

**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»
Тема: «Потоки в сетях»**

Студент гр. 9302
Преподаватель

_____ Кузнецов В.А.
_____ Тутуева А.В.

Санкт-Петербург
2021

1. Исходная формулировка задания:

Входные данные: текстовый файлы со строками в формате V1, V1, 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: Проталкивания предпотока.

2. Описание основных методов и оценка сложности

Метод	Описание	Оценка временной сложности
void inputVertex(fstream& in)	Ввод вершин из файла в список	$O(n)$
void inputEdge(fstream& in)	Ввод ребер из файла в список	$O(n)$
bool push(int uNumb)	Проталкивание избыточного потока	$O(n^2)$
void relabel(int uNumb)	Проталкивание избыточного потока вверх	$O(n)$
void preflow(int indexStart)	Задать высоту и поток вершинам	$O(n)$
void updateReverseEdgeFlow(int indexArr, int flow)	Переназначение ребер	$O(n)$
int overFlowVertex(list<vertex>& temp)	Получение номера вершины с избыточным потоком	$O(n)$
int MaxFlow()	Нахождение максимального потока в сети	$O(V^2 * E)$

3. Описание реализованных unit-тестов

Имя теста	Описание
Test1	Проверка макс. потока на исходных данных.
Test2	Проверка макс. потока на иных данных.
Test3	Проверка макс. потока на иных данных.

4. Пример работы

```
3 0 S-0 O-1
3 0 S-0 P-2
3 0 O-1 Q-3
2 0 O-1 P-2
2 0 P-2 R-4
4 0 Q-3 R-4
2 0 Q-3 T-5
3 0 R-4 T-5
Максимальный поток в сети 5
```

Рис. 1. Пример работы программы.

```
5 0 3
5 P 3
O Q 3
O P 2
P R 2
Q R 4
Q T 2
R T 3
```

Рис. 2. Входные данные в in.txt.

Test (3)	1 мс
Test (3)	1 мс
Test (3)	1 мс
Test1	1 мс
Test2	< 1 мс
Test3	< 1 мс

Рис. 3. Результат работы тестов.

5. Код программы

list.h

```
#pragma once
#include <iostream>
#include "graph.h"
#ifndef LIST_H
#define LIST_H
template<class T>
class elem {
private:
    T inf;
    elem* next;
public:
    elem(T elem_inf) { inf = elem_inf; next = nullptr; }
    ~elem() {};
    elem* getNext() { return next; }
    void setNext(elem* elem_next) { next = elem_next; }
    T getInf() { return inf; }
    T* getLink() { return &inf; }
    void setInf(T elem_inf) { inf = elem_inf; }
};

template<class T>
class list {
private:
```

```

    elem<T>* begin, * end;
    size_t size;
public:
    list() { begin = nullptr; end = nullptr; size = 0; }
    ~list() { clear(); }

    //Adding new element to end of list
    void push_back(T temp) {
        elem<T>* newElem = new elem<T>(temp);
        if (isEmpty()) begin = newElem;
        else end->setNext(newElem);
        end = newElem;
        size++;
    }

    //Adding new element to begin of list
    void push_front(T temp) {
        elem<T>* newElem = new elem<T>(temp);
        if (isEmpty()) end = newElem;
        else newElem->setNext(begin);
        begin = newElem;
        size++;
    }

    //Deleting last element from list
    void pop_back() {
        if (!isEmpty()) {
            if (size == 1) {
                end = nullptr;
                begin = nullptr;
                size = 0;
            }
            else {
                elem<T>* newEnd = begin;
                while (newEnd->getNext() != end) newEnd = newEnd->getNext();
                newEnd->setNext(nullptr);
                elem<T>* deleted = end;
                delete deleted;
                end = newEnd;
                size--;
            }
        }
        else throw "List is Empty";
    }

    //Deleting first element from list
    void pop_front() {
        if (!isEmpty()) {
            if (size == 1) {
                begin = nullptr;
                end = nullptr;
                size = 0;
            }
            else {
                elem<T>* newBeg = begin->getNext();
                elem<T>* deleted = begin;
                delete deleted;
                begin = newBeg;
                size--;
            }
        }
        else throw "List is Empty";
    }
}

```

```

//Adding element to any position in list
void insert(T data, size_t pos) {
    if (pos <= size) {
        if (isEmpty() || pos == size) push_back(data);
        else {
            if (pos == 0) push_front(data);
            else {
                elem<T>* newElem = new elem<T>(data);
                elem<T>* iter = begin;
                while (pos-- > 1)
                    iter = iter->getNext();
                newElem->setNext(iter->getNext());
                iter->setNext(newElem);
                size++;
            }
        }
    }
    else throw "Wrong index";
}

//Getting element from list by index
T* at(size_t pos) {
    if (pos < size) {
        elem<T>* iter = begin;
        while (pos-- != 0) iter = iter->getNext();
        return iter->getLink();
    }
    else throw "Wrong index";
}

//Deleting element from list by index
void remove(size_t pos) {
    if (pos < size) {
        if (pos == 0) pop_front();
        else {
            if (pos == size - 1) pop_back();
            else {
                elem<T>* iter = begin;
                while (pos-- > 1) iter = iter->getNext();
                elem<T>* nextElem = iter->getNext();
                iter->setNext(nextElem->getNext());
                size--;
            }
        }
    }
    else throw "Wrong index";
}

//Getting size of list
size_t getSize() { return size; }

//Output elements from list to console
void print_to_console() {
    elem<T>* iter = begin;
    for (size_t i = 0; i < size; i++) {
        std::cout << iter->getInf() << " ";
        iter = iter->getNext();
    }
}

//Deleting elements of list
void clear() {
    while (size) pop_back();
}

```

```

//Replacing element by index with new one
void set(size_t pos, T data) {
    if (pos < size) {
        elem<T>* iter = begin;
        while (pos-- != 0) iter = iter->getNext();
        iter->setInf(data);
    }
    else throw "Wrong index";
}

//Checking list for filling
bool isEmpty() {
    if (size == 0) return true; // 1 - Empty
    else return false;         // 0 - Filled
}

//Adding another list to front of this one
void push_front(list* lst) {
    for (size_t i = 0; i < lst->getSize(); i++)
        insert(lst->at(i), i);
}

//return last element in list
T back() {
    return end->getInf();
}

//find element in list
bool contain(T toFind) {
    elem<T>* temp = begin;
    while (temp != nullptr) {
        if (temp->getInf() == toFind) return true;
        temp = temp->getNext();
    }
    return false;
}

int indexOf(T toFind) {
    elem<T>* temp = begin;
    int i = 0;
    while (temp != nullptr) {
        if (temp->getInf() == toFind) return i;
        temp = temp->getNext();
        i++;
    }
    return -1;
}
};
#endif

```

TermWork.cpp

```

#include <fstream>
#include <iostream>
#include "graph.h"

using namespace std;

int main() {
    graph graph;
    setlocale(LC_ALL, "RUS");
    try {

```

```

        fstream in("in.txt");
        graph.inputVertex(in);
        in.open("in.txt");
        graph.inputEdge(in);
    }
    catch (exception warning) {
        cout << warning.what() << endl;
    }
    cout << endl;
    try {
        cout << "Максимальный поток в сети " << graph.MaxFlow();
    }
    catch (exception warning) {
        cout << warning.what() << endl;
    }
    return 0;
}

```

Test.cpp

```

#include "pch.h"
#include "CppUnitTest.h"
#include "..\\TermWork\\graph.h"

using namespace Microsoft::VisualStudio::CppUnitTestFixture;

namespace Test
{
    TEST_CLASS(Test)
    {
    public:
        graph test;
        TEST_METHOD(Test1)
        {
            fstream in("D:\\TermWork\\TermWork\\in.txt");
            test.inputVertex(in);
            in.open("D:\\TermWork\\TermWork\\in.txt");
            test.inputEdge(in);
            Assert::AreEqual(test.MaxFlow(), 5);
        }

        TEST_METHOD(Test2)
        {
            fstream in("D:\\TermWork\\TermWork\\test1.txt");
            test.inputVertex(in);
            in.open("D:\\TermWork\\TermWork\\test1.txt");
            test.inputEdge(in);
            Assert::AreEqual(test.MaxFlow(), 7);
        }

        TEST_METHOD(Test3)
        {
            fstream in("D:\\TermWork\\TermWork\\test2.txt");
            test.inputVertex(in);
            in.open("D:\\TermWork\\TermWork\\test2.txt");
            test.inputEdge(in);
            Assert::AreEqual(test.MaxFlow(), 9);
        }
    };
}

```

graph.h

```

#pragma once
#include "list.h"
#include <string>
#include <fstream>
using namespace std;

class vertex {
public:
    char name = ' ';
    int height = 0;
    int edgeFlow = 0;
};

class edge {
public:
    char uName = ' ';
    int uNumb = 0;
    char vName = ' ';
    int vNumb = 0;
    int flow = 0;
    int capacity = -1;
};

class graph {
private:
    list<edge> edges;
    list<vertex> vertexes;
public:
    graph() {

    }

    void inputVertex(fstream& in)
    {
        in >> noskipws;
        if (!in.is_open()) throw "Файл не открыт!";
        while (!in.eof()) {
            char c = ' ';
            vertex curVertex;
            in >> c;
            bool newVertex = true;
            for (int i = 0; i < vertexes.getSize(); i++) {
                if (vertexes.at(i)->name == c) {
                    newVertex = false;
                    break;
                }
            }
            if (newVertex) {
                curVertex.name = c;
                vertexes.push_back(curVertex);
            }
            in >> c;
            if (c != ' ')
                throw "Ошибка во входных данных!";
            in >> c;
            newVertex = true;
            for (int i = 0; i < vertexes.getSize(); i++)
                if (vertexes.at(i)->name == c) {
                    newVertex = false;
                    break;
                }
            if (newVertex) {
                curVertex.name = c;

```



```

        vertexes.push_back(curVertex);
    }
    while (c != '\n' && !in.eof()) in >> c;
}
in.close();
}

void inputEdge(fstream& in)
{
    if (vertexes.getSize() == 0) throw "Вершины не заданы";
    in >> noskipws;
    if (!in.is_open()) throw "Файл не открыт!";
    char c;
    while (!in.eof()) {
        edge curEdge;
        in.get(curEdge.uName);
        in >> c;
        if (c != ' ') throw "Ошибка во входных данных!";
        in >> curEdge.vName;
        in >> c;
        if (c != ' ') throw "Ошибка во входных данных!";
        in >> curEdge.capacity;
        if (curEdge.capacity == -1) throw "Ошибка во входных данных!";
        for (int i = 0; i < vertexes.getSize(); i++) {
            if (vertexes.at(i)->name == curEdge.uName) curEdge.uNumb = i;
            if (vertexes.at(i)->name == curEdge.vName) curEdge.vNumb = i;
        }
        edges.push_back(curEdge);
        in << c;
    }
    cout << endl;
    for (int i = 0; i < edges.getSize(); i++) {
        cout << edges.at(i)->capacity << " ";
        cout << edges.at(i)->flow << " ";
        cout << edges.at(i)->uName << "-";
        cout << edges.at(i)->uNumb << " ";
        cout << edges.at(i)->vName << "-";
        cout << edges.at(i)->vNumb << endl;
    }
}

bool push(int uNumb)
{
    for (int i = 0; i < edges.getSize(); i++) {
        if (edges.at(i)->uNumb == uNumb) {
            if (edges.at(i)->flow == edges.at(i)->capacity) continue;
            if (vertexes.at(uNumb)->height > vertexes.at(edges.at(i)-
>vNumb)->height) {
                int flow = min(edges.at(i)->capacity - edges.at(i)-
>flow,
                    vertexes.at(uNumb)->edgeFlow);
                vertexes.at(uNumb)->edgeFlow -= flow;
                vertexes.at(edges.at(i)->vNumb)->edgeFlow += flow;
                edges.at(i)->flow += flow;
                updateReverseEdgeFlow(i, flow);
                return true;
            }
        }
    }
    return false;
}

void relabel(int uNumb)
{

```

```

        int maxHeight = INT_MAX;
        for (int i = 0; i < edges.getSize(); i++) {
            if (edges.at(i)->uNumb == uNumb) {
                if (edges.at(i)->flow == edges.at(i)->capacity) continue;
                if (vertexes.at(edges.at(i)->vNumb)->height < maxHeight) {
                    maxHeight = vertexes.at(edges.at(i)->vNumb)->height;
                    vertexes.at(uNumb)->height = maxHeight + 1;
                }
            }
        }
    }

    void preflow(int indexStart)
    {
        vertexes.at(indexStart)->height = vertexes.getSize();
        for (int i = 0; i < edges.getSize(); i++) {
            if (edges.at(i)->uNumb == indexStart) {
                edges.at(i)->flow = edges.at(i)->capacity;
                vertexes.at(edges.at(i)->vNumb)->edgeFlow += edges.at(i)-
>flow;

                edge newEdge;
                newEdge.flow = -edges.at(i)->flow;
                newEdge.capacity = 0;
                newEdge.uNumb = edges.at(i)->vNumb;
                newEdge.uName = vertexes.at(edges.at(i)->vNumb)->name;
                newEdge.vNumb = indexStart;
                newEdge.vName = vertexes.at(indexStart)->name;
                edges.push_back(newEdge);
            }
        }
    }

    void updateReverseEdgeFlow(int indexArr, int flow)
    {
        int uNumb = edges.at(indexArr)->vNumb;
        int vNumb = edges.at(indexArr)->uNumb;
        for (int j = 0; j < edges.getSize(); j++) {
            if (edges.at(j)->vNumb == vNumb && edges.at(j)->uNumb == uNumb) {
                edges.at(j)->flow -= flow;
                return;
            }
        }
        edge newEdge;
        newEdge.flow = 0;
        newEdge.capacity = flow;
        newEdge.uNumb = uNumb;
        newEdge.uName = vertexes.at(uNumb)->name;
        newEdge.vNumb = vNumb;
        newEdge.vName = vertexes.at(vNumb)->name;
        edges.push_back(newEdge);
    }

    int overFlowVertex(list<vertex>& temp)
    {
        for (int i = 1; i < temp.getSize() - 1; i++)
            if (temp.at(i)->edgeFlow > 0)
                return i;

        return -1;
    }

    int MaxFlow() {
        if (vertexes.getSize() == 0) throw "Вершины не заданы";
        if (edges.getSize() == 0) throw "Рёбра не заданы";
    }

```

```

    int indexStart = 0;
    preflow(indexStart);
    while (overflowVertex(vertexes) != -1) {
        int overflowIndex = overflowVertex(vertexes);
        if (!push(overflowIndex))
            relabel(overflowIndex);
    }
    return vertexes.back().edgeFlow;
}

};

```