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: Максимальный поток

Студентка гр. 9383 _____ Хотяков Е.П.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмом Форда-Фалкерсона, реализовать алгоритм на одном из языков программирования.

Вариант 2. Поиск в ширину. Обработка совокупности вершин текущего фронта как единого целого, дуги выбираются в порядке уменьшения остаточных пропускных способностей.

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

Основные теоретические положения.

Чтобы говорить об алгоритме необходимо ввести ряд понятий:

Сеть – это такой ориентированный взвешенный граф, что имеет один исток и один сток.

Исток – вершина, из которой ребра выходят, но не входят.

Сток – вершина, в которую ребра входят, но не выходят.

Поток – это понятие, описывающее движение по графу.

Величина потока – числовая характеристика потока.

Пропускная способность ребра – свойство ребра, которое показывает, какая максимальная величина потока может пройти через ребро графа.

Максимальный поток – такая максимальная величина, которая может пройти из истока по всем ребрам графа, не вызывая переполнения ни в одной пропускной способности ребра.

Фактическая величина потока в ребре – значение, которое показывает сколько величины потока проходит через ребро.

Алгоритм Форда-Фалкерсона – алгоритм, который служит для нахождения максимального потока в сети.

Описание алгоритма.

В начале работы алгоритму на вход подается граф для поиска максимального потока, вершина-исток и вершина-сток графа.

После считывания входных данных начинает работу сам алгоритм по следующим принципам:

1. Запускается поиск пути в ширину в графе от истока к стоку.

2. Если путь найден, то происходит вычисление максимального потока – это будет величина минимального ребра этого пути.

3. Для всех ребер найденного пути поток увеличивается на найденную в пункте 2 величину, а пропускная способность на эту величину уменьшается.

4. Полученное значение максимального потока в найденном пути в пункте 2 прибавляется к значению максимального потока всего графа, после чего запускается новый поиск в ширину.

5. Алгоритм осуществляет свою работу пока существует какой-либо пусть из вершины-истока к вершине-стоку.

Для удобства отслеживания процесса работы алгоритма в консоль выводятся промежуточные результаты.

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

Сложность алгоритма по памяти: O (E), где E – количество ребер.

Особенность обхода, а точнее, обход в ширину в порядке уменьшения остаточной пропускной способности, не влияет на скорость работы алгоритма в отличие от обычного обхода в ширину, ведь мы так или иначе на каждом шаге проходим по всем вершинам текущего фронта и добавляем их потомков в открытый список, а в каком порядке мы их добавим значение не имеет.

Описание функций и структур данных.

std::map<char, std::map<char, std::pair<int, bool>>> &Graph

Структура данных, используемая для хранения направленного ориентированного графа. Представляет собой ассоциативный контейнер хранения вершин и соответствующего ей контейнера вершина-расстояние. Для каждой вершины хранится ассоциативный массив вершин, до которых можно добраться из текущей и вес пути до них. Булевое значение используется для того, чтобы отслеживать, что ребро было добавлено изначально, а не по ходу работы алгоритма.

Например, граф вида:

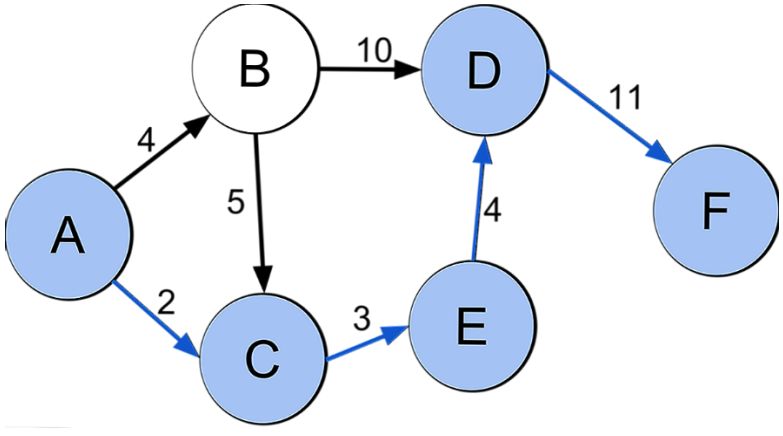

Рисунок 1 – Пример хранения графа

Визуально будет хранить в следующем виде:

A: B 4

C 2

B: C 5

D 10

C: E 3

D: F 11

E: D 4

bool BFS(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, std::map<char, char> &close, char &v0, char &vn)

чёчё Функция осуществляющая поиск в ширину. На вход принимает ссылку на граф *graph*, в котором будет осуществляться поиск, словарь с закрытыми вершинами, исток и сток.

Функция возвращает *true*, если при поиске была достигнута финальная вершина, *false* – противном случае.

int ff(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, char &v0, char &vn)

Функция, осуществляющая алгоритм Форда-Фалкерсона нахождения максимального потока в сети. На вход принимает граф *graph*, исток и сток. В качестве возвращаемого значения используется максимальный поток в графе.

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:

9
a
d
a b 8
b c 10
c d 10
h c 10
e f 8
g h 11
b e 8
a g 10
f d 8

Результат работы программы:

18
a b 8
a g 10
b c 0
b e 8
c d 10
e f 8
f d 8
g h 10
h c 10

Входные данные:

8
1
4
1 3 6
2 4 8
2 5 1
3 5 2
3 6 4
6 5 7
5 4 6
1 2 7

Результат работы программы:

13
1 2 7
1 3 6
2 4 7
2 5 0
3 5 2
3 6 4
5 4 6
6 5 4

Выводы.

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Форда-Фалкерсона, который находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро. Также был реализован алгоритм обхода в ширину в качестве индивидуализации данной лабораторной работы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

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

bool mycmp(std::pair<char, std::pair<char, int>> a, std::pair<char, std::pair<char, int>> b)
{
    return a.second.second < b.second.second;
}

void readData(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, char &v0, char &vn)
{
    int N;
    std::cin >> N;
    std::cin >> v0 >> vn;
    char source, distination;
    int weight;
```

```
        for (int i = 0; i < N; i++)
        {
            std::cin >> source >> distination >> weight;
            Graph[source][distination].first = weight;
            Graph[source][distination].second = true;
            if (!Graph[distination][source].first)
            {
                Graph[distination][source].first = 0;
                Graph[distination][source].second = false;
            }
        }
    }

    bool BFS(std::map<char, std::map<char, std::pair<int, bool>>> &Graph,
std::map<char, char> &close, char &v0, char &vn)
    {
        std::vector<std::pair<char, std::pair<char, int>>> open;
        std::pair<char, int> cur_item;
        //Для первой вершины(исток)
        for (auto it = Graph[v0].begin(); it != Graph[v0].end(); ++it)
        {
            if (it->second.first != 0)
```

```
            open.push_back(std::make_pair(v0, std::make_pair(it->first, it->second.first)));
        }
        close[v0] = '.'; //специально для истока - предыдущий к нему.

        //запускаем обход вершин
        while (1)
        {
            sort(open.begin(), open.end(), mycmp);                    //сортируем открытый список по длине ребер
            std::vector<std::pair<char, std::pair<char, int>>> tmp_open; // first - откуда пришли, second.first - сама вершина, second.second - расстояние до неё от предыдущей
            while (!open.empty())
            {
                cur_item = open.back().second;                //достаем наиболее приоритетную вершину
                if (close.find(cur_item.first) != close.end()) //в случае если вершина была повторно внесена в открытый список(как ребенок), то пропускаем её
                {
                    open.pop_back();
                    continue;
                }
```

```
            if (cur_item.first == vn) //если эта вершина - конечная, то возвращаем её
            {
                close[vn] = open.back().first;
                return true;
            }
            close[cur_item.first] = open.back().first;
//добавляем вершину в закрытый список
            open.pop_back();
//убираем вершину из открытого списка
            for (auto it = Graph[cur_item.first].begin(); it != Graph[cur_item.first].end(); it++) //проходимся по всем детям этой вершины
            {
                if (it->second.first != 0)                //если расстояние до ребенка = 0, то пропускаем её(дороги нет)
                    if (close.find(it->first) == close.end()) //если вершины нет в закрытом списке, то добавляем её к открытым
                    {
                        tmp_open.push_back(std::make_pair(cur_item.first, std::make_pair(it->first, it->second.first)));
                    }
            }
        }
        open = tmp_open; //обновляем список открытых
```

```
            if (open.empty())
            {
                return false; //если ни одного пути не было найдено
            }
            tmp_open.clear(); //отчищаем временный список
        }
    }

    int findMinFlox(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, std::map<char, char> &close, char &vn)
    {
        int minValue = Graph[close[vn]][vn].first;
        char cur_vertex = vn;
        int tmp_value;
        while (close[cur_vertex] != '.')
        {
            tmp_value = Graph[close[cur_vertex]][cur_vertex].first;
            cur_vertex = close[cur_vertex];
            if (tmp_value < minValue)
                minValue = tmp_value;
        }
        return minValue;
    }
```

```
int restorePath(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, std::map<char, char> &close, char &vn)
{
    char cur_vertex = vn;
    int minValue = findMinFlox(Graph, close, vn);
    while (close[cur_vertex] != '.')
    {
        Graph[close[cur_vertex]][cur_vertex].first -= minValue;
        Graph[cur_vertex][close[cur_vertex]].first += minValue;
        cur_vertex = close[cur_vertex];
    }
    return minValue;
}

int ff(std::map<char, std::map<char, std::pair<int, bool>>> &Graph, char &v0, char &vn)
{
    std::map<char, char> close;
    int maxFlow = 0;
    while (1)
    {
        if (BFS(Graph, close, v0, vn))
```

```
            maxFlow += restorePath(Graph, close, vn);
        else
            return maxFlow;
        close.clear();
    }
    return maxFlow;
}

int main()
{
    char v0, vn;
    std::map<char, std::map<char, std::pair<int, bool>>> Graph;
    readData(Graph, v0, vn);
    std::map<char, std::map<char, std::pair<int, bool>>> Graph_tmp =
Graph;
    int max = ff(Graph_tmp, v0, vn);
    // Graph_print
    std::cout << max << '\n';
    auto jt = Graph_tmp.begin();
    for (auto it = Graph.begin(); it != Graph.end(); ++it, ++jt)
    {
        auto j = jt->second.begin();
        for (auto i = it->second.begin(); i != it->second.end(); ++j, ++i)
```

```
            {
                if (i->second.second)
                    if (i->second.first - j->second.first >= 0)
                        std::cout << it->first << ' ' << i->first << ' ' << (i->second.first - j->second.first) << '\n';
                    else
                        std::cout << it->first << ' ' << i->first << ' ' << 0 << '\n';
            }
        }
    }
```