**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**КУРСОВАЯ РАБОТА**

**по дисциплине «Алгоритмы и Структуры Данных»**

**на тему: «Потоки в сетях»**

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8302 |  | Зубарева Е.А. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 3](#_Toc41846200)

[Описание классов и методов 3](#_Toc41846201)

[Обоснование использования структур данных 4](#_Toc41846202)

[Описание алгоритма решения 4](#_Toc41846203)

[Оценка временной сложности 5](#_Toc41846204)

[Описание unit-тестов 5](#_Toc41846205)

[Пример работы 6](#_Toc41846206)

[Код 9](#_Toc41846207)

## Постановка задачи

Сетью называется орграф без циклов с помеченными вершинами и дугами. Числа, которыми помечаются дуги сети, называются пропускными способностями дуг.

Примеры вершин сети: перекрёстки дорог, телефонные узлы, железнодорожные узлы, аэропорты, склады и т.д. Примеры дуг сети: дороги, трубы, телефонные и железнодорожные линии и т.д.

Сеть, у которой существует ровно один исток (s) и один сток(t), называется транспортной сетью.

В теории оптимизации и теории графов, задача о максимальном потоке заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока, или, что то же самое, сумма потоков в сток максимальна.

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.

Найти максимальный поток в сети используя алгоритм Форда-Фалкерсона.

## Описание классов и методов

Реализованы следующие классы:

1. Graph<VertexData, EdgeData> – представляет собой граф на основе списков смежности, где вершины и ребра могут хранить в себе дополнительные произвольные данные. Класс графа предоставляет методы добавления вершин и ребер, удаления ребер, поиска вершин и получения ребра, соединяющего вершины. Так же есть методы обхода всех вершин и вывода.
2. Graph::Vertex – вершина графа, содержит VertexData, список (List) прилегающих ребер (Graph::Edge) и внутренние параметры для вывода.
3. Graph::Edge – одностороннее ребро графа, содержит EdgeData и две соединяемые упорядоченные вершины, а так же внутренние параметры для вывода.
4. FlowNetwork – представляет собой транспортную сеть на основе класса Graph, содержит указатели на вершины истока и стока и реализует структуры, описывающие узлы и ребра транспортной сети. Реализует методы загрузки из файла, нахождения максимального потока, а также на основе методов графа реализует красивый вывод в консоль.

Так же из предыдущих лабораторной работы были использованы классы Array<T> и List<T>.

## Обоснование использования структур данных

Граф описывается списками смежности. Эта структура была выбрана потому, что каждая вершина уже хранит ребра, которые подсоединены к ней. В случае транспортной сети ребер, подсоединенных к одной вершине достаточно мало в силу отсутствия циклов и в силу того, что все пути идут из истока в сток, потому в среднем алгоритмы на основе списков смежности будут работать для транспортных сетей быстрее, чем алгоритмы на основе матрицы смежности, где им придется обходить все вершины для поиска ребер.

## Описание алгоритма решения

Алгоритм решения – алгоритм Форда - Фалкерсона, требует наличия у каждого ребра обратного ему, потому перед его запуском создаются ребра с максимальной пропускной способностью 0, поток которых может становиться отрицательным. Для быстрого доступа у каждого ребра имеется указатель на противоположное. После того, как алгоритм завершает работу, все такие ребра удаляются из графа.

Для поиска произвольного пути в графе (что требуется основному алгоритму) используется поиск в глубину.

Кроме того реализован алгоритм валидации, который с помощью поиска в глубину проверяет все пути на то, что они ведут строго из истока в сток и не имеют циклов.

## Оценка временной сложности

Время алгоритма Форда - Фалкерсона из-за поиска произвольного пути, а не кратчайшего, зависит не только от количества вершин и ребер, но и от того, какие значения максимальных пропускных способностей заданы в графе. Можно привести пример, где алгоритм работает очень долго на очень маленьком графе в определенной ситуации.

Время подготовки и удаления временных ребер – O(|E|)

Время одного поиска в глубину от истока к стоку – O(|V|)

Алгоритм валидации проверяет все пути в транспортной сети, хотя в произвольном графе вариантов путей достаточно много, в транспортной сети их может быть максимально |V| - 1, значит сложность алгоритма – O(|V|), потому что в случае, если он находит лишний путь, он сразу завершается.

## Описание unit-тестов

Кроме юнит тестов массива проверяется так же построение графа (граф строится по определенным правилам, а потом перебором всех ребер проверяется то, что эти правила сохраняются).

Далее проверяется сама транспортная сеть. Сначала, как и для графа, проверяется построение, однако в этот раз для удобства используется метод добавления ребер из строки. Потом проверяются различные результаты валидации, поиск в глубину с и без возможного пути из истока в сток и, наконец, построение максимального потока. При его построении в наперед заданной сети проверяются потоки на истоке и стоке, а так же проверяется то, что алгебраическая сумма потоков через остальные вершины равна нулю.

## Пример работы

На первом примере показан весь процесс работы программы с транспортной сетью, данной в задании. В следующих примерах показаны транспортные сети с уже построенным максимальным потоком и подписью имени файла, из которого они загружены.

Network:

empty

Network is invalid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:3

input file name:../data/example.txt

../data/example.txt

success

Network:

S O P Q R T

|--- 0/3 -->|--- 0/2 -->|--------- 0/2 -------->|--- 0/3 -->|

|--------- 0/3 -------->| |--- 0/4 -->| |

| |--------- 0/3 -------->|--------- 0/2 -------->|

Network is valid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:1

Network:

S O P Q R T

|--- 3/3 -->|--- 0/2 -->|--------- 2/2 -------->|--- 3/3 -->|

|--------- 2/3 -------->| |--- 1/4 -->| |

| |--------- 3/3 -------->|--------- 2/2 -------->|

Network is valid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:2

Edge format: A B max\_flow, where:

A - start vertex

B - end vertex

max\_flow - edge max flow (0 to remove edge)

input edge:S T 10

Network:

S O P Q R T

|--- 3/3 -->|--- 0/2 -->|--------- 2/2 -------->|--- 3/3 -->|

|--------- 2/3 -------->| |--- 1/4 -->| |

| |--------- 3/3 -------->|--------- 2/2 -------->|

|--------------------------- 0/10-------------------------->|

Network is valid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:1

Network:

S O P Q R T

|--- 3/3 -->|--- 0/2 -->|--------- 2/2 -------->|--- 3/3 -->|

|--------- 2/3 -------->| |--- 1/4 -->| |

| |--------- 3/3 -------->|--------- 2/2 -------->|

|---------------------------10/10-------------------------->|

Network is valid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:2

Edge format: A B max\_flow, where:

A - start vertex

B - end vertex

max\_flow - edge max flow (0 to remove edge)

input edge:Q S 1

Network:

S O P Q R T

|--- 3/3 -->|--- 0/2 -->|--------- 2/2 -------->|--- 3/3 -->|

|--------- 2/3 -------->| |--- 1/4 -->| |

| |--------- 3/3 -------->|--------- 2/2 -------->|

|<-------------- 0/1 ---------------| | |

|---------------------------10/10-------------------------->|

Network is invalid

commands:

1 - rebuild flow

2 - add/remove edge

3 - open file

other - exit

input command:1

Cannot rebuild max flow on invalid network!

Файл: ../data/smol.txt

Network:

S A B T

|--- 3/3 -->|--------- 3/4 -------->|

|--------- 2/2 -------->|--- 2/2 -->|

Network is valid

Файл: ../data/1.txt

Network:

S A B C D E F H G T

|--- 8/10-->|--- 1/1 -->|--- 0/1 -->|--- 0/1 -->|--- 1/1 -->|--------------------- 1/1 -------------------->|

| |--------- 2/2 -------->|--------------------------------- 2/3 -------------------------------->|

| |--------------- 2/3 -------------->|--------------------------- 1/1 -------------------------->|

| |--------------------- 0/3 -------------------->| |--------------- 1/1 -------------->|

| |--------------------------- 1/1 -------------------------->| |--------- 1/1 -------->|

| |--------------------------------- 1/1 -------------------------------->| |--- 1/1 -->|

| |--------------------------------------- 1/1 -------------------------------------->| |

| | |--------------------------------------- 1/3 -------------------------------------->|

Network is valid

Файл: ../data/2.txt

Network:

S A B C D E F G H T

|--- 2/2 -->|--- 1/1 -->|--- 1/1 -->|--- 1/1 -->|--- 1/2 -->|--- 2/2 -->|--------------- 2/2 -------------->|

|--------- 2/2 -------->|--------------- 2/2 -------------->|--------- 2/2 -------->|--------- 2/2 -------->|

|--------------- 2/2 -------------->|--------- 2/2 -------->|--------------- 2/2 -------------->|--- 2/2 -->|

| |--------------------- 1/2 -------------------->| | | | |

Network is valid

Файл: ../data/3.txt

Network:

S A B C D E F G H T

|--- 5/10-->|--- 2/2 -->|--- 2/3 -->|--- 2/2 -->|--- 2/4 -->|--- 2/3 -->|--- 2/3 -->|--- 2/3 -->|--- 5/5 -->|

| |--------------------------------------- 3/3 -------------------------------------->| | Network is valid

Файл: ../data/4.txt

Network:

S A B C D E F H T I

|--- 4/10-->|--- 2/2 -->|--- 2/3 -->|--- 2/2 -->| |--- 2/3 -->|--- 2/2 -->|--- 2/5 -->|<-- 3/4 ---|

| |--------------------- 2/2 -------------------->| | | | |

| | | | |--------------------- 2/5 -------------------->| |

|--------------------------------------------------- 3/3 -------------------------------------------------->|

Network is valid

Файл: ../data/5.txt

Network:

A8 T S A1 A2 A3 A4 A5 A6 A7

|--- 6/10-->| |--- 6/9 -->|--- 6/10-->|--- 6/8 -->|--- 6/6 -->|--- 6/7 -->|--- 6/7 -->|--- 6/8 -->|

|<-------------------------------------------------- 6/11---------------------------------------------------|

Network is valid

Файл: ../data/6.txt

Network:

A8 T S A1 A2 A3 A4 A5 A6 A7

|--- 9/10-->| |--- 9/9 -->|--- 8/10-->|--- 7/8 -->|--- 6/6 -->|--- 6/7 -->|--- 6/7 -->|--- 7/8 -->|

|<-------------- 1/1 ---------------| | |--------------- 1/1 -------------->| |

| | | | |--------------------------- 1/1 -------------------------->|

|<-------------------------------------------------- 8/11---------------------------------------------------|

Network is valid

Файл: ../data/7.txt

Network:

S B1 B2 B3 B4 B5 B6 B7 B8 T

|--- 1/1 -->|--------------------------------------------- 1/8 -------------------------------------------->|

|--------- 2/2 -------->|--------------------------------------- 2/7 -------------------------------------->|

|--------------- 3/3 -------------->|--------------------------------- 3/6 -------------------------------->|

|--------------------- 4/4 -------------------->|--------------------------- 4/5 -------------------------->|

|--------------------------- 4/5 -------------------------->|--------------------- 4/4 -------------------->|

|--------------------------------- 3/6 -------------------------------->|--------------- 3/3 -------------->|

|--------------------------------------- 2/7 -------------------------------------->|--------- 2/2 -------->|

|--------------------------------------------- 1/8 -------------------------------------------->|--- 1/1 -->|

Network is valid

## Код

main.cpp

#include <iostream>  
  
#include "flow\_network.h"  
  
#include <iostream>  
#include "graph.h"  
  
**int** main() {  
 FlowNetwork network;  
  
 **bool** run = **true**;  
 **while** (run) {  
 std::cout << "\n\n\nNetwork:\n";  
 network.print();  
 **bool** is\_valid = network.is\_valid();  
 std::cout << (is\_valid ? "Network is valid" : "Network is invalid") << "\n";  
 std::cout << "\n\ncommands: \n 1 - rebuild flow\n 2 - add/remove edge\n 3 - open file\n other - exit\n \ninput command: ";  
  
 **int** cmd;  
 std::cin >> cmd;  
 getchar();  
 **switch** (cmd) {  
 **case** 1: {  
 **if** (is\_valid) {  
 network.build\_max\_flow();  
 } **else** {  
 std::cout << "Cannot rebuild max flow on invalid network!\n";  
 }  
 **break**;  
 }  
 **case** 2: {  
 std::string line;  
 std::cout << "Edge format: A B max\_flow, where:\n A - start vertex\n B - end vertex \n max\_flow - edge max flow (0 to remove edge)\ninput edge: ";  
 std::getline(std::cin, line);  
 **if** (!network.parse\_edge(line)) {  
 std::cout << "failed to parse edge!\n";  
 }  
 **break**;  
 }  
 **case** 3: {  
 std::string filename;  
 std::cout << "input file name: ";  
 std::cin >> filename;  
 **if** (network.from\_file(filename)) {  
 std::cout << "success\n";  
 } **else** {  
 std::cout << "failed to read\n";  
 }  
 **break**;  
 }  
 **default**:  
 run = **false**;  
 **break**;  
 }  
 std::cout << "\n";  
 }  
  
 **return** 0;  
}

tests.cpp

#include "gtest/gtest.h"  
#include "../array.h"  
#include "../flow\_network.h"  
  
  
// test basic graph functionality  
TEST (graph, build\_test) {  
 Graph<std::string, std::string> graph;  
 Graph<std::string, std::string>::Vertex& v1 = graph.add\_vertex("A");  
 ASSERT\_EQ(graph.size(), 1);  
 Graph<std::string, std::string>::Vertex& v2 = graph.add\_vertex("B");  
 ASSERT\_EQ(graph.size(), 2);  
 ASSERT\_EQ(&v1, graph.get\_vertex("A"));  
 ASSERT\_EQ(&v2, graph.get\_vertex("B"));  
 graph.connect(v1, v2, "Edge");  
 ASSERT\_EQ(graph.get\_edge(v1, v2)->data, "Edge");  
 graph.clear();  
 ASSERT\_EQ(graph.size(), 0);  
}  
  
TEST (graph, iteration\_test) {  
 Graph<**int**, **int**> graph;  
 **for** (**int** i = 0; i < 10; i++) {  
 Graph<**int**, **int**>::Vertex& v1 = graph.add\_vertex(i);  
 **for** (**int** j = 0; j < i; j++) {  
 **auto** v2 = graph.get\_vertex(j);  
 ASSERT\_NE(v2, **nullptr**);  
 graph.connect(\*v2, v1, i \* j);  
 }  
 }  
  
 ASSERT\_EQ(graph.size(), 10);  
  
 graph.for\_each([&] (Graph<**int**, **int**>::Vertex& v) -> **void** {  
 v.for\_each([&](Graph<**int**, **int**>::Edge &edge) -> **void** {  
 ASSERT\_LE(edge.from->data, edge.connected->data);  
 ASSERT\_EQ(edge.data, edge.from->data \* edge.connected->data);  
 });  
 });  
}  
  
  
// flow networks test  
TEST(flow\_network, build) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S T 10"));  
 ASSERT\_FALSE(network.parse\_edge("A 10"));  
 ASSERT\_EQ(network.graph.size(), 2);  
 FlowNetwork::Vertex\* v1 = network.graph.get\_vertex("S");  
 FlowNetwork::Vertex\* v2 = network.graph.get\_vertex("T");  
 ASSERT\_NE(v1, **nullptr**);  
 ASSERT\_NE(v2, **nullptr**);  
 ASSERT\_EQ(v1, network.source);  
 ASSERT\_EQ(v2, network.target);  
 FlowNetwork::Edge\* edge1 = network.graph.get\_edge(\*v1, \*v2);;  
 FlowNetwork::Edge\* edge2 = network.graph.get\_edge(\*v2, \*v1);  
 ASSERT\_NE(edge1, **nullptr**);  
 ASSERT\_EQ(edge2, **nullptr**);  
 ASSERT\_EQ(edge1->data.max\_flow, 10);  
}  
  
TEST(flow\_network, validate\_ok) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("S B 2"));  
 ASSERT\_TRUE(network.parse\_edge("A B 3"));  
 ASSERT\_TRUE(network.parse\_edge("B T 4"));  
 ASSERT\_EQ(network.is\_valid(), **true**);  
}  
  
TEST(flow\_network, validate\_invalid\_1) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("S B 2"));  
 ASSERT\_TRUE(network.parse\_edge("A B 3"));  
 ASSERT\_EQ(network.is\_valid(), **false**);  
  
 network.clear();  
 ASSERT\_TRUE(network.parse\_edge("A T 1"));  
 ASSERT\_TRUE(network.parse\_edge("T B 2"));  
 ASSERT\_TRUE(network.parse\_edge("A B 3"));  
 ASSERT\_EQ(network.is\_valid(), **false**);  
}  
  
TEST(flow\_network, validate\_invalid\_2) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("T S 2"));  
 ASSERT\_TRUE(network.parse\_edge("A T 3"));  
 ASSERT\_EQ(network.is\_valid(), **false**);  
}  
  
TEST(flow\_network, validate\_invalid\_3) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("T B 3"));  
 ASSERT\_TRUE(network.parse\_edge("A B 2"));  
 ASSERT\_EQ(network.is\_valid(), **false**);  
}  
  
TEST(flow\_network, dfs\_success) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("A B 1"));  
 ASSERT\_TRUE(network.parse\_edge("B C 1"));  
 ASSERT\_TRUE(network.parse\_edge("C T 1"));  
 Array<FlowNetwork::Edge\*> path;  
 ASSERT\_EQ(network.search\_path(network.source, path), **true**);  
 ASSERT\_EQ(path.length(), 4);  
}  
  
TEST(flow\_network, dfs\_fail) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("A B 1"));  
 ASSERT\_TRUE(network.parse\_edge("C T 1"));  
 Array<FlowNetwork::Edge\*> path;  
 ASSERT\_EQ(network.search\_path(network.source, path), **false**);  
}  
  
TEST(flow\_network, max\_flow\_build) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.parse\_edge("S A 1"));  
 ASSERT\_TRUE(network.parse\_edge("A B 1"));  
 ASSERT\_TRUE(network.parse\_edge("B T 1"));  
 ASSERT\_TRUE(network.parse\_edge("S T 2"));  
 ASSERT\_EQ(network.is\_valid(), **true**);  
 network.build\_max\_flow();  
 ASSERT\_EQ(network.vertex\_flow(\*network.source), -3);  
 ASSERT\_EQ(network.vertex\_flow(\*network.target), 3);  
 network.graph.for\_each([&] (FlowNetwork::Vertex& vertex) -> **void** {  
 **if** (&vertex != network.source && &vertex != network.target) {  
 ASSERT\_EQ(network.vertex\_flow(vertex), 0);  
 }  
 });  
}  
  
// array tests  
TEST(Array, MassiveAddAndClearTest) {  
 Array<std::string> array;  
 **const int** size = 10000;  
 **for** (**int** i = 0; i < size; i++) {  
 array.add("test\_string");  
 }  
 ASSERT\_EQ(array.length(), size);  
 **for** (**int** i = 0; i < size; i++) {  
 ASSERT\_EQ(array[i], "test\_string");  
 }  
 array.clear();  
 ASSERT\_EQ(array.length(), 0);  
}

list.h

#include <string>  
  
  
#ifndef LAB1\_LIST\_H  
#define LAB1\_LIST\_H  
  
**template** <**typename** T>  
**class** List {  
**public**:  
 **class** Node {  
 **public**:  
 T value;  
 Node\* next = **nullptr**;  
 Node\* prev = **nullptr**;  
  
 Node() {};  
 Node(T **const**& v) : value(v) {}  
 };  
  
 **class** Iterator {  
 **public**:  
 Node\* node = **nullptr**;  
  
 Iterator() {}  
 Iterator(Node\* n) : node(n) {}  
  
 Iterator **operator**++(**int**) {  
 Node\* last = node;  
 node = node->next;  
 **return** Iterator(last);  
 }  
  
 Iterator **operator**--(**int**) {  
 Node\* last = node;  
 node = node->prev;  
 **return** Iterator(last);  
 }  
  
 T& **operator**\*() {  
 **return** node->value;  
 }  
  
 T\* **operator**->() {  
 **return** &node->value;  
 }  
  
 **bool operator**==(Iterator **const**& it) {  
 **return** it.node == node;  
 }  
  
 **bool operator**!=(Iterator **const**& it) {  
 **return** it.node != node;  
 }  
 };  
  
**private**:  
 Node\* first = **nullptr**;  
 Node\* last = **nullptr**;  
 Node \_end;  
 **int** length = 0;  
  
**public**:  
 Iterator begin() {  
 **return** first != **nullptr** ? Iterator(first) : **end**();  
 }  
  
 Iterator **end**() {  
 **return** Iterator(&\_end);  
 }  
  
 Iterator add(T **const**& value) {  
 Node\* node = **new** Node(value);  
 **if** (last != **nullptr**) {  
 last->next = node;  
 node->prev = last;  
 } **else** {  
 first = last = node;  
 }  
 \_end.prev = node;  
 node->next = &\_end;  
 last = node;  
 length++;  
 **return** Iterator(node);  
 }  
  
 **void** erase(Iterator iterator) {  
 Node\* node = iterator.node;  
 **if** (node->next == &\_end) {  
 last = \_end.prev = node->prev;  
 }  
 **if** (node->prev == **nullptr**) {  
 first = node->next;  
 } **else** {  
 node->prev->next = node->next;  
 }  
 node->next->prev = node->prev;  
 length--;  
 **delete**(node);  
 }  
  
 **void** print() {  
 std::cout << "[";  
 **for** (**auto** it = begin(); it != **end**(); it++) {  
 std::cout << \*it << " ";  
 }  
 std::cout << "]";  
 }  
  
 **void** clear() {  
 Node\* node = first;  
 **while** (node != **nullptr** && node != &\_end) {  
 Node\* next = node->next;  
 **delete**(node);  
 node = next;  
 }  
 \_end.next = \_end.prev = **nullptr**;  
 first = last = **nullptr**;  
 length = 0;  
 }  
  
 **int** get\_length() {  
 **return** length;  
 }  
  
 ~List() {  
 clear();  
 }  
};  
  
#endif //LAB1\_LIST\_H

array.h

#include <iostream>  
#include <stdlib.h>  
#include <functional>  
  
  
#ifndef SEM4LAB2\_ARRAY\_H  
#define SEM4LAB2\_ARRAY\_H  
  
**template** <**typename** T>  
**class** Array {  
 **static const int** REALLOCATION\_ELEMENTS = 16;  
  
 T\* memory\_span = **nullptr**;  
 **int** size = 0;  
 **int** allocated\_size = 0;  
  
 **void** ensure\_size(**int** size) {  
 **if** (allocated\_size < size) {  
 **int** old\_size = allocated\_size;  
 **while** (allocated\_size < size) {  
 allocated\_size += REALLOCATION\_ELEMENTS;  
 }  
  
 **if** (memory\_span != **nullptr**) {  
 T\* new\_memory\_span = **new** T[allocated\_size];  
 **for** (**int** i = 0; i < old\_size; i++) {  
 new\_memory\_span[i] = memory\_span[i];  
 }  
 **delete**[] (memory\_span);  
 memory\_span = new\_memory\_span;  
 } **else** {  
 memory\_span = **new** T[allocated\_size];  
 }  
 }  
 }  
  
**public**:  
 Array() {  
  
 }  
  
 Array(Array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
  
 std::vector<**int**> a = {1, 2};  
 }  
  
 Array(Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = **nullptr**;  
 other.size = 0;  
 other.allocated\_size = 0;  
 }  
  
 Array<T>& **operator**= (Array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
 **return** \***this**;  
 }  
  
 Array<T>& **operator**= (Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = **nullptr**;  
 other.size = 0;  
 other.allocated\_size = 0;  
 **return** \***this**;  
 }  
  
 **inline int** length() {  
 **return** size;  
 }  
  
 **inline** T& **operator**[] (**int** index) {  
 **return** memory\_span[index];  
 }  
  
 T& add(T **const**& elem) {  
 ensure\_size(size + 1);  
 **return** memory\_span[size++] = elem;  
 }  
  
 **void** resize(**int** size) {  
 **if** (size > **this**->size) {  
 ensure\_size(size);  
 **this**->size = size;  
 } **else** {  
 **if** (size == 0) {  
 clear();  
 } **else** {  
 **this**->size = size;  
 }  
 }  
 }  
  
 **bool** contains(T **const**& value) {  
 **for** (**int** i = 0; i < size; i++) {  
 **if** (memory\_span[i] == value) {  
 **return true**;  
 }  
 }  
 **return false**;  
 }  
  
 **bool** any(std::function<**bool**(T **const**&)> check) {  
 **for** (**int** i = 0; i < size; i++) {  
 **if** (check(memory\_span[i])) {  
 **return true**;  
 }  
 }  
 **return false**;  
 }  
  
 **bool** all(std::function<**bool**(T **const**&)> check) {  
 **for** (**int** i = 0; i < size; i++) {  
 **if** (!check(memory\_span[i])) {  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
  
 **void** clear() {  
 **if** (memory\_span != **nullptr**) {  
 **delete**[] (memory\_span);  
 memory\_span = **nullptr**;  
 }  
 size = allocated\_size = 0;  
 }  
  
 ~Array() {  
 clear();  
 }  
};  
  
#endif

graph.h

#include <functional>  
#include <string>  
#include <string.h>  
#include <sstream>  
  
#include "array.h"  
#include "list.h"  
  
  
#ifndef M\_GRAPH\_H  
#define M\_GRAPH\_H  
  
  
**template** <**typename** V, **typename** E>  
  
**class** Graph {  
**public**:  
 **class** Vertex;  
 **class** Edge {  
 **public**:  
 **friend class** Graph;  
  
 Vertex\* connected = **nullptr**;  
 Vertex\* from = **nullptr**;  
 E data;  
  
 **private**:  
 **bool** print\_flag = **false**;  
 **public**:  
  
 Edge() = **default**;  
 Edge(Vertex\* from, Vertex\* connected, E **const**& data) : from(from), connected(connected), data(data) {}  
 };  
  
 **class** Vertex {  
 **public**:  
 **friend class** Graph;  
  
 List<Edge> edges;  
 V data;  
  
 **private**:  
 **int** print\_position = 0;  
 std::string print\_str = "";  
  
 **public**:  
 Vertex() = **default**;  
 Vertex(V **const**& data) : data(data) {};  
  
 **void** for\_each(std::function<**void**(Edge&)> func) {  
 **for** (**auto** it = edges.begin(); it != edges.**end**(); it++) {  
 func(\*it);  
 }  
 }  
 };  
  
**private**:  
 Array<Vertex> vertices;  
  
**public**:  
 Array<Vertex>& get\_vertices() {  
 **return** vertices;  
 }  
  
 **int** size() {  
 **return** vertices.length();  
 }  
  
 Edge& connect(Vertex& v1, Vertex& v2, E **const**& data) {  
 **for** (**auto** i = v1.edges.begin(); i != v1.edges.**end**(); i++) {  
 **if** (i->connected == &v2) {  
 i->data = data;  
 **return** \*i;  
 }  
 }  
 **return** \*v1.edges.add(Edge(&v1, &v2, data));  
 }  
  
 **void** disconnect(Vertex& v1, Vertex& v2) {  
 **for** (**auto** i = v1.edges.begin(); i != v1.edges.**end**(); i++) {  
 **if** (i->connected == &v2) {  
 v1.edges.erase(i);  
 **break**;  
 }  
 }  
 }  
  
 Array<Edge\*> get\_all\_edges() {  
 Array<Edge\*> result;  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 Vertex& vertex = vertices[i];  
 **for** (**auto** i = vertex.edges.begin(); i != vertex.edges.**end**(); i++) {  
 result.add(&\*i);  
 }  
 }  
 **return** result;  
 }  
  
 Edge\* get\_edge(Vertex& v1, Vertex& v2) **const** {  
 **for** (**auto** i = v1.edges.begin(); i != v1.edges.**end**(); i++) {  
 **if** (i->connected == &v2) {  
 **return** &\*i;  
 }  
 }  
 **return nullptr**;  
 }  
  
 **template** <**typename** T>  
 Vertex\* get\_vertex(T **const**& data) {  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 **if** (vertices[i].data == data) {  
 **return** &vertices[i];  
 }  
 }  
 **return nullptr**;  
 }  
  
 Vertex& add\_vertex(V **const**& data) {  
 **return** vertices.add(Vertex(data));  
 }  
  
 **void** for\_each(std::function<**void**(Vertex&)> func) {  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 func(vertices[i]);  
 }  
 }  
  
 **void** clear() {  
 vertices.clear();  
 }  
  
 **void** print(**int** arrow\_length, std::function<std::string(V&)> vertex\_to\_string, std::function<std::string(E&)> edge\_to\_string) {  
 **if** (get\_vertices().length() == 0) {  
 std::cout << "empty\n";  
 **return**;  
 }  
  
 for\_each([] (Vertex& vertex) -> **void** {  
 vertex.for\_each([] (Edge& edge) -> **void** {  
 edge.print\_flag = **false**;  
 });  
 });  
  
 **int** string\_length = 1;  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 vertices[i].print\_position = i;  
 vertices[i].print\_str = vertex\_to\_string(vertices[i].data);  
 **if** (vertices[i].print\_str.length() > string\_length) {  
 string\_length = (**int**) vertices[i].print\_str.length();  
 }  
 }  
  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 Vertex& vertex = vertices[i];  
 std::cout << vertex.print\_str;  
 **int** len = string\_length + arrow\_length - (**int**) vertex.print\_str.length();  
 **for** (**int** j = 0; j < len; j++) {  
 std::cout << " ";  
 }  
 }  
 std::cout << "\n";  
  
 **int** counter = 0;  
 **bool** all\_printed = **false**;  
 **while** (!all\_printed) {  
 **int** line\_length = (string\_length + arrow\_length) \* vertices.length();  
 **char**\* line = **new char**[line\_length + 1];  
 **for** (**int** i = 0; i < line\_length; i++) {  
 line[i] = i % (string\_length + arrow\_length) == 0 ? '|' : ' ';  
 }  
 line[line\_length] = 0;  
  
 **int** end\_position = -1;  
 **while** (!all\_printed) {  
 Edge \*min\_edge = **nullptr**;  
 **int** cur\_min\_edge\_end = -1;  
 all\_printed = **true**;  
  
 for\_each([&] (Vertex& vertex) -> **void** {  
 vertex.for\_each([&] (Edge& edge) -> **void** {  
 **if** (!edge.print\_flag) {  
 all\_printed = **false**;  
 **int** start\_pos = std::min(edge.from->print\_position, edge.connected->print\_position);  
 **int** end\_pos = std::max(edge.from->print\_position, edge.connected->print\_position);  
 **if** (std::min(start\_pos, end\_pos) >= end\_position) {  
 **if** (min\_edge == **nullptr** || end\_pos < cur\_min\_edge\_end) {  
 min\_edge = &edge;  
 cur\_min\_edge\_end = end\_pos;  
 }  
 }  
 }  
 });  
 });  
  
 **if** (min\_edge == **nullptr**) {  
 **break**;  
 }  
 min\_edge->print\_flag = **true**;  
 end\_position = cur\_min\_edge\_end;  
  
 **int** edge\_start = 1 + std::min(min\_edge->from->print\_position, min\_edge->connected->print\_position) \* (string\_length + arrow\_length);  
 **int** edge\_end = std::max(min\_edge->from->print\_position, min\_edge->connected->print\_position) \* (string\_length + arrow\_length);  
 **for** (**int** i = edge\_start; i < edge\_end; i++) {  
 line[i] = '-';  
 }  
  
 std::string edge\_str = edge\_to\_string(min\_edge->data);  
 **int** mid = (edge\_end + edge\_start - (**int**) edge\_str.length()) / 2;  
 **for** (**char** c : edge\_str) {  
 line[mid++] = c;  
 }  
  
 **if** (min\_edge->from->print\_position < min\_edge->connected->print\_position) {  
 line[edge\_end - 1] = '>';  
 } **else** {  
 line[edge\_start] = '<';  
 }  
 }  
  
 std::cout << line << "\n";  
 **delete**[] line;  
 }  
 };  
};  
  
#endif

flow\_network.h

#include <fstream>  
#include <cstring>  
  
#include "graph.h"  
  
#ifndef M\_FLOW\_NETWORK\_H  
#define M\_FLOW\_NETWORK\_H  
  
**class** FlowNetwork {  
**public**:  
 **class** NodeData;  
 **class** EdgeData;  
 **typedef** Graph<NodeData, EdgeData>::Vertex Vertex;  
 **typedef** Graph<NodeData, EdgeData>::Edge Edge;  
  
 **class** NodeData {  
 **public**:  
 std::string name;  
 **bool** visit\_flag = **false**;  
  
 NodeData();  
 NodeData(std::string **const**& name);  
  
 **bool operator**==(NodeData **const**& other);  
 **bool operator**==(std::string **const**& str);  
 };  
  
 **class** EdgeData {  
 **public**:  
 **int** max\_flow = 0;  
 **int** flow = 0;  
 Edge\* opposite = **nullptr**;  
  
 EdgeData();  
 EdgeData(**int** max\_flow);  
 **int** remaining\_flow();  
 };  
  
 Graph<NodeData, EdgeData> graph;  
 Vertex\* source = **nullptr**;  
 Vertex\* target = **nullptr**;  
  
 **void** clear();  
 **void** reset\_vertices();  
 **void** reset\_edges();  
 Vertex& vertex\_by\_name(std::string **const** &name);  
  
 **bool** is\_valid(Vertex\* vertex = **nullptr**, Vertex\* last = **nullptr**);  
 **bool** search\_path(Vertex\* vertex, Array<Edge\*>& path);  
 **void** \_add\_temporary\_edges();  
 **void** \_remove\_temporary\_edges();  
 **void** build\_max\_flow();  
 **int** vertex\_flow(Vertex &vertex);  
  
 **bool** parse\_edge(std::string line);  
 **bool** from\_file(std::string filename);  
 **void** print();  
  
};  
  
  
  
#endif

flow\_network.cpp

#include "flow\_network.h"  
  
FlowNetwork::NodeData::NodeData() {}  
  
FlowNetwork::NodeData::NodeData(std::string **const** &name) : name(name) {}  
  
**bool** FlowNetwork::NodeData::**operator**==(**const** FlowNetwork::NodeData &other) {  
 **return** name == other.name;  
}  
  
**bool** FlowNetwork::NodeData::**operator**==(std::string **const** &str) {  
 **return** name == str;  
}  
  
FlowNetwork::EdgeData::EdgeData() {}  
  
FlowNetwork::EdgeData::EdgeData(**int** max\_flow) : max\_flow(max\_flow) {}  
  
**int** FlowNetwork::EdgeData::remaining\_flow() {  
 **return** max\_flow - flow;  
}  
  
  
  
**void** FlowNetwork::clear() {  
 graph.clear();  
 source = target = **nullptr**;  
}  
  
**void** FlowNetwork::reset\_vertices() {  
 graph.for\_each([] (Vertex& vertex) -> **void** {  
 vertex.data.visit\_flag = **false**;  
 });  
}  
  
**void** FlowNetwork::reset\_edges() {  
 graph.for\_each([] (Vertex& vertex) -> **void** {  
 vertex.for\_each([] (Edge& edge) -> **void** {  
 edge.data.flow = 0;  
 });  
 });  
}  
  
  
  
**bool** FlowNetwork::is\_valid(FlowNetwork::Vertex \*vertex, FlowNetwork::Vertex \*last) {  
 **if** (vertex == **nullptr**) {  
 **if** (source == **nullptr** || target == **nullptr**) {  
 **return false**;  
 }  
 vertex = source;  
 reset\_vertices();  
 }  
 **if** (vertex->data.visit\_flag) {  
 **return false**;  
 }  
 vertex->data.visit\_flag = **true**;  
 **bool** any = **false**;  
 **for** (**auto** it = vertex->edges.begin(); it != vertex->edges.end(); it++) {  
 **if** (it->connected != last && it->data.max\_flow > 0) {  
 any = **true**;  
 **bool** result = is\_valid(it->connected, vertex);  
 **if** (!result) {  
 **return false**;  
 }  
 }  
 }  
 **if** (!any && target != vertex) {  
 **return false**;  
 }  
 vertex->data.visit\_flag = **false**;  
 **return true**;  
}  
  
**bool** FlowNetwork::search\_path(FlowNetwork::Vertex\* vertex, Array<FlowNetwork::Edge\*>& path) {  
 **if** (vertex == target) {  
 **return true**;  
 }  
 **if** (vertex->data.visit\_flag) {  
 **return false**;  
 }  
  
 vertex->data.visit\_flag = **true**;  
 **for** (**auto** it = vertex->edges.begin(); it != vertex->edges.end(); it++) {  
 **if** (it->data.remaining\_flow() > 0) {  
 path.add(&\*it);  
 **if** (search\_path(it->connected, path)) {  
 **return true**;  
 }  
 path.resize(path.length() - 1);  
 }  
 }  
 **return false**;  
}  
  
  
**void** FlowNetwork::\_add\_temporary\_edges() {  
 **auto** edges = graph.get\_all\_edges();  
 **for** (**int** i = 0; i < edges.length(); i++) {  
 edges[i]->data.opposite = &graph.connect(\*edges[i]->connected, \*edges[i]->from, EdgeData(0));  
 edges[i]->data.opposite->data.opposite = edges[i];  
 }  
}  
  
**void** FlowNetwork::\_remove\_temporary\_edges() {  
 **auto** edges = graph.get\_all\_edges();  
 **for** (**int** i = 0; i < edges.length(); i++) {  
 **if** (edges[i]->data.max\_flow == 0) {  
 graph.disconnect(\*edges[i]->from, \*edges[i]->connected);  
 edges[i]->data.opposite = **nullptr**;  
 }  
 }  
}  
  
**void** FlowNetwork::build\_max\_flow() {  
 reset\_edges();  
  
 \_add\_temporary\_edges();  
 **while** (**true**) {  
 Array<Edge\*> path;  
 reset\_vertices();  
 **if** (!search\_path(source, path)) {  
 **break**;  
 }  
  
 **int** min\_remaining\_flow = path[0]->data.remaining\_flow();  
 **for** (**int** i = 1; i < path.length(); i++) {  
 **int** remaining\_flow = path[i]->data.remaining\_flow();  
 **if** (remaining\_flow < min\_remaining\_flow) {  
 min\_remaining\_flow = remaining\_flow;  
 }  
 }  
  
 **for** (**int** i = 0; i < path.length(); i++) {  
 path[i]->data.flow += min\_remaining\_flow;  
 path[i]->data.opposite->data.flow -= min\_remaining\_flow;  
 }  
 }  
 \_remove\_temporary\_edges();  
}  
  
**int** FlowNetwork::vertex\_flow(Vertex &vertex) {  
 **int** flow = 0;  
 graph.for\_each([&] (Vertex& v) -> **void** {  
 v.for\_each([&] (Edge& edge) -> **void** {  
 **if** (edge.connected == &vertex) {  
 flow += edge.data.flow;  
 }  
 **else if** (edge.from == &vertex) {  
 flow -= edge.data.flow;  
 }  
 });  
 });  
 **return** flow;  
}  
  
FlowNetwork::Vertex &FlowNetwork::vertex\_by\_name(std::string **const** &name) {  
 Vertex\* vertex = graph.get\_vertex(name);  
 **if** (vertex == **nullptr**) {  
 **return** graph.add\_vertex(NodeData(name));  
 }  
 **return** \*vertex;  
}  
  
**bool** FlowNetwork::parse\_edge(std::string line) {  
 Array<std::string> tokens;  
  
 **char**\* token = strtok(line.data(), " ");  
 **while** (token != **nullptr**) {  
 tokens.add(std::string(token));  
 token = strtok(**nullptr**, " ");  
 }  
  
 **if** (tokens.length() == 3) {  
 **int** max\_flow = (**int**) std::strtol(tokens[2].data(), **nullptr**, 10);  
 **if** (max\_flow > 0) {  
 Vertex& vertex1 = vertex\_by\_name(tokens[0]);  
 Vertex& vertex2 = vertex\_by\_name(tokens[1]);  
  
 graph.connect(vertex1, vertex2, EdgeData(max\_flow));  
 **if** (vertex1.data.name == "S") {  
 source = &vertex1;  
 }  
 **if** (vertex1.data.name == "T") {  
 target = &vertex1;  
 }  
 **if** (vertex2.data.name == "S") {  
 source = &vertex2;  
 }  
 **if** (vertex2.data.name == "T") {  
 target = &vertex2;  
 }  
 } **else** {  
 Vertex\* vertex1 = graph.get\_vertex(tokens[0]);  
 Vertex\* vertex2 = graph.get\_vertex(tokens[1]);  
 **if** (vertex1 != **nullptr** && vertex2 != **nullptr**) {  
 graph.disconnect(\*vertex1, \*vertex2);  
 }  
 }  
 **return true**;  
 }  
 **return false**;  
}  
  
**bool** FlowNetwork::from\_file(std::string filename) {  
 std::ifstream stream(filename);  
 **if** (stream) {  
 graph.clear();  
 std::string line;  
 **while** (std::getline(stream, line)) {  
 **if** (!line.empty() && !parse\_edge(line)) {  
 std::cout << "invalid formatted edge: " << line << "\n";  
 }  
 }  
 **return true**;  
 } **else** {  
 **return false**;  
 }  
}  
  
**void** FlowNetwork::print() {  
 graph.print(10,  
 [] (NodeData& data) -> std::string {  
 **return** data.name;  
 },  
 [] (EdgeData& data) -> std::string {  
 **char** line[64];  
 sprintf(line, "%2i/%-2i", data.flow, data.max\_flow);  
 **return** line;  
 });  
}