МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный

электротехнический университет

«ЛЭТИ» им. В.И. Ульянова (Ленина)

Кафедра САПР

Курсовая РАБОТА

по дисциплине «Алгоритмы и структуры данных»

**Тема:** Построение орграфа и поиск максимального потока методом Форда - Фалкерсона

Вариант 1

|  |  |  |
| --- | --- | --- |
| Студент гр. 9309 |  | Юшин Е.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

Оглавление

[Постановка задачи 4](#_Toc83126878)

[Обоснование выбора используемых структур данных 4](#_Toc83126879)

[Описание алгоритма решения 4](#_Toc83126880)

[Текст программы 5](#_Toc83126881)

[Пример работы программы 15](#_Toc83126882)

[Вывод 15](#_Toc83126883)

ЗАДАНИЕ

на курсовую работу

|  |  |  |
| --- | --- | --- |
| Студент Юшин Е.В. | | |
| Группа 9309 | | |
| Тема работы: Нахождение максимального потока в сети методом Форда - Фалкерсона. | | |
| Содержание пояснительной записки:  «Постановка задачи», «Обоснование выбора используемых структур данных», «Описание алгоритма решения», «Пример работы», «Листинг». | | |
|  | | |
| Дата выдачи задания: 29.03.2021 | | |
| Дата сдачи задания: 21.09.2021 | | |
|  | | |
| Студент гр. 9309 |  | Юшин Е.В. |
| Преподаватель |  | Тутуева А.В. |

Аннотация к пояснительной записке

Данная программа получает из файла информацию о дугах транспортной сети, строит транспортную сеть с помощью ассоциативного списка, после чего вычисляет максимальный поток в сети.

Summary

This program gets information about the arcs of the transport network from the file, builds the transport network using an associative list, and then calculates the maximum flow in the network.

# Постановка задачи

Программа должна считать из файла информацию о дугах транспортной сети в следующем виде: вершина 1; вершина 2; вес ребра. На основе полученной информации создается граф, представленный в памяти в виде ассоциативного массива. Максимальный поток вычисляется с помощью алгоритма Форда-Фалкерсона.

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

# Обоснование выбора используемых структур данных

Программа реализует ассоциативный массив. Его преимущество в том, что он сформирован из простейших в обращении линейных структур – списков, что значительно упрощает его реализацию и уменьшает расходуемую память. Минусом такой реализации является сложность в обращении к элементам структуры.

# Описание алгоритма решения

1. Функция main передает в конструктор название файла, содержащего данные графа.
2. Конструктор создает ассоциативный массив (список смежности для графа) и запускает функцию inp\_str(), которая осуществляет открытие файла и проверку его содержимого на правильность данных.
3. Запускается цикл вызова функции str\_correct() для каждой строки файла. Функция принимает указатель на поток ifstream и читает очередную строку, после чего разбивает ее на три подстроки: 2 вершины и вес ребра.
4. Далее происходит вызов функции cur\_edge(). Функция cur\_edge() совместно с полученной из файла дугой создает такую же, но направленную в противоположном направлении.
5. При возвращении в конструктор выполняется проверка наличия в сформированном графе S- и T- элементов.
6. Далее в функции main происходит вызов функции get\_max\_flow\_size(), которая возвращает величину максимального потока.

# Текст программы

**Source.cpp**

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <string>

#include <limits>

#include "Graphs.h"

using namespace std;

int main()

{

try

{

Graphs graph("test.txt");

cout << "Maximum flow in network is: " << graph.get\_max\_flow\_size() << endl;

}

catch (exception& e)

{

cout << "Error: " << e.what() << endl;

}

}

**List.h**

#pragma once

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <string>

#include <limits>

using namespace std;

class List2

{

int data;

int flow\_size;

List2\* next;

List2\* prev;

public:

List2(int Data, int Key)

{

next = nullptr;

prev = nullptr;

data = Data;

flow\_size = Key;

}

friend class List;

friend class ListsList;

friend class Graphs;

~List2()

{

if (next)

next->prev = prev;

if (prev)

prev->next = next;

}

};

class List

{

List2\* start;

List2\* last;

size\_t value;

public:

List()

{

start = nullptr;

last = nullptr;

value = 0;

}

bool isEmpty()

{

return value == 0;

}

size\_t getSize()

{

return value;

}

void push\_back(int data, int flow\_size)

{

if (!isEmpty())

{

last->next = new List2(data, flow\_size);

last->next->prev = last;

last = last->next;

}

else

{

start = new List2(data, flow\_size);

last = start;

}

value += 1;

}

void push\_front(int data, int flow\_size)

{

if (!isEmpty())

{

start->prev = new List2(data, flow\_size);

start->prev->next = start;

start = start->prev;

}

else

{

start = new List2(data, flow\_size);

last = start;

}

value += 1;

}

void pop\_back()

{

if (isEmpty())

throw runtime\_error("List is empty\n");

List2\* newLast = last->prev;

delete last;

last = newLast;

if (!newLast)

start = nullptr;

value -= 1;

}

void pop\_front()

{

if (isEmpty())

throw runtime\_error("List is empty\n");

List2\* newHead = start->next;

delete start;

start = newHead;

if (!newHead)

last = nullptr;

value -= 1;

}

List2\* at(size\_t index)

{

if (index >= value)

return nullptr;

if (index == 0)

return start;

if (index == (value - 1))

return last;

int count;

List2\* Cur;

if ((value / 2) > index)

{

Cur = start;

count = 0;

while (count != index)

{

Cur = Cur->next;

count++;

}

}

else

{

Cur = last;

count = value - 1;

while (count != index)

{

Cur = Cur->prev;

count--;

}

}

return Cur;

}

List2\* find\_data(int data)

{

List2\* Cur = start;

while (Cur)

{

if (Cur->data == data)

return Cur;

Cur = Cur->next;

}

return nullptr;

}

void insert(int data, int flow\_size, size\_t index)

{

if (index == 0)

push\_front(data, flow\_size);

else

{

try

{

List2\* Next = at(index);

List2\* Prev = Next->prev;

List2\* Allow = new List2(data, flow\_size);

Prev->next = Allow;

Allow->prev = Prev;

Next->prev = Allow;

Allow->next = Next;

value += 1;

}

catch (out\_of\_range& exception) { throw out\_of\_range("Index is out of range\n"); }

}

}

void remove(size\_t index)

{

try

{

if (index == 0)

pop\_front();

else if (index == (value - 1))

pop\_back();

else

{

List2\* Ignore = at(index);

delete Ignore;

value -= 1;

}

}

catch (runtime\_error& exception) { throw runtime\_error("List is empty\n"); }

catch (out\_of\_range& exception) { throw out\_of\_range("Index is out of range\n"); }

}

void set(int data, int flow\_size, size\_t index)

{

List2\* str;

try

{

str = at(index);

str->data = data;

str->flow\_size = flow\_size;

}

catch (out\_of\_range& exception) { throw out\_of\_range("Index is out of range\n"); }

}

void clear()

{

while (!isEmpty())

pop\_front();

}

~List()

{

clear();

}

};

**List2.h**

#pragma once

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <string>

#include <limits>

#include "list.h"

using namespace std;

class ListsElem

{

List\* data;

string tag;

ListsElem\* next;

public:

ListsElem(string Tag, List\* Data)

{

next = nullptr;

data = Data;

tag = Tag;

}

friend class ListsList;

friend class Graphs;

~ListsElem()

{

if (data)

delete data;

}

};

class ListsList

{

ListsElem\* start;

size\_t value;

public:

ListsList()

{

start = nullptr;

value = 0;

}

size\_t getSize()

{

return value;

}

bool add\_edge(string from, string to, int weight)

{

ListsElem\* cur = find(from);

if (!cur)

{

List\* lst = new List;

push\_back(from, lst);

cur = find(from);

}

int size\_to = get\_value(to);

if (size\_to == -1)

{

List\* lst = new List;

push\_back(to, lst);

size\_to = get\_value(to);

}

List2\* edge = cur->data->find\_data(size\_to);

if (edge)

return 0;

cur->data->push\_back(size\_to, weight);

return 1;

}

void push\_back(string tag, List\* data)

{

ListsElem\* Cur = start;

if (Cur)

{

while (Cur->next)

Cur = Cur->next;

Cur->next = new ListsElem(tag, data);

}

else

start = new ListsElem(tag, data);

value++;

}

ListsElem\* find(string tag)

{

ListsElem\* Cur = start;

while (Cur)

{

if (Cur->tag == tag)

return Cur;

Cur = Cur->next;

}

return nullptr;

}

ListsElem\* at(size\_t index)

{

ListsElem\* Cur;

Cur = start;

int count = 0;

while (count != index)

{

Cur = Cur->next;

count++;

}

return Cur;

}

int get\_value(string tag)

{

ListsElem\* Cur = start;

int index = -1;

while (Cur)

{

index++;

if (Cur->tag == tag)

return index;

Cur = Cur->next;

}

return -1;

}

~ListsList()

{

ListsElem\* Cur = start;

ListsElem\* next;

while (Cur)

{

next = Cur->next;

delete Cur;

Cur = next;

}

}

**};Graphs.h**

#pragma once

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <string>

#include <limits>

#include "list2.h"

using namespace std;

class Graphs

{

class Iterator

{

class Stack

{

struct Elem

{

ListsElem\* data;

Elem\* next;

};

Elem\* start;

public:

Stack()

{

start = nullptr;

}

bool contains(ListsElem\* E)

{

Elem\* Cur = start;

while (Cur)

{

if (Cur->data == E)

return true;

Cur = Cur->next;

}

return false;

}

bool is\_empty()

{

return (!start);

}

void push(ListsElem\* Push)

{

Elem\* New = new Elem;

New->data = Push;

New->next = start;

start = New;

}

ListsElem\* pop()

{

if (!start)

return nullptr;

Elem\* nextHead = start->next;

ListsElem\* ans = start->data;

delete start;

start = nextHead;

return ans;

}

int find\_min\_edge(ListsElem\* E, ListsList\* list)

{

int minEdge = 2147483647;

int temp;

ListsElem\* ListsCur = E;

ListsElem\* ListsParent;

Elem\* StackCur = start;

while (StackCur)

{

ListsParent = StackCur->data;

temp = ListsParent->data->find\_data(list->get\_value(ListsCur->tag))->flow\_size;

if (temp < minEdge)

minEdge = temp;

ListsCur = ListsParent;

StackCur = StackCur->next;

}

return minEdge;

}

void clear()

{

while (start)

pop();

}

~Stack()

{

clear();

}

};

Stack\* Way;

Stack\* Visited;

ListsElem\* Cur;

ListsList\* list;

public:

Iterator(ListsElem\* Start, ListsList\* lists)

{

Cur = Start;

list = lists;

Way = new Stack;

Visited = new Stack;

Visited->push(Cur);

}

ListsElem\* get\_cur()

{

return Cur;

}

void prev(int quantity)

{

ListsElem\* Prev = Way->pop();

int cur\_index = list->get\_value(Cur->tag);

int prev\_index = list->get\_value(Prev->tag);

List2\* edge = Prev->data->find\_data(cur\_index);

List2\* other\_edge = Cur->data->find\_data(prev\_index);

edge->flow\_size -= quantity;

other\_edge->flow\_size += quantity;

Cur = Prev;

}

void next()

{

List2\* Cur2 = Cur->data->at(0);

int max\_edge = 0;

while (Cur2)

{

if (!Visited->contains(list->at(Cur2->data)))

{

if (max\_edge < Cur2->flow\_size)

max\_edge = Cur2->flow\_size;

}

Cur2 = Cur2->next;

}

Cur2 = Cur->data->at(0);

while (Cur2)

{

if (!Visited->contains(list->at(Cur2->data)) && Cur2->flow\_size == max\_edge)

break;

Cur2 = Cur2->next;

}

Way->push(Cur);

Cur = list->at(Cur2->data);

Visited->push(Cur);

}

bool has\_next()

{

List2\* Cur2 = Cur->data->at(0);

while (Cur2)

{

if (Cur2->flow\_size > 0)

{

if (!Visited->contains(list->at(Cur2->data)))

return true;

}

Cur2 = Cur2->next;

}

return false;

}

bool has\_prev()

{

return (!Way->is\_empty());

}

int to\_start()

{

int quantity = Way->find\_min\_edge(Cur, list);

while (has\_prev())

prev(quantity);

Visited->clear();

Visited->push(Cur);

if (quantity == INT\_MAX)

return 0;

return quantity;

}

~Iterator()

{

delete Way;

delete Visited;

}

};

ListsList\* lists;

bool cur\_edge(string tag1, string tag2, string edge)

{

if (edge.empty() || tag2.empty())

return 0;

int weight;

try { weight = stoi(edge); }

catch (exception) { return 0; }

if (weight < 0)

return 0;

bool indicator = lists->add\_edge(tag1, tag2, weight);

if (indicator)

indicator = lists->add\_edge(tag2, tag1, 0);

return indicator;

}

bool str\_correct(ifstream\* inp\_str)

{

string str;

getline(\*inp\_str, str);

if (str.empty())

return 0;

size\_t tmp1 = 0, tmp2 = 0;

string tag1, tag2, edge;

for (int i = 0; i < str.length(); i++)

{

if (str[i] == ';')

{

if (tmp1 == 0)

tmp1 = i;

else if (tmp2 == 0)

tmp2 = i;

else

return 0;

}

}

if (tmp1 == 0 || tmp2 == 0)

return 0;

edge = str.substr(tmp2 + 1);

str.erase(tmp2);

tag2 = str.substr(tmp1 + 1);

str.erase(tmp1);

tag1 = str;

if (!cur\_edge(tag1, tag2, edge))

return 0;

return 1;

}

void inp\_str(string filename)

{

ifstream inp\_str;

inp\_str.open(filename);

if (!inp\_str.is\_open())

throw runtime\_error("Could not open file " + filename);

while (!inp\_str.eof())

{

if (!str\_correct(&inp\_str))

{

inp\_str.close();

delete lists;

throw invalid\_argument("Incorrect inp\_str");

}

}

inp\_str.close();

}

public:

Graphs(string filename)

{

lists = new ListsList;

inp\_str(filename);

if ((!lists->find("T") || (!lists->find("S"))))

throw invalid\_argument("Graph must contain S- and T- elements");

}

int get\_max\_flow\_size()

{

int max\_flow = 0;

ListsElem\* Cur = lists->find("S");

Iterator\* iter = new Iterator(Cur, lists);

while (iter->has\_next())

{

Cur = iter->get\_cur();

while (Cur->tag != "T")

{

if (iter->has\_next())

{

iter->next();

Cur = iter->get\_cur();

}

else

{

while (!iter->has\_next() && iter->has\_prev())

iter->prev(0);

Cur = iter->get\_cur();

if (Cur->tag == "S" && !iter->has\_next())

{

delete iter;

return max\_flow;

}

}

}

max\_flow += iter->to\_start();

}

delete iter;

return max\_flow;

}

~Graphs()

{

delete lists;

}

**};UnitTest4.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "pch.h"

#include "CppUnitTest.h"

#include "../CourseW0rk/Source.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace courseworkUnitTests

{

TEST\_CLASS(courseworkUnitTests)

{

public:

TEST\_METHOD(get\_max\_flow\_size\_check1)

{

ofstream tester("tester.txt");

tester << "S;O;3\n";

tester << "S;P;3\n";

tester << "O;Q;3\n";

tester << "O;P;2\n";

tester << "P;R;2\n";

tester << "Q;R;4\n";

tester << "Q;T;2\n";

tester << "R;T;3";

tester.close();

Graphs gr("tester.txt");

Assert::AreEqual(5, gr.get\_max\_flow\_size());

}

TEST\_METHOD(get\_max\_flow\_size\_check2)

{

ofstream tester("tester.txt");

tester << "S;A;8\n";

tester << "S;B;4\n";

tester << "S;C;2\n";

tester << "A;B;2\n";

tester << "B;C;1\n";

tester << "A;D;9\n";

tester << "C;F;2\n";

tester << "D;G;8\n";

tester << "F;I;1\n";

tester << "G;H;5\n";

tester << "H;I;1\n";

tester << "G;T;4\n";

tester << "H;T;3\n";

tester << "I;T;6";

tester.close();

Graphs gr("tester.txt");

Assert::AreEqual(9, gr.get\_max\_flow\_size());

}

TEST\_METHOD(get\_max\_flow\_size\_check3)

{

ofstream tester("tester.txt");

tester << "S;O;3\n";

tester << "S;P;3\n";

tester << "O;Q;3\n";

tester << "O;P;2\n";

tester << "P;R;2\n";

tester << "Q;R;4\n";

tester << "X;T;1";

tester.close();

Graphs gr("tester.txt");

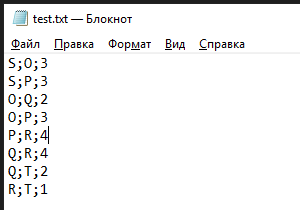
Assert::AreEqual(0, gr.get\_max\_flow\_size());

}

};

}

# Пример работы программы



## 

# Вывод

**В ходе работы над курсовым проектом были обобщены полученные в течение семестра знания. Был изучен и реализован на языке C++ алгоритм Форда-Фалкерсона, позволяющий находить максимальный поток в транспортной сети.**