**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 3](#_Toc41057483)

[2. Оценка временной сложности 3](#_Toc41057484)

[3. Описание реализованных юнит-тестов 3](#_Toc41057485)

[4. Обоснование выбора используемых структур данных 3](#_Toc41057486)

[5. Пример работы 4](#_Toc41057487)

[6. Листинг 7](#_Toc41057488)

[Course\_AISD.cpp 7](#_Toc41057489)

[List.h 7](#_Toc41057490)

[Map.h 11](#_Toc41057491)

[Flow.h 19](#_Toc41057492)

[Algoritm.h 22](#_Toc41057493)

[UnitTest1.cpp 22](#_Toc41057494)

[7. Вывод 24](#_Toc41057495)

# **Постановка задачи**

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

# 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|).

# Описание реализованных юнит-тестов

Реализованные мною тесты проверяют ситуации c 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.

# Обоснование выбора используемых структур данных

Я использую МАР 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)).

# Пример работы

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

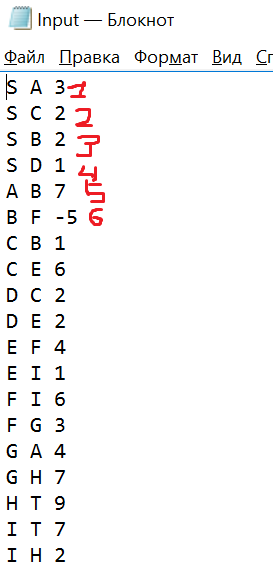


Рис. 1

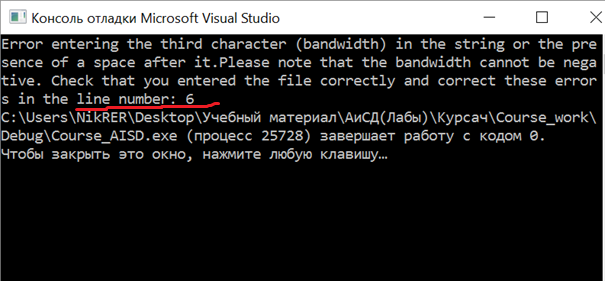


Рис. 2

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

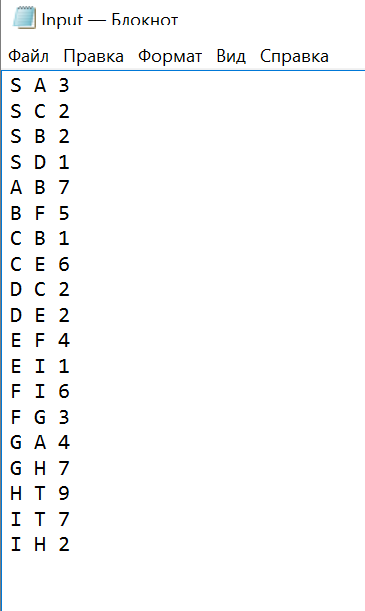


Рис. 3

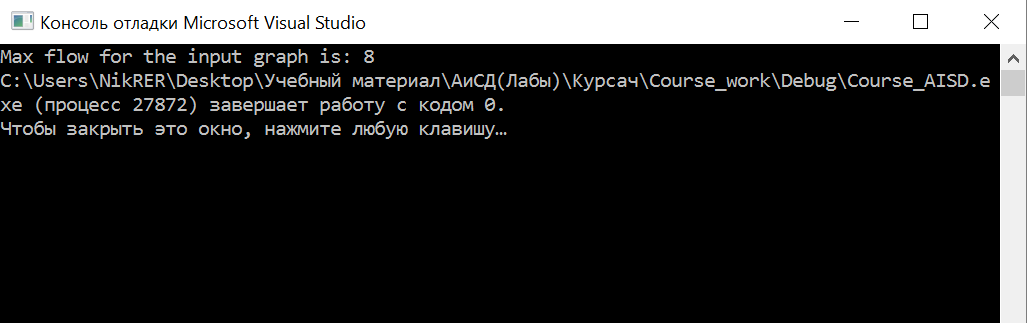


Рис. 4

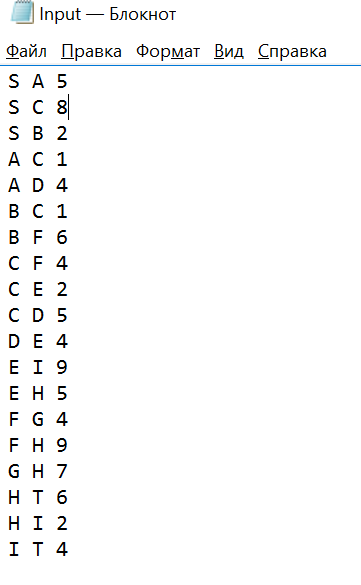


Рис. 5

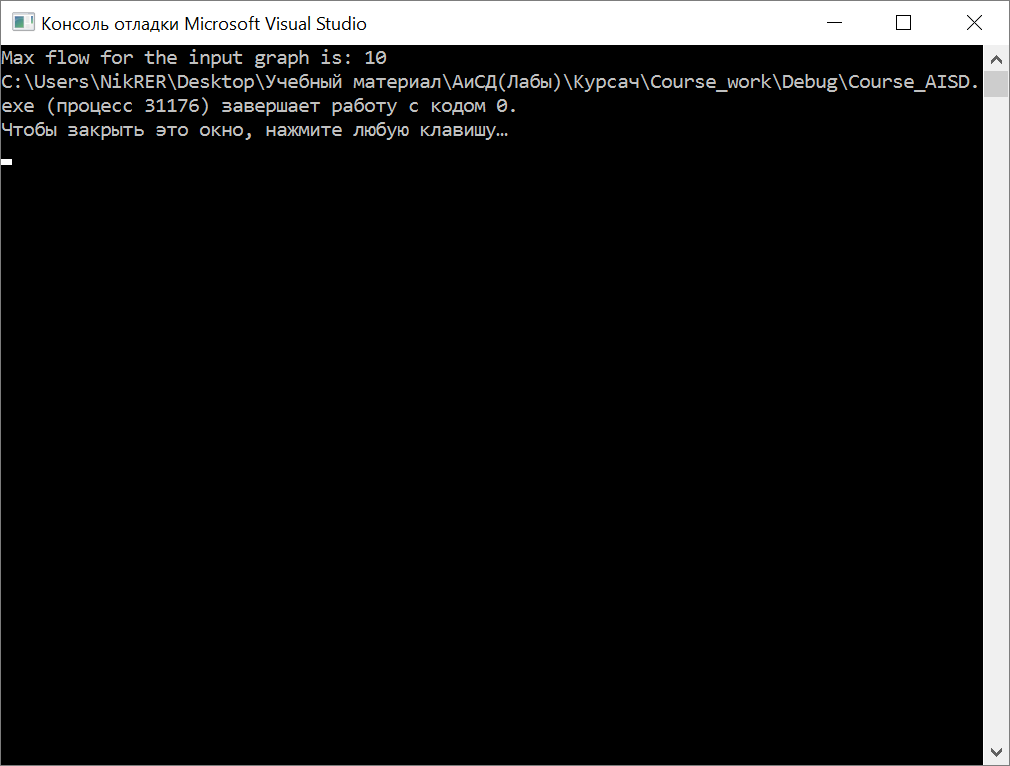


Рис. 6

# 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 "../Сourse\_work/Flow.h"

#include <fstream>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestFlowPushRelabel

{

TEST\_CLASS(UnitTestFlowPushRelabel)

{

public:

TEST\_METHOD(TestMethod\_Сorrect\_output\_for\_6\_vertexes)

{

ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Сourse\_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\\Сourse\_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\\Сourse\_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\\Сourse\_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\\Сourse\_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\_Сorrect\_output\_for\_6\_vertexes\_and\_edge\_from\_source\_to\_sink)

{

ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Сourse\_work\\UnitTest1\\Input6.txt");

Flow flow(input);

Assert::AreEqual(flow.max\_flow(), 25);

}

TEST\_METHOD(TestMethod\_Сorrect\_output\_for\_2\_vertexes\_edges\_from\_source\_to\_sink)

{

ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Сourse\_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\\Сourse\_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\_Сorrect\_output\_for\_20\_vertexes)

{

ifstream input("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Курсач\\Сourse\_work\\UnitTest1\\Input9.txt");

Flow flow(input);

Assert::AreEqual(flow.max\_flow(), 19);

}

};

}

# 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.