**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: Поток в сети**

Студент гр. 7383 _____ Лосев М.Л.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2018

**Постановка задачи.**

**Цель работы:** реализация и исследование алгоритма Форда-Фалкерсона, получение опыта и знания по его использованию. Формулировка задачи:

**Формулировка задачи:** Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

**Индивидуализация:** для поиска пути использовать поиск в ширину.

**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

…

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).

**Описание алгоритма:**

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим *кратчайший* путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется **увеличивающим путём** или **увеличивающей цепью**) максимально возможный поток:

1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью $c$.
2. Для каждого ребра на найденном пути увеличиваем поток на $c$, а в противоположном ему — уменьшаем на $c$.
3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.

4. Возвращаемся на шаг 2.

Чтобы найти кратчайший путь в графе, используем поиск в ширину:

1. Создаём очередь вершин $O$. Вначале $O$ состоит из единственной вершины $s$.
2. Отмечаем вершину $s$ как посещённую, без родителя, а все остальные как непосещённые.
3. Пока очередь не пуста, выполняем следующие шаги:
    1. Удаляем первую в очереди вершину $u$.
    2. Для всех дуг $(u, v)$, исходящих из вершины $u$, для которых вершина $v$ ещё не посещена, выполняем следующие шаги:
        1. Отмечаем вершину $v$ как посещённую, с родителем $u$.
        2. Добавляем вершину $v$ в конец очереди.
        3. Если $v = t$, выходим из обоих циклов: мы нашли кратчайший путь.
4. Если очередь пуста, возвращаем ответ, что пути нет вообще и останавливаемся.
5. Если нет, идём от $t$ к $s$, каждый раз переходя к родителю. Возвращаем путь в обратном порядке.

**Реализация**

Был использован следующие классы:

Класс Edge:

Поля:

ver_type first;
ver_type last;
flow_type capacity;
flow_type flow;
flow_type rev_flow;

Методы:

Edge()
Edge(ver_type first, ver_type last, flow_type capacity)
void print() – вывод ребра на экран
bool empty() – возвращает true тогда и только тогда когда ребро не пусто
flow_type max_flow() – возвращает максимальный поток, который может пройти по ребру
int first_index() – возвращает первый идекс ребра в матрице
int last_index() – возвращает второй идекс ребра в матрице

Класс Vertex:

Поля:

ver_type name;
ver_type prev;
bool visited;Методы:

Методы:

Vertex()
Vertex(ver_type name)
Vertex(ver_type name, ver_type prev)

bool operator < (const Vertex &v2) const
bool operator == (const Vertex &v2) const
– операторы сравнения

Класс FlowNetwork:

Поля:

int num_of_vertexes;
Edge ** adjacency_matrix;
Vertex sourse;
Vertex sink;
vector <Vertex> vertexex;
int num_of_edges;

Методы:

FlowNetwork()
~FlowNetwork()
void input_flow_network() – ввод сети
void print_actual_flow_value() – вывод всех ребер с потоками
string BFS(Vertex start, Vertex target) – поиск у ширину

int FFA – реализация алгоритма Форда - Фалкерсона
void add_edge(const Edge &e) – добавление ребра
Edge input_edge() – ввод одного ребра
void clear_visits() – обнуляет посещенности всех вершин
void zeroize_flows() – обнуляет потоки
flow_type edge_max_flow(const Edge &e) const – возвращает максимальный поток по ребру
flow_type path_max_flow(string path) – возвращает максимальный поток по пути
void add_flow(string path, flow_type flow_val) – добавляет поток

**Исследование**

Пусть |E| - множество ребер графа, |V| – множество его вершин. Алгоритм в худшем случае пройдет каждую вершину. При прохождении каждой вершины в очередь будут добавляться пути через эту вершину в смежные с ней из начальной. Добавление в очередь требует в худшем случае столько сравнений, сколько в очереди элементов. Но элементов в очереди не может быть больше, чем |V|, если не хранить несколько путей в одну и ту же вершину, что бессмысленно (для оптимальности достаточно рассматривать лишь более короткий из них). Получается O(|V|*|V|). Но, с другой стороны, очевидно, что на самом деле эта величина должна зависеть не только от |V|, но и от |E|. Некоторые исследователи полагают, что сложность алгоритма A* составляет O(|V|*|V|*log(|E|).

**Тестирование.**

Тестирование проводилось в ОС Ubuntu 18.04 компилятором GCC. Кроме того, программа прошла тесты на Stepic. Результаты тестирования показали, что программа работает корректно.

**Вывод.**

При выполнении работы был реализован и изучен Эдмондса – Карпа, который является частным случаем алгоритма Форда — Фалкерсона, который использует поиск в ширину для нахождения пути.

## ПРИЛОЖЕНИЕ А

## Код программы

```
#include <iostream>
#include <queue>
#include <string>
#include <vector>
#include <algorithm>  // for_each
#include <cmath>

using namespace std;

template <typename ver_type, typename flow_type>
class FlowNetwork
{
public:
	class Edge
	{
	public:
		ver_type first;
		ver_type last;
		flow_type capacity;
		flow_type flow;
		flow_type rev_flow;

		Edge() {   }
		Edge(ver_type first, ver_type last, flow_type capacity) :
first(first), last(last), capacity(capacity), flow(0), rev_flow(0) {  }
		void print() { cout << first << " " << last << " " << flow <<
endl; }
		bool empty() { return capacity == 0 && flow == 0 ? true :
false; }
		flow_type max_flow() { return capacity - flow; }
		int first_index() const { return first; }
		int last_index() const { return last; }
	};

	class Vertex
	{
	public:
		ver_type name;
		ver_type prev;
		bool visited;

		Vertex() { }
		Vertex(ver_type name) : name(name), visited(false) {  }
		Vertex(ver_type name, ver_type prev) : name(name), prev(prev),
visited(false) {}
		int index() const { return name; }
```

```
          bool operator < (const Vertex &v2) const { return this->name <
v2.name; }
          bool operator == (const Vertex &v2) const {     return this-
>name == v2.name; } // needed to use find func
     };

     int num_of_vertexes;
     Edge ** adjacency_matrix;
     Vertex sourse;
     Vertex sink;
     vector <Vertex> vertexex;
     int num_of_edges;

     FlowNetwork() : num_of_vertexes(256) { // any ascii character can
be the name of a vertex
          for (int i = 0; i < num_of_vertexes; i++) {
               vertexex.push_back(Vertex(i, 0));
          }    // fill vertexes list with vertexes from a to z with no
parent(prev)

          try {
               adjacency_matrix = new Edge* [num_of_vertexes];
               for (int i = 0; i < num_of_vertexes; i++){
                    adjacency_matrix[i] = new Edge [num_of_vertexes];
                    fill(adjacency_matrix[i], adjacency_matrix[i] +
num_of_vertexes, Edge(0, 0, 0));
               }

          }
          catch (...) {
               if (adjacency_matrix) {
                    for (int i = 0; i < num_of_vertexes; i++)
                         if (adjacency_matrix[i])
                              delete [] adjacency_matrix[i];
                    delete [] adjacency_matrix;
                    num_of_vertexes = 0;
                    throw;
               }
          }
     }

     void input_flow_network() {
          ver_type tmp_sourse, tmp_sink;
          cin >> num_of_edges;
          cin >> tmp_sourse;
          cin >> tmp_sink;
          sourse = Vertex(tmp_sourse);
          sink = Vertex(tmp_sink);

          for (int i = 0; i < num_of_edges; i++){
               ver_type tmp_first, tmp_last;
```

```
                flow_type tmp_capacity;
                cin >> tmp_first >> tmp_last >> tmp_capacity;
                Edge e(tmp_first, tmp_last, tmp_capacity);
                adjacency_matrix[e.first_index()][e.last_index()] = e;
          }
     }

     void print_actual_flow_value() {
          if (adjacency_matrix)
                for (int i = 0; i < num_of_vertexes; i++)
                     if (adjacency_matrix[i])
                          for (int j = 0; j < num_of_vertexes; j++)
                               if (!adjacency_matrix[i][j].empty())
                                    adjacency_matrix[i][j].print();
     }

     ~FlowNetwork() {
          if (adjacency_matrix) {
                for (int i = 0; i < num_of_vertexes; i++)
                     if (adjacency_matrix[i])
                          delete [] adjacency_matrix[i];
                delete [] adjacency_matrix;
          }
          num_of_vertexes = 0;
     }

     string BFS(Vertex start, Vertex target) {
          clear_visits();
                                                                  // set
no parenst and no visits
          queue <Vertex> ver_queue;
          ver_queue.push(start);
                                                  // 0 means no parent
          bool tag = true;
          while (!ver_queue.empty() && tag){
                Vertex curr_ver = ver_queue.front();
                ver_queue.pop();

                for (int i = 0; i < num_of_vertexes; i++) {
                                                                  // for each
vertex
                     Edge dir_edge =
adjacency_matrix[curr_ver.index()][i];
                // curr_edge
                     Edge rev_edge =
adjacency_matrix[i][curr_ver.index()];

                     //calculate max flow through the edge:
                     flow_type max_flow = 0;
                     if (!dir_edge.empty())
                          max_flow += dir_edge.max_flow();
```

```
                    if (!rev_edge.empty())
                        max_flow += rev_edge.flow;

                    if (max_flow > 0) {
                        Vertex tmp_next;

                        if (!dir_edge.empty())
    // if the flow goes through the direct edge or both edges
                            tmp_next = Vertex(dir_edge.last);
                        else
    // if the flow goes through the reverse edge
                            tmp_next = Vertex(rev_edge.first);

                        auto next_ver = find(vertexex.begin(),
vertexex.end(), Vertex(tmp_next));
                        if (!next_ver->visited && max_flow != 0) {

                            next_ver->visited = true;
                            next_ver->prev = curr_ver.name;
                            ver_queue.push(*next_ver);
                                                    // push the
next_ver to queue with prev = curr
                            if (*next_ver == target) {
                                                    // if a way
have been found
                                tag = false;
                                break;
                                                                //
leave both cycles
                            }
                        }
                    }
                }
            }

            if (ver_queue.empty())              // no path case
                return string("");
            // restore the path:
            auto curr_ver = find(vertexex.begin(), vertexex.end(),
Vertex(target));
            string path("");
            path = curr_ver->name + path;;
            while (curr_ver != find(vertexex.begin(), vertexex.end(),
Vertex(start))) {
                curr_ver = find(vertexex.begin(), vertexex.end(),
Vertex(curr_ver->prev));
                path = curr_ver->name + path;
            }

            return path;
        }
```

```
	int FFA() {
		zeroize_flows();

		sourse.visited = true;
		flow_type common_flow = 0;

		while (true) {
			string path = BFS(sourse, sink);
			auto flow_val = path_max_flow(path);
			common_flow += flow_val;
			add_flow(path, flow_val);
			path = BFS(sourse, sink);
			if (path == "") break;
		}
		return common_flow;
	}

private:
	void add_edge(const Edge &e) {
		adjacency_matrix[e.first_index()][e.last_index()] = e;
	}

	Edge input_edge() {
		ver_type tmp_first, tmp_last;
		flow_type tmp_capacity;
		cin >> tmp_first >> tmp_last >> tmp_capacity;
		return Edge(tmp_first, tmp_last, (flow_type)tmp_capacity);
	}

	void clear_visits() { // for BFS
		for (int i = 0; i < num_of_vertexes; i++) {
			vertexex[i].visited = false;
			vertexex[i].prev = 0;
		}
	}	// delete parents and visits

	void zeroize_flows() {
		for (int i = 0; i < num_of_vertexes; i++)
			for (int j = 0; j < num_of_vertexes; j++)
				if (!adjacency_matrix[i][j].empty())
					adjacency_matrix[i][j].flow = 0;
	}	// set all the flows zero

	flow_type edge_max_flow(const Edge &e) const {
		return edge_max_flow(Edge(e.first, e.last, 0));
	}

	flow_type edge_max_flow(ver_type first, ver_type last) const {
		Edge dir_edge =
adjacency_matrix[Vertex(first).index()][Vertex(last).index()];
```

```
            Edge rev_edge =
adjacency_matrix[Vertex(last).index()][Vertex(first).index()];

            flow_type max_flow = 0;
            if (!dir_edge.empty())
                  max_flow += dir_edge.max_flow();
            if (!rev_edge.empty())
                  max_flow += rev_edge.flow;

            return max_flow;
      }

      flow_type path_max_flow(string path) {
            flow_type max_flow = edge_max_flow(path[0], path[1]);
            for (int i = 0; i < path.length() - 1; i++) {
                  flow_type curr_flow = edge_max_flow(path[i], path[i +
1]);
                  if (curr_flow < max_flow)
                        max_flow = curr_flow;
            }
            return max_flow;
      }

      void add_flow(string path, flow_type flow_val) {
            for (int i = 0; i < path.length() - 1; i++) {         // for
each vertex in the path
                  Edge* dir_edge =
&adjacency_matrix[Vertex(path[i]).index()][Vertex(path[i + 1]).index()];
                  Edge* rev_edge = &adjacency_matrix[Vertex(path[i +
1]).index()][Vertex(path[i]).index()];
                  flow_type max_flow = edge_max_flow(path[i], path[i +
1]);
                  flow_type tmp_flow = flow_val;

                  if (!rev_edge->empty())
                        if (rev_edge->flow <= tmp_flow) {
                              tmp_flow -= rev_edge->flow;
                              rev_edge->flow = 0;
                        } // first part of a flow: through the reverse edge
                  if (!dir_edge->empty() && tmp_flow != 0)
                        if (dir_edge->max_flow() >= tmp_flow) {
                              dir_edge->flow += tmp_flow;
                              tmp_flow = 0;
                        } // second part of a flow: through the direct edge
                  if (tmp_flow != 0) // as a precaution
                        cout << "ERROR: sum.(rev + dir) capacity >
flow_val" << endl;
            }
      }

};
```

```
int main()
{
	FlowNetwork <char, int> g;
	g.input_flow_network();
	cout << g.FFA() << endl;
	g.print_actual_flow_value();

	return 0;
}
```