МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студентка гр. 7383 _____ Чемова К.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Цель работы.

Реализовать и исследовать алгоритм Форда-Фалкерсона поиска максимального потока в сети.

Выполнение работы.

Для представления сети используется класс, представляющий из себя список смежности. В списке хранятся ребра и поток через эти ребра. Метод read осуществляет считывание данных с клавиатуры и заполняет список смежности. После вызывается метод maxFlow, вызывающий методы find_way и change. find_way находит пусть в сети по правилу: каждый раз берется дуга, имеющая максимальную остаточную пропускную способность. Когда найден путь, вызываем метод change, который находит с помощью метода min_flow находит ребро на данном пути с минимальной остаточной пропускной способностью, которое в данном случае будет максимальным потоком для пути. Далее на всем данном пути пропускная способность дуг уменьшается на это значение, а также создается обратное ребро, оно направлено в другую сторону и на нем остаточная пропускная способность увеличивается. Если остаточная пропускная способность ребра равна максимальному потоку на пути, то ребро удаляется, так как через него уже никакой поток пройти не сможет. Алгоритм заканчивает работу, когда невозможно найти путь из истока в сток.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04 с использованием компилятора g++. В других ОС и компиляторах тестирование не проводилось.

Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении А.

Исследование алгоритма

На каждом шаге алгоритм добавляет поток увеличивающего пути к уже имеющемуся потоку. Следовательно, на каждом шаге алгоритм увеличивает

поток по крайней мере на единицу, следовательно, он сойдётся не более чем за O(f) шагов, где f – максимальный поток в графе. Можно выполнить каждый шаг за время O(E), где |E| – число рёбер в графе, тогда общее время работы алгоритма ограничено O(f*|E|).

Выводы.

В ходе лабораторной работы был изучен алгоритм Форда-Фалкерсона поиска максимального потока в сети. Была был написана программа на языке C++, реализующая данный алгоритм; исследована сложность алгоритма, по результатам сложность равна O(f*|E|).

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Результат
7 a f a b 8 b c 4 c d 3 d e 2 e b 1 e f 1 b f 6	7 a f a b 8 b c 4 c d 3 d e 2 e b 1 e f 1 b f 6
16 a e a b 2 b a 2 a d 1 d a 1 a c 3 c a 3 b c 4 c b 4 c d 1	6 a b 2 a c 3 a d 1 b a 0 b c 0 b e 3 c a 0 c b 1 c d 0 c e 2 d a 0
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

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <map>
#include <iterator>
#include <algorithm>
#include <limits>
using namespace std;

class Flow{
private:
  map <char, map<char, int>> graph;
  char u;
  char v;
public:
  Flow(){}

  void read(){
    char a, b;
    int n, w;
    cin >> n >> u >> v;
    for(int i = 0; i < n; i++){
      cin >> a >> b >> w;
      graph[a][b] = w;
    }
    maxFlow(u, v);
  }

  map <char, pair <char, int>> find_way(map <char, map <char, int>> m, char start,
char finish){
    map <char, pair<char, int>> way;
    map <char, bool> visited;
    vector <char> path; //переименовать стек
    typename map<char, map<char, int>>::iterator it;
    typename map<char, map<char, int>>::iterator t;
    for (it = m.begin(); it != m.end(); it++)
      visited[it->first] = false;
    path.push_back(start);
    it = graph.begin();
    for ( ; it != m.end(); it++){
      if (path[path.size() - 1] == finish)
        break;
      visited[path[path.size()-1]] = true;
      vector <pair<char, int>> neighborhood;
      for (pair <char, int> neighbor: m.find(path[path.size()-1])->second)
        if (!visited[neighbor.first])
          neighborhood.push_back(neighbor);
        if (neighborhood.empty()){
          if (way.size() == 1 || way.empty()){
            way.clear();
            return way;
          }
          way.erase(path[path.size()-1]);
          path.pop_back();
          continue;
        }
```

```
        pair <char, int> max = *max_element(neighborhood.begin(), neighborhood.end(),
[](pair<char, int>& n1, pair<char, int>& n2){
          return  n1.second < n2.second;
        });//находит набольший вес дуг
        way[path[path.size()-1]] = pair<char, int>(max.first, max.second);
        path.push_back(max.first);
    }
    return way;
  }

  int get_min_flow(map<char, pair<char, int>> way){
    auto t = way.begin();
    int min_flow = t->second.second;
    for (; t != way.end(); t++){
      if (min_flow > t->second.second)
        min_flow = t->second.second;
    }
    return min_flow;
  }

  void change(map<char, map<char, int>> & network, map<char, pair<char, int>> way){
    int min_flow = min_flow(way);
    for (auto t = way.begin(); t!=way.end(); t++){
      if (t->second.second - min_flow == 0)
        network[t->first].erase(t->second.first);
      else
        network[t->first][t->second.first] -= min_flow;
      network[t->second.first][t->first] += min_flow;
    }
  }

  void maxFlow(char source, char drain){
    map<char, map<char, int>> network = graph;
    map<char, pair<char, int>> way = find_way(network, source, drain);
    while (!way.empty()){
      change(network, way);
      way = find_way(network, source, drain);
    }
    int max_flow = 0;
    for_each(graph[source].begin(), graph[source].end(), [&max_flow, &network,
source](pair<char, int> neighbor){
      max_flow += neighbor.second - network[source][neighbor.first];
    });
    cout << max_flow << endl;
    for_each(graph.begin(), graph.end(), [&network](pair<char, map<char, int>>
neighbors){
      for_each(neighbors.second.begin(), neighbors.second.end(), [&network,
neighbors](pair<char, int> ver){
        std::cout << neighbors.first << " " << ver.first << " ";
        if (ver.second - network[neighbors.first][ver.first] >= 0)
         cout << ver.second - network[neighbors.first][ver.first] << endl;
        else
          cout << 0 << endl;
      });
    });
  }
};

int main(){
  Flow a;
  a.read();
```

```
  return 0;
}
```