

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. 8303 _____ Гришин К. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы

Изучить алгоритм Форда-Фалкерсона для поиска максимального потока в сети, а также имплементировать его на языке C++.

Задание, вариант 5

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 имен вершин и целого неотрицательного числа — пропускной способности (*веса*)

Входные данные:

N — количество ориентированных ребер графа

v_0 — исток

v_n — сток

$v_i v_j \omega_{ij}$ — ребро графа

$v_i v_j \omega_{ij}$ — ребро графа

...

Выходные данные:

P_{max} — величина максимального потока

$v_i v_j \omega_{ij}$ — ребро графа с фактической величиной протекающего потока

$v_i v_j \omega_{ij}$ — ребро графа с фактической величиной протекающего потока

...

Sample input	Sample output
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

Индивидуализация

Во время поиска сквозного пути, каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность.

Описание алгоритма

1. Считывается матрица, которая хранится в памяти как матрица смежности
2. Создается копия графа — граф остаточных пропускных способностей.
3. В графе остаточных пропускных способностей ищется путь от истока к стоку, причем во время поиска пути, совершается переход к вершине с наибольшей остаточной пропускной способностью.
4. Если путь не найден, основной шаг алгоритма завершается, перейти к шагу 8.
5. Иначе находится поток пути, который равен наименьшей пропускной способности среди всех ребер в сквозном пути.
6. В графе остаточных пропускных способностей, остаточная пропускная способность по направлению сквозного пути уменьшается на поток этого пути, а против направления — увеличивается.
7. После пересчета графа остаточных пропускных способностей, вернуться к шагу 3.
8. **Максимальный поток** в графе находится как сумма потоков сквозных путей, найденных во время основного шага.
9. **Реальный поток ребра** в графе определяется как разность начальной пропускной способности и остаточной, если реальный поток получился отрицательным, то он равен нулю.
10. *Алгоритм завершает свою работу.*

Описание основного класса и его методов

Класс Graph:

char переопределяется как Vertex.

Содержит в себе матрицу смежности *map*<Vertex, std<Vertex, long int>> *adjMatrix*, а также матрицу смежности *flow* для хранения реальных потоков, после работы алгоритма.

Методы класса:

Конструктор(bool log_p) — получает на вход флаг, который говорит классу, нужна ли запись логов в файл рядом с программой, а также открывает сам файл.

Деструктор — закрывает файл логов.

void addPair(Vertex a, Vertex b, long int capacity) — получает на вход вершину a, вершину b, целое число capacity. В матрицу смежности записывается ребро между вершинами a и b с пропускной способностью capacity.

long int iFordFulkerson(Vertex src, Vertex dst) — на вход получает две вершины, исток и сток соответственно, возвращает целое число — величину максимального потока сети. Функция применяет алгоритм Форда-Фалкерсона к графу, реальный поток ребер при этом записывается в матрицу смежности flow, хранящуюся в классе.

const map<Vertex, map<Vertex, long int>> *getFlow()* *const* — функция дает доступ к матрице смежности, в которой хранится реальный поток каждого ребра.

Также используется утилитарная лямбда функция для поиска пути в графе остаточных пропускных способностей:

lambda map<Vertex, Vertex> [*residualFlow*](Vertex src, Vertex dst)*findWay* — получает на вход две вершины src и dst, возвращает ассоциативный массив, по которому можно восстановить путь от вершины src до вершины dst. Поиск осуществляется по принципу, выбора ребра с наибольшей остаточной пропускной способностью.

Сложность алгоритма

Алгоритм завершается тогда, когда на графе больше не может быть достигнуто новых путей от истока к стоку, но при этом алгоритм не гарантирует, что такая ситуация будет когда-либо достигнута, но такая ситуация может быть достигнута только при нерациональных значениях потока. Однако, когда пропускными способностями ребер являются целые числа, **сложность алгоритма** можно описать как $O(E \cdot f)$, где E — число ребер графа, а f — максимальный поток в графе. Это связано с тем, что каждый путь расширения максимального потока может быть найден за время $O(E)$, при этом каждый такой путь увеличивает максимальный поток не менее чем на 1.

Хранение решения

Граф представлен в виде матрицы смежности (*map<Vertex, map<Vertex, long int>>*), ячейки матрицы хранят веса дуг.

Использованные оптимизации

Во время поиска пути от источника к стоку, от текущей вершины выбирается соседняя с ребром, у которого наибольшая остаточная пропускная способность.

Тестирование

input	output	log of first input
7	12	Added edge with capacity: 7 between a and b
a	a b 6	Added edge with capacity: 6 between a and c
f	a c 6	Added edge with capacity: 6 between b and d
a b 7	b d 6	Added edge with capacity: 9 between c and f
a c 6	c f 8	Added edge with capacity: 3 between d and e
b d 6	d e 2	Added edge with capacity: 4 between d and f
c f 9	d f 4	Added edge with capacity: 2 between e and c
d e 3	e c 2	
d f 4		Ford-Fulkerson initiated
e c 2		iteration 1:

15 a k a b 5 a d 7 b c 2 b f 4 c d 1 d f 3 d g 4 e h 6 f e 3 f i 2 f j 6 g f 5 h i 7 i k 4 j k 5	9 a b 3 a d 6 b c 0 b f 3 c d 0 d f 2 d g 4 e h 3 f e 3 f i 1 f j 5 g f 4 h i 3 i k 4 j k 5	path: a->[7;0]<-b->[6;0]<-d->[4;0]<-f Path flow is 4 iteration 2: path: a->[6;0]<-c->[9;0]<-f Path flow is 6 iteration 3: path: a->[3;4]<-b->[2;4]<-d->[3;0]<-e->[2;0]<-c->[3;6]<-f Path flow is 2 Max Flow = 4 + 6 + 2 = 12
16 a k a b 5 a c 4 a d 5 b e 4 c e 6 c f 4 d f 1 d g 3 e h 1 e i 2 f i 5 f j 4 h k 2 i k 10 j k 9	8 a b 3 a c 4 a d 1 b e 3 c e 0 c f 4 d f 1 d g 0 e h 1 e i 2 f i 2 f j 3 h k 1 i k 4 j k 3	

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм поиска максимального потока в сети — метод Форда-Фалкерсона. Данный алгоритм был имплементирован на языке C++. Во время поиска сквозного пути в графе, от каждой вершины выбиралось ребро с наибольшей пропускной способностью.

ПРИЛОЖЕНИЕ А

Исходный код (FFA.cpp)

```
#include <iostream>    // console input-output
#include <ostream>
#include <fstream>     // file output

#include <limits>      // int maximum
#include <algorithm>   // sort

#include <vector>
#include <stack>
#include <map>

class Graph{
    // rename char as Vertex
    using Vertex = char;

    // adjacency matrix
    std::map<Vertex, std::map<Vertex, long int>> adjMatrix; // [from][to]

    // adjacency matrix of flow
    std::map<Vertex, std::map<Vertex, long int>> flow;

    // flag, which says "log process"
    bool logger;

    // log file descriptor
    std::ofstream log;

public:
    Graph(bool log_p){
        logger = log_p;
        // open file
        log.open("log", std::ios::out | std::ios::trunc);
    }

    ~Graph(){
        // close file
        log.close();
    }

    // write value to adjacency matrix
    void addPair(Vertex a, Vertex b, long int capacity){
        adjMatrix[a][b] = capacity;
        if(logger)
            log << "Added edge with capacity: " << capacity
                << " between " << a << " and " << b << std::endl;
    }

    // Ford Fulkerson algorithm
    long int iFordFulkerson(Vertex src, Vertex snk){
        if(logger)
            log << std::endl << "Ford-Fulkerson initiated" << std::endl;

        flow.clear();

        // adjacency matrix of residual flows
        auto residualFlow = adjMatrix;

        // max flow accumulator
        long int max_flow = 0;

        // lambda function of path finding
        // works as dfs but chose next vertex using residual flows
        auto findWay = [this, &residualFlow](Vertex src, Vertex dst){
            std::stack<Vertex> stack;
            std::map<Vertex, bool> visited;
            std::map<Vertex, Vertex> prev;

            stack.push(src);
            while(!stack.empty()){
                // u <- current vertex
                auto u = stack.top();
                if(u == dst) break;

                stack.pop();
                visited[u] = 1;

                // neighbours of current vertex
                std::vector<Vertex> neighbours;
                for(auto p: residualFlow[u]){
                    if(!visited[p.first] && p.second > 0)
                        neighbours.push_back(p.first);
                }

                // sort neighbours ascending residual flow
                std::sort(neighbours.begin(), neighbours.end(),
                    [u, &residualFlow](const Vertex &a, const Vertex &b)
                    {return residualFlow[u][a] < residualFlow[u][b];});

                // put neighbours on stack (with max residual flow on top)
                for(auto n: neighbours){
                    stack.push(n);
                    prev[n] = u;
                }
            }
            return prev;
        };
    }
};
```

```

};

int iteration = 1;

// get path from source to sink
auto prev = findWay(src, snk);
std::vector<long int> flows;

// if snk in prev map then there is a way
while(prev.count(snk) > 0){
    if(logger)
        log << "iteration " << iteration << ":" << std::endl;

    long int path_flow = std::numeric_limits<long int>::max();
    auto temp = snk;

    if(logger)
        log << "path: ";

    std::vector<Vertex> path;

    // find min residual flow using prev map
    while(temp != src){
        if(logger)
            path.insert(path.begin(), temp);

        path_flow = std::min(path_flow, residualFlow[prev[temp]][temp]);
        temp = prev[temp];
    }
    if(logger)
        path.insert(path.begin(), temp);

    if(logger){
        for(auto it = path.begin(); it != path.end(); it++){
            if(it != path.begin()){
                auto to = *it;
                auto from = prev[to];
                log << "->[" << residualFlow[from][to]
                    << ";<< residualFlow[to][from] << "]-<<";
            }
            log << *it;
        }
        log << std::endl;
        log << "Path flow is " << path_flow << std::endl;
        log << std::endl;
        flows.push_back(path_flow);
    }

    // add path flow to max flow
    max_flow += path_flow;
    temp = snk;
    while(temp != src){
        // reduce residual flow on the way
        auto from = prev[temp];
        residualFlow[from][temp] -= path_flow;
        residualFlow[temp][from] += path_flow;
        temp = prev[temp];
    }

    // recalculate path from source to sink
    prev = findWay(src, snk);
    iteration++;
}

if(logger){
    log << "Max Flow = ";
    for(auto it = flows.begin(); it != flows.end(); it++){
        if(it != flows.begin())
            log << " + ";
        log << *it;
    }
    log << " = " << max_flow << std::endl;
    log << std::endl;
}

for(auto from_p: adjMatrix)
    for(auto to_p: from_p.second){
        auto iFlow = to_p.second - residualFlow[from_p.first][to_p.first];
        flow[from_p.first][to_p.first] = (iFlow < 0) ? 0 : iFlow;
    }

return max_flow;
}

// get adjacency matrix of flows
const auto& getFlow() const{
    return flow;
}

};

int main(){
    Graph graph(1);

    // get number of edges
    int N;
    std::cin >> N;

    // get src and dst of path
    char src, snk;
    std::cin >> src >> snk;

```



```

// get graph (two points and length between)
char from, to;
long int len;
for(int i = 0; i < N; i++){
    std::cin >> from >> to >> len;
    graph.addPair(from, to, len);
}

// get max flow
auto max_flow = graph.iFordFulkerson(src, snk);
std::cout << max_flow << std::endl;

// get flows of graph and print them
for(auto from_p: graph.getFlow()){
    for(auto to_p: from_p.second){
        std::cout << from_p.first << " "
                    << to_p.first << " "
                    << to_p.second << std::endl;
    }
}

return 0;
}

```