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

ОТЧЕТ

по практической работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потоки в сети

Студент гр. 7383 _____ Зуев Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Цель работы.

Цель работы: ознакомиться с алгоритмом Форда-Фалкерсона на примере построения алгоритма для выполнения задачи.

Формулировка задачи: Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона. Вариант 3с: представить граф в виде списка смежности и производить поиск пути по правилу: каждый раз выполняется переход по ребру, ведущему в вершину, имя которой в алфавите ближайшее к началу алфавита.

Реализация задачи.

Для хранения графа в виде списка смежности была создана структура `Vertex`.

`char name` — название вершины.

`bool is_visited` — флаг, по которому функция поиска пути понимает, что вершина уже включена в этот путь.

`map<char, Vertex*> descedants` — карта указателей на смежные вершины.

`map<char, int> lengthes` — начальные длины дуг до смежных вершин.

`map<char, int> mod_lengthes` — длины дуг до смежных вершин, увеличивающиеся алгоритмом.

`map<char, bool> is_entered` — флаг, по которому функция вывода ребер понимает, что ребро введено пользователем.

`map<char, bool> is_back` — флаг, в котором хранится информация о том, обратное ли ребро или прямое.

Для реализации задачи был создан класс `Graph`.

Ниже представлены поля класса:

`char begin` — исток.

char end — сток.

int min_b — минимальное значение длины обратного ребра.

map<char, Vertex*> vertexes — карта вершин графа, с ключами в виде названий вершин.

string current_path — путь, веса вершин которого будут увеличиваться.

Далее представлены методы класса:

void readCons() — считывает количество путей, начальную и конечную вершины и дуги графа с консоли.

void path_saturation(Vertex* current) — рекурсивная функция, перебирающая все пути графа и, если находит сток, вызывающая функцию FFA переходя сначала по прямым дугам, потом по обратным.

void FFA() — ищет минимальный вес дуги в пути, максимальное значение, которое можно добавить к длине пути, чтобы длина пути не стала больше введенной пользователем, и увеличивает длины дуг.

void start() — инициализирует начало алгоритма.

void print() — выводит в консоль максимальный поток, и ребра графа с фактической величиной протекающего потока.

В главной функции main создается класс для графа и последовательно вызываются методы считывания графа с консоли и инициализации алгоритма.

Код программы представлен в приложении Б.

Исследование сложности алгоритма.

Функция path_saturation выполняет построение пути за время O(|E|), так как в худшем случае при построении пройдет по всем ребрам.

За одно выполнение функции FFA поток в сети может увеличиться в худшем случае на 1, поэтому в худшем случае вызовов функции FFA функцией path_saturation будет равно величине максимального потока f в сети.

Функция FFA увеличивает веса дуг в пути за время O(|E|), так как путь может состоять из всех ребер графа.

Из вышеперечисленного следует, что алгоритм будет работать за время O(|E|f).

Тестирование.

Программа была собрана в компиляторе G++ в среде разработки Qt в операционной системе Linux Ubuntu 17.10.

В ходе тестирования ошибок выявлено не было.

Корректные тестовые случаи представлены в приложении А.

Выводы.

В ходе выполнения данной работы был изучен и реализован алгоритм Форда-Фалкерсона поиска максимального потока в сети. Оценена сложность реализованного алгоритма, она составляет O(|E|f).

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Выходные данные
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
4 a d a b 1 c b 1 a c 2 c d 3	2 a b 0 a c 2 c b 0 c d 2
5 a e a b 6 b c 4 c e 4 b d 3 d e 3	6 a b 6 b c 4 b d 2 c e 4 d e 2

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <vector>
#include <map>
#include <string>

using namespace std;

struct Vertex
{
   char name;
   bool is_visited;
   map<char, Vertex*> descedants;
   map<char, int> lengthes;
   map<char, int> mod_lengthes;
   map<char, bool> is_entered;
   map<char, bool> is_back;
   Vertex(char c):is_visited(0),name(c){}
};

class Graph
{
private:
   char begin;
   char end;
   int min_b;
   map<char, Vertex*> vertexes;
   string current_path;
public:
   Graph():min_b(10000){}
   void readCons()
   {
      char f, s;
      int n, length;
      cin >> n >> begin >> end;
      for(int i = 0; i<n; i++)
      {
         cin >> f;
         cin >> s;
         cin >> length;
         if(!vertexes[f])
            vertexes[f] = new Vertex(f);
         if(!vertexes[s])
            vertexes[s] = new Vertex(s);
         vertexes[f]->descedants[s] = vertexes[s];
         vertexes[f]->lengthes[s] = length;
         vertexes[f]->mod_lengthes[s] = 0;
         vertexes[f]->is_entered[s] = 1;
         vertexes[f]->is_back[s] = 0;
```

```
            if(vertexes[s]->descedants[f] == nullptr)
            {
               vertexes[s]->descedants[f] = vertexes[f];
               vertexes[s]->lengthes[f] = length;
               vertexes[s]->mod_lengthes[f] = 0;
               vertexes[s]->is_entered[f] = 0;
               vertexes[s]->is_back[f] = 1;
            }
         }
      }
      void path_saturation(Vertex* current)
      {
         for(map<char, Vertex*>::iterator it = current->descedants.begin(); it!=
current->descedants.end(); ++it)
         {
            if(it->second->is_visited)
               continue;
            current_path+=it->first;
            if(it->first == end)
               FFA();
            else
               if(!(it->second->is_visited) && !(current->is_back[it->first]))
               {
                  it->second->is_visited = 1;
                  path_saturation(it->second);
               }
            it->second->is_visited = 0;
            current_path.pop_back();
         }
         int min_b_tmp = min_b;
         for(map<char, Vertex*>::iterator it = current->descedants.begin(); it!=
current->descedants.end(); ++it)
         {
            if(it->second->is_visited)
               continue;
            current_path+=it->first;
            if(it->first == end)
               FFA();
            else
               if(!(it->second->is_visited))
               {
                  if(current->is_back[it->first] && vertexes[it->first]-
>mod_lengthes[current->name] < min_b)
                     min_b = vertexes[it->first]->mod_lengthes[current->name];
                  it->second->is_visited = 1;
                  path_saturation(it->second);
               }
            min_b = min_b_tmp;
            it->second->is_visited = 0;
            current_path.pop_back();
         }
```

```
    }

    void FFA()
    {
        int length;
        int min_length = 10000;
        int max_length = 10000;
        for(int i = 0; i < current_path.length() -1; i++)
        {
            length = vertexes[current_path[i]]->lengthes[current_path[i+1]];
            if(length < min_length)
                min_length = length;
            if(length - vertexes[current_path[i]]->mod_lengthes[current_path[i+1]]
< max_length)
                max_length = length - vertexes[current_path[i]]-
>mod_lengthes[current_path[i+1]];
            if(max_length == 0)
                return;
        }
        if(min_length > max_length)
            min_length = max_length;
        if(min_length > min_b)
            min_length = min_b;
        for(int i = 0; i < current_path.length() -1; i++)
            vertexes[current_path[i]]->mod_lengthes[current_path[i+1]] +=
min_length;
    }
    void print()
    {
        int sum = 0;
        for(map<char, int>::iterator it = vertexes[begin]->mod_lengthes.begin();
it!=vertexes[begin]->mod_lengthes.end(); ++it)
            sum+=it->second;
        cout<<sum<<endl;
        for(map<char, Vertex*>::iterator it = vertexes.begin(); it!
=vertexes.end(); ++it)
        {
            for(map<char, Vertex*>::iterator it2 = it->second-
>descedants.begin();it2 != it->second->descedants.end(); ++it2)
                if(vertexes[it->first]->is_entered[it2->first])
                {
                    cout<<it->first<<' '<<it2->first<<' '<<((it->second-
>mod_lengthes[it2->first] - it2->second->mod_lengthes[it->first])<0 ? 0 : (it-
>second->mod_lengthes[it2->first] - it2->second->mod_lengthes[it-
>first]))<<endl;
                }
        }
    }
    void start()
    {
        current_path+=begin;
```

```
        vertexes[begin]->is_visited = 1;
        path_saturation(vertexes[begin]);
        print();
    }
};

int main()
{
    Graph p;
    p.readCons();
    p.start();
}
```