Министерство науки и высшего образования

Пензенский государственный университет

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №6

по дисциплине: "Логика и основы алгоритмизации в инженерных задачах"

на тему: "Унарные и бинарные операции на графами"

Выполнили:

студенты группы 23ВВВ4

Королёв Д.В.

Алешин К.А.

Приняли:

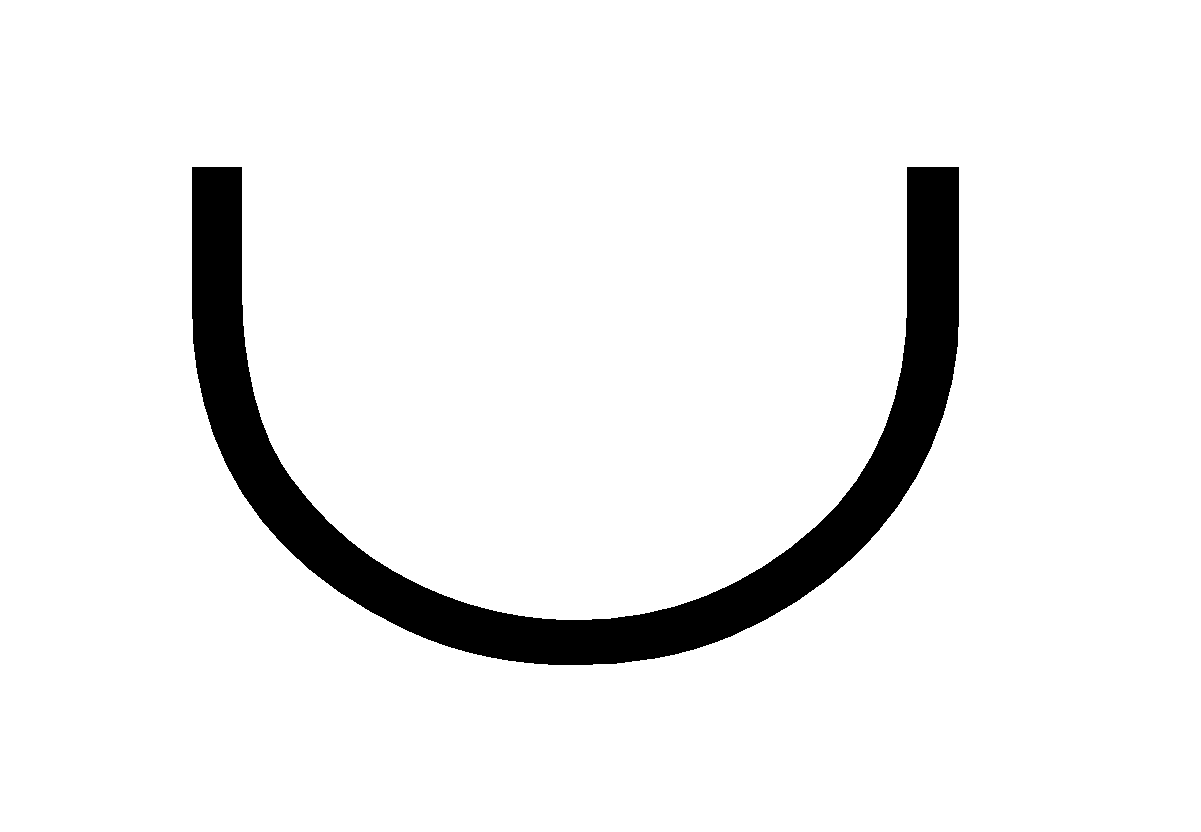
Юрова О.В.

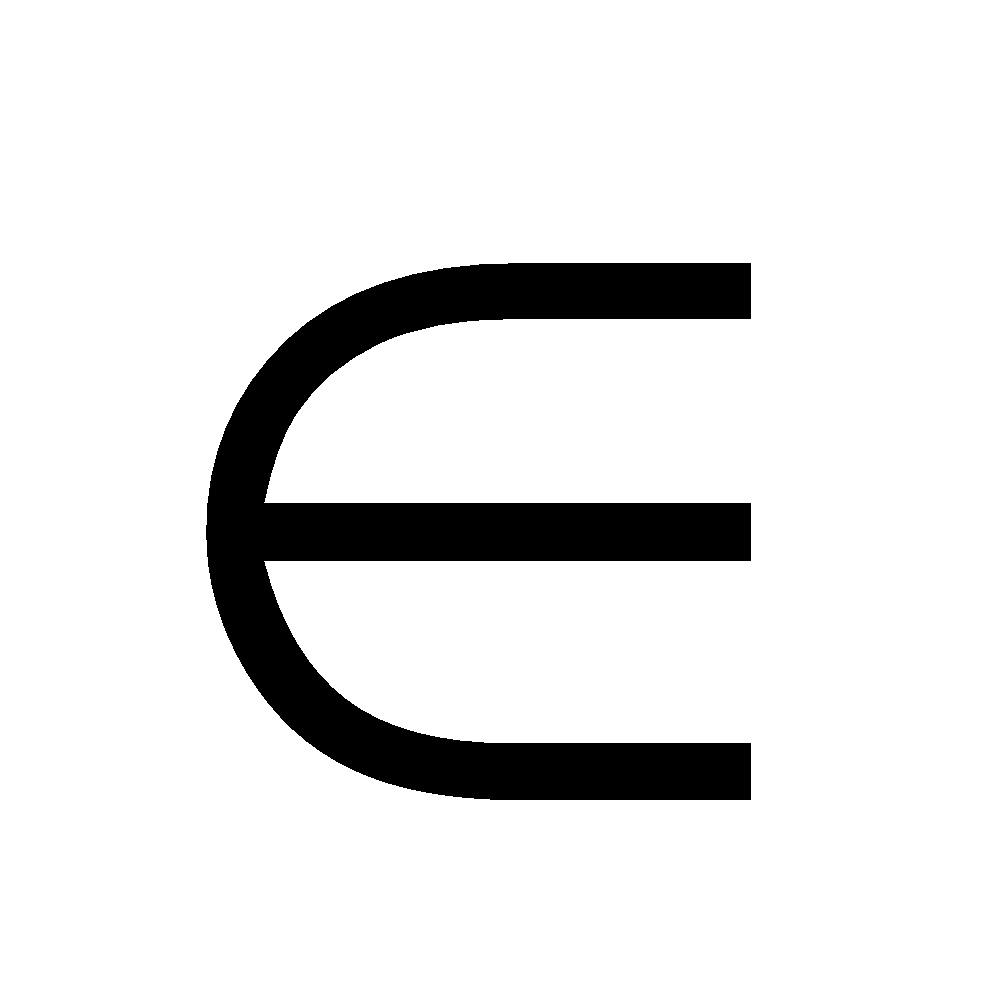
Деев М.В.

Пенза, 2024

**Цель работы** - освоение работы с основными операциями над графами

**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.

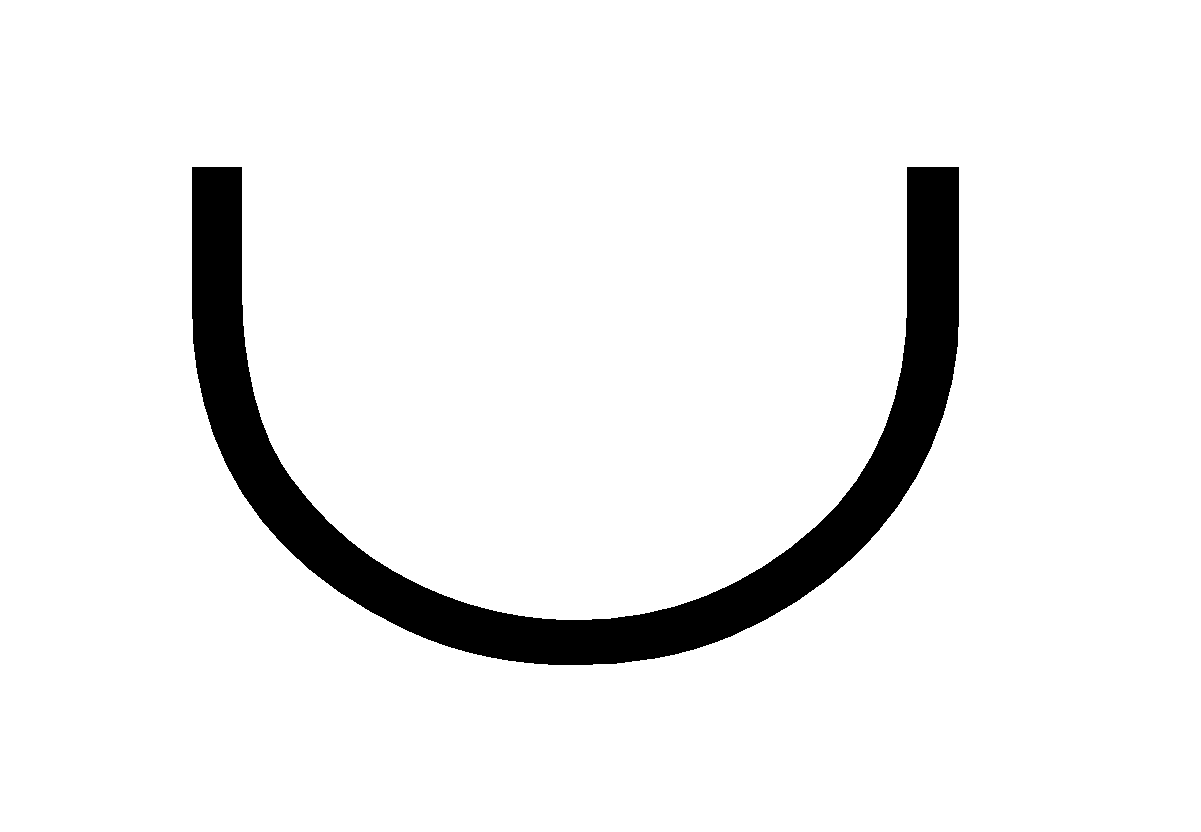
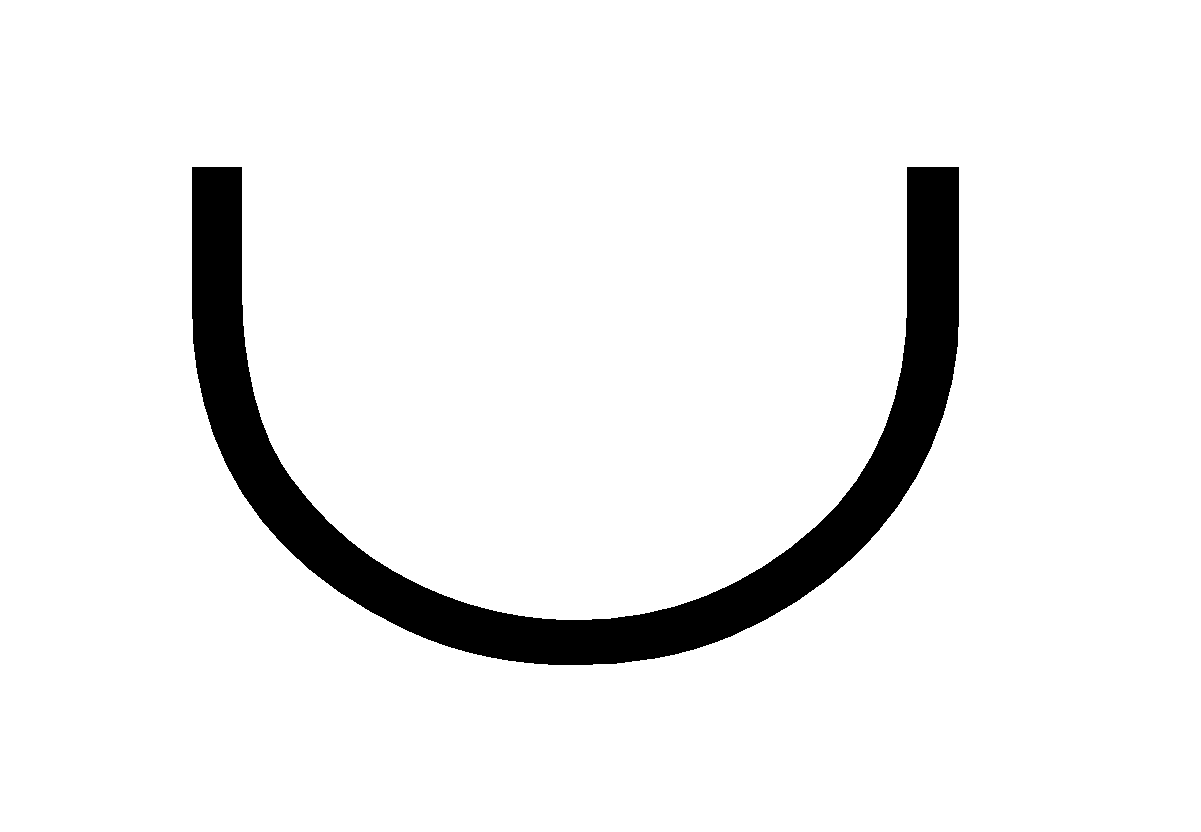
*Отождествление вершин.* В графе *G*1 выделяются вершины *и,v.* Определяют окружение *Q*1 вершины *u*,и окружение *Q*2 вершины *v,* вычисляют их объединение *Q* = *Q1* *Q2.* Затем над графом *G*1 выполняются следующие преобразования:

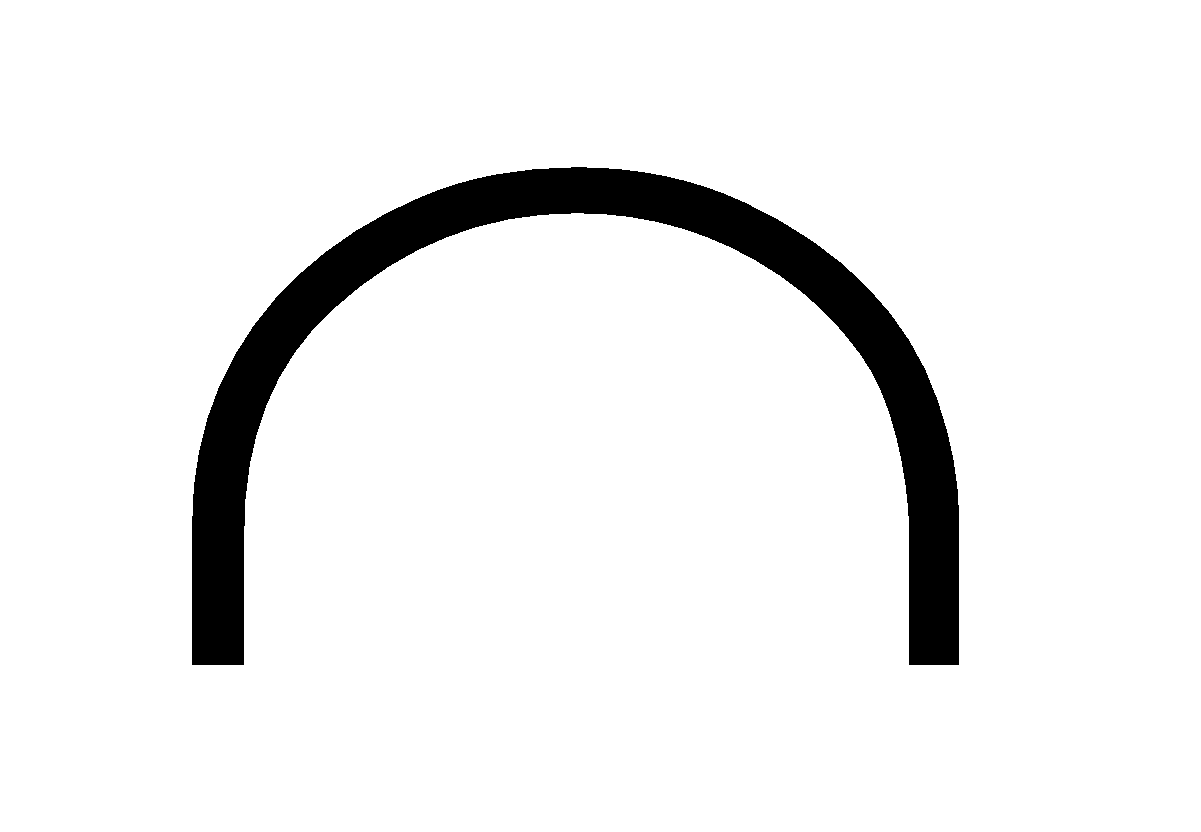
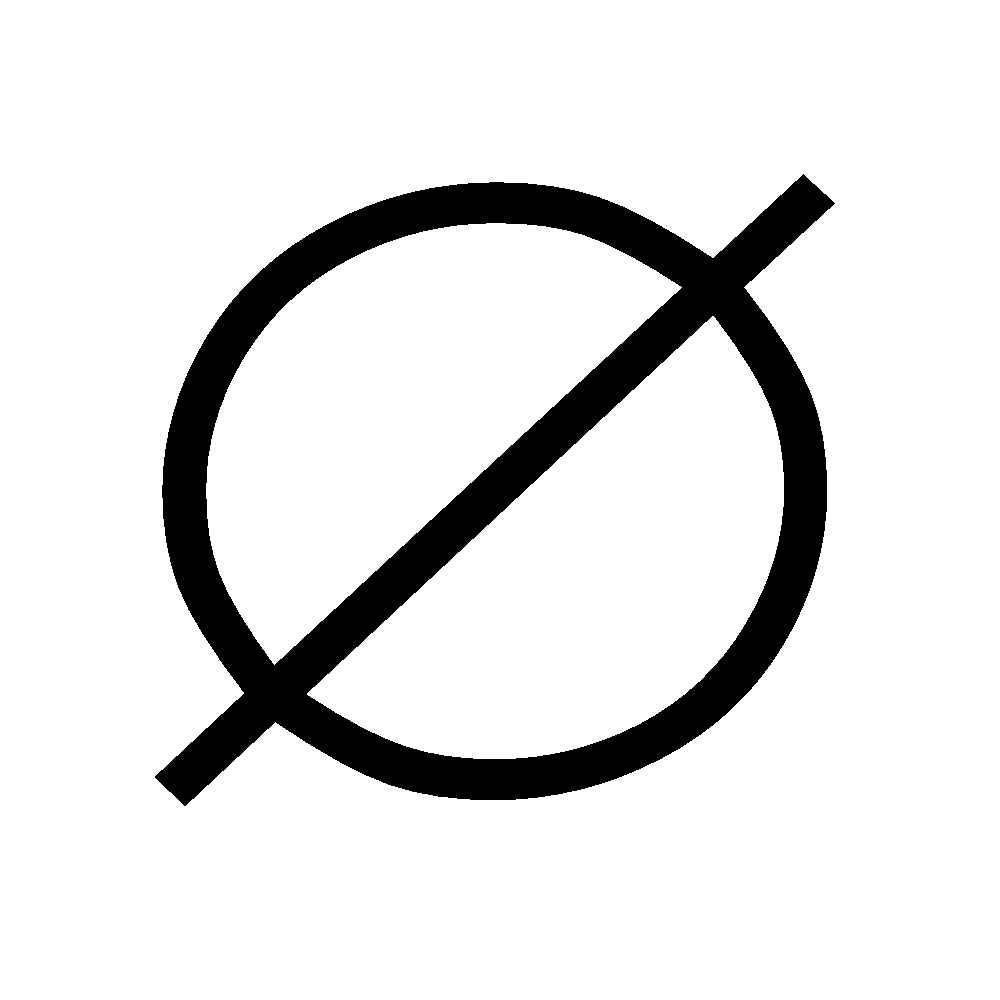
* из графа *G*1 удаляют вершины *u,* *v (H*1 *= G*1 *- u - v);*
* к графу *Н*1присоединяют новую вершину *z (H*1 *= H*1 *+z);*
* вершину *z* соединяют ребром с каждой из вершин *w*1*Q*

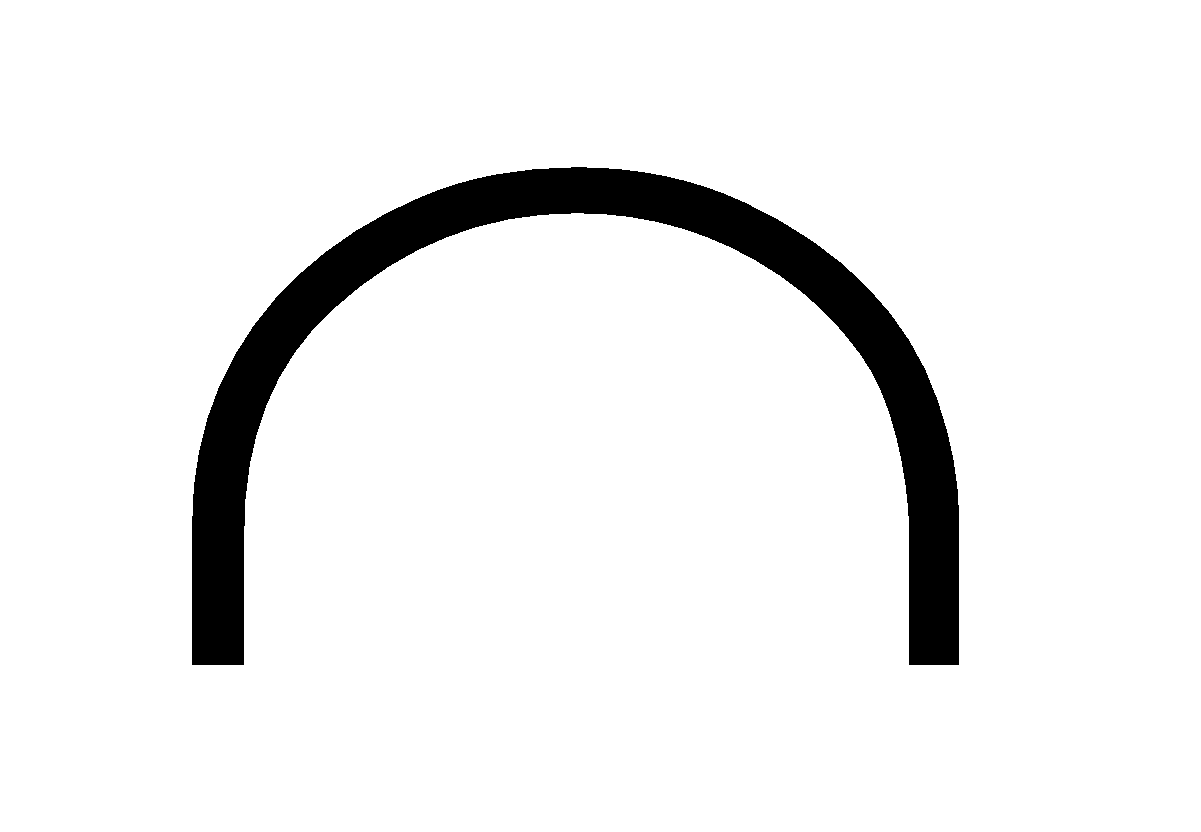
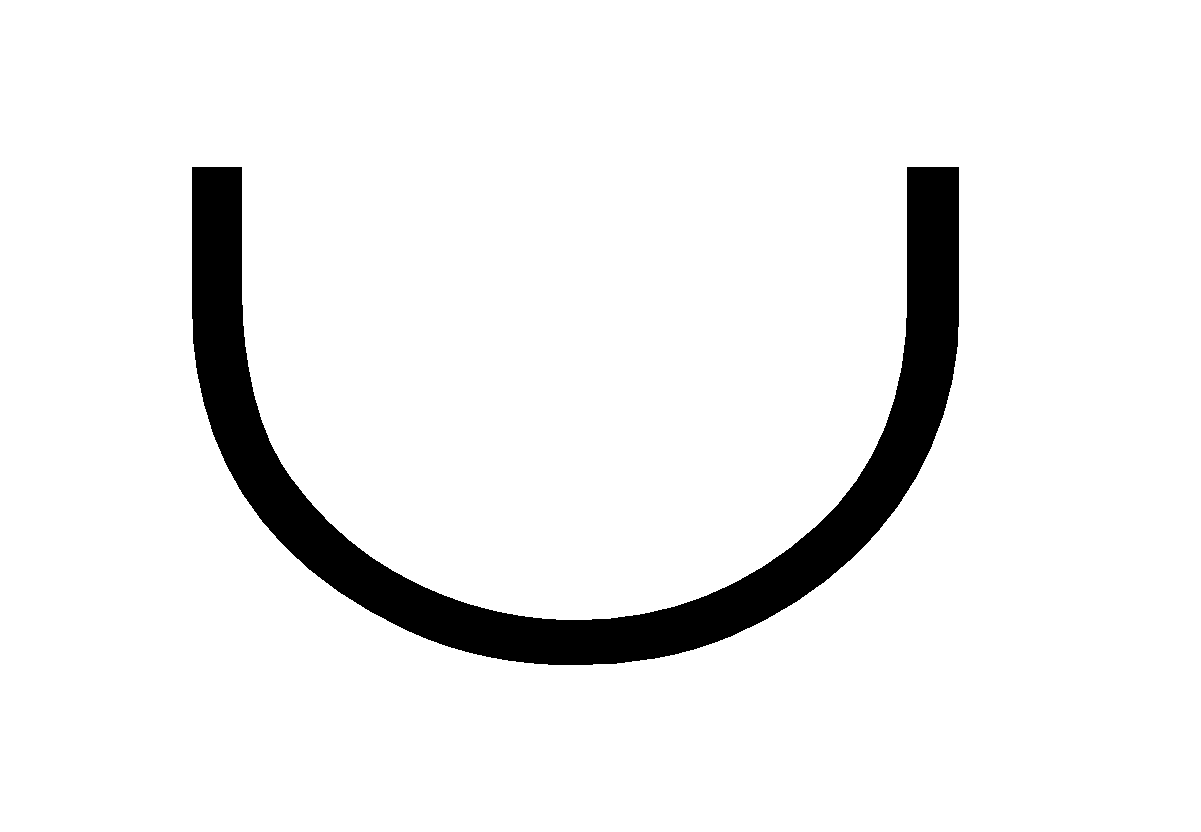
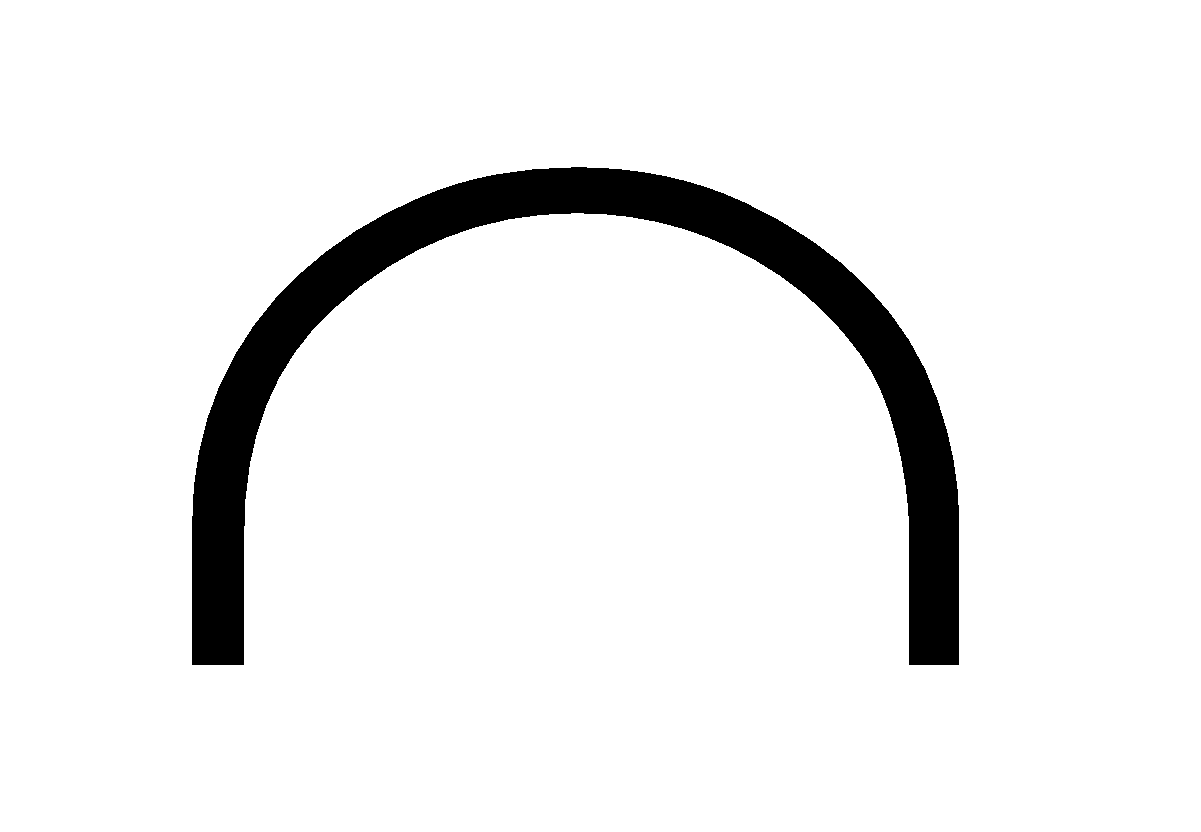
*(G*2 *= H*1 *+ zwi*, *i =* 1,2,3*,…).*

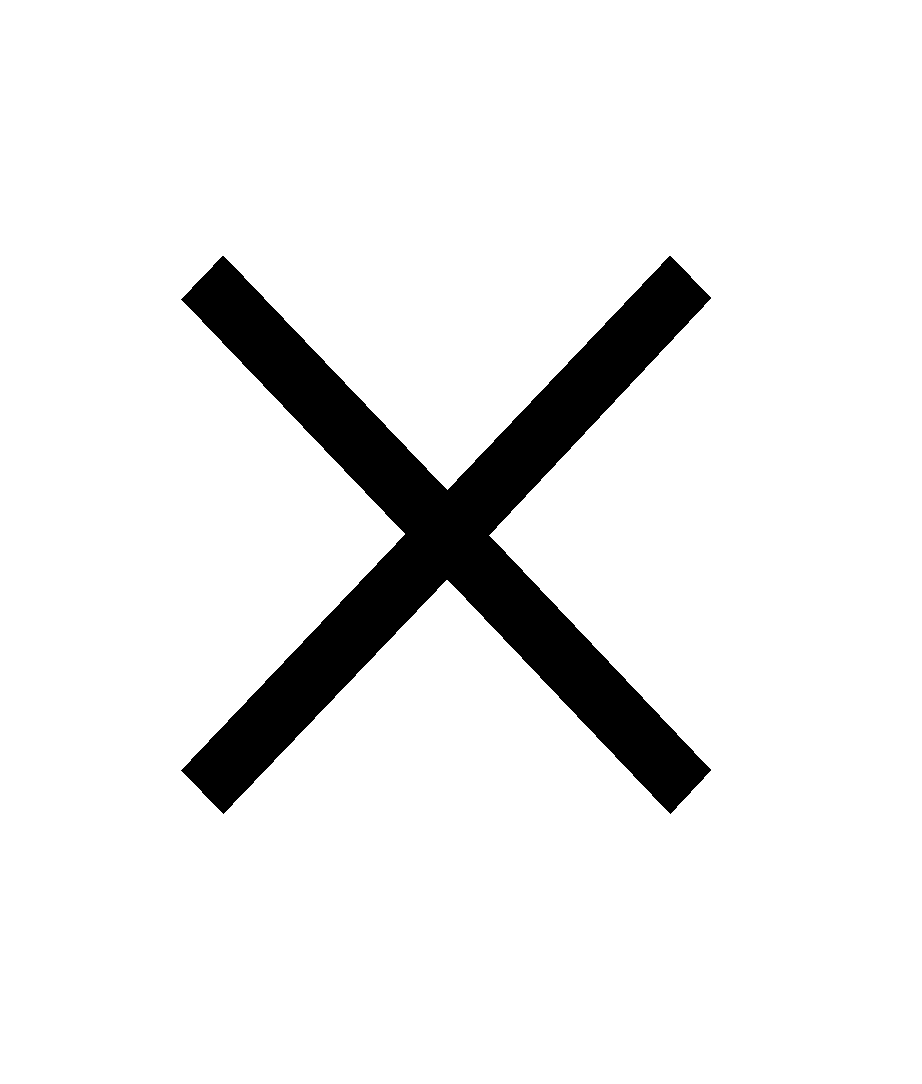
*Стягивание ребра.* Данная операция является операцией отождествления смежных вершин *и, v* в графе *G*1.

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).

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*.*

*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).

**Задание 1.**

**Листинг**

#include <iostream>

#include <cstdlib>

#include <ctime>

int\*\* create\_weighted\_adjacency\_matrix(size\_t size) {

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

if (!matrix) {

return nullptr;

}

for (size\_t i = 0; i < size; ++i) {

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

if (!matrix[i]) {

for (size\_t j = 0; j < i; ++j) {

free(matrix[j]);

}

free(matrix);

return nullptr;

}

}

for (size\_t i = 0; i < size; ++i) {

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

matrix[i][j] = rand() % 9 + 1;

matrix[j][i] = matrix[i][j];

}

}

for (size\_t i = 0; i < size; ++i) {

matrix[i][i] = 0;

}

return matrix;

}

void print\_matrix(int\*\* matrix, size\_t size) {

for (size\_t i = 0; i < size; ++i) {

for (size\_t j = 0; j < size; ++j) {

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

}

void free\_matrix(int\*\* matrix, size\_t size) {

if (!matrix) {

return;

}

for (size\_t i = 0; i < size; ++i) {

free(matrix[i]);

}

free(matrix);

}

void start() {

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

std::cout << "Enter size of the matrix (type size\_t): ";

size\_t size;

std::cin >> size;

if (!size) {

return;

}

int\*\* m1 = create\_weighted\_adjacency\_matrix(size);

if (!m1) {

std::cerr << "Error allocating memory for matrix 1\n";

return;

}

std::cout << "Matrix M1:\n";

print\_matrix(m1, size);

std::cout << std::endl;

int\*\* m2 = create\_weighted\_adjacency\_matrix(size);

if (!m2) {

std::cerr << "Error allocating memory for matrix 2\n";

return;

}

std::cout << "Matrix M2:\n";

print\_matrix(m2, size);

free\_matrix(m1, size);

free\_matrix(m2, size);

std::cout << std::endl;

}

int main() {

start();

}

Результат работы программы

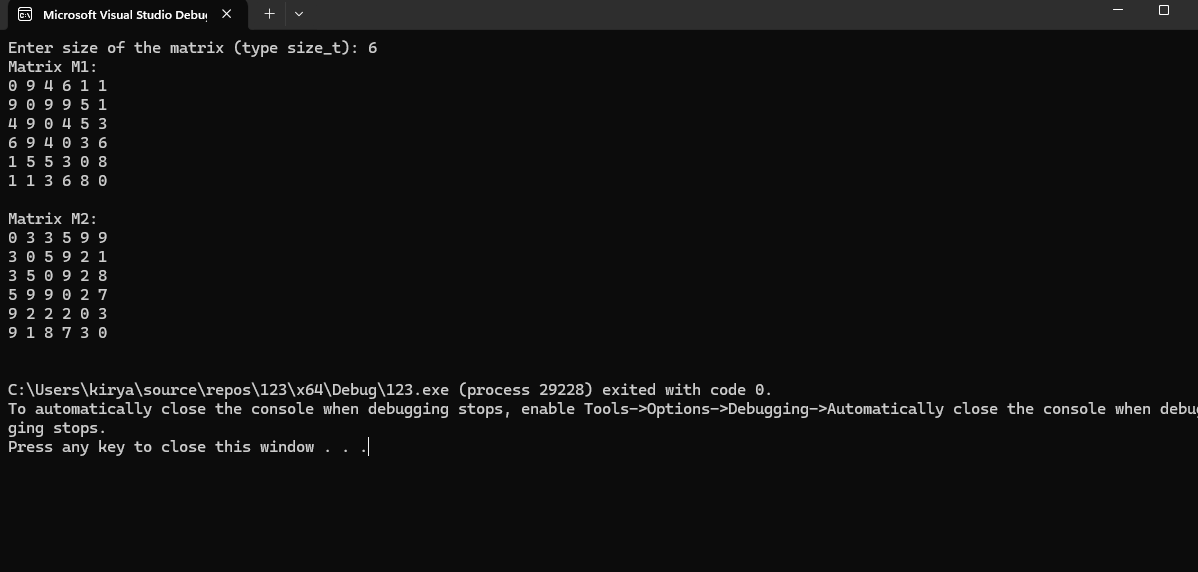


Рисунок № 1

**Задание 2.**

**Листинг**

#include <iostream>

#include <cstdlib>

#include <ctime>

int\*\* create\_weighted\_adjacency\_matrix(size\_t size)

{

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

if (!matrix)

{

return nullptr;

}

for (size\_t i = 0; i < size; ++i)

{

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

if (!matrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(matrix[j]);

}

free(matrix);

return nullptr;

}

}

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = i + 1; j < size; ++j)

{

matrix[i][j] = rand() % 5;

matrix[j][i] = matrix[i][j];

}

}

for (size\_t i = 0; i < size; ++i)

{

matrix[i][i] = 0;

}

return matrix;

}

void print\_matrix(int\*\* matrix, size\_t size)

{

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

}

void free\_matrix(int\*\* matrix, size\_t size)

{

if (!matrix)

{

return;

}

else

{

for (size\_t i = 0; i < size; ++i)

{

free(matrix[i]);

}

free(matrix);

}

}

void select\_random\_verticles(size\_t size, size\_t\* v1, size\_t\* v2)

{

\*v1 = rand() % size;

\*v2 = rand() % size;

while (\*v2 == \*v1)

{

\*v2 = rand() % size;

}

}

int\*\* contraction\_of\_a\_graph\_edge(int\*\* matrix, size\_t size)

{

size\_t v1, v2;

select\_random\_verticles(size, &v1, &v2);

for (size\_t i = 0; i < size; ++i)

{

if (i != v1 && i != v2)

{

matrix[v1][i] += matrix[v2][i];

matrix[i][v1] = matrix[v1][i];

}

}

for (size\_t i = 0; i < size; ++i)

{

matrix[v2][i] = 0;

matrix[i][v2] = 0;

}

return matrix;

}

int\*\* identification\_of\_vertices(int\*\* matrix, size\_t size)

{

int\*\* new\_matrix = (int\*\*)malloc((size - 1) \* sizeof(int\*));

if (!new\_matrix)

{

return nullptr;

}

for (size\_t i = 0; i < size - 1; ++i)

{

new\_matrix[i] = (int\*)malloc((size - 1) \* sizeof(int));

if (!new\_matrix[i]) {

for (size\_t j = 0; j < i; ++j)

{

free(new\_matrix[j]);

}

free(new\_matrix);

return nullptr;

}

}

size\_t v1, v2;

select\_random\_verticles(size, &v1, &v2);

size\_t new\_i = 0;

for (size\_t i = 0; i < size; ++i)

{

if (i == v2)

{

continue;

}

size\_t new\_j = 0;

for (size\_t j = 0; j < size; ++j)

{

if (j == v2)

{

continue;

}

if (i == v1 || j == v1)

{

new\_matrix[new\_i][new\_j] = matrix[v1][j] + matrix[v2][j == v1 ? v2 : j];

if (i == v1 && j == v1)

{

new\_matrix[new\_i][new\_j] = 0;

}

}

else

{

new\_matrix[new\_i][new\_j] = matrix[i][j];

}

++new\_j;

}

++new\_i;

}

return new\_matrix;

}

int\*\* graph\_vertex\_splits(int\*\* matrix, size\_t size)

{

size\_t new\_size = 2 \* size;

int\*\* new\_matrix = (int\*\*)malloc(new\_size \* sizeof(int\*));

if (!new\_matrix)

{

return nullptr;

}

for (size\_t i = 0; i < new\_size; i++)

{

new\_matrix[i] = (int\*)malloc(new\_size \* sizeof(int));

if (!new\_matrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(new\_matrix[j]);

}

free(new\_matrix);

return nullptr;

}

for (size\_t j = 0; j < new\_size; j++)

{

new\_matrix[i][j] = 0;

}

}

for (size\_t i = 0; i < size; i++)

{

for (size\_t j = 0; j < size; j++)

{

if (matrix[i][j] != 0)

{

int weight = matrix[i][j] / 2;

new\_matrix[i][j + size] = weight;

new\_matrix[j + size][i] = weight;

}

}

new\_matrix[i][i + size] = 1;

new\_matrix[i + size][i] = 1;

}

return new\_matrix;

}

void print\_identification\_of\_vertices(int\*\* matrix, size\_t size)

{

int\*\* new\_matrix = identification\_of\_vertices(matrix, size);

if (!new\_matrix)

{

return;

}

std::cout << "after identification\_of\_vertices\n";

print\_matrix(new\_matrix, size - 1);

free\_matrix(new\_matrix, size - 1);

std::cout << "\n\n";

}

void print\_contraction\_of\_a\_graph\_edge(int\*\* matrix, size\_t size)

{

int\*\* new\_matrix = contraction\_of\_a\_graph\_edge(matrix, size);

if (!new\_matrix)

{

return;

}

std::cout << "after contraction\_of\_a\_graph\_edge\n";

print\_matrix(new\_matrix, size);

free\_matrix(new\_matrix, size);

std::cout << "\n\n";

}

void print\_graph\_vertex\_splits(int\*\* matrix, size\_t size)

{

int\*\* new\_matrix = graph\_vertex\_splits(matrix, size);

if (!new\_matrix)

{

return;

}

std::cout << "after graph\_vertex\_splits\n";

print\_matrix(new\_matrix, 2 \* size);

free\_matrix(new\_matrix, 2 \* size);

std::cout << "\n\n";

}

void start()

{

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

while (true)

{

std::cout << "Enter size matrix (type size\_t) : ";

size\_t size;

std::cin >> size;

if (!size)

{

return;

}

int\*\* matrix = create\_weighted\_adjacency\_matrix(size);

if (!matrix)

{

std::cerr << "Error allocate\n";

return;

}

print\_matrix(matrix, size);

std::cout << std::endl;

std::cout << "identification of vertices: 1\n";

std::cout << "rib contractions: 2\n";

std::cout << "graph vertex splits: 3\n";

std::cout << "Enter operation (type int) OR -1 for finish: ";

int choice;

std::cin >> choice;

std::cout << std::endl;

if (choice == -1)

{

std::cout << "Finish\n";

return;

}

else if (choice == 1)

{

print\_identification\_of\_vertices(matrix, size);

}

else if (choice == 2)

{

print\_contraction\_of\_a\_graph\_edge(matrix, size);

}

else if (choice == 3)

{

print\_graph\_vertex\_splits(matrix, size);

}

else

{

std::cout << "Wrong opetrarino\n";

}

}

}

int main()

{

start();

return 0;

}

**Результат работы программы**

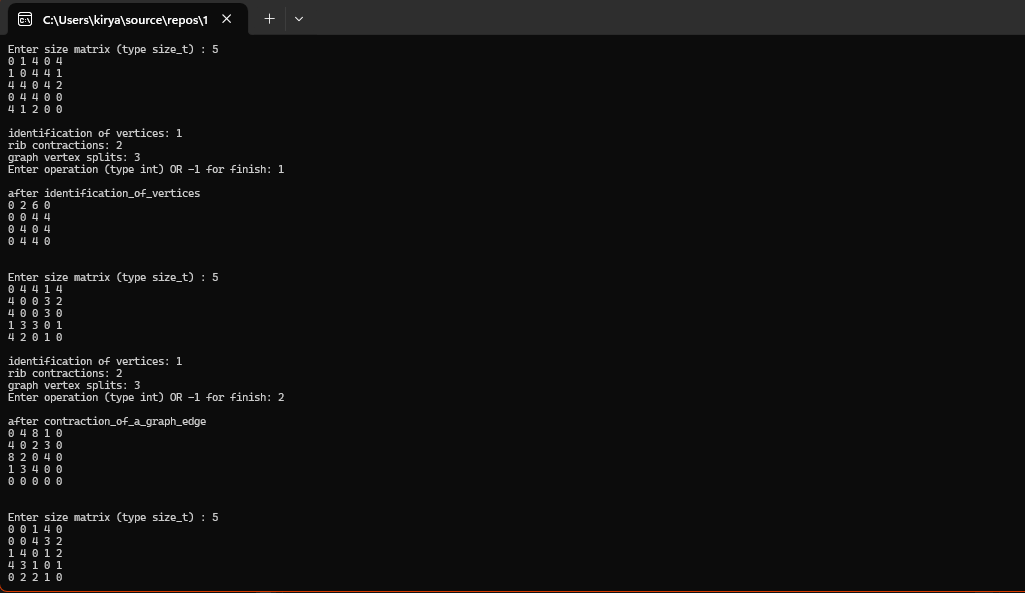


Рисунок 2)

**Задание 3.**

**Листинг**

#include <iostream>

#include <cstdlib>

#include <ctime>

int\*\* create\_weighted\_adjacency\_matrix(size\_t size)

{

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

if (!matrix)

{

return nullptr;

}

for (size\_t i = 0; i < size; ++i)

{

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

if (!matrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(matrix[j]);

}

free(matrix);

return nullptr;

}

}

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = i + 1; j < size; ++j)

{

matrix[i][j] = rand() % 5;

matrix[j][i] = matrix[i][j];

}

}

for (size\_t i = 0; i < size; ++i)

{

matrix[i][i] = 0;

}

return matrix;

}

void print\_matrix(int\*\* matrix, size\_t size)

{

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

}

void free\_matrix(int\*\* matrix, size\_t size)

{

if (!matrix)

{

return;

}

else

{

for (size\_t i = 0; i < size; ++i)

{

free(matrix[i]);

}

free(matrix);

}

}

int\*\* union\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

size\_t new\_size = size \* 2;

int\*\* mergedMatrix = (int\*\*)malloc(new\_size \* sizeof(int\*));

if (!mergedMatrix)

{

return nullptr;

}

for (size\_t i = 0; i < new\_size; ++i)

{

mergedMatrix[i] = (int\*)malloc(new\_size \* sizeof(int));

if (!mergedMatrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(mergedMatrix[j]);

}

free(mergedMatrix);

return nullptr;

}

}

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

mergedMatrix[i][j] = matrix1[i][j];

}

}

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

mergedMatrix[size + i][size + j] = matrix2[i][j];

}

}

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = size; j < new\_size; ++j)

{

mergedMatrix[i][j] = 0;

mergedMatrix[j][i] = 0;

}

}

return mergedMatrix;

}

int\*\* intersection\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

int\*\* intersected\_matrix = (int\*\*)malloc(size \* sizeof(int\*));

if (!intersected\_matrix)

{

return nullptr;

}

for (size\_t i = 0; i < size; ++i)

{

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

if (!intersected\_matrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(intersected\_matrix[j]);

}

free(intersected\_matrix);

return nullptr;

}

for (size\_t j = 0; j < size; ++j)

{

intersected\_matrix[i][j] = matrix1[i][j] < matrix2[i][j] ? matrix1[i][j] : matrix2[i][j];

}

}

return intersected\_matrix;

}

int\*\* ring\_sum\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

int\*\* summed\_matrix = (int\*\*)malloc(size \* sizeof(int\*));

if (!summed\_matrix)

{

return nullptr;

}

for (size\_t i = 0; i < size; ++i)

{

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

if (!summed\_matrix[i])

{

for (size\_t j = 0; j < i; ++j)

{

free(summed\_matrix[j]);

}

free(summed\_matrix);

return nullptr;

}

for (size\_t j = 0; j < size; ++j)

{

size\_t i\_next = (i + 1) % size;

size\_t j\_next = (j + 1) % size;

summed\_matrix[i][j] = matrix1[i][j] + matrix2[i\_next][j\_next];

}

}

return summed\_matrix;

}

void print\_union\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

int\*\* matrix = union\_matrix(matrix1, matrix2, size);

std::cout << "after union\_matrix\n";

print\_matrix(matrix, size);

free\_matrix(matrix1, size);

free\_matrix(matrix2, size);

std::cout << "\n\n";

free\_matrix(matrix, size);

return;

}

void print\_intersection\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

int\*\* matrix = intersection\_matrix(matrix1, matrix2, size);

std::cout << "after intersection\_matrix\n";

print\_matrix(matrix, size);

free\_matrix(matrix1, size);

free\_matrix(matrix2, size);

std::cout << "\n\n";

free\_matrix(matrix, size);

return;

}

void print\_ring\_sum\_matrix(int\*\* matrix1, int\*\* matrix2, size\_t size)

{

int\*\* matrix = ring\_sum\_matrix(matrix1, matrix2, size);

std::cout << "after ring\_sum\_matrix\n";

print\_matrix(matrix, size);

free\_matrix(matrix1, size);

free\_matrix(matrix2, size);

std::cout << "\n\n";

free\_matrix(matrix, size);

return;

}

void start()

{

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

while (true)

{

std::cout << "Enter size matrix (type size\_t) : ";

size\_t size;

std::cin >> size;

if (!size)

{

return;

}

int\*\* matrix1 = create\_weighted\_adjacency\_matrix(size);

int\*\* matrix2 = create\_weighted\_adjacency\_matrix(size);

if (!matrix1)

{

std::cerr << "Error allocate matrix1\n";

return;

}

if (!matrix2)

{

std::cerr << "Error allocate matrix1\n";

return;

}

print\_matrix(matrix1, size);

std::cout << std::endl;

std::cout << std::endl;

print\_matrix(matrix2, size);

std::cout << "union matrix: 1\n";

std::cout << "intersection matrix: 2\n";

std::cout << "ring sum: 3\n";

std::cout << "Enter operation (type int) OR -1 for finish: ";

int choice;

std::cin >> choice;

std::cout << std::endl;

if (choice == -1)

{

std::cout << "Finish\n";

return;

}

else if (choice == 1)

{

print\_union\_matrix(matrix1, matrix2, size);

}

else if (choice == 2)

{

print\_intersection\_matrix(matrix1, matrix2, size);

}

else if (choice == 3)

{

print\_ring\_sum\_matrix(matrix1, matrix2, size);

}

else

{

std::cout << "Wrong opetrarino\n";

}

std::cout << "\n\n";

}

}

int main()

{

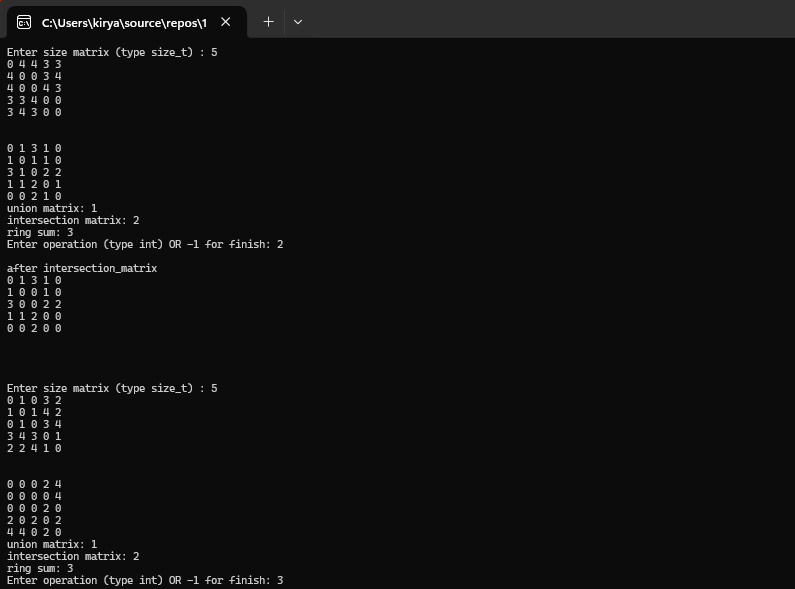
start();

return 0;

}

Результат работы программы

Рисунок 3)



**Вывод -** были получены навыки использования и написания базовых методов работы на графами.