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 и сети

Студент гр. 8383 _____ Мирсков А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучить принцип работы алгоритма Форда-Фалкерсона для нахождения максимального потока в сети. Решить с его помощью задачу.

Постановка задачи

Вар. 1. Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

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
v0 - исток
vn - сток
vivjωij - ребро графа
vivjωij - ребро графа
...

Выходные данные:
Pmax - величина максимального потока
vivjωij - ребро графа с фактической величиной протекающего потока
vivjωij - 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).

Sample Input:

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

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Реализация

Для каждого ребра заводится обратное ему с нулевым весом. Сначала ищется путь, проходящий только по ребрам с ненулевым весом, от истока к стоку с помощью поиска в ширину. Далее находится максимальный поток, который можно пустить по этому пути и из каждого ребра на пути вычитается величина этого потока, а к обратным ребрам прибавляется. Процесс останавливается, когда невозможно найти путь от истока к стоку.

Сложность по времени — O(|E|f), где E — число рёбер в графе, f - максимальный поток в графе, так как каждый увеличивающий путь может быть найден за O(|E|f) и увеличивает поток как минимум на 1.

Сложность по памяти — O(|E|), т. к. программа хранит вес для каждого ребра и обратного ему.

Описание функций и структур данных

class Edge

Класс для хранения ребра. Ребро задается начальной (char start) и конечной (char finish) вершинами и весом (int weight). Также в классе определен оператор < для упорядочения по алфавиту.

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<Edge> 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

void read_graph(Graph* graph, std::set<std::pair<char,char>>& 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<char, char>& 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<char, char>& parents, char finish)
Функция находит минимальное ребро на найденном пути и возвращает его вес.

void changeWeights(Graph* graph, std::map<char, char>& 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<char, char>>& edges, int maxFlow)
Вывод ответа в консоль

Тестирование

Тесты находятся в папке Tests.

Для тестирования был написан скрипт на python3, который запускает все тесты из папки вместе.

1	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
2	3	2

	a d a b 4 b c 2 c d 3	a b 2 b c 2 c d 2
3	5 a d a b 2 a c 2 a d 4 b c 1 c d 3	7 a b 1 a c 2 a d 4 b c 1 c d 3
4	6 a e a b 1 a c 2 c b 2 c d 3 d e 3 b e 3	3 a b 1 a c 2 b e 3 c b 2 c d 0 d e 0

Полный вывод программы для теста 3.

```
andrei@andrei-Inspiron-3558:~/shit_code/algosgit/lab3$ ./lab3
5
a
d
a b 2
a c 2
a d 4
b c 1
c d 3
Found new available path
a d
minimum weight on the path is 4
a b 2
a c 2
a d 0
b a 0
b c 1
c a 0
c b 0
c d 3
d a 4
d c 0

Found new available path
a c d
minimum weight on the path is 2
a b 2
a c 0
a d 0
b a 0
b c 1
c a 2
c b 0
c d 1
d a 4
d c 2
```

```
Found new available path
a b c d
minimum weight on the path is 1
a b 1
a c 0
a d 0
b a 1
b c 0
c a 2
c b 1
c d 0
d a 4
d c 3

7
a b 1
a c 2
a d 4
b c 1
c d 3
```

Выводы

В ходе лабораторной работы был изучен алгоритм Форда-Фалкерсона для нахождения максимального потока в сети. Для этого была написана программа, решающая задачу поиска максимального потока в произвольной сети.

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

class Edge {
public:
    char start;
    char end;
    int weight;
    bool operator<(Edge edge) const {
        if (this->start < edge.start) return true;
        if (this->start == edge.start && this->end < edge.end) return true;
           return this->start == edge.start && this->end == edge.end && this-
>weight < edge.weight;
    }
};

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

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

    std::vector<Edge> getEdges(char u) {
        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) {
        return this->data[u][v];
    }

    void addValue(char u, char v, int value) {
```

```
        this->data[u][v] += value;
    }

    void print() {
        for (auto& in: data) {
            for (auto& to: in.second) {
                std::cout << in.first << ' ' << to.first << ' ' << to.second <<
'\n';
            }
        }
    }
private:
    std::map<char, std::map<char, int>> data;
};

void read_graph(Graph* graph, std::set<std::pair<char,char>>& edges, int n) {
    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) {
    std::queue<char> q;
    q.push(start);
    parents[start] = start;
    char v;
    while (!q.empty()) {
        v = q.front();
        q.pop();
        for (Edge edge:graph->getEdges(v)) {
            char to = edge.end;
            int w = edge.weight;
            if (parents.count(to) == 0 && w > 0) {
                q.push(to);
                parents[to] = v;
```

```
                if (to == finish) break;
            }
        }
        if (parents.count(finish)) break;
    }
    return !q.empty();
}

int minWeightOnCurrentPath(Graph* graph, std::map<char, char>& parents, char
finish) {
    char prevVert = finish;
    char curVert = parents[prevVert];
    int weight = graph->getValue(curVert, prevVert);
    std::vector<char> path;
    path.push_back(finish);
    while (prevVert != curVert) {
        path.push_back(curVert);
        weight = std::min(weight, graph->getValue(curVert, prevVert));
        curVert = parents[curVert];
        prevVert = parents[prevVert];
    }
    std::cout << "Found new available path\n";
    for (auto i = path.rbegin(); i != path.rend(); i++) {
        std::cout << (*i) << ' ';
    }
    std::cout << '\n' << "minimum weight on the path is " << weight << "\n";
    return weight;
}

void changeWeights(Graph* graph, std::map<char, char>& parents, char finish,
 int flow, std::map<std::pair<char,char>,int>& answer) {
    char prevVert = finish;
    char curVert = parents[prevVert];
    while (prevVert != curVert) {
        graph->addValue(curVert, prevVert, -flow);
        answer[{curVert, prevVert}] += flow;
        graph->addValue(prevVert, curVert, flow);
        curVert = parents[curVert];
        prevVert = parents[prevVert];
    }
    graph->print();
    std::cout << '\n';
```

```
}

void fordFulkerson(Graph* graph, int& maxFlow, char start, char finish,
std::map<std::pair<char, char>, int>& answer) {
    bool isWayExist = true;
    while (isWayExist) {
        std::map<char, char> parents;
        isWayExist = bfs(graph, start, finish, parents);
        if (isWayExist) {
            int flow = minWeightOnCurrentPath(graph, parents, finish);
            maxFlow += flow;
            changeWeights(graph, parents, finish, flow, answer);
        }
    }
}

void writeAnswer(std::set<std::pair<char, char>>& edges,
std::map<std::pair<char, char>, int> answer, int maxFlow) {
    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;
    char start, finish;
    std::cin >> start >> finish;

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

```
    int maxFlow = 0;
    std::map<std::pair<char,char>,int> answer;
    fordFulkerson(graph, maxFlow, start, finish, answer);

    writeAnswer(edges, answer, maxFlow);

    delete graph;
}
```