

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

Студент гр. 8304

Рыжиков А. В.

Преподаватель

Размочаева Н. В.

Санкт-Петербург

2020

Цель работы.

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

Вариант 3. Поиск в глубину. Рекурсивная реализация.

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

Описание алгоритма.

В начале работы алгоритму на вход подается граф для поиска максимального потока, вершина-исток и вершина-сток графа. После чего производится поиск в глубину в графе.

На каждом этапе поиска в глубину с помощью очереди находится путь от истока к стоку. Из ребер пути находится ребро с минимальным весом. Из

всех ребер пути от истока к стоку вычитается вес минимального ребра пути, а к ребрам пути от стока к истоку вес минимального ребра прибавляется (если такой вершины не существует, то она достраивается). К переменной, отвечающей за максимальный поток в графе, прибавляется вес минимального ребра пути.

Цикл поиска в глубину и изменения ребер графа осуществляется до тех пор, пока поиск в глубину возможен. Результатом является значение переменной, отвечающей за максимальный поток в графе. Фактический поток через ребра определяется как разность между первоначальным ребром и ребром, после преобразований.

В консоль выводится результат работы алгоритма и промежуточные результаты, такие как текущие вершины поиска в ширину и их соседи с расстоянием до них, найденный путь, преобразованный граф.

Сложность алгоритма по операциям: $O(E * F)$, E – число ребер в графе, F – максимальный поток

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

Описание функций и структур данных.

`class Path`

Структура данных, используемая для хранения путей направленного графа. Хранит имя вершины из которой идём в вершину в которую идём и пропускную способность пути.

```
bool findPath(std::vector<Path> *paths, std::vector<Path *> *local,
std::vector<Path *> *local2, char myPoint, char *endPoint)
```

Функция поиска в графе в глубину. На вход принимает указатель на вектор путей, указатель на вектор очереди, указатель на вектор посещенных путей, имя вершины которую нужно обработать и конечную вершину.

```
void findMin(std::vector<Path *> *local, int *maxFlow)
```

Функция возвращает находит минимальную разность между пропускной способностью и значением потока среди всех посещённых вершин.

```
bool isVisitedPath(std::vector<Path *> *local, char element)
```

Функция возвращает находит минимальную разность между пропускной способностью и значением потока среди всех посещённых вершин.

Тестирование.

Входные данные:

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

Результат работы программы:

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

Входные данные:

```
8
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

3
a b 0
a c 2
a d 1
b g 0
c e 2
c f 0
d e 1
e h 3

Выводы.

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

Приложение с кодом

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>

class Path {
public:
    Path(char nameFrom, char nameOut, int bandwidth) : nameFrom(nameFrom),
nameOut(nameOut), bandwidth(bandwidth) {}

    void setFlow(int flow) {
        Path::flow = flow;
    }

    char getNameFrom() const {
        return nameFrom;
    }

    char getNameOut() const {
        return nameOut;
    }

    int getBandwidth() const {
        return bandwidth;
    }

    int getFlow() const {
```

```

        return flow;
    }

private:
    char nameFrom;
    char nameOut;
    int bandwidth;
    int flow = 0;
};

void findMin(std::vector<Path *> *local, int *maxFlow) {
    int Min = local->front()->getBandwidth();
    for (Path *path : *local) {
        if (Min > (path->getBandwidth() - path->getFlow())) {
            Min = path->getBandwidth() - path->getFlow();
        }
    }

    for (Path *path : *local) {
        path->setFlow(path->getFlow() + Min);
    }

    *maxFlow = *maxFlow + Min;
}

bool comp(Path a, Path b) {
    if (a.getNameFrom() != b.getNameFrom()) {

```



```

        return a.getNameFrom() < b.getNameFrom();
    } else {
        return a.getNameOut() < b.getNameOut();
    }
}

bool comp2(Path *a, Path *b) {
    return (a->getBandwidth() - a->getFlow()) <= (b->getBandwidth() - b->getFlow());
}

bool isVisitedPath(std::vector<Path *> *local, char element){
    for (Path *path : *local){
        if (element == path->getNameFrom()) {
            return false;
        }
    }

    return true;
}

bool findPath(std::vector<Path> *paths, std::vector<Path *> *local,
std::vector<Path *> *local2, char myPoint, char *endPoint) {
    if (myPoint == *endPoint) {
        return true;
    }

    std::vector<Path *> localPaths;
    localPaths.reserve(0);

```

```

for (auto &path: *paths) {
    if (path.getNameFrom() == myPoint) {
        char sc = path.getNameFrom();
        localPaths.emplace_back(&path);
    }
}

int size = localPaths.size();

std::sort(localPaths.begin(), localPaths.end(), comp2);

/*for (Path *path : localPaths) {
    std::cout << path->getNameFrom() << " " << path->getNameOut() << " " <<
path->getBandwidth() << "\n";
}
std::cout << "_____ \n";*/

for (Path *path : localPaths) {
    if (path->getFlow() < path->getBandwidth()) {
        if (isVisitedPath(local2, path->getNameOut())) {
            local2->emplace_back(path);
            if (findPath(paths, local, local2, path->getNameOut(), endPoint)) {
                local->emplace_back(path);
                return true;
            } else{
                local2->pop_back();
            }
        }
    }
}

```

```

    }
}

return false;

}

int main() {

    char startPoint, endPoint;

    char start, end;
    int weight;
    int count = 16;

    std::vector<Path *> local;
    local.reserve(0);

    std::vector<Path *> local2;
    local.reserve(0);

    std::vector<Path> paths;
    paths.reserve(0);

    int maxFlow = 0;

    int your_choose = 0;

```

```

std::cout << "If you want to enter data from a file, enter \'1\'\\n";

std::cin >> your_choose;

std::ifstream fin;

if (your_choose == 1) {
    std::ifstream fin;
    fin.open("C:\\Users\\Alex\\Desktop\\in.txt");

    if (fin.is_open()) {
        std::cout << "Reading from file:" << "\\n";

        fin >> count;
        fin >> startPoint;
        fin >> endPoint;
        while (count != 0) {
            fin >> start >> end >> weight;
            paths.emplace_back(Path(start, end, weight));
            count--;
        }
    } else {
        std::cout << "File not opened";
    }

    fin.close();
} else {

```

```

std::cin >> count;
std::cin >> startPoint;
std::cin >> endPoint;
while (count != 0) {
    std::cin >> start >> end >> weight;
    paths.emplace_back(Path(start, end, weight));
    count--;
}

}

```

```

std::ofstream fin2;
fin2.open("C:\\Users\\Alex\\Desktop\\out.txt");

```

```

while (findPath(&paths, &local, &local2, startPoint, &endPoint)) {
    findMin(&local, &maxFlow);

    for (Path *path : local) {
        std::cout << path->getNameFrom() << " " << path->getNameOut() << " " <<
path->getFlow() << "\n";
    }
}

```

```

if (your_choose == 1) {
    for (Path *path : local) {
        fin2 << path->getNameFrom() << " " << path->getNameOut() << " " <<

```

```

path->getFlow() << "\n";
    }
    fin2 << " _____\n";
}
std::cout << " _____\n";
local.clear();
local2.clear();
}

```

```

std::cout << maxFlow << '\n';

```

```

std::sort(paths.begin(), paths.end(), comp);

```

```

for (Path path : paths) {
    std::cout << path.getNameFrom() << " " << path.getNameOut() << " " <<
path.getFlow() << "\n";
}

```

```

if (your_choose == 1){
    std::ofstream fin;
    fin.open("C:\\Users\\Alex\\Desktop\\out.txt");

```

```

    fin2 << maxFlow << '\n';
    for (Path path : paths) {
        fin2 << path.getNameFrom() << " " << path.getNameOut() << " " <<
path.getFlow() << "\n";

```

```
    }  
}  
  
return 0;  
}
```

Приложение 2 uml

