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 3

Студент гр. 8301	____________________	Попурей Н
Преподаватель	____________________	Тутуева А.В.

Санкт-Петербург
2020

Оглавление

1. Постановка задачи

Реализовать программу принимающую список ребер из файла, представляющий собой граф. Далее следует рассчитать максимальный поток в заданном графе методом Проталкивания предпотока.

2. Оценка временной сложности

void push(функция, проталкивающая поток из u в v) – O(1)

void lift(функция, поднимающая вершину на минимальную высоту) – O(|V|)

void discharge(функция, выполняющая лифтинг и проталкивание) – O(|V| |E|)

int max_flow(функция, вычисляющая максимальный поток в сети) – O(|V|2 |E|).

3. Описание реализованных юнит-тестов

Реализованные мною тесты проверяют ситуации с 6 вершинами и с 20. Они проверяют ситуацию с одним ребром из стока в сток, а также когда есть не только ребро из истока в сток, но и другие рёбра. Так же тесты проверяют корректность обработки исключительных ситуаций, например, когда пользователь не ввел одну из позиций, либо ввёл её некорректно, а также забыл ввести исток или сток.

4. Обоснование выбора используемых структур данных

Я использую MAP для того чтобы индивидуализировать вершины индексами. Данную структуру я использую по причине того, что она позволяет не сохранять повторяющиеся данные и быстрый доступ к ним. List я использую для перебора вершин сети в функции max_flow. В структуре List есть удобный функционал в отличии от обычного массива, нам не нужно хранить размер массива, также мы можем быстро добавлять и удалять элементы, без траты времени на их перезапись в новый массив (в нашем случае push_front добавление в начало работает за O(1)).

5. Пример работы

Пример обработки ошибки (Рис. 1, Рис. 2)

Input — Блокнот

Файл Правка Формат Вид Сг

```
S A 3 1
S C 2 2
S B 2 3
S D 1 4
A B 7 5
B F -5 6
C B 1
C E 6
D C 2
D E 2
E F 4
E I 1
F I 6
F G 3
G A 4
G H 7
H T 9
I T 7
I H 2
```

Рис. 1

Консоль отладки Microsoft Visual Studio

```
Error entering the third character (bandwidth) in the string or the pre
sence of a space after it.Please note that the bandwidth cannot be nega
tive. Check that you entered the file correctly and correct these error
s in the line number: 6
C:\Users\NikRER\Desktop\Учебный материал\АиСД(Лабы)\Курсач\Course_work\
Debug\Course_AISD.exe (процесс 25728) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу…
```

Рис. 2

Пример нормальной работы программы (Рис. 3, Рис. 4) (Рис. 5, Рис. 6)

Input — Блокнот

Файл Правка Формат Вид Справка

```
S A 3
S C 2
S B 2
S D 1
A B 7
B F 5
C B 1
C E 6
D C 2
D E 2
E F 4
E I 1
F I 6
F G 3
G A 4
G H 7
H T 9
I T 7
I H 2
```

Рис. 3

Консоль отладки Microsoft Visual Studio

```
Max flow for the input graph is: 8
C:\Users\NikRER\Desktop\Учебный материал\АиСД(Лабы)\Курсач\Course_work\Debug\Course_AISD.e
xe (процесс 27872) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу…
```

Рис. 4

Input — Блокнот

Файл Правка Формат Вид Справка

```
S A 5
S C 8
S B 2
A C 1
A D 4
B C 1
B F 6
C F 4
C E 2
C D 5
D E 4
E I 9
E H 5
F G 4
F H 9
G H 7
H T 6
H I 2
I T 4
```

Рис. 5

Консоль отладки Microsoft Visual Studio

```
Max flow for the input graph is: 10
C:\Users\NikRER\Desktop\Учебный материал\АиСД(Лабы)\Курсач\Course_work\Debug\Course_AISD.
exe (процесс 31176) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу…
```

Рис. 6

6. Листинг

Course_AISD.cpp

```
#include "pch.h"
#include <iostream>
#include <fstream>
#include "Flow.h"
int main()
{
	try {
		ifstream input("input.txt");
		Flow flow(input);
		std::cout << "Max flow for the input graph is: " << flow.max_flow();
	}
	catch (exception& ex) {
		std::cout << ex.what();
	}
	return 0;
    }
```

List.h

```
#pragma once
#include<iostream>
using namespace std;
template<class T>
class List
{
private:
	class Node {
	public:
		Node(T data = T(), Node* Next = NULL) {
			this->data = data;
			this->Next = Next;
		}
		Node* Next;
		T data;
	};
public:
	void push_back(T obj) { // add to the end of the list bc
		if (head != NULL) {
			this->tail->Next = new Node(obj);
			tail = tail->Next;
		}
		else {
			this->head = new Node(obj);
			this->tail = this->head;
		}
		Size++;
	}
	void push_front(T obj) { // add to top of list bc
		if (head != NULL) {
			Node* current = new Node;
			current->data = obj;
			current->Next = this->head;
			this->head = current;
		}
		else {
			this->head = new Node(obj);
			tail = head;
		}
		this->Size++;
	}
```

```
	void pop_back() { // delete last item bc
		if (head != NULL) {
			Node* current = head;
			while (current->Next != tail)//looking for the penultimate
				current = current->Next;
			delete tail;
			tail = current;
			tail->Next = NULL;
			Size--;
		}
		else throw std::out_of_range("out_of_range");
	}
	void pop_front() { // delete the first item bc-+
		if (head != NULL) {
			Node* current = head;
			head = head->Next;
			delete current;
			Size--;
		}
		else throw std::out_of_range("out_of_range");
	}
	void insert(T obj, size_t k) {// adding an element by index (insertion before an
element that was previously available at this index) bc
		if (k >= 0 && this->Size > k) {
			if (this->head != NULL) {
				if (k == 0)
					this->push_front(obj);
				else
					if (k == this->Size - 1)
						this->push_back(obj);
					else
					{
						Node* current = new Node;//to add an item
						Node* current1 = head;//to search for the total
item

						for (int i = 0; i < k - 1; i++) {
							current1 = current1->Next;
						}
						current->data = obj;
						current->Next = current1->Next;//retells on the
trail element

						current1->Next = current;
						Size++;
					}
			}
		}
		else {
			throw std::out_of_range("out_of_range");
		}
	}
	T at(size_t k) {// getting an item by index bc
		if (this->head != NULL && k >= 0 && k <= this->Size - 1) {
			if (k == 0)
				return this->head->data;
			else
				if (k == this->Size - 1)
					return this->tail->data;
				else
				{
					Node* current = head;
					for (int i = 0; i < k; i++) {
						current = current->Next;
					}
					return current->data;
				}
```

```
		}
		else {
			throw std::out_of_range("out_of_range");
		}
	}
	void remove(int k) { // delete item by index bc
		if (head != NULL && k >= 0&&k<=Size-1) {
			if (k == 0) this->pop_front();
			else
				if (k == this->Size - 1) this->pop_back();
				else
					if (k != 0) {
						Node* current = head;
						for (int i = 0; i < k - 1; i++) {//go to pre-
element

							current = current->Next;
						}

						Node* current1 = current->Next;
						current->Next = current->Next->Next;
						delete current1;
						Size--;
					}
		}
		else {
			throw std::out_of_range("out_of_range");
		}
	}
	size_t get_size() { // getting list size bc
		return Size;
	}
	void print_to_console() { // output of list items to the console through a
separator, do not use at bc
		if (this->head != NULL) {
			Node* current = head;
			for (int i = 0; i < Size; i++) {
				cout << current->data << ' ';
				current = current->Next;
			}
		}
	}
	void clear() { // delete all list items
		if (head != NULL) {
			Node* current = head;
			while (head != NULL) {
				current = current->Next;
				delete head;
				head = current;
			}
			Size = 0;
		}
	}
	void set(size_t k, T obj)  // replacement of an element by index with a
transmitted element
	{
		if (this->head != NULL && this->get_size() >= k && k >= 0) {
			Node* current = head;
			for (int i = 0; i < k; i++) {
				current = current->Next;
			}
			current->data = obj;
		}
		else {
			throw std::out_of_range("out_of_range");
		}
```

```
	}
	bool isEmpty() { // checking for empty bc list
		return (bool)(head);
	}
	void reverse() { // reorders items in a list
		int Counter = Size;
		Node* HeadCur = NULL;
		Node* TailCur = NULL;
		for (int j = 0; j <Size; j++) {
			if (HeadCur != NULL) {
				if(head!=NULL&&head->Next==NULL){
					TailCur->Next = head;
					TailCur = head;
					head = NULL;
				}
				else {
						Node * cur = head;
					for (int i = 0; i < Counter - 2; i++)
						cur = cur->Next;
					TailCur->Next = cur->Next;
					TailCur = cur->Next;
					cur->Next = NULL;
					tail = cur;
					Counter--;
				}
			}
			else {
				HeadCur = tail;
				TailCur = tail;
				Node* cur = head;
				for (int i = 0; i < Size - 2; i++)
					cur = cur->Next;
				tail = cur;
				tail->Next = NULL;
				Counter--;
			}
		}
		head = HeadCur;
		tail = TailCur;
	}
public:
	List(Node* head = NULL, Node* tail = NULL, int Size = 0) :head(head), tail(tail),
Size(Size) {}
	~List() {
		if (head != NULL) {
			this->clear();
		}
	};
private:
	Node* head;
	Node* tail;
	int Size;
};
```

Map.h

```
#pragma once
#define COLOR_RED 1
#define COLOR_BLACK 0
#include"List.h"
using namespace std;
template<typename T, typename T1>
class Map {
public:
	class Node
	{
	public:
		Node(bool color = COLOR_RED, T key = T(), Node* parent = NULL, Node* left = NULL, Node* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}
		T key;
		T1 value;
		bool color;
		Node* parent;
		Node* left;
		Node* right;
	};

	~Map()
	{
		if (this->Root != NULL)
			this->clear();
		Root = NULL;
		delete TNULL;
		TNULL = NULL;
	}

	Map(Node* Root = NULL, Node* TNULL = new Node(0)) :Root(TNULL), TNULL(TNULL) {}

	void printTree()
	{
		if (Root)
		{
			print_helper(this->Root, "", true);
		}
		else throw std::out_of_range("Tree is empty!");
	}

	void  insert(T key, T1 value)
	{
		if (this->Root != TNULL)
		{
			Node* node = NULL;
			Node* parent = NULL;
			/* Search leaf for new element */
			for (node = this->Root; node != TNULL; )
			{
				parent = node;
				if (key < node->key)
					node = node->left;
				else if (key > node->key)
					node = node->right;
				else if (key == node->key)
					throw std::out_of_range("key is repeated");
			}
```

```
			node = new Node(COLOR_RED, key, TNULL, TNULL, TNULL, value);
			node->parent = parent;


			if (parent != TNULL)
			{
				if (key < parent->key)
					parent->left = node;
				else
					parent->right = node;
			}
			insert_fix(node);
		}
		else
		{
			this->Root = new Node(COLOR_BLACK, key, TNULL, TNULL, TNULL, value);
		}
	}

	List<T>* get_keys() {
		List<T>* list = new List<T>();
		this->ListKey(Root, list);
		return list;
	}
	List<T1>* get_values() {
		List<T1>* list = new List<T1>();
		this->ListValue(Root, list);
		return list;
	}

	T1 find(T key)
	{
		Node* node = Root;
		while (node != TNULL && node->key != key)
		{
			if (node->key > key)
				node = node->left;
			else
				if (node->key < key)
					node = node->right;
		}
		if (node != TNULL)
			return node->value;
		else
			throw std::out_of_range("Key is missing");
	}

	void remove(T key)
	{
		this->delete_node(this->find_key(key));
	}

	void clear()
	{
		this->clear_tree(this->Root);
		this->Root = NULL;
	}

	bool find_is(T key) {
		Node* node = Root;

		while (node != TNULL && node->key != key) {
			if (node->key > key)
				node = node->left;
			else
```

```
                    if (node->key < key)
                        node = node->right;
            }
            if (node != TNULL)
                return true;
            else
                return false;
        }
        void increment_value(T key) {
            Node* cur = this->find_value(key);
            cur->value++;
        }
    private:
        Node* Root;
        Node* TNULL;

        //delete functions

        void delete_node(Node* find_node)
        {
            Node* node_with_fix, * cur_for_change;
            cur_for_change = find_node;
            bool cur_for_change_original_color = cur_for_change->color;
            if (find_node->left == TNULL)
            {
                node_with_fix = find_node->right;
                transplant(find_node, find_node->right);
            }
            else if (find_node->right == TNULL)
            {
                node_with_fix = find_node->left;
                transplant(find_node, find_node->left);
            }
            else
            {
                cur_for_change = minimum(find_node->right);
                cur_for_change_original_color = cur_for_change->color;
                node_with_fix = cur_for_change->right;
                if (cur_for_change->parent == find_node)
                {
                    node_with_fix->parent = cur_for_change;
                }
                else
                {
                    transplant(cur_for_change, cur_for_change->right);
                    cur_for_change->right = find_node->right;
                    cur_for_change->right->parent = cur_for_change;
                }
                transplant(find_node, cur_for_change);
                cur_for_change->left = find_node->left;
                cur_for_change->left->parent = cur_for_change;
                cur_for_change->color = find_node->color;
            }
            delete find_node;
            if (cur_for_change_original_color == COLOR_RED)
            {
                this->delete_fix(node_with_fix);
            }
        }

        //swap links(parent and other) for rotate
        void transplant(Node* current, Node* current1)
        {
            if (current->parent == TNULL)
            {
```

```
                Root = current1;
            }
            else if (current == current->parent->left)
            {
                current->parent->left = current1;
            }
            else
            {
                current->parent->right = current1;
            }
            current1->parent = current->parent;
        }

        void clear_tree(Node* tree)
        {
            if (tree != TNULL)
            {
                clear_tree(tree->left);
                clear_tree(tree->right);
                delete tree;
            }
        }
    //find functions

        Node* minimum(Node* node)
        {
            while (node->left != TNULL)
            {
                node = node->left;
            }
            return node;
        }

        Node* maximum(Node* node)
        {
            while (node->right != TNULL)
            {
                node = node->right;
            }
            return node;
        }

        Node* grandparent(Node* current)
        {
            if ((current != TNULL) && (current->parent != TNULL))
                return current->parent->parent;
            else
                return TNULL;
        }

        Node* uncle(Node* current)
        {
            Node* current1 = grandparent(current);
            if (current1 == TNULL)
                return TNULL; // No grandparent means no uncle
            if (current->parent == current1->left)
                return current1->right;
            else
                return current1->left;
        }

        Node* sibling(Node* n)
        {
            if (n == n->parent->left)
                return n->parent->right;
```

```
        else
            return n->parent->left;
}

Node* find_key(T key)
{
        Node* node = this->Root;
        while (node != TNULL && node->key != key)
        {
            if (node->key > key)
                node = node->left;
            else
                if (node->key < key)
                    node = node->right;
        }
        if (node != TNULL)
            return node;
        else
            throw std::out_of_range("Key is missing");
}

//all print function

void print_helper(Node* root, string indent, bool last)
{
        if (root != TNULL)
        {
            cout << indent;
            if (last)
            {
                cout << "R----";
                indent += "   ";
            }
            else
            {
                cout << "L----";
                indent += "|  ";
            }
            string sColor = !root->color ? "black" : "red";
            cout << root->key << " (" << sColor << ")" << endl;
            print_helper(root->left, indent, false);
            print_helper(root->right, indent, true);
        }
}

void list_key_or_value(int mode, List<T>* list)
{
        if (this->Root != TNULL)
            this->key_or_value(Root, list, mode);
        else
            throw std::out_of_range("Tree empty!");
}

void key_or_value(Node* tree, List<T>* list, int mode)
{
        if (tree != TNULL)
        {
            key_or_value(tree->left, list, mode);
            if (mode == 1)
                list->push_back(tree->key);
            else
                list->push_back(tree->value);
            key_or_value(tree->right, list, mode);
        }
}
```

```
//fix

void insert_fix(Node* node)
{
	Node* uncle;
	/* Current node is COLOR_RED */
	while (node != this->Root && node->parent->color == COLOR_RED)//
	{
		/* node in left tree of grandfather */
		if (node->parent == this->grandparent(node)->left)//
		{
			/* node in left tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == COLOR_RED)
			{
				/* Case 1 - uncle is COLOR_RED */
				node->parent->color = COLOR_BLACK;
				uncle->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				node = this->grandparent(node);
			}
			else {
				/* Cases 2 & 3 - uncle is COLOR_BLACK */
				if (node == node->parent->right)
				{
					/*Reduce case 2 to case 3 */
					node = node->parent;
					this->left_rotate(node);
				}
				/* Case 3 */
				node->parent->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				this->right_rotate(this->grandparent(node));
			}
		}
		else {
			/* Node in right tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == COLOR_RED)
			{
				/* Uncle is COLOR_RED */
				node->parent->color = COLOR_BLACK;
				uncle->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				node = this->grandparent(node);
			}
			else {
				/* Uncle is COLOR_BLACK */
				if (node == node->parent->left)
				{
					node = node->parent;
					this->right_rotate(node);
				}
				node->parent->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				this->left_rotate(this->grandparent(node));
			}
		}
	}
	this->Root->color = COLOR_BLACK;
}

void delete_fix(Node* node)
{
```

```
Node* sibling;
while (node != this->Root && node->color == COLOR_BLACK)//
{
    sibling = this->sibling(node);
    if (sibling != TNULL)
    {
        if (node == node->parent->left)//
        {
            if (sibling->color == COLOR_BLACK)
            {
                node->parent->color = COLOR_BLACK;
                sibling->color = COLOR_RED;
                this->left_rotate(node->parent);
                sibling = this->sibling(node);
            }
            if (sibling->left->color == COLOR_RED && sibling-
>right->color == COLOR_RED)
            {
                sibling->color = COLOR_BLACK;
                node = node->parent;
            }
            else
            {
                if (sibling->right->color == COLOR_RED)
                {
                    sibling->left->color = COLOR_RED;
                    sibling->color = COLOR_BLACK;
                    this->left_rotate(sibling);
                    sibling = this->sibling(node);
                }
                sibling->color = node->parent->color;
                node->parent->color = COLOR_RED;
                sibling->right->color = COLOR_RED;
                this->left_rotate(node->parent);
                node = this->Root;
            }
        }
        else
        {
            if (sibling->color == COLOR_BLACK);
            {
                sibling->color = COLOR_RED;
                node->parent->color = COLOR_BLACK;
                this->right_rotate(node->parent);
                sibling = this->sibling(node);
            }
            if (sibling->left->color == COLOR_RED && sibling-
>right->color)

            {
                sibling->color = COLOR_BLACK;
                node = node->parent;
            }
            else
            {
                if (sibling->left->color == COLOR_RED)
                {
                    sibling->right->color = COLOR_RED;
                    sibling->color = COLOR_BLACK;
                    this->left_rotate(sibling);
                    sibling = this->sibling(node);
                }
                sibling->color = node->parent->color;
                node->parent->color = COLOR_RED;
                sibling->left->color = COLOR_RED;
                this->right_rotate(node->parent);
```

```
                                        node = Root;
                                }
                        }
                }

            }
            this->Root->color = COLOR_BLACK;
        }
    //Rotates

        void left_rotate(Node* node)
        {
            Node* right = node->right;
            /* Create node->right link */
            node->right = right->left;
            if (right->left != TNULL)
                right->left->parent = node;
            /* Create right->parent link */
            if (right != TNULL)
                right->parent = node->parent;
            if (node->parent != TNULL)
            {
                if (node == node->parent->left)
                    node->parent->left = right;
                else
                    node->parent->right = right;
            }
            else {
                this->Root = right;
            }
            right->left = node;
            if (node != TNULL)
                node->parent = right;
        }

        void right_rotate(Node* node)
        {
            Node* left = node->left;
            /* Create node->left link */
            node->left = left->right;
            if (left->right != TNULL)
                left->right->parent = node;
            /* Create left->parent link */
            if (left != TNULL)
                left->parent = node->parent;
            if (node->parent != TNULL)
            {
                if (node == node->parent->right)
                    node->parent->right = left;
                else
                    node->parent->left = left;
            }
            else
            {
                this->Root = left;
            }
            left->right = node;
            if (node != TNULL)
                node->parent = left;
        }
        void ListValue(Node* tree, List<T1>* list) {
            if (tree != TNULL) {
                ListValue(tree->left, list);
                list->push_back(tree->value);
                ListValue(tree->right, list);
```

```
            }
        }
        void ListKey(Node* tree, List<T>* list) {
            if (tree != TNULL) {
                ListKey(tree->left, list);
                list->push_back(tree->key);
                ListKey(tree->right, list);
            }
        }

        Node* find_value(T key) {
            Node* node = Root;

            while (node != TNULL && node->key != key) {
                if (node->key > key)
                    node = node->left;
                else
                    if (node->key < key)
                        node = node->right;
            }
            if (node != TNULL)
                return node;

        }
    };
```

Flow.h

```
#pragma once
#include <fstream>
#include "List.h"
#include<string>
#include"Map.h"
#include "Algorithm.h"
using namespace std;
class Flow {
public:
        ~Flow() {
            delete[] excess_flow;
            delete[] height;
            for(int i=0;i<vertexCount;++i)
            delete[] capacity_edge[i];
        }
        Flow(ifstream& file)
        {
            Map<char, int>* Map_from_char_to_number = new Map<char, int>();
            vertexCount = 0;
            int str_num = 1;
            while (!file.eof()) {
                string s1;
                getline(file, s1);
                if (s1.size() >= 5) {//greater than or equal to 5, because this is
the minimum possible input(two letters, two spaces,one digit)
                    if (!((s1[0] >= 'A' && s1[0] <= 'Z') && (s1[1] == ' '))) {
                        throw std::exception(string(("Error entering the first
character in the string or missing a space after it. Check the correctness of the input
in the file and correct these errors in the line under the number: " +
to_string(str_num))).c_str());
                    }
                    if (!((s1[2] >= 'A' && s1[2] <= 'Z') && (s1[3] == ' '))) {
                        throw std::exception(string(("Error entering the second
character in the string or missing a space after it. Check the correctness of the input
in the file and correct these errors in the line under the number: " +
to_string(str_num))).c_str());
                    }
```

```
                        string cur;
                        for (int i = 4; i < s1.size(); ++i) {
                              if (s1[i] >= '0' && s1[i] <= '9')
                                    cur += s1[i];
                              else {
                                    throw std::exception(string(("Error entering the third character (bandwidth) in the string or the presence of a space after it.Please note that the bandwidth cannot be negative. Check that you entered the file correctly and correct these errors in the line number: " + to_string(str_num))).c_str());
                              }
                        }
                        if (!Map_from_char_to_number->find_is(s1[0])) {//checking the presence of a symbol in the Map, if it is not present, we write it to the Map and assign it an individual index
                              Map_from_char_to_number->insert(s1[0], vertexCount);
                              ++vertexCount;
                        }
                        if (!Map_from_char_to_number->find_is(s1[2])) {
                              Map_from_char_to_number->insert(s1[2], vertexCount);
                              ++vertexCount;
                        }

                  }
                  else
                  {
                        throw std::exception(string(("A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: " + to_string(str_num))).c_str());
                  }
                  ++str_num;
            }
            if (Map_from_char_to_number->find_is('S'))
                  sourceVertex = Map_from_char_to_number->find('S');
            else {
                  throw std::exception("Source is missing");
            }

            if (Map_from_char_to_number->find_is('T'))
                  destinationVertex = Map_from_char_to_number->find('T');
            else {
                  throw std::exception("Sink is missing");
            }
            file.clear();
            file.seekg(ios::beg);
            excess_flow = new int[vertexCount];
            height = new int[vertexCount];
            capacity_edge = new int* [vertexCount];
            for (int i = 0; i < vertexCount; ++i) {
                  excess_flow[i] = 0;
                  height[i] = 0;
            }
            for (int i = 0; i < vertexCount; ++i) {
                  capacity_edge[i] = new int[vertexCount];
                  for (int j = 0; j < vertexCount; ++j)
                        capacity_edge[i][j] = 0;
            }
            str_num = 1;
            while (!file.eof()) {
                  string s1;
                  int vert1, vert2, cap;
                  getline(file, s1);
                  vert1 = Map_from_char_to_number->find(s1[0]);
                  vert2 = Map_from_char_to_number->find(s1[2]);
                  if(vert1==vert2)
```

```
				throw std::exception(string("The path from the vertex to itself is impossible in the string under the number: "+to_string(str_num)).c_str());
				capacity_edge[vert1][vert2] = stoi(s1.substr(4));
				++str_num;
			}
		}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
		int max_flow() {
			if (vertexCount > 2) {
				for (int i = 0; i < vertexCount; i++)
				{
					if (i == sourceVertex)
						continue;
					excess_flow[i] = capacity_edge[sourceVertex][i];
					capacity_edge[i][sourceVertex] += capacity_edge[sourceVertex][i];
				}
				height[sourceVertex] = vertexCount;
				List<int> l;
				int cur;
				int cur_index = 0;
				int old_height;
				for (int i = 0; i < vertexCount; i++)
					if (i != sourceVertex && i != destinationVertex)
						l.push_front(i);
				cur = l.at(0);
				while (cur_index < l.get_size())
				{
					old_height = height[cur];
					discharge(cur);
					if (height[cur] != old_height)
					{
						l.push_front(cur);
						l.remove(++cur_index);
						cur = l.at(0);
						cur_index = 0;
					}
					++cur_index;
					if (cur_index < l.get_size())
						cur = l.at(cur_index);

				}
				return excess_flow[destinationVertex];
			}
			else
				return capacity_edge[0][1];
		}
		void push(int edge, int vertex)
		{
			int f = min(excess_flow[edge], capacity_edge[edge][vertex]);
			excess_flow[edge] -= f;
			excess_flow[vertex] += f;
			capacity_edge[edge][vertex] -= f;
			capacity_edge[vertex][edge] += f;
		}

		void lift(int edge)
		{
			int min = 2 * vertexCount + 1;

			for (int i = 0; i < vertexCount; i++)
				if (capacity_edge[edge][i] && (height[i] < min))
					min = height[i];
			height[edge] = min + 1;
```

```
	}

	void discharge(int edge)
	{
		int vertex = 0;
		while (excess_flow[edge] > 0)
		{
			if (capacity_edge[edge][vertex] && height[edge] == height[vertex] +
1)
			{
				push(edge, vertex);
				vertex = 0;
				continue;
			}
			++vertex;
			if (vertex == vertexCount)
			{
				lift(edge);
				vertex = 0;
			}
		}
	}
private:
	int* excess_flow;
	int** capacity_edge;
	int* height;
	int vertexCount, sourceVertex, destinationVertex;
    };
```

Algoritm.h

```
#pragma once
template<typename T>
T min(T a, T b) {
	return a > b ? b : a;
    }
```

UnitTest1.cpp

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "../Course_work/Flow.h"
#include <fstream>
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestFlowPushRelabel
{
	TEST_CLASS(UnitTestFlowPushRelabel)
	{
	public:

		TEST_METHOD(TestMethod_Correct_output_for_6_vertexes)
		{
			ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный
материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input1.txt");
			Flow flow(input);
			Assert::AreEqual(flow.max_flow(), 5);
		}
		TEST_METHOD(TestMethod_Exception_entering_the_first_character) {
			try {
				ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный
материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input2.txt");
				Flow flow(input);
			}
			catch (exception & ex) {
```

```
				Assert::AreEqual(ex.what(), "Error entering the first character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: 2");
			}
		}
		TEST_METHOD(TestMethod_Exception_entering_the_second_character) {
			try {
				ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input3.txt");
				Flow flow(input);
			}
			catch (exception & ex) {
				Assert::AreEqual(ex.what(), "Error entering the second character in the string or missing a space after it. Check the correctness of the input in the file and correct these errors in the line under the number: 2");
			}
		}
		TEST_METHOD(TestMethod_Exception_entering_the_third_number_flow) {
			try {
				ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input4.txt");
				Flow flow(input);
			}
			catch (exception & ex) {
				Assert::AreEqual(ex.what(), "Error entering the third character (bandwidth) in the string or the presence of a space after it.Please note that the bandwidth cannot be negative. Check that you entered the file correctly and correct these errors in the line number: 2");
			}
		}
		TEST_METHOD(TestMethod_Exception_empty_string) {
			try {
				ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input5.txt");
				Flow flow(input);
			}
			catch (exception & ex) {
				Assert::AreEqual(ex.what(), "A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: 2");
			}
		}

	TEST_METHOD(TestMethod_Correct_output_for_6_vertexes_and_edge_from_source_to_sink)
		{
			ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input6.txt");
			Flow flow(input);
			Assert::AreEqual(flow.max_flow(), 25);
		}

	TEST_METHOD(TestMethod_Correct_output_for_2_vertexes_edges_from_source_to_sink)
		{
			ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input7.txt");
			Flow flow(input);
			Assert::AreEqual(flow.max_flow(), 20);
		}
		TEST_METHOD(TestMethod_Exception_there_is_a_path_from_the_vertex_to_itself)
{
			try {
				ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input8.txt");
				Flow flow(input);
			}
```

```
                    catch (exception & ex) {
                        Assert::AreEqual(ex.what(), "The path from the vertex to 
itself is impossible in the string under the number: 2");
                    }
                }
                TEST_METHOD(TestMethod_Correct_output_for_20_vertexes)
                {
                    ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный 
материал\\АиСД(Лабы)\\Курсач\\Course_work\\UnitTest1\\Input9.txt");
                    Flow flow(input);
                    Assert::AreEqual(flow.max_flow(), 19);
                }

        };
    }
```

7. Вывод

В данной лабораторной работе я ознакомился с алгоритмом Проталкивания предпотока и смог применить его в нахождении максимального потока в транспортной сети, а также закрепил свои навыки в объектно-ориентированном программировании.