**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Потоки в сети»**

|  |  |
| --- | --- |
| Студентка гр. 7381 | Машина Ю. Д. |
| 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.**

В первой строке указывается количество ориентированных рёбер в графе.

Во второй строке указывается исток .

В третьей строк указывается сток .

Далее в каждой строке указываются триплеты, задающие рёбра графа, причём – вершина, из которой ребро выходит, – вершина, в которую ребро входит, а – пропускная способность ребра (вес).

**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 .

В каждой последующей строке вывести триплеты рёбер: – откуда, – куда, – 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

**Индивидуализация.**

Вариант 5м. Матрица смежности. Поиск пути по правилу: каждый раз

выполняется переход по ребру, имеющему максимальную остаточную

пропускную способность.

**Описание алгоритма.**

1. Положим все потоки равными нулю.
2. Находим путь из истока в сток, каждый раз выбирая ребро с максимальной остаточной пропускной способностью; если такое не найдено, алгоритм заканчивает работу;
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 min\_path\_capacity;
5. Для каждого ребра на найденном пути увеличиваем поток на min\_path\_capacity, а в противоположном ему — уменьшаем на min\_path\_capacity;
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. Переход на шаг 2.

**Асимптотика алгоритма.**

В процессе работы алгоритма трудозатратой частью является поиск возрастающего пути, который, как известно, алгоритмом поиска в ширину можно сделать за .

Теперь необходимо найти число увеличений потока, которое составляет .

Таким образом, итоговая асимптотика работы алгоритма составляет .

Найдём затраты по памяти. В ходе работы алгоритма необходимо хранить матрицу смежности для графа: . Таким образом, асимптотика по памяти составляет .

**Описание функций и структур данных.**

Класс Graph используется для хранения графа в виде матрицы смежности.

int m\_capacity[N][N];

В двухмерном массиве хранится пропускная способность рёбер.

int m\_flow[N][N];

В двухмерном массиве хранятся потоки в сети.

bool m\_visited[N][N];

В двухмерном массиве помечаются использованные рёбра.

size\_t m\_parent[N];

В массив сохраняются предки вершин для восстановления пути.

add\_edge(size\_t u, size\_t v, size\_t c)

Добавление ребра в граф из вершины в вершину с пропускной способностью .

residual\_capacity(size\_t u, size\_t v)

Остаточная пропускная способность ребра . Вычисляется как разность между пропускной способностью и существующим потоком ребра.

augmenting\_path(size\_t source, size\_t sink)

Поиск возрастающего пути в графе из вершины в вершину с помощью поиска в ширину. Возвращаемое значение – тип : найден ли путь в графе.

find\_max\_flow(size\_t source, size\_t sink)

Функция нахождения максимального потока в графе из вершины в вершину . Возвращаемое значение – величина максимального потока.

print\_answer()

Функция вывода значений потоков в найденном максимальном потоке в сети. Функция ничего не возвращает.

**Тестирование.**

Таблица 1 – Результаты тестирования алгоритма.

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | 10  a  g  a b 20  a c 10  b d 30  c e 30  e g 40  e f 10  d c 20  d e 10  d f 40  f g 30 | 30  a b 20  a c 10  b d 20  c e 10  d c 0  d e 0  d f 20  e f 0  e g 10  f g 20 |
| 2 | 5  A  D  A B 7  B C 7  C A 7  C D 1  A D 6 | 7  A B 1  A D 6  B C 1  C A 0  C D 1 |
| 3 | 4  a  d  a b 8  a c 8  b d 8  c d 8 | 16  a b 8  a c 8  b d 8  c d 8 |
| 4 | 1  a  z  a z 1000 | 1000  a z 1000 |
| 5 | 2  a  c  b c 2  a b 3 | 2  a b 2  b c 2 |

Подробный разбор теста 1:

10

a

g

a b 20

a c 10

b d 30

c e 30

e g 40

e f 10

d c 20

d e 10

d f 40

f g 30

Searching for an augmenting path:

maximum residual capacity vertex for each member of the path:

visited: a

neighbor of a: b and residual capacity of [a; b] = 20 is better, new best vertex

is b

neighbor of a: c and residual capacity of [a; c] = 10 isn't better, best vertex

not changed

maximum residual capacity neighbor vertex of 'a' is found: b

visited: b

neighbor of b: d and residual capacity of [b; d] = 30 is better, new best vertex

is d

maximum residual capacity neighbor vertex of 'b' is found: d

visited: d

neighbor of d: c and residual capacity of [d; c] = 20 is better, new best vertex

is c

neighbor of d: e and residual capacity of [d; e] = 10 isn't better, best vertex

not changed

neighbor of d: f and residual capacity of [d; f] = 40 is better, new best vertex

is f

maximum residual capacity neighbor vertex of 'd' is found: f

visited: f

neighbor of f: g and residual capacity of [f; g] = 30 is better, new best vertex

is g

maximum residual capacity neighbor vertex of 'f' is found: g

<-- g <-- f <-- d <-- b <-- a <-- maximum flow is 20

Searching for an augmenting path:

maximum residual capacity vertex for each member of the path:

visited: a

neighbor of a: c and residual capacity of [a; c] = 10 is better, new best vertex

is c

maximum residual capacity neighbor vertex of 'a' is found: c

visited: c

neighbor of c: e and residual capacity of [c; e] = 30 is better, new best vertex

is e

maximum residual capacity neighbor vertex of 'c' is found: e

visited: e

neighbor of e: f and residual capacity of [e; f] = 10 is better, new best vertex

is f

neighbor of e: g and residual capacity of [e; g] = 40 is better, new best vertex

is g

maximum residual capacity neighbor vertex of 'e' is found: g

<-- g <-- e <-- c <-- a <-- maximum flow is 10

Searching for an augmenting path:

maximum residual capacity vertex for each member of the path:

visited: a

maximum residual capacity neighbor vertex of 'a' not found

30

a b 20

a c 10

b d 20

c e 10

d c 0

d e 0

d f 20

e f 0

e g 10

f g 20

**Выводы.**

В ходе выполнения лабораторной работы был изучен алгоритм нахождения максимального потока в графе: алгоритм Форда-Фалкерсона, который позволяет за найти максимальный поток в сети.

Для хранения графа использовалась матрица смежности.

**ПРИЛОЖЕНИЕ А**

**КОД ОСНОВНОЙ ПРОГРАММЫ**

#include "pch.h"

#include <iostream>

#include <algorithm>

#include <queue>

template <size\_t N>

class Network {

public:

Network() { //Обнуляем все потоки

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

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

m\_capacity[i][j] = 0;

m\_flow[i][j] = 0;

}

}

void add\_edge(size\_t vertex1, size\_t vertex2, int capacity) {

m\_capacity[vertex1][vertex2] = capacity;

}

int find\_max\_flow(size\_t source, size\_t sink) {

int max\_flow = 0;

while (augmenting\_path(source, sink)) {

size\_t vertex = sink;

int min\_path\_capacity = std::numeric\_limits<int>::max();

while (vertex != source) {// ищу минимальную остаточную пропускную способность на возрастающем пути

min\_path\_capacity = std::min(min\_path\_capacity, residual\_capacity(m\_parent[vertex], vertex));

vertex = m\_parent[vertex];

}

vertex = sink;

while (vertex != source) {

m\_flow[m\_parent[vertex]][vertex] += min\_path\_capacity;

m\_flow[vertex][m\_parent[vertex]] -= min\_path\_capacity;

vertex = m\_parent[vertex];

}

max\_flow += min\_path\_capacity; // обновляю максимальный поток

}

return max\_flow;

}

bool augmenting\_path(size\_t source, size\_t sink) // возрастающий путь (увеличивающий путь или увеличивающая цепь) из истока в сток с ненулевой минимальной остатточной пропускной способностью

{

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

{

m\_visited[i] = false;

}

size\_t vertex = source;

while (vertex != sink)

{

m\_visited[vertex] = true;

// Вершина, в которую идёт ребро с максимальной остаточной пропускной способностью (по умолчанию 0, потом может измениться)

size\_t max\_residual\_capacity\_vertex = 0;

// Флаг: найдено ли ребро с положительной остаточной пропускной способностью

bool max\_residual\_capacity\_vertex\_found = false;

// перебираем всех потенциальный соседей

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

{

// Если вершина ещё не посещена

// Пропускная способность положительна

if (!m\_visited[i] && residual\_capacity(vertex, i) > 0)

{

// как минимум один переход нашли

max\_residual\_capacity\_vertex\_found = true;

if (residual\_capacity(vertex, i) > residual\_capacity(vertex, max\_residual\_capacity\_vertex))

{

// нашлось ребро получше

max\_residual\_capacity\_vertex = i;

}

}

}

if (max\_residual\_capacity\_vertex\_found)

{

m\_parent[max\_residual\_capacity\_vertex] = vertex;

vertex = max\_residual\_capacity\_vertex;

}

else

{

if (vertex == source)

{

return (false);

}

vertex = m\_parent[vertex];

}

}

return (true);

}

void print\_answer()

{

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

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

if (m\_capacity[i][j] > 0)

std::cout << char(i) << " " << char(j) << " " << std::max(m\_flow[i][j], 0) << std::endl; // вывести 0 вместо отрицательного потока

}

~Network() {

}

private:

int residual\_capacity(size\_t vertex1, size\_t vertex2) {

return m\_capacity[vertex1][vertex2] - m\_flow[vertex1][vertex2];

}

int m\_capacity[N][N];

int m\_flow[N][N];

size\_t m\_parent[N];

bool m\_visited[N];

};

int main()

{

Network<300> graph;

size\_t N;

char source, sink;

std::cin >> N >> source >> sink;

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

char v, u;

std::cin >> v >> u;

int capacity;

std::cin >> capacity;

graph.add\_edge(v, u, capacity);

}

std::cout << graph.find\_max\_flow(source, sink) << std::endl;

graph.print\_answer();

return 0;

}

**ПРИЛОЖЕНИЕ Б**

**КОД ОСНОВНОЙ ПРОГРАММЫ С ПОДРОБНЫМ ВЫВОДОМ**

#include "pch.h"

#include <iostream>

#include <algorithm>

#include <queue>

template <size\_t N>

class Network {

public:

Network() { //Обнуление всех потоков

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

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

m\_capacity[i][j] = 0;

m\_flow[i][j] = 0;

}

}

void add\_edge(size\_t vertex1, size\_t vertex2, int capacity) {

m\_capacity[vertex1][vertex2] = capacity;

}

int find\_max\_flow(size\_t source, size\_t sink) {

int max\_flow = 0;

while (augmenting\_path(source, sink)) {

std::cout << " <-- " << char(sink) << " <-- ";

size\_t vertex = sink;

int min\_path\_capacity = std::numeric\_limits<int>::max();

while (vertex != source) {// ищу минимальную остаточную пропускную способность на возрастающем пути

min\_path\_capacity = std::min(min\_path\_capacity, residual\_capacity(m\_parent[vertex], vertex));

vertex = m\_parent[vertex];

std::cout << char(vertex) << " <-- ";

}

std::cout << "maximum flow is " << min\_path\_capacity << std::endl;

vertex = sink;

while (vertex != source) {

m\_flow[m\_parent[vertex]][vertex] += min\_path\_capacity;

m\_flow[vertex][m\_parent[vertex]] -= min\_path\_capacity;

vertex = m\_parent[vertex];

}

max\_flow += min\_path\_capacity; // обновляю максимальный поток

}

return max\_flow;

}

bool augmenting\_path(size\_t source, size\_t sink) // возрастающий путь (увеличивающий путь или увеличивающая цепь) из истока в сток с ненулевой минимальной остатточной пропускной способностью

{

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

{

m\_visited[i] = false;

}

std::cout << "\nSearching for an augmenting path:\nmaximum residual capacity vertex for each member of the path:\n";

size\_t vertex = source;

while (vertex != sink)

{

m\_visited[vertex] = true;

std::cout << "\nvisited: "<< char(vertex)<<"\n";

// Вершина, в которую идёт ребро с максимальной остаточной пропускной способностью (по умолчанию 0)

size\_t max\_residual\_capacity\_vertex = 0;

bool max\_residual\_capacity\_vertex\_found = false;

// перебор всех потенциальный соседей

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

{

// Если вершина ещё не посещена, пропускная способность положительна

if (!m\_visited[i] && residual\_capacity(vertex, i) > 0)

{

std::cout << "neighbor of "<<char(vertex)<< ": " << char(i) << " and ";

// как минимум один переход есть

max\_residual\_capacity\_vertex\_found = true;

std::cout << "residual capacity of [" << char(vertex) << "; " << char(i) << "] = "<< residual\_capacity(vertex, i);

//std::cout << "and";

if (residual\_capacity(vertex, i) > residual\_capacity(vertex, max\_residual\_capacity\_vertex))

{

// нашлось ребро получше

max\_residual\_capacity\_vertex = i;

std::cout << " is better, new best vertex is "<< char(max\_residual\_capacity\_vertex)<<"\n";

}

else std::cout << " isn't better, best vertex not changed\n";

}

}

if (max\_residual\_capacity\_vertex\_found)

{

std::cout << "maximum residual capacity neighbor vertex of '" << char(vertex) << "' is found: " << char(max\_residual\_capacity\_vertex) << "\n\n";

m\_parent[max\_residual\_capacity\_vertex] = vertex;

vertex = max\_residual\_capacity\_vertex;

}

else

{

std::cout << "maximum residual capacity neighbor vertex of '"<< char(vertex)<<"' not found\n\n";

if (vertex == source)

{

return (false);

}

vertex = m\_parent[vertex];

}

}

return (true);

}

void print\_answer()

{

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

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

if (m\_capacity[i][j] > 0)

std::cout << char(i) << " " << char(j) << " " << std::max(m\_flow[i][j], 0) << std::endl; // вывести 0 вместо отрицательного потока

}

~Network() {

}

private:

int residual\_capacity(size\_t vertex1, size\_t vertex2) {

return m\_capacity[vertex1][vertex2] - m\_flow[vertex1][vertex2];

}

int m\_capacity[N][N];

int m\_flow[N][N];

size\_t m\_parent[N];

bool m\_visited[N];

};

int main()

{

Network<300> graph;

size\_t N;

char source, sink;

std::cin >> N >> source >> sink;

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

char v, u;

std::cin >> v >> u;

int capacity;

std::cin >> capacity;

graph.add\_edge(v, u, capacity);

}

std::cout << graph.find\_max\_flow(source, sink) << std::endl;

graph.print\_answer();

return 0;

}