**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»**

**Вариант 2**

Студент гр. 8309 Ильин Д.Д.

Преподаватель Тутуева А.В.

Санкт-Петербург

2020

# Цель работы

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

# Описание реализуемых и вспомогательных классов

Класс Network, содержит поля: int VertexCount (количество верщин), int SourceVertex

(исток), int SinkVertex (сток), int\*\* Capacity (пропускная способность), int\*\* FlowPassed (поток), int\* ParList (вершина, связанная с текущей), int\* CurrentPathC (поток, приходящий в текущую вершину), List<int>\* Graph (граф с пропускными способностями рёбер).

Класс содержит следующие методы:

* Конструктор – получает на вход файл, в котором список строк, обрабатывает их и в результате выдает список Map’ов, где каждый Map это одна вершина.
* Деструктор – вызывает метод clear (на основе обычного удаления двоичного дерева)
* BFS(int Source, int Sink) – функция поиска кратчайшего пути в графе, где остаточная пропускная способность ещё не равна нулю. Возвращает максимальный поток кратчайшего пути.
* max\_flow() – функция возвращает максимальный поток транспортной сети.
* EdmondsKarp(int Source, int Sink) – функция вычисляет максимальный поток методом Эдмондса-Карпа.

# Оценка временной сложности алгоритмов

* BFS(int Source, int Sink) – O(E)
* max\_flow() – O(V\*E2)
* EdmondsKarp(int Source, int Sink) – O(V\*E2)

**Описание реализованных unit-тестов**

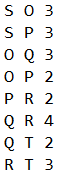
Реализованные мною тесты проверяют ситуации c 6 вершинами и с 20. Они проверяют ситуацию с одним ребром из стока в сток, а также когда есть не только ребро из истока в сток, но и другие рёбра. Тесты проверяют ситуации, когда отсутствует исток или сток, а также, когда введена отрицательная пропускная способность. Также они проверяют корректность обработки исключительных ситуаций, например, когда пользователь не ввел одну из позиций, либо ввёл её некорректно.

# Обоснование выбора используемых структур данных

Мap я использую для того, чтобы индивидуализировать вершины индексами. Я использую данную структуру, т.к. она позволяет не сохранять повторяющиеся данные и даёт быстрый доступ к своим элементам. List я использую для хранения графа и пропускных способностей. В структуре List удобный функционал в отличие от обычного массива, нам не нужно хранить размер массива, а также мы можем быстро добавлять и удалять элементы, без затраты времени на их перезапись в новый массив. Структуру Queue я использую для перебора путей потоков. Эта структура удобна тем, что не требует данных о количестве элементов, а также об их индексах.

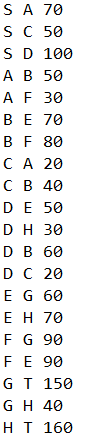
# Примеры работы программы

**1)**

****

****

**2)**





**Код**

**kursach.cpp**

#include "fstream"

#include "iostream"

#include "Network.h"

using namespace std;

int main()

{

try {

int source, sink;

ifstream input("C:\\users\\danil\\source\\repos\\kursach\\kursach\\input.txt");

Network\* transnet = new Network(input);

int MaxFlow = transnet->max\_flow();

cout << endl << "Max Flow is " << MaxFlow << endl;

delete transnet;

}

catch (exception& ex) {

std::cout << ex.what();

}

}

**Network.h**

#pragma once

#include "fstream"

#include "Map.h"

#include "Queue.h"

#include "string"

#include "Algorithm.h"

class Network {

public:

~Network() {

for (int i = 0; i < VertexCount; ++i) {

Graph[i].clear();

delete[]Capacity[i];

delete[]FlowPassed[i];

}

delete[]ParList;

delete[]CurrentPathC;

}

Network(ifstream& file) {

Map<char, int>\* Map\_from\_char\_to\_number = new Map<char, int>();

Validators(file, Map\_from\_char\_to\_number);

file.clear();

file.seekg(ios::beg);

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]);

Capacity[vert1][vert2] = stoi(s1.substr(4));

Graph[vert1].push\_back(vert2);

Graph[vert2].push\_back(vert1);

}

}

void Validators(ifstream& file, Map<char, int>\*& Map\_from\_char\_to\_number) {

int vert1, vert2, cap;

VertexCount = 0;

Map\_from\_char\_to\_number = new Map<char, int>();

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])) {

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'))

SinkVertex = Map\_from\_char\_to\_number->find('T');

else {

throw std::exception("Sink is missing");

}

Graph = new List<int>[VertexCount];

FlowPassed = new int\* [VertexCount];

Capacity = new int\* [VertexCount];

ParList = new int[VertexCount];

CurrentPathC = new int[VertexCount];

for (int i = 0; i < VertexCount; ++i) {

FlowPassed[i] = new int[VertexCount];

Capacity[i] = new int[VertexCount];

for (int j = 0; j < VertexCount; ++j) {

FlowPassed[i][j] = 0;

Capacity[i][j] = 0;

}

}

}

int BFS(int Source, int Sink) //breadth first search

{

for (int i = 0; i < VertexCount; ++i) {

ParList[i] = -1;

CurrentPathC[i] = 0;

}

Queue<int> q; //declare queue vector

q.enqueue(Source);

ParList[Source] = -1;

CurrentPathC[Source] = 999;

while (!q.empty()) //if q is not empty

{

int CurrNode = q.dequeue();

for (int i = 0; i < Graph[CurrNode].get\_size(); i++)

{

int to = Graph[CurrNode].at(i);

if (ParList[to] == -1)

{

if (Capacity[CurrNode][to] - FlowPassed[CurrNode][to] > 0)

{

ParList[to] = CurrNode;

CurrentPathC[to] = min(CurrentPathC[CurrNode], Capacity[CurrNode][to] - FlowPassed[CurrNode][to]);

if (to == Sink)

{

return CurrentPathC[Sink];

}

q.enqueue(to);

}

}

}

}

return 0;

}

int max\_flow() {

int max = EdmondsKarp(SourceVertex, SinkVertex);

return max;

}

int EdmondsKarp(int Source, int Sink)

{

int MaxFlow = 0;

while (true)

{

int Flow = BFS(Source, Sink);

if (Flow == 0)

{

break;

}

MaxFlow += Flow;

int CurrNode = Sink;

while (CurrNode != Source)

{

int PrevNode = ParList[CurrNode];

FlowPassed[PrevNode][CurrNode] += Flow;

FlowPassed[CurrNode][PrevNode] -= Flow;

CurrNode = PrevNode;

}

}

return MaxFlow;

}

private:

int SourceVertex;

int SinkVertex;

List<int>\* Graph;

int\*\* Capacity;

int\*\* FlowPassed;

int\* ParList;

int\* CurrentPathC;

int VertexCount;

};

**Queue.h**

#pragma once

template<typename T>

class Queue

{

public:

class Node

{

public:

Node(T data = T(), Node\* next = NULL) :data(data), next(next) {}

T data;

Node\* next;

};

void enqueue(T current) { // adding to the queue

if (head != NULL) {

this->tail->next = new Node(current);

this->tail = this->tail->next;

}

else {

this->head = new Node(current);

this->tail = head;

}

this->Counter++;

}

T dequeue() {// Remove from queue. Returns the deleted item

if (head != NULL) {

Node\* current = head;

T cur = head->data;

head = head->next;

delete current;

this->Counter--;

return cur;

}

else throw std::out\_of\_range("out\_of\_range");

}

T peek() { //returns an item from the beginning of the queue

if (head != NULL) {

return head->data;

}

else throw std::out\_of\_range("out\_of\_range");

}

int get\_size() { // returns the number of items in the queue

return Counter;

}

bool empty() {

if (head != NULL)

return false;

else

return true;

}

Queue(Node\* head = NULL, Node\* tail = NULL, int Counter = 0) :head(head), tail(tail), Counter(Counter) {}

~Queue() {

while (head != NULL) {

Node\* cur = head->next;

delete head;

head = cur;

}

}

private:

Node\* head;

Node\* tail;

int Counter;

};

**Algorithm.h**

#pragma once

template<typename T>

T min(T a, T b) {

return a > b ? b : a;

}

**Вывод**

В данной лабораторной работе я познакомился с алгоритмом поиска максимального потока в транспортной сети методом Эдмондса-Карпа и смог применить его в нахождении максимального потока в данной сети, а также закрепил свои навыки в ООП на языке C++.