Министерство науки и высшего образования РФ

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 22ВВП2:

Широкова И.Д.

Перкин П.О.

Приняли

Акифьев И.В.

Юрова О.В.

Пенза 2023

**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

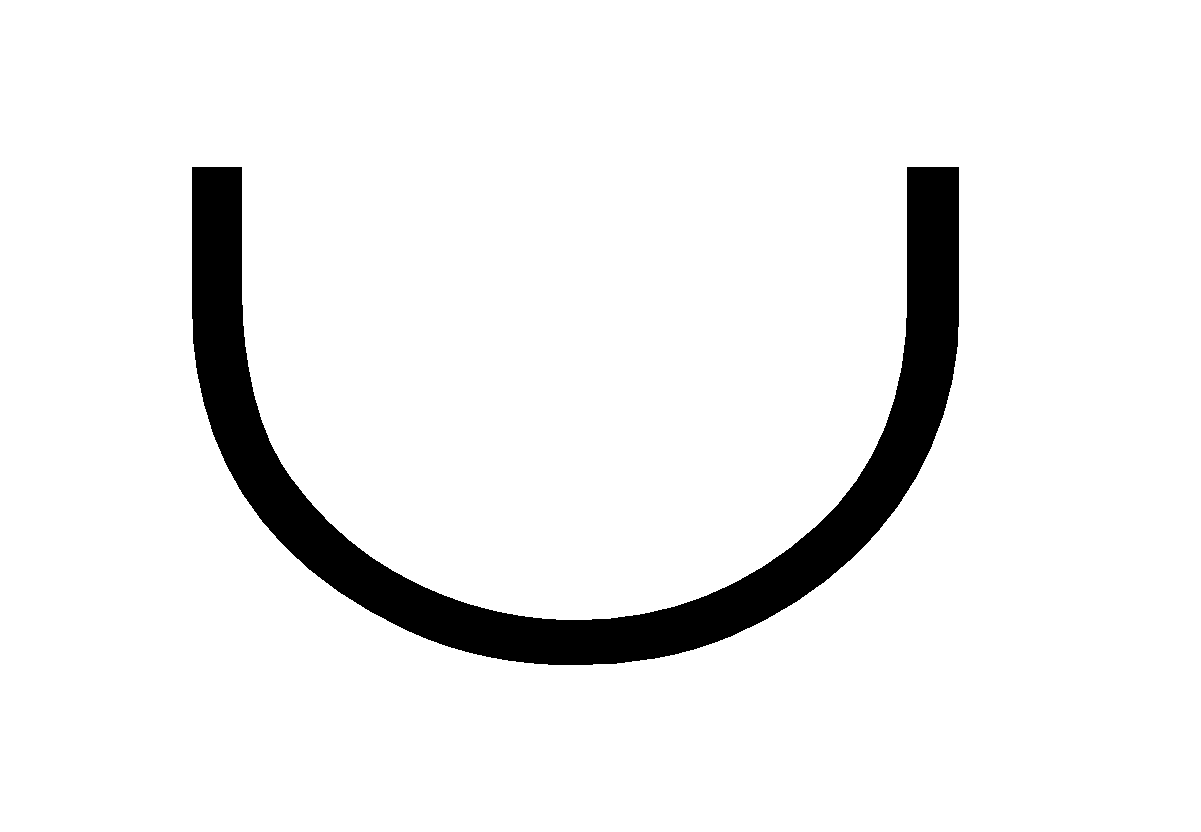
в) расщепления вершины

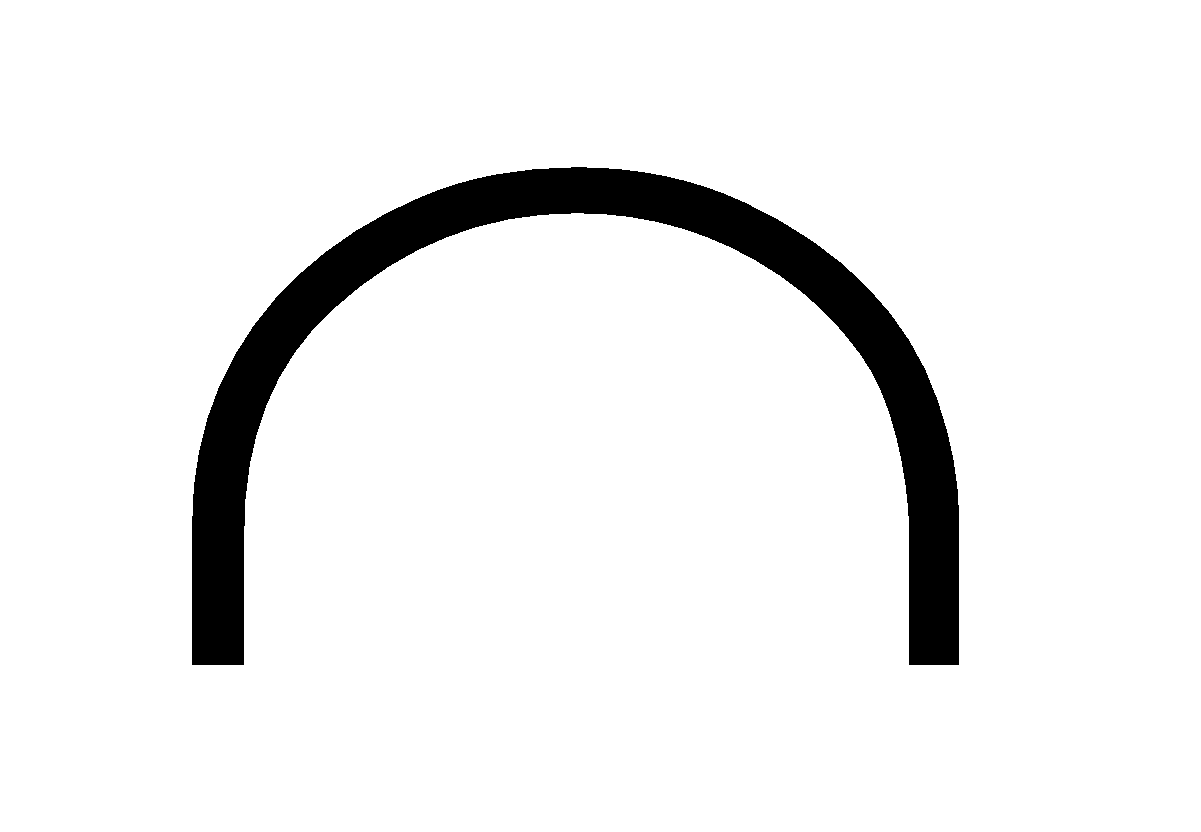
Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

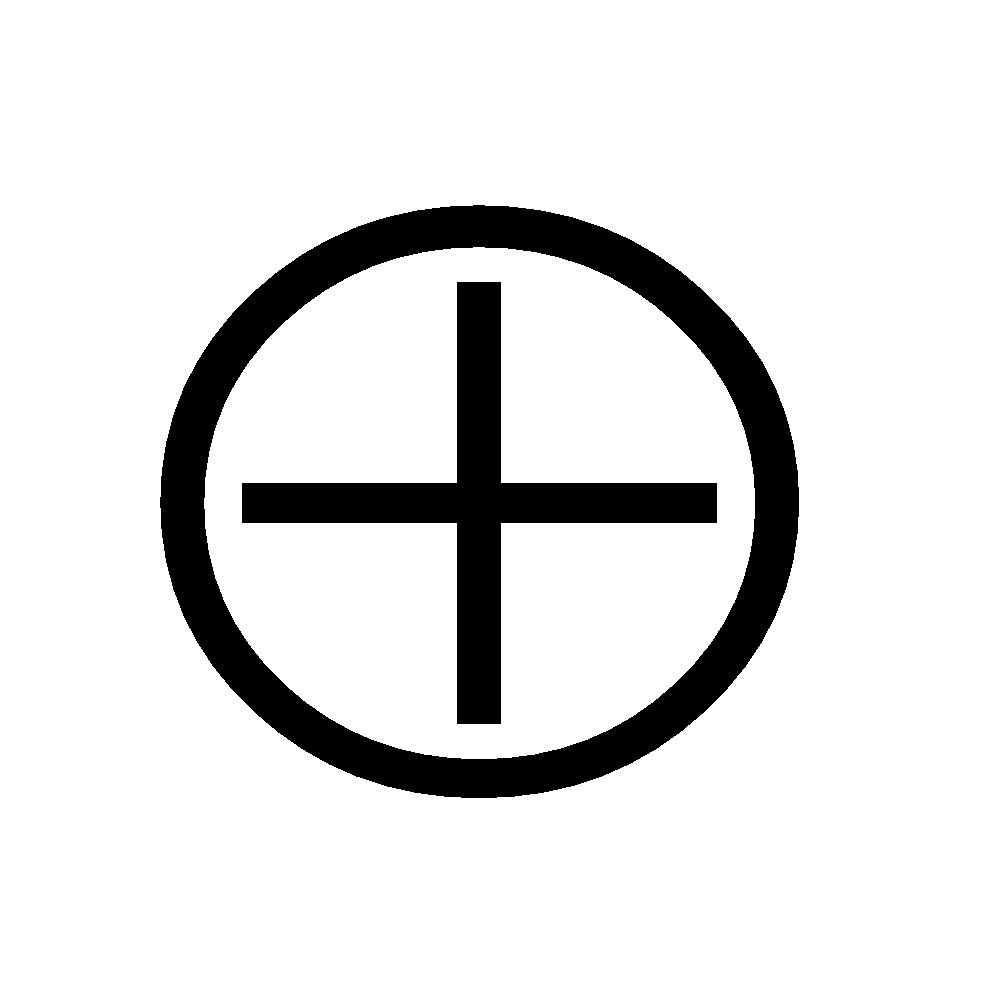
Результат выполнения операции выведите на экран.

**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.

**Листинг**

**#include <iostream>**

**#include <vector>**

**#include <ctime>**

**#include <cstdlib>**

**#include <algorithm>**

**#include <conio.h>**

**using namespace std;**

**vector<vector<int>> generateAndPrintMatrix(int num\_vertices, double edge\_probability) {**

**vector<vector<int>> adjacency\_matrix(num\_vertices + 1, vector<int>(num\_vertices + 1, 0));**

**int num\_edges = 0;**

**for (int i = 1; i <= num\_vertices; ++i) {**

**for (int j = i + 1; j <= num\_vertices; ++j) {**

**if (static\_cast<double>(rand()) / RAND\_MAX < edge\_probability) {**

**adjacency\_matrix[i][j] = 1;**

**adjacency\_matrix[j][i] = 1;**

**num\_edges++;**

**}**

**}**

**}**

**cout << "Матрица смежности:\n";**

**for (int i = 1; i <= num\_vertices; ++i) {**

**for (int j = 1; j <= num\_vertices; ++j) {**

**cout << adjacency\_matrix[i][j] << ' ';**

**}**

**cout << endl;**

**}**

**return adjacency\_matrix;**

**}**

**vector<vector<int>> matrixToList(const vector<vector<int>>& matrix) {**

**int num\_vertices = matrix.size() - 1;**

**vector<vector<int>> adjacency\_list(num\_vertices + 1);**

**for (int i = 1; i <= num\_vertices; ++i) {**

**for (int j = 1; j <= num\_vertices; ++j) {**

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

**adjacency\_list[i].push\_back(j);**

**}**

**}**

**}**

**return adjacency\_list;**

**}**

**void printList(const vector<vector<int>>& list) {**

**cout << "Список смежности:\n";**

**for (int i = 1; i < list.size(); ++i) {**

**cout << i << ": ";**

**for (const int& j : list[i]) {**

**cout << j << ' ';**

**}**

**cout << endl;**

**}**

**}**

**void printMatrix(const vector<vector<int>>& matrix) {**

**cout << "Матрица смежности:\n";**

**for (int i = 1; i < matrix.size(); i++) {**

**for (int j = 1; j < matrix[i].size(); j++) {**

**cout << matrix[i][j] << ' ';**

**}**

**cout << endl;**

**}**

**}**

**//отождествления вершин**

**void identifyVertices(const vector < vector < int>>& originalMatrix, const vector < vector < int>>& originalList, int vertex1, int vertex2, vector < vector < int>>& resultMatrix, vector < vector < int>>& resultList) {**

**// Создание копий исходной матрицы и списка смежности**

**resultMatrix = originalMatrix;**

**resultList = originalList;**

**int n = resultMatrix.size();**

**// Обновление копии матрицы смежности**

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

**if (resultMatrix[vertex2][i]) {**

**resultMatrix[vertex1][i] = 1;**

**resultMatrix[i][vertex1] = 1;**

**}**

**resultMatrix[vertex2][i] = 0;**

**resultMatrix[i][vertex2] = 0;**

**}**

**// Обновление копии списка смежности**

**for (int neighbor : resultList[vertex2]) {**

**if (find(resultList[vertex1].begin(), resultList[vertex1].end(), neighbor) == resultList[vertex1].end() && neighbor != vertex1) {**

**resultList[vertex1].push\_back(neighbor);**

**}**

**}**

**// Удаление vertex2 из всех списков смежности в копии**

**for (auto& neighbors : resultList) {**

**auto it = remove(neighbors.begin(), neighbors.end(), vertex2);**

**neighbors.erase(it, neighbors.end());**

**for (auto& neighbor : neighbors) {**

**if (neighbor > vertex2) {**

**--neighbor;**

**}**

**}**

**}**

**// Удаление vertex2 из копии матрицы**

**resultMatrix.erase(resultMatrix.begin() + vertex2);**

**for (auto& row : resultMatrix) {**

**row.erase(row.begin() + vertex2);**

**}**

**// Удаление списка смежности для vertex2 и обновление последующих индексов в копии**

**resultList.erase(resultList.begin() + vertex2);**

**}**

**//стягивания ребра**

**void contractEdge(const vector<vector<int>>& originalMatrix, const vector<vector<int>>& originalList, int vertex1, int vertex2, vector<vector<int>>&**

**resultMatrix, vector<vector<int>>& resultList) {**

**resultMatrix = originalMatrix;**

**resultList = originalList;**

**int n = resultMatrix.size();**

**// Проверка на наличие ребра между vertex1 и vertex2**

**if (!resultMatrix[vertex1][vertex2] && !resultMatrix[vertex2][vertex1]) {**

**printf("Нет ребра между %d и %d для стягивания.\n", vertex1, vertex2);**

**return;**

**}**

**// Объединение вершин в копии матрицы смежности**

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

**if (resultMatrix[vertex2][i]) {**

**resultMatrix[vertex1][i] = 1;**

**resultMatrix[i][vertex1] = 1;**

**}**

**resultMatrix[vertex2][i] = 0;**

**resultMatrix[i][vertex2] = 0;**

**}**

**resultMatrix[vertex1][vertex1] = 0;**

**// Объединение списков смежности**

**for (int neighbor : resultList[vertex2]) {**

**if (find(resultList[vertex1].begin(), resultList[vertex1].end(), neighbor) == resultList[vertex1].end() && neighbor != vertex1) {**

**resultList[vertex1].push\_back(neighbor);**

**}**

**}**

**// Удаление vertex2 из всех списков смежности в копии**

**for (auto& neighbors : resultList) {**

**auto it = remove(neighbors.begin(), neighbors.end(), vertex2);**

**neighbors.erase(it, neighbors.end());**

**}**

**// Удаление vertex2 из копии матрицы и списка смежности**

**resultMatrix.erase(resultMatrix.begin() + vertex2);**

**for (auto& row : resultMatrix) {**

**row.erase(row.begin() + vertex2);**

**}**

**resultList.erase(resultList.begin() + vertex2);**

**}**

**//расщепление вершины**

**void splitVertex(vector < vector < int>>& originalMatrix, vector < vector < int>>& originalList, int vertex, vector < vector < int>>& resultMatrix, vector < vector < int>>& resultList) {**

**int n = originalMatrix.size();**

**// Создаем новую матрицу смежности и новый список смежности**

**resultMatrix.resize(n + 1, vector<int>(n + 1, 0));**

**resultList.resize(n + 1);**

**// Копируем исходную матрицу смежности и список смежности в новые структуры**

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

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

**resultMatrix[i][j] = originalMatrix[i][j];**

**}**

**resultList[i] = originalList[i];**

**}**

**// Копируем вершину, которую отождествляем**

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

**resultMatrix[i][n] = originalMatrix[i][vertex];**

**resultMatrix[n][i] = originalMatrix[vertex][i];**

**}**

**// Добавляем новую вершину в список смежности**

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

**if (i != vertex) {**

**resultList[i].push\_back(n);**

**}**

**}**

**// Добавляем новую вершину в конец списка смежности**

**for (int i = 0; i < originalList[vertex].size(); ++i) {**

**resultList[n].push\_back(originalList[vertex][i]);**

**}**

**resultList[n].push\_back(vertex);**

**resultList[vertex].push\_back(n);**

**resultMatrix[vertex][n] = 1;**

**resultMatrix[n][vertex] = 1;**

**}**

**//объединения**

**vector < vector < int>> mergeGraphs(const vector < vector < int>>& graph1, const vector < vector < int>>& graph2) {**

**int maxSize = max(graph1.size(), graph2.size());**

**// Создание новой матрицы смежности для объединенного графа**

**vector < vector < int>> mergedGraph(maxSize, vector<int>(maxSize, 0));**

**// Копирование связей из первого графа**

**for (int i = 0; i < graph1.size(); ++i) {**

**for (int j = 0; j < graph1[i].size(); ++j) {**

**mergedGraph[i][j] = graph1[i][j];**

**}**

**}**

**// Добавление или перезапись связей из второго графа**

**for (int i = 0; i < graph2.size(); ++i) {**

**for (int j = 0; j < graph2[i].size(); ++j) {**

**if (graph2[i][j] > 0) {**

**mergedGraph[i][j] = graph2[i][j];**

**}**

**}**

**}**

**return mergedGraph;**

**}**

**//пересечения**

**vector < vector < int>> intersectGraphs(const vector < vector < int>>& graph1, const vector < vector < int>>& graph2) {**

**int size = min(graph1.size(), graph2.size());**

**vector < vector < int>> intersectionGraph(size, vector<int>(size, 0));**

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

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

**// Элемент в новой матрице будет 1, если соответствующие элементы обеих матриц равны 1**

**intersectionGraph[i][j] = (graph1[i][j] && graph2[i][j]);**

**}**

**}**

**return intersectionGraph;**

**}**

**//кольцевой суммы**

**vector < vector < int>> adaptiveRingSumGraphs(const vector < vector < int>>& graph1, const vector < vector < int>>& graph2) {**

**int maxSize = max(graph1.size(), graph2.size());**

**vector < vector < int>> ringSumGraph(maxSize, vector<int>(maxSize, 0));**

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

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

**int val1 = (i < graph1.size() && j < graph1.size()) ? graph1[i][j] : 0;**

**int val2 = (i < graph2.size() && j < graph2.size()) ? graph2[i][j] : 0;**

**ringSumGraph[i][j] = val1 != val2;**

**}**

**}**

**return ringSumGraph;**

**}**

**//декартова произведения**

**vector < vector < int>> cartesianProductGraphs(const vector < vector < int>>& graph1, const vector < vector < int>>& graph2) {**

**int n1 = graph1.size() - 1; // Учитываем, что размеры матриц начинаются с 1**

**int n2 = graph2.size() - 1;**

**int productSize = n1 \* n2;**

**vector < vector < int>> productGraph(productSize + 1, vector<int>(productSize + 1, 0)); // +1 для корректного размера**

**for (int i1 = 1; i1 <= n1; ++i1) {**

**for (int i2 = 1; i2 <= n2; ++i2) {**

**for (int j1 = 1; j1 <= n1; ++j1) {**

**for (int j2 = 1; j2 <= n2; ++j2) {**

**int v1 = (i1 - 1) \* n2 + i2; // Пересчитываем индексы**

**int v2 = (j1 - 1) \* n2 + j2;**

**if (i1 == j1 && graph2[i2][j2]) {**

**productGraph[v1][v2] = 1;**

**}**

**if (i2 == j2 && graph1[i1][j1]) {**

**productGraph[v1][v2] = 1;**

**}**

**}**

**}**

**}**

**}**

**return productGraph;**

**}**

**int main() {**

**setlocale(LC\_ALL, "RUS");**

**int num\_vertices, operation;**

**int graph\_choice, vertex1, vertex2;**

**double edge\_probability = 0.5;**

**vector<vector<int>> matrix1, matrix2, matrixCopy;**

**vector<vector<int>> list1, list2, listCopy;**

**srand(static\_cast<unsigned int>(time(nullptr)));**

**while (true) {**

**cout << "Выберите операцию:\n 1 - Сгенерировать графы\n 2 - Отождествление и Стягивание\n 3 - Расщепление вершины\n 4 - Бинарные операции\n 0 - Выход\n";**

**operation = \_getch() - '0'; // получаем символ и конвертируем его в число**

**if (operation == 0) {**

**break;**

**}**

**switch (operation) {**

**//генерация матриц**

**case 1: {**

**cout << "Введите количество вершин в первом графе: ";**

**cin >> num\_vertices;**

**cout << "Первый граф:\n";**

**matrix1 = generateAndPrintMatrix(num\_vertices, edge\_probability);**

**list1 = matrixToList(matrix1);**

**printList(list1);**

**cout << "Введите количество вершин во втором графе: ";**

**cin >> num\_vertices;**

**cout << "Второй граф:\n";**

**matrix2 = generateAndPrintMatrix(num\_vertices, edge\_probability);**

**list2 = matrixToList(matrix2);**

**printList(list2);**

**break;**

**}**

**case 2: {**

**cout << "Введите номер графа для выполнения операции (1 или 2): ";**

**cin >> graph\_choice;**

**cout << "Введите номера двух вершин (например, 1 3): ";**

**cin >> vertex1 >> vertex2;**

**if (graph\_choice == 1) {**

**cout << "отождествление:\n";**

**identifyVertices(matrix1, list1, vertex1, vertex2, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**cout << "стягивание:\n";**

**contractEdge(matrix1, list1, vertex1, vertex2, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**}**

**else if (graph\_choice == 2) {**

**cout << "отождествление:\n";**

**identifyVertices(matrix2, list2, vertex1, vertex2, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**cout << "стягивание:\n";**

**contractEdge(matrix2, list2, vertex1, vertex2, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**}**

**break;**

**}**

**case 3: {**

**cout << "Введите номер графа для выполнения операции (1 или 2): ";**

**cin >> graph\_choice;**

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

**cin >> vertex1;**

**if (graph\_choice == 1) {**

**splitVertex(matrix1, list1, vertex1, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**}**

**else if (graph\_choice == 2) {**

**splitVertex(matrix2, list2, vertex1, matrixCopy, listCopy);**

**printMatrix(matrixCopy);**

**printList(matrixToList(matrixCopy));**

**}**

**break;**

**}**

**case 4: {**

**vector<vector<int>> mergedGraph = mergeGraphs(matrix1, matrix2);**

**cout << "Объединение: \n";**

**printMatrix(mergedGraph);**

**vector<vector<int>> intersectionGraph = intersectGraphs(matrix1, matrix2);**

**cout << "Пересечение: \n";**

**printMatrix(intersectionGraph);**

**vector<vector<int>> ringSumGraph = adaptiveRingSumGraphs(matrix1, matrix2);**

**cout << "Кольцевая сумма: \n";**

**printMatrix(ringSumGraph);**

**vector<vector<int>> cartesianProductGraph = cartesianProductGraphs(matrix1, matrix2);**

**cout << "Декартово произведение: \n";**

**printMatrix(cartesianProductGraph);**

**break;**

**}**

**default:**

**cout << "Неверный выбор операции.\n";**

**break;**

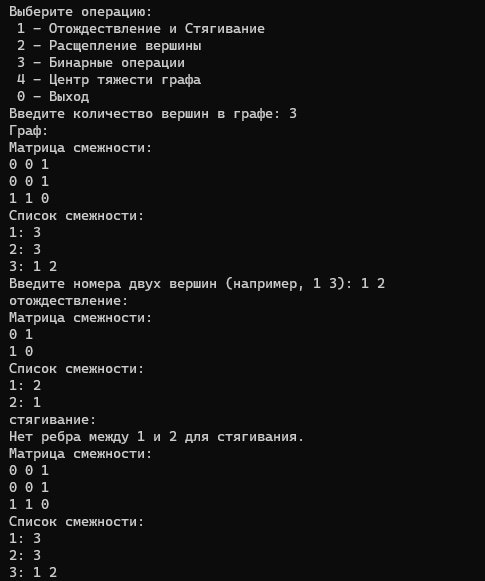
**}**

**}**

**return 0;**

**}**

**Результаты работы программы**

****

**Вывод**

В ходе выполнения работы были исследованы различные унарные и бинарные операции над графами. Подобные операции имеют важное значение в алгоритмах анализа данных и теории графов.