

**МИНОБРНАУКИ РОССИИ  
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)  
Кафедра САПР**

**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  $V_1, V_2, P$ , где  $V_1, V_2$  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 *кратчайший 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.

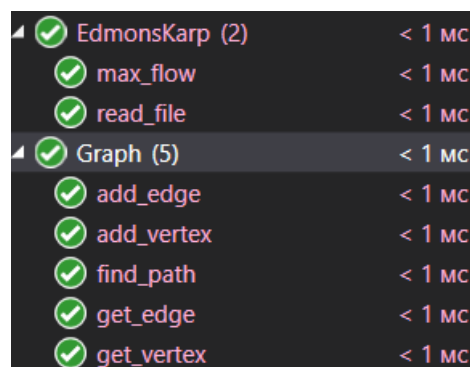
Название метода	Описание
void read_file(string path);	Создание графа по данным из файла.
int max_flow();	Поиск максимального потока.
void print_graphs();	Вывод в консоль начального и модифицированного графов.

## 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.



✓ EdmonsKarp (2)	< 1 мс
✓ max_flow	< 1 мс
✓ read_file	< 1 мс
✓ Graph (5)	< 1 мс
✓ add_edge	< 1 мс
✓ add_vertex	< 1 мс
✓ find_path	< 1 мс
✓ get_edge	< 1 мс
✓ get_vertex	< 1 мс

Рисунок 1. Список тестов и результаты тестирования.

## Пример работы программы

```
Максимальный поток:5
Начальный граф:
S O 3
S P 3
O Q 3
O P 2
P R 2
Q R 4
Q T 2
R T 3

Модифицированный граф:
S O 3
S P 2
O Q 3
O P 0
P R 2
Q R 1
Q T 2
R T 3
```

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 << "ERROR: 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");
    }

```