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. 7382 _____ Гаврилов А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2019

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 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:

P_{max} - 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).

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

Вар. 4м. Матрица смежности. Поиск пути по правилу: каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе всего друг к другу.

Пояснение задания.

На вход программе подается граф, количество ребер в нем и имена вершин- стока и истока. Программа находит максимальный поток этого графа из истока в сток и поток проходящий через ребра графа.

Описание алгоритма.

Алгоритм находит путь из истока в сток в данном графе, выбирая вершины, имена которых в алфавите ближе всего друг к другу и разность между максимальной пропускной способностью и реальным потоком, который через ребро проходит, больше нуля, затем проходит по этому пути и ищет ребро с

минимальной пропускной способностью с вычетом величины уже протекающего через нее потока, поток этого пути будет равен этой минимальной величине. Добавляет ее к максимальному потоку графа. Алгоритм повторяется пока существуют пути по указанным критериям из истока в сток. Сложность алгоритма по памяти $O(V^2+E)$, по операциям $O(f*(ElogE+E))$, где f – максимальный поток в графе.

Описание структур

Класс Graph.

Используется матрица смежности, которая хранит пару – пропускную способность ребра и реальный протекающий через ребро поток. Матрица реализована с помощью стандартного контейнера std::vector<std::vector<std::pair<int,int>>>.

Описание функций

Код программы приведен в приложении А.

Написан класс Compare, который используется для сортировки очереди с приоритетом. Он имеет один оператор:

bool operator()(const std::pair<char, char> &a,const std::pair<char, char> &b) const

const std::pair<char, char> &a,const std::pair<char, char> &b – элементы очереди.

Тип std::priority_queue <std::pair<char, char>, std::vector < std::pair<char, char>>, Compare> - очередь с приоритетом, хранящая пары «откуда»-«куда», название этого типа заменено на queue.

Методы класса Graph:

int next_track(char from, char to)

char from, char to – названия истока и стока соответственно.

Метод возвращает минимальный поток найденного пути в графе.

void add(char from, char to, int flow)

char from, char to – названия вершин «откуда» и «куда».

int flow – пропускная способность ребра между этими вершинами.

Метод добавляет ребро в граф.

int get_index(char el)

char el – название вершины.

Метод возвращает индекс этой вершины в матрице смежности.

void print_matr()

Метод выводит матрицу смежности в консоль.

int get_resid_flow(char from, char to)

char from, char to – названия вершин «откуда» и «куда».

Метод возвращает величину оставшегося потока.

void find_next(queue &q,std::set<char> &checked, char el,char to)

queue &q – очередь с приоритетом.

std::set<char> &checked – контейнер, хранящий посещенные вершины.

char el-название вершины.

char to-название стока.

Метод опускает в очередь все инцидентные поданной вершине вершины, которые не были посещены и остаточная пропускная способность которых больше нуля.

void recount_flow(std::map<char,char> &track, int flow, char from, char to)

std::map<char,char> &track-контейнер, хранящий ребра, по которым осуществлялся переход: «куда»-«откуда».

int flow-реальный поток, на который необходимо увеличить поток в ребре

char from, char to – названия истока и стока соответственно.

Метод пересчитывает потоки в графе.

int find_min_flow(std::map<char, char> &track,char from, char to)

std::map<char,char> &track-контейнер, хранящий ребра, по которым осуществлялся переход: «куда»-«откуда».

char from, char to – названия истока и стока соответственно.

Метод ищет минимальный поток этого пути.

void find_flow(char from, char to)

char from, char to – названия истока и стока соответственно.

Метод находит и выводит потоки в всех ребрах графа и сам максимальный поток графа.

void print_flows()

Метод печатает потоки графа.

void print_queue(queue q)

queue q-очередь с приоритетом.

Метод выводит состояние очереди.

int main()

Главная функция программы. Считывает входные данные, запускает алгоритм поиска максимального потока в графе.

Тестирование.

Вводимые данные	Результат:
30	26
a	a b 6
n	a c 6
a b 6	a d 8
a c 6	a e 6
a d 8	b c 2
a e 9	b f 4
b c 3	c d 4
b f 4	c g 4
c d 4	d e 0
c g 4	d h 5

d e 3 d h 5 d i 10 e i 6 f c 9 f j 5 g d 10 g k 5 h g 8 h l 5 h m 12 i h 7 i m 7 j g 10 j n 6 k h 12 k j 8 k n 9 l k 8 l n 7 m l 7 m n 6	d i 7 e i 6 f c 0 f j 4 g d 0 g k 5 h g 1 h l 5 h m 5 i h 6 i m 7 j g 0 j n 4 k h 0 k j 0 k n 9 l k 4 l n 7 m l 6 m n 6
11 a g a b 3 a d 3 b c 4	5 a b 2 a d 3 b c 2 c a 0 c d 1

c a 3 c d 1 c e 2 d e 2 d f 6 e b 1 e g 1 f g 9	c e 1 d e 0 d f 4 e b 0 e g 1 f g 4
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

Вывод.

В процессе выполнения лабораторной работы был изучен и реализован алгоритм Форда-Фалкерсона с учетом индивидуального задания. Разработана программа, которая ищет максимальный поток в графе, используя данный алгоритм. Алгоритм исследован на сложность по памяти и по операциям.

Приложение А.

```
     #include <iostream>
     #include <vector>
     #include <map>
     #include <queue>
     #include <set>

     //#define TEST

     class Compare{
                                          //Класс-компоратор      для
сортировки пар в очереди с приоритетом
     public:
     bool   operator()(const   std::pair<char,   char>   &a,const
std::pair<char, char> &b) const
     {
          return abs(a.first-a.second)>abs(b.first-b.second);
     }
     };


     typedef std::priority_queue <std::pair<char, char>,
          std::vector < std::pair<char, char>>,
          Compare> queue;
     //очередь с приоритетом

     std::set<std::pair<char,char>> input;                        //
ребра, поданные на вход

     class Graph{
     std::vector<std::vector<std::pair<int,int>>> matr;   //матрица
смежности
     std::vector<char> v;
     int next_track(char from, char to){                          //
метод, который ищет путь и возвращает поток, проходящий через него
          queue q;
          std::set<char> checked;                                 //
список посещенных вершин
          std::map<char,char> track;                              //
текущий путь, хранящий пары: "куда", "откуда"
```

```
     char cur=from;
     checked.insert(cur);
     find_next(q,checked,from, to);
#ifdef TEST
     print_matr();
     print_queue(q);
#endif
     while(!q.empty()){
//пока существуют ребра, по которым можно пройти
          cur=q.top().second;
#ifdef TEST
          std::cout<<"Текущий переход: "<<q.top().first<<"->"
          <<q.top().second<<std::endl;
#endif
          checked.insert(cur);
          track[cur]=q.top().first;
          q.pop();
          if(cur==to){
#ifdef TEST
               std::cout<<"Найден конец!\n";
#endif
               int min=find_min_flow(track,from, to);
//находим поток
               recount_flow(track, min, from, to);
//делаем перерасчет проходящего потока
               return min;
          }
          find_next(q,checked,cur, to);
#ifdef TEST
          if(q.empty()){
               std::cout<<"Путей больше нет!\n";
               continue;
          }
          print_queue(q);
#endif
     }
     return 0;
}

public:
Graph(){}
```

```
     void add(char from, char to, int flow){
     //метод добавления ребра в граф
          int fr,t;
          fr=get_index(from);
          if(fr==-1)
               v.push_back(from);
          t=get_index(to);
          if(t==-1)
               v.push_back(to);
          if(matr.size()<v.size()){
               matr.resize(v.size());
               for(int i=0;i<matr.size();i++){
                    matr[i].resize(matr.size());
               }
          }
          matr[get_index(from)][get_index(to)]={flow,0};
     }
     int get_index(char el){
     //получение индекса вершины по ее букве
          for(int i=0;i<v.size();i++){
               if(v[i]==el)
                    return i;
          }
          return -1;
     }
     void print_matr(){
     //метод печати матрицы
          std::cout<<"   ";
          for(int i=0;i<v.size();i++)
               std::cout<<v[i]<<"   ";
          std::cout<<std::endl;
          for(int i=0;i<matr.size();i++){
               std::cout<<v[i]<<" ";
               for(int j=0;j<matr[i].size();j++)
                    std::cout<<matr[i][j].first<<"/"<<matr[i]
[j].second<<' ';
               std::cout<<std::endl;
          }
     }
     int get_resid_flow(char from, char to){
     //получение максимально возможного потока через данное ребро
на текущем этапе
```

```
          if(get_index(from)==-1 || get_index(to)==-1)
               return -1;
          return        matr[get_index(from)][get_index(to)].first-
matr[get_index(from)][get_index(to)].second;
     }

     void find_next(queue &q,std::set<char> &checked, char el,char
to){      //метод ищет инцидентные вершины, которые не были
посещены и опускает их в очередь
          if(get_resid_flow(el,to)>0){
               while(!q.empty()){
                    q.pop();
               }
               q.push({el,to});
               return;
          }
          for(int k=0;k<v.size();k++){

               if(get_resid_flow(el,v[k])<=0 || checked.find(v[k])!
=checked.end())
                    continue;
               q.push({el,v[k]});
               checked.insert(v[k]);
          }
     }
     void recount_flow(std::map<char,char> &track, int flow, char
from, char to){                    //метод проходит по пути и
пересчитывает проходящий через ребра поток
          char cur=to;
     #ifdef TEST
          std::cout<< "Пересчет потока, проходящего через ребра
найденного пути.\n";
     #endif
          while(cur!=from){
               matr[get_index(track[cur])]
[get_index(cur)].second+=flow;
               matr[get_index(cur)][get_index(track[cur])].second-
=flow;
               cur=track[cur];
          }
     }
```

```
     int find_min_flow(std::map<char, char> &track,char from, char
to){                          //метод поиска минимального потока в
пути
          int min=get_resid_flow(track[to],to);
          char cur=track[to];
     #ifdef TEST
          std::string str;
          str.push_back(cur);
          str.push_back(to);
     #endif
          while(cur!=from){
               if(get_resid_flow(track[cur],cur)<min)
                    min=get_resid_flow(track[cur],cur);
               cur=track[cur];
     #ifdef TEST
               str.insert(str.begin(),cur);
     #endif
          }
     #ifdef TEST
               std::cout<<"Путь: "<<str<<std::endl
               <<"Поток пути: "<<min<<std::endl;
     #endif
          return min;
     }

     void find_flow(char from, char to){
          //метод поиска максимального потока в графе
     #ifdef TEST
          int i=2;
          std::cout<<"Начинаем алгоритм!\n";
     #endif
          int max_gr_flow=0,min_track_flow=0;
     #ifdef TEST
          std::cout<<"Путь #1\n";
     #endif
          min_track_flow=next_track(from, to);
          while(min_track_flow>0){
     //пока существует путь
     #ifdef TEST
          std::cout<<"Путь #"<<i<<std::endl;
          i++;
     #endif
```

```
                max_gr_flow+=min_track_flow;                        //
увеличиваем максимальный поток на величину потока пути
                min_track_flow=next_track(from, to);
           }
           std::cout<<max_gr_flow<<std::endl;
           print_flows();
     }
     void print_flows(){
     //метод печати реальных потоков в изначальных ребрах
           for(auto k:input){
                if(matr[get_index(k.first)]
[get_index(k.second)].second>=0)
                     std::cout<<k.first<<'             '<<k.second<<'
'<<matr[get_index(k.first)][get_index(k.second)].second<<std::endl;
                else
                     std::cout<<k.first<<'             '<<k.second<<'
'<<0<<std::endl;
           }
     }
     void print_queue(queue q){
     //метод печати очереди с приоритетом
           std::cout<<"Ребра в очереди:\n";
           while(!q.empty()){
                std::cout<<q.top().first<<"-
>"<<q.top().second<<std::endl;
                q.pop();
           }
           std::cout<<std::endl;
     }
     };

     int main(){
     Graph a;
     int n,flow;
     char from,to,first,second;
     std::cin>>n>>from>>to;
     for(int i=0;i<n;i++){
           std::cin>>first>>second>>flow;
           input.insert(std::pair<char,char>(first,second));
           a.add(first,second,flow);
     }
     a.find_flow(from,to);
```

```
    return 0;
}
```