|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| Министерство науки и высшего образования Российской Федерации | | | | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  «МИРЭА – Российский технологический университет»  РТУ МИРЭА | | | | | | |
| **Институт** | | ИКБ | | | | |
|  | | | | |  | |
| **Специальность (направление):** | | | | 09.03.02 (информационные системы и технологии) | | |
|  | | | | | |  |
| **Кафедра:** | | КБ-3 «Разработка программных решений и системного программирования» | | | | |
|  | | | | | |  |
| **Дисциплина:** | | | «Алгоритмы и структуры данных» | | | |

Практическая работа

на тему:

|  |
| --- |
| Программа по графам |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Студент: |  | |  | 17.10.2024 |  | Крашенинников М.В. |
|  | | *подпись* |  | *Дата* |  | *инициалы и фамилия* |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа: | БСБО-16-23 |  | Шифр: | 23Б0107 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Преподаватель: |  |  | 17.10.2024 |  | Филатов В.В. |
|  | *подпись* |  | *дата* |  | *инициалы и фамилия* |

**Москва 2024 г.**

**1. Задание (вариант 7)**

1. Определить и вывести ВСЕ (т.е. необязательно самые короткие) незамкнутые пути в орграфе заданной длины х (вводится с клавиатуры). Способ представления графа – список смежности

**2. Термины**

### 1) Ориентированный граф (орграф) — граф, где рёбра (дуги) имеют направление. Каждое ребро указывает, от какой вершины оно исходит и в какую ведёт.

### 2) Взвешенный граф — это граф, в котором каждому ребру (дуге) назначено числовое значение — вес, обычно соответствующий стоимости или длине.

### 3) Список смежности — структура данных, где для каждой вершины хранится список вершин, к которым есть дуги, и соответствующие веса этих дуг.

### 4) Путь в графе — это последовательность вершин, соединённых рёбрами (дугами), в которой каждое ребро ведёт от одной вершины к следующей.

### 5) Незамкнутый путь — путь, который не возвращается в начальную вершину, то есть начальная и конечная вершины различны.

### 6) Дуга — ориентированное ребро графа, которое соединяет одну вершину с другой и имеет направление.

### 7) Поиск в глубину (DFS) — алгоритм обхода графа, который исследует все возможные пути от одной вершины к другой, углубляясь до конца каждого пути, прежде чем переходить к следующему.

**3. Описание программы**

1. Основная цель программы

Программа предназначена для работы с ориентированным графом, представленным с использованием списка смежности. Она позволяет добавлять вершины и дуги, удалять их, редактировать, а также выполнять поиск всех путей в графе с общей стоимостью x, используя алгоритм поиска в глубину (DFS).

2. Структуры и классы

Vertex: представляет вершину графа. Вершина содержит название (name), уникальный номер/индекс (number), и флаг посещения (visited), который используется в DFS для пометки посещённых вершин.

Edge: представляет ориентированную дугу графа. Она содержит указатели на вершины начала и конца дуги (from, to), а также стоимость дуги (cost).

AdjNode: элемент списка смежности, который хранит дугу и указатель на следующий элемент списка.

AdjList: управляет списком смежности для каждой вершины, предоставляя доступ к дугам, исходящим из неё.

AdjListIterator и VertexIterator: итераторы для обхода вершин и элементов списка смежности.

3. Алгоритм добавления и удаления вершин и дуг

ADD\_V (добавление вершины) — метод добавляет новую вершину с именем name в массив вершин. Если были удалённые вершины, программа использует их место для новых вершин, чтобы избежать "дыр" в данных.

ADD\_E (добавление дуги) — добавляет дугу между двумя вершинами с именами vName и wName и стоимостью дуги c. Дуга добавляется в список смежности для вершины vName.

DEL\_V (удаление вершины) — метод удаляет вершину и все исходящие и входящие дуги. Вершина "помечается" удалённой через замену имени на символ '-'.

DEL\_E (удаление дуги) — удаляет дугу между вершинами vName и wName.

4. Поиск в глубину (DFS)

DFS\_START — запускает алгоритм DFS для всех вершин графа, которые не были посещены. Метод инициализирует путь и переменные для хранения текущей стоимости и длины пути.

DFS — основной метод для поиска путей с использованием рекурсии. Он начинает с текущей вершины (currentIndex) и посещает все её смежные вершины, пока не найдёт путь с суммарной стоимостью, равной заданной длине length.

Важная деталь — рекурсивное добавление стоимости дуги к пути с помощью функции getEdgeCost.

5. Другие вспомогательные методы

showAdjacencyList: выводит текущий список смежности, показывая для каждой вершины её смежные вершины и стоимость дуг.

setLength: задаёт длину пути, которую должен искать DFS.

FIRST и NEXT: методы для нахождения первой непосещённой вершины и последующих вершин в списке смежности.

6. Общий процесс работы программы

Пользователь добавляет вершины и дуги, создавая граф.

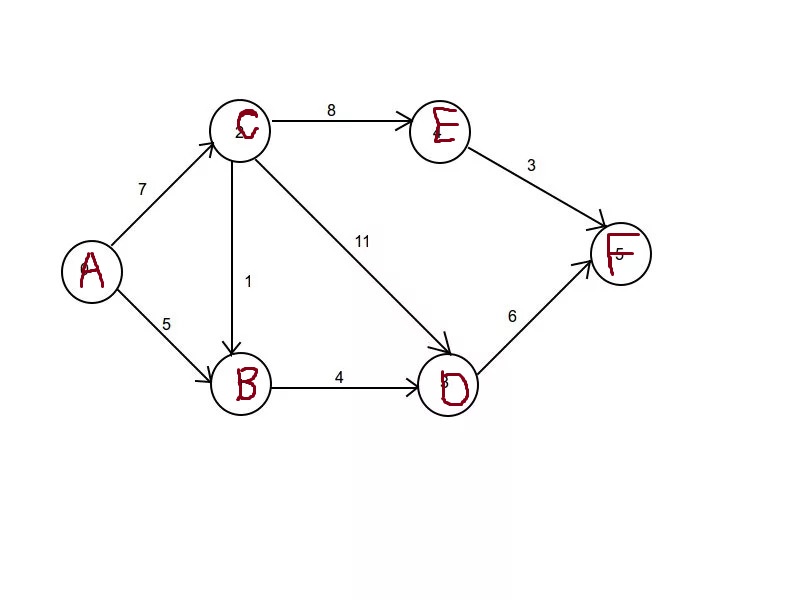
Задаётся целевая длина пути (сумма весов дуг).

Программа запускает поиск в глубину (DFS), находя все пути в графе, у которых суммарная стоимость дуг равна заданной длине.

Все найденные пути выводятся на экран.

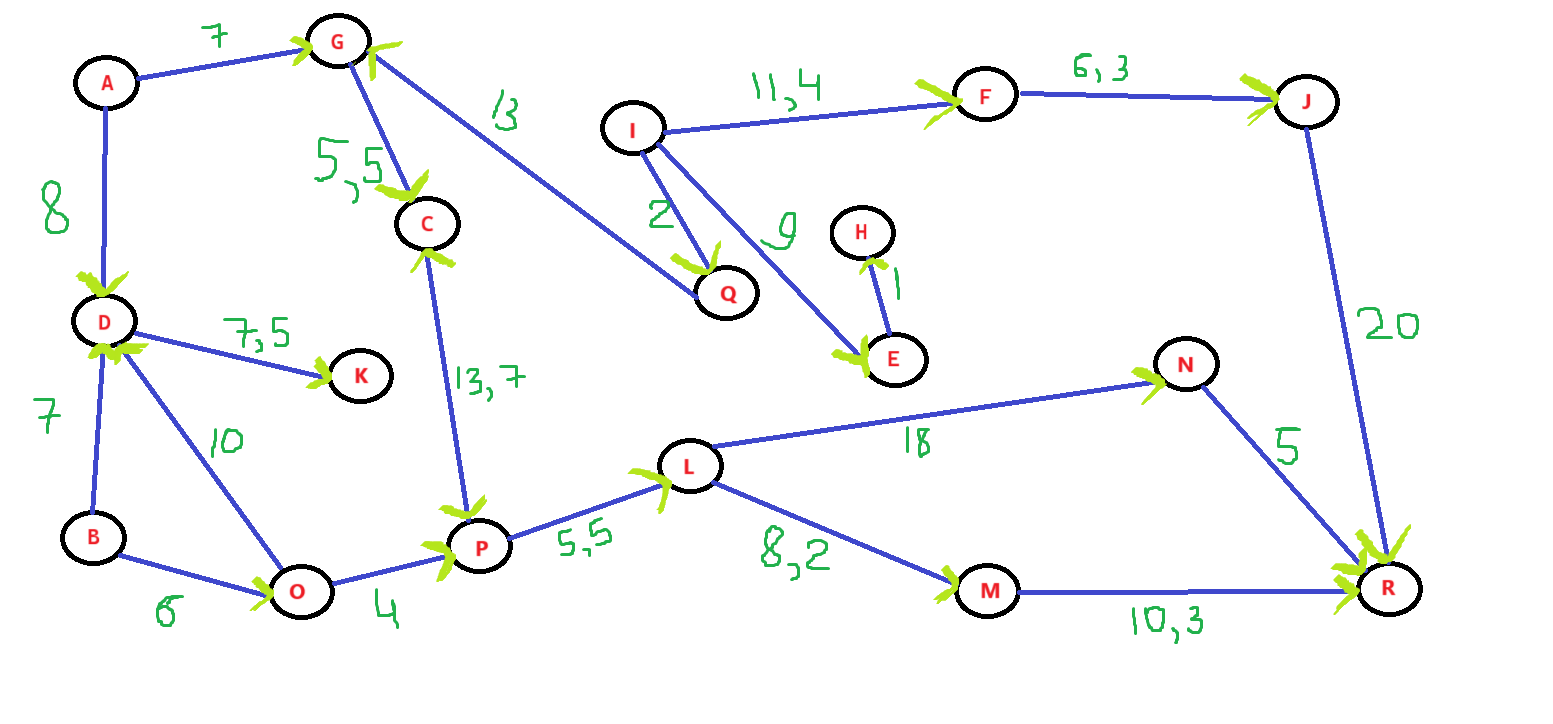
Программа реализована таким образом, чтобы можно было легко управлять графом, добавлять и удалять вершины/дуги, а также находить специфические пути по их стоимости.

**4. Изображения графов**

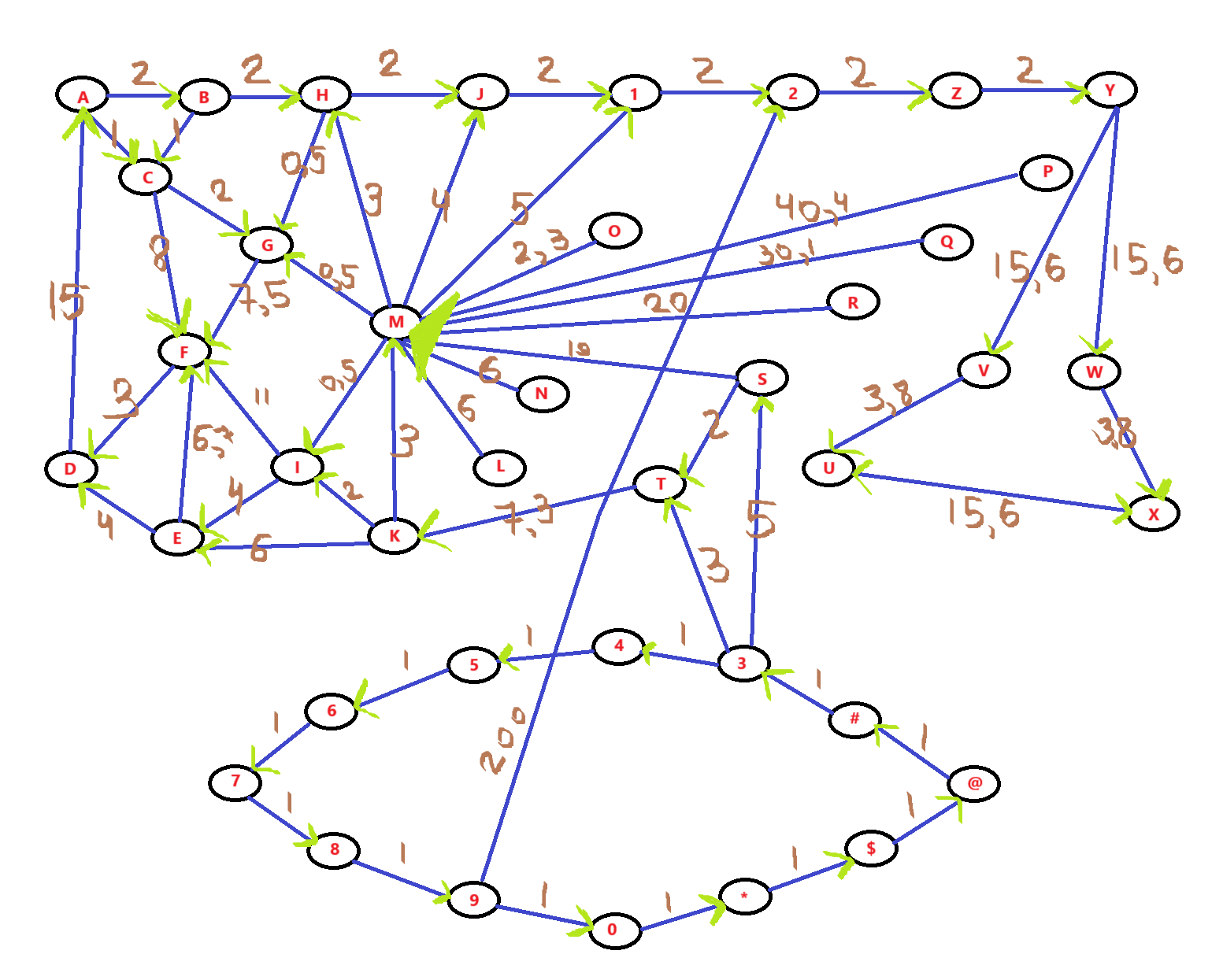


**GRAPH1:**

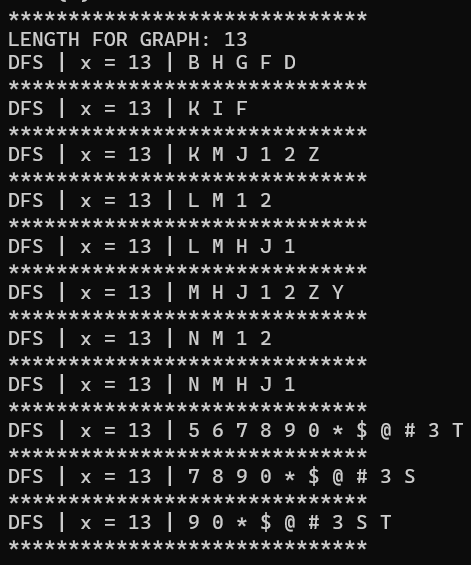
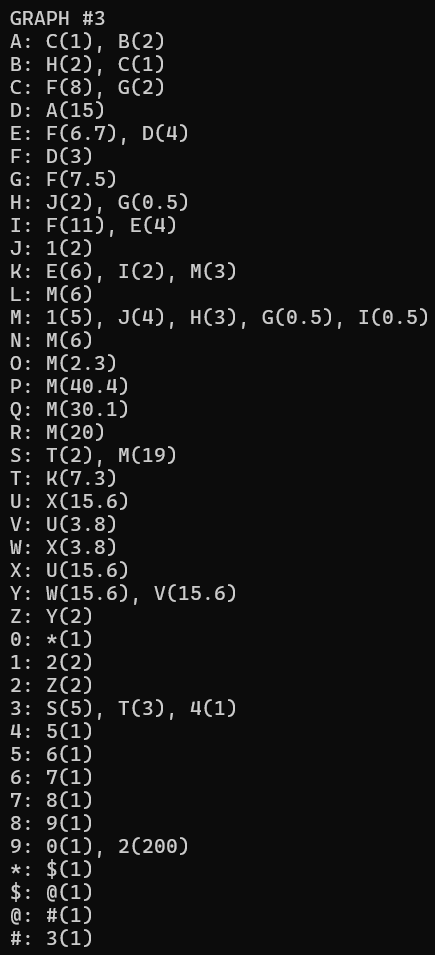
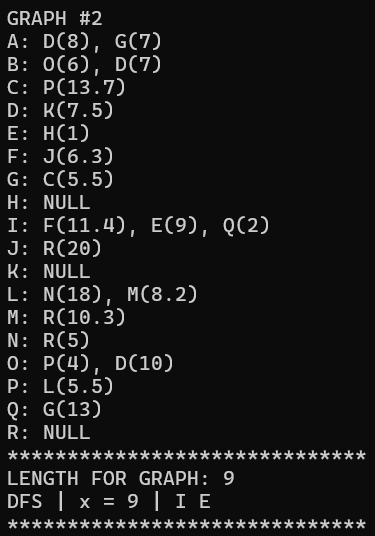
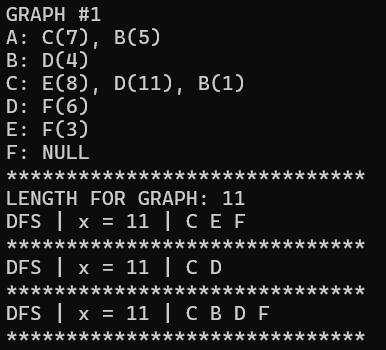
**GRAPH2:**



**GRAPH3:**



**5. Скриншот работы программы**



**6. Исходный код**

Для удобства - [matw0x/ALGORITHMS (github.com)](https://github.com/matw0x/ALGORITHMS)

**Структуры:**

#pragma once

#include "edge.h"

struct Vertex {

    char name; // название узла

    size\_t number; // номер узла

    bool visited; // был ли посещён

    Vertex(const char& name, const size\_t& number) :

        name(name), number(number), visited(false) {}

    Vertex() : name(' '), number(size\_t(-1)), visited(false) {}

    void setVisited() noexcept {

        visited = true;

    }

    void operator = (const Vertex& v) {

        name = v.name;

        number = v.number;

        visited = v.visited;

    }

    bool operator == (const Vertex\* v) const noexcept {

        return this->name == v->name && this->number == v->number;

    }

};

class VertexIterator {

private:

    Vertex\* vertices;

    size\_t index, count;

public:

    VertexIterator(Vertex\* v, const size\_t& c, const size\_t& start = 0) :

        vertices(v), index(start), count(c) {

        moveToValid();

    }

    void moveToValid() {

        while (index < count && vertices[index].name == '-') {

            ++index;

        }

    }

    VertexIterator& operator ++ () noexcept {

        ++index;

        moveToValid();

        return \*this;

    }

    Vertex& operator\*() const noexcept {

        return vertices[index];

    }

    bool operator != (const VertexIterator& other) const {

        return index != other.index;

    }

    static VertexIterator begin(Vertex\* vertices, size\_t count) noexcept {

        return VertexIterator(vertices, count);

    }

    static VertexIterator end(Vertex\* vertices, size\_t count) noexcept {

        return VertexIterator(vertices, count, count);

    }

};

#pragma once

#include "vertex.h"

struct Vertex;

struct Edge {

    Vertex \*from, \*to; // откуда и куда

    float cost; // стоимость (вес) дуги

    Edge(Vertex\* from, Vertex\* to, const float& cost) :

        from(from), to(to), cost(cost) {}

    // ~Edge(); - по факту реализация описана в ~Graph()

};

#pragma once

#include "vertex.h"

#include "edge.h"

struct AdjNode {

    Edge\* edge; // дуга (данные о двух вершинах и стоимости дуги)

    AdjNode\* next; // следующий элемент смежности

    AdjNode(Edge\* e) :

        edge(e), next(nullptr) {}

    ~AdjNode() {

        delete edge;

    }

};

struct AdjList {

    AdjNode\* head; // главный элемент

    AdjList() : head(nullptr) {}

    ~AdjList() {

        AdjNode\* current = head;

        while (current != nullptr) {

            AdjNode\* temp = current;

            current = current->next;

            delete temp;

        }

    }

};

class AdjListIterator {

private:

    AdjNode\* current;

public:

    AdjListIterator(AdjNode\* start) : current(start) {}

    AdjListIterator& operator ++ () noexcept {

        if (current != nullptr) current = current->next;

        return \*this;

    }

    AdjNode\* operator \* () const noexcept {

        return current;

    }

    bool operator != (const AdjListIterator& other) const noexcept {

        return current != other.current;

    }

    static AdjListIterator begin(AdjNode\* start) noexcept {

        return AdjListIterator(start);

    }

    static AdjListIterator end() noexcept {

        return AdjListIterator(nullptr);

    }

};

**Сам граф:**

#pragma once

#include <iostream>

#include <string>

#include "vertex.h"

#include "edge.h"

#include "adjacency\_list.h"

const std::string stars(30, '\*'); // для удобного вывода

#define PRINT\_STARS std::cout << stars + '\n';

class Graph {

private:

    Vertex\* vertices; // все вершины

    AdjList\* adjLists; // массив списков смежности (для каждой вершины отдельный список)

    size\_t countVertices, // кол-во вершин - исходное

           countEdges, // кол-во дуг - исходное

           NV, // кол-во вершин - фактическое (текущее) для вершин

           NE; // кол-во вершин - фактическое (текущее) для дуг

    float length; // та самая длина x

    size\_t\* deletedNumbers; // удалённые индексы | решение проблем с дырами

    size\_t deletedCount; // кол-во удалённых

    const size\_t UNREAL\_VALUE = this->countVertices + 1; // для неопределённых индексов

    Vertex\* findVertexByName(const char& vertexName) const noexcept {

        for (size\_t i = 0; i != countVertices; ++i) {

            if (vertices[i].name == vertexName && vertices[i].name != '-') return &vertices[i];

        }

        return nullptr;

    }

    void DFS(const size\_t& currentIndex, size\_t\* path, size\_t& pathLength, float& currentCost) const noexcept {

        vertices[currentIndex].setVisited();

        path[pathLength++] = currentIndex;

        auto getEdgeCost = [&](const size\_t& from, const size\_t& to) -> float {

            AdjNode\* current = adjLists[from].head;

            while (current != nullptr) {

                if (current->edge->to->number == to) {

                    return current->edge->cost;

                }

                current = current->next;

            }

            return 0.0f;

        };

        if (currentCost == length) {

            if (path[0] != currentIndex) {

                std::cout << "DFS | x = " << length << " | ";

                for (size\_t i = 0; i != pathLength; ++i) {

                    std::cout << vertices[path[i]].name << ' ';

                }

                std::cout << std::endl;

                PRINT\_STARS;

            }

        } else if (currentCost < length) {

            size\_t i = FIRST(vertices[currentIndex].name);

            while (i != UNREAL\_VALUE) {

                if (!vertices[i].visited) {

                    float edgeCost = getEdgeCost(currentIndex, i);

                    currentCost += edgeCost;

                    DFS(i, path, pathLength, currentCost);

                    currentCost -= edgeCost; // откатываем вес дуги после рекурсии

                }

                i = NEXT(vertices[currentIndex].name, i);

            }

        }

        vertices[currentIndex].visited = false;

        --pathLength;

    }

public:

    Graph(const size\_t& cV) :

    countVertices(cV), countEdges(cV\*cV - cV), deletedCount(NE = NV = 0) {

        vertices = new Vertex[countVertices];

        adjLists = new AdjList[countVertices];

        deletedNumbers = new size\_t[countVertices];

    }

    ~Graph() {

        for (size\_t i = 0; i != NV; ++i) adjLists[i].~AdjList();

        delete[] adjLists;

        delete[] vertices;

        delete[] deletedNumbers;

    }

    void showAdjacencyList() const noexcept {

        for (auto it = VertexIterator::begin(vertices, countVertices);

            it != VertexIterator::end(vertices, countVertices);

            ++it) {

            std::cout << (\*it).name << ": ";

            bool isEmpty = true;

            for (auto adjIt = AdjListIterator::begin(adjLists[(\*it).number].head);

                adjIt != AdjListIterator::end();

                ++adjIt) {

                isEmpty = false; // т.к. иное не может в принципе зайти в цикл

                std::cout << (\*adjIt)->edge->to->name

                        << '(' << (\*adjIt)->edge->cost << ')';

                if ((\*adjIt)->next != nullptr) std::cout << ", ";

            }

            if (isEmpty) std::cout << "NULL";

            std::cout << std::endl;

        }

        PRINT\_STARS;

    }

    void ADD\_V(const char& name) noexcept {

        for (size\_t i = 0; i != countVertices; ++i) {

            if (vertices[i].name == name) {

                std::cout << "ADD\_V: Vertex " << name << " already exists!\n";

            }

        }

        if (deletedCount > 0) {

            size\_t vertexNumber = deletedNumbers[--deletedCount];

            vertices[vertexNumber].name = name;

            adjLists[vertexNumber].head = nullptr;

            ++NV;

            return;

        }

        if (NV >= countVertices) {

            std::cout << "ADD\_V: Could not add a vertex " << name << " | NV >= countVertices!\n";

            return;

        } else {

            vertices[NV].name = name;

            vertices[NV].number = NV;

            adjLists[NV++].head = nullptr;

        }

    }

    void ADD\_E(const char& vName, const char& wName, const float& c) noexcept {

        if (NE >= countEdges) {

            std::cout << "ADD\_E: Could not add an edge from/to " << vName << '/' << wName << " | NE >= countEdges!\n";

            return;

        }

        Vertex\* from = findVertexByName(vName);

        Vertex\* to = findVertexByName(wName);

        if (from == nullptr) {

            std::cout << "ADD\_E: The vertex " << vName << " was not found!\n";

            return;

        }

        if (to == nullptr) {

            std::cout << "ADD\_E: The vertex " << wName << " was not found!\n";

            return;

        }

        if (from->name == '-' || to->name == '-') {

            std::cout << "ADD\_E: One of the vertices " << '[' << vName

            << ", " << wName << "] is deleted!\n";

            return;

        }

        AdjNode\* newNode = new AdjNode(new Edge(from, to, c));

        newNode->next = adjLists[from->number].head;

        adjLists[from->number].head = newNode;

        ++NE;

    }

    void DEL\_V(const char& name) noexcept {

        Vertex\* vertexToDelete = findVertexByName(name);

        if (vertexToDelete == nullptr) {

            std::cout << "DEL\_V: The vertex " << name << " was not found!\n";

        }

        // исходящая и внутренности

        AdjNode\* current = adjLists[vertexToDelete->number].head;

        while (current != nullptr) {

            AdjNode\* temp = current;

            current = current->next;

            delete temp;

            --NE;

        }

        adjLists[vertexToDelete->number].head = nullptr;

        // входящие

        for (size\_t i = 0; i != NV; ++i) {

            if (i != vertexToDelete->number) {

                AdjNode\* prev = nullptr;

                AdjNode\* current = adjLists[i].head;

                while (current != nullptr) {

                    if (current->edge->to->name == name) {

                        if (prev == nullptr) {

                            adjLists[i].head = current->next;

                        } else {

                            prev->next = current->next;

                        }

                        delete current;

                        --NE;

                        break;

                    }

                    prev = current;

                    current = current->next;

                }

            }

        }

        vertexToDelete->name = '-';

        deletedNumbers[deletedCount++] = vertexToDelete->number;

        --NV;

    }

    void DEL\_E(const char& vName, const char& wName) noexcept {

        Vertex\* from = findVertexByName(vName);

        Vertex\* to = findVertexByName(wName);

        if (from == nullptr) {

            std::cout << "ADD\_E: The vertex " << vName << " was not found!\n";

            return;

        }

        if (to == nullptr) {

            std::cout << "ADD\_E: The vertex " << wName << " was not found!\n";

            return;

        }

        if (from->name == '-' || to->name == '-') {

            std::cout << "ADD\_E: One of the vertices " << '[' << vName

            << ", " << wName << "] is deleted!\n";

            return;

        }

        AdjNode\* prev = nullptr;

        AdjNode\* current = adjLists[from->number].head;

        while (current != nullptr) {

            if (current->edge->to == to) {

                if (prev == nullptr) {

                    adjLists[from->number].head = current->next;

                } else {

                    prev->next = current->next;

                }

                delete current;

                --NE;

                return;

            }

            prev = current;

            current = current->next;

        }

    }

    void EDIT\_V(const char& name, const bool& newVisited) const noexcept {

        bool success = false;

        for (size\_t i = 0; i != NV; ++i) {

            if (vertices[i].name == name) {

                vertices[i].visited = newVisited;

                return;

            }

        }

        if (!success) std::cout << "EDIT\_V: Vertex " << name << " was not found!\n";

    }

    void EDIT\_E(const char& vName, const char& wName, const float& newCost) const noexcept {

        Vertex\* from = findVertexByName(vName);

        Vertex\* to = findVertexByName(wName);

        if (from == nullptr) {

            std::cout << "EDIT\_E: Vertex " << vName << " was not found!\n";

            return;

        }

        if (to == nullptr) {

            std::cout << "EDIT\_E: Vertex " << wName << " was not found!\n";

            return;

        }

        if (from->name == '-' || to->name == '-') {

            std::cout << "ADD\_E: One of the vertices [" << vName

            << ", " << wName << "] is deleted!\n";

        }

        AdjNode\* current = adjLists[from->number].head;

        while (current != nullptr) {

            if (current->edge->to == to) current->edge->cost = newCost;

            current = current->next;

        }

    }

    size\_t FIRST(const char& name) const noexcept {

        Vertex\* from = findVertexByName(name);

        if (from == nullptr) {

            std::cout << "FIRST: Vertex " << name << " was not found!\n";

            return UNREAL\_VALUE;

        }

        AdjNode\* current = adjLists[from->number].head;

        while (current != nullptr && current->edge->to->visited) {

            current = current->next;

        }

        if (current == nullptr) return UNREAL\_VALUE;

        return current->edge->to->number;

    }

    size\_t NEXT(const char& name, const size\_t& i) const noexcept {

        Vertex\* from = findVertexByName(name);

        if (from == nullptr) {

            std::cout << "NEXT: Vertex " << name << " was not found!\n";

            return UNREAL\_VALUE;

        }

        AdjNode\* current = adjLists[from->number].head;

        while (current != nullptr) {

            if (current->edge->to->number == i) {

                current = current->next;

                while (current != nullptr && current->edge->to->visited) {

                    current = current->next;

                }

                if (current == nullptr) return UNREAL\_VALUE;

                return current->edge->to->number;

            }

            current = current->next;

        }

        return UNREAL\_VALUE;

    }

    Vertex\* VERTEX(const char& name, const size\_t& i) const noexcept {

        Vertex\* from = findVertexByName(name);

        if (from == nullptr) {

            std::cout << "VERTEX: Vertex " << name << " was not found!\n";

            return nullptr;

        }

        AdjNode\* current = adjLists[from->number].head;

        while (current != nullptr) {

            if (current->edge->to->number == i) return current->edge->to;

            current = current->next;

        }

        return nullptr;

    }

    void DFS\_START() const noexcept {

        if (NV == 0 || countVertices == 0) {

            return;

        }

        size\_t\* path = new size\_t[countVertices];

        float currentCost = 0.0f;

        size\_t pathLength = 0;

        for (size\_t i = 0; i != NV; ++i) {

            if (!vertices[i].visited) {

                DFS(i, path, pathLength, currentCost);

            }

            currentCost = 0.0f;

            pathLength = 0;

        }

        delete[] path;

    }

    void setLength(const float& x) {

        length = x;

    }

};

**Точка входа:**

#include <iostream>

#include <utility>

#include "graph.h"

typedef std::tuple<char, char, float> E;

void creating(Graph& g, const char\* v, const E\* e, const size\_t& cV, const size\_t& cE);

void solveWithInput(Graph& g);

void solveWithCycle(Graph& g);

int main() {

    const size\_t COUNT\_VERTICES\_1 = 6;

    const size\_t COUNT\_EDGES\_1 = 8;

    char v1[COUNT\_VERTICES\_1] = { 'A', 'B', 'C', 'D', 'E', 'F' };

    E e1[COUNT\_EDGES\_1] = {

        {'A', 'B', 5}, {'A', 'C', 7}, {'B', 'D', 4}, {'C', 'B', 1},

        {'C', 'D', 11}, {'C', 'E', 8}, {'D', 'F', 6}, {'E', 'F', 3}

    };

    Graph graph1(COUNT\_VERTICES\_1);

    creating(graph1, v1, e1, COUNT\_VERTICES\_1, COUNT\_EDGES\_1);

    std::cout << "GRAPH #1\n";

    graph1.showAdjacencyList();

    solveWithInput(graph1);

    // solveWithCycle(graph1);

    // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    const size\_t COUNT\_VERTICES\_2 = 18;

    const size\_t COUNT\_EDGES\_2 = 21;

    char v2[COUNT\_VERTICES\_2] = {

        'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',

        'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R'

    };

    E e2[COUNT\_EDGES\_2] = {

        {'A', 'G', 7}, {'A', 'D', 8}, {'B', 'D', 7}, {'B', 'O', 6},

        {'C', 'P', 13.7}, {'D', 'K', 7.5}, {'E', 'H', 1}, {'F', 'J', 6.3},

        {'G', 'C', 5.5}, {'I', 'Q', 2}, {'I', 'E', 9}, {'I', 'F', 11.4},

        {'J', 'R', 20}, {'L', 'M', 8.2}, {'L', 'N', 18}, {'M', 'R', 10.3},

        {'N', 'R', 5}, {'O', 'D', 10}, {'O', 'P', 4}, {'P', 'L', 5.5},

        {'Q', 'G', 13}

    };

    Graph graph2(COUNT\_VERTICES\_2);

    creating(graph2, v2, e2, COUNT\_VERTICES\_2, COUNT\_EDGES\_2);

    std::cout << "GRAPH #2\n";

    graph2.showAdjacencyList();

    solveWithInput(graph2);

    // solveWithCycle(graph2);

    // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    const size\_t COUNT\_VERTICES\_3 = 40;

    const size\_t COUNT\_EDGES\_3 = 57;

    char v3[COUNT\_VERTICES\_3] = {

        'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',

        'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',

        'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2', '3',

        '4', '5', '6', '7', '8', '9', '\*', '$', '@', '#'

    };

    E e3[COUNT\_EDGES\_3] = {

        {'A', 'B', 2}, {'A', 'C', 1}, {'B', 'C', 1}, {'B', 'H', 2},

        {'C', 'G', 2}, {'C', 'F', 8}, {'D', 'A', 15}, {'E', 'D', 4},

        {'E', 'F', 6.7}, {'F', 'D', 3}, {'G', 'F', 7.5}, {'H', 'G', 0.5},

        {'H', 'J', 2}, {'I', 'E', 4}, {'I', 'F', 11}, {'J', '1', 2},

        {'K', 'M', 3}, {'K', 'I', 2}, {'K', 'E', 6}, {'L', 'M', 6},

        {'M', 'I', 0.5}, {'M', 'G', 0.5}, {'M', 'H', 3}, {'M', 'J', 4},

        {'M', '1', 5}, {'N', 'M', 6}, {'O', 'M', 2.3}, {'P', 'M', 40.4},

        {'Q', 'M', 30.1}, {'R', 'M', 20}, {'S', 'M', 19}, {'S', 'T', 2},

        {'T', 'K', 7.3}, {'U', 'X', 15.6}, {'V', 'U', 3.8}, {'W', 'X', 3.8},

        {'X', 'U', 15.6}, {'Y', 'V', 15.6}, {'Y', 'W', 15.6}, {'Z', 'Y', 2},

        {'0', '\*', 1}, {'1', '2', 2}, {'2', 'Z', 2}, {'3', '4', 1},

        {'4', '5', 1}, {'5', '6', 1}, {'6', '7', 1}, {'7', '8', 1},

        {'8', '9', 1}, {'9', '2', 200}, {'9', '0', 1}, {'\*', '$', 1},

        {'$', '@', 1}, {'@', '#', 1}, {'#', '3', 1}, {'3', 'T', 3}, {'3', 'S', 5}

    };

    Graph graph3(COUNT\_VERTICES\_3);

    creating(graph3, v3, e3, COUNT\_VERTICES\_3, COUNT\_EDGES\_3);

    std::cout << "GRAPH #3\n";

    graph3.showAdjacencyList();

    solveWithInput(graph3);

    // solveWithCycle(graph3);

    return 0;

}

void creating(Graph& g, const char\* v, const E\* e, const size\_t& cV, const size\_t& cE) {

    for (size\_t i = 0; i != cV; ++i) {

        g.ADD\_V(v[i]);

    }

    for (size\_t i = 0; i != cE; ++i) {

        g.ADD\_E(std::get<0>(e[i]), std::get<1>(e[i]), std::get<2>(e[i]));

    }

}

void solveWithInput(Graph& g) {

    float x;

    std::cout << "LENGTH FOR GRAPH: "; std::cin >> x;

    g.setLength(x);

    g.DFS\_START();

    std::cout << std::endl;

}

void solveWithCycle(Graph& g) {

    for (int i = 0; i <= 5e3; ++i) {

        float x = i \* 0.1f;

        g.setLength(x);

        g.DFS\_START();

    }

    std::cout << std::endl;

}

**Вывод:** в процессе реализации программы для поиска всех незамкнутых путей в ориентированном графе я значительно углубил свои знания в области графов и алгоритмов. Я научился представлять графы с помощью списков смежности, что позволило оптимально организовать данные и оптимизировать доступ к ним. Также понял, как эффективно отслеживать посещённые узлы и управлять состоянием пути, что является важным аспектом работы с графами.

В целом, это задание не только укрепило мои знания в области алгоритмов и структур данных, но и развило критическое мышление и способности к решению проблем.

**7. Литература**

1. Тюкачев Н. А., Хлебостроев В. Г. - C#. Алгоритмы и структуры данных, стр. 185, 5.7.2. Приближенные алгоритмы раскраски графа
2. Иванов Б. Н. - Дискретная математика. Алгоритмы и программы. Расширенный курс, стр. 356, Глава 7. Теория графов. Алгоритмы на графах
3. Лекции и практики – преподаватель Филатов В. В.