**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Российский химико-технологический университет имени Д.И. Менделеева»**
**Кафедра информационных компьютерных технологий**

# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5

Выполнил студент группы .....КС-30.....Сидоров Сергей Александрович
Ссылка на репозиторий: ..https://github.com/MUCTR-IKT-CPP/Sidorov.S.A_KS-30_2sem/tree/main/lab5

Приняли: ..... Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович

Дата сдачи: .....30.05.2021

# Оглавление

# Описание задачи.

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

- Максимальное/Минимальное количество генерируемых вершин
- Максимальное/Минимальное количество генерируемых ребер
- Максимальное количество ребер связанных с одной вершины
- Генерируется ли направленный граф
- Максимальное количество входящих и выходящих ребер
- Ребра графа содержат веса
- Максимальное/Минимальное значения веса
- Могут ли существовать в графе недоступные области

Сгенерированный граф должен быть описан в рамках одного класса (этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:

- Выдача матрицы смежности
- Выдача матрицы инцидентности
- Выдача список смежности
- Выдача списка ребер

Поиск длиннейшего пути в графе

# Описание структуры.

Под графом в математике понимается абстракция реальной системы объектов безотносительно их природы, обладающих парными связями.

**Вершина графа** – это некоторая точка связанная с другими точками

**Ребро графа** – это линия соединяющая две точки и олицетворяющая связь между ними

**Граф** – это множество вершин соединённых друг с другом произвольным образом множеством ребер

Ориентированным графом называют такой граф в котором каждое ребро имеет направление движения, и как правило не предполагает возможности обратного перемещения.

Связанную с каким либо ребром вершину называются инцидентной, это такая вершина которая каким либо образом принадлежит ребру.

Соседними называются те вершины которые соединены между собой ребром

Помимо ориентации у графа часто можно указывать какие либо значения которые будут присущи ребрам, и при различных маршрутах на графе можно суммировать эти веса для конкретного маршрута, таким образом оценивая какие либо параметры.

Для описания графа используют один из двух удобных для вычисления вариантов:

- Матрица смежности, это двумерная таблица для которой столбцы и строки соответствуют вершинам, а значения в таблицы соответствуют ребрам, для невзвешенного графа они могут быть просто 1 если связь есть и идет в нужном направлении и 0 если ее нет, а для взвешенного графа будут стоять конкретные значения.
- Матрица инциндетности, это матрица в которой строки соответствуют вершинам, а столбцы соответствуют связям, и ячейки ставиться 1 если связь выходит из вершины, -1 если входит и 0 во всех остальных случаях.
- Список смежности, это список списков, содержащий все вершины, а внутренние списки для каждой вершины содержат все смежные ей.
- Список ребер, это список строк в которых хранятся все ребра вершины, а внутренние значение содержит две вершины к которым присоединено это ребро.

Для поиска кратчайшего пути в графе часто используют поиск в ширину, так же существует некоторый набор различных решений этой задачи:

- Алгоритм Дейкстры находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

## Выполнение задачи.

Реализация данной работы делится на 2 части. Первая – класс графа. Вторая – класс генератора графов.

Рассмотрим реализацию самого графа.

Для использования в графе используются 3 структуры – структура узла:

```
struct GraphVertex
{
    int value, len;
    GraphVertex* adj;
};
```

Структура ребра с весом (для взвешенного графа):

```
struct EdgeWithWeight {
    int from_v, to_v, length;
};
```

И структура ребра без веса (для не взвешенного графа):

```
struct EdgeWithoutWeight {
    int from_v, to_v;
};
```

Они используются для хранения данных о ребрах и узлах в основном классе графа. В данном классе имеется несколько свойств:

```
int N;  // Общее кол-во узлов в графе
    int N_EDGES = 0;  // Общее кол-во ребер
    bool* visited;
    vector<vector<int>> adjLists;

public:

    bool directed;  // Направленный ли граф
    bool weighted;  // Взвешенный ли граф
    GraphVertex** heading;  // Массив указателей на узлы, для представления списка смежнос
ти

    EdgeWithoutWeight* edges_no_lengt = nullptr;  // Указатель на массив не взвешенных реб
ер
    EdgeWithWeight* edges_lengt = nullptr;  // Указатель на массив взвешенных ребер
```

Первые два свойства используются для обозначения текущего числа узлов и ребер соответственно. Публичные свойства directed и wieghted определяют основные два варианта графа – ориентирован ли он и взвешен ли он. Указатель на указатели типа Node – это некий массив, который используется для хранения данных о графе для быстрого выведения списка смежности. Указатели на структуры взвешенных и не взвешенных ребер используются для хранения в них самих ребер графа.

Так как по условию задачи на граф нужно было отвести строго 1 класс, то у класса Graph существует 2 конструктора, которые применяются для создания взвешенного или не взвешенного графа. Они работают идентично друг другу, единственное их отличие в том – массив каких ребер они принимают и массив каких ребер заполняют. Рассмотрим конструктор для взвешенных графов:

```
DiffGraph(vector<EdgeWithWeight> edges, int n, int N, bool directed)
    {
        visited = new bool[N];

        edges_lengt = new EdgeWithWeight[n];
        for(int i = 0; i < n; i++){
            edges_lengt[i] = edges[i];
        }
        // Выставляем то, что граф взвешен
        this->weighted = true;

        // Выставляем тип графа
        this->directed = directed;

        // Выделяем память
        heading = new GraphVertex*[N]();
        this->N = N;

        // Инициализируем указатели на направления для всех вершин
        for (int i = 0; i < N; i++) {
            heading[i] = nullptr;
        }

        // Добавляем ребра направленному графу
        for (unsigned i = 0; i < n; i++)
        {
            int from_v = edges[i].from_v;
            int to_v = edges[i].to_v;
            int length = edges[i].length;

            // Вставка в начало
            GraphVertex* newNode = adjListForVertex(to_v, length, heading[from_v]);

            // Указатели направления должны указывать на новый узел
            heading[from_v] = newNode;
            this->N_EDGES++;

            // Если граф не направленный, то каждому направлению создаем ещё и обратное
            if(!this->directed){

                newNode = adjListForVertex(from_v, length, heading[to_v]);

                // Меняем направление, чтобы оно вело на новый узел
                heading[to_v] = newNode;
            }
        }
        if(!this->directed){
            reverseEdges();
```

```
        }
        adjLists = adjacencyMat();
    }
```

Конструктор принимает все необходимые параметры, такие как вектор ребер, кол-во ребер, кол-во вершин и логический параметр – ориентирован ли граф.

В конструкторе происходит заполнение всех свойств класса, а так же обработка входных данных. Узлы создаются с помощью метода getAdjListNode – это метод, который принимает вершину, из которой выходит ребро, его вес, и куда оно направлено. Возвращает указатель на структуру узел. Листинг приведен ниже:

```
GraphVertex* adjListForVertex(int valueue, int length, GraphVertex* heading)
      {
          GraphVertex* newNode = new GraphVertex;
          newNode->value = valueue;
          newNode->len = length;

          // Новый узел должен указывать на текущее направление
          newNode->adj = heading;

          return newNode;
      }
```

Этот метод, также идет в двух перегрузках – для взвешенного и не взвешенного графа.

После заполнения свойств класса, конструктор вызывает метод completeEdges (если граф должен быть не ориентированным). Данный метод удаляет все повторения в ребрах (если таковые имеются), и дублирует каждое ребро в обратном направлении – это связано с реализацией графа. В обоих случаях – и для ориентированного случая и для не ориентированного, технически он всегда является ориентированным, однако во втором случае каждому ребру создается противоположное по направлению – это делает возможным реализацию в рамках строго 1 класса:

```
void reverseEdges(){
        if(this->weighted){
            vector<EdgeWithWeight> tmp_edges(this->N_EDGES);
            for(int i = 0; i < this->N_EDGES; i++){
                tmp_edges[i] = this->edges_lengt[i];
            }

            // Ищем
            for(int i = 0; i < this->N_EDGES; i++){
                EdgeWithWeight tmp_el = tmp_edges[i];
                for( auto iter = tmp_edges.begin(); iter != tmp_edges.end() - 1; iter++ ){
                    if(( iter->from_v == tmp_el.to_v) && (iter->to_v == tmp_el.from_v)){
                        tmp_edges.erase(iter);
                        break;
                    }

                }
                cout << i;
            }
```

```
            int tmp_size = tmp_edges.size();
            for(int i = 0; i < tmp_size; i++){
                tmp_edges.push_back({tmp_edges[i].to_v, tmp_edges[i].from_v, tmp_edges[i].
length});
            }

            delete[] edges_lengt;
            edges_lengt = new EdgeWithWeight[tmp_edges.size()];
            for(int i = 0; i < tmp_edges.size(); i++){
                edges_lengt[i] = tmp_edges[i];
            }
            this->N_EDGES = tmp_edges.size();

        }else{
            vector<EdgeWithoutWeight> tmp_edges(this->N_EDGES);
            for(int i = 0; i < this->N_EDGES; i++){
                tmp_edges[i] = this->edges_no_lengt[i];
            }

            // Ищем
            for(int i = 0; i < this->N_EDGES; i++){
                EdgeWithoutWeight tmp_el = tmp_edges[i];
                for( auto iter = tmp_edges.begin(); iter != tmp_edges.end() - 1; iter++ ){
                    if(( iter->from_v == tmp_el.to_v) && (iter->to_v == tmp_el.from_v)){
                        tmp_edges.erase(iter);
                        break;
                    }

                }
            }

            int tmp_size = tmp_edges.size();
            for(int i = 0; i < tmp_size; i++){
                tmp_edges.push_back({tmp_edges[i].to_v, tmp_edges[i].from_v});
            }

            delete[] edges_no_lengt;
            edges_no_lengt = new EdgeWithoutWeight[tmp_edges.size()];
            for(int i = 0; i < tmp_edges.size(); i++){
                edges_no_lengt[i] = tmp_edges[i];
            }
            this->N_EDGES = tmp_edges.size();
        }
    }
```

Конструктор для не взвешенного графа работает аналогично конструктору для взвешенного графа. Его реализация показана ниже:

```
DiffGraph(vector<EdgeWithoutWeight> edges, int n, int N, bool directed)
    {
        edges_no_lengt = new EdgeWithoutWeight[n];
        for(int i = 0; i < n; i++){
```

```
            edges_no_lengt[i] = edges[i];
        }

        // Выставляем то, что граф не взвешен
        this->weighted = false;

        // Выставляем тип графа
        this->directed = directed;

        // Выделяем память
        heading = new GraphVertex*[N]();
        this->N = N;

        // Инициализируем указатели на направления для всех вершин
        for (int i = 0; i < N; i++) {
            heading[i] = nullptr;
        }


        // Добавляем ребра направленному графу
        for (unsigned i = 0; i < n; i++)
        {
            int from_v = edges[i].from_v;
            int to_v = edges[i].to_v;

            // Вставка в начало
            GraphVertex* newNode = adjListForVertex(to_v, heading[from_v]);

            // Указатели направления должны указывать на новый узел
            heading[from_v] = newNode;

            this->N_EDGES++;

            // Если граф не направленный, то каждому направлению создаем ещё и обратное
            if(!this->directed){

                newNode = adjListForVertex(from_v, heading[to_v]);

                // Меняем направление, чтобы оно вело на новый узел
                heading[to_v] = newNode;
            }
        }
        if(!this->directed){
            reverseEdges();
        }
        adjLists = adjacencyMat();
    }
```

Помимо этого, также имеется деструктор, который очищает всю динамическую память для предотвращения утечек:

```
~DiffGraph() {
        for (int i = 0; i < N; i++) {
```

```
            delete[] heading[i];
        }
        delete[] edges_lengt;
        delete[] edges_no_lengt;
        delete[] heading;
    }
```

Далее идут методы, которые выводят некую информацию о данном объекте графа:

**Матрица смежности:**

Матрица смежности в данном случае является квадратной матрицей, где каждый столбец и строка – это вершины графа. Она показывает расстояние между ними (или просто есть ли между ними связь – для не взвешенного графа). Метод реализован простым перебором всех узлов и проверкой существования смежных ребер между ними:

```
vector<vector<int>> adjacencyMat(){

        // Создаем двумерный вектор размера NxN (кол-во узлов)
        vector<vector<int>> main_matrix (this->N, vector<int>(this->N, 0));

        // Заполняем матрицу смежности
        if(this->weighted){
            for(int i = 0; i < N; i++){
                GraphVertex* current_node = heading[i];
                while (current_node != nullptr){
                    main_matrix[i][current_node->value] = current_node->len;
                    current_node = current_node->adj;
                }
            }
        }else{
            for(int i = 0; i < N; i++){
                GraphVertex* current_node = heading[i];
                while (current_node != nullptr){
                    main_matrix[i][current_node->value] = 1;
                    current_node = current_node->adj;
                }
            }
        }

        return main_matrix;
    }
```

**Матрица инцидентности:**

Матрица инцидентности показывает какие ребра какое отношение имеют к каждой вершине. Реализация аналогична – простой перебор и проверка принадлежности ребра каждой вершине. Для не ориентированного графа – ребра удваиваются (т.к. идут попарно в разных направлениях):

```
vector<vector<int>> incidenceMat(){
        // Создаем двумерный вектор размера NxM (кол-во узлов x кол-во ребер)
        vector<vector<int>> main_matrix (this->N, vector<int>(this->N_EDGES, 0));

        if(this->weighted){
            for(int i = 0; i < this->N; i++){
                for(int j = 0; j < this->N_EDGES; j++){
                    if(i == edges_lengt[j].from_v){
                        main_matrix[i][j] = 1;
                    }
                    if(i == edges_lengt[j].to_v){
                        main_matrix[i][j] = -1;
                    }
                }
            }
        }else{
            for(int i = 0; i < this->N; i++){
                for(int j = 0; j < this->N_EDGES; j++){
                    if(i == edges_no_lengt[j].from_v){
                        main_matrix[i][j] = 1;
                    }
                    if(i == edges_no_lengt[j].to_v){
                        main_matrix[i][j] = -1;
                    }
                }
            }
        }


        return main_matrix;
    }
```

**Список смежности:**

Список смежности – все смежные вершины для каждой вершины. Его реализация сочетает в себе перебор всех вершин по ребрам и удаление повторяющихся элементов, чтобы получить правильный результат:

```
// Метод выдачи списка смежности
    vector<vector<int>> adjacencyList(){
        vector<vector<int>> main_matrix (this->N, vector<int>(0));

        if(weighted){
            for(int i = 0; i < N; i++){
                for(int j = 0; j < N_EDGES; j++){
                    if(edges_lengt[j].from_v == i){
                        main_matrix[i].push_back(edges_lengt[j].to_v);
                    }else if(edges_lengt[j].to_v == i){
                        main_matrix[i].push_back(edges_lengt[j].from_v);
                    }
                }
                sort(main_matrix[i].begin(), main_matrix[i].end());
                main_matrix[i].erase(unique(main_matrix[i].begin(), main_matrix[i].end()),
 main_matrix[i].end());
            }
        }else{
            for(int i = 0; i < N; i++){
                for(int j = 0; j < N_EDGES; j++){
                    if(edges_no_lengt[j].from_v == i){
                        main_matrix[i].push_back(edges_no_lengt[j].to_v);
                    }else if(edges_no_lengt[j].to_v == i){
                        main_matrix[i].push_back(edges_no_lengt[j].from_v);
                    }
                }
                sort(main_matrix[i].begin(), main_matrix[i].end());
                main_matrix[i].erase(unique(main_matrix[i].begin(), main_matrix[i].end()),
main_matrix[i].end());
            }
        }

        return main_matrix;
    }
```

**Список ребер:**

Список ребер – это, по сути, все ребра, которые есть в графе, выведенные по порядку. Самый простой в реализации метод, т.к. все ребра и так хранятся в графе в виде соответсвтвующих структур. Метод сам определяет какие ребра надо перебирать в зависимости от типа графа – взвешенный/не везвешенный. Листинг:

```
vector<vector<int>> edgesList(){

        int edges_amount = N_EDGES;
        if(!directed){
            edges_amount = N_EDGES / 2;
        }
        vector<vector<int>> main_matrix (edges_amount, vector<int>(2));

        if(weighted){
            for(int i = 0; i < edges_amount; i++){
                main_matrix[i][0] = edges_lengt[i].from_v;
                main_matrix[i][1] = edges_lengt[i].to_v;
            }
        }else{
            for(int i = 0; i < edges_amount; i++){
                main_matrix[i][0] = edges_no_lengt[i].from_v;
                main_matrix[i][1] = edges_no_lengt[i].to_v;
            }
        }

        return main_matrix;

    }
```

**Поиск длиннейшего пути:**

Метод поиска длиннейшего пути работает по принципу поиска в глубину (DFS). Он перебирает длиннейшие пути для каждой точки графа, после чего отсеивает из них путь с самым большим суммарным весом ребер, по которым этот путь построен. В результате, получается самый длинный путь. Реализация выполнена с помощью 3-х методов 1) Метод DFS (поиска в глубину) 2) Метод поиска самого длинного пути из точки 3) Основной метод, перебирающий все точки и находящий наибольший путь из всех полученных. Он возвращает вектор ребер, по которым был построен путь.

```
// Метод поиска в глубина (DFS)
    void DFS(int vertex, int& current_length, vector<EdgeWithWeight>& vertexes_graph){
        visited[vertex] = true;
        vector<int> adjList = adjLists[vertex];

        for(int i = 0; i < N; i++)
            if(!visited[i] && adjList[i] != 0){
                current_length += adjList[i];
                vertexes_graph.push_back({vertex, i, adjList[i]});
                DFS(i, current_length, vertexes_graph);
            }
    }

    // Метод поиска длиннейшего пути из вершины
    vector<EdgeWithWeight> searchLongestRootForOneVertex(int vertex){
        vector<EdgeWithWeight> vertexes_graph;
        int sum_root = 0;
        DFS(vertex, sum_root, vertexes_graph);
        for(int i = 0; i < N; i++){
            visited[i] = false;
        }
        return vertexes_graph;
    }

    // Метод поиска длиннейшего пути в графе
    vector<EdgeWithWeight> findLongestRoot(){
        vector<vector<EdgeWithWeight>> tmp_vect (N, vector<EdgeWithWeight>());

        int max_len = -1;
        int sum = 0;
        int longest_i = 0;
        for(int i = 0; i < N; i++){
            tmp_vect[i] = searchLongestRootForOneVertex(i);
        }

        for(int i = 0; i < N; i++){
            sum = 0;
            for(int j = 0; j < tmp_vect[i].size(); j++){
                sum += tmp_vect[i][j].length;
            }
            if(sum > max_len){
                max_len = sum;
                longest_i = i;
```

```
            }
        }

        return tmp_vect[longest_i];

    }
```

Стоит отметить, что все выше описанные методы не просто выводят информацию в консоль, а возвращают её из функций, таким образом они могут использоваться для получения и дальнейшей обработки информации из графа.

Далее описана реализация генератора графов. Класс графа на этом заканчивается.

Генератор графов представлен в виде отдельного самостоятельного класса, который имеет один основной метод. Использование класса наиболее логично, так как если бы реализация была функциональной, то приходилось бы передавать слишком большое кол-во аргументов.

Начнем с перечисления свойств генератора:

```
private:
        private:
        int max_vertexes = 10;
        int min_vertexes = 5;

        int max_edges = 9;
        int min_edges = 3;

        int max_edges_for_vertex = 3;

        bool directed = true;

        int max_incoming_edges = 5;
        int max_outcoming_edges = 5;

        bool is_lengthed = true;

        int max_length = 10;
        int min_length = 1;

        bool is_legal_islands = false;

        DiffGraph* result_graph = nullptr;
```

Свойства, соответственно, отвечают за следующие параметры:

Максимальное и минимальное кол-во вершин, Максимальное и минимальное кол-во ребер, Максимальное кол-во ребер к каждой вершине, ориентирован ли генерируемый граф, Максимальное кол-во входящих и исходящих вершин, взвешен ли генерируемый граф, максимальный и минимальный вес ребер графа, могут ли быть в графе недоступные области (острова) и техническое свойство – указатель на результирующий граф.

Все данные свойства приватны. Они могут выставляться через методы сеттеры, которые описаны для всех свойств (кроме последнего). Чтобы не было необходимости менять такое большое кол-во свойств каждый раз при создании объекта, у всех у них выставлены значения по умолчанию.

Единственный и основной метод (помимо сеттеров), производит анализ свойств и генерацию, на их основе, графа. Он выглядит следующим образом:

```
DiffGraph* generateRandomGraph(){

            int vertexes_amount = this->min_vertexes + rand() % (this-
>max_vertexes - this->min_vertexes + 1);
            int edges_amount = this->min_edges + rand() % (this->max_edges - this-
>min_edges + 1);

            int* vertex_edges = new int[vertexes_amount];
            int* vertex_outcoming_edges = new int[vertexes_amount];
            int* vertex_incoming_edges = new int[vertexes_amount];
            bool* vertex_empty = new bool[vertexes_amount];

            for(int i = 0; i < vertexes_amount; i++){
                vertex_edges[i] = 0;
                vertex_outcoming_edges[i] = 0;
                vertex_incoming_edges[i] = 0;
                vertex_empty[i] = 1;
            }

            vector<EdgeWithWeight> edges_lengt(edges_amount);
            vector<EdgeWithoutWeight> edges_no_lengt(edges_amount);

            if(this->is_lengthed){
                for(int i = 0; i < edges_amount; i++){
                    edges_lengt[i] = {rand() % (edges_amount + 1), rand() % (edges_amount
+ 1), this->min_length + rand() % (this->max_length - this->min_length + 1)};

                    while((edges_lengt[i].to_v == edges_lengt[i].from_v) || (vertex_edges[
edges_lengt[i].to_v] > this-
>max_edges_for_vertex) || (vertex_edges[edges_lengt[i].from_v] > this-
>max_edges_for_vertex)){
                        edges_lengt[i].to_v = rand() % (edges_amount + 1);
                        if(vertex_edges[edges_lengt[i].from_v] > this-
>max_edges_for_vertex){
                            edges_lengt[i].from_v = rand() % (edges_amount + 1);
                        }
                    }

                    while((vertex_outcoming_edges[edges_lengt[i].from_v] >= this-
>max_outcoming_edges) || (vertex_incoming_edges[edges_lengt[i].to_v] >= this-
>max_incoming_edges)){
                        if(vertex_edges[edges_lengt[i].from_v] >= this-
>max_edges_for_vertex){
                            edges_lengt[i].from_v = rand() % (edges_amount + 1);
                        }else{
                            edges_lengt[i].to_v = rand() % (edges_amount + 1);
```

```
                    }
                }

                vertex_empty[edges_lengt[i].from_v] = 0;
                vertex_empty[edges_lengt[i].to_v] = 0;
                vertex_edges[edges_lengt[i].from_v]++;
                vertex_edges[edges_lengt[i].to_v]++;
                vertex_incoming_edges[edges_lengt[i].to_v]++;
                vertex_outcoming_edges[edges_lengt[i].from_v]++;
            }

            if(!this->is_legal_islands){
                int vertexes_empty_num = 0;
                for(int i = 0; i < vertexes_amount; i++){
                    if(vertex_empty[i]){
                        vertexes_empty_num++;
                        if(i != 0){
                            for(int j = 0; j< edges_amount; j++){
                                if(edges_lengt[j].to_v == i){
                                    edges_lengt[j].to_v--;
                                }else if(edges_lengt[j].from_v == i){
                                    edges_lengt[j].from_v--;
                                }
                            }
                        }else{
                            for(int j = 0; j< edges_amount; j++){
                                if(edges_lengt[j].to_v == i){
                                    edges_lengt[j].to_v++;
                                }else if(edges_lengt[j].from_v == i){
                                    edges_lengt[j].from_v++;
                                }
                            }
                        }

                    }
                }
                vertexes_amount -= vertexes_empty_num;
            }

            delete[] vertex_edges;
            delete[] vertex_outcoming_edges;
            delete[] vertex_incoming_edges;

            this-
>result_graph = new DiffGraph(edges_lengt, edges_amount, vertexes_amount, this->directed);

            return result_graph;

        }else{
            for(int i = 0; i < edges_amount; i++){
                edges_no_lengt[i] = {rand() % (edges_amount + 1), rand() % (edges_amou
nt + 1)};
```

```
                    while((edges_no_lengt[i].to_v == edges_no_lengt[i].from_v) || (vertex_
edges[edges_no_lengt[i].to_v] > this-
>max_edges_for_vertex) || (vertex_edges[edges_no_lengt[i].from_v] > this-
>max_edges_for_vertex)){
                        edges_no_lengt[i].to_v = rand() % (edges_amount + 1);
                        if(vertex_edges[edges_no_lengt[i].from_v] > this-
>max_edges_for_vertex){
                            edges_no_lengt[i].from_v = rand() % (edges_amount + 1);
                        }
                    }

                    while((vertex_outcoming_edges[edges_no_lengt[i].from_v] >= this-
>max_outcoming_edges) || (vertex_incoming_edges[edges_no_lengt[i].to_v] >= this-
>max_incoming_edges)){
                        if(vertex_edges[edges_no_lengt[i].from_v] >= this-
>max_edges_for_vertex){
                            edges_no_lengt[i].from_v = rand() % (edges_amount + 1);
                        }else{
                            edges_no_lengt[i].to_v = rand() % (edges_amount + 1);
                        }
                    }

                    vertex_empty[edges_no_lengt[i].from_v] = 0;
                    vertex_empty[edges_no_lengt[i].to_v] = 0;
                    vertex_edges[edges_no_lengt[i].from_v]++;
                    vertex_edges[edges_no_lengt[i].to_v]++;
                    vertex_incoming_edges[edges_no_lengt[i].to_v]++;
                    vertex_outcoming_edges[edges_no_lengt[i].from_v]++;
                }

                if(!this->is_legal_islands){
                    int vertexes_empty_num = 0;
                    for(int i = 0; i < vertexes_amount; i++){
                        if(vertex_empty[i]){
                            vertexes_empty_num++;
                            if(i != 0){
                                for(int j = 0; j< edges_amount; j++){
                                    if(edges_no_lengt[j].to_v == i){
                                        edges_no_lengt[j].to_v--;
                                    }else if(edges_no_lengt[j].from_v == i){
                                        edges_no_lengt[j].from_v--;
                                    }
                                }
                            }else{
                                for(int j = 0; j< edges_amount; j++){
                                    if(edges_no_lengt[j].to_v == i){
                                        edges_no_lengt[j].to_v++;
                                    }else if(edges_no_lengt[j].from_v == i){
                                        edges_no_lengt[j].from_v++;
                                    }
                                }
                            }

                        }
```

```
                    }
                    vertexes_amount -= vertexes_empty_num;
                }

                delete[] vertex_edges;
                delete[] vertex_outcoming_edges;
                delete[] vertex_incoming_edges;

                this-
>result_graph = new DiffGraph(edges_no_lengt, edges_amount, vertexes_amount, this-
>directed);
                return result_graph;
            }
```

Возвращает указатель на сгенерированный граф, чтобы им можно было пользоваться далее. Сам метод работает следующим образом: сперва, он определяет кол-во вершин и кол-во ребер (случайным образом из диапазона), далее он проверяет какой граф надо генерировать – ориентированный или нет / взвешенный или нет. В зависимости от выбранной ветки, он либо начинает создавать ребра с весами, либо без весов. После того, как ребра и вершины сгенерированы, идет проверка того, сколько ребер связаны с каждой вершиной, сколько из них входят в каждую вершину, сколько выходят, и идет коррекция, то есть, ребра начинают менять вершины из которых / к котором идут, в следствие чего, получается нужный список ребер и вершин. По сути, для метода не играет никакой роли какой граф надо делать – ориентированный или нет – он просто передаст параметр конструктору графа, в котором либо создадутся обратные пары ребер, либо нет. Гораздо большую роль играет взвешен ли граф, так как от этого зависит тип структуры ребер. Из-за этого, метод поделен на две части – одна для взвешенных графов, вторая – для не взвешенных. Такая реализация позволяет уместить всё в один класс, и не создавать разные версии генераторов.

Деструктор класса очищает динамическую память, где был сохранен сам граф.

Помимо данных двух классов, была написана одна функция, которая выводила информацию о ребрах графа, однако она использовалась лишь для удобства в процессе разработки класса графа. К лабораторной работе напрямую она не имеет отношения.

Проверка графа была произведена в главной функции программы, где был создан один граф вручную, а второй с помощью генератора. На них на обоих были протестированы все методы графов (на втором графе не был протестирован последний метод, так как для задания его условий необходимо знание структуры конкретного данного графа, что проблематично, учитывая, что граф генерируется каждый раз случайный).

Далее представлен код тестов:

```
// кол-во узлов
    int N = 6;

    // кол-во ребер
    int n = edges.size();
```

```cpp
    // создаем граф
    DiffGraph graph(edges, n, N, 1);

    cout << "Матрица смежности" << endl;
    vector<vector<int>> test = graph.adjacencyMat();
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            cout << test[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "Матрица инциндентности" << endl;
    vector<vector<int>> test_1 = graph.incidenceMat();
    for(int i = 0; i < N; i++){
        for(int j = 0; j < test_1[i].size(); j++){
            cout << test_1[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "Список смежности" << endl;
    vector<vector<int>> test_2 = graph.adjacencyList();
    for(int i = 0; i < N; i++){
        cout << i << ": ";
        for(int j = 0; j < test_2[i].size(); j++){
            cout << test_2[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "Список ребер" << endl;
    vector<vector<int>> test_3 = graph.edgesList();
    for(int i = 0; i < test_3.size(); i++){
        cout << i << ": ";
        for(int j = 0; j < test_3[i].size(); j++){
            cout << test_3[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "Поиск наидлиннейшего маршрута в графе" << endl;
    vector<EdgeWithWeight> res_longest = graph.findLongestRoot();
    for(int i = 0; i < res_longest.size(); i++){
        cout << res_longest[i].from_v << " -
> " << res_longest[i].to_v << " : " << res_longest[i].length;
        cout << endl;
    }
```

И тест генератора:

```
DiffGraphGenerator new_graph;
    DiffGraph* res = new_graph.generateRandomGraph();

    cout << "Матрица смежности" << endl;
    vector<vector<int>> res_test = res->adjacencyMat();
    for(int i = 0; i < res_test.size(); i++){
        for(int j = 0; j < res_test[i].size(); j++){
            cout << res_test[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "Матрица инциндентности" << endl;
    vector<vector<int>> res_test_1 = res->incidenceMat();
    for(int i = 0; i < res_test_1.size(); i++){
        for(int j = 0; j < res_test_1[i].size(); j++){
            cout << res_test_1[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "Список смежности" << endl;
    vector<vector<int>> res_test_2 = res->adjacencyList();
    for(int i = 0; i < res_test_2.size(); i++){
        cout << i << ": ";
        for(int j = 0; j < res_test_2[i].size(); j++){
            cout << res_test_2[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "Список ребер" << endl;
    vector<vector<int>> res_test_3 = res->edgesList();
    for(int i = 0; i < res_test_3.size(); i++){
        cout << i << ": ";
        for(int j = 0; j < res_test_3[i].size(); j++){
            cout << res_test_3[i][j] << "\t";
        }
        cout << endl;
    }
```

## Заключение.

В ходе данной лабораторной работы было создано две структуры (два класса): структура граф и структура генератор графа. Также, были реализованы различные алгоритмы, например, алгоритм поиска в глубину – DFS, с помощью которого был выполнен поиск самого длинного пути во всем графе (не циклического пути).

Были изучены и на практике реализованы методы для генерации и выдачи матрицы смежности, матрицы инцидентности, списка смежности и списка ребер ориентированных, не ориентированных, взвешенных и не взвешенных графов.