**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по курсовой работе**

**по дисциплине «АиСД»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8302 |  | Смирнов Е.Д. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 3](#_Toc42171368)

[Описание классов и методов 4](#_Toc42171369)

[Описание алгоритма решения 5](#_Toc42171370)

[Обоснование использования структур данных 5](#_Toc42171371)

[Оценка временной сложности 5](#_Toc42171372)

[Описание unit-тестов 6](#_Toc42171373)

[Пример работы 7](#_Toc42171374)

[Код 14](#_Toc42171375)

## 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 Эдмондса - Карпа.

## Описание классов и методов

Реализованы следующие классы:

1. Graph<V, E> – реализует граф на базе матрицы смежности. Граф позволяет присваивать вершинам и ребрам дополнительные данные V и E, которые в транспортной сети используются для ее параметров. Реализует методы добавление вершин, соединения, разъединения вершин ребром, получения ребра между вершинами. Также реализован метод вывода графа в консоль.
2. Graph::Vertex – вершина графа, содержит V и индекс вершины в матрице смежности. Так же содержит указатель на граф. Реализует метод for\_each позволяющий перебирать все присоединенные ребра.
3. Graph::Edge – одностороннее ребро графа, содержит E и две соединяемые упорядоченные вершины. Также содержит флаг active, который показывает, существует ли ребро. Методы для итерации могут перебирать как активные, так и все возможные ребра.
4. Queue<T> - очередь на базе списка List<T> - реализует основной функционал очереди, требуется для поиска в ширину
5. Canvas – реализует поле для рисования символами. Позволяет рисовать как одиночные символы, так и текст и стрелки, что позволяет изобразить граф.
6. FlowNetwork – реализует транспортную сеть, используя класс Graph. Реализует методы валидации, поиска в ширину и нахождения максимального потока алгоритмом Эдмондса - Карпа. Кроме того реализованы методы загрузки и сохранения в файл и красивого вывода в консоль.

Кроме того были использованы слегка модифицированные версии классов Array<T> и List<T>.

## Описание алгоритма решения

Для проверки транспортной сети на корректность была реализована функция validate(), которая может отдавать 4 результата – транспортная сеть корректна, и 3 вида ошибки: отсутствует исток или сток, существует путь не из центра в сток, существует цикл. Для этого перебираются все пути из истока.

Для агоритма Эдмондса – Карпа требуется и был реализован алгоритм поиска в ширину для нахождения кратчайшего пути из истока в сток. Данный алгоритм может в процессе работы алгоритма сталкиваться с несуществующими ребрами с отрицательным потоком. Потому итерация производится по всем ребрам из каждой вершины (даже несуществующим).

Так же был реализован сам алгоритм Эдмондса – Карпа, который пока может, ищет кратчайший путь и пускает по нему поток, равный минимальной оставшейся пропускной способности ребер этого пути.

## Обоснование использования структур данных

Как говорилось выше, граф реализован на основе матрицы смежности. Данная структура была выбрана, т.к. в процессе работы алгоритма требуется изменять ребра, которых не было в оригинальном графе и эта структура данных позволяет делать это за единичное время.

## Оценка временной сложности

Операции над графом работают за константное время.

Поиск в ширину работает за O(|V|^2) на графах с матрицей смежности.

Алгоритм Эдмондса – Карпа совершает O(|V||E|) прирощений, на каждое из которых вызывается поиск в ширину, время которого O(|V|^2), значит общее время на этой структуре данных будет O(|E||V|^3)

Проверка на корректность перебирает все пути, ее время могло бы быть экспоненциальным для произвольного графа, но в данном случае путей без цикла может быть не более, чем вершин, а значит время работы O(|V|).

## Описание unit-тестов

Сначала идут юнит тесты для массива и очереди. Потом проверяется работа абстрактного графа через построение и перебор вершин и ребер.

Потом идут тесты транспортной сети:

1. Тест на построение.
2. Тесты на все возможные результаты валидации.
3. Тесты на поиск в ширину – варианты когда путь есть и когда путь нет, вариант когда кратчайший путь проходит через ребро с нулевой оставшейся пропускной способностью.
4. Тест на построение максимального потока (проверяется что поток через каждую вершину кроме истока и стока суммарно равен 0, а на истоке и стоке равен определенному значению)
5. Тест на сохранение и чтение из файла.

## Пример работы

Далее приведено 7 примеров графов с построенными максимальными потоками.

FLOW NETWORK:

E

/^

/ |

F<--E->F 2/3 ---/ | D<--\

/ | / \

/ | / \

/ | / \

F->H 2/2 | / C->D 2/2

/ | / \

/ | / \

/ | / \

H<--/ \ / C

| \ / ^

| \ / |

| \ / |

| \ / |

| \/ |

H->T 2/5 /\ B->C 2/3

| / \ |

| / \ |

| / \ |

V / A->E 2/2 |

T<--\------D->T 2/5 -----------/ \ /-->B

\ \ /

\ \ /

\ \ /

I->T 3/4 \ A->B 2/2

\ \ /

\ \ /

\ \ /

I<--S->I 3/3 ---\ /--S->A 4/10-->A

\ /

\ /

S

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

A3<A2->A3 6/8 --\

/ \

/ \

A4<-A3->A4 6/6 --/ A2<--\

/ \

/ \

/ \

A4->A5 6/7 A1->A2 6/10

/ \

/ \

/ \

A5<--/ A1

| ^

| |

| |

| |

| |

A5->A6 6/7 S->A1 6/9

| |

| |

| |

V |

A6 S

\

\

\

A6->A7 6/8

\

\

\

\--A7 /--A8->T 6/10->T

\ /

\ /

\--A7->A8 6/11-A8

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

A3<A2->A3 7/8 --\

/ \

/ \

A4<-A3->A4 6/6 --/ A2<--\

/ / / \

/ / / \

/ / / \

A4->A5 6/7 / / A1->A2 8/10

/ / / \

/ / / \

/ / / \

A5<--/ A3->A6 1/1 / A1

| / / /^

| / / / |

| / / / |

| / / / |

| / / / |

A5->A6 6/7 / A2->A7 1/1 /S->A1 9/9

| / / / |

| / / / |

| / / / |

V / / / |

A6<--------/ / A1->A8 1/1 S

\ / /

\ / /

\ / /

A6->A7 7/8 / /

\ / /

\ / /

\ / /

\--A7<----------/ /--A8->T 9/10->T

\ / /

\ / /

\--A7->A8 8/11-A8<-------/

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

E<-D->E 0/1 ---\

/^ \

/ | \

F<----------\ / | D<--\

| \ / | /^ \

| \/ | / | \

| /\ | / | \

| / \ | / | C->D 0/1

| / \ | / | \

| / \ | / | \

| E->T 1/1 \| / | \

H<---------A->H 1/1 /----------\ / | C

\ | / \ / | /^

\ | / \ / | / |

\ | / \ / | / |

\ | / \ / | / |

\ | / \/ | / |

H->T 1/1 1/1 D->T 1/1 A->D 1/3 /B->C 0/1

\ | / / \ | / |

\ |/ / \ | / /|

\ | / \ | / / |

\ | / A->E 1/3 | / / |

G<---------\|----A->G 1/1 -----/----------\ |/ /-->B

\ \ / \ / / /

\ | / \ /|A->C 2/2

\ | / \ / | / /

G->T 1/1 | / \/ A->B 1/1

\ | / /\ | / /

\ | / / \ | / /

\ V / / \|/ /

\-->T<---------C->T 2/3 1/3 ->A 8/10-->A---/

/

/

S

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

E<-D->E 0/2 2 -\---\

/^ \ \ \

/ | \ \ \

F<--E->F 2/2 ---/ | \ D<--\

| / | \ \

| / | \ \

| / | \ \

| / | \ C->D 0/1

| / | \ \

| / | \ \

| / | \ \

G<----E->G 2/2 E->H 2/2 \ B->E 2/2 ----->C

\ | / \ \ ^

\ | / \ / \ |

\ | / \ / \ |

\ | / \ / \ |

\ | / \ / \ |

G->T 2/2 2/2 \ / B->C 0/1

\ | / \ / \ |

\ |/ \ / \ |

\ | \ / \ |

\/| A->E 2/2 \|

H<--------/\| S->C 2/2 S->B 2/2 -/-->B

\ \ / \ /

\ | / \ /

\ | / \ /

H->T 2/2 | / \ A->B 0/1

\ | / \ /

\ | / \ /

\ V / \ /

\-->T /--S->A 2/2 -->A

/

/

S

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

E<-D->E 2/4 ---\

/ \

/ \

F<--E->F 2/3 ---/ D<--\

/ \

/ \

/ \

F->G 2/3 C->D 2/2

/ \

/ \

/ \

G<--/ C

| ^

| |

| |

| |

| |

G->H 2/3 B->C 2/3

| |

| |

| |

V |

H<---------------A->H 3/3 ----------------\ /-->B

\ \ /

\ \ /

\ \ /

H->T 5/5 \ A->B 2/2

\ \ /

\ \ /

\ \ /

\-->T /--S->A 5/10-->A

/

/

S

validation result: RESULT\_OK: network is valid

FLOW NETWORK:

B5

/^

/ |

B6 / | B4

^ / | /^

| / | / |

| / | / |

| / | / |

| / | / |

| / | / |

| B5->T 4/4 | / |

B7<--------\ | / | / | /-------B3

\ \| / | / | / /

\ | / | / |/ /

\ |\ / | / / /

\ | \ / | / / /

\ | \/ | / / /

B7->T 2/2 3/3 S->B5 B4->T 4/5 / /

\ | / \ | / / /

\ |/ \ | / / /

\ | \ | / / /

\ | \ |/ / /

B8<----S->B8 1/8 ->B7 2/7 / S->B3 3/3 ->B2 2/2 ---B2

\ \ S->B6 3/6 /| / / /

\ | \ / | S->B4 4/4 / /

\ | \ / | / / /

B8->T 1/1 | \ / | / / /

\ | \/ | / / /

\ | /\ | / / /

\ V / \ | / / /

\-->T<---------B3->T 3/6 2/7 >B1 1/1 -B1---/

\ | /

\|/

S

validation result: RESULT\_OK: network is valid

## Код

main.cpp

#include "graph.h"  
#include "flow\_network.h"  
#include "canvas.h"  
  
**void** await\_user() {  
 std::cout << "press enter to continue...";  
 getchar();  
}  
  
**int** main() {  
 FlowNetwork network;  
 network.read("../../coursework/data.txt");  
  
 **while** (**true**) {  
 std::cout << "FLOW NETWORK:\n";  
 network.print();  
  
 **bool** is\_validated = **false**;  
 **switch** (network.validate()) {  
 **case** FlowNetwork::*RESULT\_OK*:  
 is\_validated = **true**;  
 std::cout << "validation result: RESULT\_OK: network is valid\n";  
 **break**;  
 **case** FlowNetwork::*RESULT\_CYCLE*:  
 std::cout << "validation result: RESULT\_CYCLE: graph contains cycles\n";  
 **break**;  
 **case** FlowNetwork::*RESULT\_INVALID\_PATH*:  
 std::cout << "validation result: INVALID\_PATH: some path does not end at target\n";  
 **break**;  
 **case** FlowNetwork::*RESULT\_NO\_TERMINATIONS*:  
 std::cout << "validation result: RESULT\_NO\_TERMINATIONS: no source or target\n";  
 **break**;  
 }  
  
 std::cout << "\n\n";  
 std::cout << "options:" << "\n";  
 std::cout << "type 1 to load network from file" << "\n";  
 std::cout << "type 2 to build max flow through network" << "\n";  
 std::cout << "type 3 to add vertices or edges" << "\n";  
 std::cout << "type 4 to disconnect vertices" << "\n";  
 std::cout << "type 5 to clear network" << "\n";  
 std::cout << "type 6 to reset flow" << "\n";  
 std::cout << "type 7 to save network" << "\n";  
 std::cout << "type other number to exit" << "\n";  
 std::cout << "input command: ";  
  
 **int** command;  
 std::cin >> command;  
 getchar(); // consume \n  
  
 **switch** (command) {  
 **case** 1: {  
 std::string name;  
 std::cout << "input filename: ";  
 std::cin >> name;  
 getchar();  
 std::cout << (network.read(name) ? "success" : "fail") << "\n";  
 await\_user();  
 **break**;  
 }  
 **case** 2: {  
 **if** (is\_validated) {  
 network.build\_max\_flow();  
 } **else** {  
 std::cout << "flow network is not valid\n";  
 await\_user();  
 }  
 **break**;  
 }  
 **case** 3: {  
 std::string line;  
 std::cout << "input edge (START\_NAME END\_NAME MAX\_FLOW, S = start, T = target): ";  
 std::getline(std::cin, line);  
 **if** (!network.add\_edge\_from\_string(line)) {  
 await\_user();  
 }  
 **break**;  
 }  
 **case** 4: {  
 std::string start, end;  
 std::cout << "input edge to remove (START\_NAME END\_NAME): ";  
 std::cin >> start;  
 std::cin >> end;  
 getchar(); // consume \n  
 FlowNetwork::Vertex\* v1 = network.mGraph.get\_vertex(start);  
 FlowNetwork::Vertex\* v2 = network.mGraph.get\_vertex(end);  
 **if** (v1 != **nullptr** && v2 != **nullptr**) {  
 FlowNetwork::Edge& edge = network.mGraph.get\_edge(\*v1, \*v2);  
 **if** (edge.active) {  
 network.mGraph.disconnect(\*v1, \*v2);  
 } **else** {  
 std::cout << "no such edge exists\n";  
 await\_user();  
 }  
 } **else** {  
 std::cout << "no such vertices\n";  
 }  
  
 **break**;  
 }  
 **case** 5: {  
 network.clear();  
 **break**;  
 }  
 **case** 6: {  
 network.reset\_edges();  
 **break**;  
 }  
 **case** 7: {  
 std::string name;  
 std::cout << "input filename: ";  
 std::cin >> name;  
 getchar();  
 std::cout << (network.save(name) ? "success" : "fail") << "\n";  
 await\_user();  
 **break**;  
 **break**;  
 }  
 **default**:  
 **return** 0;  
 }  
  
 std::cout << "\n\n\n\n----------------------------------------------\n";  
  
 }  
  
}

test.cpp

#include <string>  
  
#include "gtest/gtest.h"  
#include "rb\_map.h"  
#include "array.h"  
#include "queue.h"  
#include "graph.h"  
#include "flow\_network.h"  
  
  
// array tests  
TEST (array, massive\_add\_and\_check) {  
 array<std::string> arr;  
 **for** (**int** i = 0; i < 1000000; i++) {  
 arr.add(i % 2 ? "abc" : "edf");  
 }  
 ASSERT\_EQ(arr.length(), 1000000);  
 **for** (**int** i = 0; i < 1000000; i++) {  
 ASSERT\_EQ(arr[i], i % 2 ? "abc" : "edf");  
 }  
 arr.clear();  
 ASSERT\_EQ(arr.length(), 0);  
}  
  
// keep hardest test for rb\_map  
TEST (rb\_map, massive\_random\_load) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 1000000; i++) {  
 map[rand() % 1000000] = rand();  
 }  
 ASSERT\_EQ(map.*length*(), map.*tree\_size*());  
  
 **int** remaining\_length = map.*length*();  
 **for** (**int** i = 0; i < 1000000; i++) {  
 **if** (map.*remove*(rand() % 1000000)) {  
 remaining\_length--;  
 }  
 }  
  
 ASSERT\_EQ(map.*length*(), map.*tree\_size*());  
 ASSERT\_EQ(map.*length*(), remaining\_length);  
}  
  
// queue tests  
TEST (queue, queue\_and\_dequeue) {  
 Queue<std::string> queue;  
 queue.enqueue("a");  
 queue.enqueue("b");  
 ASSERT\_EQ(queue.size(), 2);  
 ASSERT\_EQ(queue.dequeue(), "a");  
 ASSERT\_EQ(queue.dequeue(), "b");  
}  
  
// 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);  
 graph.connect(\*v2, v1, i \* j);  
 }  
 }  
  
 ASSERT\_EQ(graph.size(), 10);  
 graph.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.add\_edge\_from\_string("S T 10"));  
 ASSERT\_FALSE(network.add\_edge\_from\_string("A 10"));  
 ASSERT\_EQ(network.mGraph.size(), 2);  
 FlowNetwork::Vertex\* v1 = network.mGraph.get\_vertex("S");  
 FlowNetwork::Vertex\* v2 = network.mGraph.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.mGraph.get\_edge(\*v1, \*v2);;  
 FlowNetwork::Edge& edge2 = network.mGraph.get\_edge(\*v2, \*v1);  
 ASSERT\_EQ(edge1.active, **true**);  
 ASSERT\_EQ(edge2.active, **false**);  
 ASSERT\_EQ(edge1.data.max\_flow, 10);  
}  
  
TEST(flow\_network, validate\_ok) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S B 2"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 3"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("B T 4"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_OK*);  
}  
  
TEST(flow\_network, validate\_no\_terminators) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S B 2"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 3"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_NO\_TERMINATIONS*);  
  
 network.clear();  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A T 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("T B 2"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 3"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_NO\_TERMINATIONS*);  
}  
  
TEST(flow\_network, validate\_cycle) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("T S 2"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A T 3"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_CYCLE*);  
}  
  
TEST(flow\_network, validate\_invalid\_path) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("T B 3"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 2"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_INVALID\_PATH*);  
}  
  
TEST(flow\_network, bfs\_success) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("B C 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("B T 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("C T 1"));  
 ASSERT\_EQ(network.bfs().length(), 4);  
  
 FlowNetwork::Edge& edge = network.mGraph.get\_edge(\*network.mGraph.get\_vertex("B"), \*network.mGraph.get\_vertex("T"));  
 ASSERT\_EQ(edge.active, **true**);  
 edge.data.flow = edge.data.max\_flow;  
 ASSERT\_EQ(network.bfs().length(), 5);  
}  
  
TEST(flow\_network, bfs\_fail) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("C T 1"));  
 ASSERT\_EQ(network.bfs().length(), 0);  
}  
  
TEST(flow\_network, max\_flow\_build) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("B T 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S T 2"));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_OK*);  
 network.build\_max\_flow();  
 ASSERT\_EQ(network.get\_flow\_at\_source(), 3);  
 ASSERT\_EQ(network.get\_flow\_at\_target(), 3);  
 network.mGraph.for\_each([&] (FlowNetwork::Vertex& vertex) -> **void** {  
 **if** (&vertex != network.source && &vertex != network.target) {  
 ASSERT\_EQ(network.get\_flow\_at(vertex), 0);  
 }  
 });  
}  
  
  
TEST(flow\_network, save\_and\_read) {  
 FlowNetwork network;  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S A 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("A B 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("B T 1"));  
 ASSERT\_TRUE(network.add\_edge\_from\_string("S T 2"));  
  
 std::string filename = "unit\_test.tmp";  
 **if** (network.save(filename)) {  
 network.clear();  
 ASSERT\_TRUE(network.read(filename));  
 ASSERT\_EQ(network.validate(), FlowNetwork::*RESULT\_OK*);  
 network.build\_max\_flow();  
 ASSERT\_EQ(network.get\_flow\_at\_source(), 3);  
 ASSERT\_EQ(network.get\_flow\_at\_target(), 3);  
 } **else** {  
 std::cout << "failed to test save & read: failed to write to file";  
 }  
}

list.h

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

array.h

#include <iostream>  
  
  
#ifndef M\_ARRAY\_H  
#define M\_ARRAY\_H  
  
**template** <**typename** T>  
**class** array {  
 T\* allocated\_memory = **nullptr**;  
 **int** allocated\_memory\_size = 0;  
  
 **int** size = 0;  
  
 **void** ensure\_size(**int** size) {  
 **if** (allocated\_memory\_size < size) {  
 **int** old\_size = allocated\_memory\_size;  
 **while** (allocated\_memory\_size < size) {  
 **if** (allocated\_memory\_size < 32) {  
 allocated\_memory\_size += 16;  
 } **else** {  
 allocated\_memory\_size \*= 2;  
 }  
 }  
  
 **if** (allocated\_memory != **nullptr**) {  
 T\* new\_memory = **new** T[allocated\_memory\_size];  
 **for** (**int** i = 0; i < old\_size; i++) {  
 new\_memory[i] = allocated\_memory[i];  
 }  
 **delete**[] (allocated\_memory);  
 allocated\_memory = new\_memory;  
 } **else** {  
 allocated\_memory = **new** T[allocated\_memory\_size];  
 }  
 }  
 }  
  
**public**:  
 array() = **default**;  
  
 array(**int** size) {  
 resize(size);  
 }  
  
 array(array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 allocated\_memory[i] = other.allocated\_memory[i];  
 }  
 }  
  
 array(array<T>&& other) **noexcept** {  
 allocated\_memory = other.allocated\_memory;  
 size = other.size;  
 allocated\_memory\_size = other.allocated\_memory\_size;  
 other.allocated\_memory = **nullptr**;  
 other.size = 0;  
 other.allocated\_memory\_size = 0;  
 }  
  
 array<T>& **operator**= (array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 allocated\_memory[i] = other.allocated\_memory[i];  
 }  
 **return** \***this**;  
 }  
  
 array<T>& **operator**= (array<T>&& other) **noexcept** {  
 allocated\_memory = other.allocated\_memory;  
 size = other.size;  
 allocated\_memory\_size = other.allocated\_memory\_size;  
 other.allocated\_memory = **nullptr**;  
 other.size = 0;  
 other.allocated\_memory\_size = 0;  
 **return** \***this**;  
 }  
  
 **int** length() **const** {  
 **return** size;  
 }  
  
 T& **operator**[] (**int** index) {  
 **return** allocated\_memory[index];  
 }  
  
 **void** resize(**int** new\_size) {  
 **if** (new\_size == 0) {  
 clear();  
 } **else** {  
 ensure\_size(new\_size);  
 size = new\_size;  
 }  
 }  
  
 T& add(T **const**& elem) {  
 ensure\_size(size + 1);  
 **return** allocated\_memory[size++] = elem;  
 }  
  
 **void** add\_all(array<T> **const**& arr) {  
 **for** (**int** i = 0; i < arr.size; i++) {  
 add(arr[i]);  
 }  
 }  
  
 **void** clear() {  
 **if** (allocated\_memory != **nullptr**) {  
 **delete**[] (allocated\_memory);  
 allocated\_memory = **nullptr**;  
 }  
 size = allocated\_memory\_size = 0;  
 }  
  
 ~array() {  
 clear();  
 }  
};  
  
#endif

canvas.h

#include <iostream>  
#include <string.h>  
  
#ifndef M\_CANVAS\_H  
#define M\_CANVAS\_H  
  
**class** Canvas {  
**public**:  
 **struct** Point {  
 **public**:  
 **float** x, y;  
 };  
  
 **int** width;  
 **int** height;  
 **char**\* canvas;  
  
 Canvas(**int** width, **int** height);  
 **void** print();  
 **void** set\_abs(**int** x, **int** y, **char** c, **bool** keepArrow = **false**);  
 **void** set(**float** x, **float** y, **char** c, **bool** keepArrow = **false**);  
 Point line\_abs(**int** ix1, **int** iy1, **int** ix2, **int** iy2, **bool** keepArrow = **false**);  
 Point line(**float** x1, **float** y1, **float** x2, **float** y2, **bool** keepArrow = **false**);  
 Point arrow(**float** x1, **float** y1, **float** x2, **float** y2, **bool** offset = **false**);  
 **void** text(**float** fx, **float** fy, std::string text, **bool** center = **true**, **bool** vertical = **false**);  
 ~Canvas();  
};  
  
#endif

canvas.cpp

#include <cmath>  
#include "canvas.h"  
  
Canvas::Canvas(**int** width, **int** height) : width(width), height(height) {  
 canvas = **new char**[height \* (width + 1) + 1];  
 memset(canvas, ' ', (size\_t) (height \* (width + 1) + 1));  
 **for** (**int** i = 0; i < height; i++) {  
 canvas[i \* (width + 1) + width] = '\n';  
 }  
 canvas[height \* (width + 1)] = '\0';  
}  
  
**void** Canvas::print() {  
 std::cout << canvas;  
}  
  
**void** Canvas::set\_abs(**int** x, **int** y, **char** c, **bool** keepArrow) {  
 **if** (x < 0 || y < 0 || x > width || y > width) {  
 **return**;  
 }  
 **if** (keepArrow) {  
 **char** check = canvas[y \* (width + 1) + x];  
 **if** (check == 'V' || check == '^' || check == '>' || check == '<') {  
 **return**;  
 }  
 }  
 canvas[y \* (width + 1) + x] = c;  
}  
  
**void** Canvas::set(**float** x, **float** y, **char** c, **bool** keepArrow) {  
 set\_abs((**int**) lroundf(x \* (width - 1)), (**int**) lroundf(y \* (height - 1)), c, keepArrow);  
}  
  
Canvas::Point Canvas::line\_abs(**int** ix1, **int** iy1, **int** ix2, **int** iy2, **bool** keepArrow) {  
 **int** ixx1 = ix1;  
 **int** iyy1 = iy1;  
 **while** (ix1 < ix2 - abs(iy1 - iy2)) {  
 set\_abs(ix1++, iy1, '-', keepArrow);  
 }  
 **while** (ix1 > ix2 + abs(iy1 - iy2)) {  
 set\_abs(ix1--, iy1, '-', keepArrow);  
 }  
 **while** (iy1 < iy2 - abs(ix1 - ix2)) {  
 set\_abs(ix1, iy1++, '|', keepArrow);  
 }  
 **while** (iy1 > iy2 + abs(ix1 - ix2)) {  
 set\_abs(ix1, iy1--, '|', keepArrow);  
 }  
  
 **int** lx = ix1, ly = iy1;  
 **while** (iy1 != iy2) {  
 **if** (iy1 < iy2) {  
 **if** (ix1 < ix2) {  
 set\_abs(ix1++, iy1++, '\\', keepArrow);  
 } **else if** (ix1 == ix2) {  
 set\_abs(ix1, iy1++, '-', keepArrow);  
 } **else** {  
 set\_abs(ix1--, iy1++, '/', keepArrow);  
 }  
 } **else** {  
 **if** (ix1 < ix2) {  
 set\_abs(ix1++, iy1--, '/', keepArrow);  
 } **else if** (ix1 == ix2) {  
 set\_abs(ix1, iy1--, '-', keepArrow);  
 } **else** {  
 set\_abs(ix1--, iy1--, '\\', keepArrow);  
 }  
 }  
 }  
  
 **if** (pow(lx - ix2, 2) + pow(ly - iy2, 2) > pow(lx - ixx1, 2) + pow(ly - iyy1, 2)) {  
 **return** {(lx \* 0.5f + ix2 \* 0.5f) / (width - 1), (ly \* 0.5f + iy2 \* 0.5f) / (height - 1)};  
 } **else** {  
 **return** {(lx \* 0.5f + ixx1 \* 0.5f) / (width - 1), (ly \* 0.5f + iyy1 \* 0.5f) / (height - 1)};  
 }  
}  
  
Canvas::Point Canvas::line(**float** x1, **float** y1, **float** x2, **float** y2,**bool** keepArrow) {  
 **int** ix1 = (**int**) lroundf(x1 \* (width - 1));  
 **int** ix2 = (**int**) lroundf(x2 \* (width - 1));  
 **int** iy1 = (**int**) lroundf(y1 \* (height - 1));  
 **int** iy2 = (**int**) lroundf(y2 \* (height - 1));  
 **return** line\_abs(ix1, iy1, ix2, iy2, keepArrow);  
}  
  
  
Canvas::Point Canvas::arrow(**float** x1, **float** y1, **float** x2, **float** y2, **bool** offset) {  
 **int** ix1 = (**int**) lroundf(x1 \* (width - 1));  
 **int** ix2 = (**int**) lroundf(x2 \* (width - 1));  
 **int** iy1 = (**int**) lroundf(y1 \* (height - 1));  
 **int** iy2 = (**int**) lroundf(y2 \* (height - 1));  
  
 **if** (offset) {  
 **if** (ix1 + 1 < ix2) {  
 ix1++;  
 ix2--;  
 } **else if** (ix1 - 1 > ix2) {  
 ix1--;  
 ix2++;  
 }  
 **if** (iy1 + 1 < iy2) {  
 iy1++;  
 iy2--;  
 } **else if** (iy1 - 1 > iy2) {  
 iy1--;  
 iy2++;  
 }  
 }  
  
 **auto** result = line\_abs(ix2, iy2, ix1, iy1, **true**);  
  
 **if** (abs(ix1 - ix2) < abs(iy1 - iy2)) {  
 **if** (iy1 > iy2) {  
 set\_abs(ix2, iy2 + 1, '^');  
 } **else** {  
 set\_abs(ix2, iy2 - 1, 'V');  
 }  
 } **else if** (ix2 > ix1) {  
 set\_abs(ix2 - 1, iy2, '>');  
 } **else** {  
 set\_abs(ix2 + 1, iy2, '<');  
 }  
  
 **return** result;  
}  
  
  
**void** Canvas::text(**float** fx, **float** fy, std::string text, **bool** center, **bool** vertical) {  
 **int** x = (**int**) lroundf(fx \* (width - 1));  
 **int** y = (**int**) lroundf(fy \* (height - 1));  
 **if** (center) {  
 **if** (vertical) {  
 y -= text.length() / 2;  
 } **else** {  
 x -= text.length() / 2;  
 }  
 }  
 **for** (**int** i = 0; i < text.length(); i++) {  
 set\_abs(x + (vertical ? 0 : i), y + (vertical ? i : 0), text[i]);  
 }  
}  
  
Canvas::~Canvas() {  
 **delete**[] (canvas);  
}

queue.h

#include "list.h"  
  
  
#ifndef M\_QUEUE\_H  
#define M\_QUEUE\_H  
  
**template** <**typename** T>  
**class** Queue {  
 list<T> mList;  
**public**:  
 **int** size() **const** {  
 **return** mList.get\_length();  
 }  
  
 **bool** empty() **const** {  
 **return** mList.get\_length() == 0;  
 }  
  
 T dequeue() {  
 **auto** node = mList.\_erase(mList.begin());  
 T result = std::move(node->value);  
 **delete**(node);  
 **return** result;  
 }  
  
 **void** enqueue(T **const** &value) {  
 mList.add(value);  
 }  
  
 **void** clear() {  
 mList.clear();  
 }  
};  
  
#endif

graph.h

#include <iostream>  
#include <functional>  
#include <string>  
#include <string.h>  
#include <sstream>  
#include <math.h>  
  
#include "list.h"  
#include "array.h"  
#include "canvas.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;  
  
 **bool** active = **false**;  
 Vertex\* connected = **nullptr**;  
 Vertex\* from = **nullptr**;  
 E data;  
  
 **private**:  
 **float** printX = 0, printY = 0;  
  
 **public**:  
  
 Edge() = **default**;  
 Edge(Vertex\* from, Vertex\* connected, E **const**& data) : active(**true**), from(from), connected(connected), data(data) {}  
  
 Edge& opposite() {  
 **return** from->graph->get\_edge(\*connected, \*from);  
 }  
  
 E& **operator**\*() { **return** data; }  
 E **const**& **operator**\*() **const** { **return** data; }  
 E\* **operator**->() { **return** &data; }  
 E **const**\* **operator**->() **const** { **return** &data; }  
 };  
  
 **class** Vertex {  
 **public**:  
 **friend class** Graph;  
  
 Graph\* graph = **nullptr**;  
 **int** index;  
 V data;  
  
 **private**:  
 std::string print\_str = "";  
 **float** printX = 0, printY = 0;  
  
 **public**:  
 Vertex() = **default**;  
 Vertex(Graph\* graph, **int** index, V **const**& data) : graph(graph), index(index), data(data) {};  
  
 V& **operator**\*() { **return** data; }  
 V **const**& **operator**\*() **const** { **return** data; }  
 V\* **operator**->() { **return** &data; }  
 V **const**\* **operator**->() **const** { **return** &data; }  
  
 **void** for\_each(std::function<**void**(Edge&)> func, **bool** iterateInactive = **false**) {  
 **for** (**int** i = 0; i < graph->vertices.length(); i++) {  
 Edge& edge = graph->matrix[index][i];  
 **if** (iterateInactive || edge.active) {  
 func(edge);  
 }  
 }  
 }  
 };  
  
**private**:  
 array<Vertex> vertices;  
 array<array<Edge>> matrix;  
  
**public**:  
 array<Vertex>& get\_vertices() {  
 **return** vertices;  
 }  
  
 array<Vertex> **const**& get\_vertices() **const** {  
 **return** vertices;  
 }  
  
 **int** size() **const** {  
 **return** vertices.length();  
 }  
  
 **void** connect(Vertex& v1, Vertex& v2, E **const**& data) {  
 matrix[v1.index][v2.index].active = **true**;  
 matrix[v1.index][v2.index].data = data;  
 }  
  
 **void** disconnect(Vertex& v1, Vertex& v2) {  
 matrix[v1.index][v2.index].active = **false**;  
 }  
  
 Edge& get\_edge(**int** index1, **int** index2) {  
 **return** matrix[index1][index2];  
 }  
  
 Edge& get\_edge(Vertex& v1, Vertex& v2) {  
 **return** get\_edge(v1.index, v2.index);  
 }  
  
 **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) {  
 Vertex& vertex = vertices.add(Vertex(**this**, vertices.length(), data));  
 matrix.resize(vertices.length());  
 **for** (**int** i = 0; i < matrix.length(); i++) {  
 matrix[i].resize(vertices.length());  
 **for** (**int** j = i == matrix.length() - 1 ? 0 : vertices.length() - 1; j < vertices.length(); j++) {  
 Edge &edge = matrix[i][j];  
 edge.from = &vertices[i];  
 edge.connected = &vertices[j];  
 }  
 }  
 **return** vertex;  
 }  
  
 **void** for\_each(std::function<**void**(Vertex&)> func) {  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 func(vertices[i]);  
 }  
 }  
  
 **void** for\_each(std::function<**void**(Edge&)> func, **bool** iterateInactive = **false**) {  
 **for** (**int** i = 0; i < vertices.length(); i++) {  
 **for** (**int** j = 0; j < vertices.length(); j++) {  
 Edge& edge = matrix[i][j];  
 **if** (iterateInactive || edge.active) {  
 func(edge);  
 }  
 }  
 }  
 }  
  
 **void** clear() {  
 vertices.clear();  
 matrix.clear();  
 }  
  
 **void** print(**int** arrow\_length, std::function<std::string(Vertex&)> vertex\_to\_string, std::function<std::string(Edge&)> edge\_to\_string) {  
 **if** (get\_vertices().*length*() == 0) {  
 std::cout << "graph is empty\n";  
 **return**;  
 }  
  
 Canvas canvas(80, 40);  
 for\_each([&] (Vertex& vertex) -> **void** {  
 vertex.print\_str = vertex\_to\_string(vertex);  
 vertex.printX = 0.5f + std::sin(vertex.index \* 6.282f / vertices.length()) \* 0.425f;  
 vertex.printY = 0.5f + std::cos(vertex.index \* 6.282f / vertices.length()) \* 0.425f;  
 });  
  
 for\_each([&] (Edge& edge) -> **void** {  
 **auto** point = canvas.arrow(edge.from->printX, edge.from->printY, edge.connected->printX, edge.connected->printY);  
 edge.printX = point.x;  
 edge.printY = point.y;  
 });  
  
  
 for\_each([&] (Vertex& vertex) -> **void** {  
 canvas.text(vertex.printX, vertex.printY, vertex.print\_str);  
 });  
  
 for\_each([&] (Edge& edge) -> **void** {  
 canvas.text(edge.printX, edge.printY, edge\_to\_string(edge));  
 });  
  
 canvas.print();  
 };  
};  
  
#endif

flow\_network.h

#include <fstream>  
#include <cstring>  
  
#include "graph.h"  
#include "queue.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** visited = **false**;  
 Vertex\* path\_next = **nullptr**;  
 Vertex\* path\_last = **nullptr**;  
  
 NodeData();  
 NodeData(std::string **const**& name);  
  
 **void** reset();  
 **bool operator**==(NodeData **const**& other);  
 **bool operator**==(std::string **const**& str);  
 };  
  
 **class** EdgeData {  
 **public**:  
 **int** max\_flow = 0;  
 **int** flow = 0;  
  
 EdgeData();  
 EdgeData(**int** max\_flow);  
 **void** reset();  
 **int** remaining\_flow();  
 };  
  
 **enum** ValidationResult : **int** {  
 *RESULT\_OK* = 0,  
 *RESULT\_CYCLE* = 1,  
 *RESULT\_INVALID\_PATH* = 2,  
 *RESULT\_NO\_TERMINATIONS* = 3  
 };  
  
  
 Graph<NodeData, EdgeData> mGraph;  
 Vertex\* source = **nullptr**;  
 Vertex\* target = **nullptr**;  
  
 **void** clear();  
 **void** reset\_vertices();  
 **void** reset\_edges();  
 Vertex& get\_or\_add\_vertex(std::string **const**& name);  
  
 // check if graph has no cycles and all paths from source are leading to target  
 // this function will reset vertices  
 ValidationResult validate(Vertex\* vertex = **nullptr**, Vertex\* last = **nullptr**);  
 array<Vertex\*> bfs();  
 **void** build\_max\_flow();  
 **int** get\_flow\_at(Vertex& vertex);  
 **int** get\_flow\_at\_source();  
 **int** get\_flow\_at\_target();  
  
 **bool** add\_edge\_from\_string(std::string line);  
 **bool** read(std::string filename);  
 **bool** save(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) {}  
  
**void** FlowNetwork::NodeData::reset() {  
 visited = **false**;  
 path\_next = path\_last = **nullptr**;  
}  
  
**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) {}  
  
**void** FlowNetwork::EdgeData::reset() {  
 flow = 0;  
}  
  
**int** FlowNetwork::EdgeData::remaining\_flow() {  
 **return** max\_flow - flow;  
}  
  
  
  
**void** FlowNetwork::clear() {  
 mGraph.clear();  
 source = target = **nullptr**;  
}  
  
**void** FlowNetwork::reset\_vertices() {  
 mGraph.for\_each([] (Vertex& vertex) -> **void** {  
 vertex.data.reset();  
 });  
}  
  
**void** FlowNetwork::reset\_edges() {  
 mGraph.for\_each([] (Edge& edge) -> **void** {  
 edge.data.reset();  
 }, **true**);  
}  
  
FlowNetwork::ValidationResult FlowNetwork::validate(FlowNetwork::Vertex \*vertex, FlowNetwork::Vertex \*last) {  
 **if** (vertex == **nullptr**) {  
 **if** (source == **nullptr** || target == **nullptr**) {  
 **return** *RESULT\_NO\_TERMINATIONS*;  
 }  
 vertex = source;  
 reset\_vertices();  
 }  
 **if** (vertex->data.visited) {  
 **return** *RESULT\_CYCLE*;  
 }  
 vertex->data.visited = **true**;  
 **bool** any = **false**;  
  
 **for** (**int** i = 0; i < mGraph.size(); i++) {  
 Edge& edge = mGraph.get\_edge(vertex->index, i);  
 **if** (edge.active && edge.connected != last) {  
 any = **true**;  
 ValidationResult result = validate(edge.connected, vertex);  
 **if** (result != *RESULT\_OK*) {  
 **return** result;  
 }  
 }  
 }  
 **if** (!any && target != vertex) {  
 **return** *RESULT\_INVALID\_PATH*;  
 }  
 vertex->data.visited = **false**;  
 **return** *RESULT\_OK*;  
}  
  
array<FlowNetwork::Vertex \*> FlowNetwork::bfs() {  
 reset\_vertices();  
  
 Queue<Vertex\*> queue;  
 queue.enqueue(source);  
 source->data.visited = **true**;  
  
 **while** (!queue.empty()) {  
 Vertex\* vertex = queue.dequeue();  
 **if** (vertex == target) {  
 array<Vertex\*> path;  
 **while**(vertex != **nullptr**) {  
 path.add(vertex);  
 vertex = vertex->data.path\_last;  
 }  
 **return** path;  
 }  
  
 vertex->for\_each([&] (Edge& edge) -> **void** {  
 **if** (!edge.connected->data.visited && edge.data.remaining\_flow() > 0) {  
 edge.connected->data.visited = **true**;  
 edge.connected->data.path\_last = vertex;  
 queue.enqueue(edge.connected);  
 }  
 }, **true**);  
 }  
  
 **return** array<Vertex\*>();  
}  
  
**void** FlowNetwork::build\_max\_flow() {  
 reset\_edges();  
  
 **while** (**true**) {  
 array<Vertex\*> path = bfs();  
 **if** (path.length() == 0) {  
 **break**;  
 }  
  
 array<Edge\*> path\_edges;  
 **for** (**int** i = 0; i < path.length() - 1; i++) {  
 path\_edges.add(&mGraph.get\_edge(\*path[i + 1], \*path[i]));  
 }  
  
 **int** min\_remaining\_flow = path\_edges[0]->data.remaining\_flow();  
 **for** (**int** i = 1; i < path\_edges.length(); i++) {  
 **int** remaining\_flow = path\_edges[i]->data.remaining\_flow();  
 **if** (remaining\_flow < min\_remaining\_flow) {  
 min\_remaining\_flow = remaining\_flow;  
 }  
 }  
  
 **for** (**int** i = 0; i < path\_edges.length(); i++) {  
 path\_edges[i]->data.flow += min\_remaining\_flow;  
 path\_edges[i]->opposite().data.flow -= min\_remaining\_flow;  
 }  
 }  
}  
  
**int** FlowNetwork::get\_flow\_at(Vertex& vertex) {  
 **int** flow = 0;  
 mGraph.for\_each([&] (Edge& edge) -> **void** {  
 **if** (edge.connected == &vertex) {  
 flow += edge.data.flow;  
 }  
 **if** (edge.from == &vertex) {  
 flow -= edge.data.flow;  
 }  
 });  
 **return** flow;  
}  
  
**int** FlowNetwork::get\_flow\_at\_source() {  
 **return** source != **nullptr** ? -get\_flow\_at(\*source) : 0;  
}  
  
**int** FlowNetwork::get\_flow\_at\_target() {  
 **return** target != **nullptr** ? get\_flow\_at(\*target) : 0;  
}  
  
FlowNetwork::Vertex &FlowNetwork::get\_or\_add\_vertex(std::string **const** &name) {  
 Vertex\* vertex = mGraph.get\_vertex(name);  
 **if** (vertex == **nullptr**) {  
 **return** mGraph.add\_vertex(NodeData(name));  
 }  
 **return** \*vertex;  
}  
  
**bool** FlowNetwork::add\_edge\_from\_string(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) {  
 Vertex& vertex1 = get\_or\_add\_vertex(tokens[0]);  
 Vertex& vertex2 = get\_or\_add\_vertex(tokens[1]);  
 mGraph.connect(vertex1, vertex2, EdgeData(std::strtol(tokens[2].data(), **nullptr**, 10)));  
 **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;  
 }  
 **return true**;  
 } **else if** (!line.empty()) {  
 std::cout << "invalid formatted edge: " << line << "\n";  
 }  
 **return false**;  
}  
  
**bool** FlowNetwork::read(std::string filename) {  
 std::ifstream stream(filename);  
 **if** (stream) {  
 mGraph.clear();  
 std::string line;  
 **while** (std::getline(stream, line)) {  
 add\_edge\_from\_string(line);  
 }  
 **return true**;  
 } **else** {  
 **return false**;  
 }  
}  
  
**bool** FlowNetwork::save(std::string filename) {  
 std::ofstream stream(filename);  
 **if** (stream) {  
 mGraph.for\_each([&] (Edge& edge) -> **void** {  
 stream << edge.from->data.name << " " << edge.connected->data.name << " " << edge.data.max\_flow << "\n";  
 });  
 **return true**;  
 } **else** {  
 **return false**;  
 }  
}  
  
**void** FlowNetwork::print() {  
 **if** (mGraph.size() == 0) {  
 std::cout << "flow network is emtpy\n";  
 **return**;  
 }  
 mGraph.print(9,  
 [] (Vertex& v) -> std::string {  
 **return** v.data.name;  
 },  
 [] (Edge& edge) -> std::string {  
 **char** line[64];  
 sprintf(line, "%s->%s %2i/%-2i", edge.from->data.name.data(), edge.connected->data.name.data(), edge.data.flow, edge.data.max\_flow);  
 **return** line;  
 });  
}