Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский институт)»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

О т ч ё т

п о д о м а ш н е м у з а д а н и ю № 1

Дисциплина: Анализ Алгоритмов

Тема работы: Алгоритмы обхода графов в глубину

Студентки гр. ИУ7-51б _____ **Сушина А.Д.**

Преподаватель _____ **Волкова Л.Л.**

Москва, 2019г

Оглавление

Введение

Граф, или неориентированный граф G— это упорядоченная пара $G := (V, E)$, где V— непустое множество вершин или узлов, а E— множество пар (в случае неориентированного графа— неупорядоченных) вершин, называемых рёбрами.

Поиск в глубину (англ. Depth-first search, DFS)— один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Поиск в глубину часто используется для проверки связности, поиска цикла и компонент сильной связности и для топологической сортировки.

Цель данной работы: изучить работу рекурсивного и нерекурсивного алгоритмов обхода графов в глубину.

Задачи:

1. Изучить алгоритмы обхода графа в глубину.
2. Реализовать рекурсивный и нерекурсивный алгоритмы обхода графов в глубину.
3. Провести замеры времени и сделать выводы об эффективности каждого из алгоритмов.

1. Аналитическая часть

1.1. Описание алгоритмов

Общая идея

При поиске в глубину посещается первая вершина, затем необходимо идти вдоль ребер графа, до попадания в тупик. Вершина графа является тупиком, если все смежные с ней вершины уже посещены. После попадания в тупик нужно возвращаться назад вдоль пройденного пути, пока не будет обнаружена вершина, у которой есть еще не посещенная вершина, а затем необходимо двигаться в этом новом направлении. Процесс оказывается завершенным при возвращении в начальную вершину, причем все смежные с ней вершины уже должны быть посещены.

Таким образом, основная идея поиска в глубину – когда возможные пути по ребрам, выходящим из вершин, разветвляются, нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

Алгоритм поиска в глубину

Шаг 1. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная.

Шаг 2. Для последней помеченной как посещенная вершины выбирается смежная вершина, являющаяся первой помеченной как не посещенная, и ей присваивается значение посещенная. Если таких вершин нет, то берется предыдущая помеченная вершина.

Шаг 3. Повторить шаг 2 до тех пор, пока все вершины не будут помечены как посещенные.

Рекурсивный алгоритм обхода графа в глубину

Пусть задан граф G=(V, E), где V— множество вершин графа, E— множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в белый цвет. Выполним следующие действия:

1. Пройдём по всем вершинам $Vi \in V$.
    - Если вершина Vi белая, выполним для неё `DFS(Vi)`.

Процедура DFS (параметр Vi— вершина)

1. Перекрашиваем вершину Vi в черный цвет.
2. Для всякой вершины U, смежной с вершиной и окрашенной в белый цвет, рекурсивно выполняем процедуру `DFS(u)`.

Нерекурсивный алгоритм обхода графа в глубину

Процедура DFS (параметр v— вершина)

1. Кладём на стек вершину .
2. Пока стек не пуст…

1. Берём верхнюю вершину .
2. Если она белая…
    1. Перекрашиваем её в чёрный цвет.
    2. Кладём в стек все смежные с белые вершины.

2. Конструкторская часть

2.1. Разработка алгоритмов

На рисунках 1-2 представлены схемы рекурсивного и нерекурсивного алгоритмов обхода графа в глубину.

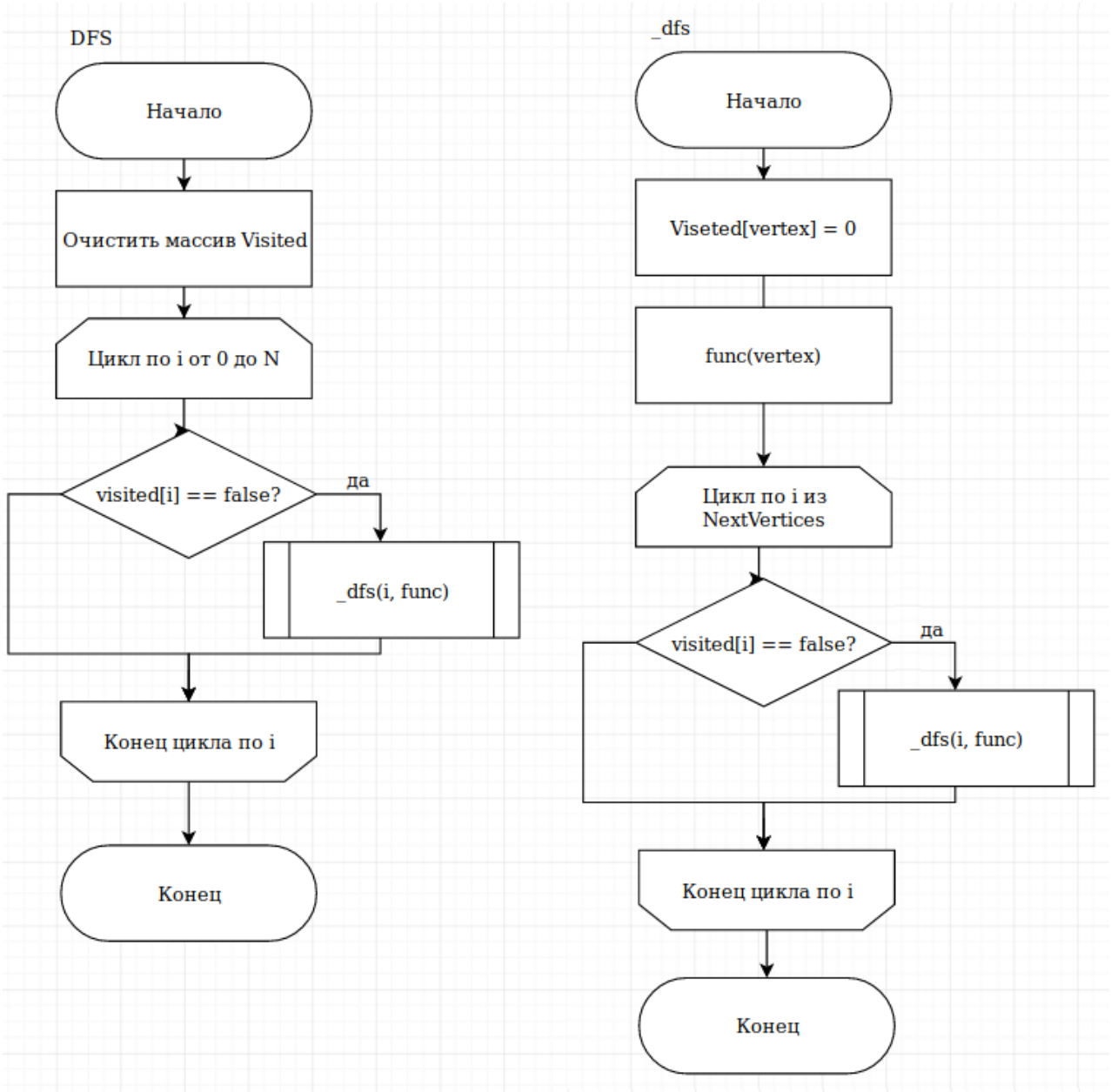

Рис 1. Схема рекурсивного алгоритма обхода графа в глубину

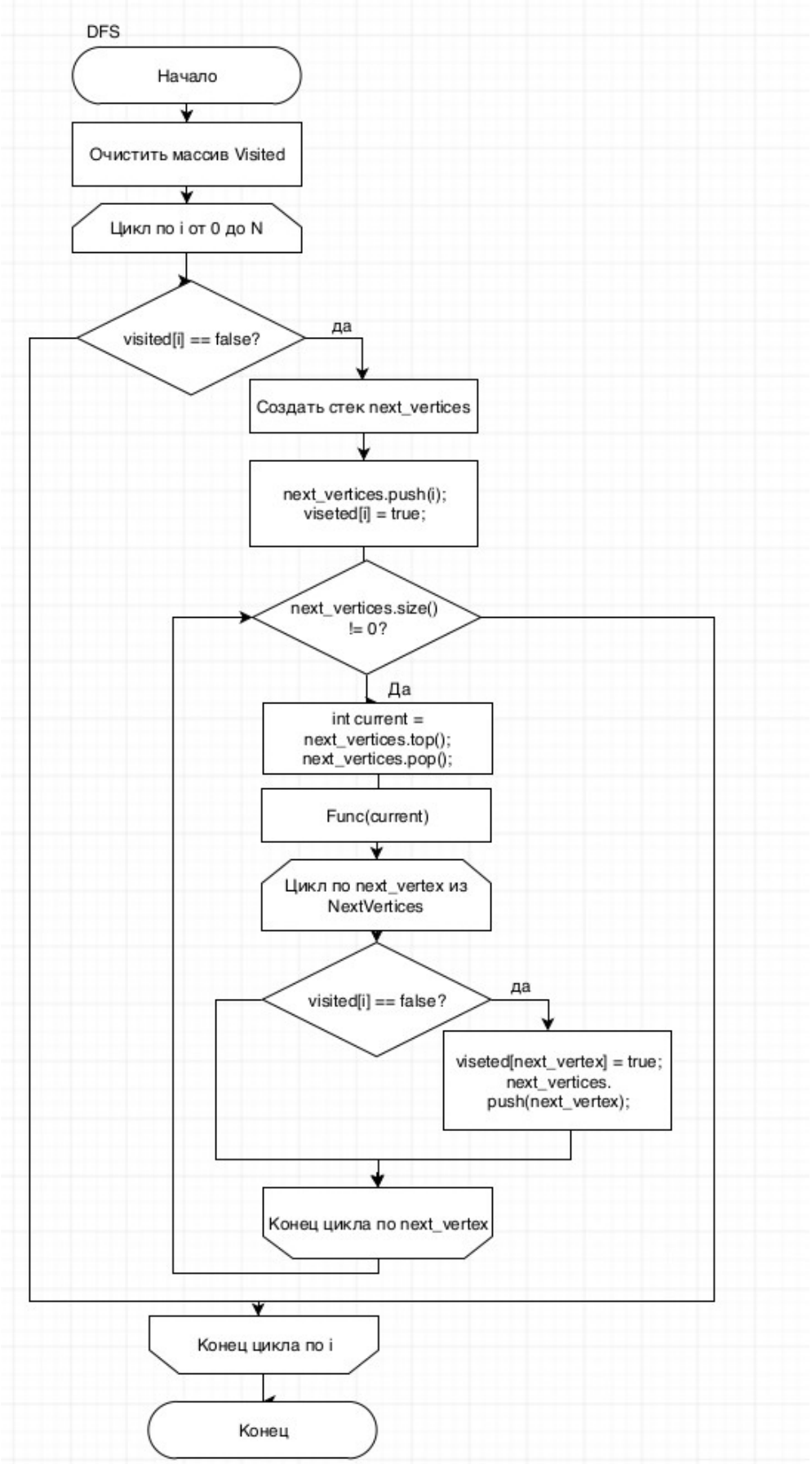

Рис 2. Схема нерекурсивного алгоритма обхода графа в глубину

3. Технологическая часть

3.1. Требования к программному обеспечению

На вход программе поступает граф и вершина, с которой требуется начать обход в глубину. На выходе должены быть напечатаны вершины графа в порядке обхода в глубину.

3.2. Средства реализации

Для реализации программы была выбрана среда разработки Qt Creator. Язык разработки — C++. Для реализации графа были написаны интерфейс Ighaph и класс Cmatrixgraph, реализующий структуру данных граф на матрице смежности. Для замеров времени была использована библиотека ctime. Также были использованы стандартные библиотеки vector и stack.

3.3. Листинг кода

Код программы находится на листингах 1-3.

Листинг 1. Реализация неориентированного графа

```
struct IGraph {
    virtual ~IGraph() = 0;

    // Добавление ребра от from к to.
    virtual void AddEdge(int from, int to) = 0;

    virtual int VerticesCount() const  = 0;

    virtual std::vector<int> GetNextVertices(int vertex) const = 0;
    virtual std::vector<int> GetPrevVertices(int vertex) const = 0;
};

IGraph::~IGraph() {}

class CMatrixGraph: public Igraph {
public:
    CMatrixGraph(int n) :
        viseted(n, false),
        matrix(n)
    {
        for (size_t i = 0; i < matrix.size(); i++) {
            matrix[i].resize(n);
            for (int j = 0; j < matrix.size(); j++) {
                matrix[i][j] = false;
            }
        }
    }
    CMatrixGraph(const IGraph &graph) {
        matrix.resize(graph.VerticesCount());
        for (size_t i = 0; i < matrix.size(); i++) {
            matrix[i].resize(graph.VerticesCount());
            std::vector<int> next = graph.GetNextVertices(i);
            for (size_t j = 0; j < next.size(); j++) {
                matrix[i][next[j]] = true;
            }
        }
```

```
    }
    ~CMatrixGraph() {}

    void AddEdge(int from, int to) override {
        matrix[from][to] = true;
    }

    int VerticesCount() const  override {
        return (int)matrix.size();
    }

    std::vector<int> GetNextVertices(int vertex) const override {
        std::vector<int> next;
        for (size_t i = 0; i < matrix.size(); i++) {
            if (matrix[vertex][i])
                next.push_back(i);
        }
        return next;

    }
    std::vector<int> GetPrevVertices(int vertex) const override {
        std::vector<int> prev;
        for (size_t i = 0; i < matrix.size(); i++) {
            if (matrix[i][vertex])
                prev.push_back(i);
        }
        return prev;
    }

    void DFS(int vertex,  void (*func)(int));

private:
    std::vector<bool> viseted;
    std::vector<std::vector<bool>> matrix;

};
```

Листинг 2. Рекурсивный алгоритм обхода графа в глубину

```
void DFS(int vertex,  void (*func)(int)) {
        for (int i = 0; i < viseted.size(); i++)
        {
            viseted[i] = false;
        }
        for (int i = 0; i < this->VerticesCount(); i++){
            if (!viseted[i]) {
                this->_dfs(i, func);
            }
        }

    }


void _dfs(int vertex,  void (*func)(int)) {

        if (viseted[vertex]) return;
        viseted[vertex] = true;
```

```
        func(vertex);
        for (int next_vertex: this->GetNextVertices(vertex))
        {
            if (!viseted[next_vertex])
            {
                _dfs(next_vertex, func);
            }
        }
    }
```

Листинг 3. Нерекурсивный алгоритм обхода графа в глубину

```
    void DFS2(int vertex, void (*func)(int))
    {
        for (int i = 0; i < viseted.size(); i++)
        {
            viseted[i] = false;
        }
        for (int i = 0; i < this->VerticesCount(); i++){
            if (!viseted[i]) {
                std::stack<int> next_vertices;

                next_vertices.push(i);
                viseted[i] = true;

                while (next_vertices.size())
                {
                    int current = next_vertices.top();
                    next_vertices.pop();

                    func(current);

                    for (int next_vertex: this->GetNextVertices(current))
                    {
                        if (!viseted[next_vertex])
                        {
                            viseted[next_vertex] = true;
                            next_vertices.push(next_vertex);
                        }
                    }
                }
            }
        }
    }
```

4. Экспериментальная часть

4.1. Постановка эксперимента по замеру времени

Были проведены временные эксперименты для графов размером от 100 до 3000 вершин с шагом 100. Для каждого графа были созданы 90% всех ребер. Для каждого замера взят средний результат из 5 замеров. Результаты экспериментов представлены на рисунке 3.

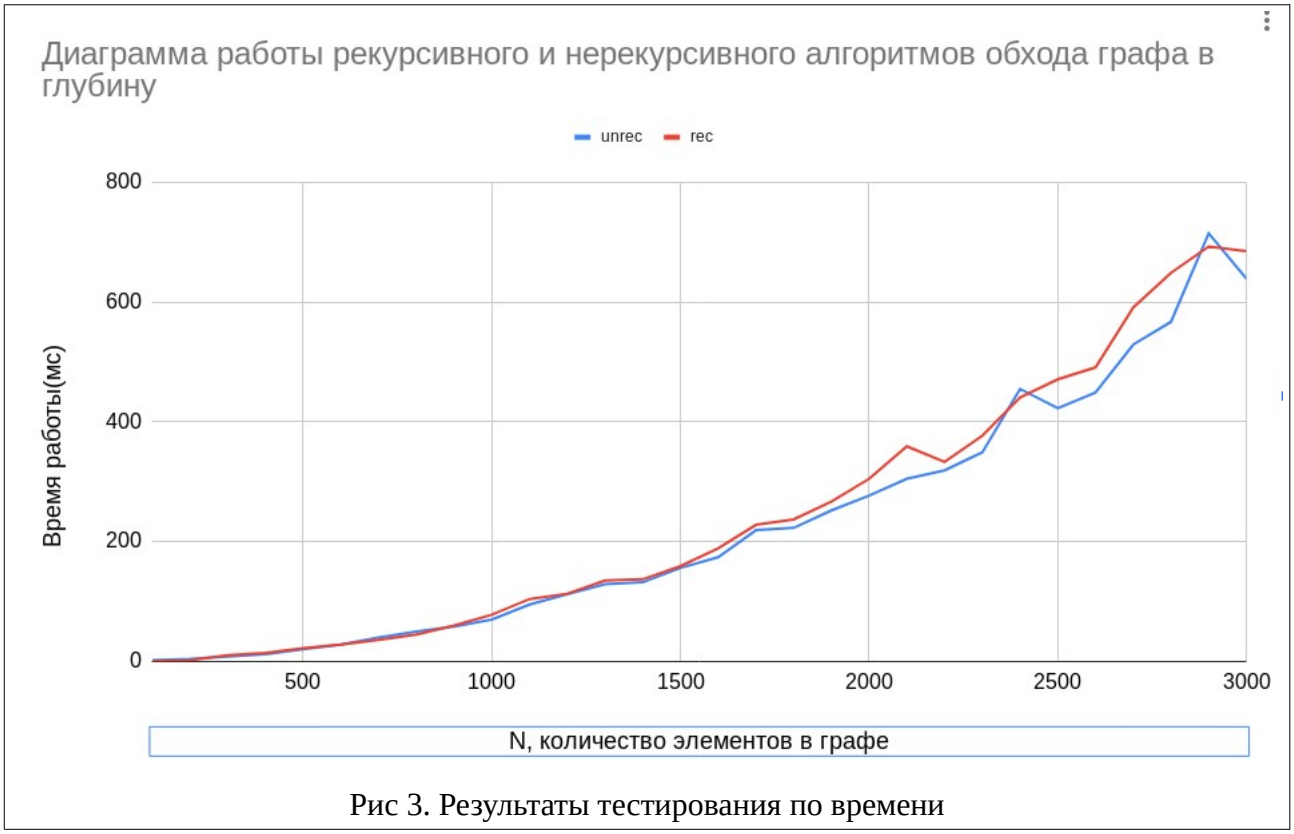

Рис 3. Результаты тестирования по времени

4.2. Выводы

По результатам тестирования можно сделать вывод, что алгоритмы работают примерно с одной скоростью. Это объясняется тем, что нерекурсивный алгоритм также использует стек, однако использует его явно, в то время как рекурсивный алгоритм использует стек вызовов. Однако рекурсивный алгоритм может сильно нагружать стек вызовов при работе с большими графами, поэтому в этих случаях целесообразнее применять алгоритм нерекурсивный алгоритм.

Заключение

В ходе работы были изучены и реализованы рекурсивный и нерекурсивный алгоритмы обхода графа в глубину, были проведены эксперименты по замеру времени. Исходя из результатов экспериментов, можно сделать вывод, что среди алгоритмов нет явного лидера по времени выполнения, однако в некоторых случая целесообразнее использовать нерекурсивный алгоритм, чтобы не нагружать стек вызовов.