**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 № 4

Выполнил студент группы.....КС-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: …….24.03.2023……..

## 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 входящи и выходящих ребер

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

В качестве проверки работоспособности, требуется сгенерировать 10 графов с возрастающим количеством вершин и ребер(количество выбирать в зависимости от сложности расчета для вашего отдельно взятого ПК). На каждом из сгенерированных графов требуется выполнить поиск кратчайшего пути или подтвердить его отсутствие из точки А в точку Б, выбирающиеся случайным образом заранее, поиском в ширину и поиском в глубину, замерев время требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

## Описание метода/модели.

В математике граф - это абстрактное представление реальной системы объектов, независимо от их природы, которые связаны парными связями.

Каждая вершина графа представляет собой точку, которая связана с другими точками через ребра.

Ребро графа - это линия, соединяющая две точки и представляющая связь между ними.

Граф - это множество вершин, соединенных друг с другом произвольным образом с помощью множества ребер.

Инцидентная вершина - это вершина, которая связана с каким-либо ребром, то есть принадлежит ему. Соседними или смежными вершинами называются те, которые соединены между собой ребром. Количество ребер, входящих и выходящих из графа, определяет его степень. Для ориентированного графа количество входящих ребер называется степенью входа или полустепенью входа, а количество исходящих ребер - степенью выхода или полустепенью выхода.

Изолированная вершина - это вершина, к которой и из которой не идет ни одно ребро. Ориентированный граф - это граф, в котором каждое ребро имеет направление движения, и обратное перемещение обычно не предполагается. Направление ребер указывается стрелками. Графы могут быть конечными (с конечными наборами точек и вершин) или бесконечными.

Графы могут быть простыми или сложными. Простыми считают графы, не содержащие петель и кратных ребер, которые требуют большей тщательности при разработке и реализации алгоритмов на них. Для описания графа используют один из двух удобных для вычисления вариантов:

• Матрица смежности — это двумерная таблица, в которой столбцы и строки соответствуют вершинам, а значения в таблице соответствуют ребрам. Для невзвешенного графа значения могут быть просто 1, если связь есть и идет в нужном направлении, и 0, если ее нет, а для взвешенного графа — конкретные значения.

• Матрица инциндентности — это матрица, в которой строки соответствуют вершинам, а столбцы — связям, и ячейки содержат 1, если связь выходит из вершины, -1, если входит, и 0 во всех остальных случаях.

• Список смежности — это список списков, содержащий все вершины. Внутренние списки для каждой вершины содержат все смежные ей.

• Список ребер — это список строк, в которых хранятся все ребра вершины. Внутренние значения содержат две вершины, к которым присоединено это ребро.

Обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Выделяют два варианта обхода: обход в глубину (или поиск в глубину, DFS) и обход в ширину (или поиск в ширину, BFS).

DFS (Deep first search) следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту. BFS (Breadth first search) следует концепции «расширяйся, поднимаясь на высоту птичьего полета» («go wide, bird’s eye-view»). Вместо того, чтобы двигаться по определенному пути до конца, BFS предполагает движение вперед по одному соседу за разВместо следования по пути, BFS подразумевает посещение ближайших к s соседей за одно действие (шаг), затем посещение соседей соседей и так до тех пор, пока не будет обнаружено t..

## Выполнение задачи.

Код:

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <queue>
#include <chrono>

struct Pair
{
    int src, dest;

    Pair(int src, int dest) :
        src(src),
        dest(dest)
    {}
};

class Graph {
public:
    std::vector<int> vertex;
    std::vector<Pair> pairs;
    bool directed;


    Graph(bool directed) : directed(directed) {}
    //печать матрицы смежности, используя список вершин и пары ребер
    void printMatrixAdj() {
        std::cout << std::setw(5) << " ";
        for (int vert : vertex)
            std::cout << std::setw(5) << "x" + std::to_string(vert);
        std::cout << std::endl;
        for (int vert : vertex) {
            std::cout << std::setw(5) << "x" + std::to_string(vert);
            for (int next_vert : vertex) {
                bool check = false;
                for (Pair pair : pairs)
                    if (pair.src == vert && pair.dest == next_vert) {
                        std::cout << std::setw(5) << 1;
                        check = true;
                        break;
                    }

                if (!check)
                    std::cout << std::setw(5) << 0;
            }
            std::cout << std::endl;
        }
    }
    //печать матрицы инцидентности
    void printMatrixInc() {
        std::cout << std::setw(5) << " ";
        int index = 0;
        for (Pair pair : pairs)
            std::cout << std::setw(5) << "e" + std::to_string(index++);
        std::cout << std::endl;

        for (int vert : vertex) {
            std::cout << std::setw(5) << "x" + std::to_string(vert);
            for (Pair pair : pairs)
                if (!directed)
                    (vert == pair.src || vert == pair.dest) ? std::cout << std::setw(5) << 1 :
std::cout << std::setw(5) << 0;
```

```
                else if (vert == pair.src)
                    std::cout << std::setw(5) << 1;
                else if (vert == pair.dest)
                    std::cout << std::setw(5) << -1;
                else
                    std::cout << std::setw(5) << 0;
            std::cout << std::endl;
        }
    }
    //печатает список смежности для данного графа
    void printListAdj() {
        for (int vert : vertex) {
            std::cout << std::setw(5) << "x" + std::to_string(vert) << std::setw(5) << "->";
            for (Pair pair : pairs)
                if (pair.src == vert)
                    std::cout << std::setw(5) << "x" + std::to_string(pair.dest);
            std::cout << std::endl;
        }
    }
    //выводит список пар вершин графа
    void printPairs() {
        for (Pair pair : pairs)
            std::cout << std::setw(3) << "x" + std::to_string(pair.src) << std::setw(4) << " -
> " << std::setw(3) << "x" + std::to_string(pair.dest) << std::endl;
    }
};

struct Vertex
{
private:
    //максимальное число связей, которые может иметь данная вершина
    int max_in_out;
    int max_out;
public:
    //текущее число связей, которые имеет данная вершина. `num_in_out` обозначает общее число
входящих и исходящих связей, а `num_out` - число исходящих связей
    int num_in_out;
    int num_out;
    //вектор, хранящий индексы вершин, с которыми имеются связи
    std::vector<int> bounds_with_vert;

    //Конструктор для создания новой вершины. В качестве аргументов принимает индекс вершины,
максимальное число входящих и исходящих связей.
    Vertex() {};

    Vertex(int index, int max_in_out, int max_out) {
        bounds_with_vert.push_back(index);
        this->max_in_out = max_in_out;
        this->max_out = max_out;

        num_in_out = 0;
        num_out = 0;
    }
    //Метод для проверки, можно ли добавить еще одну связь к данной вершине.
    //Если это возможно, то метод возвращает случайную вершину графа, с которой еще нет связи.
В противном случае метод возвращает -1
    int checkForAdd(int num_vert) {
        int next_vert = -1;
        if (max_in_out > num_in_out && max_out > num_out && bounds_with_vert.size() !=
num_vert) {
            next_vert = 0 + rand() % (num_vert);
            while (std::find(bounds_with_vert.begin(), bounds_with_vert.end(), next_vert) !=
bounds_with_vert.end())
                next_vert = 0 + rand() % (num_vert);
        }
        return next_vert;
    }
    //Метод для добавления новой связи к данной вершине.
```

```
    //Принимает индекс вершины, с которой требуется добавить связь, и указатель на вершину, с
которой происходит соединение.
    void addNewPair(int index, Vertex* vert_with) {
        num_in_out++;
        num_out++;
        bounds_with_vert.push_back(index);
        vert_with->num_in_out++;
    }
};


/*
Метод генерирует случайный неориентированный/ориентированный граф с заданным числом вершин и
ребер.
Метод также определяет максимальное число выходящих ребер у каждой вершины.
Количество входящих ребер для каждой вершины генерируется случайно в диапазоне от 0 до
max_in_out.
Граф создается на основе списка ребер.
*/
void generationGraph(Graph* graph, int max_vert, int max_pairs, int max_out, bool directed,
int max_in_out, int min_vert = 2, int min_pairs = 0) {
    int num_vert = min_vert + rand() % (max_vert - min_vert + 1);
    for (int i = 0; i < num_vert; i++)
        graph->vertex.push_back(i);

    if (min_pairs = 0)
        min_pairs = num_vert;

    int num_pairs = min_pairs + rand() % (max_pairs - min_pairs + 1);

    if (num_pairs > num_vert * (num_vert - 1) / 2)
        num_pairs = num_vert * (num_vert - 1) / 2;
    std::map<int, Vertex> vertexies;
    int checker_for_pairs = 0;
    for (int vert : graph->vertex)
        vertexies[vert] = Vertex(vert, max_in_out, max_out);


    int current_num_pairs = 0;
    while (current_num_pairs < num_pairs) {
        int vertex = 0 + rand() % (num_vert + 1);
        int ind_next = vertexies[vertex].checkForAdd(num_vert);
        if (ind_next != -1) {
            current_num_pairs++;
            graph->pairs.push_back(Pair(vertex, ind_next));
            vertexies[vertex].addNewPair(ind_next, &vertexies[ind_next]);
            if (!directed) {
                vertexies[ind_next].bounds_with_vert.push_back(vertex);
                vertexies[ind_next].num_out++;
                graph->pairs.push_back(Pair(ind_next, vertex));
            }
        }
    }
}

void deepFirstSearchBegining(Graph* graph, int from, int final);
void breadthFirstSearch(Graph* graph, int from, int final);

int main()
{
    srand(time(NULL));
    Graph* graph = new Graph(false);
    generationGraph(graph, 15, 30, 10, false, 10, 12, 15);
    graph->printMatrixAdj();
    graph->printMatrixInc();
    graph->printListAdj();
    graph->printPairs();
    breadthFirstSearch(graph, 1, 5);
    deepFirstSearchBegining(graph, 1, 5);
```

```
    std::cout << "----------------------------\n";
    for (int i = 0; i < 10; i++) {
        graph = new Graph(true);
        generationGraph(graph, 10 * (i + 1), 10 * ((i + 1) * 10) / 2, 10 * ((i + 1) * 10) / 2,
true, 10 * ((i + 1) * 10) / 2, 10 * (i + 1) - 1, 10 * ((i + 1) * 10) / 4);

        int from = rand() % (graph->vertex.size());
        int to = rand() % (graph->vertex.size());
        while (to == from)
            to = rand() % (graph->vertex.size());
        std::cout << "Deep search\n";
        std::chrono::high_resolution_clock::time_point start =
std::chrono::high_resolution_clock::now();
        deepFirstSearchBegining(graph, from, to);
        std::chrono::high_resolution_clock::time_point end =
std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> sec_diff = end - start;
        std::cout << "Time: " << sec_diff.count() << " sec." << std::endl;
        std::cout << "Breadth search\n";
        start = std::chrono::high_resolution_clock::now();
        breadthFirstSearch(graph, from, to);
        end = std::chrono::high_resolution_clock::now();
        sec_diff = end - start;
        std::cout << "Time: " << sec_diff.count() << " sec." << std::endl;
        std::cout << "----------------------------\n";
    }
}

struct DFSVert
{
    int index;
    bool was_checked;
    std::vector<DFSVert*> pair_vert;

    DFSVert(int index) :index(index), was_checked(false) {}
};

std::string deepFirstSearch(DFSVert* vert, int final, std::string route_before) {
    if (vert->index == final)
        return route_before;

    vert->was_checked = true;
    std::vector<std::string> route;
    bool checker = false;

    for (DFSVert* next_vert : vert->pair_vert) {
        if (!next_vert->was_checked) {
            route.push_back(deepFirstSearch(next_vert, final, route_before + "->" +
std::to_string(next_vert->index)));
            checker = true;
        }
    }

    if (!checker) {
        return "-";
    }

    std::string final_str = "-";

    for (std::string item : route) {
        std::string str = item;
        if (str.back() != '-' && (str.length() < final_str.length() || final_str == "-")) {
            final_str = item;
        }
    }
    return final_str;
}

void deepFirstSearchBegining(Graph* graph, int from, int final) {
```

```
    std::vector<DFSVert> vertexies;
    for (int vert : graph->vertex)
        vertexies.push_back(DFSVert(vert));
    std::sort(graph->pairs.begin(), graph->pairs.end(), [](Pair left, Pair right) {
        if ((left.src < right.src) || (left.src == right.src && left.dest < right.dest))
            return true;
        return false;
    });
    for (Pair pair : graph->pairs)
        vertexies[pair.src].pair_vert.push_back(&vertexies[pair.dest]);

    std::string route = deepFirstSearch(&vertexies[from], final, std::to_string(from));
    if (route.back() == '-')
        std::cout << "Not Found\n";
    else
        std::cout << "Was found: " << route << std::endl;
}

struct BFSVert
{
    std::string road_before;
    int index;
    bool was_open;
    std::vector<BFSVert*> pair_vert;

    BFSVert(int index) :index(index), was_open(false), road_before("") {}
};

void breadthFirstSearch(Graph* graph, int from, int final) {
    std::vector<BFSVert> vertexies;
    for (int vert : graph->vertex)
        vertexies.push_back(BFSVert(vert));
    for (Pair pair : graph->pairs)
        vertexies[pair.src].pair_vert.push_back(&vertexies[pair.dest]);

    std::queue<BFSVert> queue_road;
    queue_road.push(vertexies[from]);

    while (!queue_road.empty()) {
        if (queue_road.front().index != final) {
            if (!queue_road.front().pair_vert.empty())
                for (BFSVert* vert : queue_road.front().pair_vert) {
                    if (!vert->was_open) {
                        vert->road_before += queue_road.front().road_before +
std::to_string(queue_road.front().index) + "->";
                        vert->was_open = true;
                        queue_road.push(*vert);
                    }
                }
            queue_road.pop();
        }
        else {
            std::cout << "Was found: "
                << queue_road.front().road_before << final << std::endl;
            queue_road.pop();
            return;
        }
    }
    std::cout << "Not found\n";
}
```

Изначально был создан и описан класс графа, содержащий вершины и ребра, а также методы вывода матриц смежности и инцидентности, спасков смежности и ребер.

В ходе генерации ребер была также использована структура вершины, содержащая необходимые параметры для проверки на возможность добавления пары.

Для метода поиска в глубину используется рекурсивная функция, а также функция подготовки к рекурсии, создающая вектор структур вершин графа. Данная структура содержит параметры индекса данной вершины, была ли она пройдена и все вершины, в которые можно попасть из неё.

В рекурсивной функции происходит изначальная проверка на конечную вершину и если это она, то возвращается пройденный маршрут, иначе идет обход дальше по смежным с данной вершиной не пройденным точкам. Если ни одной такой точки не найдено и это последняя точка то идет выброс с значением “-“ означающим, что до конечной точки не дошло.

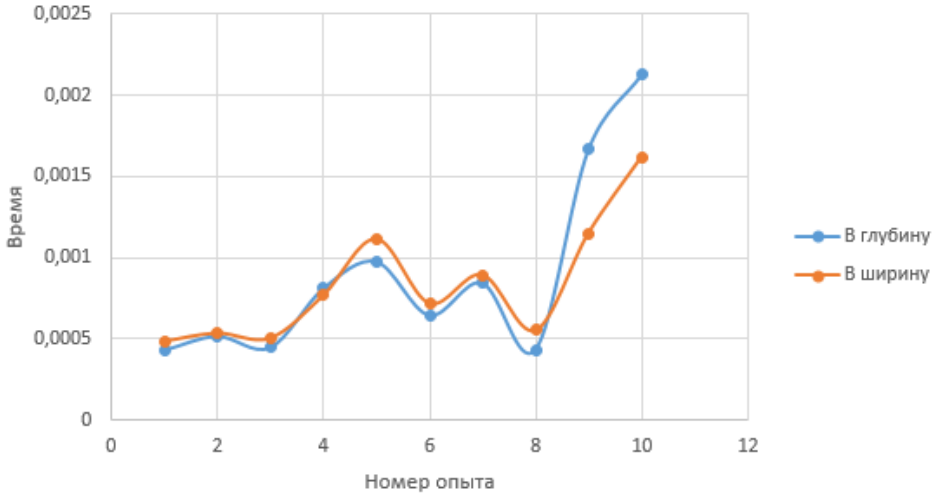
Также из всех пройденных маршрутов выбирается самый минимальный по длине проверкой длины строки.

Для поиска в ширину используется одна функция, в которой происходит обход с помощью очереди. Алгоритм следующий:

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развёрнутый.
    1. Если узел u является целевым узлом, то завершить поиск с результатом «успех».
    2. В противном случае, в конец очереди добавляются все преемники узла u, которые ещё не развёрнуты и не находятся в очереди.
3. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
4. Вернуться к п. 2.

В данном алгоритме также используется структура содержащая в себе в точке до прибытия в неё, индекс вершины, проверку на её раскрытие и вершины в которые можно из неё попасть.

Для проверки результатов был проведен тест на 10 графов, у которых возрастало количество вершин и количество ребер. Результаты вышли следующими:



| | В глубину | | В ширину | |
|---|---|---|---|---|
| Номер | Время | Путь | Время | Путь |
| 1 | 0,0004324 | - | 0,000488 | - |
| 2 | 0,0005182 | 9->0->7->2->4->3->16->14 | 0,0005377 | 9->18->14 |
| 3 | 0,0004491 | - | 0,0005076 | - |
| 4 | 0,0008117 | 24->5->11->7->18->22->0->10->20->1->21 | 0,0007691 | 24->36->6->21 |
| 5 | 0,0009775 | 16->3->8->2->4->14->46 | 0,0011152 | 16->43->39->46 |
| 6 | 0,0006479 | - | 0,0007201 | - |
| 7 | 0,0008463 | - | 0,0008947 | - |
| 8 | 0,0004353 | - | 0,0005544 | - |
| 9 | 0,0016664 | 85->21->11 | 0,0011504 | 85->21->11 |
| 10 | 0,0021272 | 41->1->16->57->56->2->8->10->0->25->27->5->18->19->21->14->9->48->31->6->39->50->70->15->13->23->22->51->76->11->74->28->71->37->84->24->20->72->3->33->82 | 0,0016177 | 41->1->67->33->82 |

## Заключение.

Результаты тестов указывают на то, что если в графе отсутствуют пути до конечной точки, то при использовании поиска в глубину время работы алгоритма будет меньше, чем при использовании поиска в ширину. Однако, если в графе есть маршруты до конечной точки, то при использовании поиска в глубину найденный путь может быть длиннее оптимального пути, поскольку алгоритм движется по точкам последовательно, что может привести к пропуску необходимой точки. Следовательно, поиск в ширину является более стабильным и надежным алгоритмом, но может работать медленнее, если конечная точка находится в отдаленной области.