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. 9383 _____ Моисейченко К.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить алгоритм Форда-Фалкерсона – поиска максимального потока в сети. Реализовать данный алгоритм на языке программирования C++.

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 ω_{ij} – ребро графа

v_i v_j ω_{ij} – ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

v_i v_j ω_{ij} – ребро графа с фактической величиной протекающего потока

v_i v_j ω_{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

Вариант 5. Поиск в глубину. Итеративная реализация.

Основные теоретические положения.

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток.

Исток – вершина, из которой рёбра только выходят.

Сток – вершина, из которой рёбра только входят.

Поток – абстрактное понятие, показывающее движение по графу.

Величина потока – числовая характеристика движения по графу (сколько всего выходит из стока = сколько всего входит в сток).

Пропускная способность – свойство ребра, показывающее, какая максимальная величина потока может пройти через это ребро.

Максимальный поток (максимальная величина потока) – максимальная величина, которая может быть выпущена из стока, которая может пройти через все рёбра графа, не вызывая переполнения ни в одном ребре.

Фактическая величина потока в ребре – значение, показывающее, сколько величины потока проходит через это ребро.

Выполнение работы.

Реализация алгоритма Форда-Фалкерсона:

Алгоритм ищет максимальный поток в графе итеративным способом:

1. Находится путь в графе
2. Обновляются веса ребер в найденном пути.
3. Когда путей от истока к стоку не будет, алгоритм завершает свою работу.

Сложность.

Временная сложность алгоритма можно оценить как O(E*S), где S - сумма ребер, приходящих в сток.

Описание функций и структур данных.

void read_graph(Graph& G) - считывание ребра и заполнение графа

int Ford_Fulkerson(Graph& G, char start, char finish) - алгоритм Форда-Фалкерсона

bool check(Graph G, char start) - проверка условия основного цикла

std::string search_path(Graph& G, char cur, char finish, std::string path) - поиск пути в графе

int search_min_weight(std::string path, Graph G) - поиск минимального веса в пути

void decrease_weight(Graph& G, std::string path, int min) - уменьшение веса у нынешних ребер

void create_reversed_edges(Graph& G, std::string path, int min) - создание обратных ребер

std::vector <Answer> get_edges(Graph G, Graph G2) - составление списка пройденных ребер для ответа.

Примеры работы программы.

Test 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
```

Test output:

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

Hide

Рисунок 1 - Пример работы программы №1.

Test input:

```
5
1
4
1 2 20
1 3 1
2 3 20
2 4 1
3 4 20
```

Test output:

```
21
1 2 20
1 3 1
2 3 19
2 4 1
3 4 20
```

Hide

Рисунок 2 - Пример работы программы №2.

Test input:

```
3
a
c
a b 5
b c 7
a c 10
```

Test output:

```
15
a b 5
a c 10
b c 5
```

Рисунок 3 - Пример работы программы №3.

Выводы.

Изучен алгоритм Форда-Фалкерсона – поиска максимального потока в сети. Был изучен и итеративно реализован итеративный обход графа в глубину.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: solution.h

```
##pragma once
#include <iostream>
#include <map>
#include <tuple>
#include <vector>
#include <algorithm>

class Edge {
public:
    Edge(char next, float weight) {
        this->next_ = next;
        this->weight_ = weight;
        this->isVisited = 0;
    }

    char next()const {
        return this->next_;
    }

    float weight()const {
        return this->weight_;
    }
    void inc(int min) {
        this->weight_ = this->weight_ + min;
    }

    void dec(int min) {
        this->weight_ = this->weight_ - min;
    }

private:
    char next_;
    float weight_;
public:
    int isVisited;
};

using Graph = std::map<char, std::vector<Edge>>;

class Answer {
public:
    Answer(char out_, char in_, int weight_) {
        this->out = out_;
        this->in = in_;
        this->weight = weight_;
    }

public:
    char out;
    char in;
    int weight;
};
```

```
void read_graph(Graph& G);
int Ford_Fulkerson(Graph& G, char start, char finish);
bool check(Graph G, char start);
std::string search_path(Graph& G, char cur, char finish, std::string
path);
int search_min_weight(std::string path, Graph G);
void decrease_weight(Graph& G, std::string path, int min);
void create_reversed_edges(Graph& G, std::string path, int min);
std::vector <Answer> get_edges(Graph G, Graph G2);
```

Название файла: solution.cpp

```
#include "solution.h"

void read_graph(Graph& G)
{
    char out, in;
    float weight;
    while (std::cin >> out >> in >> weight)
    {
        Edge g(in, weight);
        if (G.count(out) == 0)
        {
            std::vector<Edge> d = { g };
            G[out] = d;
        }
        else
        {
            G[out].push_back(g);
            std::sort(G[out].begin(),  G[out].end(),  [](Edge&  p1,
Edge& p2) { return (p1.next() < p2.next()); });
        }
    }


}

int Ford_Fulkerson(Graph& G, char start, char finish)
{
    int max_flow = 0;
    bool check_ = check(G, start);
    char cur = start;
    std::string path;
    path.push_back(cur);
    while (check_)
    {
        path = search_path(G, cur, finish, path);
        if (path.empty())
            break;
        int min = search_min_weight(path, G);
        max_flow = max_flow + min;
        decrease_weight(G, path, min);
        create_reversed_edges(G, path, min);
        if (path.size() > 1)
        {
            path.pop_back();
            cur = path.back();
```

```
    check_ = check(G, start);
}
if (path.size() == 1)
    continue;
std::string new_path;
char out = path[0];
char in = path[1];
new_path.push_back(out);
int break_cycle = 0;
for (int i = 0; i != path.size() - 1; i++)
{
    for (int j = 0; j != G[out].size(); j++)
    {
        if (G[out][j].next() == in)
        {
            if (G[out][j].weight() > 0)
            {
                new_path.push_back(G[out][j].next());
                out = G[out][j].next();
                in = path[i + 2];
                cur = in;
                break;
            }
            else
            {
                cur = new_path[new_path.size() - 1];
                break_cycle = 1;
                break;
            }
        }
    }
    if (break_cycle)
        break;
}

for (int i = new_path.size(); i != path.size(); i++)
{
    out = path[i];
    in = path[i + 1];
    for (int y = 0; y != G[out].size(); y++)
    {
        if (G[out][y].next() == in)
        {
            G[out][y].isVisited = 0;
        }
    }
}
path = new_path;
if (path.size() >= 1)
{
    check_ = check(G, start);
}
else
{
    check_ = 0;
    break;
}
```

```
        }
        return max_flow;
    }

    bool check(Graph G, char start)
    {
        for (int i = 0; i != G[start].size(); i++)
        {
            if (G[start][i].weight() > 0)
            {
                return 1;
            }
        }
        return 0;
    }

    std::string search_path(Graph& G, char cur, char finish, std::string
path)
    {
        while (cur != finish)
        {
            char now = cur;
            for (int i = 0; i != G[cur].size(); i++)
            {

                int continue_cycle = 0;
                for (int j = 0; j != path.size(); j++)
                {
                    if (G[cur][i].next() == path[j])
                    {
                        continue_cycle = 1;
                    }
                }
                if (continue_cycle)
                {
                    continue;
                }

                if ((!G[G[cur][i].next()].empty() && G[cur][i].isVisited
== 0 && G[cur][i].weight() > 0) || (G[cur][i].next() == finish &&
G[cur][i].weight() > 0))
                {
                    G[cur][i].isVisited = 1;
                    cur = G[cur][i].next();
                    path.push_back(cur);
                    break;
                }
                else
                {
                    G[cur][i].isVisited = 1;
                    if (i == (G[cur].size() - 1)) {
                        path.pop_back();
                        cur = path.back();
                        break;
                    }
                }
            }
```

```
        if (cur == now)
        {
            if (path.size() < 2)
            {
                return path;
            }
            else
            {
                path.pop_back();
                cur = path.back();
            }
        }
    }
    return path;
}

int search_min_weight(std::string path, Graph G)
{
    int min = 10000;
    for (int i = 0; i != path.size() - 1; i++)
    {
        char out = path[i];
        char in = path[i + 1];
        for (int j = 0; j != G[out].size(); j++)
        {
            if (G[out][j].next() == in)
            {
                if (min > G[out][j].weight())
                    min = G[out][j].weight();
            }
        }
    }
    return min;
}

void decrease_weight(Graph& G, std::string path, int min)
{
    for (int i = 0; i != path.size() - 1; i++)
    {
        char out = path[i];
        char in = path[i + 1];

        Edge g(in, min);
        for (int j = 0; j != G[out].size(); j++)
        {
            if (G[out][j].next() == in)
                G[out][j].dec(min);
        }

    }

}

void create_reversed_edges(Graph& G, std::string path, int min)
{
    std::reverse(path.begin(), path.end());
    for (int i = 0; i != path.size() - 1; i++)
```

```
        {
            char out = path[i];
            char in = path[i + 1];

            Edge g(in, min);
            if (G.count(out) == 0)
            {
                std::vector<Edge> d = { g };
                G[out] = d;
            }
            else
            {
                int is_find = 0;
                for (int j = 0; j != G[out].size(); j++)
                {
                    if (G[out][j].next() == in)
                    {
                        G[out][j].inc(min);
                        is_find = 1;
                    }
                }

                if (is_find == 0)
                {
                    G[out].push_back(g);
                    std::sort(G[out].begin(), G[out].end(), [](Edge& p1,
Edge& p2) { return (p1.next() < p2.next()); });
                }

            }
        }

    }

    std::vector <Answer> get_edges(Graph G, Graph G2)
    {
        std::vector <Answer> answer;
        for (auto i = G2.begin(); i != G2.end(); i++)
        {
            int l = 0;
            int l2 = 0;

            for (auto j = i->second.begin(); j != i->second.end(); j++)
            {
                char out = i->first;
                char in = j->next();
                int l2 = j->weight();
                int l = 0;
                for (int k = 0; k != G[out].size(); k++)
                {
                    if (G[out][k].next() == in)
                        l = l2 - G[out][k].weight();
                }
                if (l < 0)
                    l = 0;
                Answer g(out, in, l);
                answer.push_back(g);
```

```
            }
        }
        std::sort(answer.begin(), answer.end(), [](Answer& p1, Answer&
p2) {
            if (p1.out < p2.out)
            {
                return true;
            }
            if (p1.out == p2.out)
            {
                return (p1.in < p2.in);
            }
            return false;
        });
        return answer;
    }
```

Название файла: main.cpp

```
    #include "solution.h"

    int main()
    {
        char start, finish;
        int N;
        std::cin >> N;
        std::cin >> start;
        std::cin >> finish;
        Graph G, G2;
        read_graph(G);
        G2 = G;
        int max_flow = Ford_Fulkerson(G, start, finish);
        std::cout << max_flow << "\n";
        std::vector <Answer> answer = get_edges(G, G2);
        for (int i = 0; i != answer.size(); i++) {
            std::cout << answer[i].out << " " << answer[i].in << " " <<
answer[i].weight << "\n";
        }
    return 0;
    }
```