Пензенский государственный университет

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе № 6

по дисциплине: "Логика и ОА в ИЗ"

## на тему: " Унарные и бинарные операции над графами "

Выполнили:

Трундов Н.А.

Евдокимов Р.E.

Приняли:

Акифьев И.В.

Юрова О.В.

Пенза, 2023

**Цель:**

Изучить принцип работы с графами с помощью унарных и бинарных операций.

**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 *и,v.* Определяют окружение *Q*1 вершины *u*,и окружение *Q*2 вершины *v,* вычисляют их объединение *Q* = *Q1* * Q2.* Затем над графом *G*1 выполняются следующие преобразования:

* из графа *G*1 удаляют вершины *u,* *v (H*1 *= G*1 *- u - v);*
* к графу *Н*1присоединяют новую вершину *z (H*1 *= H*1 *+z);*
* вершину *z* соединяют ребром с каждой из вершин *w*1*Q*

*(G*2 *= H*1 *+ zwi*, *i =* 1,2,3*,…).*

*Стягивание ребра.* Данная операция является операцией отождествления смежных вершин *и, 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 *VG = V*1*V*2*; UG = U*1* U*2 (рис. 1).

**U**

*v*1

*v*2

*v*3

*v*4

*v*3

*v*4

*v*5

*v*2

*v*1

*v*3

*v*4

*v*5

Рис. 1. Объединение графов *G*1, *G*2

Объединение графов *G*1 и *G*2 называется дизъюнктным, если *V*1*V*2 *= *. При дизъюнктном объединении никакие два из объединяемых графов не должны иметь общих вершин.

*Пересечение.* Граф *G* называется пересечением графов *G*1, *G*2,если *VG = V*1*V*2и *UG = U*1*U*2 (риc.2). Операция "пересечения" записывается следующим образом: *G = G*1*G*2*.*

**∩**

*v*1

*v*2

*v*3

*v*5

*v*3

*v*4

*v*6

*v*2

*v*1

*v*6

*v*4

*v*5

*v*1

*v*4

*v*6

*v*5

*v*3

*v*2

Рис.2. Пересечение графов *G*1, *G*2*.*

*Декартово произведение.* Граф *G* называется декартовым произведением графов *G*1 и *G*2 если *VG* = *V*1*V*2 —декартово произведение множеств вершин графов *G*1, *G*2, а множество ребер *U*c задается следующим образом: вершины (*zi*, *vk*) и (*zj*, *vl*) смежны в графе *G* тогда и только тогда, когда *zi* = *zj*(*i* = *j*), a *v*k и *vl* смежны в *G*2 или *vk* = *vl*(*k* = *l*), смежны в графе *G*1 (см. рис.3).

**X**

*z*1

*z*2

*v*1

*v*3

*v*2

*z*1*v*1

*z*1*v*2

*z*1*v*3

*z*2*v*1

*z*2*v*2

*z*2*v*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  *G*2 (рис.4).

**⊕**

*v*1

*v*2

*v*3

*v*5

*v*3

*v*4

*v*6

*v*2

*v*1

*v*4

*v*5

*v*1

*v*4

*v*6

*v*5

*v*3

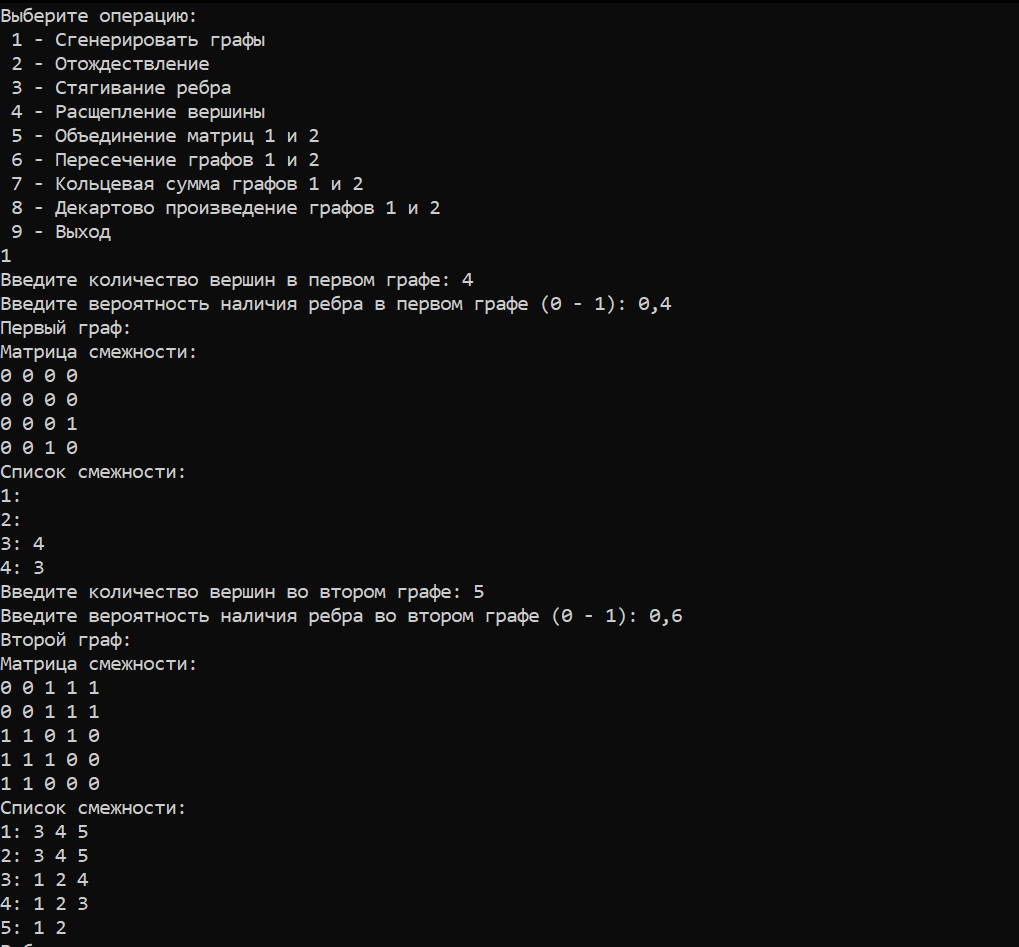
*v*2

Рис.4. Кольцевая сумма графов *G1, G2*

### **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.

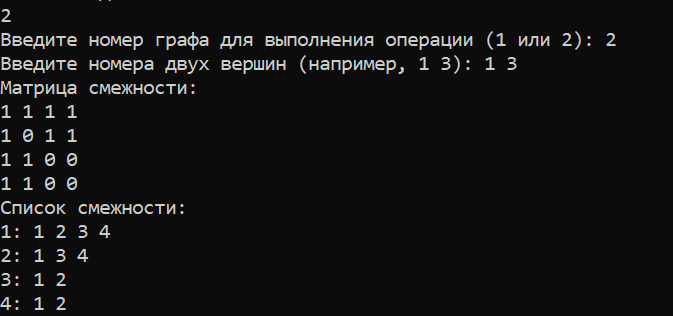
\* Для указанных графов преобразуйте представление матриц смежности в списки смежности. Выведите полученные списки на экран.



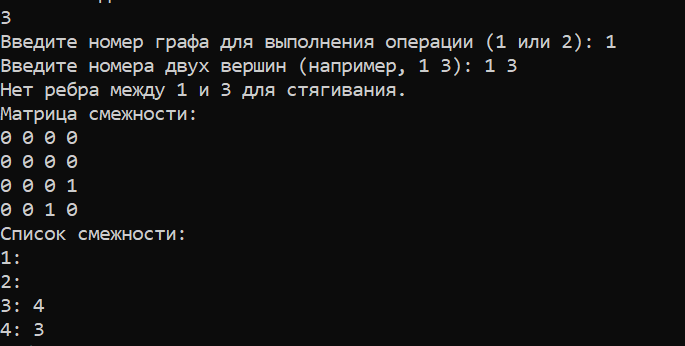
### **Задание 2**

1. Для матричной формы представления графов выполните операцию:

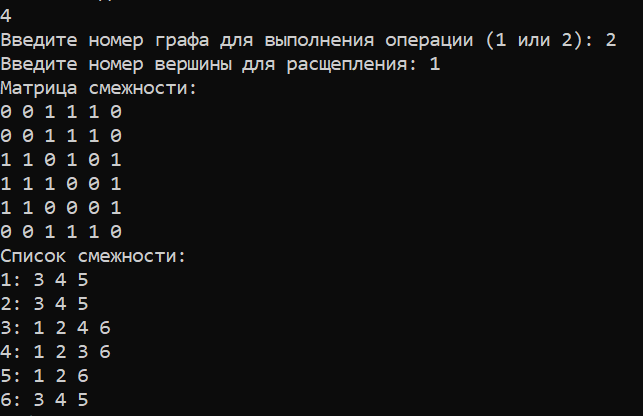
а) отождествления вершин



б) стягивания ребра



в) расщепления вершины

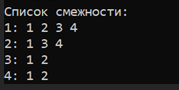


Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

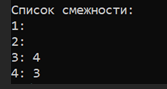
Результат выполнения операции выведите на экран.

1. \* Для представления графов в виде списков смежности выполните операцию:

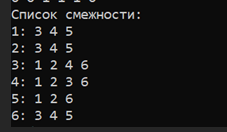
а) отождествления вершин



б) стягивания ребра



в) расщепления вершины



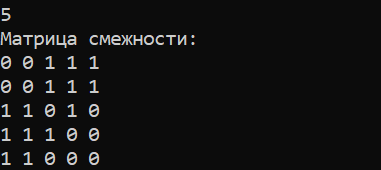
Номера выбираемых для выполнения операции вершин ввести с клавиатуры.

Результат выполнения операции выведите на экран.

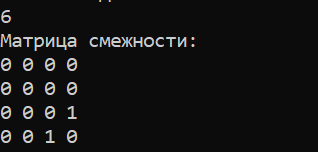
**Задание 3**

1. Для матричной формы представления графов выполните операцию:

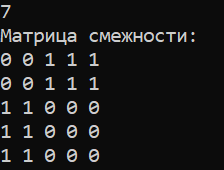
а) объединения *G* = *G*1  *G*2



б) пересечения *G* = *G*1  *G*2



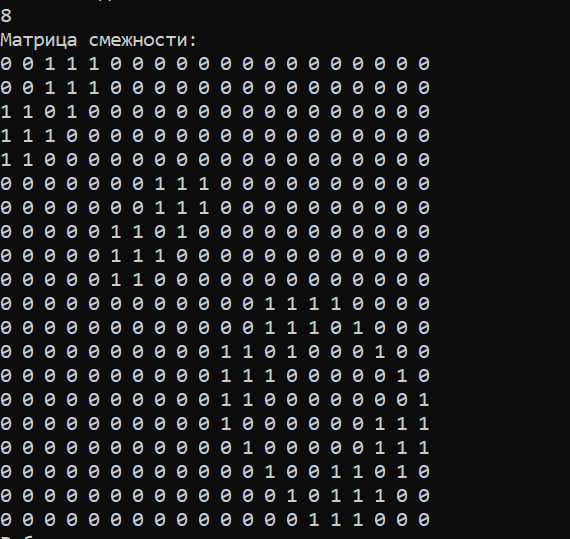
в) кольцевой суммы *G* = *G*1  *G*2



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.

**Вывод:**

Изучили принцип работы с графами с помощью унарных и бинарных операций.

Приложение А (“Листинг”)

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <cstdio>

#include <vector>

#include <ctime>

#include <cstdlib>

#include <locale.h>

#include <algorithm>

using namespace std;

vector < vector < int>> mergeVertices(const vector < vector < int>>& G, int vertex1, int vertex2) {

vector < vector < int>> result = G;

// Добавление соседей vertex2 к списку смежности vertex1, если они уникальны

for (int neighbor : G[vertex2]) {

if (neighbor != vertex1) {

// Добавляем только уникальных соседей

if (find(result[vertex1].begin(), result[vertex1].end(), neighbor) == result[vertex1].end()) {

result[vertex1].push\_back(neighbor);

}

}

}

// Перенаправление ребер, идущих к vertex2, на vertex1

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

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

if (result[i][j] == vertex2) {

result[i][j] = vertex1;

// Удаление дубликатов, если они появились после перенаправления

if (i == vertex1) {

result[i].erase(unique(result[i].begin(), result[i].end()), result[i].end());

}

}

}

}

// Удаление петель у vertex1, если они есть

result[vertex1].erase(remove(result[vertex1].begin(), result[vertex1].end(), vertex1), result[vertex1].end());

// Удаление vertex2 из списка смежности всех вершин

for (auto& neighbors : result) {

neighbors.erase(remove(neighbors.begin(), neighbors.end(), vertex2), neighbors.end());

}

// Удаление списка смежности для vertex2

result.erase(result.begin() + vertex2);

return result;

}

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++;

}

}

}

printf("Матрица смежности:\n");

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

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

printf("%d ", adjacency\_matrix[i][j]);

}

printf("\n");

}

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) {

printf("Список смежности:\n");

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

printf("%d: ", i);

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

printf("%d ", j);

}

printf("\n");

}

}

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

printf("Матрица смежности:\n");

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

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

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

}

printf("\n");

}

}

//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]);

}

}

//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;

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

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

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

while (true) {

printf("Выберите операцию:\n 1 - Сгенерировать графы\n 2 - Отождествление\n 3 - Стягивание ребра\n 4 - Расщепление вершины\n 5 - Объединение матриц 1 и 2\n 6 - Пересечение графов 1 и 2\n 7 - Кольцевая сумма графов 1 и 2\n 8 - Декартово произведение графов 1 и 2\n 9 - Выход\n");

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

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

printf("Ошибка: ожидался ввод числа.\n");

while (getchar() != '\n'); // Очистка ввода

continue;

}

if (operation == 9) {

break;

}

switch (operation) {

case 1: {

printf("Введите количество вершин в первом графе: ");

scanf("%d", &num\_vertices);

printf("Введите вероятность наличия ребра в первом графе (0 - 1): ");

scanf("%lf", &edge\_probability);

printf("Первый граф:\n");

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

list1 = matrixToList(matrix1);

printList(list1);

printf("Введите количество вершин во втором графе: ");

scanf("%d", &num\_vertices);

printf("Введите вероятность наличия ребра во втором графе (0 - 1): ");

scanf("%lf", &edge\_probability);

printf("Второй граф:\n");

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

list2 = matrixToList(matrix2);

printList(list2);

break;

}

case 2:

case 3: {

printf("Введите номер графа для выполнения операции (1 или 2): ");

scanf("%d", &graph\_choice);

printf("Введите номера двух вершин (например, 1 3): ");

scanf("%d %d", &vertex1, &vertex2);

if (graph\_choice == 1) {

if (operation == 2) {

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

}

else {

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

}

printMatrix(matrixCopy);

printList(matrixToList(matrixCopy));

}

else if (graph\_choice == 2) {

if (operation == 2) {

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

}

else {

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

}

printMatrix(matrixCopy);

printList(matrixToList(matrixCopy));

}

break;

}

case 4: {

printf("Введите номер графа для выполнения операции (1 или 2): ");

scanf("%d", &graph\_choice);

printf("Введите номер вершины для расщепления: ");

scanf("%d", &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 5: {

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

printMatrix(mergedGraph);

break;

}

case 6: {

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

printMatrix(intersectionGraph);

break;

}

case 7: {

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

printMatrix(ringSumGraph);

break;

}

case 8: {

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

printMatrix(cartesianProductGraph);

break;

}

}

default:

printf("Неверный выбор операции.\n");

break;

}

}

return 0;

}