**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.8304 \_\_\_\_\_\_\_\_\_\_\_\_\_ Холковский К.В.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_ Размочаева Н.В.

Санкт-Петербург

2019

**Задание.**

Вариант 4

Поиск в глубину. Итеративный метод.

**Цель работы.**

Разработать программу, которая находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

**Описание алгоритма.**

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим любой 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:
4. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью {\displaystyle c\_{\min }}.
5. Для каждого ребра на найденном пути увеличиваем поток на {\displaystyle c\_{\min }}, а в противоположном ему — уменьшаем на {\displaystyle c\_{\min }}.
6. 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.
7. Возвращаемся на шаг 2.

Сложность алгоритма O(Ef), где E – количество ребер в графе, f – максимальный поток в графе.

**Описание функций и структур данных.**

Граф хранится в словаре std::map<**char**, elem>, где elem – структура, где хранится информация о предыдущем элементе и о том, присутствует ли данный элемент в пути(Нужно для поиска в глубину итеративным способом) Также в ней хранится информация о вершинах в которые можно попасть из данной и пропускная способность и поток этих путей.

**struct** elem {  
 **char** prev;  
 std::vector<std::pair<**char**, elemInfo>> ways;  
 **bool** isUsed;  
};

**struct** elemInfo{  
 **int** size;  
 **int** flow;  
};

Была определена функция итеративного поиска в глубину std::string find(std::map<**char**, elem> dict, **char** start, **char** end), возвращающая путь, если он был найден и строку, состоящую из символа start, если путь не был найден.

Функция возвращающая минимальную пропускную способность на пути: **int** minSize(std::string a, std::map<**char**, elem> dict)

Функция изменения потоков и модификации пропускных способностей:

**void** change(std::string a, std::map<**char**, elem>& dict)

**Выводы.**

В ходе выполнения данной работы была написана программа, которая находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

**Тестирование**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 8 1 4 1 2 7 1 3 6 2 4 8 2 5 1 3 5 2 3 6 4 6 5 7 5 4 6 | 13  1 2 7  1 3 6  2 4 7  2 5 0  3 5 2  3 6 4  5 4 6  6 5 4 |
| 5  a d  a b 20  b c 20  c d 20  a c 1  b d 1 | 21  a b 20  a c 1  b c 19  b d 1  c d 20 |
| 9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11 b e 8 a g 10 f d 8 | 18  a b 8  a g 10  b c 0  b e 8  c d 10  e f 8  f d 8  g h 10  h c 10 |

**Исходный код**

#include **<iostream>**#include **<map>**#include **<vector>**#include **<algorithm>**#include **<string>  
  
struct** elemInfo{  
 **int** size;  
 **int** flow;  
};  
  
**struct** elem {  
 **char** prev;  
 std::vector<std::pair<**char**, elemInfo>> ways;  
 **bool** isUsed;  
};  
  
**char** select(std::vector<std::pair<**char**, elemInfo>> **const**& arr, std::map<**char**, elem> & dict){  
 **char** needed = 0;  
 **for**(**auto** i:arr){  
 **if**(!dict[i.first].isUsed && i.second.size>0) {  
 needed = i.first;  
 }  
 }  
 **return** needed;  
}  
  
std::string find(std::map<**char**, elem> dict, **char** start, **char** end){  
 **char** curr = start;  
 std::string path = {start};  
  
 **while**(curr != end && !(curr == start && select(dict[curr].ways, dict) == 0)) {  
 dict[curr].isUsed = **true**;  
 **if**(select(dict[curr].ways, dict) != 0) {  
 **char** prev = curr;  
 curr = select(dict[curr].ways, dict);  
 dict[curr].prev = prev;  
 path += curr;  
 }  
 **else** {  
 path.erase(path.end()-1);  
 curr = dict[curr].prev;  
 }  
 }  
 **return** path;  
}  
  
**int** minSize(std::string a, std::map<**char**, elem> dict) {  
 **int** min = 2147483647;  
 **while**(a.size()>1){  
 **for**(**auto** i:dict[a[0]].ways)  
 **if**(i.first == a[1] && i.second.size < min) {  
 min = i.second.size;  
 }  
 a.erase(a.begin());  
 }  
 **return** min;  
}  
  
**void** change(std::string a, std::map<**char**, elem>& dict){  
 **int** min = minSize(a,dict);  
  
 **while**(a.size()>1){  
 **for**(**auto**& i:dict[a[0]].ways)  
 **if**(i.first == a[1]) {  
 i.second.flow += min;  
 i.second.size -= i.second.flow;  
  
 **for**(**auto**& j:dict[a[1]].ways) {  
 **if**(j.first == a[0]) {  
 j.second.flow -= min;  
 j.second.size -= j.second.flow;  
 }  
 }  
 }  
 a.erase(a.begin());  
 }  
  
}  
  
**bool** cmp(std::pair<**char**, elemInfo> **const**& a, std::pair<**char**, elemInfo> **const**& b) {  
 **return** a.first < b.first;  
}  
  
**int** main() {  
 **int** N;  
 **char** start, end;  
 std::cin >> N >> start >> end;  
 **char** a, b;  
 **int** c = 0;  
 std::map<**char**, elem> my\_map;  
 **for**(**int** i = 0; i < N; ++i) {  
 std::cin >> a >> b >> c;  
 my\_map[a].ways.push\_back({b,{c,0}});  
 }  
 std::string cur\_way = find(my\_map,start,end);  
 **while** (cur\_way != std::string(1,start)) {  
  
 change(cur\_way, my\_map);  
 cur\_way = find(my\_map,start,end);  
  
 }  
 **int** sum = 0;  
 **for**(**auto** i:my\_map[start].ways)  
 sum += i.second.flow;  
 std::cout << sum <<std::endl;  
 **for**(**auto** k:my\_map) {  
 std::sort(k.second.ways.begin(),k.second.ways.end(),cmp);  
 **for**(**auto** i:k.second.ways)  
 std::cout << k.first << **" "** << i.first << **" "** << std::max(0,i.second.flow) << std::endl;  
 }  
 **return** 0;  
}