МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ИС

КУРСОВАЯ РАБОТА «ГРАФЫ»

по дисциплине «Алгоритмы и структуры данных»

Тема: Реализация алгоритма поиска минимального остова на основе алгоритма Краскала

Студентка гр. 3373 Бельская В.И.

Преподаватель Молдовян Д.Н.

Санкт-Петербург
2024

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студентка Бельская В.И.

Группа 3373

Тема работы: **Реализация алгоритма поиска минимального остова на основе алгоритма Краскала**

Исходные данные:

Разработать приложение для поиска минимального остова графа.

Входные данные:

На вход подается матрица смежности в виде текстового файла, например:

A B C

0 3 1

3 0 2

1 2 0

Выходные данные: результат в виде отсортированных по имени пар и суммарный вес. Максимальный размер входных данных: 50 вершин.

Содержание пояснительной записки:

Введение, способ представления данных в памяти, инициализация графа, сортировка ребер графа по возрастанию весов, описание алгоритма Краскала, сортировка ребер в лексикографическом порядке, оценка временной сложности, результат выполнения задачи, заключение, список использованных источников, приложение А−исходный код.

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 02.12.2024

Дата сдачи реферата: 13.12.2024

Дата защиты реферата: 13.12.2024

Студентка		Бельская В.И.
Преподаватель		Молдовян Д.Н.

АННОТАЦИЯ

Алгоритма Краскала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

Задача: разработать консольное приложение для нахождения минимального остова графа. В проекте описывается теоретическая составляющая: реализация консольного приложения для построения системы непересекающихся множеств (СНМ), а также описание выбранных алгоритмов сортировки, обхода графов и СНМ и их предназначение в данном приложении.

В отчете присутствует исходный код программы.

SUMMARY

Kraskal's algorithm is an effective algorithm for constructing a minimal spanning tree of a weighted connected undirected graph.

Task: to develop a console application for finding the minimum backbone of a graph. The project describes the theoretical component: the implementation of a console application for building a system of disjoint sets (SNM), as well as a description of the selected sorting algorithms, graph traversal and SNM and their purpose in this application.

The report contains the source code of the program.

Введение

Цель работы: Реализовать алгоритм поиска минимального остова на основе алгоритма Краскала (Крускала). Продемонстрировать знания следующих вопросов:

- сортировка,
- обход графов,
- хранение графов,
- построение системы непересекающихся множеств.

Входные данные:

Любой текстовый файл со смежной матрицей графа:

A B C

0 3 1

3 0 2

1 2 0

Результат в виде отсортированных по имени пар и суммарный вес:

A C

B C

3

Максимальный размер входных данных: 50 вершин. Вес ребра ограничен интервалом от 0 до 1023 включительно.

АРХИТЕКТУРА ПРОЕКТА

1.1. STRUCT EDGE (СТРУКТУРА РЕБРО)

Структура ребро содержит в себе:

- 2 указателя на объекты структуры Node (структура вершина): *src* и *dest*
- целочисленное поле, отвечающее за вес ребра: *weight*
- конструктор

struct Edge	
src	*Node
dest	*Node
weight	int

1.2. STRUCT NODE (СТРУКТУРА ВЕРШИНА)

Структура вершина содержит в себе:

- строковое имя вершины: *Name*
- Вектор указателей на объекты структуры Edge, ребер, инцидентных с вершиной: *edges*
- конструктор

struct Node	
Name	string
edges	vector<class Edge*>
weight	int

1.3. STRUCT GRAPH (СТРУКТУРА ГРАФ)

Структура граф содержит в себе:

- Вектор указателей на объекты структуры *Edge*
- Вектор указателей на объекты структуры *Node*

struct Node	
edges	vector<Edge*>
nodes	vector<Node*>

1.4. STRUCT UNIONFIND (СТРУКТУРА СИСТЕМЫ НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ)

Структура системы непересекающихся множеств содержит в себе:

- Вектор parent, хранящий корни вершин
- Вектор rank, хранящий высоту вершин

struct UnionFind	
parent	vector<int>
rank	vector<int>

2. ОСНОВНОЙ ФУНКЦИОНАЛ

2.1. ИНИЦИАЛИЗАЦИЯ ГРАФА

Граф инициализируется путем чтения матрицы смежности графа из файла, название которого вводится пользователем. С помощью функции readAdjacencyMatrix() производится парсинг записи и формирование графа. Функция считывает первую строку, содержащую имена вершин, и добавляет их в вектор имен вершин графа. После переходит к матрице смежности. Если очередной элемент матрицы не равен нулю, то с помощью функции addEdge() создается ребро и помещается в вектор рёбер графа.

2.2. ПОИСК МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА

В начале рёбра сортируются по весу по возрастанию. Использована встроенная в библиотеку <algorithm> функция sort, так как она оптимизирована для большинства входных данных. Данная функция автоматически подстраивается под входные данные и выбирает оптимальный из вариантов: быстрая сортировка, сортировка кучей или сортировка вставками. В нашем случае скорее всего используется сортировка вставками, так как она наиболее оптимальна для небольшого объёма данных. Инициализируется структура данных **UnionFind** для работы с непересекающимися множествами, а также создаётся новый граф для минимального остовного дерева (MST). Далее рёбра

добавляются в MST поочередно: для каждого ребра проверяется, образует ли оно цикл в уже построенном остовном дереве, с помощью структуры Union-Find. Если ребро не образует цикл, оно добавляется в дерево. В процессе добавления рёбер система непересекающихся множеств помогает гарантировать, что в графе не будет циклов. Таким образом, алгоритм строит минимальное остовное дерево, избегая зацикливания.

2.3. ОБХОД ГРАФА В ШИРИНУ

Используется очередь (queue<Node*>) для хранения вершин, которые нужно обработать. Множество visited гарантирует, что каждая вершина будет посещена только один раз. Сначала мы помещаем стартовую вершину в очередь и помечаем её как посещённую. Затем, пока очередь не пуста, мы:

1. Извлекаем вершину из очереди.
2. Выводим её имя.
3. Перебираем все рёбра этой вершины, проверяем соседей, которые ещё не посещены, и добавляем их в очередь.

2.4. ОБХОД ГРАФА В ГЛУБИНУ

Обход графа в глубину в этом коде реализован с использованием структуры данных **стек (stack)** для итеративного подхода (или рекурсивно посещают, если используется рекурсивный алгоритм). Начальный узел добавляется в стек, и пока стек не пуст, извлекается последний элемент. Если вершина ещё не была посещена, она добавляется в множество **visited**, и её имя выводится. Затем перебираются смежные вершины, и если они ещё не посещены, они добавляются в стек для последующей

обработки. Обход заканчивается, когда стек (или рекурсивный вызов) пуст и все вершины графа были посещены. Это позволяет обрабатывать вершины глубже, начиная с начальной, в порядке обхода графа.

2.4.1. Рекурсивный DFS

Использует системный стек (стек вызовов функций) для хранения текущего состояния обхода. Код выглядит более компактным и проще в понимании. Однако есть ограничение по глубине рекурсии, которое может привести к переполнению стека, если граф слишком глубокий.

2.4.2. Итеративный DFS

Использует явный стек данных вместо системного стека. Код становится немного сложнее из-за ручного управления стеком. Нет ограничения по глубине рекурсии, так как стек контролируется пользователем.

2.5. СИСТЕМА НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ

Система непересекающихся множеств необходима, чтобы избежать зацикливания, при построении Минимального Остовного Дерева (MST).

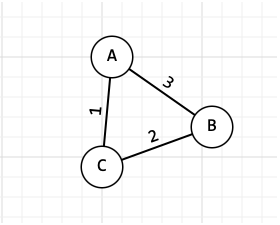
Конструктор UnionFind() инициализирует объект структуры, заполняя вектор parent по правилу: каждой вершине соответствует она же.
Метод (UnionFind) findParent() необходим для сопоставления всем вершинам множества корень этого множества, а также для сжатия пути (Path Compression) от одной вершины до корня.
Метод (UnionFind) unionSets() объединяет множества, операция объединения зависит от ранка множества, ранк - это мера высоты дерева

(в данном контексте множества). Если ранки обоих множеств равны, то 2 множество присоединяется к 1, а ранк увеличивается, в противном случае множестве с меньшим ранком присоединяется ко множеству с большим ранком. Это позволяет присоединять деревья с меньшей высотой к дереву с большей высотой, тем самым оптимизируя алгоритм присоединения.

При добавлении нового ребра в MST, проверяется равны ли корень множества одной вершины корню другой вершины, если да, то вершины уже находятся в одном множестве, соответственно их объединение приведет к зацикливанию, в противном случае, множества объединяются, а ребро добавляется в MST.

3. РЕЗУЛЬТАТ РЕШЕНИЯ ЗАДАЧИ

Исходный граф:



3.1. ВИДЫ ПРЕДСТАВЛЕНИЯ ГРАФА

3.1.1. МАТРИЦА СМЕЖНОСТИ

```
   A B C
A 0 3 1
B 3 0 2
C 1 2 0
```

3.1.2. СПИСКИ СМЕЖНОСТИ

```
A: B C
B: A C
C: A B
```

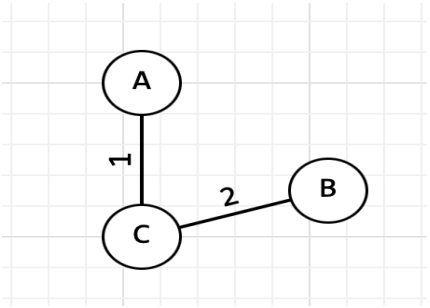
3.1.3. МАТРИЦА ИНЦИДЕНТНОСТИ

```
  e1    e2    e3
A 1     1     0
B 1     0     1
C 0     1     1
e1 = BA
e2 = CA
e3 = CB
```

3.2. ПОИСК МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА. АЛГОРИТМ КРАСКАЛА

```
Минимальное остовое дерево:
A --(1)--> C
B --(2)--> C


Вес MCT:3
```



В приложении MST представлен в виде списка рёбер.

3.3. ОБХОД ГРАФА

3.3.1. ОБХОД ГРАФА В ГЛУБИНУ(DFS)

```
A B C
```

3.3.2. ОБХОД ГРАФА В ШИРИНУ(BFS)

```
A B C
```

3.4. СОРТИРОВКА ГРАФА

```
A --(1)--> C
B --(2)--> C


Вес MCT:3
```

ЗАКЛЮЧЕНИЕ

При выполнении курсовой работы были получены практические навыки работы с графами, были продемонстрированы знание следующих

вопросов: сортировка, обход графов, хранение графов и построение системы непересекающихся множеств. Также был закодирован оптимальный алгоритм, для поиска минимального остовного дерева.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Т. Корман “Introduction to Algorithms”
2. Кормен Томас Х. И. Алгоритмы. Построение и анализ
3. Райнхард Дистель “Graph Theory”
4. GeeksForGeeks URL: https://www.geeksforgeeks.org/
   (Дата обращения: 06.12.2024)
5. Markoutte.me URL: https://markoutte.me/ (Дата обращения: 01.12.2024)

ПРИЛОЖЕНИЕ А “GRAPHS”

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <stack>
#include <queue>
#include <sstream>
#include <map>
#include <algorithm>
#include <set>

using namespace std;
struct Node { //вершины
    string name;
    vector<class Edge*> edges; //список инцидентных вершине рёбер
    Node(const string& nodeName) : name(nodeName) {}
};
struct Edge { //рёбра
    Node* src; //указатели на вершины-источники
    Node* dest; //вершины-приемники
    int weight; //вес ребра
    Edge(Node* s, Node* d, int w) : src(s), dest(d), weight(w) {}
};
struct Graph {
    vector<Node*> nodes;
    vector<Edge*> edges;
    Node* getNodeByName(const string& name) { //возвращает вершину
по имени
        for (Node* node : nodes) {
```

```
            if (node->name == name) {
                return node;
            }
        }
        return nullptr;
    }
    void addEdge(Node* src, Node* dest, int weight) { //добавляет
ребро в граф и связывает его с вершинами
        Edge* edge = new Edge(src, dest, weight);
        edges.push_back(edge);
        src->edges.push_back(edge);
        dest->edges.push_back(edge);
    }
};
bool readAdjacencyMatrix(const string& filename, Graph& graph) {
    //читает матрицу смежности из файла и строит граф
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Ошибка открытия файла!" << endl;
        return false;
    }

    string line;
    vector<string> nodeNames;
    vector<vector<int>> matrix;
    // Чтение первой строки
    if (getline(file, line)) {
        stringstream ss(line);
        string name;
        while (ss >> name) {
            nodeNames.push_back(name);
        }
    }
    // Чтение матрицы смежности
    while (getline(file, line)) {
```

```
        stringstream ss(line);
        vector<int> row;
        int weight;
        while (ss >> weight) {
            row.push_back(weight);
        }
        matrix.push_back(row);
    }
    for (const string& name : nodeNames) {
        Node* newNode = new Node(name);
        graph.nodes.push_back(newNode);
    }
    for (int i = 0; i < nodeNames.size(); ++i) {
        for (int j = i; j < nodeNames.size(); ++j) {
            if (matrix[i][j] != 0) {
                Node* src = graph.nodes[i];
                Node* dest = graph.nodes[j];
                graph.addEdge(src, dest, matrix[i][j]);
            }
        }
    }
    return true;
}
//сравнение рёбер графа по весу
bool compareEdges(Edge edge1, Edge edge2) {
    return edge1.weight < edge2.weight;
}
struct UnionFind {
    vector<int> parent;
    vector<int> rank;
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
```

```
            parent[i] = i; //изначально каждый элемент является
своим родителем
        }
    }

    //определяет корень множества, к которому принадлежит элемент
    int findParent(int u) {
        if (parent[u] != u) {
            parent[u] = findParent(parent[u]); //сжатие пути
            /* после вызова findParent,
               parent[u] обновляется так,
               чтобы сразу указывать на корень множества
               Это ускоряет последующие запросы */
        }
        return parent[u];
    }
    // реализует объединение двух множеств
    bool unionSets(int u, int v) {
        int rootU = findParent(u);
        int rootV = findParent(v);
        if (rootU != rootV) {
            /*надо, чтобы дерево
              с меньшим рангом стало
              дочерним для дерева
              с большим рангом*/
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            }
            else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            }
            else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
```

```
            return true; //деревья были разными
        }
        return false;//корни совпали(уже одно множество)
    }
};
//Алгоритм Крускала
Graph kruskal(Graph& graph) {//для нахождения минимального
остовного дерева
    sort(graph.edges.begin(), graph.edges.end(),
        //graph.edges — это контейнер со всеми рёбрами графа.
        [](Edge* e1, Edge* e2) {//анонимная функция для сравнения
рёбер по весу
            return e1->weight < e2->weight;
        }
    );
 // помогает объединять компоненты связности и определять,
принадлежат ли две вершины одной компоненте
    UnionFind uf(graph.nodes.size());
    Graph mst;
    //Граф сортируется по весам ребер.
    //Ребра добавляются в МОС, если они не образуют цикл.
    set<Node*> addedNodes;
    for (Edge* edge : graph.edges) {

        int srcIndex = -1, destIndex = -1;
        for (int i = 0; i < graph.nodes.size(); ++i) {
            if (graph.nodes[i] == edge->src) {
                srcIndex = i;
            }
            if (graph.nodes[i] == edge->dest) {
                destIndex = i;
            }
        }
        if (uf.unionSets(srcIndex, destIndex)) {
            mst.edges.push_back(edge);
```

```
            if (addedNodes.count(edge->src) == 0) {
                mst.nodes.push_back(edge->src);
                addedNodes.insert(edge->src);
            }
            if (addedNodes.count(edge->dest) == 0) {
                mst.nodes.push_back(edge->dest);
                addedNodes.insert(edge->dest);
            }
        }
    }
    return mst;
}



void BFS(Graph& graph, Node* startNode) { //обход в ширину
    set<Node*> visited;//хранит уже посещённые узлы, чтобы
избежать повторного посещения
    queue<Node*> q; //очередь для узлов, которые нужно обработать
    q.push(startNode);//начальный узел добавляется в очередь для
обработки
    visited.insert(startNode);//начальный узел помечается как
посещённый
    while (!q.empty()) {
        Node* current = q.front();
        q.pop();// удаляется из очереди
        cout << current->name << " ";
        for (Edge* edge : current->edges) {
            Node* neighbor = (edge->src == current) ? edge->dest :
edge->src;
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                q.push(neighbor);
            }
        }
```

```
    }
}
//Итеративный вариант использует стек для управления обходом,
вместо рекурсивных вызовов
void DFSIterative(Graph& graph, Node* startNode) { //итеративный
обход в глубину
    set<Node*> visited;
    stack<Node*> s; //стек для управления обходом графа
    s.push(startNode);
    while (!s.empty()) {
        Node* current = s.top();
        s.pop();
        if (visited.find(current) == visited.end()) {
            visited.insert(current);
            cout << current->name << " ";
            for (auto it = current->edges.rbegin(); it !=
current->edges.rend(); ++it) {
                Node* neighbor = (*it)->src == current ?
(*it)->dest : (*it)->src;
                if (visited.find(neighbor) == visited.end()) {
                    s.push(neighbor);
                }
            }
        }
    }
}
void DFSRecursive(Node* node, set<Node*>& visited) { //рекурсивный
обход в глубину
    if (visited.find(node) != visited.end()) return;
    visited.insert(node);
    cout << node->name << " ";
    for (Edge* edge : node->edges) {
        Node* neighbor = (edge->src == node) ? edge->dest :
edge->src;
        if (visited.find(neighbor) == visited.end()) {
```

```
            DFSRecursive(neighbor, visited);
        }
    }
}
void DFS(Graph& graph, Node* startNode) {
    set<Node*> visited;
    DFSRecursive(startNode, visited);
}
vector<vector<string>> toAdjacencyList(Graph& graph) { //список
смежности
    vector<vector<string>> adjList(graph.nodes.size());
    map<Node*, int> nodeIndex;
    for (int i = 0; i < graph.nodes.size(); ++i) {
        nodeIndex[graph.nodes[i]] = i;
    }
    for (Edge* edge : graph.edges) {
        int srcIndex = nodeIndex[edge->src];
        int destIndex = nodeIndex[edge->dest];
        adjList[srcIndex].push_back(edge->dest->name);
        adjList[destIndex].push_back(edge->src->name);
    }
    return adjList;
}
vector<vector<int>> toIncidenceMatrix(Graph& graph) { //матрица
инцидентности
    int m = graph.edges.size();
    int n = graph.nodes.size();
    vector<vector<int>> incMatrix(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
        Edge* edge = graph.edges[i];
        int srcIndex = -1, destIndex = -1;
        for (int j = 0; j < n; ++j) {
            if (graph.nodes[j] == edge->src) {
                srcIndex = j;
            }
```

```
            if (graph.nodes[j] == edge->dest) {
                destIndex = j;
            }
        }
        incMatrix[i][srcIndex] = 1;
        incMatrix[i][destIndex] = 1;
    }
    return incMatrix;
}

vector<vector<int>> toAdjacencyMatrix(Graph& graph) { //матрица
смежности
    int n = graph.nodes.size();
    vector<vector<int>> adjMatrix(n, vector<int>(n, 0));
    map<Node*, int> nodeIndex;
    for (int i = 0; i < graph.nodes.size(); ++i) {
        nodeIndex[graph.nodes[i]] = i;
    }
    for (Edge* edge : graph.edges) {
        int srcIndex = nodeIndex[edge->src];
        int destIndex = nodeIndex[edge->dest];
        adjMatrix[srcIndex][destIndex] = edge->weight;
        adjMatrix[destIndex][srcIndex] = edge->weight;  \
    }
    return adjMatrix;
}
void printAdjacencyMatrix(Graph& graph) { //матрица смежности
    vector<vector<int>> adjMatrix = toAdjacencyMatrix(graph);
    cout << "   ";
    for (int i = 0; i < graph.nodes.size(); ++i) {
        cout << graph.nodes[i]->name << " ";
    }
    cout << endl;
    for (int i = 0; i < adjMatrix.size(); ++i) {
        cout << graph.nodes[i]->name << " ";
```

```
        for (int j = 0; j < adjMatrix[i].size(); ++j) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}
void printIncidenceMatrix(Graph& graph) { //матрица инцидентности
    vector<vector<int>> incMatrix = toIncidenceMatrix(graph);
    cout << "  ";
    for (int i = 0; i < graph.edges.size(); ++i) {
        cout << "e" << i + 1 << "     ";
    }
    cout << endl;
    for (int i = 0; i < graph.nodes.size(); ++i) {
        cout << graph.nodes[i]->name << " ";
        for (int j = 0; j < incMatrix.size(); ++j) {
            cout << incMatrix[j][i] << "      ";
        }
        cout << endl;
    }
    for (int i = 0; i < graph.edges.size(); ++i) {
        cout << "e" << i + 1 << " = " <<
graph.edges[i]->dest->name << graph.edges[i]->src->name << endl;
    }
    cout << endl;
}
void printAdjacencyList(Graph& graph) { //список смежности
    auto adjList = toAdjacencyList(graph);
    for (int i = 0; i < adjList.size(); ++i) {
        cout << graph.nodes[i]->name << ": ";
        for (const string& neighbor : adjList[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
```

```
}
int main() {
    Graph graph;
    setlocale(LC_ALL, "Russian");
    int menuChoose = -1;
    while (menuChoose) {
        cout <<
"\n\n----------------------------------------------------------------
------------------------------";
        cout << "\n\nВыберите пункт меню: \n"
            << "1. Создать граф по матрице смежности (чтение из
файла).\n"
            << "2. Поиск минимального остового дерева(алгоритм
Крускала).\n"
            << "3. Обход графа\n"
            << "4. Вывести граф\n"
            << "0. Выйти из программы.\n"
            << "\nВаш выбор: ";
        cin >> menuChoose;
        cout <<
"\n\n----------------------------------------------------------------
------------------------------\n\n";
        switch (menuChoose) {
        case 0: {
            cout << "\n\n Хорошего вечера :)";
            menuChoose = 0;
        }break;
        case 1: {
            string filename;
            cout << "\n\nВведите название файла: ";
            cin >> filename;
            if (readAdjacencyMatrix(filename, graph)) {
                cout << "\n\nГраф успешно построен!\n\n";
                for (Edge* edge : graph.edges) {
```

```
                    cout << edge->src->name << " --(" <<
edge->weight << ")--> " << edge->dest->name << endl;
                }
            }
            else {
                cout << "Ошибка при считывании графа!" << endl;
            }
        }break;
        case 2: {
            Graph mst = kruskal(graph);
            cout << "Минимальное остовое дерево:" << endl;
            for (Edge* edge : mst.edges) {
                cout << edge->src->name << " --(" << edge->weight
<< ")--> " << edge->dest->name << endl;
            }
            int mstWeight = 0;
            for (Edge* edge : mst.edges) {
                mstWeight += edge->weight;
            }
            cout << "\n\nВес МСТ:" << mstWeight;
        }break;
        case 3: {
            int obhodChoose = -1;
            cout << "\n\nВыберите тип обхода:\n1. В глубину(DFS) -
рекурсивный\n2. В глубину(DFS) - итеративный\n3. В
ширину(BFS)\nВаш выбор:";
            cin >> obhodChoose;
            switch (obhodChoose) {
            case 1: {
                DFS(graph, graph.nodes[0]);
                break;
            }
            case 2: {
                DFSIterative(graph, graph.nodes[0]);
                break;
```

```
            }
            case 3: {
                BFS(graph, graph.nodes[0]);
                break;
            }
            }
   }break;
        case 4: {
            int printChoose = -1;
            cout << "\n\nВыберите представление графа: \n1. Список
смежности \n2. Матрица смежности \n3. Матрица инциндентности
\n\nВаш выбор:";
            cin >> printChoose;
            switch (printChoose) {
            case 1: {
                printAdjacencyList(graph);
                break;
            }
            case 2: {
                printAdjacencyMatrix(graph);
                break;
            }
            case 3: {
                printIncidenceMatrix(graph);
                break;
            }
            }
        }break;
  }
    }
    return 0;
}
```