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

Студент гр. 9383 Рыбников Р.А.

Преподаватель Фирсов М.А.

Санкт-Петербург

2021

Цель работы

Изучить алгоритм Форда-Фалкерсона – поиска максимального потока всети. 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, нужно создать граф. Граф представляет собой массив рёбер. Для работы алгоритма, все потоки должны быть равны 0 изначально.

Поиск пути в графе происходит поиском в глубину (не вариативное задание).

Суть алгоритма в том, что если нашелся путь от начала до конца графа(исток/сток), то в этом найденном пути ищется максимальный поток. Найденное значение максимального потока для данного пути прибавляется к общему конечному значению максимального потока всего графа в целом.

Для прямых ребер поток увеличивается на найденную величину, для обратных ребер поток уменьшается на найденную величину.

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

Если в графе величина пропускной способности ребра – это иррациональное число, то алгоритм может не сойтись и на бесконечности. Проблема отсутствует при целых числах.

Описание функций и структур данных

- class Edge – класс ребра графа, который имеет поля истока, стока и веса ребра. Так же в этом классе реализованы геттеры Start(), End(), W() для получения значений стартовой/финальной вершины графа, а так же веса ребра. Сеттер SetW() – позволяет установить новое значение веса ребра.
- class Orgraph – класс, который хранит в себе структуру ориентированного графа. Базируется на векторе рёбер. Имеет геттер EdgesCount(), который возвращает число рёбер. Так же в этом классе перегружен оператор [], чтобы была возможность обращаться к ребру по его индексу. Присутствует сеттер SetEdgesCount(), который позволяет задать новое количество рёбер графа.
- class FF – представляет собой реализацию алгоритма Форда-Фалкерсона. Рассматриваемый путь хранится в вектора рёбер m_Path. Просмотренные вершины хранятся в векторе char-ов m_Visited, используется для реализации обхода графа. В классе реализованы следующие методы: void Visit(edge) – выполняет обход графа, void MyPath() – сканирует выбранный путь графа, uncignet int GetMinW() – возвращает минимальную пропускную способность текущего пути.

Тестирование

```
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

Program ended with exit code: 0
```

Рисунок 1 – Проверка программы №1

```
1
a f
f a 9
-------
0
f a 0

Program ended with exit code: 0
```

Рисунок 2 – Проверка программы №2

```
3
a c
a b 7
a c 6
b c 4
-------
10
a b 4
a c 6
b c 4

Program ended with exit code: 0
```

Рисунок 3 – Проверка программы №3

Вывод

В ходе выполнения лабораторной работы были применены на практике сведения об алгоритме Форда-Фалкерсона, а именно реализован алгоритм на Языке Программирования C++. Проведены тестовые запуски программы. Поставленная задача выполнена.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp:

```
#include <iostream>
#include <sstream>
#include "Orgraph.h"
#include "Resolver.h"


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

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

	// вывод графа
	//cout << g << endl;

	// поиск решения
   FF go;
	go.Init(g, in, out);
   std::cout << "-------" << std::endl;
	// вывод результата
   std::cout << go << std::endl;

   return 0;
}
```

FF.h

```
#pragma once
#include "Orgraph.h"
#include <set>
#include <vector>

class FF
{
	int m_MaxStream;
	Orgraph m_Graph;
	Orgraph m_Res;
	std::set<char> m_Visited;
	std::vector<Orgraph::Edge> m_Path;	// текущий путь

	char m_In;
	char m_Out;
```

```
	void Visit(Orgraph::Edge e);
	void MyPath();		// производит анализ одного найденого пути
	unsigned int GetMinW();
public:
	void Init(Orgraph& g, char in, char out);

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

FF.cpp

```
#include "FF.h"

std::ostream& operator<<(std::ostream& os, FF& r)
{
	os << r.m_MaxStream << std::endl;
	return os << r.m_Res;
}
void FF::Init(Orgraph& g, char in, char out)
{
	// принимаем данные
	m_Graph = g;
	m_In = in;
	m_Out = out;
	// создаем копию графа с нулевыми ребрами
	m_Res.SetEdgesCount(g.EdgesCount());
	for (auto i : g) m_Res.Set(Orgraph::Edge(i.Start(), i.End(), 0));

	// поиск макс потоков по всей сети (по всему графу)
	Visit(Orgraph::Edge(0, in, 0));

	// поиск макс потока (те сумма весов входных ребер в выходной узел)
	m_MaxStream = 0;
	for (auto i : m_Res) {
		if (i.End() != out) continue;
		m_MaxStream += i.W();
	}
}


void FF::Visit(Orgraph::Edge e)
{
	if (m_Visited.find(e.End()) != m_Visited.end()) return;
	m_Visited.insert(e.End());
	m_Path.push_back(e);

	// если дошли до конечной вершины
	if (e.End() == m_Out) MyPath();
	else {
		// перебор всех исходящих вершин
		for (auto e : m_Graph[e.End()])
			Visit(e);
	}

	m_Visited.erase(e.End());
	m_Path.pop_back();
}
void FF::MyPath() // производит анализ одного найденого пути
{
	// берем мин пропускную способность на текущем пути
```

```
	auto minW = GetMinW();
	// добавляем эту мин пропускную способность на все ребра результирующего графа, по
которому идет путь
	for (auto e : m_Path) {
		// пропуск стартового перехода из пустоты
		if (e.Start() == 0) continue;
		// добавляем вес
		auto edge = m_Res.Get(e.Start(), e.End());
		edge.SetW(edge.W() + minW);
		m_Res.Set(edge);
	}
}
unsigned int FF::GetMinW()	// находит мин пропускную способность из текущего пути
{
	unsigned int min = 0;
	bool hasMin = false;
	for (auto e : m_Path) {
		// пропуск стартового перехода из пустоты
		if (e.Start() == 0) continue;
		// поиск
		if (!hasMin || e.W() < min) {
			min = e.W();
			hasMin = true;
		}
	}
	return min;
}
```

Orgraph.h

```
#pragma once
#include <iostream>
#include <vector>

class Orgraph
{
public:
	class Edge;
private:
	std::vector<Edge> m_Edges;
	int m_EdgesCount;
public:

	int EdgesCount() { return m_EdgesCount; }
	Edge& operator[](int index) { return m_Edges[index]; }
	std::vector<Edge> operator[](char node);		// возвращает все исходящие ребра из указанной
вершины, причем все исходящие вершины сортируются
	void Set(Edge e);
	Edge& Get(char start, char end);				// получает ребро из графа
	void SetEdgesCount(int count);	// задает новое количество ребер

	std::vector<Edge>::iterator begin() { return m_Edges.begin(); }
	std::vector<Edge>::iterator end() { return m_Edges.end(); }

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

class Orgraph::Edge {
	char m_Start;		// откуда
	char m_End;			// куда
```

```
	unsigned int m_W;	// вес
public:
	Edge() {
		m_Start = 0;
		m_End = 0;
		m_W = 0;
	}
	Edge(char start, char end, float w) {
		this->m_Start = start;
		this->m_End = end;
		this->m_W = w;
	}

	const char Start() { return m_Start; }
	const char End() { return m_End; }
	const unsigned int W() { return m_W; }
	void SetW(unsigned int value) {
		m_W = value;
	}

	friend std::istream& operator>>(std::istream& is, Edge& e);
	friend std::ostream& operator<<(std::ostream& os, Edge& e);
	friend bool operator<(const Orgraph::Edge& e1, const Orgraph::Edge& e2);
	friend bool operator>(const Orgraph::Edge& e1, const Orgraph::Edge& e2);
};

inline bool operator<(const Orgraph::Edge& e1, const Orgraph::Edge& e2) {
	return e1.m_W < e2.m_W;
}
inline bool operator>(const Orgraph::Edge& e1, const Orgraph::Edge& e2) {
	return e1.m_W > e2.m_W;
}
```

Orgraph.cpp

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

std::istream& operator>>(std::istream& is, Orgraph& g)
{
	Orgraph::Edge e;
	g.m_Edges.clear();
	for(int i=0;i< g.m_EdgesCount;++i){
		is >> e;
		g.m_Edges.push_back(e);
	}
	return is;
}
std::ostream& operator<<(std::ostream& os, Orgraph& g) {
	for (auto i = g.m_Edges.begin(); i < g.m_Edges.end(); ++i)
		os << *i << std::endl;
	return os;
}
std::istream& operator>>(std::istream& is, Orgraph::Edge& e) {
	return is >> e.m_Start >> e.m_End >> e.m_W;
}
std::ostream& operator<<(std::ostream& os, Orgraph::Edge& e) {
	return os << e.m_Start << ' ' << e.m_End << ' ' << e.m_W;
}
std::vector<Orgraph::Edge> Orgraph::operator[](char node)
{
```

```
	std::vector<Edge> res;
	for (auto i = m_Edges.begin(); i < m_Edges.end(); ++i) {
		if (i->Start() == node) res.push_back(*i);
	}
	std::sort(res.begin(), res.end(), std::less<Edge>());
	return res;
}

void Orgraph::Set(Edge e) {
	// попытка записать заного ребро
	for (int i = 0; i < m_Edges.size(); ++i) {
		if (m_Edges[i].Start() == e.Start() && m_Edges[i].End() == e.End()) {
			m_Edges[i] = e;
			return;
		}
	}
	// вставка вконец
	m_Edges.push_back(e);
}
void Orgraph::SetEdgesCount(int count) {
	m_Edges.clear();
	m_EdgesCount = count;
}
Orgraph::Edge& Orgraph::Get(char start, char end) {
	for (int i = 0; i < m_Edges.size(); ++i) {
		if (m_Edges[i].Start() == start && m_Edges[i].End() == end)
			return m_Edges[i];
	}

	throw "error";
}
```