Министерство науки и высшего 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 24ВВВ3
Азаров М.С.
Кукушкин А.А.

Приняли:
к.т.н., доцент Юрова О.В.
к.т.н., Деев М.В.

Пенза 2025

Цель работы – Изучение и практическое освоение унарных и бинарных операций над графами, включая операции, уменьшающие или увеличивающие число элементов графа, такие как удаление и добавление вершин и рёбер, отождествление и расщепление вершин, стягивание рёбер.

Общие сведения.

Все унарные операции над графами можно объединить в две группы. Первую группу составляют операции, с помощью которых из исходного графа G_1, можно построить граф G_2 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 G_1 выделяются вершины u,v. Определяют окружение Q_1 вершины u, и окружение Q_2 вершины v, вычисляют их объединение $Q = Q_1 \ Q_2$. Затем над графом G_1 выполняются следующие преобразования:

— из графа G_1 удаляют вершины u, v $(H_1 = G_1 - u - v)$;

— к графу H_1 присоединяют новую вершину z $(H_1 = H_1 + z)$;

— вершину z соединяют ребром с каждой из вершин $w_1 \in Q$

$(G_2 = H_1 + zw_i, i = 1,2,3,\ldots)$.

Стягивание ребра. Данная операция является операцией отождествления смежных вершин u, v в графе G_1.

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

Объединение. Граф G называется объединением или наложением графов G_1 и G_2, если $V_G = V_1 V_2$; $U_G = U_1 \ U_2$ (рис. 1).

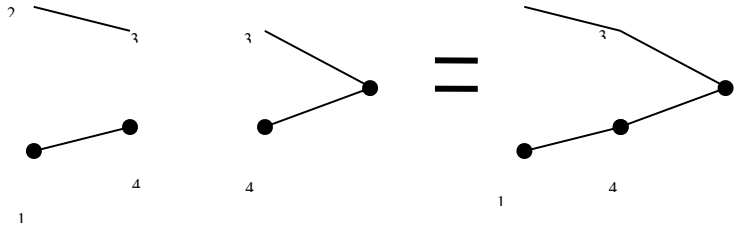

Рис. 1. Объединение графов G_1, G_2

Объединение графов G_1 и G_2 называется дизъюнктным, если $V_1V_2 = \emptyset$. При дизъюнктном объединении никакие два из объединяемых графов не должны иметь общих вершин.

Пересечение. Граф G называется пересечением графов G_1, G_2, если $V_G = V_1V_2$ и $U_G = U_1U_2$ (рис.2). Операция "пересечения" записывается следующим образом: $G = G_1G_2$.

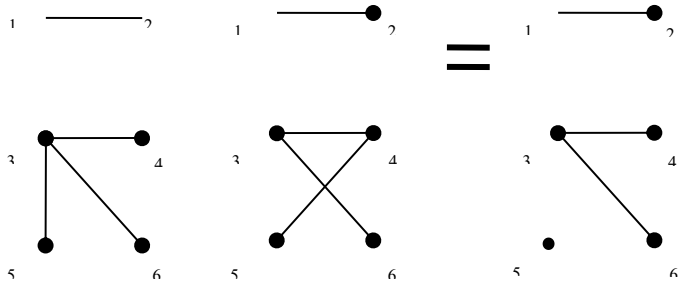

Рис.2. Пересечение графов G_1, G_2.

Декартово произведение. Граф G называется декартовым произведением графов G_1 и G_2 если $V_G = V_1 \times V_2$ —декартово произведение множеств вершин графов G_1, G_2, а множество ребер U_c задается следующим образом: вершины (z_i, v_k) и (z_j, v_l) смежны в графе G тогда и только тогда, когда $z_i = z_j (i = j)$, а v_k и v_l смежны в G_2 или $v_k = v_l (k = l)$, смежны в графе G_1 (см. рис.3).

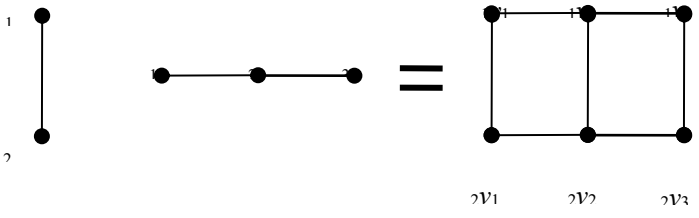

Рис. 3. Декартово произведение графов G_1, G_2

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: $G = G_1 \oplus G_2$ (рис.4).

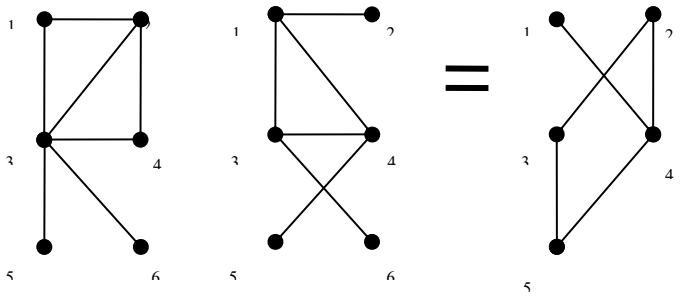

Рис.4. Кольцевая сумма графов G_1, G_2

Лабораторное задание:

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) две матрицы M_1, M_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.

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 $G = G_1 \quad G_2$

б) пересечения $G = G_1 \quad G_2$

в) кольцевой суммы $G = G_1 \oplus G_2$

Результат выполнения операции выведите на экран.

Задание 4 *

1. Для матричной формы представления графов выполните операцию декартова произведения графов $G = G_1 \times G_2$.

Результат выполнения операции выведите на экран.

Примечание: задания, помеченные символом * выполняются по указанию преподавателя.

Задание 1.
Код программы

C

```
    #define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

#define MAX_VERTICES 11
#define MAX_MATRIX (MAX_VERTICES * MAX_VERTICES)

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    int numVertices;
    Node** adjLists;
} Graph;

void generateRandomMatrix(int matrix[MAX_MATRIX][MAX_MATRIX], int n, int
probability) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (i == j) {
                matrix[i][j] = 0;
            }
            else {
                matrix[i][j] = matrix[j][i] = (rand() % 100 < probability) ?
1 : 0;
            }
        }
    }
}

void printMatrix(int matrix[MAX_MATRIX][MAX_MATRIX], int n, const char* name)
{
    printf("%s (%dx%d):\n", name, n, n);
    printf("    ");
    for (int i = 0; i < n; i++) printf("%2d ", i);
    printf("\n");
    for (int i = 0; i < n; i++) {
        printf("%2d: ", i);
        for (int j = 0; j < n; j++) {
            printf("%2d ", matrix[i][j]);
        }
```

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

Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;
    graph->adjLists = (Node**)malloc(vertices * sizeof(Node*));
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = dest;
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = src;
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void printAdjList(Graph* graph, const char* name) {
    printf("%s (spiski smezhnosti):\n", name);
    for (int i = 0; i < graph->numVertices; i++) {
        printf("%d: ", i);
        Node* temp = graph->adjLists[i];
        while (temp) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
    printf("\n");
}

void freeGraph(Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        Node* current = graph->adjLists[i];
        while (current) {
            Node* next = current->next;
            free(current);
            current = next;
        }
    }
    free(graph->adjLists);
    free(graph);
```

```
    }

    void matrixToAdjList(int matrix[MAX_MATRIX][MAX_MATRIX], int n, Graph* graph)
{
        for (int i = 0; i < graph->numVertices; i++) {
            Node* current = graph->adjLists[i];
            while (current) {
                Node* next = current->next;
                free(current);
                current = next;
            }
            graph->adjLists[i] = NULL;
        }
        graph->numVertices = n;

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (matrix[i][j] == 1) {
                    addEdge(graph, i, j);
                }
            }
        }
    }

    void identifyVerticesMatrix(int matrix[MAX_MATRIX][MAX_MATRIX], int* n, int u,
int v) {
        if (u == v || u < 0 || u >= *n || v < 0 || v >= *n) {
            printf("Nekorrektnye vershiny!\n");
            return;
        }

        int newN = *n - 1;
        int newMatrix[MAX_MATRIX][MAX_MATRIX] = { 0 };
        int map[MAX_VERTICES];
        int newIdx = 0;
        int z_index = newN - 1;

        for (int i = 0; i < *n; i++) {
            if (i != u && i != v) {
                map[i] = newIdx++;
            }
        }

        for (int i = 0; i < *n; i++) {
            if (i == u || i == v) continue;
            for (int j = 0; j < *n; j++) {
                if (j == u || j == v) continue;
                newMatrix[map[i]][map[j]] = matrix[i][j];
            }
        }

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

```
            if (i == u || i == v) continue;
            if (matrix[u][i] == 1 || matrix[v][i] == 1) {
                newMatrix[z_index][map[i]] = newMatrix[map[i]][z_index] = 1;
            }
        }

        for (int i = 0; i < newN; i++) {
            for (int j = 0; j < newN; j++) {
                matrix[i][j] = newMatrix[i][j];
            }
        }
        *n = newN;
    }

    void contractEdgeMatrix(int matrix[MAX_MATRIX][MAX_MATRIX], int* n, int u,
int v) {
        if (u < 0 || u >= *n || v < 0 || v >= *n || matrix[u][v] == 0) {
            printf("Rebro ne sushhestvuet ili nekorrektnye vershiny!\n");
            return;
        }
        identifyVerticesMatrix(matrix, n, u, v);
    }

    void splitVertexMatrix(int matrix[MAX_MATRIX][MAX_MATRIX], int* n, int v) {
        if (*n >= MAX_VERTICES - 1) {
            printf("Previsheno maksimalnoe chislo vershin!\n");
            return;
        }
        if (v < 0 || v >= *n) {
            printf("Nekorrektnaya vershina!\n");
            return;
        }

        int newN = *n + 1;
        int newVertex = newN - 1;
        int tempMatrix[MAX_MATRIX][MAX_MATRIX] = { 0 };

        for (int i = 0; i < *n; i++) {
            for (int j = 0; j < *n; j++) {
                tempMatrix[i][j] = matrix[i][j];
            }
        }

        for (int i = 0; i < newN; i++) {
            for (int j = 0; j < newN; j++) {
                matrix[i][j] = 0;
            }
        }

        for (int i = 0; i < *n; i++) {
            for (int j = 0; j < *n; j++) {
                if (i != v && j != v) {
```

```
                    matrix[i][j] = tempMatrix[i][j];
                }
            }
        }

        int count = 0;
        for (int i = 0; i < *n; i++) {
            if (tempMatrix[v][i] == 1 && i != v) {
                if (count % 2 == 0) {
                    matrix[v][i] = matrix[i][v] = 1;
                }
                else {
                    matrix[newVertex][i] = matrix[i][newVertex] = 1;
                }
                count++;
            }
        }

        matrix[v][newVertex] = matrix[newVertex][v] = 1;
        *n = newN;
    }

    Graph* identifyVerticesList(Graph* graph, int u, int v) {
        if (u == v || u < 0 || u >= graph->numVertices || v < 0 || v >= graph-
>numVertices) {
            printf("Nekorrektnye vershiny!\n");
            return NULL;
        }

        int newN = graph->numVertices - 1;
        Graph* newGraph = createGraph(newN);
        int map[MAX_VERTICES];
        int newIdx = 0;
        int z_index = newN - 1;

        for (int i = 0; i < graph->numVertices; i++) {
            if (i != u && i != v) {
                map[i] = newIdx++;
            }
        }

        for (int i = 0; i < graph->numVertices; i++) {
            if (i == u || i == v) continue;
            Node* temp = graph->adjLists[i];
            while (temp) {
                if (temp->vertex != u && temp->vertex != v && temp->vertex > i) {
                    addEdge(newGraph, map[i], map[temp->vertex]);
                }
                temp = temp->next;
            }
        }
```

```
    for (int i = 0; i < graph->numVertices; i++) {
        if (i == u || i == v) continue;
        Node* tempU = graph->adjLists[u];
        bool connected = false;
        while (tempU && !connected) {
            if (tempU->vertex == i) connected = true;
            tempU = tempU->next;
        }
        if (!connected) {
            Node* tempV = graph->adjLists[v];
            while (tempV && !connected) {
                if (tempV->vertex == i) connected = true;
                tempV = tempV->next;
            }
        }
        if (connected) {
            addEdge(newGraph, z_index, map[i]);
        }
    }
    return newGraph;
}

Graph* contractEdgeList(Graph* graph, int u, int v) {
    bool edgeExists = false;
    Node* temp = graph->adjLists[u];
    while (temp) {
        if (temp->vertex == v) {
            edgeExists = true;
            break;
        }
        temp = temp->next;
    }
    if (!edgeExists) {
        printf("Rebro mezhdu %d i %d ne sushhestvuet!\n", u, v);
        return NULL;
    }
    return identifyVerticesList(graph, u, v);
}

Graph* splitVertexList(Graph* graph, int v) {
    if (graph->numVertices >= MAX_VERTICES - 1) {
        printf("Previsheno maksimal'noe chislo vershin!\n");
        return NULL;
    }
    if (v < 0 || v >= graph->numVertices) {
        printf("Nekorrektnaya vershina!\n");
        return NULL;
    }

    int newN = graph->numVertices + 1;
    Graph* newGraph = createGraph(newN);
    int newVertex = newN - 1;
```

```
            for (int i = 0; i < graph->numVertices; i++) {
                if (i == v) continue;
                Node* temp = graph->adjLists[i];
                while (temp) {
                    if (temp->vertex > i && temp->vertex != v) {
                        addEdge(newGraph, i, temp->vertex);
                    }
                    temp = temp->next;
                }
            }

            Node* temp = graph->adjLists[v];
            int count = 0;
            while (temp) {
                if (temp->vertex != v) {
                    if (count % 2 == 0) {
                        addEdge(newGraph, v, temp->vertex);
                    }
                    else {
                        addEdge(newGraph, newVertex, temp->vertex);
                    }
                    count++;
                }
                temp = temp->next;
            }

            addEdge(newGraph, v, newVertex);
            return newGraph;
        }

        void unionGraphs(int m1[MAX_MATRIX][MAX_MATRIX], int
m2[MAX_MATRIX][MAX_MATRIX],
            int result[MAX_MATRIX][MAX_MATRIX], int n1, int n2, int* n_result) {
            *n_result = n1 + n2;
            for (int i = 0; i < *n_result; i++) {
                for (int j = 0; j < *n_result; j++) {
                    result[i][j] = 0;
                }
            }
            for (int i = 0; i < n1; i++) {
                for (int j = 0; j < n1; j++) {
                    result[i][j] = m1[i][j];
                }
            }
            for (int i = 0; i < n2; i++) {
                for (int j = 0; j < n2; j++) {
                    result[n1 + i][n1 + j] = m2[i][j];
                }
            }
        }
```

```
      void intersectGraphs(int m1[MAX_MATRIX][MAX_MATRIX], int
m2[MAX_MATRIX][MAX_MATRIX],
          int result[MAX_MATRIX][MAX_MATRIX], int n1, int n2, int* n_result) {
          if (n1 != n2) {
              *n_result = 0;
              return;
          }

          int n = n1;
          for (int i = 0; i < n; i++) {
              for (int j = 0; j < n; j++) {
                  result[i][j] = (m1[i][j] == 1 && m2[i][j] == 1) ? 1 : 0;
              }
          }

          bool isolated[MAX_VERTICES] = { false };
          for (int i = 0; i < n; i++) {
              bool hasEdge = false;
              for (int j = 0; j < n; j++) {
                  if (result[i][j] == 1 || result[j][i] == 1) {
                      hasEdge = true;
                      break;
                  }
              }
              isolated[i] = !hasEdge;
          }

          int newN = 0;
          int map[MAX_VERTICES];
          for (int i = 0; i < n; i++) {
              if (!isolated[i]) {
                  map[i] = newN++;
              }
          }

          int temp[MAX_MATRIX][MAX_MATRIX] = { 0 };
          for (int i = 0; i < n; i++) {
              if (isolated[i]) continue;
              for (int j = 0; j < n; j++) {
                  if (!isolated[j]) {
                      temp[map[i]][map[j]] = result[i][j];
                  }
              }
          }

          for (int i = 0; i < newN; i++) {
              for (int j = 0; j < newN; j++) {
                  result[i][j] = temp[i][j];
              }
          }
          *n_result = newN;
      }
```

```
      void ringSumGraphs(int m1[MAX_MATRIX][MAX_MATRIX], int
m2[MAX_MATRIX][MAX_MATRIX],
          int result[MAX_MATRIX][MAX_MATRIX], int n1, int n2, int* n_result) {
          if (n1 != n2) {
              *n_result = 0;
              return;
          }

          int n = n1;
          for (int i = 0; i < n; i++) {
              for (int j = 0; j < n; j++) {
                  result[i][j] = (m1[i][j] != m2[i][j]) ? 1 : 0;
              }
          }

          bool isolated[MAX_VERTICES] = { false };
          for (int i = 0; i < n; i++) {
              bool hasEdge = false;
              for (int j = 0; j < n; j++) {
                  if (result[i][j] == 1 || result[j][i] == 1) {
                      hasEdge = true;
                      break;
                  }
              }
              isolated[i] = !hasEdge;
          }

          int newN = 0;
          int map[MAX_VERTICES];
          for (int i = 0; i < n; i++) {
              if (!isolated[i]) {
                  map[i] = newN++;
              }
          }

          int temp[MAX_MATRIX][MAX_MATRIX] = { 0 };
          for (int i = 0; i < n; i++) {
              if (isolated[i]) continue;
              for (int j = 0; j < n; j++) {
                  if (!isolated[j]) {
                      temp[map[i]][map[j]] = result[i][j];
                  }
              }
          }

          for (int i = 0; i < newN; i++) {
              for (int j = 0; j < newN; j++) {
                  result[i][j] = temp[i][j];
              }
          }
          *n_result = newN;
```

```
    }

    void cartesianProduct(int m1[MAX_MATRIX][MAX_MATRIX], int
m2[MAX_MATRIX][MAX_MATRIX],
        int result[MAX_MATRIX][MAX_MATRIX], int n1, int n2, int* n_result) {
        *n_result = n1 * n2;
        for (int i = 0; i < *n_result; i++) {
            for (int j = 0; j < *n_result; j++) {
                result[i][j] = 0;
            }
        }
        for (int i = 0; i < n1; i++) {
            for (int j = 0; j < n1; j++) {
                for (int k = 0; k < n2; k++) {
                    for (int l = 0; l < n2; l++) {
                        int idx1 = i * n2 + k;
                        int idx2 = j * n2 + l;
                        if (i == j && m2[k][l] == 1) {
                            result[idx1][idx2] = 1;
                        }
                        else if (k == l && m1[i][j] == 1) {
                            result[idx1][idx2] = 1;
                        }
                    }
                }
            }
        }
    }

    int main() {
        srand(time(NULL));

        int orig_n1, orig_n2;
        printf("Vvedite razmer G1 (3-%d): ", MAX_VERTICES / 2);
        scanf("%d", &orig_n1);
        printf("Vvedite razmer G2 (3-%d): ", MAX_VERTICES / 2);
        scanf("%d", &orig_n2);

        if (orig_n1 < 3 || orig_n1 > MAX_VERTICES / 2 || orig_n2 < 3 || orig_n2 >
MAX_VERTICES / 2) {
            printf("Nekorrektnyj razmer grafov!\n");
            return 1;
        }

        int n1 = orig_n1, n2 = orig_n2;

        int orig_matrix1[MAX_MATRIX][MAX_MATRIX] = { 0 };
        int orig_matrix2[MAX_MATRIX][MAX_MATRIX] = { 0 };

        int matrix1[MAX_MATRIX][MAX_MATRIX] = { 0 };
        int matrix2[MAX_MATRIX][MAX_MATRIX] = { 0 };
```

```
            generateRandomMatrix(matrix1, n1, 70);
            generateRandomMatrix(matrix2, n2, 70);

            for (int i = 0; i < n1; i++)
                for (int j = 0; j < n1; j++) orig_matrix1[i][j] = matrix1[i][j];
            for (int i = 0; i < n2; i++)
                for (int j = 0; j < n2; j++) orig_matrix2[i][j] = matrix2[i][j];

            printf("\n=== ZADANIE 1: Generaciya grafov ===\n");
            printMatrix(matrix1, n1, "G1 (matrica smezhnosti)");
            printMatrix(matrix2, n2, "G2 (matrica smezhnosti)");

            Graph* g1 = createGraph(n1);
            Graph* g2 = createGraph(n2);
            matrixToAdjList(matrix1, n1, g1);
            matrixToAdjList(matrix2, n2, g2);

            printAdjList(g1, "G1");
            printAdjList(g2, "G2");

            printf("\n=== ZADANIE 2: Unarnye operacii ===\n");

            int choice;
            printf("Vyberite operaciyu:\n");
            printf("1 - Otozhdestvlenie vershin (matrica)\n");
            printf("2 - Styagivanie rebra (matrica)\n");
            printf("3 - Rasshheplenie vershiny (matrica)\n");
            printf("4 - Otozhdestvlenie vershin (spiski)\n");
            printf("5 - Styagivanie rebra (spiski)\n");
            printf("6 - Rasshheplenie vershiny (spiski)\n");
            scanf("%d", &choice);

            int u, v, vertex;

            switch (choice) {
            case 1:
                printf("Vvedite vershiny u i v dlya otozhdestvleniya (0-%d): ", n1 -
1);
                scanf("%d %d", &u, &v);
                printf("DO: n1=%d\n", n1);
                identifyVerticesMatrix(matrix1, &n1, u, v);
                printf("POSLE: n1=%d\n", n1);
                printMatrix(matrix1, n1, "Rezul'tat otozhdestvleniya vershin
(matrica)");
                matrixToAdjList(matrix1, n1, g1);
                break;
            case 2:
                printf("Vvedite vershiny u i v rebra dlya styagivaniya (0-%d): ", n1
- 1);
                scanf("%d %d", &u, &v);
                printf("DO: n1=%d\n", n1);
                contractEdgeMatrix(matrix1, &n1, u, v);
```

```
            printf("POSLE: n1=%d\n", n1);
            printMatrix(matrix1, n1, "Rezul'tat styagivaniya rebra (matrica)");
            matrixToAdjList(matrix1, n1, g1);
            break;
        case 3:
            printf("Vvedite vershinu dlya rasshhepleniya (0-%d): ", n1 - 1);
            scanf("%d", &vertex);
            printf("DO: n1=%d\n", n1);
            splitVertexMatrix(matrix1, &n1, vertex);
            printf("POSLE: n1=%d\n", n1);
            printMatrix(matrix1, n1, "Rezul'tat rasshhepleniya vershiny
(matrica)");
            matrixToAdjList(matrix1, n1, g1);
            break;
        case 4: {
            printf("Vvedite vershiny u i v dlya otozhdestvleniya (0-%d): ", g1-
>numVertices - 1);
            scanf("%d %d", &u, &v);
            Graph* newGraph = identifyVerticesList(g1, u, v);
            if (newGraph) {
                printAdjList(newGraph, "Rezul'tat otozhdestvleniya vershin
(spiski)");
                freeGraph(newGraph);
            }
            break;
        }
        case 5: {
            printf("Vvedite vershiny u i v rebra dlya styagivaniya (0-%d): ", g1-
>numVertices - 1);
            scanf("%d %d", &u, &v);
            Graph* newGraph = contractEdgeList(g1, u, v);
            if (newGraph) {
                printAdjList(newGraph, "Rezul'tat styagivaniya rebra (spiski)");
                freeGraph(newGraph);
            }
            break;
        }
        case 6: {
            printf("Vvedite vershinu dlya rasshhepleniya (0-%d): ", g1-
>numVertices - 1);
            scanf("%d", &vertex);
            Graph* newGraph = splitVertexList(g1, vertex);
            if (newGraph) {
                printAdjList(newGraph, "Rezul'tat rasshhepleniya vershiny
(spiski)");
                freeGraph(newGraph);
            }
            break;
        }
        default:
            printf("Nekorrektnyj vybor!\n");
        }
```

```
          printf("\n=== ZADANIE 3: Binarnye operacii ===\n");

          int resultUnion[MAX_MATRIX][MAX_MATRIX] = { 0 };
          int resultIntersect[MAX_MATRIX][MAX_MATRIX] = { 0 };
          int resultRingSum[MAX_MATRIX][MAX_MATRIX] = { 0 };
          int n_union, n_intersect, n_ringsum;

          unionGraphs(orig_matrix1, orig_matrix2, resultUnion, orig_n1, orig_n2,
&n_union);
          printMatrix(resultUnion, n_union, "G1 G2 (ob'edinenie)");

          if (orig_n1 == orig_n2) {
              intersectGraphs(orig_matrix1, orig_matrix2, resultIntersect, orig_n1,
orig_n2, &n_intersect);
              printMatrix(resultIntersect, n_intersect, "G1 G2 (peresechenie)");

              ringSumGraphs(orig_matrix1, orig_matrix2, resultRingSum, orig_n1,
orig_n2, &n_ringsum);
              printMatrix(resultRingSum, n_ringsum, "G1 (+) G2 (kol'cevaya summa)");
          }
          else {
              printf("Peresechenie i kol'cevaya summa trebuyut grafy odinakovogo
razmera!\n");
          }

          printf("\n=== ZADANIE 4: Dekartovo proizvedenie ===\n");

          int cartResult[MAX_MATRIX][MAX_MATRIX] = { 0 };
          int n_cart;
          cartesianProduct(orig_matrix1, orig_matrix2, cartResult, orig_n1, orig_n2,
&n_cart);
          printMatrix(cartResult, n_cart, "G1 x G2 (dekartovo proizvedenie)");

          freeGraph(g1);
          freeGraph(g2);
          return 0;
      }
```

Результаты работы программ

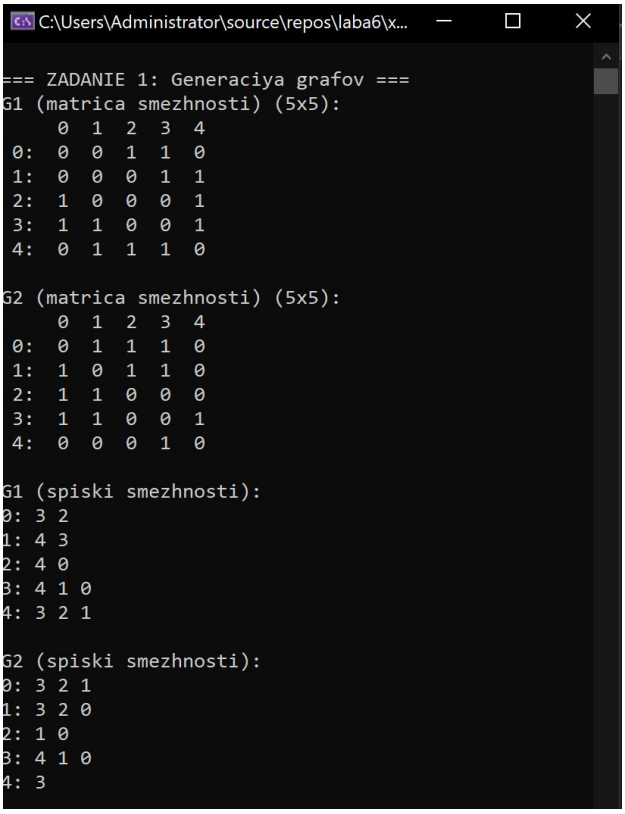

Рисунок 1 - Результат работы программы (задание1)

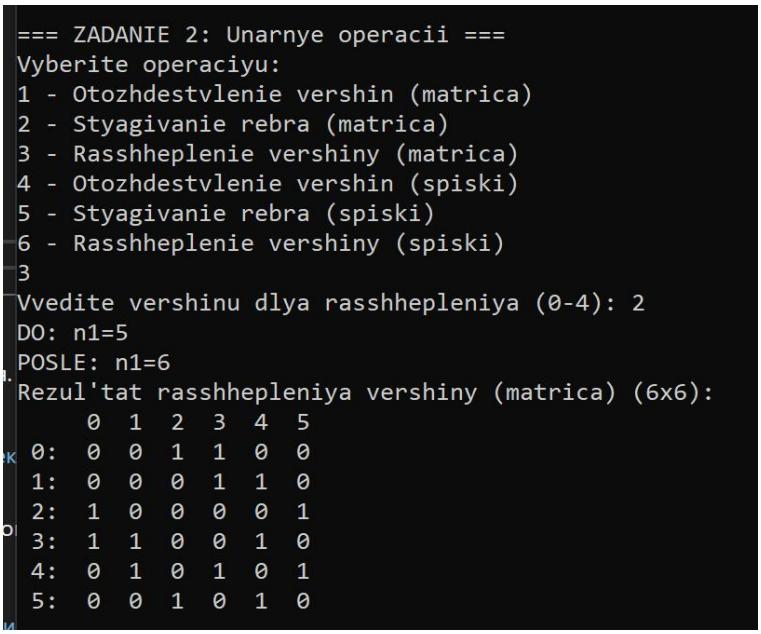

Рисунок 2 - Результат работы программы (задание2)

```
=== ZADANIE 3: Binarnye operacii ===
G1 G2 (ob'edinenie) (10x10):
     0  1  2  3  4  5  6  7  8  9
 0:  0  0  1  1  0  0  0  0  0  0
 1:  0  0  0  1  1  0  0  0  0  0
 2:  1  0  0  0  1  0  0  0  0  0
 3:  1  1  0  0  1  0  0  0  0  0
 4:  0  1  1  1  0  0  0  0  0  0
 5:  0  0  0  0  0  0  1  1  1  0
 6:  0  0  0  0  0  1  0  1  1  0
 7:  0  0  0  0  0  1  1  0  0  0
 8:  0  0  0  0  0  1  1  0  0  1
 9:  0  0  0  0  0  0  0  0  1  0

G1 G2 (peresechenie) (5x5):
     0  1  2  3  4
 0:  0  0  1  1  0
 1:  0  0  0  1  0
 2:  1  0  0  0  0
 3:  1  1  0  0  1
 4:  0  0  0  1  0

G1 (+) G2 (kol'cevaya summa) (4x4):
     0  1  2  3
 0:  0  1  0  0
 1:  1  0  1  1
 2:  0  1  0  1
 3:  0  1  1  0
```

Рисунок 3 - Результат работы программы (задание3)

```
=== ZADANIE 4: Dekartovo proizvedenie ===
G1 x G2 (dekartovo proizvedenie) (25x25):
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 0:  0  1  1  1  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0
 1:  1  0  1  1  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0
 2:  1  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0
 3:  1  1  0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0
 4:  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
 5:  0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0
 6:  0  0  0  0  0  1  0  1  1  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0
 7:  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0
 8:  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0
 9:  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1
10:  1  0  0  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  1  0  0  0  0
11:  0  1  0  0  0  0  0  0  0  0  1  0  1  1  0  0  0  0  0  0  0  1  0  0  0
12:  0  0  1  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  1  0  0
13:  0  0  0  1  0  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  0  0  1  0
14:  0  0  0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1
15:  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1  1  1  0  1  0  0  0  0  0
16:  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  1  0  1  1  0  0  1  0  0  0
17:  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  1  1  0  0  0  0  0  1  0  0
18:  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  1  1  0  0  1  0  0  0  1  0
19:  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0  0  1
20:  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  1  1  0
21:  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0  0  1  0  1  1  0
22:  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0  1  1  0  0  0
23:  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  1  1  0  0  1
24:  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  1  0  0  0  1  0
```

Рисунок 3 - Результат работы программы (задание4)

Вывод: В ходе выполнения лабораторной работы была разработана программа на языке C, реализующая основные унарные и бинарные операции над графами.