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: Алгоритм Форда-Фалкерсона.

Студент гр. 7304 Субботин А.С.

Преподаватель Филатов А.Ю.

Санкт-Петербург

2019

Цель работы

Изучить и реализовать на языке программирования C++ алгоритм Форда-Фалкерсона, который ищет максимальный поток в сети.

Формулировка задачи

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона

Ход работы

Написана структура, полностью определяющая ребро графа, а также класс, содержащий функции: выбора списка ребра, которым можно воспользоваться из текущей вершины, обнуление отметки вершин, преобразование значения потока через путь от истока до стока. Главный метод класса, оперируя данными возможностями, просматривает различные варианты перемещения из вершины, выбирает ребро с максимально возможным потоком (если такого не найдено - происходит откат на шаг назад с запоминанием факта посещения вершины, если вершина при этом стартовая - окончание работы метода). Если рёбра из предыдущего шага могут выстроить путь - действует метод преобразования рёбер (в зависимости от минимального потока) и сброса отметок вершин. При вызове конструктора класса происходит считывание исходных данных, необходимые сортировки, а также создаётся дополнительный вектор, предназначенный для более удобного поиска рёбер, исходящих из данной вершины, вывод результата.

Результаты работы программы

Входные данные	Результат
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
3 a f a b 7 a c 6 b d 6	0 a b 0 a c 0 b d 0

Выводы

В ходе выполнения данной лабораторной работы был исследован и реализован алгоритм Форда-Фалкерсона, вместе с тем проведены необходимые оптимизации его реализации, просмотрено множество примеров и получен требуемый результат.

Приложение А. Код программы

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

typedef int type;
using namespace std;

typedef struct edge{
    char head, tail;
    type* out, *ent;
    bool checked;
} edge;

bool Sort(edge first, edge second){
    if(first.head == second.head)
        return first.tail < second.tail;
    return first.head < second.head;
}

bool Compare(edge first, edge second)
{
    if(first.head == second.head)
        return *(first.out) > *(second.out);
    return first.head < second.head;
}

class Ford{
public:
    vector<edge> graph, easy; //Вектор, который получили, и удобный вектор с
обратными путями (если не дубли)
    char start, finish, now; //Исток, сток и текущая вершина
    type count, help, min, sum;//Размеры векторов graph и easy, минимальный поток
в пути и максимальный в сети
    edge* elem; //Указатель для перемещения по вектору
    vector <type> prices; //Вектор пропускных способностей отрезков пути
    vector <edge*> way;
    bool trouble;

    Ford(){
        type price;
        cin >> count >> start >> finish;
        for(type i(0); i < count; i++){
            edge node;
            cin >> node.head >> node.tail >> price;
            node.out = new type(price);
            node.ent = new type(0);
            node.checked = false;
            graph.push_back(node);
        }
        easy = graph;
        help = count;
        for(type i(0); i < count; i++){
            bool rewerse = true;
            for(type j(0); j < count; j++)
                if(i != j && easy.at(i).head == easy.at(j).tail && easy.at(i).tail
== easy.at(j).head)
                    rewerse = false;
            if(rewerse){
                edge node;
```

```
            node.head = easy.at(i).tail;
            node.tail = easy.at(i).head;
            node.out = easy.at(i).ent;
            node.ent = easy.at(i).out;
            node.checked = easy.at(i).checked;
            easy.push_back(node);
            help++;
        }
    }
    if(count > 0){
        elem = nullptr;
        sort(easy.begin(), easy.end(), Compare);
        trouble = true;
        for(type i(0); i < count; i++)
            if(graph.at(i).head == start)
                trouble = false;
        if(!trouble)
            elem = Headseeker(start);
    }
    now = start;
    sum = 0;
}

void Process(){
    if(*(elem->out) && !elem->checked){
        elem->checked = true;
        for(type i(0); i < help; i++)
            if(easy.at(i).tail == now)
                easy.at(i).checked = true;
        prices.push_back(*(elem->out));
        way.push_back(elem);
        if(elem->tail != finish)
            now = elem->tail;
        else{
            min = 9999;
            while(!prices.empty()){
                if(min > prices.back())
                    min = prices.back();
                prices.pop_back();
            }
            sum += min;
            Sub();
            Checkfailer();
            sort(easy.begin(), easy.end(), Compare);
            now = start;
        }
        elem = Headseeker(now);
        Process();
    }
    else{
        now = elem->head;
        elem = &elem[1];
        if(elem->head != now){
            if(now == start)
                return;
            elem = Headseeker(way.back()->head);
            way.pop_back();
            prices.pop_back();
        }
        Process();
    }
}
```

```
    struct edge* Headseeker(char head)
    {
        int i = 0;
        while(head != easy.at(i++).head);
        return &easy.at(i-1);
    }

    void Checkfailer()
    {
        for(type i(0); i < help; i++)
            easy.at(i).checked = false;
    }

    void Sub()
    {
        struct edge* node;
        while(!way.empty()){
            node = way.back();
            *(node->out) -= min;
            *(node->ent) += min;
            way.pop_back();
        }
    }

    void Out()
    {
        cout << sum << endl;
        if(count > 0)
            sort(graph.begin(), graph.end(), Sort);
        for(type i(0); i < count; i++)
            for(type j(i+1); j < count; j++)
                if(graph.at(i).head == graph.at(j).tail && graph.at(i).tail ==
graph.at(j).head){
                    if(*(graph.at(i).ent) > *(graph.at(j).ent)){
                        *(graph.at(i).ent) -= *(graph.at(j).ent);
                        *(graph.at(j).ent) = 0;
                    }
                    else{
                        *(graph.at(j).ent) -= *(graph.at(i).ent);
                        *(graph.at(i).ent) = 0;
                    }
                }
        for(type i(0); i < count; i++)
            cout << graph.at(i).head << " " << graph.at(i).tail << " " <<
*(graph.at(i).ent) << endl;
    }
};

int main()
{
    Ford dot;
    if(dot.count > 0 && dot.elem)
        dot.Process();
    dot.Out();
    return 0;
}
```