Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

по лабораторной работе №6

по курсу «Л и О А в ИЗ»

на тему «Унарные и бинарные операции над графами»

Выполнили студенты группы 24ВВВ4:

Кошелев Р.Д.

Кондратьев С.В.

Приняли к.т.н., доцент:

Юрова О.В.

к.э.н., доцент:

Акифьев И.В.

Пенза 2025

Цель работы: создать два неориентированных помеченных графа, представленных в виде матриц смежности, и преобразовать их в списки смежности. Реализовать базовые операции над графами (отождествление вершин, стягивание ребра, расщепление вершины) как для матричной, так и для списковой формы представления. Выполнить операции объединения, пересечения и кольцевой суммы для двух графов в матричной форме представления.

Общие сведения.

Все унарные операции над графами можно объединить в две группы. Первую группу составляют операции, с помощью которых из исходного графа G1, можно построить граф G2 с меньшим числом элементов. В группу входят операции удаления ребра или вершины, отождествления вершин, стягивание ребра. Вторую группу составляют операции, позволяющие строить графы с большим числом элементов. В группу входят операции расщепления вершин, добавления ребра.

Отождествление вершин. В графе G1 выделяются вершины u,v. Определяют окружение Q1 вершины u, и окружение Q2 вершины v, вычисляют их объединение Q = Q1 Q2. Затем над графом G1 выполняются следующие преобразования:

— из графа G1 удаляют вершины u, v (H1 = G1 - u - v);

— к графу H1 присоединяют новую вершину z (H1 = H1 +z);

— вершину z соединяют ребром с каждой из вершин w1∈Q

(G2 = H1 + zwi, i = 1,2,3,…).

Стягивание ребра. Данная операция является операцией отождествления смежных вершин u, v в графе G1.

Наиболее важными бинарными операциями являются: объединение, пересечение, декартово произведение и кольцевая сумма.

Объединение. Граф G называется объединением или наложением графов G1 и G2, если VG = V1 V2; UG = U1 U2 (рис. 1).

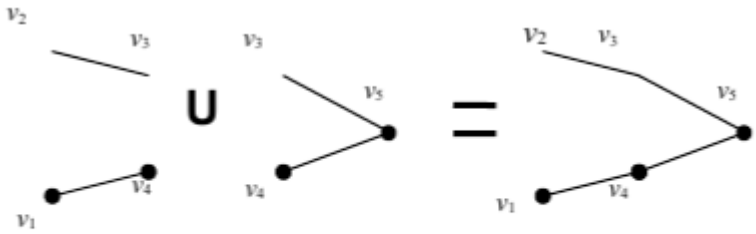

Рис. 1. Объединение графов G_1, G_2

Объединение графов G1 и G2 называется дизъюнктным, если V1 V2 = ∅. При дизъюнктном объединении никакие два из объединяемых графов не должны иметь общих вершин.

Пересечение. Граф G называется пересечением графов G1, G2, если VG = V2 и UG = U1 U2 (рис.2). Операция "пересечения" записывается следующим образом: G = G1 G2.

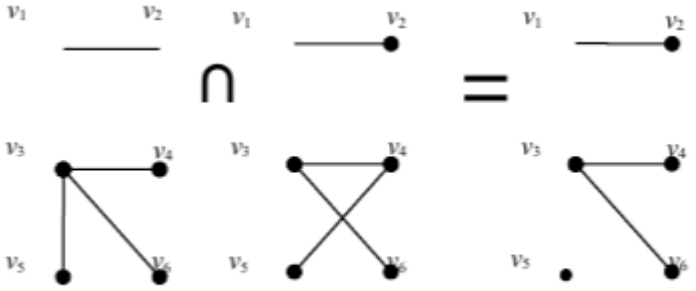

Рис.2. Пересечение графов G_1, G_2.

Декартово произведение. Граф G называется декартовым произведением графов G1 и G2 если VG = V1×V2 —декартово произведение множеств вершин графов G1, G2, а множество ребер Uc задается следующим образом: вершины (zi, vk) и (zj, vl) смежны в графе G тогда и только тогда, когда zi = zj(i = j), а vk и vl смежны в G2 или vk = vl(k = l), смежны в графе G1 (см. рис.3).

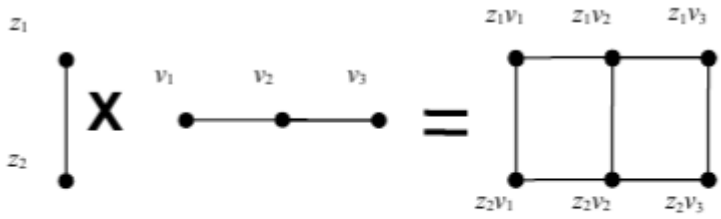

Рис. 3. Декартово произведение графов G_1, G_2

Кольцевая сумма графов представляет граф, который не имеет изолированных вершин и состоит из ребер, присутствующих либо в первом исходном графе, либо во втором. Кольцевая сумма определяется следующим соотношением: G = G1 ⊕ G2 (рис.4).

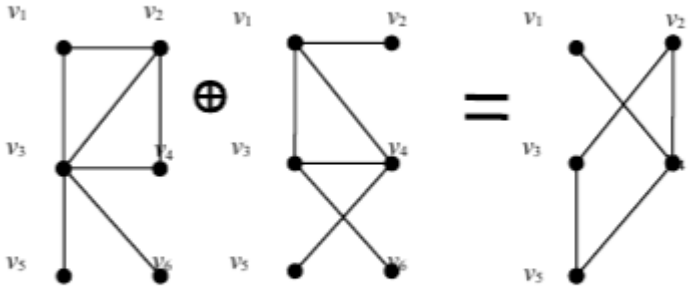

Рис.4. Кольцевая сумма графов G_1, G_2

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) две матрицы M1, M2 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.

2. * 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 = G1 G2

б) пересечения G = G1 G2

в) кольцевой суммы G = G1 $\oplus$ G2

Результат выполнения операции выведите на экран.

Задание 4 *

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

Результат выполнения операции выведите на экран.

Результат работы программы

```
=== Главное меню ===
1 – Создать/пересоздать графы G1 и G2
2 – Работа с одним графом (просмотр/отождествление/стягивание/расщепление)
3 – Операции над двумя графами (объединение/пересечение/XOR/декартово произведение)
4 – Показать текущие матрицы
5 – Выход
Ваш выбор:
```

Рисунок 1 – Главное меню

```
Ваш выбор: 1
Введите число вершин для G1: 2
Введите число вершин для G2: 3
Матрица M1 (G1) (n = 2):
1 1
1 1

Матрица M2 (G2) (n = 3):
1 0 0
0 1 0
0 0 0
```

Рисунок 2 – Создание неориентированных графов

```
=== Главное меню ===
1 – Создать/пересоздать графы G1 и G2
2 – Работа с одним графом (просмотр/отождествление/стягивание/расщепление)
3 – Операции над двумя графами (объединение/пересечение/XOR/декартово произведение)
4 – Показать текущие матрицы
5 – Выход
Ваш выбор: 2
С каким графом хотите работать? (1 или 2): 1

--- Работа с графом 1 (n = 2) ---
1 – Показать матрицу
2 – Показать список смежности
3 – Отождествление вершин
4 – Стягивание ребра
5 – Расщепление вершины
6 – Вернуться в главное меню
Ваш выбор:
```

Рисунок 3 – Выбор режима работы с одним графом

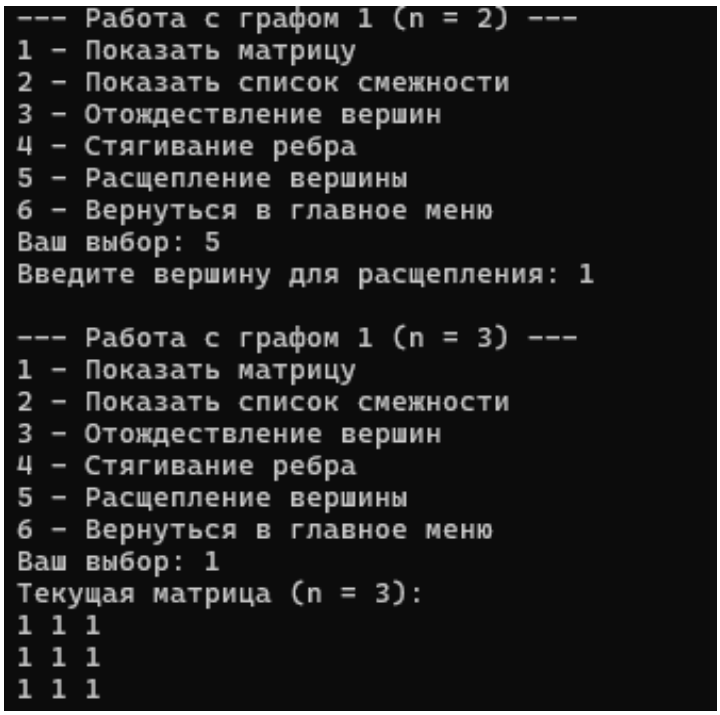

Рисунок 4 – Операция расщепления вершины

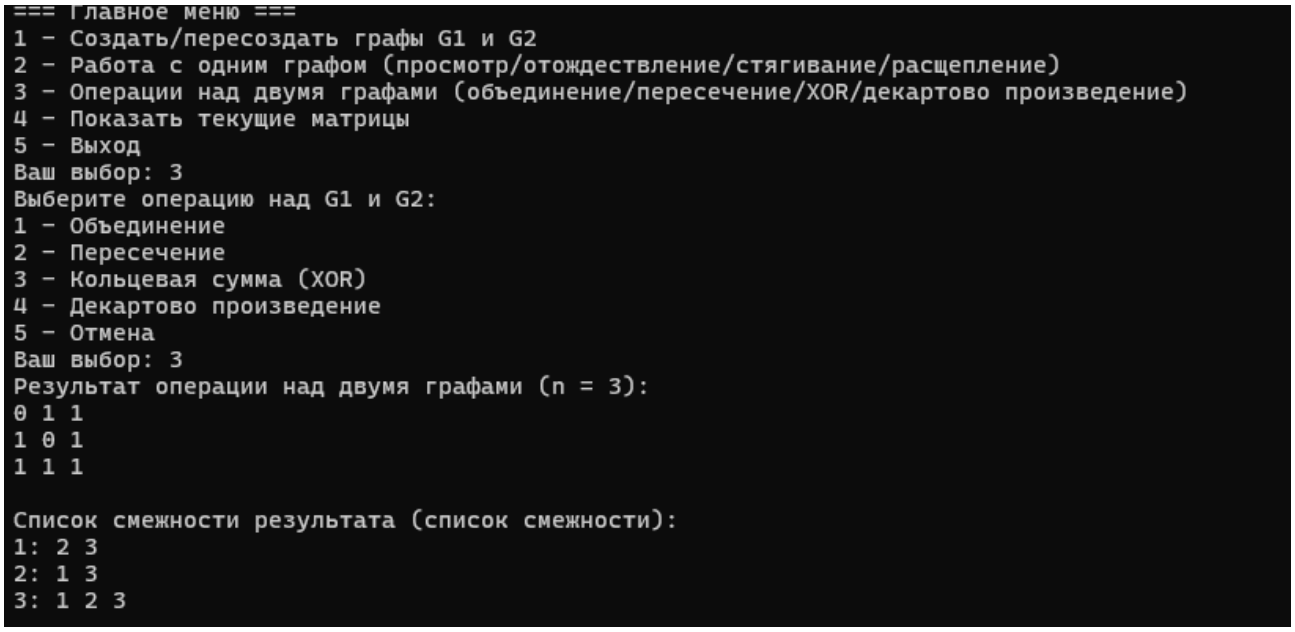

Рисунок 5 – Режим работы с двумя графами и операция кольцевой суммы на ними

Вывод: в ходе выполнения работы были успешно реализованы и протестированы основные операции теории графов в различных формах представления.

Приложение А

Листинг

```
#define _CRT_SECURE_NO_WARNINGS
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <string.h>

/* Проверка, что строка состоит только из цифр */
int is_valid_int(const char* s) {
    if (!s || *s == '\0') return 0;
    while (*s) {
        if (!isdigit((unsigned char)*s)) return 0;
        s++;
    }
    return 1;
}

/* Считывание положительного целого числа с проверкой */
int read_positive_int(const char* prompt) {
    char buffer[128];
    long n;

    while (1) {
        printf("%s", prompt);
```

```
        if (!fgets(buffer, sizeof(buffer), stdin)) {
            fprintf(stderr, "Ошибка ввода!\n");
            exit(EXIT_FAILURE);
        }

        buffer[strcspn(buffer, "\n")] = '\0'; // убрать \n

        if (!is_valid_int(buffer)) {
            printf("Ошибка: введите положительное целое число!\n");
            continue;
        }

        n = atol(buffer);
        if (n <= 0) {
            printf("Ошибка: число должно быть больше нуля!\n");
            continue;
        }
        if (n > 1000000) { // защита от нереально большого ввода
            printf("Ошибка: слишком большое число вершин!\n");
            continue;
        }
        return (int)n;
    }
}

/* Генерация n x n матрицы (симметричной), включая петли */
int** generate_matrix(int n) {
```

```
    int i, j;
    int** a = (int**)malloc(n * sizeof(int*));
    if (!a) { perror("malloc"); exit(EXIT_FAILURE); }

    for (i = 0; i < n; ++i) {
        a[i] = (int*)calloc(n, sizeof(int));
        if (!a[i]) { perror("calloc"); exit(EXIT_FAILURE); }
    }

    for (i = 0; i < n; ++i) {
        for (j = i; j < n; ++j) {
            int edge = rand() % 2;
            a[i][j] = edge;
            a[j][i] = edge;
        }
    }
    return a;
}

/* Печать матрицы n x n */
void print_matrix(int** a, int n, const char* name) {
    printf("%s (n = %d):\n", name, n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            printf("%d ", a[i][j]);
        printf("\n");
    }
```

```
    printf("\n");
}

/* Преобразование матрицы в список смежности */
int** matrix_to_adjlist(int** a, int n, int** out_deg) {
    int* deg = (int*)calloc(n, sizeof(int));
    if (!deg) { perror("calloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (a[i][j]) deg[i]++;

    int** adj = (int**)malloc(n * sizeof(int*));
    if (!adj) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < n; ++i) {
        adj[i] = deg[i] ? (int*)malloc(deg[i] * sizeof(int)) : NULL;
        int pos = 0;
        for (int j = 0; j < n; ++j)
            if (a[i][j]) adj[i][pos++] = j + 1;
    }

    *out_deg = deg;
    return adj;
}

/* Печать списка смежности */
```

```
void print_adjlist(int** adj, int* deg, int n, const char* name) {
    printf("%s (список смежности):\n", name);
    for (int i = 0; i < n; ++i) {
        printf("%d:", i + 1);
        if (deg[i] == 0) printf(" (нет соседей)");
        else for (int k = 0; k < deg[i]; ++k) printf(" %d", adj[i][k]);
        printf("\n");
    }
    printf("\n");
}

/* Освобождение памяти */
void free_matrix(int** a, int n) {
    if (!a) return;
    for (int i = 0; i < n; ++i) {
        if (a[i]) free(a[i]);
    }
    free(a);
}

void free_adjlist(int** adj, int* deg, int n) {
    if (!adj) return;
    for (int i = 0; i < n; ++i) if (adj[i]) free(adj[i]);
    free(adj);
    free(deg);
}
```

```
/* (a) Отождествление вершин (v1 и v2 — 1-based) */
int** merge_vertices(int** a, int* n, int v1, int v2) {
    if (!a) return NULL;
    v1--; v2--;
    if (v1 < 0 || v2 < 0 || v1 >= *n || v2 >= *n) return a;
    if (v1 == v2) return a;

    int new_n = *n - 1;
    int** b = (int**)malloc(new_n * sizeof(int*));
    if (!b) { perror("malloc"); exit(EXIT_FAILURE); }
    for (int i = 0; i < new_n; ++i) {
        b[i] = (int*)calloc(new_n, sizeof(int));
        if (!b[i]) { perror("calloc"); exit(EXIT_FAILURE); }
    }

    for (int i = 0, bi = 0; i < *n; ++i) {
        if (i == v2) continue;
        for (int j = 0, bj = 0; j < *n; ++j) {
            if (j == v2) continue;
            if (i == v1 || j == v1) {
                int val;
                /* Если строка/столбец содержит v1 или v2 — объединяем рёбра */
                if (i == v1 && j == v1) {
                    /* петля на объединённой вершине: если была петля у v1 или v2 */
                    val = (a[v1][v1] || a[v2][v2] || a[v1][v2]) ? 1 : 0;
                }
                else {
```

```
                int orig_j = (j == v1) ? v1 : j;
                /* Проверяем связи от объединённой вершины к orig_j */
                val = (a[v1][j] || a[v2][j]) ? 1 : 0;
            }
            b[bi][bj] = val;
        }
        else {
            b[bi][bj] = a[i][j];
        }
        bj++;
    }
    bi++;
  }

  free_matrix(a, *n);
  *n = new_n;
  return b;
}

/* (б) Стягивание ребра (v1 и v2 — 1-based) */
int** contract_edge(int** a, int* n, int v1, int v2) {
  if (!a) return NULL;
  v1--; v2--;
  if (v1 < 0 || v2 < 0 || v1 >= *n || v2 >= *n) return a;
  if (a[v1][v2] == 0) {
    printf("Ребро между вершинами %d и %d отсутствует!\n", v1 + 1, v2 + 1);
    return a;
```

```
    }
    return merge_vertices(a, n, v1 + 1, v2 + 1);
}

int** split_vertex(int** a, int* n, int v1) {
    if (!a) return NULL;
    v1--;
    if (v1 < 0 || v1 >= *n) return a;

    int old_n = *n;
    int new_n = old_n + 1;

    // создаём новую матрицу
    int** b = (int**)malloc(new_n * sizeof(int*));
    if (!b) { perror("malloc"); exit(EXIT_FAILURE); }
    for (int i = 0; i < new_n; ++i) {
        b[i] = (int*)calloc(new_n, sizeof(int));
        if (!b[i]) { perror("calloc"); exit(EXIT_FAILURE); }
    }

    // копируем старую матрицу в верхний левый угол
    for (int i = 0; i < old_n; ++i)
        for (int j = 0; j < old_n; ++j)
            b[i][j] = a[i][j];

    int new_v = new_n - 1; // индекс новой вершины
```

```
    // новая вершина наследует все связи исходной
    for (int j = 0; j < old_n; ++j) {
        if (a[v1][j]) {
            b[new_v][j] = 1;
            b[j][new_v] = 1;
        }
    }

    // добавляем связь между исходной и новой вершинами
    b[v1][new_v] = 1;
    b[new_v][v1] = 1;

    // если у исходной вершины была петля — дублируем и у новой
    if (a[v1][v1]) {
        b[new_v][new_v] = 1;
        b[v1][v1] = 1;
    }

    free_matrix(a, old_n);
    *n = new_n;
    return b;
}

/* Дополнение меньшей матрицы нулями до new_n x new_n */
int** resize_matrix(int** A, int old_n, int new_n) {
    int** B = (int**)malloc(new_n * sizeof(int*));
    if (!B) { perror("malloc"); exit(EXIT_FAILURE); }
```

```
    for (int i = 0; i < new_n; ++i) {
        B[i] = (int*)calloc(new_n, sizeof(int));
        if (!B[i]) { perror("calloc"); exit(EXIT_FAILURE); }
        if (i < old_n) {
            for (int j = 0; j < old_n; ++j)
                B[i][j] = A[i][j];
        }
    }
    return B;
}

/* --- Функции для операций над двумя графами --- */
int** union_graph(int** G1, int n1, int** G2, int n2, int* out_n) {
    int new_n = (n1 > n2) ? n1 : n2;
    int** A = resize_matrix(G1, n1, new_n);
    int** B = resize_matrix(G2, n2, new_n);
    int** G = (int**)malloc(new_n * sizeof(int*));
    if (!G) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < new_n; ++i) {
        G[i] = (int*)malloc(new_n * sizeof(int));
        if (!G[i]) { perror("malloc"); exit(EXIT_FAILURE); }
        for (int j = 0; j < new_n; ++j)
            G[i][j] = (A[i][j] || B[i][j]) ? 1 : 0;
    }

    free_matrix(A, new_n);
```

```
    free_matrix(B, new_n);
    *out_n = new_n;
    return G;
}

int** intersection_graph(int** G1, int n1, int** G2, int n2, int* out_n) {
    int new_n = (n1 > n2) ? n1 : n2;
    int** A = resize_matrix(G1, n1, new_n);
    int** B = resize_matrix(G2, n2, new_n);
    int** G = (int**)malloc(new_n * sizeof(int*));
    if (!G) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < new_n; ++i) {
        G[i] = (int*)malloc(new_n * sizeof(int));
        if (!G[i]) { perror("malloc"); exit(EXIT_FAILURE); }
        for (int j = 0; j < new_n; ++j)
            G[i][j] = (A[i][j] && B[i][j]) ? 1 : 0;
    }

    free_matrix(A, new_n);
    free_matrix(B, new_n);
    *out_n = new_n;
    return G;
}

int** xor_graph(int** G1, int n1, int** G2, int n2, int* out_n) {
    int new_n = (n1 > n2) ? n1 : n2;
```

```
    int** A = resize_matrix(G1, n1, new_n);
    int** B = resize_matrix(G2, n2, new_n);
    int** G = (int**)malloc(new_n * sizeof(int*));
    if (!G) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < new_n; ++i) {
        G[i] = (int*)malloc(new_n * sizeof(int));
        if (!G[i]) { perror("malloc"); exit(EXIT_FAILURE); }
        for (int j = 0; j < new_n; ++j)
            G[i][j] = (A[i][j] != B[i][j]) ? 1 : 0;
    }

    free_matrix(A, new_n);
    free_matrix(B, new_n);
    *out_n = new_n;
    return G;
}

/* Декартово произведение графов */
int** cartesian_product(int** G1, int n1, int** G2, int n2, int* out_n) {
    int new_n = n1 * n2;
    int** G = (int**)malloc(new_n * sizeof(int*));
    if (!G) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < new_n; ++i) {
        G[i] = (int*)calloc(new_n, sizeof(int));
        if (!G[i]) { perror("calloc"); exit(EXIT_FAILURE); }
```

```
}

for (int u1 = 0; u1 < n1; ++u1) {
    for (int u2 = 0; u2 < n2; ++u2) {
        int u = u1 * n2 + u2;

        /* Соединяем с соседями по G1 (фиксируем u2, меняем v1) */
        for (int v1 = 0; v1 < n1; ++v1) {
            if (G1[u1][v1]) {  // теперь учитываем и петли
                int v = v1 * n2 + u2;
                G[u][v] = G1[u1][v1];
                G[v][u] = G1[u1][v1];
            }
        }

        /* Соединяем с соседями по G2 (фиксируем u1, меняем v2) */
        for (int v2 = 0; v2 < n2; ++v2) {
            if (G2[u2][v2]) {  // теперь учитываем и петли
                int v = u1 * n2 + v2;
                G[u][v] = G2[u2][v2];
                G[v][u] = G2[u2][v2];
            }
        }

        /* Петли: если есть петля в G1[u1][u1] или G2[u2][u2] */
        if (G2[u2][u2] == 1) {
            G[u][u] = G2[u2][u2];
```

```
            }
        }
    }

    *out_n = new_n;
    return G;
}

/* Основная программа: интерактивное меню, можно делать операции бесконечно */
int main(void) {
    srand((unsigned)time(NULL));
    setlocale(LC_ALL, "rus");

    int n1 = 0, n2 = 0;
    int** M1 = NULL;
    int** M2 = NULL;

    while (1) {
        printf("=== Главное меню ===\n");
        printf("1 - Создать/пересоздать графы G1 и G2\n");
        printf("2 - Работа с одним графом (просмотр/отождествление/стягивание/расщепление)\n");
        printf("3 - Операции над двумя графами (объединение/пересечение/XOR/декартово произведение)\n");
        printf("4 - Показать текущие матрицы\n");
        printf("5 - Выход\n");
```

```
        int choice = read_positive_int("Ваш выбор: ");


        if (choice == 1) {
            if (M1) { free_matrix(M1, n1); M1 = NULL; n1 = 0; }
            if (M2) { free_matrix(M2, n2); M2 = NULL; n2 = 0; }
            n1 = read_positive_int("Введите число вершин для G1: ");
            n2 = read_positive_int("Введите число вершин для G2: ");
            M1 = generate_matrix(n1);
            M2 = generate_matrix(n2);
            print_matrix(M1, n1, "Матрица M1 (G1)");
            print_matrix(M2, n2, "Матрица M2 (G2)");
        }
        else if (choice == 2) {
            if (!M1 && !M2) {
                printf("Графы ещё не созданы. Выберите пункт 1 для создания
графов.\n");
                continue;
            }
            int g_choice = read_positive_int("С каким графом хотите работать? (1 или
2): ");
            if (g_choice != 1 && g_choice != 2) { printf("Некорректный выбор
графа.\n"); continue; }
            int** M = (g_choice == 2) ? M2 : M1;
            int* n = (g_choice == 2) ? &n2 : &n1;
            if (!M) { printf("Выбранный граф не создан.\n"); continue; }


            while (1) {
                printf("\n--- Работа с графом %d (n = %d) ---\n", g_choice, *n);
```

```
printf("1 - Показать матрицу\n");
printf("2 - Показать список смежности\n");
printf("3 - Отождествление вершин\n");
printf("4 - Стягивание ребра\n");
printf("5 - Расщепление вершины\n");
printf("6 - Вернуться в главное меню\n");

int op = read_positive_int("Ваш выбор: ");
if (op == 1) {
    print_matrix(M, *n, "Текущая матрица");
}
else if (op == 2) {
    int* deg = NULL;
    int** adj = matrix_to_adjlist(M, *n, &deg);
    print_adjlist(adj, deg, *n, "Список смежности");
    free_adjlist(adj, deg, *n);
}
else if (op == 3) {
    int v1 = read_positive_int("Введите первую вершину: ");
    int v2 = read_positive_int("Введите вторую вершину: ");
    if (v1 == v2 || v1 > *n || v2 > *n) printf("Некорректные вершины!\n");
    else {
        M = merge_vertices(M, n, v1, v2);
        if (g_choice == 1) M1 = M; else M2 = M;
    }
}
else if (op == 4) {
```

```
            int v1 = read_positive_int("Введите первую вершину ребра: ");
            int v2 = read_positive_int("Введите вторую вершину ребра: ");
            if (v1 == v2 || v1 > *n || v2 > *n) printf("Некорректные вершины!\n");
            else {
                M = contract_edge(M, n, v1, v2);
                if (g_choice == 1) M1 = M; else M2 = M;
            }
        }
        else if (op == 5) {
            int v = read_positive_int("Введите вершину для расщепления: ");
            if (v > *n) printf("Некорректная вершина!\n");
            else {
                M = split_vertex(M, n, v);
                if (g_choice == 1) M1 = M; else M2 = M;
            }
        }
        else if (op == 6) {
            break;
        }
        else {
            printf("Неизвестная операция!\n");
        }
    }
}
else if (choice == 3) {
    if (!M1 || !M2) { printf("Оба графа должны быть созданы (пункт 1).\n");
continue; }
```

```
    int res_n = 0;
    printf("Выберите операцию над G1 и G2:\n");
    printf("1 - Объединение\n");
    printf("2 - Пересечение\n");
    printf("3 - Кольцевая сумма (XOR)\n");
    printf("4 - Декартово произведение\n");
    printf("5 - Отмена\n");
    int op2 = read_positive_int("Ваш выбор: ");
    int** G = NULL;

    if (op2 == 1) G = union_graph(M1, n1, M2, n2, &res_n);
    else if (op2 == 2) G = intersection_graph(M1, n1, M2, n2, &res_n);
    else if (op2 == 3) G = xor_graph(M1, n1, M2, n2, &res_n);
    else if (op2 == 4) G = cartesian_product(M1, n1, M2, n2, &res_n);
    else { printf("Отмена или неизвестная операция.\n"); }

    if (G) {
        print_matrix(G, res_n, "Результат операции над двумя графами");
        int* deg = NULL;
        int** adj = matrix_to_adjlist(G, res_n, &deg);
        print_adjlist(adj, deg, res_n, "Список смежности результата");
        free_adjlist(adj, deg, res_n);
        free_matrix(G, res_n);
    }
}
else if (choice == 4) {
```

```
            if (M1) print_matrix(M1, n1, "Матрица M1 (G1)"); else printf("M1 не создана\n");
            if (M2) print_matrix(M2, n2, "Матрица M2 (G2)"); else printf("M2 не создана\n");
        }
        else if (choice == 5) {
            printf("Выход...\n");
            break;
        }
        else {
            printf("Неизвестный режим работы!\n");
        }
    }

    if (M1)
        free_matrix(M1, n1);
    if (M2) free_matrix(M2, n2);
    return 0;
}
```