

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

**ОТЧЕТ**  
**по лабораторной работе №3**  
**по дисциплине «Построение и анализ алгоритмов»**  
**Тема: Потоки в сети**

Студент гр. 8304

\_\_\_\_\_

Мешков М.А.

Преподаватель

\_\_\_\_\_

Размочаева Н.В.

Санкт-Петербург

2020

### **Цель работы.**

Научиться применять алгоритм Форда-Фалкерсона и оценивать его сложность.

### **Постановка задачи.**

Вариант 3. Способ поиска пути: поиск в глубину, рекурсивная реализация.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

$N$  - количество ориентированных рёбер графа

$v_0$  - исток

$v_n$  - сток

$v_i v_j w_{ij}$  - ребро графа

$v_i v_j w_{ij}$  - ребро графа

...

Выходные данные:

$P_{\max}$  - величина максимального потока

$v_i v_j w_{ij}$  - ребро графа с фактической величиной протекающего потока

$v_i v_j w_{ij}$  - ребро графа с фактической величиной протекающего потока

...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

### **Описание алгоритма.**

В начале алгоритма поток через все ребра равен нулю, для каждого ребра добавляется соответствующее ребро в обратную сторону.

На каждой итерации алгоритма происходит поиск пути из истока в сток (в данном варианте с помощью рекурсивного поиска в глубину) такого чтобы ни у одного ребра на этом пути оставшаяся пропускная способность не была равна нулю, затем на данном пути находится ребро с минимальной оставшейся пропускной способностью, найденная величина добавляется к величине потока текущего ребра, вычитается из величины оставшейся пропускной способности текущего ребра, вычитается из величины потока соответствующего ребра в обратную сторону, прибавляется к величине оставшейся пропускной способности соответствующего ребра в обратную сторону.

Алгоритм заканчивает свою работу когда он не может больше найти путь.

Искомая величина максимального потока находится в процессе исполнения алгоритма — на каждой итерации искомая величина увеличивается (в самом начале равна нулю) на величину потока рассматриваемого пути.

### **Оценка сложности алгоритма.**

На каждом шаге алгоритм добавляет поток пути к уже имеющемуся потоку. Если пропускные способности всех рёбер — целые числа, то и потоки через все рёбра всегда будут целыми. Следовательно, на каждом шаге алгоритм увеличивает поток по крайней мере на единицу, следовательно, он сойдётся не более чем за  $O(f)$  шагов, где  $f$  — максимальный поток в графе. Можно выполнить каждый шаг за время  $O(E)$ , где  $E$  — число рёбер в графе, тогда общее время работы алгоритма ограничено  $O(Ef)$ .

### **Описание функций и структур данных.**

Для решения задачи были реализованы:

Структура Edge для хранения ребра графа.

Структура Vertex для хранения вершины графа.

Псевдоним Path, представляющий собой массив ребер, для хранения пути.

Структура Graph для хранения графа.

Функция findPath для поиска пути в графе в соответствии с вариантом.

Функция findMinCapacity для нахождения пропускной способности пути.

Функция changeFlow для модификации потока пути.

Функция addReverseEdges для добавления ребер в обратную сторону.

Функция printGraph для печати графа пользователю.

Функция printPath для печати пути пользователю.

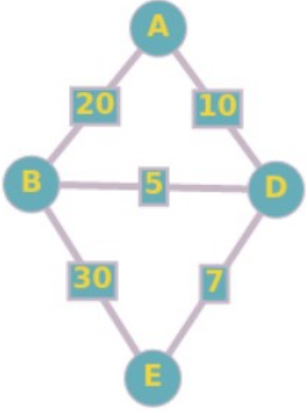
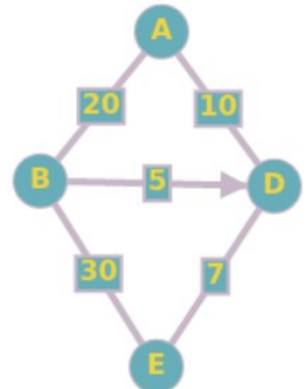
Функция readGraph для получения графа от пользователя.

Функция findMaxFlow для запуска алгоритма Форда-Фалкерсона.

Функция main принимает ввод и выводит ответ. Для подробного вывода процесса работы алгоритма при запуске нужно передать опцию -v.

### Тестирование.

Граф с пропускными способностями ребер	Ввод	Вывод
	<pre> 7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2 </pre>	<pre> 12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2 </pre>

 <pre> graph TD     A((A)) -- 20 --&gt; B((B))     A((A)) -- 10 --&gt; D((D))     B((B)) -- 5 --&gt; D((D))     B((B)) -- 30 --&gt; E((E))     D((D)) -- 7 --&gt; E((E)) </pre>	10 a e a b 20 b a 20 a d 10 d a 10 b e 30 e b 30 d e 7 e d 7 b d 5 d b 5	30 a b 20 a d 10 b a 0 b d 0 b e 23 d a 0 d b 3 d e 7 e b 0 e d 0
 <pre> graph TD     A((A)) -- 20 --&gt; B((B))     A((A)) -- 10 --&gt; D((D))     B((B)) -- 10 --&gt; D((D))     B((B)) -- 30 --&gt; E((E))     D((D)) -- 7 --&gt; E((E)) </pre>	9 a e a b 20 b a 20 a d 10 d a 10 b e 30 e b 30 d e 7 e d 7 b d 5	27 a b 20 a d 7 b a 0 b d 0 b e 20 d a 0 d e 7 e b 0 e d 0

### Выводы.

В ходе работы был реализован алгоритм Форда-Фалкерсона, программа была протестирована, была оценена сложность алгоритма.

## ПРИЛОЖЕНИЕ А.

### ИСХОДНЫЙ КОД

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <cassert>

bool debug = false;

struct Vertex;
struct Edge {
    Vertex *source = nullptr, *target = nullptr;
    int capacity = 0;
    int flow = 0;
    bool isReverseEdge = false;
    Edge *reverseEdge = nullptr;
};
struct EdgePtrComparator {
    bool operator()(const Edge *a, const Edge *b) const;
};
struct Vertex {
    char name = '?';
    bool inPath = false;
    std::set<Edge*, EdgePtrComparator> edges;
};
bool EdgePtrComparator::operator()(const Edge *a, const Edge *b) const {
    if (a->source->name != b->source->name)
        return a->source->name < b->source->name;
    if (a->target->name != b->target->name)
        return a->target->name < b->target->name;
    return a < b;
}
using Path = std::vector<Edge*>;
struct VertexPtrComparator {
    bool operator()(const Vertex *a, const Vertex *b) const {
        return a->name < b->name;
    }
};
struct Graph {
    Vertex *source = nullptr, *target = nullptr;
    std::set<Vertex*, VertexPtrComparator> vertices;
};

bool findPath(Vertex *source, Vertex *target, Path &path, bool clearPath
= true) {
    if (clearPath) {
```

```

        path.clear();
        source->inPath = true;
    }
    for (auto edgeItr = source->edges.rbegin(); edgeItr != source-
>edges.rend(); edgeItr++) {
        auto edge = *edgeItr;
        if (!edge->target->inPath && edge->capacity > 0) {
            path.push_back(edge);
            if (edge->target == target)
                return true;
            edge->target->inPath = true;
            bool pathIsFound = findPath(edge->target, target, path,
false);
            edge->target->inPath = false;
            if (pathIsFound) {
                source->inPath = false;
                return true;
            }
            path.pop_back();
        }
    }
    source->inPath = false;
    return false;
}

int findMinCapacity(const Path &path) {
    assert(!path.empty());
    int minCapacity = path.front()->capacity;
    for (auto &edge: path)
        minCapacity = std::min(minCapacity, edge->capacity);
    return minCapacity;
}

void changeFlow(const Path &path, int flowChange) {
    assert(flowChange >= 0);
    for (auto &edge: path) {
        edge->flow += flowChange;
        assert(edge->capacity >= flowChange);
        edge->capacity -= flowChange;

        edge->reverseEdge->flow -= flowChange;
        edge->reverseEdge->capacity += flowChange;
    }
}

void addReverseEdges(Graph &graph) {
    for (auto &vertex : graph.vertices) {
        for (auto &edge : vertex->edges) {

```

```

        assert(!(edge->isReverseEdge && edge->reverseEdge ==
nullptr));
        if (edge->reverseEdge != nullptr)
            continue;

        auto reverseEdge = new Edge;
        reverseEdge->reverseEdge = edge;
        reverseEdge->isReverseEdge = true;
        reverseEdge->target = edge->source;
        reverseEdge->source = edge->target;

        edge->reverseEdge = reverseEdge;
        edge->target->edges.insert(reverseEdge);
    }
}

void printGraph(const Graph &graph, bool printReverseEdges = true, bool
printProperties = true) {
    for (auto &vertex : graph.vertices) {
        for (auto &edge : vertex->edges) {
            if (edge->isReverseEdge && !printReverseEdges)
                continue;
            std::cout << vertex->name << " " << edge->target->name;
            if (printProperties) {
                std::cout
                    << " {capacity: " << edge->capacity
                    << ", flow: " << edge->flow
                    << ", isReverseEdge: " << (edge->isReverseEdge ?
"true" : "false")
                    << "}";
            }
            else {
                std::cout << " " << edge->flow;
            }
            std::cout << std::endl;
        }
    }
}

void printPath(const Path &path) {
    for (auto &edge : path) {
        std::cout << edge->source->name << " ";
    }
    assert(!path.empty());
    std::cout << path.back()->target->name << std::endl;
}

```



```

int findMaxFlow(Graph &graph) {
    if (debug) std::cout << "Adding reverse edges:" << std::endl;
    addReverseEdges(graph);
    if (debug) printGraph(graph);

    int maxFlow = 0;
    Path path;

    if (debug) std::cout << "Searching a path." << std::endl;
    while (findPath(graph.source, graph.target, path)) {
        if (debug) {
            std::cout << "Path is found: " << std::endl;
            printPath(path);
        }

        int minCapacity = findMinCapacity(path);
        if (debug) std::cout << "Min capacity = " << minCapacity <<
std::endl;

        if (debug) std::cout << "Changing the flow through the path." <<
std::endl;
        changeFlow(path, minCapacity);
        if (debug) {
            std::cout << "Modified graph:" << std::endl;
            printGraph(graph);
        }

        maxFlow += minCapacity;
        if (debug) std::cout << "Flow value = " << maxFlow << std::endl;
    }
    if (debug) std::cout << "Path is not found - the algorithm is
complete." << std::endl;
    return maxFlow;
}

Graph readGraph() {
    int vertexNumber;
    std::cin >> vertexNumber;

    char sourceName, targetName;
    std::cin >> sourceName >> targetName;

    std::map<char, Vertex *> vertices;
    for (int i = 0; i < vertexNumber; i++) {
        char edgeSourceName, edgeTargetName;
        int capacity;
        std::cin >> edgeSourceName >> edgeTargetName >> capacity;
    }
}

```

```

        auto &edgeSource = vertices[edgeSourceName];
        if (edgeSource == nullptr)
            edgeSource = new Vertex{edgeSourceName};
        auto &edgeTarget = vertices[edgeTargetName];
        if (edgeTarget == nullptr)
            edgeTarget = new Vertex{edgeTargetName};

        auto edge = new Edge;
        edge->source = edgeSource;
        edge->target = edgeTarget;
        edge->capacity = capacity;
        edgeSource->edges.insert(edge);
    }

    Graph graph;
    graph.source = vertices.at(sourceName);
    graph.target = vertices.at(targetName);
    for (auto &vertexWithName : vertices) {
        Vertex *vertex = vertexWithName.second;
        graph.vertices.insert(vertex);
    }

    return graph;
}

int main(int ac, char** av) {
    if (ac > 1)
        debug = true;

    auto graph = readGraph();

    int maxFlow = findMaxFlow(graph);

    std::cout << maxFlow << std::endl;
    printGraph(graph, false, false);

    return 0;
}

```