МИНИСТЕРВСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №10

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

на тему: "Поиск расстояний во взвешенном графе"

Выполнили:

Студенты группы 24ВВВ3:

Плотников И.А.

Виноградов Б.С.

Приняли:

Деев М.В.

Юрова О. В.

Пенза 2025

Цель

Изучение алгоритма поиска расстояний во взвешенном графе.

Лабораторное задание

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного взвешенного графа *G*. Выведите матрицу на экран.

2. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки C++.

3.* Сгенерируйте (используя генератор случайных чисел) матрицу смежности для ориентированного взвешенного графа *G*. Выведите матрицу на экран и осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием.

Задание 2

1. Для каждого из вариантов сгенерированных графов (ориентированного и не ориентированного) определите радиус и диаметр.

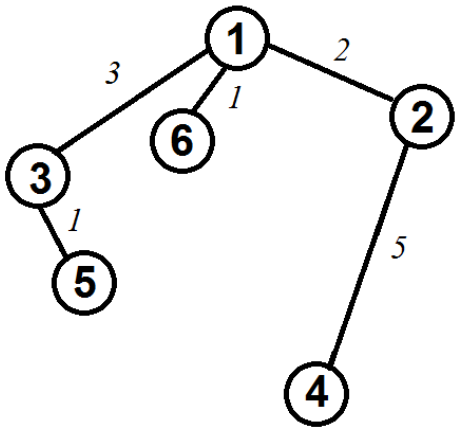
2. Определите подмножества периферийных и центральных вершин.

Задание 3*

1. Модернизируйте программу так, чтобы получить возможность запуска программы с параметрами командной строки (см. описание ниже). В качестве параметра должны указываться тип графа (взвешенный или нет) и наличие ориентации его ребер (есть ориентация или нет).

Пояснительный текст к программе

Во взвешенном графе в отличие от не взвешенного каждое ребро имеет вес, отличный от нуля. Поэтому в матрице смежности взвешенного графа содержится информация не только о наличии ребра, но и о его весе.



$$G = \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 1 \\ 2 & 0 & 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Рисунок 1 – Граф

Поиск расстояний между вершинами в таком графе также возможно построить используя процедуры обхода графа. Отличие от поиска расстояний в не взвешенном графе будет состоять в том, что при обновлении расстояния до вершины при ее посещении оно будет увеличиваться не на 1, а на величину веса ребра.

Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

Вход: G – матрица смежности графа, v – исходная вершина.

Выход: DIST – вектор расстояний до всех вершин от исходной.

Алгоритм ПОШ

1.1. для всех i положим DIST [i] = -1 пометим как "не посещенную";

1.2. **ВЫПОЛНЯТЬ** BFSD (v).

1.3 для всех i вывести DIST [i] на экран;

Алгоритм BFSD(v):

2.1. Создать пустую очередь Q = {};

2.2. Поместить v в очередь Q.push(v);

2.3. Обновить вектор расстояний DIST [x] = 0;

2.4. **ПОКА** Q != ∅ очередь не пуста **ВЫПОЛНЯТЬ**

2.5. v = Q.front() установить текущую вершину;

2.6. Удалить первый элемент из очереди Q.pop();

2.7. вывести на экран v;

2.8. **ДЛЯ** i = 1 **ДО** size_G **ВЫПОЛНЯТЬ**

2.9. **ЕСЛИ** G(v,i) > 0 **И** DIST = = -1

2.10. **ТО**

2.11. Поместить i в очередь Q.push(i);

2.12. Обновить вектор расстояний DIST [i] = DIST [v] + G(v,i);

Реализация состоит из подготовительной части, в которой все вершины помечаются как не посещенные (п.1.1). Не посещенные вершины помечаются – 1, т.к. значение 0 и 1 могут быть расстояниями. Расстояние 0 – от исходной вершины до самой себя.

В самой процедуре сначала создается пустая очередь (п. 2.1), в которую помещается исходная вершина, из которой начат обход (п.2.2). Расстояние до этой вершины (п.2.3) устанавливается равным 0 (расстояние до самой себя).

Далее итерационно, пока очередь не опустеет, из нее извлекается первый элемент, который становится текущей вершиной (п. 2.5, 2.6). Затем в цикле просматривается **v**-я строка матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с **v** не посещенную вершину (п.2.9), эта вершина помещается в очередь (п.2.11) и для нее обновляется вектор расстояния (п.2.12). Расстояние до новой **i**-й вершины вычисляется как расстояние до текущей **v**-й вершины плюс вес ребра до новой вершины G(v,i).

После просмотра строки матрицы смежности алгоритм делает следующую итерацию цикла 2.4 или заканчивает работу, если очередь пуста.

Если для всех пар вершин графа определены расстояния, то можно вычислить эксцентриситет

Если G - граф, содержащий непустое множество n вершин V и множество ребер E и $d(v_i, v_j)$ – расстояние между двумя произвольными вершинами v_i и v_j, тогда для фиксированной вершины v величина

$$e(v) = \max d(v, v_j),$$

где v, $v_j \in V$ и $j = 1 \ldots n$ называется **эксцентриситетом** вершины v_i.

Другими словами **эксцентриситет** вершины – расстояние до наиболее удаленной вершины графа.

Максимальный эксцентриситет среди эксцентриситетов всех вершин графа называется **диаметром** графа G и обозначается через $D(G)$.

Вершина v_i называется **периферийной**, если её эксцентриситет равен диаметру графа $e(v_i) = d(G)$.

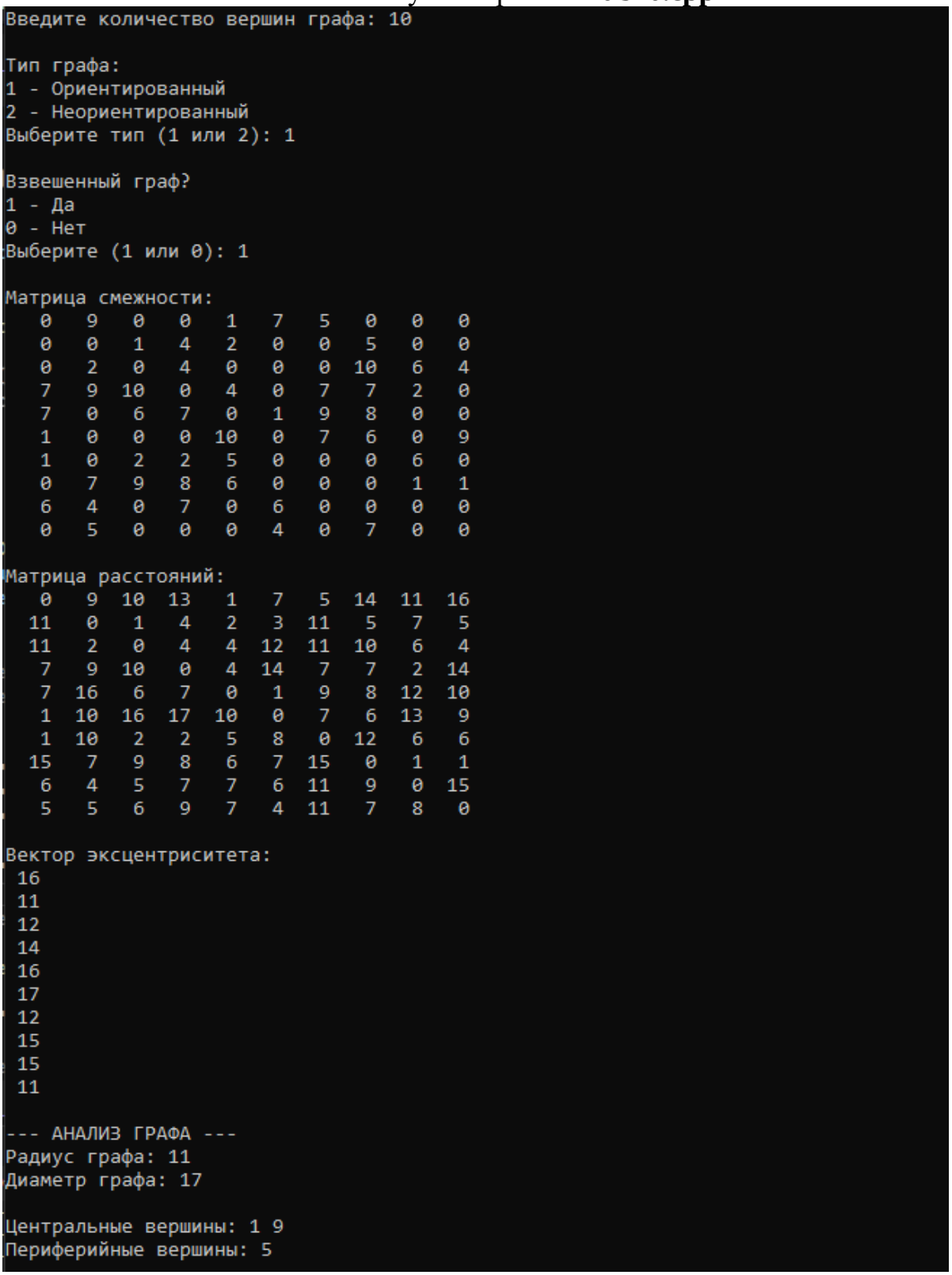
Минимальный из эксцентриситетов вершин графа называется его **радиусом** и обозначается через $r(G)$.

Вершина v_i называется **центральной**, если её эксцентриситет равен радиусу графа $e(v_i) = r(G)$.

Множество всех центральных вершин графа называется его **центром**. Граф G может иметь единственную центральную вершину или несколько центральных вершин.

Результаты работы программы

1 Рис. - Результат работы **lab10.cpp**



```
Введите количество вершин графа: 10

Тип графа:
1 - Ориентированный
2 - Неориентированный
Выберите тип (1 или 2): 1

Взвешенный граф?
1 - Да
0 - Нет
Выберите (1 или 0): 1

Матрица смежности:
   0   9   0   0   1   7   5   0   0   0
   0   0   1   4   2   0   0   5   0   0
   0   2   0   4   0   0   0  10   6   4
   7   9  10   0   4   0   7   7   2   0
   7   0   6   7   0   1   9   8   0   0
   1   0   0   0  10   0   7   6   0   9
   1   0   2   2   5   0   0   0   6   0
   0   7   9   8   6   0   0   0   1   1
   6   4   0   7   0   6   0   0   0   0
   0   5   0   0   0   4   0   7   0   0

Матрица расстояний:
   0   9  10  13   1   7   5  14  11  16
  11   0   1   4   2   3  11   5   7   5
  11   2   0   4   4  12  11  10   6   4
   7   9  10   0   4  14   7   7   2  14
   7  16   6   7   0   1   9   8  12  10
   1  10  16  17  10   0   7   6  13   9
   1  10   2   2   5   8   0  12   6   6
  15   7   9   8   6   7  15   0   1   1
   6   4   5   7   7   6  11   9   0  15
   5   5   6   9   7   4  11   7   8   0

Вектор эксцентриситета:
 16
 11
 12
 14
 16
 17
 12
 15
 15
 11

--- АНАЛИЗ ГРАФА ---
Радиус графа: 11
Диаметр графа: 17

Центральные вершины: 1 9
Периферийные вершины: 5
```

Вывод: В ходе выполнения лабораторной работы была разработана программа для выполнения заданий Лабораторной работы №10 – поиск расстояний во взвешенном графе.

Приложение А Листинг

Файл lab10.cpp

```
// (поиск расстояний, взвешенный граф),
вводить количество вершин при запуске
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <locale>
#include <limits>
#include <iomanip>
#include <queue>
#include <cstring>

using namespace std;

void clearScreen();
int isInteger(const string& message);
void bfsd(int** G, int numG, int** GD, int s);
void printMatrix(int** Matrix, int numG);
void analyzeDistances(int numG, int** GD,
int* ecc);

int main(int argc, char* argv[]) {
    setlocale(LC_ALL, "Rus");
    clearScreen();
    srand(time(NULL));

    int** G = nullptr;
    int** GD = nullptr;
    int numG = 0;
    int* ecc = nullptr;
```

```
    bool useCommandLine = false;
    bool isWeighted = false;
    bool isOriented = false;
    bool vertFromCmd = false;

    for (int i = 1; i < argc; ++i) {
        if (strcmp(argv[i], "-weighted") == 0 && i
+ 1 < argc) {
            isWeighted = (atoi(argv[i + 1]) != 0);
            useCommandLine = true;
            i++;
        }
        else if (strcmp(argv[i], "-oriented") == 0
&& i + 1 < argc) {
            isOriented = (atoi(argv[i + 1]) != 0);
            useCommandLine = true;
            i++;
        }
        else if (strcmp(argv[i], "-vert") == 0 && i
+ 1 < argc) {
            numG = atoi(argv[i + 1]);
            vertFromCmd = true;
            useCommandLine = true;
            i++;
        }
    }

    if (useCommandLine) {
        cout << "Режим: командная строка\n";
        cout << "Взвешенный: " << (isWeighted ?
"да" : "нет") << "\n";
```

```
        cout << "Ориентированный: " <<
(isOriented ? "да" : "нет") << "\n";
        if (vertFromCmd) {
            cout << "Количество вершин: " <<
numG << "\n";
        }
        cout << "\n";
    }

    if (!vertFromCmd) {
        numG = isInteger("\nВведите количество
вершин графа: ");
    }

    while (numG <= 0) {
        cout << "Ошибка! Количество вершин
должно быть положительным\n";
        numG = isInteger("Введите количество
вершин графа: ");
    }

    ecc = (int*)malloc(numG * sizeof(int));
    G = (int**)malloc(sizeof(int*) * numG);
    GD = (int**)malloc(sizeof(int*) * numG);

    for (int i = 0; i < numG; i++) {
        G[i] = (int*)malloc(numG * sizeof(int));
        GD[i] = (int*)malloc(numG * sizeof(int));
    }

    if (!useCommandLine) {
        cout << "\nТип графа:\n";
```

```
        cout << "1 — Ориентированный\n";
        cout << "2 — Неориентированный\n";
        int orientChoice = isInteger("Выберите
тип (1 или 2): ");
        while (orientChoice != 1 && orientChoice
!= 2) {
            cout << "Ошибка! Введите 1 или 2\n";
            orientChoice = isInteger("Выберите
тип (1 или 2): ");
        }
        isOriented = (orientChoice == 1);

        cout << "\nВзвешенный граф?\n";
        cout << "1 — Да\n";
        cout << "0 — Нет\n";
        int weightChoice = isInteger("Выберите
(1 или 0): ");
        while (weightChoice != 0 &&
weightChoice != 1) {
            cout << "Ошибка! Введите 0 или 1\n";
            weightChoice = isInteger("Выберите (1
или 0): ");
        }
        isWeighted = (weightChoice == 1);
    }

    for (int i = 0; i < numG; i++) {
        for (int j = 0; j < numG; j++) {
            if (i == j) {
                G[i][j] = 0;
            } else {
                int hasEdge = rand() % 2;
```

```
            if (hasEdge) {
                if (isWeighted) {
                    G[i][j] = rand() % 10 + 1;
                } else {
                    G[i][j] = 1;
                }
            } else {
                G[i][j] = 0;
            }

            if (!isOriented && i < j) {
                G[j][i] = G[i][j];
            }
        }
    }
}

cout << "\nМатрица смежности: \n";
printMatrix(G, numG);

cout << "\nМатрица расстояний: \n";
for (int i = 0; i < numG; i++) {
    bfsd(G, numG, GD, i);
}
printMatrix(GD, numG);

cout << "\nВектор эксцентриситета: \n";
for (int i = 0; i < numG; i++) {
    ecc[i] = 0;
    for (int j = 0; j < numG; j++) {
        if (GD[i][j] != -1 && GD[i][j] > ecc[i])
{
```

```
                ecc[i] = GD[i][j];
            }
        }
        cout << std::setw(3) << ecc[i] << "\n";
    }

    analyzeDistances(numG, GD, ecc);

    for (int i = 0; i < numG; i++) {
        free(G[i]);
        free(GD[i]);
    }
    free(G);
    free(GD);
    free(ecc);

    return 0;
}

void clearScreen() {
#ifdef _WIN32
    system("cls");
#else
    system("clear");
#endif
}

int isInteger(const string& message) {
    int value;
    while (true) {
        cout << message;
        if (!(cin >> value)) {
```

```
            cout << "Ошибка: введено не
число.\n";
            cin.clear();

cin.ignore(numeric_limits<streamsize>::max(),
'\n');
            continue;
        }
        if (cin.peek() != '\n') {
            cout << "Ошибка: введено не целое
число.\n";
            cin.clear();

cin.ignore(numeric_limits<streamsize>::max(),
'\n');
            continue;
        }
        return value;
    }
}

void bfsd(int** G, int numG, int** GD, int s) {
    queue<int> q;
    int v;
    int* distance = (int*)malloc(numG *
sizeof(int));

    for (int i = 0; i < numG; i++) {
        distance[i] = -1;
    }

    distance[s] = 0;
```

```
    q.push(s);

    while(!q.empty()) {
        v = q.front();
        q.pop();

        for (int i = 0; i < numG; i++) {
            if (G[v][i] > 0 && distance[i] == -1) {
                q.push(i);
                distance[i] = distance[v] + G[v][i];
            }
        }
    }
    for (int i = 0; i < numG; i++) {
        GD[s][i] = distance[i];
    }

    free(distance);
}

void printMatrix(int** Matrix, int numG){
    for (int i = 0; i < numG; i++) {
        for (int j = 0; j < numG; j++) {
            cout << std::setw(4) << Matrix[i][j];
        }
        cout << "\n";
    }
}

void analyzeDistances(int numG, int** GD,
int* ecc) {
    int radius = numeric_limits<int>::max();
```

```
    int diameter = numeric_limits<int>::min();

    for (int i = 0; i < numG; i++) {
        if (ecc[i] < radius) radius = ecc[i];
        if (ecc[i] > diameter) diameter = ecc[i];
    }

    cout << "\n--- АНАЛИЗ ГРАФА ---\n";
    cout << "Радиус графа: " << radius << endl;
    cout << "Диаметр графа: " << diameter <<
endl;

    cout << "\nЦентральные вершины: ";
    for (int i = 0; i < numG; i++) {
        if (ecc[i] == radius) cout << i << " ";
    }
    cout << endl;

    cout << "Периферийные вершины: ";
    for (int i = 0; i < numG; i++) {
        if (ecc[i] == diameter) cout << i << " ";
    }
    cout << endl;
}
```