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. 7383 _____ Тян Е.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

СОДЕРЖАНИЕ

1. ЦЕЛЬ РАБОТЫ

Цель работы: исследовать и реализовать задачу нахождения максимального потока в сети, используя алгоритм Форда-Фалкерсона.

Формулировка задачи: необходимо разработать программу, которая решает задачу нахождения максимального потока в сети, а также фактическую величину потока протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имен вершин и целого неотрицательного числа – пропускной способности (веса).

В ответе выходные ребра должны быть отсортированы в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные ребра, даже если поток в них равен 0).

Входные данные:

N – количество ориентированных ребер графа

v_0 – исток

v_n – сток

$v_i \;\; v_j \;\; \omega_{ij}$ – ребро графа

$v_i \;\; v_j \;\; \omega_{ij}$ – ребро графа

…

Выходные данные:

$P_{\max}$ – величина максимального потока

$v_i \;\; v_j \;\; \omega_{ij}$ – ребро графа с фактической величиной протекающего потока

$v_i \;\; v_j \;\; \omega_{ij}$ – ребро графа с фактической величиной протекающего потока

…

2. РЕАЛИЗАЦИЯ ЗАДАЧИ

В данной работе используются главная функция (int main()) и класс (flow), содержащий конструктор (flow(int n = 0)) и методы (void path(vector<int> &p), int ford_fulk(), void print(int n)), деструктор(~flow()).

Свойства класса flow:

• edge – матрица смежности, хранящая веса ребер графа;

• up_down_edge – матрица смежности, хранящая ребра обратные к исходным;

• vertex – массив вершин графа;

• edge_out – матрица смежности, хранящая фактические величины протекающего потока ребер;

• out_in – отсортированный вектор кортежей, хранящий выходные данные.

Параметры, передаваемые в конструктор flow(int n = 0):

• n – количество ориентированный ребер графа.

Параметры, передаваемые в метод void path(vector<int> &p):

• p – вектор, хранящий путь.

Параметры передаваемые в void print(int n):

• n – количество ориентированный ребер графа.

В функции main() считываются количество ориентированный ребер графа, создается объект класса flow. При вызове конструктора по умолчанию считываются исток и сток, инициализируется граф в виде матрицы смежности в лице массива edge. Далее в функции main() вызывается метод int ford_fulk(). Пока существует путь из истока в сток вызывается метод void path(vector<int> &p), который находит путь по правилу, каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе друг к другу. Когда путь найден, начинается обход данного пути, и создаются обратные ребра для каждого ребра в данном графе, которые хранятся в массиве up_down_edge. Величина максимального потока увенчивается на минимальную пропускную способность ребра. являющегося

частью пути.

Когда все существующие пути пройдены, в функции main() вызывается метод void print(int n), который выводит результат на экран в необходимом виде: ребра отсортированы в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные ребра, даже если поток в них равен 0).

3. ТЕСТИРОВАНИЕ

Программа была собрана в компиляторе G++ в OS Linux Ubuntu 12.04. Программа может быть скомпилирована с помощью команды:

g++ <имя файла>.cpp -std=c++11

Тестовые случаи представлены в Приложении А.

Исходя из тестовых случаев можно увидеть, что тестовые случаи не выявили некорректной работы программы, что говорит о том, что по результатам тестирования было показано, что поставленная задача была выполнена.

4. ИССЛЕДОВАНИЕ

Компиляция была произведена в компиляторе в g++ Version 4.2.1. В других ОС и компиляторах тестирование не проводилось.

На каждом шаге работы алгоритма поток может увеличиться по крайней мере на единицу, тогда он сойдется не более чем за O(f) шагов, где f – максимальный поток в графе. Поиск пути в графе осуществляется грубо говоря поиском в глубину. Сложность метода поиска в глубину – O($|E| + |V|$), где E – количество ребер графа, а V – количество вершин графа. Тогда общая сложность алгоритма данной программы – $O(f \cdot |E| + f \cdot |V|)$.

Ранее была приведена сложность алгоритма по времени. Сложность по памяти же будет O($E + f \cdot V$). Так как необходима память для хранения обратных ребер O(E), которые строятся по мере обхода графа, также каждый раз выделяется память для стека, так как осуществляете обход в глубину O($f \cdot V$).

5. ВЫВОД

В ходе выполнения лабораторной работы была решена задача нахождения максимального потока в сети, используя алгоритм Форда-Фалкерсона. Полученный алгоритм имеет сложность линейную как по времени, так и по памяти.

Была написана программа строящая граф в виде матрицы смежности, вычисляющая кратчайший путь от заданной вершины до конечной, по правилу, каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе друг к другу, если такой существует.

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

№	Ввод	Вывод
1	10 a e a b 16 b d 12 d e 20 c b 10 b c 4 d c 9 c f 14 d f 7 f e 4 a c 13	16 a b 16 a c 0 b c 4 b d 12 c b 0 c f 4 d c 0 d e 12 d f 0 f e 4
2	16 w z w x 2 x w 2 w v 1 v w 1 w y 3 y w 3 x y 4 y x 4 y v 1 v y 1 y z 2 z y 2 x z 3 z x 3 v z 1 z v 1	6 v w 0 v y 0 v z 1 w v 1 w x 2 w y 3 x w 0 x y 0 x z 3 y v 0 y w 0 y x 1 y z 2 z v 0 z x 0 z y 0

3	9 k t k m 5 m o 5 o s 5 s t 5 m n 7 n t 7 k l 7 l p 8 p n 7	12 k l 7 k m 5 l p 7 m n 0 m o 5 n t 7 o s 5 p n 7 s t 5

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <stack>

using namespace std;

class flow
{
public:
   flow(int n = 0)
   {
      edge = new int*[n];
      for(int i = 0; i < n; i++)
      {
         edge[i] = new int[n];
         for(int j = 0; j < n; j++)
         {
            edge[i][j] = 0;
         }
      }
      up_down_edge = new int*[n];
      for(int i = 0; i < n; i++)
      {
         up_down_edge[i] = new int[n];
         for(int j = 0; j < n; j++)
         {
            up_down_edge[i][j] = 0;
         }
      }
      char s, t;
      cin >> s;
      cin >> t;
      if(s == t)
      {
         cout << "Wrong input!" << endl;
         return;
      }
      vertex.push_back(s);
      vertex.push_back(t);
      for(int i = 0; i < n; i++)
      {
         char b, e;
         int w, j, k;
         cin >> b >> e >> w;
         if(b != e)
         {
            j = find(vertex.begin(), vertex.end(), b) - vertex.begin();
            if(j == vertex.size())
            {
```

```
            vertex.push_back(b);
            j = vertex.size() - 1;
        }
        k = find(vertex.begin(), vertex.end(), e) - vertex.begin();
        if(k == vertex.size())
        {
            vertex.push_back(e);
            k = vertex.size() - 1;
        }
        edge[j][k] = w;

    }else{
        cout << "Wrong input!" << endl;
        return;
    }
  }
  edge_out = new int*[n];
  for(int i = 0; i < n; i++)
  {
    edge_out[i] = new int[n];
    for(int j = 0; j < n; j++){
        edge_out[i][j] = -1;
        if(edge[i][j] > 0)
        {
            edge_out[i][j] = 0;
        }
    }
  }
}

void path(vector<int> &p)
{
  bool vis[vertex.size()];
  for(int i = 0; i < vertex.size(); i++)
    vis[i] = false;
  stack<int> indiv;
  indiv.push(0);
  do
  {
    vector<char> arr;
    int top_vertex = indiv.top();
    p.push_back(top_vertex);
    if(top_vertex == 1)
        return;
    vis[top_vertex] = true;
    int k = arr.size();
    for(int i = 1; i < vertex.size(); i++)
    {
        if(vis[i] == false && (edge[top_vertex][i] > 0 || up_down_edge[top_vertex][i] > 0))
        {
            arr.push_back(vertex[i]);
        }
    }
    if(arr.size() == k)
    {
```

```
            int i = indiv.top();
            indiv.pop();
            p.pop_back();
            while(!indiv.empty() && !p.empty() && (edge[indiv.top()][i] > 0 ll
up_down_edge[indiv.top()][i] > 0))
            {
               if(indiv.top() == 1)
               {
                  if(p.back() != 1)
                     p.push_back(1);
                  return;
               }
               p.pop_back();
               i = indiv.top();
               indiv.pop();
            }
         }else{
            sort(arr.begin(), arr.end());
            for(int i = arr.size() - 1; i >= 0; i--)
            {
               indiv.push(find(vertex.begin(), vertex.end(), arr[i]) - vertex.begin());
            }
         }
      }
      while(!indiv.empty());
      return;
   }

   int ford_fulk()
   {
      vector<int> p;
      int res_flow = 0;
      do
      {
         vector<int> edge_flow;
         p.clear();
         path(p);
         if(p.empty())
            return res_flow;
         for(int i = 0; i < p.size() - 1; i++)
         {
            if(edge[p[i]][p[i + 1]] > 0)
               edge_flow.push_back(edge[p[i]][p[i + 1]]);
            else
               edge_flow.push_back(up_down_edge[p[i]][p[i + 1]]);
         }
         int minim = (*min_element(edge_flow.begin(), edge_flow.end()));
         res_flow += minim;
         for(int i = 0; i < p.size() - 1; i++)
         {
            up_down_edge[p[i + 1]][p[i]] += minim;
            edge[p[i]][p[i + 1]] -= minim;
            if(edge[p[i]][p[i + 1]] < 0)
               edge[p[i]][p[i + 1]] += minim;
            if(edge_out[p[i]][p[i + 1]] != -1)
```

```
                edge_out[p[i]][p[i + 1]] +=minim;
            if(edge[p[i]][p[i + 1]] == 0){
               edge[p[i]][p[i + 1]] = -1;
               up_down_edge[p[i + 1]][p[i]] = -1;
            }
         }
      }
      while(!p.empty());
      return res_flow;
   }

   void print2(int n)
   {
      for(int i = 0; i< vertex.size(); i++)
      {
         for(int j =0 ; j < vertex.size(); j++)
         {
            if(edge_out[i][j] > -1)
            {
               if(edge_out[j][i] > 0)
               {
                  edge_out[j][i] = abs(edge_out[j][i] - edge_out[i][j]);
                  edge_out[i][j] = 0;
               }
               if(up_down_edge[j][i] > 0 && edge[j][i] == -1)
                  edge_out[i][j] = 0;
               auto tmp = make_tuple(vertex[i], vertex[j], edge_out[i][j]);
               out_in.push_back(tmp);
            }
         }
      }
      sort(out_in.begin(), out_in.end());
      for(int i = 0; i < n; i++)
      {
         cout << get<0>(out_in[i]) << ' ' << get<1>(out_in[i]) << ' ' << get<2>(out_in[i]) << endl;
      }
   }

   ~flow()
   {
      delete edge;
      delete up_down_edge;
      delete edge_out;
   }

private:
   int** edge;
   int** up_down_edge;
   vector< char > vertex;
   int** edge_out;
   vector<tuple<char,char,int>> out_in;
};

int main()
{
```

```
    int n;
    cin >> n;
    flow F(n);
    int max_flow = F.ford_fulk();
    if(max_flow == 0)
        return 0;
    cout << max_flow << endl;
    F.print2(n);
    return 0;
}
```