**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 1: Алгоритм Форда - Фалкерсона.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8309 |  | Иванов Д.К. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2020

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |  |
| --- | --- | --- | --- |
| **Студенту Иванову Д.К.** | | | |
| **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:  1 2 33  2 5 32  1 1 35  5 5 22  3 5 28  2 2 44  2 1 23  4 1 37  Найти максимальный поток в сети используя алгоритм проталкивания предпотока. | | | | | |
| **Содержание пояснительной записки:**  «Цель работы», «Описание алгоритма решения», «Организация данных», «Пример работы», «Код программы». | | | |
| Дата выдачи задания: 15.03.2020 | | | |
| Студент |  | Иванов Д.К. | |
| Преподаватель |  | Тутуева А.В | |

**Цель работы:**

Создать программу, производящую поиск максимального потока в сети.

**Описание алгоритма решения:**

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь максимально возможный поток:
4. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью *c*min.
5. Для каждого ребра на найденном пути увеличиваем поток на *c*min, а в противоположном ему - уменьшаем на *c*min.
6. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
7. Возвращаемся на шаг 2.

\*Если пропускные способности вещественны, алгоритм может работать бесконечно долго, не сходясь к оптимальному решению

**Организация данных:**

|  |  |  |
| --- | --- | --- |
| **Название** | **Описание работы метода** | **Оценка временной сложности** |
| NewVertex | Добавление новой вершины | O(N) |
| NewEdge | Добавление нового ребра | O(1) |
| Contains | Проверка считанную строку | O(N) |
| FF | Функция основго алгоритма Форда - Фалкерсона | O(e\*F) |
| Output | Функция вывода | O(N\*M) |
| FileRead | Функция считывания файла | O(N\*(A+B+C)) |

|  |  |
| --- | --- |
| **Название Unit-теста** | **Описание работы** |
| NewVertex\_contains | Проверка на работу contains, проверка на работу NewVertex при правильных условиях, проверка на работу NewVertex при попытке вставки узла с повторяющимся элементом |
| NewEdge\_FF | Проверка на работу алгоритма при условии отсутствия пути, проверка на работу алгоритма при стандартных условиях, проверка на работу алгоритма при условии слишком больших индексов |
| FileDoesen’tOpenTest | Проверка на правильное открытие файла |

**Пример работы:**

Введенные данные:

1 2 50

1 4 10

1 8 20

2 3 90

2 6 50

2 10 50

3 4 100

4 2 20

4 5 20

5 1 50

6 5 100

6 7 40

7 1 70

7 6 40

8 7 50

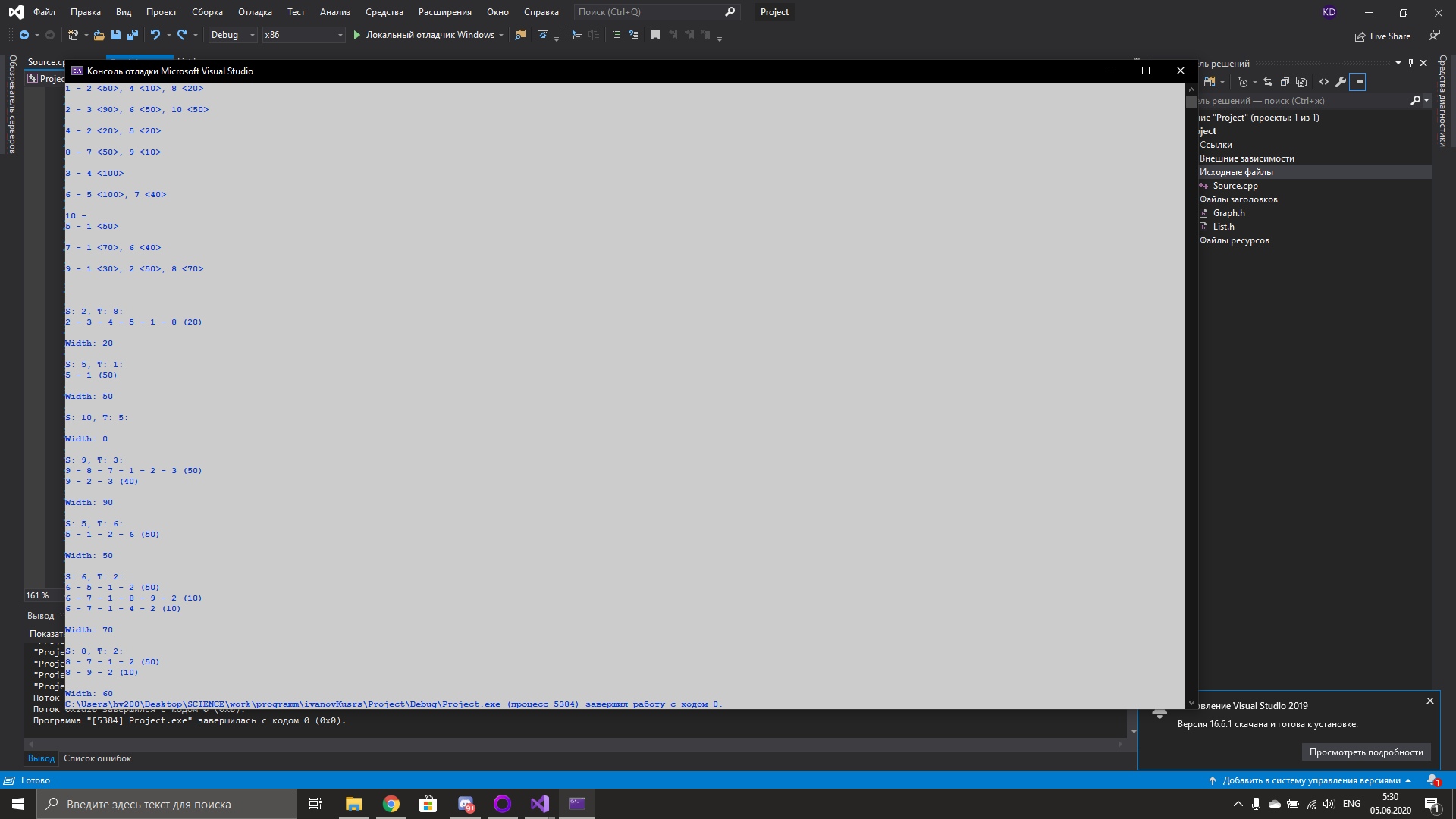
8 9 10

9 1 30

9 2 50

9 8 70

Пример правильного вывода программы:



**Код программы**

**List.h**

#ifndef list\_H

#define list\_H

using namespace std;

#include <iostream>

#include <exception>

template<typename T>

class List

{

public:

List();

~List();

void pop\_front();

void pop\_back();

void push\_back(T data);

void push\_front(T data);

void insert(T value, int index);

void clear();

void removeAt(int index);

unsigned int GetSize()

{

return Size;

}

T& operator[] (const int index);

void output\_console();

void set(T value, int index);

bool isEmpty();

bool contains(T search);

int find\_last(List<T>& search); // Compare 2 lists. Get first element (bigger list, not in ()) index where match starts, or get -1 if there is no match.

private:

template<typename T>

class Node

{

public:

Node\* pNext;

Node\* pPrev;

T data;

Node(T data = T(), Node\* pNext = nullptr, Node\* pPrev = nullptr) // by default, next and previous pointers are nullptr

{

this->data = data;

this->pNext = pNext;

this->pPrev = pPrev;

}

};

int Size;

Node<T>\* head;

Node<T>\* tail;

};

template<typename T>

List<T>::List()// constructor

{

Size = 0;

head = nullptr;

tail = nullptr;

}

template<typename T>

List<T>::~List() // destructor

{

clear();

}

template<typename T>

void List<T>::pop\_front()

{

if (Size == 0)

throw exception("Error, list is empty");

Node<T>\* temp = head;

head = head->pNext;

delete temp;

Size--;

}

template<typename T>

void List<T>::pop\_back()

{

if (Size == 0)

throw exception("Error, list is empty");

removeAt(Size - 1);

}

template<typename T>

void List<T>::push\_back(T data)

{

if (head == nullptr)

{

head = new Node<T>(data);

tail = head;

}

else

{

Node<T>\* temp = new Node<T>(data);

temp->pNext = nullptr;

temp->pPrev = tail;

tail->pNext = temp;

tail = temp;

}

Size++;

}

template<typename T>

void List<T>::push\_front(T data)

{

head = new Node<T>(data, head);

Size++;

if (Size == 1)

tail = head;

}

template<typename T>

void List<T>::insert(T data, int index)

{

if (index == 0)

push\_front(data);

else

{

if (index < 0)

throw exception("Error, index is negative");

if (index > Size)

throw exception("Error, index is bigger than list size + 1");

Node<T>\* previous = this->head;

Node<T>\* nex = this->head;

for (int i = 0; i < index - 1; i++)

{

previous = previous->pNext;

nex = nex->pNext;

}

Node<T>\* newNode = new Node<T>(data, previous->pNext);

previous->pNext = newNode;

nex = nex->pNext;

nex->pPrev = previous;

if (index != Size)

{

previous = previous->pNext;

nex = nex->pNext;

nex->pPrev = previous;

}

else

tail = nex;

Size++;

}

}

template<typename T>

void List<T>::clear()

{

while (Size)

pop\_front();

}

template<typename T>

void List<T>::removeAt(int index)

{

if (index == 0)

pop\_front();

else

{

if (index < 0)

throw exception("Error, index is negative");

if (index >= Size)

throw exception("Error, index is bigger than list size");

Node<T>\* previous = this->head;

for (int i = 0; i < index - 1; i++)

previous = previous->pNext;

Node<T>\* toDelete = previous->pNext;

if (index != Size - 1)

{

Node<T>\* nex = toDelete->pNext;

nex->pPrev = previous;

}

else

tail = previous;

previous->pNext = toDelete->pNext;

delete toDelete;

Size--;

}

}

template<typename T>

T& List<T>::operator[](const int index)

{

int counter = 0;

Node<T>\* current = this->head;

while (current != nullptr)

{

if (counter == index)

return current->data;

current = current->pNext;

counter++;

}

}

template<typename T>

void List<T>::output\_console()

{

Node<T>\* current = this->head;

if (Size == 0)

cout << "Error, ist is empty";

else

while (current != nullptr)

{

cout << current->data << endl;

current = current->pNext;

}

}

template<typename T>

void List<T>::set(T data, int index)

{

if (index < 0)

throw exception("Error, index is negative");

if (index >= Size)

throw exception("Error, index is bigger than list size");

int counter = 0;

Node<T>\* current = this->head;

while (current != nullptr)

{

if (counter == index)

break;

current = current->pNext;

counter++;

}

current->data = data;

}

template<typename T>

bool List<T>::isEmpty()

{

if (Size == 0)

return true;

else

return false;

}

template<typename T>

bool List<T>::contains(T search)

{

Node<T>\* temp = head;

for (unsigned int i = 0; i < GetSize(); ++i)

if (temp->data == search)

return true;

return false;

}

template<typename T>

int List<T>::find\_last(List<T>& search)

{

if (Size == 0)

throw exception("Error, main list contains 0 items");

if (search.GetSize() > Size)

throw exception("Error, included list is bigger than main one");

if (search.GetSize() == 0)

throw exception("Error, included list contains 0 items");

Node<T>\* field = this->tail;

int steps = 0;

bool match = false;

for (int i = Size - 1; i >= 0; i--)

{

if (field->data == search[search.GetSize() - 1])

{

steps = 0;

for (int j = search.GetSize() - 2; j >= 0; j--)

{

steps++;

field = field->pPrev;

if (field->data != search[j])

{

match = false;

for (int k = 0; k < steps; k++)

field = field->pNext;

steps = 0;

break;

}

if (j == 0)

match = true;

}

if ((match == true) || (search.GetSize() == 1))

{

return (i - steps);

}

}

field = field->pPrev;

}

return -1;

}

#endif list\_H

**Graph.h**

#include <string>

#include <limits>

#include <cstdlib>

#include <fstream>

#include "List.h"

struct Graph

{

void fileRead(string fileName)

{

ifstream file;

file.open(fileName, ios::in);

if (!file.is\_open())

{

throw invalid\_argument("Error, file doesn't open");

}

file.unsetf(ios::skipws);

while (!file.eof())

{

string temp;

getline(file, temp, ' ');

string arrival = temp;

if (contains(arrival) == false)

NewVertex(arrival);

temp.clear();

getline(file, temp, ' ');

string T = temp;

if (contains(T) == false)

NewVertex(T);

temp.clear();

getline(file, temp, '\n');

float width = stof(temp);

newEdge(arrival, T, width);

}

}

void NewVertex(string name)

{

for (unsigned int i = 0; i < graph.GetSize(); ++i)

if (graph[i]->name == name)

throw exception("Attempt to add a new vertex with the same name");

Node\* temp = new Node(name);

graph.push\_back(temp);

}

void newEdge(string nameDeparture, string nameDestination, float Width)

{

newEdge(stringToInt(nameDeparture), stringToInt(nameDestination), Width);

}

void output()

{

for (unsigned int i = 0; i < graph.GetSize(); ++i)

{

if (graph[i]->name != "")

cout << graph[i]->name;

else

cout << i;

cout << " - ";

for (unsigned int j = 0; j < graph[i]->edges.GetSize(); ++j)

{

if (graph[i]->edges[j]->T->name != "")

cout << graph[i]->edges[j]->T->name;

else

cout << graph[i]->edges[j]->T;

cout << " <" << graph[i]->edges[j]->Width << ">";

if (j < graph[i]->edges.GetSize() - 1)

cout << ", ";

else

cout << endl;

}

cout << endl;

}

}

bool contains(string name)

{

for (unsigned int i = 0; i < graph.GetSize(); ++i)

if (graph[i]->name == name)

return true;

return false;

}

float FF(string sourceName, string outletName)

{

return FF(stringToInt(sourceName), stringToInt(outletName));

}

float FF(unsigned int sourceIndex, unsigned int outletIndex)

{

struct rout

{

List<Node\*> path;

float flow;

rout(Node& finalNode)

{

this->flow = finalNode.flow;

Node\* current = &finalNode;

while (current != nullptr)

{

path.push\_front(current);

current = current->S;

}

}

};

if (sourceIndex >= graph.GetSize() || outletIndex >= graph.GetSize())

throw exception("Out of range");

Node\* source = graph[sourceIndex];

Node\* outlet = graph[outletIndex];

List<rout\*> pathways;

source->flow = numeric\_limits<float>::infinity();

Node\* current = source;

while (1)

{

Node possible;

for (unsigned int i = 0; i < current->edges.GetSize(); ++i)

{

if (current->edges[i]->remain > 0)

if (current->edges[i]->T->flow == NULL)

{

Node::Edge\* p = current->edges[i];

possible.addEdge(p);

}

}

if (possible.edges.GetSize() == 0)

{

if (current == source)

{

float total = 0;

for (unsigned int i = 0; i < pathways.GetSize(); ++i)

{

total += pathways[i]->flow;

for (unsigned int j = 0; j < pathways[i]->path.GetSize(); ++j)

{

cout << pathways[i]->path[j]->name;

if (j != pathways[i]->path.GetSize() - 1)

cout << " - ";

}

cout << " (" << pathways[i]->flow << ")" << endl;

}

source->flow = 0;

for (unsigned int i = 0; i < graph.GetSize(); ++i)

for (unsigned int j = 0; j < graph[i]->edges.GetSize(); ++j)

graph[i]->edges[j]->remain = graph[i]->edges[j]->Width;

return total;

}

else

{

Node\* temp = current->S;

for (unsigned int i = 0; i < temp->edges.GetSize(); ++i)

if (temp->edges[i]->T == current)

{

temp->edges[i]->remain = 0;

break;

}

current->S = nullptr;

current->flow = NULL;

current = temp;

continue;

}

}

else

{

unsigned int max = 0;

for (unsigned int i = 0; i < possible.edges.GetSize(); ++i)

if (possible.edges[i]->remain > possible.edges[max]->remain)

max = i;

Node\* next = possible.edges[max]->T;

next->S = current;

if (current->flow > possible.edges[max]->remain)

{

next->flow = possible.edges[max]->remain;

possible.edges[max]->remain = 0;

}

else

{

next->flow = current->flow;

possible.edges[max]->remain -= current->flow;

}

if (next == outlet)

{

pathways.push\_back(new rout(\*next));

Node\* p = next;

while (p != source)

{

for (unsigned int i = 0; i < p->S->edges.GetSize(); ++i)

if (p->S->edges[i]->T == p)

{

p->S->edges[i]->remain += (p->flow - next->flow);

if (p != next)

p->flow = NULL;

Node\* temp = p->S;

p->S = nullptr;

p = temp;

break;

}

}

next->flow = NULL;

current = source;

}

else

current = next;

}

}

}

private:

void newEdge(unsigned int departure, unsigned int T, float Width)

{

if (departure > graph.GetSize() - 1 || T > graph.GetSize() - 1)

throw exception("Out of range");

Node::Edge\* temp = new Node::Edge(graph[T], Width);

graph[departure]->addEdge(temp);

}

unsigned int stringToInt(string input)

{

unsigned int answer;

bool success = false;

for (unsigned int i = 0; i < graph.GetSize(); ++i)

{

if (graph[i]->name == input)

{

answer = i;

success = true;

break;

}

}

if (success == false)

throw exception("Couldn't find such vertex");

return answer;

}

struct Node

{

struct Edge

{

Node\* T;

float Width;

float remain;

Edge(Node\* T = nullptr, unsigned int Width = NULL)

{

this->T = T;

this->Width = Width;

this->remain = Width;

}

};

void addEdge(Edge\* newEdge)

{

this->edges.push\_back(newEdge);

};

string name;

List<Edge\*> edges;

Node\* S;

float flow;

Node(string name = "")

{

this->name = name;

this->S = nullptr;

this->flow = 0;

}

};

List<Node\*> graph;

};

**Source.cpp**

#include <iostream>

#include "Graph.h"

using namespace std;

int main()

{

Graph test;

test.fileRead("input.txt");

test.output();

for (unsigned int i = 0; i < 7; ++i)

{

unsigned int source = 1 + rand() % 10;

unsigned int outlet = 1 + rand() % 10;

while (outlet == source)

outlet = 1 + rand() % 10;

cout << "\n\nS: " << source << ", T: " << outlet << ":" << endl;

float weight = test.FF(to\_string(source), to\_string(outlet));

cout << endl << "Width: " << weight;

}

}

**UnitTest1.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "C:\Users\user\Desktop\kurs\_4\_sem\Graph.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

TEST\_METHOD(NewVertex\_contains)

{

Graph test;

test.NewVertex("1");

test.NewVertex("2");

test.NewVertex("3");

test.NewVertex("4");

Assert::AreEqual(false, test.contains("6"));

Assert::AreEqual(true, test.contains("3"));

test.NewVertex("6");

Assert::AreEqual(true, test.contains("6"));

try

{

test.NewVertex("3");

}

catch (const std::exception& ex)

{

Assert::AreEqual(ex.what(), "Attempt to add a new vertex with the same name");

}

}

TEST\_METHOD(newEdge\_FF)

{

Graph test;

test.NewVertex("1");

test.NewVertex("2");

test.NewVertex("3");

test.NewVertex("4");

float compare = 0;

Assert::AreEqual(compare, test.FF(0, 1));

test.newEdge("1", "2", 5);

compare = 5;

Assert::AreEqual(compare, test.FF(0, 1));

try

{

test.FF(10, 1);

}

catch (const std::exception& ex)

{

Assert::AreEqual(ex.what(), "Out of range");

}

test.newEdge("1", "4", 10);

test.newEdge("4", "2", 20);

test.newEdge("2", "3", 90);

test.newEdge("3", "4", 100);

compare = 15;

Assert::AreEqual(compare, test.FF(0, 1));

compare = 20;

Assert::AreEqual(compare, test.FF(2, 1));

try

{

test.FF("5", "1");

}

catch (const std::exception& ex)

{

Assert::AreEqual(ex.what(), "Couldn't find such vertex");

}

}

TEST\_METHOD(fileDoesntOpenTest)

{

Graph test;

try

{

test.fileRead("input.txt");

}

catch (invalid\_argument error)

{

Assert::AreEqual("Error, file doesnt open", error.what());

}

}

};

}