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

Чупраков С. В.

Пичаев И. А.

Приняли:

Митрохин М. А.  
Юрова О.В.

Пенза 2024

**Цель работы**

Приобрести навыки программирования и работы с унарными и бинарными операциями над графами, изучение их свойств.

**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.

### **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**

1. Для матричной формы представления графов выполните операцию:

а) объединения *G* = *G*1  *G*2

б) пересечения *G* = *G*1  *G*2

в) кольцевой суммы *G* = *G*1  *G*2

Результат выполнения операции выведите на экран.

**Задание 4**

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

Результат выполнения операции выведите на экран.

**Ход работы**

**Задание 1**

Для создания матриц смежности использована функция createG, которая создает матрицу размеом size на size. Для вывода использована функция printG. На основе полученных матриц смежности создаются списки смежности для этого использует структура Graph хранящая количество вершин и массив указателей на списки смежности для каждой вершины и функции edges для вызова остальных вспомогательных функций и создания конечного списка смежности, createNode для создания нового узла, addEdge для создания ребра между вершинами, createAGraph для инициализации графа.

**Задание 2**

1. Операция объединения (слияния) вершин (unionV) в которой две вершины объединяются в одну, при этом сохраняются все их связи (рёбра) с другими вершинами. После объединения одна из вершин исчезает, а её соседей нужно "перенести" к оставшейся вершине.

Алгоритм:

1. Проверка на самобъединение: Если вершины одинаковые (то есть, v1 == v2), объединение не выполняется.
2. Перенос рёбер: Все соседи вершины v2 (то есть вершины, смежные с v2) добавляются в список смежности вершины v1.
3. Удаление рёбер: Все рёбра, которые исходили от вершины v2, удаляются из списка смежности других вершин. Это нужно для того, чтобы не оставить дублирующихся рёбер после объединения.
4. Удаление вершины v2: После того как все соседи вершины v2 были перенесены в v1, вершина v2 удаляется из графа. Вершины, стоящие после v2, сдвигаются на одну позицию.
5. Уменьшение количества вершин: Общее количество вершин в графе уменьшается на 1.

2. Операция стягивания вершин (styag) позволяет объединить две вершины и все их рёбра в одну вершину. В процессе стягивания вершины теряют свою индивидуальность, но их связи остаются в графе.

Алгоритм:

1. Проверка на существование ребра между вершинами: Сначала проверяется, существуют ли рёбра между вершинами v1 и v2. Если рёбер нет, операция не выполняется.
2. Удаление рёбер между вершинами: Если ребра существуют, они удаляются из обоих списков смежности — и для v1, и для v2.
3. Перенос соседей: Все соседи вершины v2 (которые не являются самими вершинами v1 или v2) добавляются в список смежности вершины v1. При этом уже существующие рёбра проверяются, чтобы избежать дублирования.
4. Удаление вершины v2: Вершина v2 удаляется из графа, и её рёбра удаляются.

3. Операция расщепления вершины (splitV) позволяет разделить одну вершину на две новые вершины. Все рёбра, исходящие от исходной вершины, переносятся на две новые вершины.

Алгоритм:

1. Проверка на корректность вершины: Если вершина v находится за пределами допустимых вершин графа (меньше 0 или больше или равно числу вершин), операция не выполняется.
2. Добавление новых вершин: Создаются две новые вершины, которые получат новый номер.
3. Перенос рёбер: Все рёбра, исходящие от вершины v, перенаправляются на одну из двух новых вершин. Как именно рёбра будут разделяться между новыми вершинами, зависит от выбранного алгоритма. В простом варианте, это может быть произвольное разделение или перенос всех рёбер на одну новую вершину.
4. Удаление старой вершины: После расщепления старая вершина удаляется из графа.

**Задание 3**

Операция объединения двух графов аналогична объединению множеств. В контексте графов это значит, что в результате операции объединения сохраняются все рёбра, которые существуют хотя бы в одном из двух графов.

Алгоритм:

1. Определение размера нового графа: создаём новый граф, размер которого равен размеру большего из двух исходных графов: sizemax = max(size1, size2) (в этом графе будет столько вершин, сколько в большем из двух графов).
2. Создаём вспомогательные переменные: Gmax и Gmin, чтобы всегда было проще работать с более крупным и более мелким графами. Мы будем использовать эти графы для обработки данных.
3. Инициализация нового графа: создаем новый граф Gtemp с размером sizemax (это граф для объединения).
4. Заполнение новых рёбер: В цикле заполняем новый граф с использованием логической операции ИЛИ (OR). Для каждой пары вершин i и j из более мелкого графа (в диапазоне от 0 до sizemin), результат матрицы смежности для нового графа будет, что если в одном из графов есть рёбра (значение 1), то в новом графе тоже будет ребро (значение 1).
5. Копирование рёбер из большего графа. Далее, для рёбер, которые присутствуют только в более крупном графе (если sizemax больше sizemin), копируем эти рёбра в новый граф.

Важно заметить, что для неориентированных графов, граф симметричен относительно главной диагонали.

Операция пересечения графов аналогична пересечению множеств. Это означает, что в результирующем графе будут только те рёбра, которые существуют в обоих графах одновременно. То есть, если в одном из графов ребро отсутствует, то оно не попадёт в новый граф.

Алгоритм:

1. Определение размера нового графа: создаём новый граф, размер которого равен меньшему из размеров исходных графов: sizemin = min(size1, size2). Это означает, что новый граф будет иметь столько вершин, сколько в меньшем из исходных графов.
2. Инициализация нового графа: создаём новый граф Gtemp с размером sizemin (это граф для пересечения).
3. Заполнение рёбер: В цикле для каждой пары вершин i и j из меньшего графа (в диапазоне от 0 до sizemin), результат матрицы смежности для нового графа будет означает, что ребро будет присутствовать в новом графе только если оно есть в обоих графах.

Операция симметрической разности графов (XOR) работает по аналогии с симметрической разностью множеств. В графах это означает, что сохраняются только те рёбра, которые присутствуют либо в одном из графов, либо в другом, но не одновременно в обоих. Это делается с помощью операции исключающего ИЛИ (XOR).

Алгоритм:

1. Определение размера нового графа: Новый граф будет иметь размер, равный разнице в размерах двух графов: sizemax−sizeminsizemax−sizemin. Это гарантирует, что новый граф будет иметь размер, пропорциональный разнице между размерами исходных графов.
2. Инициализация нового графа: создаём новый граф Gtemp, который будет хранить результат операции XOR.
3. Заполнение рёбер: В цикле для каждой пары вершин i и j, если рёбра между этими вершинами присутствуют только в одном графе, то результат будет 1. В результате операции XOR для рёбер, которые присутствуют в обоих графах, будет 0.

**Задание 4**

Инициализация матрицы смежности нового графа: в первой части функции происходит обнуление всей матрицы смежности для нового графа G3​. Размер матрицы G3 произведению числа вершин в G1​ и G2​. Все элементы этой матрицы изначально устанавливаются в 0

Заполнение матрицы смежности нового графа:  
Далее происходит вложенный цикл, который перебирает все возможные пары вершин для нового графа:

Внешние два цикла i и j обходят вершины из G1​ и G2​.

Внутренние два цикла k и l также обходят вершины графов G1​ и G2​.

Внутри этих циклов происходит проверка: если существует ребро между вершинами i и k в G1​ и если j = l, то ставим ребро между вершинами i \* size2 + j и k \* size2 + l в новом графе G3​.

Аналогичная проверка происходит для пар j и l и i=k, для новых вершин i \* size2 + j и k \* size2 + l в новом графе G3​.

Если выполнена хотя бы одна проверка ребро в новом графе появится.

### **Результаты работы программы**

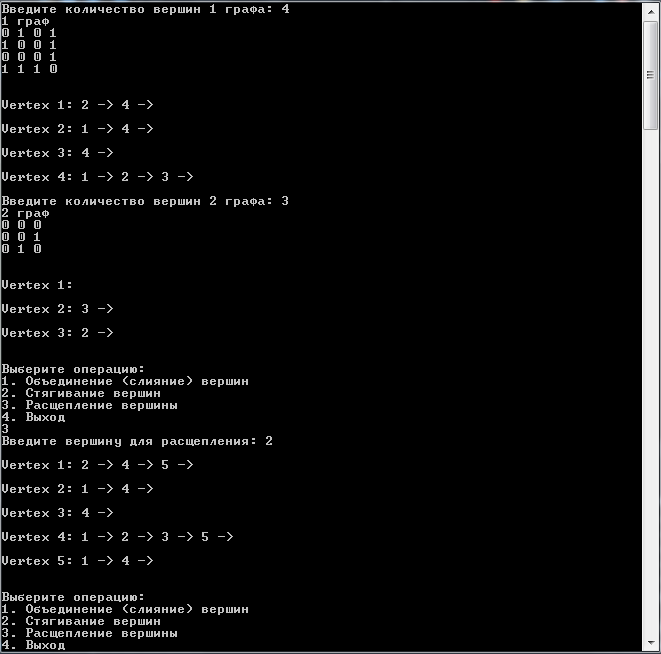


Рисунок 1 — Результаты работы программы 1

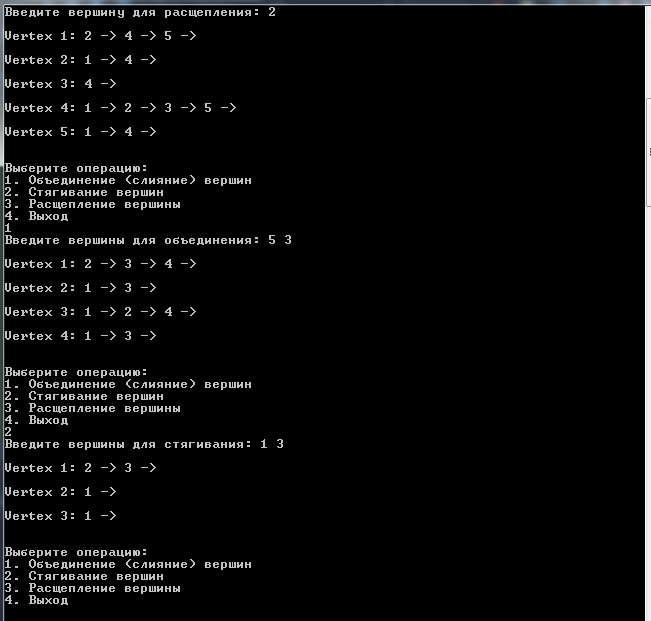


Рисунок 2 — Результаты работы программы 1

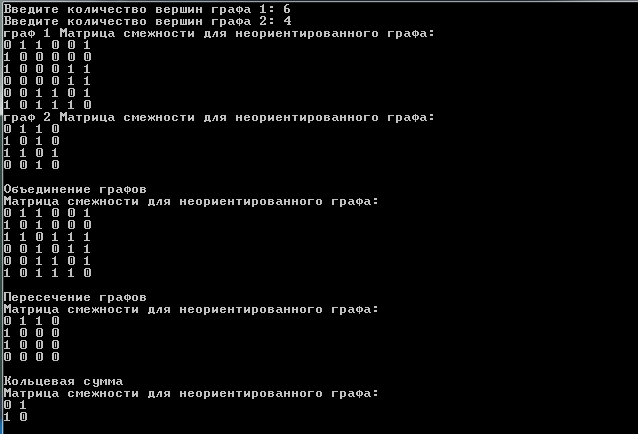


Рисунок 3 — Результаты работы программы 3

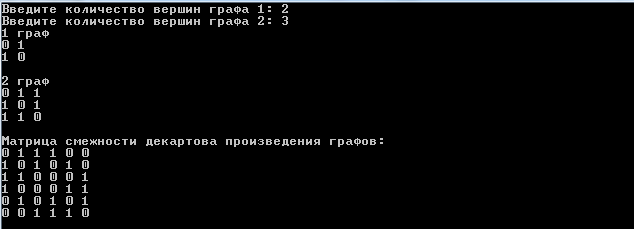


Рисунок 4 — Результаты работы программы 4

**Вывод**

Приобретены навыки программирования и работы с унарными и бинарными операциями над графами, изучены их свойства.

**Листинг**

**Задание 1-2**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "stdio.h"

#include "stdlib.h"

#include "cstdlib"

#include <windows.h>

#include <locale.h>

struct node {

int vertex;

struct node\* next;

};

struct Graph {

int numVertices;

struct node\*\* adjLists;

};

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createAGraph(int vertices) {

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

graph->numVertices = vertices;

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

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

graph->adjLists[i] = NULL;

}

return graph;

}

void addEdge(struct Graph\* graph, int s, int d) {

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

while (temp) {

if (temp->vertex == d) {

return;

}

temp = temp->next;

}

struct node\* newNode = createNode(d);

if (graph->adjLists[s] == NULL || graph->adjLists[s]->vertex > d) {

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

graph->adjLists[s] = newNode;

}

else {

temp = graph->adjLists[s];

while (temp->next && temp->next->vertex < d) {

temp = temp->next;

}

newNode->next = temp->next;

temp->next = newNode;

}

temp = graph->adjLists[d];

while (temp) {

if (temp->vertex == s) {

return;

}

temp = temp->next;

}

newNode = createNode(s);

if (graph->adjLists[d] == NULL || graph->adjLists[d]->vertex > s) {

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

graph->adjLists[d] = newNode;

}

else {

temp = graph->adjLists[d];

while (temp->next && temp->next->vertex < s) {

temp = temp->next;

}

newNode->next = temp->next;

temp->next = newNode;

}

}

struct Graph\* edges(int\*\* G, int size) {

struct Graph\* graph = createAGraph(size);

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

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

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

addEdge(graph, i, j);

}

}

}

return graph;

}

void printGraph(struct Graph\* graph) {

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

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

printf("\nVertex %d: ", v + 1);

while (temp) {

printf("%d -> ", temp->vertex + 1);

temp = temp->next;

}

printf("\n");

}

printf("\n");

}

int\*\* createG(int size) {

int\*\* G = NULL;

G = (int\*\*)malloc(size \* sizeof(int\*));

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

G[i] = (int\*)malloc(size \* sizeof(int));

}

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

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

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

if (i == j) G[i][j] = 0;

G[j][i] = G[i][j];

}

}

return G;

}

void printG(int\*\* G, int size) {

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

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

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

}

printf("\n");

}

printf("\n");

return;

}

void sortList(struct node\* head) {

if (!head) return;

int swapped;

struct node\* ptr1;

struct node\* lptr = NULL;

do {

swapped = 0;

ptr1 = head;

while (ptr1->next != lptr) {

if (ptr1->vertex > ptr1->next->vertex) {

int temp = ptr1->vertex;

ptr1->vertex = ptr1->next->vertex;

ptr1->next->vertex = temp;

swapped = 1;

}

ptr1 = ptr1->next;

}

lptr = ptr1;

} while (swapped);

}

void unionV(struct Graph\* graph, int v1, int v2) {

if (v1 == v2) return;

struct node\* temp = graph->adjLists[v2];

while (temp) {

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

temp = temp->next;

}

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

struct node\* adj = graph->adjLists[i];

struct node\* p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[i] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

}

else {

p = adj;

adj = adj->next;

}

}

}

graph->adjLists[v2] = NULL;

for (int i = v2 + 1; i < graph->numVertices; i++) {

graph->adjLists[i - 1] = graph->adjLists[i];

}

graph->adjLists[graph->numVertices - 1] = NULL;

graph->numVertices--;

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

struct node\* adj = graph->adjLists[i];

while (adj) {

if (adj->vertex > v2) {

adj->vertex--;

}

adj = adj->next;

}

}

}

void styag(struct Graph\* graph, int v1, int v2) {

if (v1 == v2) return;

int flag = 0;

struct node\* temp = graph->adjLists[v1];

while (temp) {

if (temp->vertex == v2) {

flag = 1;

}

temp = temp->next;

}

if (flag) {

struct node\* adj = graph->adjLists[v1];

struct node\* p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[v1] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

break;

}

else {

p = adj;

adj = adj->next;

}

}

adj = graph->adjLists[v2];

p = NULL;

while (adj) {

if (adj->vertex == v1) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[v2] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

break;

}

else {

p = adj;

adj = adj->next;

}

}

struct node\* temp = graph->adjLists[v2];

while (temp) {

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

temp = temp->next;

}

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

adj = graph->adjLists[i];

p = NULL;

while (adj) {

if (adj->vertex == v2) {

if (p) {

p->next = adj->next;

}

else {

graph->adjLists[i] = adj->next;

}

struct node\* del = adj;

adj = adj->next;

free(del);

}

else {

p = adj;

adj = adj->next;

}

}

}

graph->adjLists[v2] = NULL;

for (int i = v2 + 1; i < graph->numVertices; i++) {

graph->adjLists[i - 1] = graph->adjLists[i];

}

graph->adjLists[graph->numVertices - 1] = NULL;

graph->numVertices--;

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

struct node\* adj = graph->adjLists[i];

while (adj) {

if (adj->vertex > v2) {

adj->vertex--;

}

adj = adj->next;

}

}

}

else { return; }

}

void splitV(struct Graph\* graph, int v) {

if (v < 0 || v >= graph->numVertices) return;

int newV1 = graph->numVertices;

int newV2 = graph->numVertices + 1;

graph->numVertices += 1;

graph->adjLists = (struct node\*\*)realloc(graph->adjLists, graph->numVertices \* sizeof(struct node\*));

graph->adjLists[newV1] = NULL;

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

while (temp) {

if (temp->vertex != v) {

addEdge(graph, newV1, temp->vertex);

}

temp = temp->next;

}

}

int main(void) {

setlocale(LC\_ALL, "");

int sizeG1 = 0, sizeG2 = 0, choice;

int\*\* G1 = NULL;

int\*\* G2 = NULL;

printf("Введите количество вершин 1 графа: ");

scanf("%d", &sizeG1);

G1 = createG(sizeG1);

printf("1 граф\n");

printG(G1, sizeG1);

struct Graph\* graph1 = edges(G1, sizeG1);

printGraph(graph1);

printf("Введите количество вершин 2 графа: ");

scanf("%d", &sizeG2);

G2 = createG(sizeG2);

printf("2 граф\n");

printG(G2, sizeG2);

struct Graph\* graph2 = edges(G2, sizeG2);

printGraph(graph2);

while (1) {

printf("\nВыберите операцию:\n");

printf("1. Объединение (слияние) вершин\n");

printf("2. Стягивание вершин\n");

printf("3. Расщепление вершины\n");

printf("4. Выход\n");

scanf("%d", &choice);

int v1, v2;

switch (choice) {

case 1:

printf("Введите вершины для объединения: ");

scanf("%d %d", &v1, &v2);

unionV(graph1, v1 - 1, v2 - 1);

printGraph(graph1);

break;

case 2:

printf("Введите вершины для стягивания: ");

scanf("%d %d", &v1, &v2);

styag(graph1, v1 - 1, v2 - 1);

printGraph(graph1);

break;

case 3:

printf("Введите вершину для расщепления: ");

scanf("%d", &v1);

splitV(graph1, v1 - 1);

printGraph(graph1);

break;

case 4:

printf("Выход из программы.\n");

return 0;

default:

printf("Неверный выбор. Попробуйте снова.\n");

}

}

}

// #define \_CRT\_SECURE\_NO\_WARNINGS

//#include <stdio.h>

//#include <stdlib.h>

//#include <stdbool.h>

//#include <time.h>

//#include <windows.h>

//#include <locale.h>

//

//

//int\*\* createG(int size) {

//

//

// int\*\* G = NULL;

// G = (int\*\*)malloc(size \* sizeof(int\*));

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

// G[i] = (int\*)malloc(size \* sizeof(int));

// }

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

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

// G[i][j] = 0;

// }

// }

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

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

// int edge = rand() % 2;

// G[i][j] = edge;

// G[j][i] = edge; // Для неориентированного графа

// }

// }

// return G;

//

//}

//

//

//int\*\* printG(int\*\* G, int size) {

// printf("Матрица смежности для неориентированного графа:\n");

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

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

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

// }

// printf("\n");

// }

// return G;

//}

//

//

//

//

//int\*\* delv(int\*\* G, int size, int v) {

// int\*\* Gtemp = createG(size - 1);

//

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

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

// if (i < v && j < v) Gtemp[i][j] = G[i][j];

// if (i > v && j > v) Gtemp[i - 1][j - 1] = G[i][j];

// if (i > v && j < v) Gtemp[i - 1][j] = G[i][j];

// if (i < v && j > v) Gtemp[i][j - 1] = G[i][j];

//

//

// }

// }

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

// free(G[i]);

// }

// free(G);

// G = NULL;

// return Gtemp;

//}

//

//int\*\* unionv(int\*\* G, int size, int v1, int v2) {

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

// if (G[v2][i] == 1) {

// G[v1][i] = G[v2][i];

// G[i][v1] = G[i][v2];

// }

// }

// G = delv(G, size, v2);

// return G;

//

//}

//

//

//

//

//int\*\* contrE(int\*\* G, int size, int v1, int v2) {

// if (G[v1][v2] != 1) {

// printf("нет ребра");

// return G;

//

// }

// else {

// G[v1][v2] = 0;

// G[v2][v1] = 0;

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

// if (G[v2][i] == 1) {

// G[v1][i] = G[v2][i];

// G[i][v1] = G[i][v2];

// }

// }

// G = delv(G, size, v2);

// return G;

//

//

// }

//}

//

//int\*\* splitV(int\*\* G, int size, int v) {

// int\*\* Gtemp = createG(size + 1);

//

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

// if (G[v][i] == 1) {

// Gtemp[i][size] = G[i][v];

// Gtemp[size][i] = G[i][v];

//

//

// }

// }

//

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

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

// Gtemp[i][j] = G[i][j];

// }

// }

//

// Gtemp[v][v] = 0;

// Gtemp[size][size] = 0;

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

// free(G[i]);

// }

//

// free(G);

// G = NULL;

//

// return Gtemp;

//}

//

//int\*\* unionG(int\*\* G1, int\*\* G2, int size1, int size2) {

// int sizemax = (size1 > size2) ? size1 : size2;

// int sizemin = (size1 < size2) ? size1 : size2;

//

// int\*\* Gmax = (size1 > size2) ? G1 : G2;

// int\*\* Gmin = (size1 < size2) ? G1 : G2;

//

// int\*\* Gtemp = createG(sizemax);

//

// for (int i = 0; i < sizemin; i++) {

// for (int j = 0; j < sizemin; j++) {

// Gtemp[i][j] = Gmin[i][j] | Gmax[i][j];

// }

// }

// for (int i = 0; i < sizemax; i++) {

// for (int j = sizemin; j < sizemax; j++) {

// Gtemp[i][j] = Gmax[i][j];

// Gtemp[j][i] = Gtemp[i][j];

// }

// }

//

// return Gtemp;

//

//}

//

//int\*\* intersectoinG(int\*\* G1, int\*\* G2, int size1, int size2) {

// int sizemin = (size1 < size2) ? size1 : size2;

//

// int\*\* Gtemp = createG(sizemin);

//

// for (int i = 0; i < sizemin; i++) {

// for (int j = 0; j < sizemin; j++) {

// Gtemp[i][j] = G1[i][j] & G2[i][j];

// }

// }

// return Gtemp;

//}

//

//int\*\* xorG(int\*\* G1, int\*\* G2, int size1, int size2) {

// int sizemax = (size1 > size2) ? size1 : size2;

// int sizemin = (size1 < size2) ? size1 : size2;

//

// int\*\* Gmax = (size1 > size2) ? G1 : G2;

// int\*\* Gmin = (size1 < size2) ? G1 : G2;

//

// int\*\* Gtemp = createG(sizemax - sizemin);

//

// for (int i = sizemin; i < sizemax; i++) {

// for (int j = sizemin; j < sizemax; j++) {

// Gtemp[i - sizemin][j - sizemin] = Gmax[i][j];

// }

// }

// return Gtemp;

//

//}

//

//

//

//

//int main(void) {

//

// srand(54352345342);

// int sizeG1 = 3,

// sizeG2 = 5;

// setlocale(LC\_ALL, "");

//

// printf("Введите количество вершин графа 1: ");

// scanf("%d", &sizeG1);

// printf("Введите количество вершин графа 2: ");

// scanf("%d", &sizeG2);

//

// int\*\* G1 = createG(sizeG1);

// int\*\* G2 = createG(sizeG2);

// int\*\* G3 = createG(sizeG1\*sizeG2);

//

// //printf("граф 1 ");

// //printG(G1, sizeG1);

//

//

// //printf("граф 2 ");

// //printG(G2, sizeG2);

//

//

// printf("Граф 1 (матрица смежности):\n");

// printG(G1, sizeG1);

//

// printf("Граф 2 (матрица смежности):\n");

// printG(G2, sizeG2);

//

//

//

//

//

// int\*\* G3 = unionG(G1, G2, sizeG1, sizeG2);

// int sizeG3 = (sizeG1 > sizeG2) ? sizeG1 : sizeG2;

// printf("\nОбъединение графов\n");

//

// printG(G3, sizeG3);

//

// int\*\* G4 = intersectoinG(G1, G2, sizeG1, sizeG2);

// int sizeG4 = (sizeG1 < sizeG2) ? sizeG1 : sizeG2;

// printf("\nПересечение графов\n");

//

// printG(G4, sizeG4);

//

// G2 = delV(G2, g2, 2);

// nG2--;

// printf("\nГраф Godel\n");

// printf(G2, sizeG2);

// G2 = unionv(G2, sizeG2, 2, 3);

// sizeG2--;

// printG(G2, sizeG2);

// G2 = contrE(G2, sizeG2, 2, 3);

// sizeG2--;

// printG(G2, sizeG2);

// return 0;

// int\*\* G5 = xorG(G1, G2, sizeG1, sizeG2);

// int sizeG5 = (sizeG1 < sizeG2) ? sizeG2 - sizeG1 : sizeG1 - sizeG2;

// printf("\nКольцевая сумма\n");

//

// printG(G5, sizeG5);

//

//

//

//

// return 0;

//

//

//

//

//}

**Задание 3**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#include <windows.h>

#include <locale.h>

int\*\* generG(int size) {

int\*\* G = NULL;

G = (int\*\*)malloc(size \* sizeof(int\*));

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

G[i] = (int\*)malloc(size \* sizeof(int));

}

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

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

G[i][j] = 0;

}

}

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

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

int edge = rand() % 2;

G[i][j] = edge;

G[j][i] = edge; // Для неориентированного графа

}

}

return G;

}

int\*\* printG(int\*\* G, int size) {

printf("Матрица смежности для неориентированного графа:\n");

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

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

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

}

printf("\n");

}

return G;

}

int\*\* delv(int\*\* G, int size, int v) {

int\*\* Gtemp = generG(size - 1);

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

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

if (i < v && j < v) Gtemp[i][j] = G[i][j];

if (i > v && j > v) Gtemp[i - 1][j - 1] = G[i][j];

if (i > v && j < v) Gtemp[i - 1][j] = G[i][j];

if (i < v && j > v) Gtemp[i][j - 1] = G[i][j];

}

}

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

free(G[i]);

}

free(G);

G = NULL;

return Gtemp;

}

int\*\* unionG(int\*\* G1, int\*\* G2, int size1, int size2) {

int sizemax = (size1 > size2) ? size1 : size2;

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gmax = (size1 > size2) ? G1 : G2;

int\*\* Gmin = (size1 < size2) ? G1 : G2;

int\*\* Gtemp = generG(sizemax);

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

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

Gtemp[i][j] = Gmin[i][j] | Gmax[i][j];

}

}

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

for (int j = sizemin; j < sizemax; j++) {

Gtemp[i][j] = Gmax[i][j];

Gtemp[j][i] = Gtemp[i][j];

}

}

return Gtemp;

}

int\*\* intersectoinG(int\*\* G1, int\*\* G2, int size1, int size2) {

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gtemp = generG(sizemin);

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

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

Gtemp[i][j] = G1[i][j] & G2[i][j];

}

}

return Gtemp;

}

int\*\* xorG(int\*\* G1, int\*\* G2, int size1, int size2) {

int sizemax = (size1 > size2) ? size1 : size2;

int sizemin = (size1 < size2) ? size1 : size2;

int\*\* Gmax = (size1 > size2) ? G1 : G2;

int\*\* Gmin = (size1 < size2) ? G1 : G2;

int\*\* Gtemp = generG(sizemax - sizemin);

for (int i = sizemin; i < sizemax; i++) {

for (int j = sizemin; j < sizemax; j++) {

Gtemp[i - sizemin][j - sizemin] = Gmax[i][j];

}

}

return Gtemp;

}

int main(void) {

int sizeG1 = 3, sizeG2 = 5;

setlocale(LC\_ALL, "");

printf("Введите количество вершин графа 1: ");

scanf("%d", &sizeG1);

printf("Введите количество вершин графа 2: ");

scanf("%d", &sizeG2);

int\*\* G1 = generG(sizeG1);

int\*\* G2 = generG(sizeG2);

printf("граф 1 ");

printG(G1, sizeG1);

printf("граф 2 ");

printG(G2, sizeG2);

int\*\* G3 = unionG(G1, G2, sizeG1, sizeG2);

int sizeG3 = (sizeG1 > sizeG2) ? sizeG1 : sizeG2;

printf("\nОбъединение графов\n");

printG(G3, sizeG3);

int\*\* G4 = intersectoinG(G1, G2, sizeG1, sizeG2);

int sizeG4 = (sizeG1 < sizeG2) ? sizeG1 : sizeG2;

printf("\nПересечение графов\n");

printG(G4, sizeG4);

int\*\* G5 = xorG(G1, G2, sizeG1, sizeG2);

int sizeG5 = (sizeG1 < sizeG2) ? sizeG2 - sizeG1 : sizeG1 - sizeG2;

printf("\nКольцевая сумма\n");

printG(G5, sizeG5);

return 0;

}

**Задание 4**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "stdio.h"

#include "cstdlib"

#include <windows.h>

#include <locale.h>

int\*\* createG(int size) {

int\*\* G = NULL;

G = (int\*\*)malloc(size \* sizeof(int\*));

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

G[i] = (int\*)malloc(size \* sizeof(int));

}

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

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

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

if (i == j) G[i][j] = 0;

G[j][i] = G[i][j];

}

}

return G;

}

void printG(int\*\* G, int size) {

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

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

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

}

printf("\n");

}

printf("\n");

return;

}

int\*\* decUG(int\*\* G1, int size1, int\*\* G2, int size2, int\*\* G3) {

for (int i = 0; i < size1 \* size2; i++) {

for (int j = 0; j < size1 \* size2; j++) {

G3[i][j] = 0;

}

}

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

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

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

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

if (G1[i][k] == 1 && j == l) {

G3[i \* size2 + j][k \* size2 + l] = 1;

}

if (G2[j][l] == 1 && i == k) {

G3[i \* size2 + j][k \* size2 + l] = 1;

}

}

}

}

}

return G3;

}

int main(void) {

setlocale(LC\_ALL, "");

int sizeG1 = 0, sizeG2 = 0, sizeG3 = 0;

int\*\* G1 = NULL;

int\*\* G2 = NULL;

int\*\* G3 = NULL;

srand(1235235235);

printf("Введите количество вершин графа 1: ");

scanf("%d", &sizeG1);

printf("Введите количество вершин графа 2: ");

scanf("%d", &sizeG2);

G1 = createG(sizeG1);

printf("1 граф\n");

printG(G1, sizeG1);

G2 = createG(sizeG2);

printf("2 граф\n");

printG(G2, sizeG2);

G3 = createG(sizeG1 \* sizeG2);

G3 = decUG(G1, sizeG1, G2, sizeG2, G3);

printf("Матрица смежности декартова произведения графов:\n");

printG(G3, sizeG1 \* sizeG2);

}