сгенерированных матриц

printf("\nМатрица смежности графа G1:\n");

print\_adjacency\_matrix(G1, vertices1);

printf("\nМатрица смежности графа G2:\n");

print\_adjacency\_matrix(G2, vertices2);

// Выполнение декартова произведения графов

int\*\* result;

int result\_vertices;

cartesian\_product(G1, vertices1, G2, vertices2, &result, &result\_vertices);

// Вывод результата декартова произведения

printf("\nМатрица смежности декартова произведения G = G1 X G2:\n");

print\_adjacency\_matrix(result, result\_vertices);

// Освобождение выделенной памяти

for (int i = 0; i < vertices1; i++) {

free(G1[i]);

}

for (int i = 0; i < vertices2; i++) {

free(G2[i]);

}

for (int i = 0; i < result\_vertices; i++) {

free(result[i]);

}

free(G1);

free(G2);

free(result);

return 0;

}

**Результат работы программы**

На рисунке 7 показана реализация задания №4



Рисунок 4 - Результат работы программы №7.

**Вывод**

Познакомились с унарными и бинарными операциями над графами, изучили их работу и применили их в лабораторной работе.