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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студентка гр. 9383 Сергиенкова А.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++.

Основные теоретические положения.

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.

Постановка задачи.

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:

Pmax - 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
```

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
```

Выполнение работы:

Описание алгоритма

Для того чтобы реализовать алгоритм, мы сначала строим граф, который состоит из массива рёбер. Все потоки должны быть равны 0 изначально.

Для нахождения пути в графе, используется поиск в глубину, итеративная реализация (вариативное задание).

Суть алгоритма:

Если удалось найти путь из истока в сток, то выполняется поиск максимального потока для этого пути. Далее это максимальное значение потока прибавляется к конечному значению максимального потока для всего графа.

Сложность по памяти – линейная O(|E|), где E – число рёбер.

Если величина пропускной способности – это иррациональное число, то алгоритм может работать бесконечно. При целых числах такой проблемы не возникает и время работы ограничено.

Вариант 4. Поиск в глубину. Итеративная реализация.

Описание функций и структур данных:

- class Resolver – содержит в себе реализацию алгоритма Форда-Фалкерсона;
- Graph _Graph – исходный граф;
- Graph _Result – результирующий граф;
- **char** _In – исток;
- **char** _Out – сток;
- **int** _MaxStream – максимальный поток из истока в сток;
- std::vector<**char**> _Path – текущий путь;
- **void** Go(**char** point) – переходит в указанную вершину при сканировании вершин;
- **void** CheckPath()– проверяет найденый путь (по алгоритму);
- class Graph – класс, который хранит в себе структуру ориентированного графа;
- std::map<**char**, std::vector<Edge>> _Data – набр ребер на каждый узел;
- **int** m_EdgesCount – количество рёбер;
- **int** EdgesCount() { **return** m_EdgesCount; } – возвращение количества рёбер;
- std::vector<Edge> **operator**[](**char** node) – возвращает сортированный по весам вектор ребер из указанной вершины;
- **void** Set(Edge e) – задает ребро;
- Edge& Get(**char** from, **char** to) – возвращает ссылку на ребро;
- class Finder – выполняет вариативное задание;
- **bool** Find(Graph& g, **char** item) – ищет в указанном графе указанное значение

- **class** Edge – хранит ребро графа.
- Edge(**char** from, **char** to, **unsigned int** weight) – ребро;
- **char** From() – откуда;
- **char** To() – куда;
- **unsigned int** Weight() – с каким весом;
- **void** SetWeight(**int** weight) { _Weight = weight; } – возвращает вес.

Тестирование

```
5
a
b
a b 12
b a 12
b c 2
a c 4
c b 12
a->c: 4
a->b: 12
b->c: 2
b->a: 12
c->b: 12

16
a->c: 4
a->b: 12
b->c: 0
b->a: 0
c->b: 4
```

Рисунок 1 – Тест программы №1

```
1
f
k
k f 8
k->f: 8

0
k->f: 0
```

Рисунок 2 – Тест программы №2

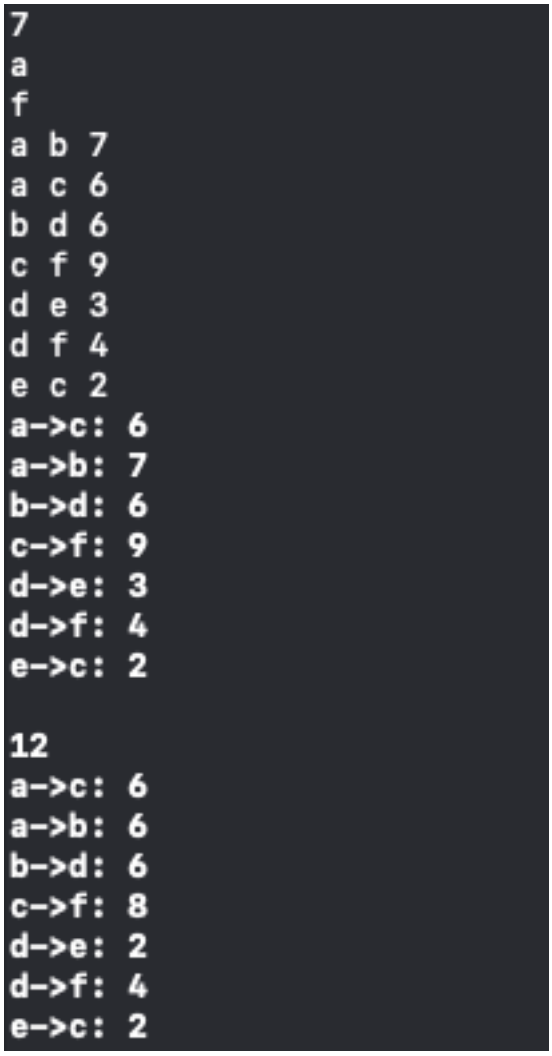

Рисунок 3 – Тест программы №3

Вывод

Изучен алгоритм Форда-Фалкерсона – поиска максимального потока в сети. Реализован данный алгоритм на языке программирования C++. Был изучен и итеративно реализован обход графа в глубину.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Main.cpp:

```
#include <iostream>
#include <sstream>
#include "Graph.h"
#include "Resolver.h"
#include "Finder.h"

int main() {
    // ввод данных
    Graph g;
    //std::stringstream ss("7\na\nf\na b 7\na c 6\nb d 6\nc f 9\nd
e 3\nd f 4\ne c 2");
    std::istream& is = std::cin;
    //std::istream& is = ss;

    // ввод данных
    int count;
    is >> count;
    g.Reset(count);
    char in, out;
    is >> in >> out;
    is >> g;

    // вывод графа
    std::cout << g << std::endl;

    // производим решение
    Resolver r;
    r.Resolve(g, in, out);
    // вывод результата
    std::cout << r << std::endl;

    // производим поиск в глубину
    //Finder f;
    //std::cout << "Find result: " << f.Find(g, 'e') << std::endl;

    system("pause");
    return 0;
}
```

Resolver.h

```
#pragma once
#include <iostream>
#include "Graph.h"

class Resolver
{
    Graph _Graph;   // исходный граф
```

```
    Graph _Result;  // результирующий граф
    char _In;       // исток
    char _Out;      // сток
    int _MaxStream; // максимальный поток из истока в сток
    std::vector<char> _Path; // текущий путь

    void Go(char point); // переходит в указанную вершину при
сканировании вершин
    bool Contains(std::vector<char>& path, char point);
    void CheckPath();    // проверяет найденый путь (по алгоритму)
public:
    void Resolve(Graph &g, char in, char out);

    friend std::ostream& operator<<(std::ostream& os, Resolver&
r);
};


#include "Resolver.h"

std::ostream& operator<<(std::ostream& os, Resolver& r)
{
    os << r._MaxStream << std::endl;
    return os << r._Result;
}

void Resolver::Resolve(Graph& g, char in, char out)
{
    // принимаем данные
    _Graph = g;
    _In = in;
    _Out = out;

    // создаем копию исходного графа, но с 0 весами
    _Result.Reset(_Graph.EdgesCount());
    for (auto listI : _Graph) {
     for (auto edge : listI.second) {
          _Result.Set(Edge(edge.From(), edge.To(), 0));
     }
    }

    // сканируем все пути
    _Path.clear();
    Go(in);

    // поиск макс потока из истока в сток (сумма весов входящих
ребер в сток из результата)
    _MaxStream = 0;
    for (auto listI : _Result) {
     for (auto edge : listI.second) {
          // пропуск ненужных ребер
          if (edge.To() != _Out) continue;
          // суммируем вес
          _MaxStream += edge.Weight();
     }
```

```
    }
}

void Resolver::Go(char point)
{
    // ограничитель
    if (Contains(_Path, point)) return;

    // добавляем элемент в путь
    _Path.push_back(point);

    // если дошли до конца
    if (point == _Out) {
     CheckPath();
    }
    else {
     for (auto e : _Graph[point]) Go(e.To());
    }

    // удаляем элемент из пути
    _Path.pop_back();
}

bool Resolver::Contains(std::vector<char>& path, char point)
{
    for (auto i : path)
     if (i == point) return true;
    return false;
}
void Resolver::CheckPath()    // проверяет найденый путь (по
алгоритму)
{
    // поиск мин пропускной способности на пути
    unsigned int minW = 0;
    for (int i = 1; i < _Path.size(); ++i) {
     auto& e = _Graph.Get(_Path[i - 1], _Path[i]);
     if (minW == 0 || e.Weight() < minW) minW = e.Weight();
    }

    // прибавляем этот мин ко всему на результирующем графе
    for (int i = 1; i < _Path.size(); ++i) {
     auto& e = _Result.Get(_Path[i - 1], _Path[i]);
     e.SetWeight(e.Weight() + minW);
    }
}
```

Graph.h

```
#pragma once
#include <iostream>
#include <map>
#include <vector>
#include "Edge.h"

// граф
```

```
class Graph
{
    std::map<char, std::vector<Edge>> _Data; // набор ребер на
каждый узел
    int m_EdgesCount;
public:

    int EdgesCount() { return m_EdgesCount; }
    void Reset(int edgesCount) { _Data.clear(); m_EdgesCount =
edgesCount; }

    std::vector<Edge> operator[](char node); // возвращает
сортированный по весам вектор ребер из указанной вершины
    void Set(Edge e);    // задает ребро
    Edge& Get(char from, char to); // возаращает ссылку на ребро

    std::map<char, std::vector<Edge>>::iterator begin() { return
_Data.begin(); }
    std::map<char, std::vector<Edge>>::iterator end() { return
_Data.end(); }

    friend std::istream& operator>>(std::istream& is, Graph& g);
    friend std::ostream& operator<<(std::ostream& os, Graph& g);
};
```

Graph.cpp

```
#include "Graph.h"
#include <algorithm>

std::istream& operator>>(std::istream& is, Graph& g) {
    // ввод всех ребер, засовывая их в нужные вектора
    Edge e;
    for(int i=0;i<g.m_EdgesCount;++i){
     is >> e;
     g._Data[e.From()].push_back(e);
    }

    // сортировка всех векторов
    for (auto i : g._Data) {
     std::sort(i.second.begin(), i.second.end());
     g._Data[i.first] = i.second;
    }

    // вывод результатов
    return is;
}
std::ostream& operator<<(std::ostream& os, Graph& g) {
    for (auto i : g._Data)
     for (auto j : i.second)
          os << j << std::endl;
    return os;
}
```

```
std::vector<Edge> Graph::operator[](char node){ // возвращает
сортированный по весам вектор ребер из указанной вершины
    return _Data[node];
}
void Graph::Set(Edge e)  // задает ребро
{
    // берем список исходящих ребер
    auto list = _Data[e.From()];
    // пытаемся заменить ребро (если ребро существует)
    for (int i = 0; i < list.size(); ++i) {
     // пропускаем ненужные ребра
     if (list[i].From() != e.From() || list[i].To() != e.To())
continue;
     // перезапись ребра
     list[i] = e;
     // завершаем алгоритм
     return;
    }
    // вставка ребра вконец (если ребра еще нет)
    list.push_back(e);
    _Data[e.From()] = list;
}
Edge& Graph::Get(char from, char to)    // возаращает ссылку на
ребро
{
    auto& list = _Data[from];
    for (int i = 0; i < list.size(); ++i) {
     if (list[i].To() == to) return list[i];
    }
    throw "could not found edge";
}
```

Finder.h

```
#pragma once
#include "Graph.h"

class Finder
{
public:
    bool Find(Graph& g, char item); // ищет в указанном графе
указанное значение
};
```

Finder.cpp

```
#include "Finder.h"
#include <set>
#include <vector>

bool Finder::Find(Graph& g, char item) // ищет в указанном графе
указанное значение
```

```
{
    std::cout << "Start finding " << item << "..." << std::endl;
    // создаем набор посещенных вершин
    std::set<char> visited;
    // создаем текущий путь
    std::vector<char> path;

    // запускаем цикл поиска в глубину
    while (true) {
     // берем первую попавшуюся вершину, которую не посещали и
помещаем ее в начало пути
     bool findFirst = false;
     for (auto i : g) {
          for (auto e : i.second) {
               // пропускаем все посещенные вершины
               if (visited.find(e.To()) != visited.end()) continue;
               // вставка вершины в начало пути
               path.push_back(e.To());
               visited.insert(e.To());
               std::cout << e.To();
               // если посетили искомую то выводим результат
               if (e.To() == item) {
                    std::cout << std::endl;
                    return true;
               }
               // завершаем цикл
               findFirst = true;
               break;
          }
          if (findFirst) break;
     }
     // если в путь начинать не от куда то все перебрали
     //if (!findFirst) break;

     // из первой вершины посещаем все возможные
     while (path.size()) {
          // пытаемся посещать вершины, пока это возможно
(пропростаем вглубину)
          auto hasVisit = false;
          auto edges = g[path[path.size() - 1]];
          for (auto i = edges.begin(); i != edges.end();) {
               // если вершину посещали то не переходим по ребру
(пропускаем ребро)
               if (visited.find(i->To()) != visited.end()) {
                    ++i;
                    continue;
               }
               // посещаем вершину
               path.push_back(i->To());
               visited.insert(i->To());
               std::cout << i->To();
               hasVisit = true;
               // если посетили искомую то выводим результат
               if (i->To() == item) {
                    std::cout << std::endl;
```

```
                    return true;
                }
                // получаем итератор для посещенной вершины
                edges = g[path[path.size() - 1]];
                i = edges.begin();
            }

            // откат назад
            path.pop_back();
            if (hasVisit)std::cout << std::endl;
        }
    }

    // говорим, что ничего не нашли
    return false;
}
```

Edge.h

```
#pragma once
#include <iostream>

// ребро графа
class Edge
{
    char _From;
    char _To;
    unsigned int _Weight;
public:

    Edge();
    Edge(char from, char to, unsigned int weight);

    char From();         // откуда
    char To();           // куда
    unsigned int Weight();    // с каким весом
    void SetWeight(int weight) { _Weight = weight; }

    friend std::istream& operator>>(std::istream& is, Edge& e);
    friend std::ostream& operator<<(std::ostream& os, Edge& e);
    friend bool operator<(const Edge& a, const Edge& b);
    friend bool operator>(const Edge& a, const Edge& b);
};
```

Edge.cpp

```
#include "Edge.h"


char Edge::From() {
    return _From;
```

```
}
char Edge::To() {
    return _To;
}
unsigned int Edge::Weight() {
    return _Weight;
}

std::istream& operator>>(std::istream& is, Edge& e) {
    return is >> e._From >> e._To >> e._Weight;
}
std::ostream& operator<<(std::ostream& os, Edge& e) {
    return os << e._From << "->" << e._To << ": " << e._Weight;
}
bool operator<(const Edge& a, const Edge& b) {
    return a._Weight < b._Weight;
}
bool operator>(const Edge& a, const Edge& b) {
    return a._Weight > b._Weight;
}
Edge::Edge() {
    _From = 0;
    _To = 0;
    _Weight = 0;
}
Edge::Edge(char from, char to, unsigned int weight) {
    _From = from;
    _To = to;
    _Weight = weight;
}
```