**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра CАПР**

ОТЧЕТ

по курсовой работе

по дисциплине «Алгоритмы и структуры данных»

Вариант 2

ТЕМА: Поток в сетях

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Головатюк К.А. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2022

### Постановка задачи

Входные данные: текстовый файлы со строками в формате V1, V2, P, где V1, V2 направленная дуга транспортной сети, а P – ее пропускная способность. Исток всегда обозначен как S, сток – как T

Пример файла для сети с изображения выше:

S O 3

S P 3

O Q 3

O P 2

P R 2

Q R 4

Q T 2

R T 3

Найти максимальный поток в сети используя алгоритм:

Вариант 2. Эдмондса — Карпа.

### Цель работы

Научиться реализовывать алгоритм нахождения потока в сети. Использовать принципы модульности и ООП. Отточить навыки юнит-тестирования в процессе проверки корректности написанных методов.

### Описание реализуемых классов и алгоритмов

**EdmonsKarp** – Класс реализующий алгоритм Эдмондса — Карпа.

**Graph** – класс хранящий в себе список всех вершин.

**Queue** – класс очереди.

**Vertex** – класс вершины, содержит список ребер исходящих из этой вершины.

**Edge** – класс ребра, содержит указатель на вершину в которое ведет ребро.

**Алгоритм Эдмондса — Карпа.:**

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.

2.В остаточной сети находим *кротчайший путь* из источника в сток. Если такого пути нет, выходим из цикла.

3.Пускаем через найденный путь максимально возможный поток:

4.На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью.

5.Для каждого ребра на найденном пути увеличиваем поток на минимальную пропускную способность, а в противоположном ему — уменьшаем на нее.

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.Возвращаемся на шаг 2.

Отличие алгоритма Эдмондса — Карпа от Форда — Фалкерсона в том, что ищется кратчайший путь, а не любой.

**Алгоритм поиска пути:**

Поиск пути реализован через обход в ширину:

1. Добавляем в очередь начальную вершину.

2. Достаем из очереди самую верхнюю вершину.

3. Добавляем и помечаем как посещенный все смежные вершины, которые не посещены и вес ребра до них больше 0.

4. Повторяем пункты 2 и 3 пока есть вершины в очереди или пока не придем в конечную вершину.

### Описание реализуемых методов класса EdmonsKarp.

|  |  |
| --- | --- |
| **Название метода** | **Описание** |
| void read\_file(string path); | Создание графа по данным из файла. |
| int max\_flow(); | Поиск максимального потока. |
| void print\_graphs(); | Вывод в консоль начального и модифицированного графов. |

### Описание реализуемых методов класса Graph.

|  |  |
| --- | --- |
| **Название метода** | **Описание** |
| Vertex\* add\_vertex(char c); | Добавление вершины в граф. |
| Edge\* add\_edge(char a, char b, int flow); | Добавление ребра в граф. |
| Vertex\* get\_vertex(char c); | Поиск вершины в графе. |
| Edge\* get\_edge(char a, char b); | Поиск ребра в графе. |
| string find\_path(); | Поиск пути от истока в сток. |
| **Название приватного метода** |  |
| void push\_queue(Vertex\* vert, string way); | Добавление вершины в очередь. |
| Queue\* pop(); | Извлечение вершины из очереди. |

### Описание реализованных unit-тестов

Для проверки были написаны unit-тесты для всех публичных методов.

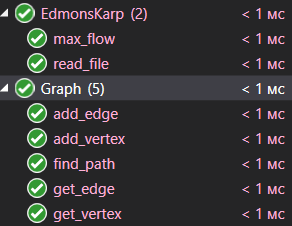


Рисунок 1. Список тестов и результаты тестирования.

### Пример работы программы

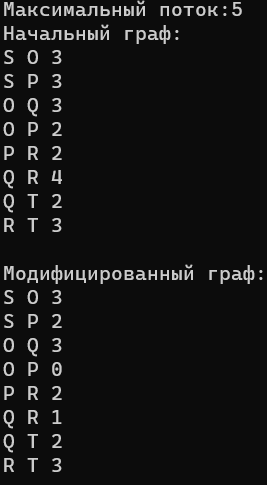


Рисунок 2. Пример работы.

### Вывод

Был изучен алгоритм Эдмондса — Карпа. При реализации были улучшены навыки разработки структур данных, реализации алгоритмов и создания тестов.

### Листинг

Файл main.cpp:

#include <iostream>

#include "EdmonsKarp.h"

using namespace std;

int main() {

setlocale(LC\_ALL, "Russian");

EdmonsKarp test;

test.read\_file("input.txt");

cout << "Максимальный поток:" << test.max\_flow() << endl;

test.print\_graphs();

}

Файл EdmonsKarp.h:

#pragma once

#include <fstream>

#include <iostream>

#include "Graph.h"

using namespace std;

class EdmonsKarp

{

public:

void read\_file(string path);

int max\_flow();

Graph network;

Graph residual\_network;

void print\_graphs();

};

Файл EdmonsKarp.cpp:

#include "EdmonsKarp.h"

void EdmonsKarp::read\_file(string path)

{

ifstream file(path);

if (!file.is\_open())

{

cout << "ERORR: FILE" << endl;

return;

}

string str;

char a, b;

int flow;

while (file >> a >> b >> flow)

{

network.add\_edge(a, b, flow);

residual\_network.add\_edge(a, b, flow);

}

file.close();

}

int EdmonsKarp::max\_flow()

{

string path;

path = residual\_network.find\_path();

while (!path.empty())

{

int min = residual\_network.get\_edge(path[0], path[1])->flow;

for (int i = 0; i < path.size() - 1; i++) {

int a = residual\_network.get\_edge(path[i], path[i + 1])->flow;

if (a < min)

min = a;

}

for (int u = 0, v = 1; u < path.size() - 1; u++, v++)

{

residual\_network.get\_edge(path[u], path[v])->flow -= min;

if (residual\_network.get\_edge(path[v], path[u]))

residual\_network.get\_edge(path[v], path[u])->flow += min;

else

residual\_network.add\_edge(path[v], path[u], min);

}

path = residual\_network.find\_path();

}

Edge\* a = network.get\_vertex('S')->edgies;

int flow = 0;

while (a)

{

flow += a->flow - residual\_network.get\_edge('S', a->name)->flow;

a = a->next;

}

return flow;

}

void EdmonsKarp::print\_graphs()

{

cout << "Начальный граф:" << endl;

Vertex\* vert = network.head;

while (vert)

{

Edge\* edge = vert->edgies;

while (edge)

{

cout << vert->name << " " << edge->name << " " << edge->flow << endl;

edge = edge->next;

}

vert = vert->next;

}

cout << endl << "Модифицированный граф:" << endl;

vert = network.head;

while (vert)

{

Edge\* edge = vert->edgies;

while (edge)

{

cout << vert->name << " " << edge->name << " "

<< edge->flow - residual\_network.get\_edge(vert->name, edge->name)->flow << endl;

edge = edge->next;

}

vert = vert->next;

}

}

Файл Graph.h:

#pragma once

#include <string>

using namespace std;

class Vertex;

class Queue {

public:

Queue() : vert(nullptr), next(nullptr) {};

Vertex\* vert;

string way;

Queue\* next;

};

class Edge {

public:

Edge(char c) : name(c), visited(false), flow(0), vert(nullptr), next(nullptr) {};

char name;

int flow;

bool visited;

Vertex\* vert;

Edge\* next;

};

class Vertex {

public:

Vertex(char c) : name(c), visited(false), next(nullptr), edgies(nullptr) {};

char name;

bool visited;

Edge\* edgies;

Vertex\* next;

};

class Graph {

public:

Graph() : queue(nullptr), head(nullptr) {};

Vertex\* add\_vertex(char c);

Edge\* add\_edge(char a, char b, int flow);

Vertex\* get\_vertex(char c);

Edge\* get\_edge(char a, char b);

string find\_path();

Vertex\* head;

private:

void push\_queue(Vertex\* vert, string way);

Queue\* pop();

Queue\* queue;

};

Файл Graph.cpp:

#include "Graph.h"

Vertex\* Graph::add\_vertex(char c)

{

Vertex\* buff = get\_vertex(c);

if (buff)

return buff;

if (head == nullptr)

{

head = new Vertex(c);

return head;

}

Vertex\* iter = head;

while (iter->next)

{

iter = iter->next;

}

iter->next = new Vertex(c);

return iter->next;

}

Vertex\* Graph::get\_vertex(char c)

{

Vertex\* buff = head;

while (buff)

{

if (buff->name == c)

return buff;

buff = buff->next;

}

return nullptr;

}

Edge\* Graph::get\_edge(char a, char b)

{

Edge\* buff = get\_vertex(a)->edgies;

while (buff)

{

if (buff->name == b)

return buff;

buff = buff->next;

}

return nullptr;

}

Edge\* Graph::add\_edge(char a, char b, int flow)

{

Vertex\* vert\_a = add\_vertex(a);

Vertex\* vert\_b = add\_vertex(b);

if (vert\_a->edgies == nullptr)

{

vert\_a->edgies = new Edge(b);

vert\_a->edgies->vert = vert\_b;

vert\_a->edgies->flow = flow;

return vert\_a->edgies;

}

Edge\* iter = vert\_a->edgies;

while (iter->next)

{

iter = iter->next;

}

iter->next = new Edge(b);

iter->next->flow = flow;

iter->next->vert = vert\_b;

return iter->next;

}

string Graph::find\_path()

{

Vertex\* node = head;

while (node)

{

node->visited = false;

Edge\* edge = node->edgies;

while (edge)

{

edge->visited = false;

edge = edge->next;

}

node = node->next;

}

delete queue;

queue = nullptr;

push\_queue(get\_vertex('S'), "S");

Queue\* curent = pop();

curent->vert->visited = true;

while (curent) {

if (curent->vert->name == 'T')

break;

Edge\* edge = curent->vert->edgies;

while (edge)

{

edge->visited = true;

if ((edge->vert->visited == false) && (edge->flow > 0)) {

push\_queue(edge->vert, curent->way + edge->name);

edge->vert->visited == true;

}

edge = edge->next;

}

curent = pop();

}

if (curent)

return curent->way;

return "";

}

void Graph::push\_queue(Vertex\* vert, string way)

{

if (queue == nullptr) {

queue = new Queue();

queue->vert = vert;

queue->way = way;

return;

}

Queue\* iter = queue;

while (iter->next)

{

iter = iter->next;

}

iter->next = new Queue();

iter->next->vert = vert;

iter->next->way = way;

}

Queue\* Graph::pop()

{

if (queue == nullptr)

return nullptr;

Queue\* buff = queue;

if (queue->next)

queue = queue->next;

else

queue = nullptr;

return buff;

}

Файл test.cpp:

#include "pch.h"

#include "EdmonsKarp.h"

TEST(EdmonsKarp, read\_file) {

EdmonsKarp test;

test.read\_file("input.txt");

Vertex\* it\_v = test.network.head;

char a[] = { 'S', 'O', 'P', 'Q', 'R', 'T' };

int pos\_a = 0;

while (it\_v)

{

ASSERT\_EQ(it\_v->name, a[pos\_a]);

it\_v = it\_v->next;

pos\_a++;

}

}

TEST(EdmonsKarp, max\_flow) {

EdmonsKarp test;

test.read\_file("input.txt");

ASSERT\_EQ(test.max\_flow(), 5);

}

TEST(Graph, add\_vertex) {

Graph test;

test.add\_vertex('a');

test.add\_vertex('b');

test.add\_vertex('c');

test.add\_vertex('d');

Vertex\* it\_v = test.head;

ASSERT\_EQ(it\_v->name, 'a');

it\_v = it\_v->next;

ASSERT\_EQ(it\_v->name, 'b');

it\_v = it\_v->next;

ASSERT\_EQ(it\_v->name, 'c');

it\_v = it\_v->next;

ASSERT\_EQ(it\_v->name, 'd');

}

TEST(Graph, add\_edge) {

Graph test;

test.add\_edge('a', 'b', 0);

test.add\_edge('a', 'c', 0);

test.add\_edge('b', 'c', 0);

Vertex\* it\_v = test.head;

ASSERT\_EQ(test.head->edgies->name, 'b');

ASSERT\_EQ(test.head->edgies->next->name, 'c');

ASSERT\_EQ(test.head->next->edgies->name, 'c');

}

TEST(Graph, get\_vertex) {

Graph test;

test.add\_vertex('a');

test.add\_vertex('b');

test.add\_vertex('c');

test.add\_vertex('d');

ASSERT\_EQ(test.get\_vertex('a')->name, 'a');

ASSERT\_EQ(test.get\_vertex('b')->name, 'b');

ASSERT\_EQ(test.get\_vertex('c')->name, 'c');

ASSERT\_EQ(test.get\_vertex('d')->name, 'd');

}

TEST(Graph, get\_edge) {

Graph test;

test.add\_edge('a', 'b', 0);

test.add\_edge('a', 'c', 0);

test.add\_edge('b', 'c', 0);

Vertex\* it\_v = test.head;

ASSERT\_EQ(test.get\_edge('a', 'b')->name, 'b');

ASSERT\_EQ(test.get\_edge('a', 'c')->name, 'c');

ASSERT\_EQ(test.get\_edge('b', 'c')->name, 'c');

}

TEST(Graph, find\_path) {

Graph test;

test.add\_edge('S', 'O', 1);

test.add\_edge('S', 'P', 1);

test.add\_edge('O', 'Q', 1);

test.add\_edge('O', 'P', 1);

test.add\_edge('P', 'R', 1);

test.add\_edge('Q', 'R', 1);

test.add\_edge('Q', 'T', 1);

test.add\_edge('R', 'T', 1);

ASSERT\_STREQ(test.find\_path().c\_str(), "SOQT");

}