**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»**

Студентка гр. 7382 Лящевская А. П.

Преподаватель Фирсов М. А.

Санкт-Петербург

2019

**Цель работы.**

Изучить алгоритмы поиска максимального потока в сети. Изучить

алгоритмы нахождения пути в графе. Написать программу для нахождения

потока в сети используя указанный алгоритм. Показать асимптотику

алгоритма по времени и по памяти. Провести тестирование, предоставить

работающую программу и отчёт.

**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

v0 - исток

vn - сток

vivjωij - ребро графа

vivjωij - ребро графа

...

Выходные данные:

Pmax - величина максимального потока

vivjωij - ребро графа с фактической величиной протекающего потока

vivjωij - 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).

**Пример входных данных.**

*7*

*a*

*f*

*a b 7*

*a c 6*

*b d 6*

*c f 9*

*d e 3*

*d f 4*

*e c 2*

**Выходные данные.**

В первой строке выходных данных вывести величину максимального потока 𝑃𝑚𝑎𝑥.

В каждой последующей строке вывести триплеты рёбер: 𝑣𝑖 – откуда, 𝑣𝑗 – куда, 𝑓𝑖𝑗 – фактическая величина потока через ребро.

***Пример выходных данных.***

*12*

*a b 6*

*a c 6*

*b d 6*

*c f 8*

*d e 2*

*d f 4*

*e c 2*

**Вариант 2c.**

Списки смежности. Поиск пути через поиск в глубину.

**Описание алгоритма.**

Подающиеся на вход программе ребра программа обрабатывает так, что помимо реального ребра, она создает еще и мнимые(обратные).

Затем она сразу приступает к поиску потока. Функция поиска

*int find\_flow(size\_t start, size\_t finish, int flow, int count\_rec)*

принимает на вход следующие элементы: *size\_t start* – индекс стартовой вершины*, size\_t finish* – индекс вершины финиша*, int flow* – поток*, int count\_rec* – счетчик рекурсии (нужен для корректного вывода действий функции). Действия функции:

1. Проверка на достижение вершины финиша. Если он достигнут, то возвращается поток *flow.*
2. Текущая вершина помечается посещенной.
3. Далее происходит перебор всех вершин инцидентных с текущей.
4. И если инцидентная вершина не была посещена. И по ней можно пустить ненулевой поток, то: 5-7
5. В *min\_result* записываем значение функции *find\_flow*, но уже со следующими параметрами: в качестве стартовой вершины передается инцидентная, в качестве финишной – финишная, а в качестве потока минимум из потока переданного текущей функции и потока, который можно пусть по выбранному ребру, и счетчик рекурсии + 1.
6. Если *min\_result*> 0 (можно пустить ненулевой поток по выбранному ребру), то вычитаем этот поток(*min\_result*) из остаточной пропускной способности выбранного ребра. А к обратному ребру прибавляем *mit\_result* к остаточной пропускной способности.
7. Возвращаем *min\_result*.
8. В общем цикле поиска всех потоков происходит суммирование найденных потоков и снятие меток посещенности для всех вершин пока находится ненулевые потоки.

**Асимптотика алгоритма.**

Добавляя поток увеличивающего пути к уже имеющемуся потоку, максимальный поток будет получен, когда нельзя будет найти увеличивающий путь. Тем не менее, если величина пропускной способности — иррациональное число, то алгоритм может работать бесконечно. В целых числах таких проблем не возникает и время работы ограничено O(|*E*|*f*), где E — число рёбер в графе, *f* — максимальный поток в графе, так как каждый увеличивающий путь может быть найден за O(E) и увеличивает поток как минимум на 1.

Сложность по памяти составляет O(|V|+|E|).

**Описание структур**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название структуры** | **Тип** | **Объект** | **Параметры** | **Описание** | **Возвращаемое значение** |
| struct Connect | - | type value; | - | Именование вершины | - |
| - | double weight; | - | Вес пути до этой вершины | - |
| - | bool real; | - | Реальное ребро при real = 1, при real = 0 – ребро мнимое. | - |
| struct Edge | - | char from; | - | Вершина начала ребра. | - |
| - | char to; | - | Вершина конца ребра. | - |
| - | int weight; | - | Вес ребра. | - |
| class Vertex | public | char key; | - | Наименование вершины. | - |
| public | bool visited = false; | - | Метка посещенности, при false – не посещена. | - |
| public | Connect \*connects; | - | Массив связей с вершиной key. | - |
| public | size\_t size\_v; | - | Хранит размер массива connects. | - |
| public | void resize\_con() | - | Увеличивает размер connects по надобности. | - |
| public | size\_t find\_con(const char litter) | const char litter –  символ для поиска. | Производит поиск среди элементов массива connects | Возвращает индекс элемента в connects. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| сlass Directed\_Graph | private | Vertex \*list | - | Хранит список вершин. | - |
| private | size\_t size\_l = 0 | - | Хранит размер list | - |
| private | void print() | - | Распечатывает граф | - |
| private | void push(const type key, const type value, const float weight) | const type key – начало ребра; const type value – конец ребра; const float weight – вес ребра; | Добавляет ребро из key в value весом weight. | - |
| private | size\_t find\_or\_add(const type litter) | const char litter –  символ для поиска. | Находит элемент в противном случае добавляет его | Возвращает индекс элемента в list. |
| private | void resize\_list() | - | Изменяет размер list | - |
| private | void unvisit() | - | Снимает метки посещенности вершин. | - |
| private | void print\_flow() | - | Распечатывает поток со всеми ребрами. | - |
| private | Int find\_flow(size\_t start, size\_t finish, int flow, count\_rec) | size\_t start – индекс начальной вершины;  size\_t finish – индекс конечной вершины;  int flow – текущий поток;  int count\_rec – счетчик рекурсии; | Находит поток в графе. | Возвращает полученный поток. |

**Тестирование.**

|  |  |  |
| --- | --- | --- |
| Номер  теста | Тест: | Результат: |
| 1 | 19  a  i  a b 2  a c 3  a d 5  b c 4  b d 1  c j 5  c e 4  c i 2  c f 3  d e 12  d g 9  d f 4  e i 34  f h 6  f i 5  g f 12  g h 8  h i 7  j e 6 | 10  a b 2  a c 3  a d 5  b c 2  b d 0  c e 0  c f 0  c i 0  c j 5  d e 4  d f 0  d g 1  e i 9  f h 1  f i 0  g f 1  g h 0  h i 1  j e 5 |
| 2 | 11  a  g  a b 3  a d 3  b c 4  c a 3  c d 1  c e 2  d e 2  d f 6  e b 1  e g 1  f g 9 | 5  a b 2  a d 3  b c 2  c a 0  c d 1  c e 1  d e 0  d f 4  e b 0  e g 1  f g 4 |

|  |  |  |
| --- | --- | --- |
| 3 | 30  a  n  a b 6  a c 6  a d 8  a e 9  b c 3  b f 4  c d 4  c g 4  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 | 26  a b 6  a c 6  a d 8  a e 6  b c 2  b f 4  c d 4  c g 4  d e 0  d h 5  d i 7  e i 6  f c 0  f j 4  g d 0  g k 5  h g 0  h l 5  h m 7  i h 7  i m 6  j g 1  j n 6  k h 0  k j 3  k n 9  l k 7  l n 5  m l 7  m n 6 |
| 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 ходе выполнения лабораторной работы был изучен алгоритм нахождения максимального потока в графе Форда-Фалкерсона.

Для хранения графа использовались списки смежности.

**Приложение 1. Код программы.**

* **3**[**.c**](https://github.com/makometr/AiSD/pull/37/files#diff-240c2792fd92b595432e18629f6e16b6)**pp**

#include <iostream>

#include <algorithm>

#include <vector>

#include <math.h>

int INF = (int) 1e9;

using namespace std;

struct Connect //Список связей

{

char value;

int weight;

bool real;

};

struct Edge //Ребро

{

char from;

char to;

int weight;

};

class Vertex //Вершина

{

public:

char key;

bool visited = false;

Connect \*connects;

size\_t size\_v = 0;

void resize\_con() //Функция изменения размера списка инцидентных ребер

{

if(size\_v%5 != 0 && size\_v) return;

Connect \*tmp = new Connect[size\_v + 5];

for(int i = 0; i < size\_v; i++)

tmp[i] = connects[i];

if(size\_v) delete [] connects;

connects = tmp;

};

size\_t find\_con(const char litter) //Функция нахождения вершины

{

for(int i = 0; i < size\_v; i++)

if(connects[i].value == litter)

return i;

return 0;

};

};

int comp(const Edge &val1, const Edge &val2) //Функция сравнения ребер

{

if(val1.from == val2.from)

return val1.to < val2.to;

return val1.from < val2.from;

};

class Directed\_Graph{

private:

Vertex \*list;

size\_t size\_l = 0;

public:

~Directed\_Graph()

{

for(size\_t i = 0; i < size\_l; i++)

delete [] list[i].connects;

delete [] list;

}

Directed\_Graph()

: list(new Vertex[3])

{};

void print() //Функция вывода графа

{

for(int i = 0; i < size\_l; i++)

{

cout << list[i].key << " : ";

for(int j = 0; j < list[i].size\_v; j++)

if(list[i].connects[j].real)

cout << list[i].connects[j].value << "(" << list[i].connects[j].weight << ") ";

cout << endl;

}

};

void push(const char key, const char value, const int weight, bool real)//Функция добавляет новое ребро

{

find\_or\_add(value);

Vertex &v = list[find\_or\_add(key)];

v.resize\_con();

v.connects[v.size\_v].value = value;

v.connects[v.size\_v].weight = weight;

v.connects[v.size\_v].real = real;

v.size\_v++;

};

size\_t find\_or\_add(const char litter) //Функция либо находит, либо добавляет новую вершину

{

for(int i = 0; i < size\_l; i++)

if(list[i].key == litter)

return i;

resize\_list();

list[size\_l].key = litter;

return size\_l++;

};

void resize\_list() //Функция изменения размера списка вершин с их ребрами

{

if(size\_l%3 != 0 || !size\_l) return;

Vertex \*tmp = new Vertex[size\_l + 3];

for(int i = 0; i < size\_l; i++)

{

tmp[i] = list[i];

tmp[i].connects = new Connect[list[i].size\_v];

tmp[i].connects = list[i].connects;

}

list = tmp;

for(int i = 0; i < size\_l; i++)

list[i].connects = tmp[i].connects;

};

void unvisit() //Функция обращает все ребра в непройденные

{

for(int i = 0; i < size\_l; i++)

list[i].visited = false;

};

int find\_flow(size\_t start, size\_t finish, int flow, int count\_rec) //Функция нахождения потока

{

cout << "Проходим в вершину (" << list[start].key << ")." << endl;

if(start == finish)

{

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << "Дошли до конца." << endl;

return flow; //возвращяем полученный минимум на пути

}

list[start].visited = true; //помечаем ребро посещенным

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << "Перебираем все инцидентные вершины для (" << list[start].key <<"):" << endl;

for(size\_t edge = 0; edge < list[start].size\_v; edge++) //Перебираем все инцидентные ребра вершине start

{

size\_t to = find\_or\_add(list[start].connects[edge].value); //Находим индекс инцидентной вершины в списке

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << edge + 1 << ") вершина (" << list[to].key << ") с остаточной пропускной способностью - " << list[start].connects[edge].weight << endl;

if(!list[to].visited && list[start].connects[edge].weight > 0)//Если вершина не посещена и остаточный вес ребра не нулевой, проходим

{

int min\_result = find\_flow(to, finish, min(flow, list[start].connects[edge].weight), count\_rec + 1);//Полученный максимальный поток через минимальное ребро

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << "Полученный поток - " << min\_result << endl;

if(min\_result > 0) //И если поток не нулевой, проходим

{

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << "Вычитаем из текущей пропускной способности реального ребра поток: " << list[start].connects[edge].weight

<< " - " << min\_result << " = " << list[start].connects[edge].weight - min\_result << endl;

list[start].connects[edge].weight -= min\_result; //Вычитаем поток из пропускной способности для реального ребра

//Обратное

size\_t con = list[to].find\_con(list[start].key); //Находим индекс обратного ребра

for(int k = 0; k < count\_rec; k++)

cout << "\t";

cout << "Добавляем к мощности мнимого ребра поток: " << list[to].connects[con].weight

<< " + " << min\_result << " = " << list[to].connects[con].weight + min\_result << endl;

list[to].connects[con].weight += min\_result; //И к обратному прибавляем поток

return min\_result; //Возвращаем результат

}

}

else

{

for(int k = 0; k < count\_rec; k++)

cout << "\t";

if(list[to].visited)

cout << "Вершина (" << list[to].key <<") уже посещена." << endl;

else

cout << "Недостаточная пропускная способность." << endl;

}

}

return 0; //если не нашли поток из этой вершины вернем 0

}

void print\_flow() //Функция печати ребер с их потоком

{

vector <Edge> vec; //Контейнер для ребер с потоком

for(int i = 0; i < size\_l; i++)

for(int j = 0; j < list[i].size\_v; j++)

if(!list[i].connects[j].real) //Добавляем, если ребро мниное

vec.push\_back({list[i].connects[j].value, list[i].key, abs(list[i].connects[j].weight < 0 ? 0 : list[i].connects[j].weight) });

sort(vec.begin(), vec.end(), comp); //Сортируем содержимое контейнера с помощью comp

for\_each(vec.begin(), vec.end(), [](const Edge& obj){cout << obj.from << " " << obj.to << " " << obj.weight << endl;});//Выводим

if(!vec.empty()) vec.clear(); //Очистка контейнера

};

};

int main(){

setlocale(LC\_ALL, "rus");

int N;

char start, finish;

cin >> N >> start >> finish; // считываем количество ребер, начальную и конечную вершины

Directed\_Graph graf;

char from, to;

int weight;

for(size\_t i = 0; i < N; i++)

{

cin >> from >> to >> weight;

graf.push(from, to, weight, true); //Добавляем реальное ребро

graf.push(to, from, 0, false); //Добавляем мнимое ребро

}

cout << "Представление графа в виде списков:" << endl;

graf.print(); //Вывод графа

cout << endl;

Нахождение максимального потока

cout << "Нахождение максимального потока." << endl;

int max\_flow = 0;

int iter\_res;

int i = 1;

cout << "\_\_\_\_Круг #" << i++ << endl;

while((iter\_res = graf.find\_flow(graf.find\_or\_add(start), graf.find\_or\_add(finish), INF, 0)) > 0) //Пока есть путь в графе

{

cout << "\_\_\_\_Круг #" << i++ << endl;

graf.unvisit(); //Пометим ребра непройденными

max\_flow += iter\_res; //К итоговому потоку добавим найденный

}

cout << max\_flow << endl; //Выводим максимальный поток

graf.print\_flow(); //Выводим ребра с их потоками

return 0;

}