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 u,v. Определяют окружение Q_1 вершины u, и окружение Q_2 вершины v, вычисляют их объединение $Q = Q_1 \cup Q_2$. 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 $V_G = V_1 \cup V_2$; $U_G = U_1 \cup U_2$ (рис. 1).

Объединение графов G_1 и G_2 называется дизъюнктным, если $V_1 \cap V_2 = \varnothing$. При дизъюнктном объединении никакие два из объединяемых графов не должны иметь общих вершин.

Пересечение. Граф G называется пересечением графов G_1, G_2, если $V_G = V_1 \cap V_2$ и $U_G = U_1 \cup U_2$ (рис.2). Операция "пересечения" записывается следующим образом: $G = G_1 \cap G_2$.

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).

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

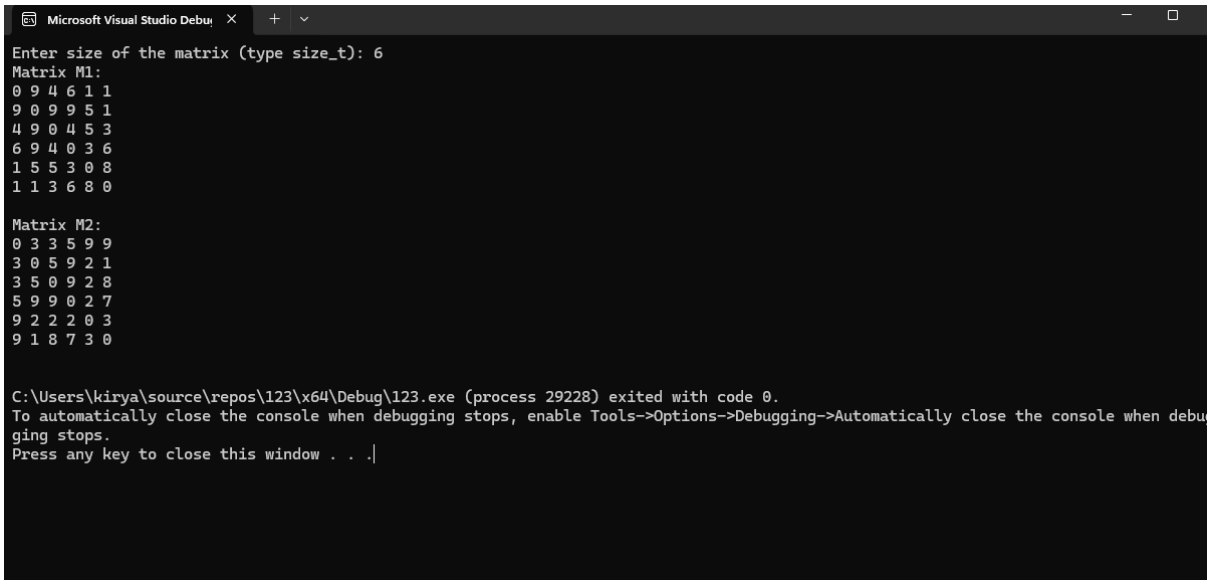


Рисунок № 1

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

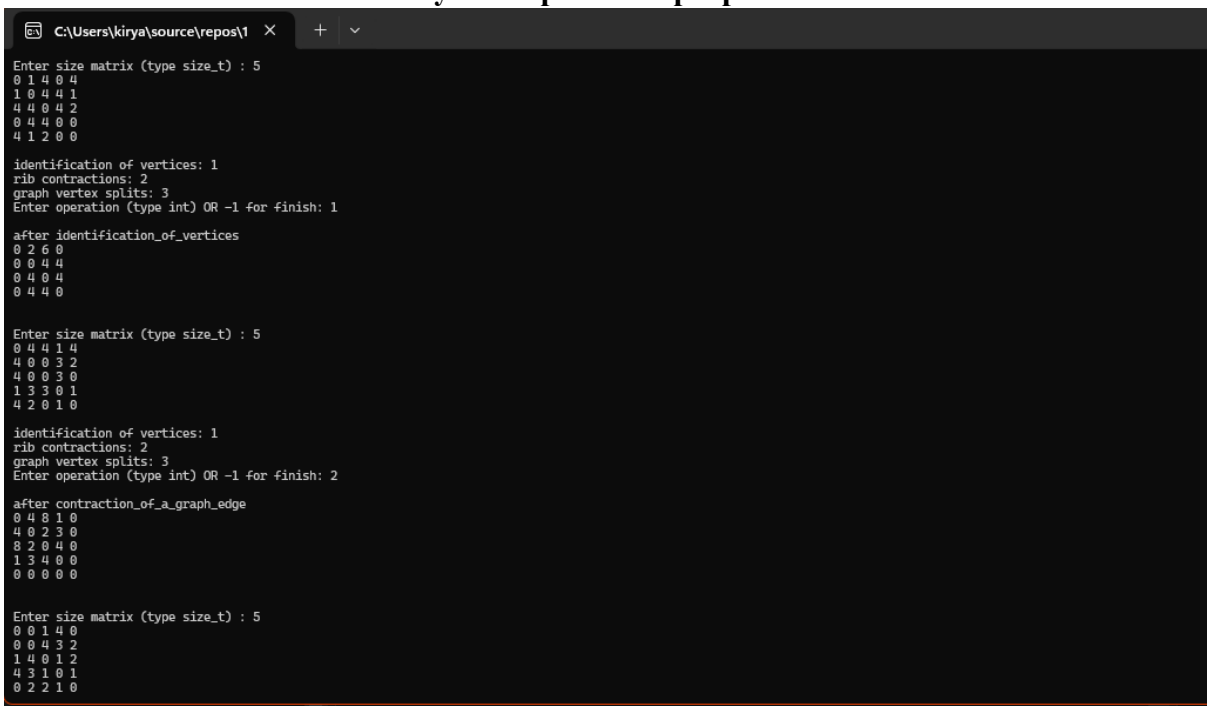


Рисунок 2)

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)
{
```

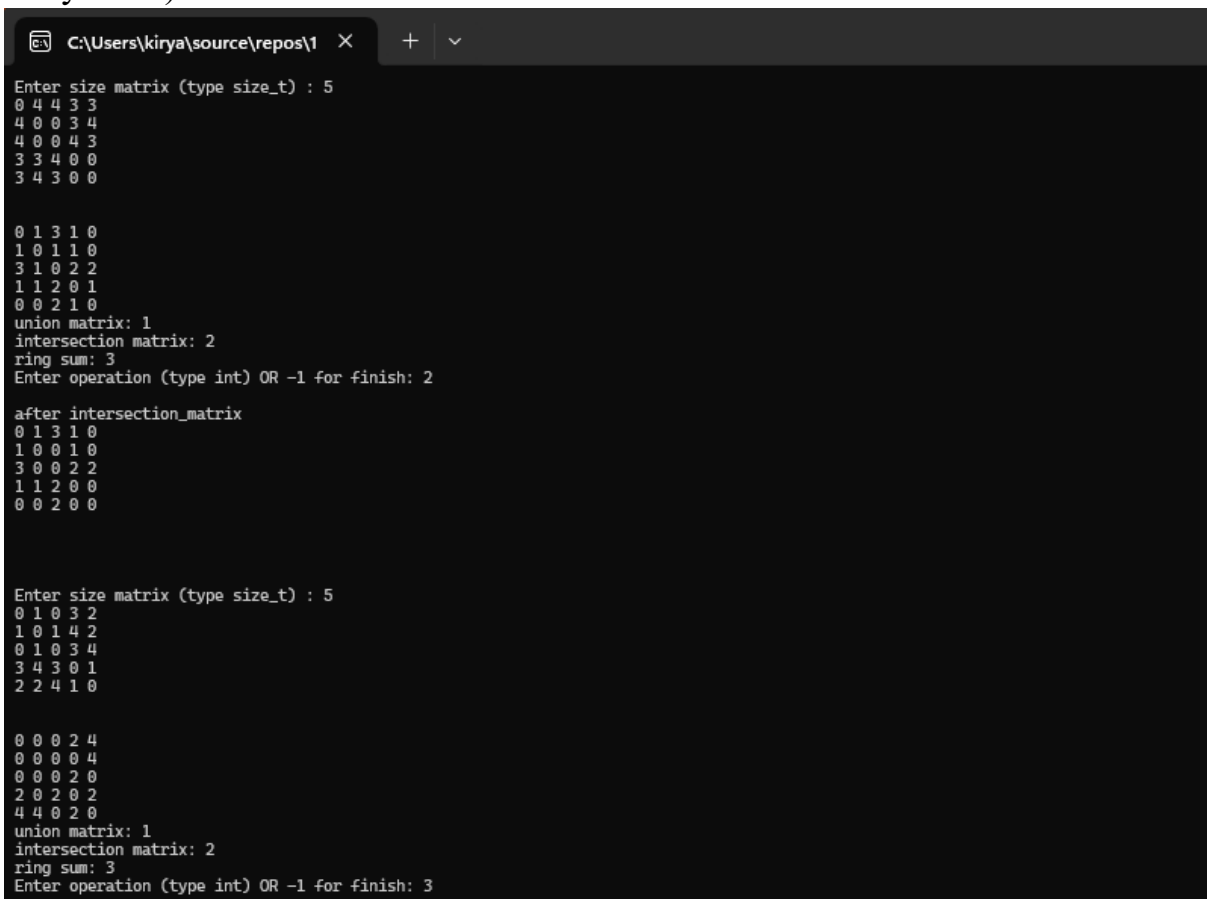
```
    int** matrix = ring_sum_matrix(matrix1, matrix2,
size);
    std::cout << "after ring_sum_matrix\n";
    print_matrix(matrix, size);

    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.