**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: Алгоритм проталкивания предпотока.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8309 |  | Носов А.С. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |  |
| --- | --- | --- | --- |
| **Студенту Носову А. С.** | | | |
| **Группа 8309** | | | |
| **Тема работы: Алгоритмы на графах** | | | |
| **Исходные данные:**  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. | | | | | |
| **Содержание пояснительной записки:**  «Исходная формулировка», «Цель работы», «Постановка задачи», «Обоснование выбора используемых структур данных», «Описание алгоритма решения», «Организация данных», «Пример работы», «Код программы». | | | |
| Дата выдачи задания: 15.03.2020 | | | |
| Студент |  | Носов А.С. | |
| Преподаватель |  | Тутуева А.В. | |

**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.

**Цель работы:**

Создать программу, производящую поиск максимального потока в сети.

**Постановка задачи:**

Необходимо реализовать создание динамически расширяемого орграфа из файла, и найти в нем максимальный поток сети.

**Обоснование выбора используемых структур данных:**

Так как нам необходимо иметь быстрый доступ к информации мы будем использовать собственный контейнер “Vect” основанный на динамическом массиве с реализацией вызова элементов через операторные скобки. Также создадим две структуры Edge и Vertex, первая будет хранить информацию о потоке и проходимости и название вершин, между которыми она расположена. Вторая структура будет хранить информацию о вершине, такую как проводимость, название и высоту. Сохранены все предметы в Vect как уже было сказано для быстрого доступа элементам нашего графа.

**Описание алгоритма решения:**

На каждом шаге будем рассматривать некоторый предпоток - т.е. функцию, которая по свойствам напоминает поток, но не обязательно удовлетворяет закону сохранения потока. На каждом шаге будем пытаться применить какую-либо из двух операций: проталкивание потока или поднятие вершины. Если на каком-то шаге станет невозможно применить какую-либо из двух операций, то мы нашли требуемый поток.

**Организация данных:**

|  |  |  |
| --- | --- | --- |
| **Название** | **Описание работы метода** | **Оценка временной сложности** |
| int getMaxFlow(); | Возвращает максимальный поток в сети. | O(V^(2)\*E) |
| void addVertexs(string filelink) | Функция добавляет вершины в систему из текстового файла, путем прохода всего файла и поиска в системе уже существующих вершин. | O(N\*(K+K)) |
| void addEdge(string flink) | Функция добавляет ребра в систему из текстового файла и позицию вершины в OURvec. | O(N) |
| bool push(int u); | Используется для создания потока из узла, который имеет избыточный поток. Если у вершины есть избыточный поток, и есть соседний с меньшей высотой (в остаточном графе), мы продвигаем поток из вершины в соседний с меньшей высотой. Количество проталкиваемого потока через ребро равно минимуму избыточного потока и вместимости ребра. | O(N\*2) |
| void relabel(int u); | Используется, когда вершина имеет избыточный поток, и ни одна из ее смежных не находится на более низкой высоте. Мы в основном увеличиваем высоту вершины, чтобы мы могли выполнить push (). Чтобы увеличить высоту, мы выбираем минимальную смежную высоту и добавляем к ней 1. | O(N) |
| void preflow(int s); | Инициализирует высоты и потоки всех вершин | O(N) |
| void updateReverseEdgeFlow(int i, int flow); | Функция для изменения ребра в графе. | O(N) |
| int overFlowVertex(OURvector<Vertex>& ver) | Функция для возврата индекса переполненной вершины. | O(N) |

|  |  |
| --- | --- |
| **Название Unit-теста** | **Описание работы** |
| FileNotOpenVertexTest | Проверяем ошибку на некорректность файла при создании вершин. |
| EdgesTestErrorVer | Проверяем ошибку, при которой хотим создать ребра не объявив вершины. |
| FileNotOpenEdgesTest | Проверяем ошибку на некорректность файла при создании ребер. |
| VertexNoCreateTestMax | Проверка вызова максимального потока без созданных вершин. |
| EdgesNoCreateTestMax | Проверка вызова максимального потока без созданных ребер. |
| TrueWorkTest | Правильная работа поиска максимального потока в сети (ожидается 5). |

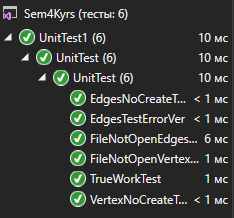


Рисунок 1 – результат проверки тестов

**Пример работы:**

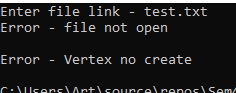


Рисунок 2 – на вход принят некорректный файл

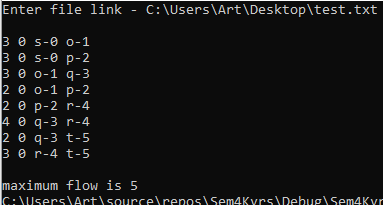


Рисунок 3 – пример правильной работы программы

**Код программы**

**main.cpp**

#include "Push.h"

using namespace std;

int main()

{

Graph mg;

string flink;

cout << "Enter file link - ";

cin >> flink;

try

{

mg.addVertexs(flink);

mg.addEdge(flink);

}

catch (invalid\_argument error)

{

cout << error.what() << endl;

}

cout << endl;

try

{

cout << "maximum flow is " << mg.getMaxFlow();

}

catch (out\_of\_range error)

{

cout << error.what() << endl;

}

return 0;

}

**Vect.h**

#pragma once

#include<iostream>

#include<stdio.h>

using namespace std;

template<typename T>

class OURvector

{

public:

OURvector()

{

arr = nullptr;

sizeArr = 0;

}

~OURvector()

{

delete[]arr;

}

void push\_back(T data)

{

if (arr == nullptr)

{

sizeArr = sizeArr + 1;

arr = new T[sizeArr];

arr[sizeArr - 1] = data;

}

else

{

sizeArr++;

T\* tmp;

tmp = (T\*)realloc(arr, sizeArr \* sizeof(T));

if (tmp != nullptr)

{

arr = tmp;

arr[sizeArr - 1] = data;

}

}

}

T& operator[](const int index)

{

return arr[index];

}

int size()

{

return sizeArr;

}

T back()

{

if (sizeArr == 0)

{

throw out\_of\_range("vector size equal 0!");

}

return arr[sizeArr - 1];

}

private:

T\* arr;

int sizeArr;

};

**Push.h**

#include<iostream>

#include<fstream>

#include<algorithm>

#include"Vect.h"

class Graph

{

public:

Graph();

~Graph();

// возвращает максимальный поток от s до t

int getMaxFlow();

void addVertexs(string filelink)

{

fstream file(filelink, ios::in);

if (!file.is\_open())

{

throw invalid\_argument("Error - file not open");

}

while (!file.eof())

{

char symb = ' ';

Vertex dt;

file.get(symb);

bool verNew = true;

for (int i = 0; i < ver.size(); i++)

{

if (ver[i].name == symb)

{

verNew = false;

break;

}

}

if (verNew == true)

{

dt.name = symb;

ver.push\_back(dt);

}

file.get(symb);

file.get(symb);

verNew = true;

for (int i = 0; i < ver.size(); i++)

{

if (ver[i].name == symb)

{

verNew = false;

break;

}

}

if (verNew == true)

{

dt.name = symb;

ver.push\_back(dt);

}

while (symb != '\n' && !file.eof())

{

file.get(symb);

}

}

file.close();

}

void addEdge(string flink)

{

if (ver.size() == 0)

{

throw out\_of\_range("Error - Vertex no creates");

}

fstream file(flink, ios::in);

if (!file.is\_open())

{

throw invalid\_argument("Error - file not open");

}

while (!file.eof())

{

Edge dt;

file.get(dt.uName);

file.get();

file.get(dt.vName);

file.get();

file >> dt.capacity;

for (int i = 0; i < ver.size(); i++)

{

if (ver[i].name == dt.uName)

{

dt.uNumb = i;

}

if (ver[i].name == dt.vName)

{

dt.vNumb = i;

}

}

edge.push\_back(dt);

file.get();

}

cout << endl;

for (int i = 0; i < edge.size(); i++)

{

cout << edge[i].capacity << " ";

cout << edge[i].flow << " ";

cout << edge[i].uName << "-";

cout << edge[i].uNumb << " ";

cout << edge[i].vName << "-";

cout << edge[i].vNumb << endl;

}

}

struct Vertex

{

char name = ' ';

int height = 0;

int e\_flow = 0;

};

private:

struct Edge

{

int flow = 0;

int capacity = 0;

char uName = ' ';

int uNumb = 0;

char vName = ' ';

int vNumb = 0;

};

OURvector<Edge> edge;

OURvector<Vertex> ver;

bool push(int u);

// Функция для перемаркировки вершины u

void relabel(int u);

// Эта функция вызывается для инициализации

// предварительный поток

void preflow(int s);

// Функция для изменения края

void updateReverseEdgeFlow(int i, int flow);

// возвращает индекс переполненной вершины

int overFlowVertex(OURvector<Vertex>& ver)

{

for (int i = 1; i < ver.size() - 1; i++)

{

if (ver[i].e\_flow > 0)

return i;

}

// -1 если переполненная вершина отсутствует

return -1;

}

};

Graph::Graph()

{

}

Graph::~Graph()

{

}

void Graph::preflow(int s)

{

// Делаем h исходной вершины равным no. вершин

// Высота остальных вершин равна 0.

ver[s].height = ver.size();

//

for (int i = 0; i < edge.size(); i++)

{

// Если текущее ребро идет от источника

if (edge[i].uNumb == s)

{

// Поток равен емкости

edge[i].flow = edge[i].capacity;

// Инициализируем избыточный поток для соседнего v

ver[edge[i].vNumb].e\_flow += edge[i].flow;

// Добавить ребро из v в s в остаточном графе с

// емкость равна 0

Edge ed;

ed.flow = -edge[i].flow;

ed.capacity = 0;

ed.uNumb = edge[i].vNumb;

ed.uName = ver[edge[i].vNumb].name;

ed.vNumb = s;

ed.vName = ver[s].name;

edge.push\_back(ed);

}

}

}

// Обновление обратного потока для потока, добавленного в ih Edge

void Graph::updateReverseEdgeFlow(int i, int flow)

{

int u = edge[i].vNumb;

int v = edge[i].uNumb;

for (int j = 0; j < edge.size(); j++)

{

if (edge[j].vNumb == v && edge[j].uNumb == u)

{

edge[j].flow -= flow;

return;

}

}

// добавляем обратный край в остаточный график

Edge ed;

ed.flow = 0;

ed.capacity = flow;

ed.uNumb = u;

ed.uName = ver[u].name;

ed.vNumb = v;

ed.vName = ver[v].name;

edge.push\_back(ed);

}

// Чтобы вытолкнуть поток из переполняющейся вершины u

bool Graph::push(int u)

{

// Пройдите через все ребра, чтобы найти соседний (из вас)

// к какому потоку можно подтолкнуть

for (int i = 0; i < edge.size(); i++)

{

// Проверяет u текущего ребра так же, как задано

// переполняющая вершина

if (edge[i].uNumb == u)

{

// если поток равен пропускной способности, то нет толчка

// возможно

if (edge[i].flow == edge[i].capacity)

continue;

// Push возможен только если высота смежных

// меньше высоты переполняющейся вершины

if (ver[u].height > ver[edge[i].vNumb].height)

{

// Поток, который нужно протолкнуть, равен минимуму

// оставшийся поток на краю и избыточный поток.

int flow = min(edge[i].capacity - edge[i].flow,

ver[u].e\_flow);

// Уменьшаем лишний поток для переполняющейся вершины

ver[u].e\_flow -= flow;

// Увеличить избыточный поток для соседних

ver[edge[i].vNumb].e\_flow += flow;

// Добавить остаточный поток (с емкостью 0 и отрицательным

// течь)

edge[i].flow += flow;

updateReverseEdgeFlow(i, flow);

return true;

}

}

}

return false;

}

// функция для перемаркировки вершины u

void Graph::relabel(int u)

{

// Инициализируем минимальную высоту соседней

int mh = INT\_MAX;

// Находим соседний с минимальной высотой

for (int i = 0; i < edge.size(); i++)

{

if (edge[i].uNumb == u)

{

// если поток равен емкости, то нет

// перемаркировка

if (edge[i].flow == edge[i].capacity)

continue;

// Обновляем минимальную высоту

if (ver[edge[i].vNumb].height < mh)

{

mh = ver[edge[i].vNumb].height;

// Обновление высоты вас

ver[u].height = mh + 1;

}

}

}

}

// основная функция для печати максимального потока графика

int Graph::getMaxFlow()

{

if (ver.size() == 0)

{

throw out\_of\_range("Error - Vertex no create");

}

if (edge.size() == 0)

{

throw out\_of\_range("Error - Edges no create");

}

int s = 0;

preflow(s);

// цикл до тех пор, пока ни одна из вершин не будет переполнена

while (overFlowVertex(ver) != -1)

{

int u = overFlowVertex(ver);

if (!push(u))

relabel(u);

}

// ver.back () возвращает последнюю вершину, чья

// e\_flow будет конечным максимальным потоком

return ver.back().e\_flow;

}

**UnitTest1.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "Push.h"

#include "Vect.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest

{

TEST\_CLASS(UnitTest)

{

public:

TEST\_METHOD(FileNotOpenVertexTest)

{

Graph myGr;

try

{

myGr.addVertexs("test.txt");

}

catch (invalid\_argument error)

{

Assert::AreEqual("Error - file not open", error.what());

}

}

TEST\_METHOD(EdgesTestErrorVer)

{

Graph myGr;

try

{

myGr.addEdge("C:\\Users\\Art\\Desktop\\test.txt");

}

catch (out\_of\_range error)

{

Assert::AreEqual("Error - Vertex no creates", error.what());

}

}

TEST\_METHOD(FileNotOpenEdgesTest)

{

Graph myGr;

myGr.addVertexs("C:\\Users\\Art\\Desktop\\test.txt");

try

{

myGr.addEdge("test.txt");

}

catch (invalid\_argument error)

{

Assert::AreEqual("Error - file not open", error.what());

}

}

TEST\_METHOD(VertexNoCreateTestMax)

{

Graph myGr;

try

{

myGr.getMaxFlow();

}

catch (out\_of\_range error)

{

Assert::AreEqual("Error - Vertex no create", error.what());

}

}

TEST\_METHOD(EdgesNoCreateTestMax)

{

Graph myGr;

myGr.addVertexs("C:\\Users\\Art\\Desktop\\test.txt");

try

{

myGr.getMaxFlow();

}

catch (out\_of\_range error)

{

Assert::AreEqual("Error - Edges no create", error.what());

}

}

TEST\_METHOD(TrueWorkTest)

{

Graph myGr;

myGr.addVertexs("C:\\Users\\Art\\Desktop\\test.txt");

myGr.addEdge("C:\\Users\\Art\\Desktop\\test.txt");

Assert::AreEqual(5, myGr.getMaxFlow());

}

};

}