

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

Логинов Е.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

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

Формулировка задания.

Вариант 4. Поиск в глубину. Итеративная реализация.

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). 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).

Входные данные:

N - количество ориентированных рёбер 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

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

Соответствующие выходные данные

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Теоретические сведения.

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток. Исток – вершина, из которой рёбра только выходят.

Сток – вершина, в которую 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.

Описание алгоритма.

Строится остаточная сеть, в которой изначальный поток через каждое ребро равен 0. Затем ищется путь от истока к стоку по ребрам, вес которых не равен 0. Для найденного пути определяется минимальная пропускная способность, эта величина добавляется к величине потока и вычитается из всех весов ребер. После этого ищется новый путь от истока к стоку до тех пор, пока возможно найти путь.

Оценка сложности.

Оценка по времени: $O((R + E) * F)$, где R число ребер в графе, E – кол-во вершин, F – максимальный поток, $(R + E)$ – поиск в глубину.

Оценка по памяти: $O(R + E)$, где R – число ребер в графе, E – кол-во вершин.

Описание структур данных и функций алгоритма.

1. struct edge

```
struct edge
{
    char first; // начальная вершина
    char second; // вершина назначения
    int weight; // вес ребра

    int forward; // поток в найденном пути
    int back; // поток в противоположном пути
    bool bidirectional; // двунаправленное ребро
};
```

Структура предназначена для хранения ребер графа с вершинами first, second и весом weight, forward – величина потока в найденном пути, back – величина потока в обратном пути, bidirectional – флаг, является ли ребро двунаправленным. Если это так, то происходит слияние рёбер.

2. Class Graph

```
class Graph
{
private:

    char start; // исток
    char finish; // сток
    int N; // количество ребер
    vector <edge> edges; // ребра
    vector <char> solutions; // найденный путь
    vector <char> checkedpoint; // просмотренные вершины
};
```

Класс для осуществления работы алгоритма Форда-Фалкерсона. Содержит контейнеры для хранения ребер, вершин, найденного пути, количество вершин, сток и исток.

3. Graph()

Конструктор класса, выполняет считывание информации о графе: количество ребер, вершина стока и истока, данные о всех ребрах (добавляются в контейнер edges). Также выполняется проверка на то, является ли текущее ребро двунаправленным. Если это так, то к обратному потоку добавляется прямой поток.

4. Bool isChecked(char elem)

Функция для проверки является ли вершина уже просмотренной. Получает на вход вершину и проводит поиск в контейнере checkedpoint.

5. bool Search()

Функция для нахождения пути к стоку. В начале работы создается стек `s`, в который помещается вершина истока, после этого начинается основной цикл `while(!s.empty())`.

Работает на основе обхода в глубину, итеративно перебирая все ребра из текущей рассматриваемой вершины. В случае, если ребро ведет в вершину, которая не рассматривалась ранее, то поиск начинается от этой нерассмотренной вершины, а после возвращаем и продолжаем перебирать ребра. При нахождении стока фиксируется минимальная пропускная способность и ф-ия возвращает `true`;

6. `int FordFulkerson()`

Функция, реализующая алгоритм Форда-Фалкерсона. Запускается функция `Search` в цикле `while()` до тех пор, пока она не вернет `false`. Для найденного пути вычисляется минимальная пропускная способность через функцию `calc_min()`. После каждого найденного пути минимальный вес ребра вычитается из весов ребер потока и прибавляется к величине максимального потока. Так получают фактические пропускные способности для каждого ребра.

7. `int calc_min()`

Функция определяет минимальную пропускную способность. Изначально наименьший вес ребра инициализируется большим числом и изменяется в процессе поиска.

8. `void print(int maxFlow)`

Функция для печати результата работы программы, получает на вход переменную с результатом работы алгоритма.

Тестирование.

12

a

e

a b 5

a c 4

a d 2

b c 1

b g 5

b f 3

c g 4

c f 1

d c 2

d f 1

g e 5

f e 8

10

a b 0

a c 0

a d 2

b c 0

b f 3

b g 0

c f 1

c g 0

d c 1

d f 1

f e 5

g e 5

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 c

a b 7

b c 12

b a 5

7

a b 7

b a 0

b c 7

Пример 1

Пример 2

Пример 3

7	Current path: abdecf(10)
a	ab(3)
f	bd(2)
a b 7	de(3)
a c 6	ec(2)
b d 6	cf(3)
c f 9	
d e 3	Current min: 2
d f 4	Recount:
e c 2	straight:ab(3-2)=1
Current path: acf(0)	back:ba(1+2)=3
ac(6)	straight:bd(2-2)=0
cf(9)	back:db(0+2)=2
Current min: 6	straight:de(3-2)=1
Recount:	back:ed(1+2)=3
straight:ac(6-6)=0	straight:ec(2-2)=0
back:ca(0+6)=6	back:ce(0+2)=2
straight:cf(9-6)=3	straight:cf(3-2)=1
back:fc(3+6)=9	back:fc(1+2)=3
Max Flow: 6	Max Flow: 12
Current path: abdf(6)	
ab(7)	12
bd(6)	a b 6
df(4)	a c 6
Current min: 4	b d 6
Recount:	c f 8
straight:ab(7-4)=3	d e 2
back:ba(3+4)=7	d f 4
straight:bd(6-4)=2	e c 2
back:db(2+4)=6	
straight:df(4-4)=0	
back:fd(0+4)=4	
Max Flow: 10	

Пример 4, с отладкой

Способ хранения частичных решений.

Частичные решения хранятся в контейнере solutions.

Выводы.

В ходе выполнения лабораторной работы был изучен и реализованы на языке программирования C++ алгоритм Форда-Фалкерсона, определяющий максимальный поток в сети.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits.h>
#include <stack>
using namespace std;
#define SHOW_DEBUG

struct edge
{
    char first; // начальная вершина
    char second; // вершина назначения
    int weight; // вес ребра

    int forward; // поток в найденном пути
    int back; // поток в противоположном пути
    bool bidirectional; // двунаправленное ребро
};

bool compare(edge a, edge b)
{
    if (a.first == b.first)
        return a.second < b.second;
    return a.first < b.first;
}

class Graph
{
private:
    char start; // исток
    char finish; // сток
    int N; // количество ребер
    vector <edge> edges; // ребра
    vector <char> solutions; // найденный путь
    vector <char> checkedpoint; // просмотренные вершины
public:
    Graph()
    {
        cin >> N;
        cin >> start >> finish;
        for (int i = 0; i < N; i++)
        {
```

```

        edge elem;
        cin >> elem.first >> elem.second >> elem.weight; //
считывание ребра
        elem.forward = elem.weight;
        elem.back = 0;
        elem.bidirectional = false;

        bool flag = true;
        for (int i = 0; i < edges.size(); i++) // проверка на
существование обратного ребра
        {
            if (edges.at(i).first == elem.second &&
edges.at(i).second == elem.first)
            {
                edges.at(i).back += elem.forward;
                flag = false;
                edges.at(i).bidirectional = true;
                break;
            }
        }
        if ( !flag) // если получено обратное ребро, то не
добавляем
            continue;
        edges.push_back(elem); // добавляем ребро
    }
}

bool isChecked(char elem)
{
    for (int i = 0; i < checkedpoint.size(); i++)
        if (checkedpoint.at(i) == elem)
            return true;
    return false;
}

bool Search()
{
    stack<char> s;
    s.push(start);

    while(!s.empty())
    {
        char elem = s.top();
        checkedpoint.push_back(elem); // добавляем в вектор
просмотренных вершин
        s.pop();

        bool found = false;
        for (int i = 0; i < edges.size(); i++)
        {

```

```

        if (elem == edges.at(i).first) // прямой ход по ребру
        {
            if (isChecked(edges.at(i).second) ||
edges.at(i).forward == 0)
                continue; // если далее путь просмотрен или
пропускная способность равна 0, то не рассматриваем

            if(edges.at(i).second == finish)
            {
                solutions.push_back(elem);
                solutions.push_back(finish);
                return true;
            }

            found = true;
            s.push(edges.at(i).second);
        }

        if (elem == edges.at(i).second) // обратный ход по
ребру
        {
            if (isChecked(edges.at(i).first) ||
edges.at(i).back == 0)
                continue; // если далее путь просмотрен или
пропускная способность равна 0, то не рассматриваем

            if(edges.at(i).first == finish)
            {
                solutions.push_back(elem);
                solutions.push_back(finish);
                return true;
            }

            found = true;
            s.push(edges.at(i).first);
        }
    }

    if(found) {
        solutions.push_back(elem);
    }
    else
        solutions.pop_back();

}
return false;
}

int calc_min()
{
    int min = INT_MAX;

```

```

    for (int i = 1; i < solutions.size(); i++)
    {
        for (int j = 0; j < edges.size(); j++)
        {
            if (edges.at(j).first == solutions.at(i - 1) &&
edges.at(j).second == solutions.at(i))
            {
                if(edges.at(j).forward < min)
                    min = edges.at(j).forward;
            }
            if (edges.at(j).second == solutions.at(i - 1) &&
edges.at(j).first == solutions.at(i))
            {
                if(edges.at(j).back < min)
                    min = edges.at(j).back;
            }
        }
    }
    return min;
}

void print(int maxFlow)
{
    cout << maxFlow << endl; // вывод данных
    sort(edges.begin(), edges.end(), compare);
    for (int i = 0; i < edges.size(); i++)
    {
        int elem = max(edges.at(i).weight - edges.at(i).forward, 0
- edges.at(i).back);
        if (edges.at(i).bidirectional == true)
        {
            if (elem < 0)
                elem = 0;
            cout << edges.at(i).first << " " << edges.at(i).second
<< " " << elem << endl;
            swap(edges.at(i).first, edges.at(i).second);
            swap(edges.at(i).back, edges.at(i).forward);
            edges.at(i).bidirectional = false;
            sort(edges.begin(), edges.end(), compare);
            i--;
        }
        else
            cout << edges.at(i).first << " " << edges.at(i).second
<< " " << elem << endl;
    }
}

int FordFulkerson()
{
    int maxFlow = 0; // максимальный поток
    int min = INT_MAX; // минимальная пропускная способность

```

```

while (Search()) // пока есть путь к стоку
{

#ifdef SHOW_DEBUG
    cout << "Current path: " << start;
    for (int i = 1; i < solutions.size(); i++) {
        cout << solutions.at(i);
    }
    cout << "(" << maxFlow << ")"<< endl;

    for (int i = 1; i < solutions.size(); i++)
    {
        for (int j = 0; j < edges.size(); j++)
        {
            if (edges.at(j).first == solutions.at(i - 1) &&
edges.at(j).second == solutions.at(i))
            {
                cout <<" " << edges.at(j).first <<
edges.at(j).second << "(" << edges.at(j).forward<<")" << endl;
            }
            if (edges.at(j).second == solutions.at(i - 1) &&
edges.at(j).first == solutions.at(i))
            {
                cout <<" " << edges.at(j).second <<
edges.at(j).first << "(" << edges.at(j).back<<") - back" << endl;
            }
        }
    }
#endif

    int min = calc_min(); // минимальная пропускная
способность

#ifdef SHOW_DEBUG
    cout << "Current min: " << min << endl;
    cout << "Recount:" << endl;
#endif

    for (int i = 1; i < solutions.size(); i++)
    {
        for (int j = 0; j < edges.size(); j++)
        {
            if (edges.at(j).first == solutions.at(i - 1) &&
edges.at(j).second == solutions.at(i))
            {
#ifdef SHOW_DEBUG
                cout << "straight:" << edges.at(j).first <<
edges.at(j).second << "(";
                cout << edges.at(j).forward << "-" << min<<")"
<< "=" << edges.at(j).forward-min << endl;
#endif
            }
        }
    }
}

```

```

        edges.at(j).forward -= min;
#ifdef SHOW_DEBUG
        cout << "back:" << edges.at(j).second <<
edges.at(j).first <<"(";
        cout << edges.at(j).forward << "+" << min
<<)"<< "=" << edges.at(j).forward+min << endl;
#endif
        edges.at(j).back += min;
    }
    if (edges.at(j).second == solutions.at(i - 1) &&
edges.at(j).first == solutions.at(i))
    {
#ifdef SHOW_DEBUG
        cout << "straight:" << edges.at(j).first <<
edges.at(j).second <<"(";
        cout << edges.at(j).forward << "+" << min <<
"=" << edges.at(j).forward+min << endl;
#endif
        edges.at(j).forward += min;
#ifdef SHOW_DEBUG
        cout << "back:" << edges.at(j).second <<
edges.at(j).first <<"(";
        cout << edges.at(j).forward << "-" << min <<
"=" << edges.at(j).forward-min << endl;
#endif
        edges.at(j).back -= min;
    }
    }
    }
    maxFlow += min;
#ifdef SHOW_DEBUG
    cout << "Max Flow: " << maxFlow << endl;
#endif
    checkedpoint.clear();
    solutions.clear();
    min = INT_MAX;

    cout<<endl;
}

    return maxFlow;
}
};
int main()
{
    Graph task;
    int maxFlow = task.FordFulkerson();
    task.print(maxFlow);

    return 0;
}

```