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. 8383 _____ Федоров И.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Найти максимальный поток в сети и фактические величины потока через каждое ребро с помощью алгоритма Форда-Фалкерсона.

Задание.

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 ребер графа.

υ_0 - исток.

υ_n - сток.

$\upsilon_i \quad \upsilon_j \quad \omega_{ij}$ - ребро графа.

$\upsilon_i \quad \upsilon_j \quad \omega_{ij}$ - ребро графа.

…

Выходные данные:

$P_{max} -$ величина максимального потока.

$\upsilon_i \quad \upsilon_j \quad \omega_{ij}$ - ребро графа с фактической величиной протекающего потока.

$\upsilon_i \quad \upsilon_j \quad \omega_{ij}$ - ребро графа с фактической величиной протекающего потока.

…

В ответе выходные ребра 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).

Sample Input:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Sample Output:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Вариант №1: Поиск в ширину. Поочередная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

Описание алгоритма.

В начале работы алгоритма создается "остаточный" граф, каждое ребро которого имеет остаточную емкость, которая равна пропускной способности ребра за вычетом текущего потока через него. Остаточный граф инициализируется в начале как исходный граф (копируется), т.к. первоначально все потоки через ребра равны 0 и поэтому остаточная емкость будет равна пропускной способности.

Главный цикл алгоритма работает до тех пор, пока существует улучшающий (расширяющий) путь от истока к стоку. Путь может быть любым, в данном случае идет поиск в ширину, в начале рассматриваются вершины в алфавитном порядке. Переход не осуществляется в посещенные вершины, а так же если остаточная емкость ребра равна 0, т.к. в таком случае улучшать нечего и ребро уже "переносит" через себя максимум.

Если путь не найден, алгоритм завершает работу. Иначе, проходя по найденному пути находит расширяющий поток через этот путь, равный минимальной остаточной емкости вдоль пути. После чего это значение вычитается из всех ребер вдоль пути и добавляется вдоль обратных ребер. Поток пути нужно добавлять вдоль обратных ребер потому, что позже может потребоваться отправить поток в обратном направлении (отменить действие). Максимальный поток увеличивается на расширяющий поток, после чего цикл повторяется.

В конце работы, когда больше нет улучшающих путей от истока к стоку, выводится полученный максимальный поток и фактические потоки через все ребра. Фактические потоки ребер равны пропускной способности за вычетом остаточной емкости (то, на сколько мы не улучшили данный путь). Если разность ≤ 0, то фактический поток через данной ребро равен нулю (поток через него не проходит).

Сложность алгоритма по времени будет $O(\text{max\_flow} \cdot E)$, где max_flow - максимальный поток, а E - число ребер, т.к. в худшем случае мы можем добавлять 1 единичный поток на каждой итерации, а так же проходить по всем ребрам.

Сложность алгоритма по памяти будет $O(2E + 2V + V + E) = O(E + V)$, т.к. сам граф хранится как список смежности, кроме того, в процессе работы хранится остаточный граф, найденный путь и очередь посещенных вершин, все в общем виде будет $O(E + V)$.

Описание функций и структур данных

- `using Graph = std::map<char, std::map<char, int>>;`

Граф хранится как список смежности в ассоциативном контейнере `std::map`, который хранит вершину и контейнер смежных с ней ребер (он хранят конечную вершину и пропускную способность). Благодаря такой структуре удобно проходить по соседям каждой из вершин.

- `std::pair<int, bool> findPath(Graph& residGraph,char start,char goal, std::map<char, char>& currPath);`

Функция `findPath()` осуществляет поиск "расширяющего" пути. На вход принимает остаточный граф, сток и исток, а так же ассоциативный контейнер, в который будет записан путь (откуда - куда). Возвращает пару значения, первое из которых - путь полученного пути, второе - логическое значение, показывающее был ли найден путь.

- `int algoFF(Graph& graph, char start, char goal);`

Функция `algoFF()` осуществляет поиск максимального потока с помощью алгоритма Форда-Фалкерсона. Принимает на вход исходный граф, а так же вершины стока и истока. В конце работы выводит полученный максимальный поток и фактические величины протекающих через ребра графа потоки. Возвращает максимальный поток.

- `void readGraph(Graph& graph, std::istream& stream = std::cin);`

Функция `readGraph()` осуществляет считывание графа. Принимает на вход ссылку на граф, в который будут записывать ребра и вершины, а также поток `std::istream`, откуда будет осуществлять ввод. По умолчанию поток равен стандартному `std::cin`. После считывания вызывает функция поиска максимального потока `algoFF()`.

Выводы.

В ходе работы была рассмотрена задача поиска максимального потока в заданной сети и фактических величин потоков через каждое ребро. Был рассмотрен и реализован решающий данную задачу алгоритм Форда-Фалкерсона.

Тестирование.

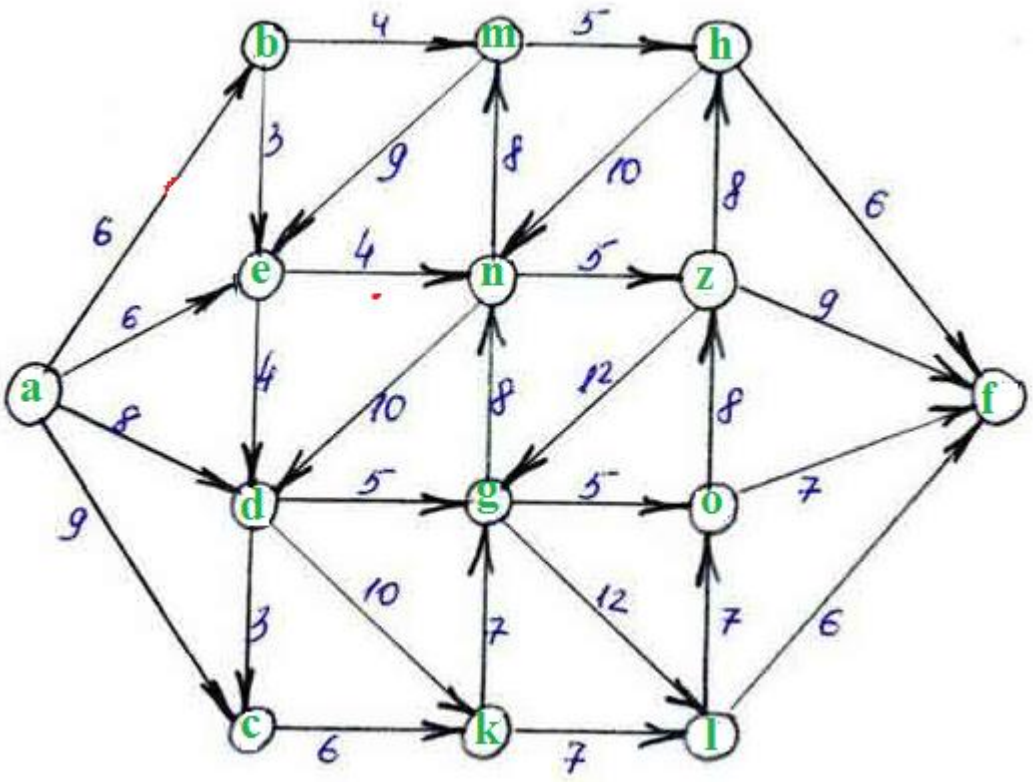
Таблица 1 - тестирование алгоритма

Входные данные	Результат
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
5 s t s a 1 s b 1 b t 1 a t 1 a b 1	2 a b 0 a t 1 b t 1 s a 1 s b 1
10 a f a b 16 a c 13 c b 4 b d 12 b c 10	23 a b 12 a c 11 b c 0 b d 12 c b 0 c e 11 d c 0

c e 14 e d 7 d c 9 d f 20 e f 4	d f 19 e d 7 e f 4
1 a f a f 100	100 a f 100
31 a f a b 6 a e 6 a d 8 a c 9 b m 4 b e 3 e n 4 e d 4 d k 10 d g 5 d c 3 c k 6 m e 9 m h 5 n m 8 n z 5 n d 10 g o 5	26 a b 6 a c 6 a d 8 a e 6 b e 2 b m 4 c k 6 d c 0 d g 5 d k 7 e d 4 e n 4 g l 6 g n 0 g o 5 h f 4 h n 0 k g 6 k l 7 l f 6

g l 12	l o 7
g n 8	m e 0
k g 7	m h 4
k l 7	n d 0
h f 6	n m 0
h n 10	n z 4
z h 8	o f 7
z f 9	o z 5
z g 12	z f 9
o f 7	z g 0
o z 8	z h 0
l o 7	
l f 6	

Последний граф имеет вид:



Приложение

```
#include <iostream>
#include <map>
#include <deque>
#include <queue>
#include <stack>
#include <climits>
#include <algorithm>
#include <fstream>

using Graph = std::map<char, std::map<char, int>>;          //граф
хранится как список смежности
std::string LINE = "----------------------------------------\n";

// поиск улучшающего пути
std::pair<int, bool> findPath(Graph & residGraph, char start, char goal,
std::map<char, char>& currPath){
  // контейнер посещенных вершин
  std::map<char, bool> visitVertex;
  char current;
  currPath.clear();

  // сначала выбирается вершина в алфавитном порядке
  // поэтому смежные вершины кладутся в очередь с приоритетом
  auto compare = [&current] (char a, char b) {
    return a > b;
    //return residGraph[current][a] < residGraph[current][b];
  };
  std::priority_queue <char, std::vector<char>, decltype(compare)>
pathVertex(compare);

  // сначала только стартовая
  pathVertex.push(start);
  visitVertex[start] = true;

  //пока не просмотрели все вершины
  while (!pathVertex.empty()){
    current = pathVertex.top();
    pathVertex.pop();
    visitVertex[current] = true;
    if (current == goal) break;

    //просматриваем соседей и кладем в очередь
    for (auto& neighbour: residGraph[current]){
      // пропускаются посещенные вершины и пути с нулевой остаточной
      // стоимостью, т.к. такой путь нельзя улучшить
         if (!visitVertex[neighbour.first] && neighbour.second > 0){
             pathVertex.push(neighbour.first);
          currPath[neighbour.first] = current;
          visitVertex[neighbour.first] = true;
```

```
            }
        }
    }

    // минимальной стоимости присвоить макс. значение.
    int minRes = std::numeric_limits<int>::max();

    // если нашли путь, то найдем минимальное значение на нем
    if (visitVertex[goal]){
        current = goal;
        char parr;
        while(current != start){
            parr = currPath[current];
            if (minRes > residGraph[parr][current])
                minRes = residGraph[parr][current];
            current = currPath[current];
        }
    }

    std::pair<int, bool> res(minRes, visitVertex[goal]);

    return res;

}



//алгоритм Форда-Фалкерсона
int algoFF(Graph& graph, char start, char goal)
{                                       // "остаточный граф", в начале равен
исходному
    Graph residGraph = graph;           // остаточные емкости = пропусным
способностям
    std::map<char, char> currPath;      // текущий путь сын-родитель
    std::pair<int, bool> findRes;       // результат поиска улучшающего пути
    std::string strPath = "";
    char current, parrent;
    long maxFlow = 0;                   // максимальный поток
    bool cycle = true;
    findRes = findPath(residGraph, start, goal, currPath);

    //пока выходит найти улучшающий путь в "остаточном" графе

    while (findRes.second){
       strPath = "";
       strPath += goal;
       current = goal;

       // по найденному пути обновляем остаточные мощности в
       // "остаточном" графе
       while (current != start){
```

```
            parrent = currPath[current];                    //вычитаем
поток пути из ребер
            residGraph[current][parrent] += findRes.first;  //вдоль пути и
+ вдоль обратных
            residGraph[parrent][current] -= findRes.first;
            std::string add;
            add = parrent; add += " ---> ";
            strPath.insert(0, add);
            current = parrent;
        }

      // обновляем максимальный поток
       maxFlow += findRes.first;


       if (true){
           std::cout << LINE;
           std::cout << "Найденный путь: " << strPath << "\nМинимальный
вес: " << findRes.first << "\tтекущий поток: " << maxFlow << std::endl;
       }
       findRes = findPath(residGraph, start, goal, currPath);

    }

    int resFlow;

    std::cout << LINE << "Максимальный поток сети = " << maxFlow <<
std::endl;
    std::cout << "Фактические величины потоков через ребра: " <<
std::endl;


    // проходим по всем ребрам исходного и остаточного графа и
    // выводим фактические потоки через них равные = пропускной
способности - остаточные емкости
    // т.к. в остаточном графе емкости ребер - на сколько не улучшили в
итоге
    for (auto& vert: graph){
      for (auto adjEdge: graph[vert.first]) {
        if (adjEdge.second < residGraph[vert.first][adjEdge.first]){
            resFlow = 0;
        }
        else{
            resFlow = adjEdge.second -
residGraph[vert.first][adjEdge.first];
        }

        std::cout << vert.first << " " << adjEdge.first << " " << resFlow
<< std::endl;

      }
    }
```

```
    return maxFlow;
}



// считывание графа
void readGraph(Graph& graph, std::istream& stream = std::cin){
    int count, capacity;
    char start, goal;
    char parrent, current;
    stream >> count >> start >> goal;
    for (int i = 0; i < count; i++) {
//Считывание вершин графа
       stream >> parrent >> current >> capacity;
       graph[parrent][current] = capacity;
    }

    algoFF(graph, start, goal);
}


int main()
{
    Graph graph;
    bool file = true;
    std::string fileName = "test1.txt";
    std::ifstream fin;
    fin.exceptions(std::ifstream::badbit | std::ifstream::failbit);

    if (file) {
      try
      {
        fin.open(fileName);
        readGraph(graph, fin);
        fin.close();
      }
      catch(const std::ifstream::failure& exep)
      {
          std::cout << exep.what() << std::endl;
          //std::cout << exep.code()  << std::endl;
          readGraph(graph);
      }
    }
    else{
       readGraph(graph);
    }

    return 0;
}
```