МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по практической работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потоки в сети

Студент гр. 8304 _____ Воропаев А.О.
Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2020

Вариант 3.

Цель работы.

Построение и анализ алгоритма Форда-Фалкерсона на основе на решения задачи о нахождении максимального потока в сети.

Расширение.

Поиск в глубину. Рекурсивная реализация.

Основные теоретические положения.

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).

Описание алгоритма.

Для решения поставленной задачи была написана рекурсивная функция ff_algoritm, которая осуществляет поиск максимального потока путем поиска в глубину. Хранение пути осуществляется при помощи хранения всех введённых рёбер, и обратных для них рёбер, которые создаются при введении исходного ребра; для таких рёбер переменная is_reverse устанавливается в значение true. Функция принимает проверяемую вершину и минимальную пропускную способность, найденного на данный момент пути. В теле функции переданная вершина обозначается как посещённая (для этого был создан словарь std::map<char, bool> visited), затем мы проходим по всем детям данной вершины и запускаем рекурсивную функцию от каждой из них, если в них еще осталась пропускная способность(текущий поток меньше максимального через данную вершину для прямого ребра; текущий поток не равен нуля для обратного ребра). Затем, для каждой вершины, от которой была запущена функция, к значению

текущего потока прибавляется(отнимается для обратного ребра) найденный с помощью рекурсии поток Рекурсивная функция возвращает значение, если проверяемая вершина является стоком графа либо не имеет детей.

Сложность алгоритма.

Добавляя поток увеличивающего пути к уже имеющемуся потоку, максимальный поток будет получен, когда нельзя будет найти увеличивающий путь. Время работы ограничено $O(|E|f)$ где E — число рёбер в графе, f — максимальный поток в графе, так как каждый увеличивающий путь может быть найден за $O(E)$ и увеличивает поток как минимум на 1.

Описание основных структур данных и функций.

Программа считывает размер квадрата с консоли и выводит результат в консоль.

1) struct edge – структура, содержащая данные о ребрах

char destination – вершина, в которую входит данное ребро

int max – максимальная пропускная способность ребра

int current – текущий поток в ребре

bool is_reverse – флаг, обозначающий обратное ли ребро

2) struct node – структура, содержащая вектор всех детей для каждой вершины.

std::vector<edge> destinations

3) void input() – функция, предназначенная для считывания данных.

Запись данных производится в словарь std::map<char, node> dict, в котором каждой вершине сопоставляются данные об этой вершине.

4) ff_algorithm(char u, int c_min) – рекурсивная функция, осуществляющая работу алгоритма Форда-Фалкерсона. Для более подробного описания работы функции см. «Описание алгоритма».

char u – обрабатываемая вершина

int c_min – минимальный поток в пути, на данный момент

Тестирование.

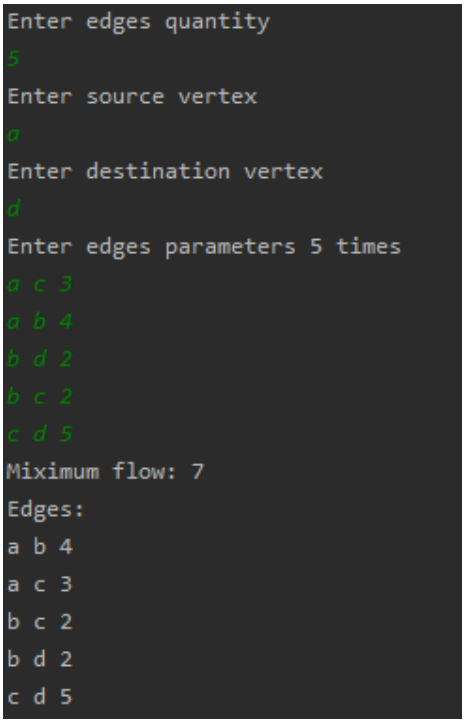

Рисунок 1 – Результаты 1-ого теста

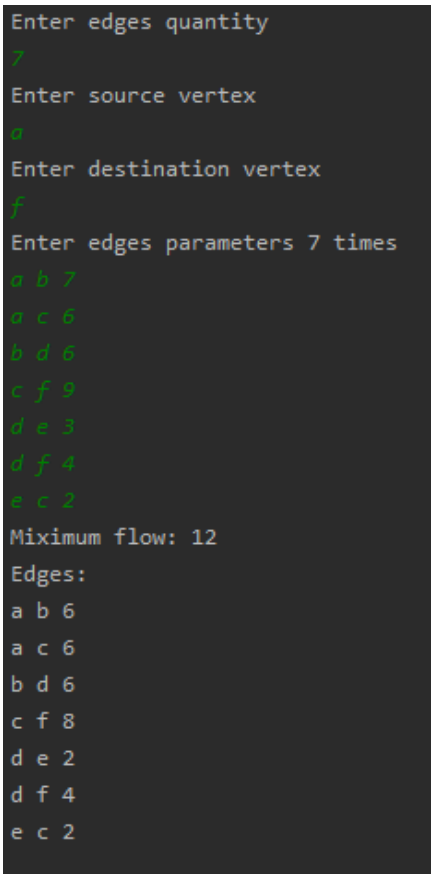

Рисунок 2 – Результаты 2-го теста

Вывод.

Был получен опыт в реализации алгоритма Форда-Фалкерсона с помощью рекурсивного поиска в глубину. Также в ходе работы была проанализирована сложность работы алгоритма.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

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

char source, destination;

struct edge {
    char destination{};
    int max{};
    int current{};
    bool is_reverse = false;
};


struct node {
    std::vector<edge> destinations;
};

std::map<char, node> dict;
std::map<char, bool> visited;



void input() {
    int quantity;

    std::cout << "Enter edges quantity\n";
    std::cin >> quantity;

    std::cout << "Enter source vertex\n";
    std::cin >> source;

    std::cout << "Enter destination vertex\n";
    std::cin >> destination;


    node tmp = node();
    char from, to;
    int len;
    edge to_child{};

    std::cout << "Enter edges parameters " << quantity << " times\n";
    for(int i = 0; i < quantity; ++i){
        std::cin >> from >> to >> len;

        bool f = false;
        for(auto& k : dict[from].destinations) {
            if(k.destination == to) {
                k = {to, len, 0, false};
                f = true;
                break;
            }
        }
        if(f)
```

```
            continue;

        if(dict.find(from) == dict.end()) {
            to_child = {to, len, 0};
            tmp.destinations.push_back(to_child);
            dict[from] = tmp;
            visited[from] = false;
        }
        else {
            to_child = {to, len, 0};
            dict[from].destinations.push_back(to_child);
        }
        tmp.destinations.clear();

        if(from != source) {
            if(dict.find(to) == dict.end()) {
                to_child = {from, len, 0, true};
                tmp.destinations.push_back(to_child);
                dict[to] = tmp;
                visited[from] = false;
            }
            else{
                to_child = {from, len, 0, true};
                dict[from].destinations.push_back(to_child);
            }
            tmp.destinations.clear();

        }
    }

    for(auto& i : dict) {
        std::sort(i.second.destinations.begin(), i.second.destinations.end(), [](edge
e1, edge e2){return e1.max < e2.max;});
    }
}



int ff_algorithm(char u, int c_min) {

    if(u == destination)
        return c_min;

    visited[u] = true;

    for(auto& v : dict[u].destinations) {
        if (v.destination != u) {
            if (!visited[v.destination] && (v.current < v.max && !v.is_reverse)) {

                int flow = ff_algorithm(v.destination, std::min(c_min, v.max - v.cur-
rent));
                if (flow > 0) {
                    v.current += flow;
                    return flow;
                }
            } else if (!visited[v.destination] && (v.is_reverse && v.current > 0)) {
                int flow = ff_algorithm(v.destination, std::min(c_min, v.current));
                if (flow > 0) {
                    v.current -= flow;
                    return flow;
                }
```

```
            }
        }
    }
    return 0;
}


int main() {

    input();

    int maxFlow = 0;
    int iterationResult = 0;
    while (true) {
        iterationResult = ff_algorithm(source, 1000);
        if(iterationResult <= 0)
            break;
        for (auto& i : visited)
            i.second = false;
        maxFlow += iterationResult;

    }

    for(auto& i : dict) {
        std::sort(i.second.destinations.begin(), i.second.destinations.end(), [](edge
e1, edge e2){return e1.destination < e2.destination;});
    }


    std::cout << "Miximum flow: " << maxFlow << "\nEdges:" << std::endl;
    for(auto& i : dict) {
        for(auto& j : i.second.destinations) {
            if(!j.is_reverse)
                std::cout << i.first << " " << j.destination << " " << j.current <<
std::endl;
        }
    }
    return 0;
}
```