**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

Курсовая РАБОТА

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

Тема: Потоки в сетях (вар.1)

|  |  |  |
| --- | --- | --- |
| Студент гр. 7302 |  | Карманов Д.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2019

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Карманов Д.А. | | |
| Группа 7302 | | |
| Тема работы: Нахождение максимального потока в сети | | |
| Исходные данные:  Список вершин и пропускных способностей между ними | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Постановка задачи», «Обоснование выбора используемых данных», «Описание алгоритма решения», «Пример работы», «Листинг», «Заключение», «Список использованных источников» | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страницы. | | |
| Дата выдачи задания: | | |
| Дата сдачи реферата: | | |
| Дата защиты реферата: | | |
| Студент |  | Карманов Д.А. |
| Преподаватель |  | Тутуева А.В. |

**Аннотация**

В данной курсовой работе рассматривается задача о нахождении максимального потока в сети. Для реализации поиска используется алгоритм Форда-Фалкерсона, работающего с остаточными пропускными способностями ребер графа.

**Summary**

In the course work, the problem of finding the maximum flow in web is reviewed. To implement the search, the Ford-Fulkerson algorithm is used, which works with residual capacities of the graph edges.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Постановка задачи | 6 |
| 2. | Обоснование выбранных структур данных | 6 |
| 3. | Описание алгоритма решения | 7 |
| 4. | Пример работы программы | 8 |
| 5. | Листинг | 11 |
|  | Заключение | 25 |
|  | Список использованных источников | 26 |

**введение**

Алгоритм Форда-Фалкерсона – это алгоритм нахождения максимального потока в сети.

Суть алгоритма:

1. Остаточные пропускные способности всех ребер приравниваются к их первоначальным пропускным способностям;

2. В цикле

1. Текущий узел – исток -> шаг 2;

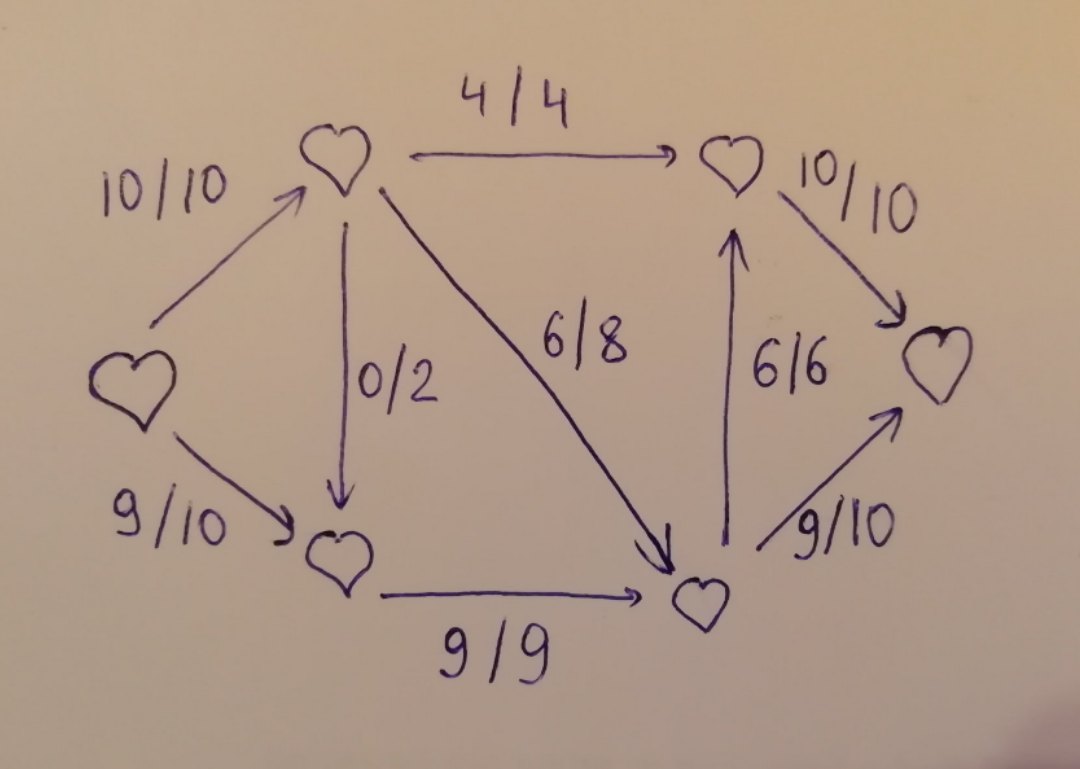
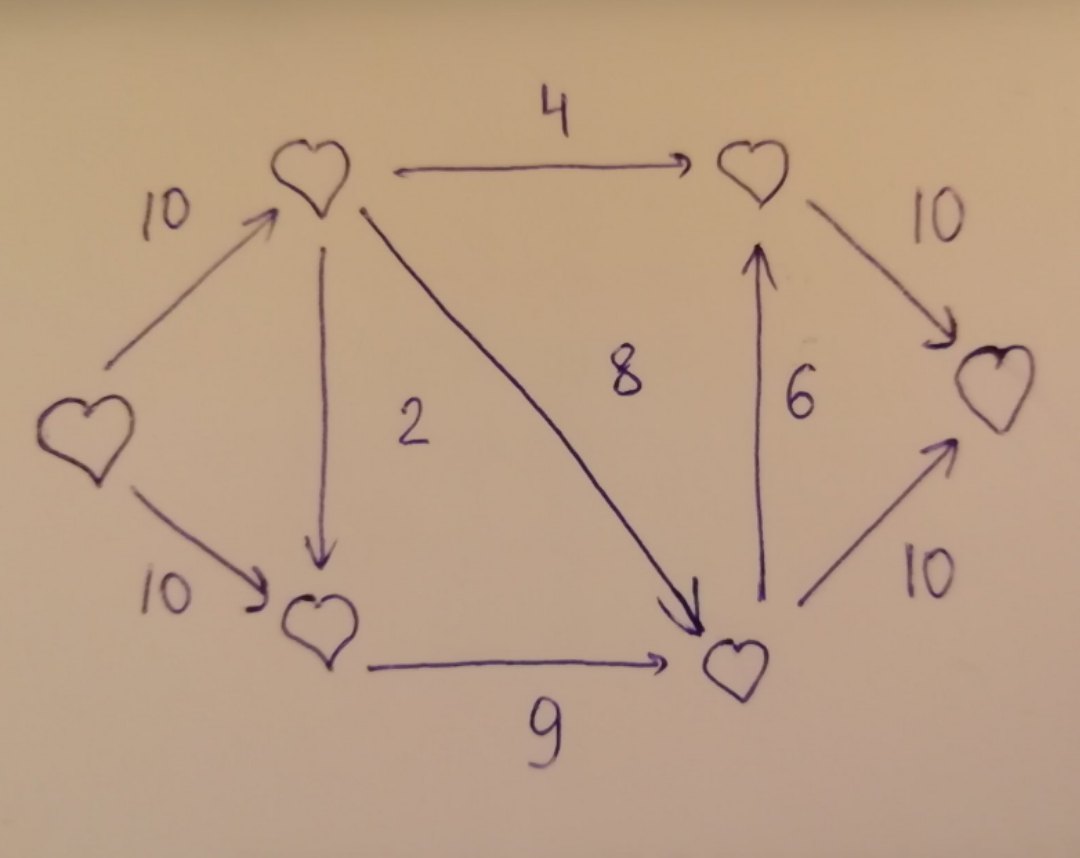
2. Ищем у текущей вершины смежные вершины, у которых нет метки и остаточная пропускная способность ребра до них не равна 0. Если таких нет -> шаг 4, иначе -> шаг 3;

3. Ищем среди смежных вершин вершину, до которой ребро имеет минимальную пропускную способность, переходим к ней. Если такая вершина – сток, то -> шаг 5, иначе -> шаг 2;

4. Если текущая вершина не исток, то переходим к предыдущей вершине и -> шаг 2, иначе выходим из цикла.

5. Находим путь от истока к стоку, идя по меткам, вычисляя минимальный поток по этому пути. Найденный поток суммируем к максимальному потоку в сети, переходим к шагу 1.

*Пример*



Максимальный поток в этой сети равен 19

**1. Постановка задачи**

Построить сеть по исходным данным и найти максимальный поток.

**2. ОБОСНОВАНИЕ ВЫБРАННЫХ СТРУКТУР ДАННЫХ**

Для работы с графом используется структура Graph с полями-классами Vertex и Edge, и полями: LinkedList<Vertex\*> vertexes, LinkedList<Edges\*> edges и size\_t max\_flow.

Также входят методы:

1. private

bool check\_vertex(); //Проверяет есть ли вершина в списке вершин

bool check\_edge(); //Проверяет есть ли ребро в списке ребер

void push\_vertex(); //Добавляет вершину в список вершин

void push\_edge(); //Добавляет ребро в список ребер

void set\_adjacent(); //Используется в step\_two() для поиска смежных к текущей вершине вершин

Vertex\* get\_vertex(); //Возвращает указатель на вершину в списке

void step\_one(); //Первый шаг в алгоритме Форда-Фалкерсона

void step\_two(); //Второй шаг…

void step\_three(); //Третий шаг…

void step\_four(); //Четвертый шаг…

void step\_five(); //Пятый шаг…

2. public

void build\_web();

void print\_web();

void Ford\_Fulk();

Причина выбора структуры данных связного списка в легкости его написания, читабельности кода и невысокой временной сложностью операций O(n).

**3. ОПИСАНИЕ АЛГОРИТМА РЕШЕНИЯ**

1. Считать входной файл с исходными данными – вершинами и пропускной способностью ребра между ними.

Ввести проверки на ввод:

1) Вершина задается одним заглавным символом;

2) Есть исток и сток;

3) Нет циклов.

2. Запустить алгоритм Форда-Фалкерсона

2.1. Установить метку на исток;

2.2. В цикле

2.2.1. Текущий узел – исток;

2.2.2. Ищем у текущей вершины смежные вершины, у которых нет метки и остаточная пропускная способность ребра до них не равна 0. Если таких нет -> шаг 4, иначе -> шаг 3;

2.2.3. Ищем среди смежных вершин вершину, до которой ребро имеет минимальную пропускную способность, переходим к ней. Если такая вершина – сток, то -> шаг 5, иначе -> шаг 2;

2.2.4. Если текущая вершина не исток, то переходим к предыдущей вершине и -> шаг 2, иначе выходим из цикла.

2.2.5. Находим путь от истока к стоку, идя по меткам, вычисляя минимальный поток по этому пути. Найденный поток суммируем к максимальному потоку в сети, переходим к шагу 1.

**4. ПРИМЕР РАБОТЫ ПРОГРАММЫ**

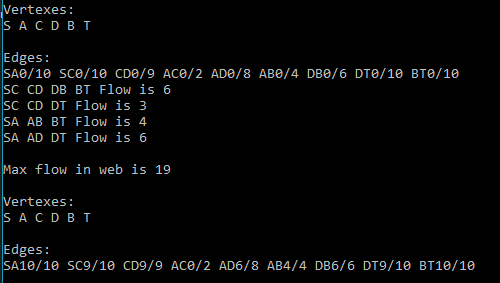
Программа выводит граф по типу «Вершина1\_Вершина2\_Текущий поток через ребро\_Максимальный поток через ребро»

*Пример 1*

Исходные данные



Результат работы

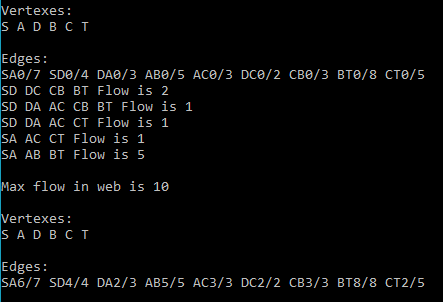


*Пример 2*

Исходные данные



Результат работы

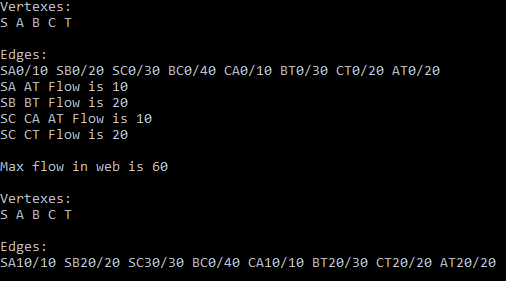


*Пример 3*

Исходные данные



Результат работы

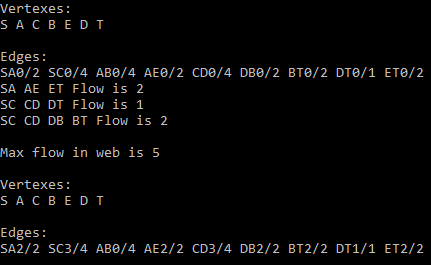


*Пример 4*

Исходные данные



Результат работы



**5. ЛИСТИНГ**

*Ford\_Fulk.h*

#pragma once

#include "..//Alg\_sem4\_l1/LinkedList.h"

#include "..//Alg\_sem4\_l1/LinkedList.cpp"

#include <fstream>

class Web

{

class Vertex

{

char title{}; //vertex's name

size\_t flow = 0; //flow from prev vertex to the vertex

Vertex\* prev\_vert{}; //prev vertex for the vertex

public:

char get\_title() { return title; };

void set\_title(char title) { this->title = title; };

size\_t get\_flow() { return flow; };

void set\_flow(int flow) { this->flow = flow; };

Vertex\* get\_prev\_vert() { return prev\_vert; };

void set\_prev\_vert(Vertex \*prev\_vert) { this->prev\_vert = prev\_vert; };

Vertex(char title) { this->title = title; };

~Vertex() {};

};

class Edge

{

//Vertices adjacent to this edge

Vertex \*vertex1{};

Vertex \*vertex2{};

//residual and primary bandwidth

int resud\_bw{};

int prim\_bw{};

public:

Vertex\* get\_vertex1() { return vertex1; }

Vertex\* get\_vertex2() { return vertex2; }

int get\_resud\_bw() { return resud\_bw; }

void set\_resud\_bw(int resud\_bw) { this->resud\_bw = resud\_bw; };

int get\_prim\_bw() { return prim\_bw; }

void set\_prim\_bw(int prim\_bw) { this->prim\_bw = prim\_bw; };

Edge(Vertex\* vertex1, Vertex \*vertex2, int prim\_bw, int resud\_bw)

{

this->vertex1 = vertex1; this->vertex2 = vertex2; this->prim\_bw = prim\_bw; this->resud\_bw = resud\_bw;

};

~Edge() {};

};

//LinkedList<Vertex\*> adjacents; //for tests

LinkedList<Edge\*> edges;

LinkedList<Vertex\*> vertexes;

size\_t max\_flow{};

bool check\_vertex(char title, LinkedList<Vertex\*> \*vertexes);

bool check\_edge(char title1, char title2, LinkedList<Edge\*> \*edges);

void push\_vertex(char title, LinkedList<Vertex\*> \*vertexes);

void push\_edge(char title1, char title2, int prim\_bw);

void set\_adjacent(LinkedList<Vertex\*> \*vertexes, char title) //resudal\_wb > 0 and vertex hasn't mark

{

auto \*cur = edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == title && cur->data->get\_vertex2()->get\_flow() == 0

&& cur->data->get\_vertex2()->get\_prev\_vert() == nullptr && cur->data->get\_resud\_bw() > 0)

{

vertexes->push\_back(cur->data->get\_vertex2());

}

cur = cur->next;

}

}

Vertex\* get\_vertex(char title)

{

auto \*cur = vertexes.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_title() == title)

return cur->data;

cur = cur->next;

}

return nullptr;

}

void step\_one(); //Current node is S

void step\_two(Vertex \*vertex); //search for adjacent vertexes

void step\_three(Vertex \*vertex, LinkedList<Vertex\*> \*vertexes); //take vertex with minimal resudal\_wb and go to it

void step\_four(Vertex \*vertex, LinkedList<Vertex\*> \*adjacent); //roll back

void step\_five(Vertex \*vertex); //Residual path determination

public:

Web() {};

~Web() { edges.clear(); vertexes.clear(); };

void build\_web();

void print\_web();

void Ford\_Fulk();

*Ford\_Fulk.cpp*

#include "pch.h"

#include "Ford\_Fulk.h"

using namespace std;

bool CHECK\_S = 0;

bool CHECK\_T = 0;

bool is\_cap\_symbol(char symbol)

{

if (symbol >= 'A' && symbol <= 'Z')

return true;

else return false;

}

void Web::print\_web()

{

try {

if (vertexes.get\_head() == nullptr)

throw exception("Doesn't exsist");

auto \*cur = vertexes.get\_head();

cout << "\n\nVertexes:\n";

while (cur != nullptr)

{

cout << cur->data->get\_title() << " ";

cur = cur->next;

}

cout << endl;

cout << "\nEdges:\n";

auto \*cur1 = edges.get\_head();

while (cur1 != nullptr)

{

cout << cur1->data->get\_vertex1()->get\_title() << cur1->data->get\_vertex2()->get\_title() << cur1->data->get\_prim\_bw() - cur1->data->get\_resud\_bw();

cout << "/" << cur1->data->get\_prim\_bw() << " ";

cur1 = cur1->next;

}

cout << endl;

}

catch (exception)

{

return;

}

}

bool Web::check\_vertex(char title, LinkedList<Vertex\*> \*vertexes)

{

if (vertexes == nullptr)

return false;

auto \*cur = vertexes->get\_head();

while (cur != nullptr) {

if (cur->data->get\_title() == title)

return true;

cur = cur->next;

}

return false;

}

bool Web::check\_edge(char title1, char title2, LinkedList<Edge\*> \*edges)

{

if (edges == nullptr)

return false;

auto \*cur = edges->get\_head();

while (cur != nullptr) {

if (cur->data->get\_vertex1()->get\_title() == title1 && cur->data->get\_vertex2()->get\_title() == title2)

return true;

cur = cur->next;

}

return false;

}

void Web::push\_vertex(char title, LinkedList<Vertex\*> \*vertexes)

{

if (!check\_vertex(title, vertexes)) {

Vertex \*vertex = new Vertex(title);

vertexes->push\_back(vertex);

}

}

void Web::push\_edge(char title1, char title2, int prim\_bw)

{

if (!check\_edge(title1, title2, &edges)) {

Vertex \*vertex1 = get\_vertex(title1);

Vertex \*vertex2 = get\_vertex(title2);

Edge \*edge = new Edge(vertex1, vertex2, prim\_bw, prim\_bw);

edges.push\_back(edge);

}

}

void Web::build\_web()

{

ifstream inputFile("web.txt", ios::in);

if (!inputFile.is\_open())

return;

string text;

char title1;

char title2;

size\_t flow;

size\_t count = 0; //counter

while (!inputFile.eof())

{

getline(inputFile, text);

for (size\_t i = 0; i < text.length(); i++)

{

try {

if (!is\_cap\_symbol(text[i]) && count != 2)

throw exception("Incorrect input");

if (count == 0 && is\_cap\_symbol(text[i])) {

title1 = text[i];

count++;

}

else if (count == 1 && is\_cap\_symbol(text[i])) {

title2 = text[i];

count++;

}

else if (count == 2)

{

string flow\_s = "";

while (text[i] != '\0') {

flow\_s += text[i];

i++;

}

flow = stoi(flow\_s);

try {

if (title1 == title2)

throw exception("Cycle");

if (title1 == 'S' || title2 == 'S')

CHECK\_S = 1;

if (title1 == 'T' || title2 == 'T')

CHECK\_T = 1;

push\_vertex(title1, &vertexes);

push\_vertex(title2, &vertexes);

push\_edge(title1, title2, flow);

title1 = title2 = '\0';

count = 0;

}

catch (exception)

{

exit(-1);

}

}

}

catch(exception)

{

exit(-1);

}

}

}

inputFile.close();

}

void Web::Ford\_Fulk()

{

try {

if (CHECK\_S == 0 || CHECK\_T == 0)

throw exception("S or T don't exist");

//Initialization

auto \*cur = edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

break;

}

cur = cur->next;

}

step\_one();

cout << "\nMax flow in web is " << max\_flow;

}

catch (exception)

{

return;

}

}

void Web::step\_one() //current node is S

{

auto \*cur1 = vertexes.get\_head();

while (cur1->data->get\_title() != 'S')

cur1 = cur1->next;

step\_two(cur1->data);

}

void Web::step\_two(Vertex\* vertex) //search for adjacent vertexes

{

LinkedList<Vertex\*> adjacent;

set\_adjacent(&adjacent, vertex->get\_title());

if (adjacent.get\_size() != 0)

step\_three(vertex, &adjacent);

else step\_four(vertex, &adjacent);

}

void Web::step\_three(Vertex \*vertex, LinkedList<Vertex\*> \*vertexes) //take vertex with minimal resudal\_wb and go to it

{

auto \*cur1 = edges.get\_head();

auto \*cur = vertexes->get\_head();

auto \*temp = cur1;

int min = 999999999;

//Search for minimal resudal

while (cur != nullptr)

{

while (cur1 != nullptr)

{

if (cur1->data->get\_vertex1()->get\_title() == vertex->get\_title()

&& check\_vertex(cur1->data->get\_vertex2()->get\_title(), vertexes)

&& cur1->data->get\_resud\_bw() <= min)

{

min = cur1->data->get\_resud\_bw();

temp = cur1;

}

cur1 = cur1->next;

}

cur = cur->next;

}

Vertex \*k\_vertex = temp->data->get\_vertex2();

k\_vertex->set\_flow(temp->data->get\_resud\_bw());

k\_vertex->set\_prev\_vert(temp->data->get\_vertex1());

if (k\_vertex->get\_title() == 'T')

step\_five(k\_vertex);

else step\_two(k\_vertex);

}

void Web::step\_four(Vertex \*vertex, LinkedList<Vertex\*> \*adjacent) //Roll back

{

if(vertex->get\_title() != 'S')

step\_two(vertex->get\_prev\_vert());

}

void Web::step\_five(Vertex \*vertex) //Residual path determination

{

LinkedList<Edge\*> path;

Vertex \*temp = vertex;

size\_t min = temp->get\_flow();

//Determine the end - to - end path and find the minimum flow.

while (temp->get\_prev\_vert() != nullptr)

{

auto \*cur = edges.get\_head();

while (cur)

{

if (temp->get\_flow() < min)

min = temp->get\_flow();

if (cur->data->get\_vertex2() == temp && cur->data->get\_vertex1() == temp->get\_prev\_vert())

break;

cur = cur->next;

}

path.push\_front(cur->data);

temp = temp->get\_prev\_vert();

}

max\_flow += min;

//Сalculate the residual capacities along this path and remove the marks.

auto \*cur = path.get\_head();

while (cur != nullptr)

{

cout << cur->data->get\_vertex1()->get\_title() << cur->data->get\_vertex2()->get\_title() << " ";

cur = cur->next;

}

cout << "Flow is " << min << endl;

cur = path.get\_head();

while (cur)

{

cur->data->set\_resud\_bw(cur->data->get\_resud\_bw() - min);

if (cur->data->get\_vertex1()->get\_title() != 'S'){

cur->data->get\_vertex1()->set\_flow(0);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->get\_vertex2()->set\_flow(0);

cur->data->get\_vertex2()->set\_prev\_vert(nullptr);

cur = cur->next;

}

step\_one();

}

*Alg\_sem4\_coursework.cpp*

#include "pch.h"

#include <iostream>

#include "Ford\_Fulk.h"

using namespace std;

int main()

{

cout << "Karmanov Dmitry, gr.7302, v.1;\n";

Web web;

web.build\_web();

web.print\_web();

web.Ford\_Fulk();

web.print\_web();

}

*FordFulkTests.cpp*

#include "stdafx.h"

#include "CppUnitTest.h"

#include "..//Alg\_sem4\_coursework/Ford\_Fulk.h"

#include "..//Alg\_sem4\_coursework/Ford\_Fulk.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Ford\_Fulk\_Tests

{

TEST\_CLASS(FordFulk)

{

public:

TEST\_METHOD(check\_vertex\_adn\_edges)

{

Web web;

Assert::IsTrue(!web.check\_vertex('A', &web.vertexes));

Assert::IsTrue(!web.check\_edge('A', 'D', &web.edges));

web.push\_vertex('S', &web.vertexes);

Assert::IsTrue(web.check\_vertex('S', &web.vertexes));

Assert::IsTrue(!web.check\_vertex('D', &web.vertexes));

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

Assert::IsTrue(web.check\_edge('S', 'C', &web.edges));

Assert::IsTrue(!web.check\_edge('S', 'K', &web.edges));

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

Assert::IsTrue(web.check\_vertex('D', &web.vertexes));

Assert::IsTrue(web.check\_edge('D', 'B', &web.edges));

}

TEST\_METHOD(push\_vertex)

{

Web web1;

Web web2;

web1.push\_vertex('S', &web1.vertexes);

web1.push\_vertex('A', &web1.vertexes);

Assert::IsTrue(web1.vertexes.get\_size() == 2 && web2.vertexes.get\_size() == 0);

}

TEST\_METHOD(push\_edge)

{

Web web1;

Web web2;

web1.push\_vertex('S', &web1.vertexes);

web1.push\_vertex('A', &web1.vertexes);

web1.push\_vertex('D', &web1.vertexes);

web1.push\_edge('S', 'A', 10);

web1.push\_edge('A', 'D', 12);

Assert::IsTrue(web1.edges.get\_size() == 2 && web2.edges.get\_size() == 0);

}

TEST\_METHOD(set\_adjacent)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

auto \*cur = web.edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->set\_resud\_bw(cur->data->get\_prim\_bw());

cur->data->set\_prim\_bw(0);

cur = cur->next;

}

web.set\_adjacent(&web.adjacents, 'A');

Assert::IsTrue(web.adjacents.get\_size() == 3);

}

TEST\_METHOD(set\_adjacent\_empty)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

auto \*cur = web.edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->set\_resud\_bw(cur->data->get\_prim\_bw());

cur->data->set\_prim\_bw(0);

cur = cur->next;

}

web.set\_adjacent(&web.adjacents, 'T');

Assert::IsTrue(web.adjacents.get\_size() == 0);

}

TEST\_METHOD(step\_two)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

auto \*cur = web.edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->set\_resud\_bw(cur->data->get\_prim\_bw());

cur->data->set\_prim\_bw(0);

cur = cur->next;

}

web.set\_adjacent(&web.adjacents, 'A');

Assert::IsTrue(web.adjacents.get\_size() == 3);

}

TEST\_METHOD(step\_two\_empty)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

auto \*cur = web.edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->set\_resud\_bw(cur->data->get\_prim\_bw());

cur->data->set\_prim\_bw(0);

cur = cur->next;

}

web.set\_adjacent(&web.adjacents, 'T');

Assert::IsTrue(web.adjacents.get\_size() == 0);

}

TEST\_METHOD(step\_three)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

auto \*cur = web.edges.get\_head();

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'S')

{

cur->data->get\_vertex1()->set\_flow(999999);

cur->data->get\_vertex1()->set\_prev\_vert(nullptr);

}

cur->data->set\_resud\_bw(cur->data->get\_prim\_bw());

cur->data->set\_prim\_bw(0);

cur = cur->next;

}

web.set\_adjacent(&web.adjacents, 'A');

cur = web.edges.get\_head();

auto \*cur1 = web.adjacents.get\_head();

int max = 0;

while (cur1 != nullptr)

{

while (cur != nullptr)

{

if (cur->data->get\_vertex1()->get\_title() == 'A'

&& web.check\_vertex(cur->data->get\_vertex2()->get\_title(), &web.adjacents)

&& cur->data->get\_resud\_bw() >= max)

max = cur->data->get\_resud\_bw();

cur = cur->next;

}

cur1 = cur1->next;

}

Assert::IsTrue(max = 8);

}

TEST\_METHOD(step\_five)

{

Web web;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 5);

web.push\_edge('A', 'T', 10);

web.vertexes.at(0)->set\_flow(999999999);

web.vertexes.at(0)->set\_prev\_vert(nullptr);

web.vertexes.at(1)->set\_flow(5);

web.vertexes.at(1)->set\_prev\_vert(web.vertexes.at(0));

web.vertexes.at(2)->set\_flow(10);

web.vertexes.at(2)->set\_prev\_vert(web.vertexes.at(1));

auto \*cur = web.vertexes.get\_head();

size\_t min = cur->data->get\_flow();

while (cur != nullptr)

{

if (min > cur->data->get\_flow())

min = cur->data->get\_flow();

cur = cur->next;

}

Assert::IsTrue(min = 5);

}

TEST\_METHOD(Ford\_Fulk)

{

Web web;

CHECK\_T = 1;

CHECK\_S = 1;

web.push\_vertex('S', &web.vertexes);

web.push\_vertex('A', &web.vertexes);

web.push\_vertex('C', &web.vertexes);

web.push\_vertex('D', &web.vertexes);

web.push\_vertex('B', &web.vertexes);

web.push\_vertex('T', &web.vertexes);

web.push\_edge('S', 'A', 10);

web.push\_edge('S', 'C', 10);

web.push\_edge('A', 'C', 2);

web.push\_edge('A', 'D', 8);

web.push\_edge('A', 'B', 4);

web.push\_edge('C', 'D', 9);

web.push\_edge('D', 'B', 6);

web.push\_edge('D', 'T', 10);

web.push\_edge('B', 'T', 10);

web.Ford\_Fulk();

Assert::IsTrue(web.max\_flow == 19);

}

TEST\_METHOD(Ford\_Fulk\_exception)

{

Web web;

CHECK\_T = 0;

CHECK\_S = 0;

try {

web.Ford\_Fulk();

}

catch (const std::exception& error) {

Assert::AreEqual("S or T don't exist", error.what());

}

}

};

}

**заключение**

В ходе работы над данным проектом была изучена тема сетей и максимального потока в них, закреплены практические знания работы с классами.

**список использованных источников**

1.<https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0_%E2%80%94_%D0%A4%D0%B0%D0%BB%D0%BA%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0><http://aliev.me/runestone/BasicDS/InfixPrefixandPostfixExpressions.html#tbl-example1>

2.<https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D0%BC%D0%B0%D0%BA%D1%81%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%BC_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B5>

3.<https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%BE%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C>