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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

ТЕМА: Потоки в сети

Вариант 2

Студентка гр. 8303 _____ Потураева М.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение алгоритма Форда-Фалкерсона для поиска максимального потока в сети.

Задание.

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).

Пример входных данных.

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Пример выходных данных.

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Индивидуализация.

Поиск в ширину. Обработка совокупности вершин текущего фронта как единого целого, дуги выбираются в порядке уменьшения остаточных пропускных способностей. При равенстве остаточных пропускных способностей выбирается та дуга, из начала которой выходит меньше дуг, при этом учитываются только дуги с положительными остаточными пропускными способностями, не ведущие в вершины текущего или прошлых фронтов.

Описание алгоритма.

В начале работы алгоритма создается граф. С помощью поиска в ширину производится поиск пути от истока в сток с применением индивидуализации. Если такой путь существует, то мы в этом пути ищем дугу с наименьшей пропускной способностью, затем обновляем граф: для всех дуг, которые попали в путь от истока в сток и для обратных им пересчитывается их пропускная способность.

Затем производиться подсчет максимального потока, к этому шагу сразу переходим, если путь не был найден. Выводятся фактические потоки для дуг.

Сложность алгоритма по операциям: O (E * F), где E – число ребер в графе, F – максимальный поток, при нецелых числах алгоритм может выполняться бесконечно.

Сложность алгоритма по памяти: O (N + E), где N – количество вершин, E – количество ребер.

Описание основных функций и структур.

class Edge

Класс для хранения ребра. Ребро задается начальной (char start) и конечной (char end) вершинами и весом (int weight).

class Graph

Класс для хранения графа. Данные хранятся в виде списка смежности в поле data. Также в классе реализованы следующие функции для работы с графом:

bool isEdge(char u, char v) — проверяет существует ли ребро

void addEdge(char u, char v, int w) — добавленяет ребра в граф

std::vector getEdges(char u) — возвращает все ребра исходящие из вершины u

int getValue(char u, char v) — возарвщает вес ребра

void addValue(char u, char v, int value) — прибавляет значение к весу ребра

void print() - выводит описание ребер графа в консоль

bool cmp(Edge a, Edge b)-компаратор для выбора ребра по правилу

индивидуализации

void read_graph(Graph* graph, std::set<std::pair>& edges, int n)

Чтение графа, для каждого ребра, если отсутствует обратное к нему ребро, то оно добавляется с весом 0.

bool bfs(Graph* graph, int start, int finish, std::map& parents)

Поиск пути с помощью поиска в ширину. Функция возвращает true, если путь найден, false иначе. Путь запоминается в переменной parents.

int minWeightOnCurrentPath(Graph* graph, std::map& parents, char finish)

Функция находит минимальное ребро на найденном пути и возвращает его вес.

void changeWeights(Graph* graph, std::map& parents, char finish, int flow)

Функция вычитает из весов на пути из истока в сток значение flow. При этом прибавляет flow к значениям ребер, обратным к тем, которые лежат на пути.

void fordFulkerson(Graph* graph, int& maxFlow, char start, char finish)

Функция которая которая запускает две предыдущие функции пока путь существует и прибавляет к максимальному потоку значение, минимального ребра на текущем пути.

void writeAnswer(std::set<std::pair>& edges, int maxFlow)

Вывод ответа в консоль.

ТЕСТИРОВАНИЕ.

№	Input	Output
1	7 a f a b 7 a c 6 b d 6 c f 9 d e 3	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2

	d f 4 e c 2	
2	10 a h a b 5 a c 4 a d 1 b g 1 c e 2 c f 3 d e 6 e h 4 f h 4 g h 8	6 a b 1 a c 4 a d 1 b g 1 c e 2 c f 2 d e 1 e h 3 f h 2 g h 1
3	5 a d a b 1000 a c 1000 b c 1 b d 1000 c d 1000	2000 a b 1000 a c 1000 b c 0 b d 1000 c d 1000
4	5 a d a c 1 a b 1 b d 1 c d 1 b c 1	2 a b 1 a c 1 b c 0 b d 1 c d 1
5	14 a f	12 a b 6 a c 6

	a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2 b a 7 c a 6 d b 6 f c 9 e d 3 f d 4 c e 2	b a 0 b d 6 c a 0 c e 0 c f 8 d b 0 d e 2 d f 4 e c 2 e d 0 f c 0 f d 0

Выводы.

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Форда - Фалкерсона, который находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро.

Приложения А. Исходный код.

Lab3.cpp

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

class Edge {
public:
    char start;//исток
    char end;//сток
    int weight;//вес

};

class Graph {
public:
    bool isEdge(char u, char v) {//существует ли ребро
        return this->data[u].count(v);
    }

    void addEdge(char u, char v, int w) {//добавляем ребро
        this->data[u][v] = w;
    }

    std::vector<Edge> getEdges(char u) {//получить ребра выходящие из вершины
        std::vector<Edge> edgesFromU;
        for (auto nextEdge : this->data[u]) {
            edgesFromU.push_back(Edge({ u, nextEdge.first, nextEdge.second }));
        }
        return edgesFromU;
    }


    int getValue(char u, char v) {//вес ребра
        return this->data[u][v];
    }

    void addValue(char u, char v, int value) {//добавить вес
        this->data[u][v] += value;
    }

    void print() {//выводит ребра графа в консоль
        for (auto& in : data) {
            for (auto& to : in.second) {
                std::cout << in.first << ' ' << to.first << ' ' << to.second << '\n';
            }
        }
    }

    bool cmp(Edge a, Edge b) {
        if (a.weight < b.weight) return true;
        int count1 = this->getEdges(a.start).size();
        int count2 = this->getEdges(b.start).size();
        if (a.weight == b.weight && count1 < count2) return true;
        return false;
    }
private:
    std::map<char, std::map<char, int>> data;//список смежности для графа
```

```
};



void read_graph(Graph* graph, std::set<std::pair<char, char>>& edges, int n) {//считываем
граф
    char u, v;
    int w;
    for (int i = 0; i < n; i++) {
        std::cin >> u >> v >> w;
        edges.insert({ u,v });
        graph->addEdge(u, v, w); // прямое ребро
        if (!graph->isEdge(v, u)) {
            graph->addEdge(v, u, 0); // обратное
        }
    }
}

bool bfs(Graph* graph, int start, int finish, std::map<char, char>& parents) {//поиск в ши-
рину
    std::queue<char> q;//очередь вершин
    q.push(start);
    parents[start] = start;//путь
    char v;
    auto cmp{
        [&graph](Edge a, Edge b) {
        return graph->cmp(a,b); }
    };
    while (!q.empty()) {//пока очередь не пуста
        v = q.front();//берем первую вершину
        q.pop();//убираем из очереди
        std::vector<Edge> vec = graph->getEdges(v);
        std::sort(vec.begin(), vec.end(), cmp);
        for (Edge edge : graph->getEdges(v)) {//проходим по всем ребрам вершины
            char to = edge.end;// вершина обрабатываемая
            int w = edge.weight;//вес
            if (parents.count(to) == 0 && w > 0) {//если в пути еще нет этой вершины и вес
больше 0
                q.push(to);//помещаем в очередь
                parents[to] = v;//увеличиваем путь
                if (to == finish) break;//если дошли до стока
            }
        }
        if (parents.count(finish)) break;//если дошли до стока
    }
    return !q.empty();
}

int minWeightOnCurrentPath(Graph* graph, std::map<char, char>& parents, char finish) {//ищем
мин пропускную способность в увеличивающем пути
    char prevVert = finish;//начинаем со стока
    char curVert = parents[prevVert];//предок стока
    int weight = graph->getValue(curVert, prevVert);//вес ребра
    std::vector<char> path;//путь
    path.push_back(finish);
    while (prevVert != curVert) {//пока не дошли до истока
        path.push_back(curVert);
        weight = std::min(weight, graph->getValue(curVert, prevVert));//выбираем минимум
        curVert = parents[curVert];//переходим к след вершине
        prevVert = parents[prevVert];
    }
    std::cout << "Found new available path\n";
    for (auto i = path.rbegin(); i != path.rend(); i++) {//увеличивающий путь
```

```
        std::cout << (*i) << ' ';
    }
    std::cout << '\n' << "minimum weight on the path is " << weight << "\n";//мин элемент
    return weight;
}

void changeWeights(Graph* graph, std::map<char, char>& parents, char finish,
    int flow, std::map<std::pair<char, char>, int>& answer) {//изменеие весов в графе
    char prevVert = finish;//начианем со стока
    char curVert = parents[prevVert];
    while (prevVert != curVert) {//пока не дошли до итсока
        graph->addValue(curVert, prevVert, -flow);//отнимаем поток в сторону стока
        answer[{curVert, prevVert}] += flow;//прибавляем к  ответу
        graph->addValue(prevVert, curVert, flow);//прибавляем поток в сторону истока
        curVert = parents[curVert];//след вершина
        prevVert = parents[prevVert];
    }
    graph->print();//печать графа
    std::cout << '\n';
}

void fordFulkerson(Graph* graph, int& maxFlow, char start, char finish,
std::map<std::pair<char, char>, int>& answer) {//запуск алгоритма
    bool isWayExist = true;//есть ли путь
    while (isWayExist) {//пока есть путь
        std::map<char, char> parents;//путь
        isWayExist = bfs(graph, start, finish, parents);//ищем путь
        if (isWayExist) {//если нашли
            int flow = minWeightOnCurrentPath(graph, parents, finish);//ищем мин поток
            maxFlow += flow;//прибавляем к макс потоку
            changeWeights(graph, parents, finish, flow, answer);//пересчет пропускных спо-
собностей
        }
    }
}

void writeAnswer(std::set<std::pair<char, char>>& edges, std::map<std::pair<char, char>,
int> answer, int maxFlow) {//вывод ответа
    std::cout << maxFlow << '\n';
    for (auto edge : edges) {
        char u = edge.first;
        char v = edge.second;
        std::cout << u << ' ' << v << ' ';
        if (edges.count({ v,u })) {
            std::cout << answer[{u, v}] << '\n';
        }
        else {
            std::cout << answer[{u, v}] - answer[{v, u}] << '\n';
        }
    }
}

int main() {
    // ввод данных
    int n;
    std::cin >> n;//количество вершин
    char start, finish;//исток и сток
    std::cin >> start >> finish;

    auto* graph = new Graph;
    std::set<std::pair<char, char>> edges;//ребра графа
    read_graph(graph, edges, n);//считывание графа

    int maxFlow = 0;
```

```
    std::map<std::pair<char, char>, int> answer;//ответ
    fordFulkerson(graph, maxFlow, start, finish, answer);//запуск алгоритма

    writeAnswer(edges, answer, maxFlow);//вывод ответа

    delete graph;
}
```