

Дисциплина: «Алгоритмы и структуры данных»

Практическая работа
на тему:
Программа по графам

Студент: _____ 29.11.2024 _____ Альзоаби А.Ф.
подпись Дата инициалы и фамилия

Группа: БСБО-16-23 Шифр: 23Б0045

Преподаватель:	29.11.2024	Филатов В.В.
<i>подпись</i>	<i>дата</i>	<i>инициалы и фамилия</i>

Москва 2024 г.

1. **Задание.** вывести на экран все существующие пути в ациклическом орграфе. Способ представления графа: матрица смежности
2. **Термины.**

1) Ациклический орграф — это направленный граф, в котором отсутствуют циклы. То есть, в таком графе невозможно начать движение от вершины и вернуться в ту же вершину, следуя по рёбрам графа. Ациклические ориентированные графы (Acyclic Directed Graph, DAG) широко используются в различных задачах, таких как топологическая сортировка, планирование, управление зависимостями и многие другие.

2) Алгоритм поиска в глубину (Depth-First Search, DFS) — это метод обхода графа, при котором исследуются как можно более глубокие ветви графа, прежде чем возвращаться и исследовать соседние вершины. Алгоритм используется для поиска всех путей от одной вершины к другой, для проверки связности графа, нахождения компонент связности, а также для решения задач на деревьях и графах, таких как нахождение топологической сортировки.

Начинаем с исходной вершины. Рекурсивно посещаем соседние вершины, которые еще не были посещены. Когда все соседи текущей вершины исследованы, возвращаемся к предыдущей вершине и продолжаем обход.

3) Матрица смежности — это способ представления графа в виде двумерной матрицы, где строки и столбцы соответствуют вершинам графа, а элементы матрицы показывают наличие или отсутствие рёбер между вершинами. Если существует ребро от вершины i к вершине j , то в ячейке матрицы будет стоять значение 1 (или другой вес, если граф взвешенный), иначе 0.

4) Рекурсия — это метод решения задачи, при котором функция вызывает саму себя для решения подзадачи. В DFS рекурсия используется для обхода графа: функция вызывается для каждой смежной вершины, пока не будут исследованы все возможные пути от текущей вершины.

5) Путь в графе — это последовательность рёбер, которые соединяют вершины. Путь начинается с одной вершины и заканчивается на другой, переходя от вершины к вершине через рёбра графа. В случае ориентированного графа путь может двигаться только по направлению

рёбер. В ациклическом графе путь не может повторять вершины, так как это нарушало бы аксиому "отсутствие циклов".

3. Описание алгоритма.

Стартовая функция для решения задания. Сначала происходит проверка на пустоту вектора вершин, а затем – дуг. Далее происходит объявление вектора в векторе из строк, который будет хранить в себе все существующие пути. Так как я использую DFS через рекурсию, то мне нужно добавить вспомогательную функцию `findAllPaths`, которая будет исполняться для каждой вершинки. В конце выводится результат.

```
void runSolving() {  
    if (data.vertices.empty()) {  
        std::cerr << "Have no vertices!\n";  
        return;  
    }  
  
    if (data.edges.empty()) {  
        std::cerr << "Have no edges!\n";  
        return;  
    }  
  
    std::vector<std::vector<std::string>> allPaths;  
  
    for (size_t start = 0; start != data.vertices.size(); ++start) {  
        std::vector<std::string> path;  
        findAllPaths(start, path, allPaths);  
    }  
  
    std::cout << "\n\n\t\tANSWER:\n";  
    for (const auto& path : allPaths) {  
        for (const auto& vertex : path) {  
            std::cout << vertex << ' ' ;  
        }  
    }  
}
```

```

        std::cout << std::endl;
    }
}

```

Перейдём к другой функции. Сразу же меняем значение метки на true (посещена), чтобы не «наступить» на неё лишний раз, затем добавляем в путь нашу вершину. isLeaf отвечает за то, есть ли ещё непосещённые соседи. Проходимся по всей строке у текущей вершины и ищем соседей, которые не были посещены. Нашли соседа – рекурсивно вызываем функцию от соседа и так далее. После того, как поработали с соседями, обрабатываем логику со всеми существующими путями – добавляем найденные в результат и откатываемся: очищаем посещение вершины и удаляем из пути.

```

void findAllPaths(const size_t& current, std::vector<std::string>& path,
    std::vector<std::vector<std::string>>& allPaths) {
    data.vertices[current].visited = true;
    path.push_back(data.vertices[current].name);

    bool isLeaf = true;

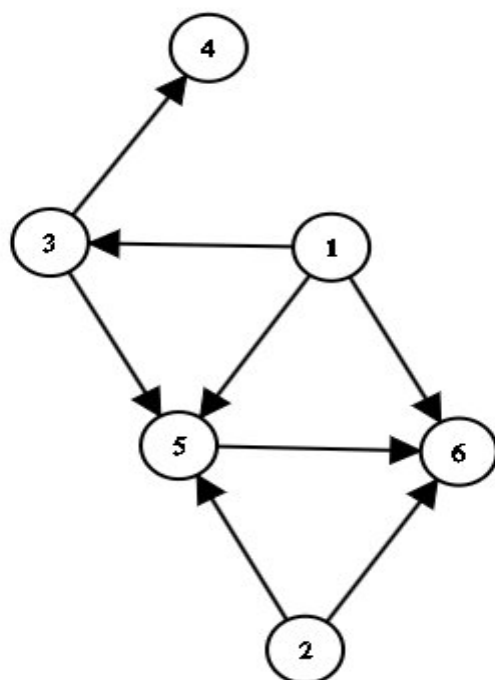
    for (size_t next = 0; next != data.adjacencyMatrix[current].size();
++next) {
        if (data.adjacencyMatrix[current][next]
&& !data.vertices[next].visited) {
            isLeaf = false;
            findAllPaths(next, path, allPaths);
        }
    }

    if (isLeaf) {
        allPaths.push_back(path);
    }

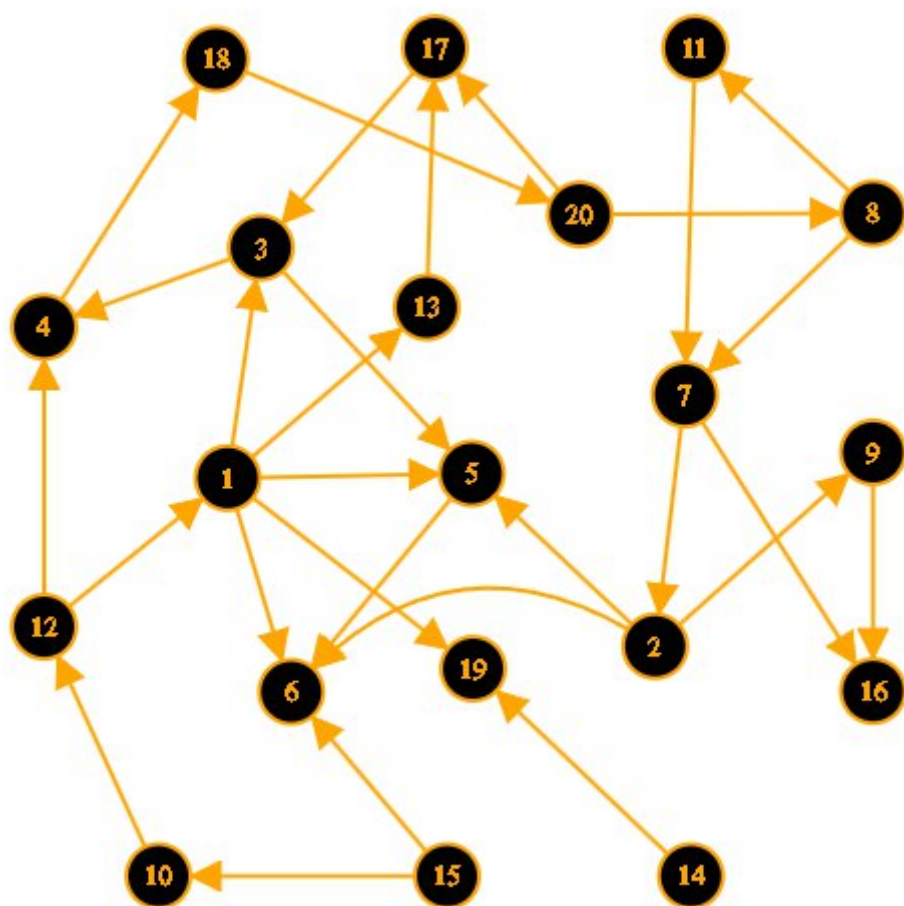
    data.vertices[current].visited = false;
    path.pop_back();
}

```

Первый:



Второй:



5. Скриншот работы программы:

Для первого и второго графов соответственно.

Vertices

1 2 3 4 5 6

Adjacency Matrix:

		1	2	3	4	5	6
1		0	0	1	0	1	1
2		0	0	0	0	1	1
3		0	0	0	1	1	0
4		0	0	0	0	0	0
5		0	0	0	0	0	1
6		0	0	0	0	0	0

ANSWER :

1 3 4
1 3 5 6
1 5 6
1 6
2 5 6
2 6
3 4
3 5 6
4
5 6
6

Vertices

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Adjacency Matrix:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1		0	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0
2		0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
3		0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
5		0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7		0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
8		0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
9		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10		0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
11		0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
12		1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
14		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15		0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
16		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17		0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
19		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20		0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0

ANSWER:

```
1 3 4 18 20 8 7 2 5 6
1 3 4 18 20 8 7 2 6
1 3 4 18 20 8 7 2 9 16
1 3 4 18 20 8 7 16
1 3 4 18 20 8 11 7 2 5 6
1 3 4 18 20 8 11 7 2 6
1 3 4 18 20 8 11 7 2 9 16
1 3 4 18 20 8 11 7 16
1 3 4 18 20 17
1 3 5 6
1 5 6
1 6
1 13 17 3 4 18 20 8 7 2 5 6
1 13 17 3 4 18 20 8 7 2 6
1 13 17 3 4 18 20 8 7 2 9 16
1 13 17 3 4 18 20 8 7 16
1 13 17 3 4 18 20 8 11 7 2 5 6
1 13 17 3 4 18 20 8 11 7 2 6
1 13 17 3 4 18 20 8 11 7 2 9 16
1 13 17 3 4 18 20 8 11 7 16
1 13 17 3 5 6
1 19
2 5 6
2 6
2 9 16
3 4 18 20 8 7 2 5 6
3 4 18 20 8 7 2 6
3 4 18 20 8 7 2 9 16
3 4 18 20 8 7 16
3 4 18 20 8 11 7 2 5 6
3 4 18 20 8 11 7 2 6
3 4 18 20 8 11 7 2 9 16
```

```
3 4 18 20 8 11 7 16
3 4 18 20 17
3 5 6
4 18 20 8 7 2 5 6
4 18 20 8 7 2 6
4 18 20 8 7 2 9 16
4 18 20 8 7 16
4 18 20 8 11 7 2 5 6
4 18 20 8 11 7 2 6
4 18 20 8 11 7 2 9 16
4 18 20 8 11 7 16
4 18 20 17 3 5 6
5 6
6
7 2 5 6
7 2 6
7 2 9 16
7 16
8 7 2 5 6
8 7 2 6
8 7 2 9 16
8 7 16
8 11 7 2 5 6
8 11 7 2 6
8 11 7 2 9 16
8 11 7 16
9 16
10 12 1 3 4 18 20 8 7 2 5 6
10 12 1 3 4 18 20 8 7 2 6
10 12 1 3 4 18 20 8 7 2 9 16
10 12 1 3 4 18 20 8 7 16
10 12 1 3 4 18 20 8 11 7 2 5 6
10 12 1 3 4 18 20 8 11 7 2 6
```

10 12 1 3 4 18 20 8 7 2 9 16
10 12 1 3 4 18 20 8 7 16
10 12 1 3 4 18 20 8 11 7 2 5 6
10 12 1 3 4 18 20 8 11 7 2 6
10 12 1 3 4 18 20 8 11 7 2 9 16
10 12 1 3 4 18 20 8 11 7 16
10 12 1 3 4 18 20 17
10 12 1 3 5 6
10 12 1 5 6
10 12 1 6
10 12 1 13 17 3 4 18 20 8 7 2 5 6
10 12 1 13 17 3 4 18 20 8 7 2 6
10 12 1 13 17 3 4 18 20 8 7 2 9 16
10 12 1 13 17 3 4 18 20 8 7 16
10 12 1 13 17 3 4 18 20 8 11 7 2 5 6
10 12 1 13 17 3 4 18 20 8 11 7 2 6
10 12 1 13 17 3 4 18 20 8 11 7 2 9 16
10 12 1 13 17 3 4 18 20 8 11 7 16
10 12 1 13 17 3 5 6
10 12 1 19
10 12 4 18 20 8 7 2 5 6
10 12 4 18 20 8 7 2 6
10 12 4 18 20 8 7 2 9 16
10 12 4 18 20 8 7 16
10 12 4 18 20 8 11 7 2 5 6
10 12 4 18 20 8 11 7 2 6
10 12 4 18 20 8 11 7 2 9 16
10 12 4 18 20 8 11 7 16
10 12 4 18 20 17 3 5 6
11 7 2 5 6
11 7 2 6
11 7 2 9 16
11 7 16

12 1 3 4 18 20 8 7 2 5 6
12 1 3 4 18 20 8 7 2 6
12 1 3 4 18 20 8 7 2 9 16
12 1 3 4 18 20 8 7 16
12 1 3 4 18 20 8 11 7 2 5 6
12 1 3 4 18 20 8 11 7 2 6
12 1 3 4 18 20 8 11 7 2 9 16
12 1 3 4 18 20 8 11 7 16
12 1 3 4 18 20 17
12 1 3 5 6
12 1 5 6
12 1 6
12 1 13 17 3 4 18 20 8 7 2 5 6
12 1 13 17 3 4 18 20 8 7 2 6
12 1 13 17 3 4 18 20 8 7 2 9 16
12 1 13 17 3 4 18 20 8 7 16
12 1 13 17 3 4 18 20 8 11 7 2 5 6
12 1 13 17 3 4 18 20 8 11 7 2 6
12 1 13 17 3 4 18 20 8 11 7 2 9 16
12 1 13 17 3 4 18 20 8 11 7 16
12 1 13 17 3 5 6
12 1 19
12 4 18 20 8 7 2 5 6
12 4 18 20 8 7 2 6
12 4 18 20 8 7 2 9 16
12 4 18 20 8 7 16
12 4 18 20 8 11 7 2 5 6
12 4 18 20 8 11 7 2 6
12 4 18 20 8 11 7 2 9 16
12 4 18 20 8 11 7 16
12 4 18 20 17 3 5 6
13 17 3 4 18 20 8 7 2 5 6
13 17 3 4 18 20 8 7 2 6

13 17 3 4 18 20 8 7 2 9 16
 13 17 3 4 18 20 8 7 16
 13 17 3 4 18 20 8 11 7 2 5 6
 13 17 3 4 18 20 8 11 7 2 6
 13 17 3 4 18 20 8 11 7 2 9 16
 13 17 3 4 18 20 8 11 7 16
 13 17 3 5 6
 14 19
 15 6
 15 10 12 1 3 4 18 20 8 7 2 5 6
 15 10 12 1 3 4 18 20 8 7 2 6
 15 10 12 1 3 4 18 20 8 7 2 9 16
 15 10 12 1 3 4 18 20 8 7 16
 15 10 12 1 3 4 18 20 8 11 7 2 5 6
 15 10 12 1 3 4 18 20 8 11 7 2 6
 15 10 12 1 3 4 18 20 8 11 7 2 9 16
 15 10 12 1 3 4 18 20 8 11 7 16
 15 10 12 1 3 4 18 20 17
 15 10 12 1 3 5 6
 15 10 12 1 5 6
 15 10 12 1 6
 15 10 12 1 13 17 3 4 18 20 8 7 2 5 6
 15 10 12 1 13 17 3 4 18 20 8 7 2 6
 15 10 12 1 13 17 3 4 18 20 8 7 2 9 16
 15 10 12 1 13 17 3 4 18 20 8 7 16
 15 10 12 1 13 17 3 4 18 20 8 11 7 2 5 6
 15 10 12 1 13 17 3 4 18 20 8 11 7 2 6
 15 10 12 1 13 17 3 4 18 20 8 11 7 2 9 16
 15 10 12 1 13 17 3 4 18 20 8 11 7 16
 15 10 12 1 13 17 3 5 6
 15 10 12 1 19
 15 10 12 4 18 20 8 7 2 5 6
 15 10 12 4 18 20 8 7 2 6

15 10 12 4 18 20 8 7 2 9 16
 15 10 12 4 18 20 8 7 16
 15 10 12 4 18 20 8 11 7 2 5 6
 15 10 12 4 18 20 8 11 7 2 6
 15 10 12 4 18 20 8 11 7 2 9 16
 15 10 12 4 18 20 8 11 7 16
 15 10 12 4 18 20 17 3 5 6
 16
 17 3 4 18 20 8 7 2 5 6
 17 3 4 18 20 8 7 2 6
 17 3 4 18 20 8 7 2 9 16
 17 3 4 18 20 8 7 16
 17 3 4 18 20 8 11 7 2 5 6
 17 3 4 18 20 8 11 7 2 6
 17 3 4 18 20 8 11 7 2 9 16
 17 3 4 18 20 8 11 7 16
 17 3 5 6
 18 20 8 7 2 5 6
 18 20 8 7 2 6
 18 20 8 7 2 9 16
 18 20 8 7 16
 18 20 8 11 7 2 5 6
 18 20 8 11 7 2 6
 18 20 8 11 7 2 9 16
 18 20 8 11 7 16
 18 20 17 3 4
 18 20 17 3 5 6
 19
 20 8 7 2 5 6
 20 8 7 2 6
 20 8 7 2 9 16
 20 8 7 16
 20 8 11 7 2 5 6

20 8 11 7 2 6
 20 8 11 7 2 9 16
 20 8 11 7 16
 20 17 3 4 18
 20 17 3 5 6

6. Исходный код.

// Алгоритм: Вывести на экран все существующие пути в ациклическом орграфе

// Способ представления графа: Матрица смежности

// Студент: Вариант 45 - Альзоаби Адель, БСБО-16-23

```
#include <iostream>
```

```
#include <string>
```

```
#include <vector>
```

```
#include <queue>
```

```
#include <sstream>
```

```
#include <iomanip>
```

```
struct Vertex {
```

```
    std::string name;
```

```
    bool visited; // mark
```

```
    size_t index;
```

```
    Vertex(const std::string& n, const bool& v, const size_t& i) :
```

```
        name(n), visited(v), index(i) {}
```

```
    Vertex() : name(""), visited(false), index(size_t(-1)) {}
```

```
    bool operator == (const Vertex& other) const {
```

```
        return this->name == other.name;
```

```
    }
```

```
};
```

```
struct Edge {
```

```
    Vertex start, end;
```

```
    float cost;
```

```
    Edge(const Vertex& v, const Vertex& w, const float& c = 0.0) : start(v),  
        end(w), cost(c) {}
```

```
};
```

```
struct GraphData {  
    std::vector<std::vector<bool>> adjacencyMatrix;  
    std::vector<Vertex> vertices;  
    std::vector<Edge> edges;  
};
```

```
class Graph {  
private:  
    GraphData data;  
  
    bool isVertexExists(const Vertex& vertex) {  
        return findVertexByName(vertex.name) != nullptr;  
    }  
  
    bool isVertexExists(const std::string& name) {  
        return findVertexByName(name) != nullptr;  
    }  
  
    bool isVertexExists(const Vertex* vertex) const {  
        return vertex != nullptr;  
    }  
  
    Vertex* findVertexByName(const std::string& name) {  
        for (auto& vertex : data.vertices) {  
            if (vertex.name == name) {  
                return &vertex;  
            }  
        }  
  
        return nullptr;  
    }  
};
```

```

    }

    void findAllPaths(const size_t& current, std::vector<std::string>& path,
std::vector<std::vector<std::string>>& allPaths) {

        data.vertices[current].visited = true;

        path.push_back(data.vertices[current].name);

        bool isLeaf = true;

        for (size_t next = 0; next != data.adjacencyMatrix[current].size();
++next) {

            if (data.adjacencyMatrix[current][next]
&& !data.vertices[next].visited) {

                isLeaf = false;

                findAllPaths(next, path, allPaths);

            }

        }

        if (isLeaf) {

            allPaths.push_back(path);

        }

        data.vertices[current].visited = false;

        path.pop_back();

    }

public:

    void ADD_V(const std::string& name, const bool& mark) {

        if (isVertexExists(name)) {

            std::cerr << "Can't add a vertex " << name << " : already exists!\n";

            return;

        }

        data.vertices.push_back(Vertex(name, mark, data.vertices.size()));

```

```

        for (auto& row : data.adjacencyMatrix) {
            row.push_back(false);
        }
        data.adjacencyMatrix.emplace_back(data.vertices.size(), false);
    }

void DEL_V(const std::string& name) {
    Vertex* toDelete = findVertexByName(name);

    if (!isVertexExists(toDelete)) {
        std::cerr << "Can't delete a vertex " << name << " : does not
exists!\n";
        return;
    }

    size_t index = toDelete->index;
    data.vertices.erase(data.vertices.begin() + index);
    data.adjacencyMatrix.erase(data.adjacencyMatrix.begin() + index);

    for (auto& row : data.adjacencyMatrix) {
        row.erase(row.begin() + index);
    }

    for (size_t i = 0; i != data.vertices.size(); ++i) {
        data.vertices[i].index = i;
    }
}

void ADD_E(const std::string& v, const std::string& w) {
    Vertex *start = findVertexByName(v), *end = findVertexByName(w);

    if (!isVertexExists(start)) {
        std::cerr << "Vertex " << v << " does not exist!\n";
    }
}

```

```

        return;
    }

    if (!isVertexExists(end)) {
        std::cerr << "Vertex " << w << " does not exist!\n";
        return;
    }

    if (data.adjacencyMatrix[start->index][end->index]) {
        std::cerr << "Edge " << v << " -> " << w << " already exists!\n";
        return;
    }

    data.adjacencyMatrix[start->index][end->index] = true;
    data.edges.push_back(Edge(*start, *end));
}

void DEL_E(const std::string& v, const std::string& w) {
    Vertex *start = findVertexByName(v), *end = findVertexByName(w);

    if (!isVertexExists(start)) {
        std::cerr << "Vertex " << v << " does not exist!\n";
        return;
    }

    if (!isVertexExists(end)) {
        std::cerr << "Vertex " << w << " does not exist!\n";
        return;
    }

    if (!data.adjacencyMatrix[start->index][end->index]) {
        std::cerr << "Edge " << v << " -> " << w << "does not exists!\n";
        return;
    }
}

```

```

    }

    data.adjacencyMatrix[start->index][end->index] = false;

    for (auto edge = data.edges.begin(); edge != data.edges.end(); ++edge) {
        if (edge->start == *start && edge->end == *end) {
            data.edges.erase(edge);
            break;
        }
    }
}

void EDIT_V(const std::string& name, const bool& mark) {
    Vertex* vertex = findVertexByName(name);

    if (!isVertexExists(vertex)) {
        std::cerr << "Can't edit a vertex " << name << " : does not
exists!\n";
        return;
    }

    vertex->visited = mark;
}

void EDIT_E(const std::string& v, const std::string& w, const float& c) {
    Vertex *start = findVertexByName(v), *end = findVertexByName(w);

    if (!isVertexExists(start)) {
        std::cerr << "Vertex " << v << " does not exist!\n";
        return;
    }

    if (!isVertexExists(end)) {
        std::cerr << "Vertex " << w << " does not exist!\n";
    }
}

```

```

        return;
    }

    if (!data.adjacencyMatrix[start->index][end->index]) {
        std::cerr << "Can't edit an edge : does not exists!\n";
        return;
    }

    for (auto edge = data.edges.begin(); edge != data.edges.end(); ++edge) {
        if (edge->start == *start && edge->end == *end) {
            edge->cost = c;
        }
    }
}
}

```

```

size_t FIRST(const Vertex& v) {
    if (!isVertexExists(v)) {
        std::cerr << "A vertex does not exists!\n";
        return data.vertices.size();
    }

    for (size_t i = 0; i != data.adjacencyMatrix.size(); ++i) {
        if (data.adjacencyMatrix[v.index][i]) {
            for (size_t j = 0; j != data.vertices.size(); ++j) {
                if (data.vertices[j].index == i) {
                    return j;
                }
            }
        }
    }
}
}

```



```

        return data.vertices.size();
    }

    size_t NEXT(const Vertex& v, const size_t& i) {
        if (!isVertexExists(v)) {
            std::cerr << "A vertex does not exists!\n";
            return data.vertices.size();
        }

        for (size_t j = i + 1; j < data.adjacencyMatrix[v.index].size(); ++j) {
            if (data.adjacencyMatrix[v.index][j]) {
                return j;
            }
        }

        return data.vertices.size();
    }

    Vertex VERTEX(const Vertex& v, const size_t& i) {
        if (!isVertexExists(v)) {
            std::cerr << "A vertex does not exists!\n";
            return Vertex();
        }

        for (size_t j = 0, adjacentIndex = 0; j !=
data.adjacencyMatrix[v.index].size(); ++j) {
            if (data.adjacencyMatrix[v.index][j]) {
                if (adjacentIndex == i) {
                    return data.vertices[j];
                }

                ++adjacentIndex;
            }
        }
    }

```

```

    }

    return Vertex();
}

void showVertices() const {
    std::cout << "\nVertices\n";

    for (auto it = data.vertices.begin(); it != data.vertices.end(); ++it) {
        std::cout << it->name << ' ';
    }

    std::cout << std::endl;
}

void showAdjacencyMatrix() const {
    std::cout << "\nAdjacency Matrix:\n";

    size_t columnWidth = 0;

    for (const auto& vertex : data.vertices) {
        columnWidth = std::max(columnWidth, vertex.name.size());
    }

    columnWidth = std::max(columnWidth, size_t(2));

    std::cout << std::setw(columnWidth + 4) << " ";
    for (const auto& vertex : data.vertices) {
        std::cout << std::setw(columnWidth + 2) << vertex.name;
    }

    std::cout << std::endl;

    for (size_t i = 0; i < data.adjacencyMatrix.size(); ++i) {

```

```

        std::cout << std::setw(columnWidth + 2) << data.vertices[i].name << "
|";

        for (size_t j = 0; j < data.adjacencyMatrix[i].size(); ++j) {
            std::cout << std::setw(columnWidth + 2) <<
data.adjacencyMatrix[i][j];
        }

        std::cout << std::endl;
    }
}

```

```

void runSolving() {
    if (data.vertices.empty()) {
        std::cerr << "Have no vertices!\n";
        return;
    }

    if (data.edges.empty()) {
        std::cerr << "Have no edges!\n";
        return;
    }

    std::vector<std::vector<std::string>> allPaths;

    for (size_t start = 0; start != data.vertices.size(); ++start) {
        std::vector<std::string> path;
        findAllPaths(start, path, allPaths);
    }

    std::cout << "\n\n\t\tANSWER:\n";

    for (const auto& path : allPaths) {

```

```

        for (const auto& vertex : path) {
            std::cout << vertex << ' ';
        }
        std::cout << std::endl;
    }
}

};

void run(Graph& graph, const int& countVertices, const std::string& allEdges) {
    for (int i = 1; i <= countVertices; ++i) {
        graph.ADD_V(std::to_string(i), false);
    }

    std::istringstream iss(allEdges);
    std::string start, end;

    while (iss >> start >> end) {
        graph.ADD_E(start, end);
    }

    graph.showVertices();
    graph.showAdjacencyMatrix();
    graph.runSolving();
}

int main() {
    Graph graph1;
    const std::string edges1 = R"(
        1 3
        1 5
        1 6
        2 5
    )";

```

```
    2 6
    3 4
    3 5
    5 6
);
```

```
Graph graph2;
```

```
const std::string edges2 = R"(
```

```
    1 3
    1 5
    1 6
    2 5
    2 6
    3 4
    3 5
    5 6
    4 18
    18 20
    20 8
    8 11
    11 7
    7 16
    8 7
    7 2
    2 9
    9 16
    15 10
    10 12
    12 1
    1 13
```

```
13 17
14 19
17 3
1 19
15 6
12 4
20 17

)";

run(graph1, 6, edges1);
run(graph2, 20, edges2);

return 0;
}
```

7. Литература.

- 1) Тюкачев Н. А., Хлебостроев В. Г. - С#. 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.