



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2021 年春季学期 计算学部《软件构造》课程

Lab 2 实验报告

姓名	梁浩
学号	1190200717
班号	1903008
电子邮件	3235962608@qq.com
手机号码	15961817952

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	1
3.1 Poetic Walks	1
3.1.1 Get the code and prepare Git repository	2
3.1.2 Problem 1: Test Graph <String>	2
3.1.3 Problem 2: Implement Graph <String>	3
3.1.3.1 Implement ConcreteEdgesGraph	4
3.1.3.2 Implement ConcreteVerticesGraph	6
3.1.4 Problem 3: Implement generic Graph<L>	10
3.1.4.1 Make the implementations generic	10
3.1.4.2 Implement Graph.empty()	11
3.1.5 Problem 4: Poetic walks	13
3.1.5.1 Test GraphPoet	13
3.1.5.2 Implement GraphPoet	13
3.1.5.3 Graph poetry slam	17
3.1.6 使用 Eclemma 检查测试的代码覆盖率	17
3.1.7 Before you're done	18
3.2 Re-implement the Social Network in Lab1	19
3.2.1 FriendshipGraph 类	19
3.2.2 Person 类	20
3.2.3 客户端 main()	21
3.2.4 测试用例	21
3.2.5 提交至 Git 仓库	21
4 实验进度记录	22
5 实验过程中遇到的困难与解决途径	22
6 实验过程中收获的经验、教训、感想	23
6.1 实验过程中收获的经验教训	23
6.2 针对以下方面的感受	23

1 实验目标概述

本次实验训练抽象数据类型 (ADT) 的设计、规约、测试，并使用面向对象编程 (OOP) 技术实现 ADT。具体来说：

针对给定的应用问题，从问题描述中识别所需的 ADT；

设计 ADT 规约 (pre-condition、post-condition) 并评估规约的质量；

根据 ADT 的规约设计测试用例；

ADT 的泛型化；

根据规约设计 ADT 的多种不同的实现；针对每种实现，设计其表示 (representation)、表示不变性 (rep invariant)、抽象过程 (abstractionfunction)

使用 OOP 实现 ADT，并判定表示不变性是否违反、各实现是否存在表示泄露 (rep exposure)；

测试 ADT 的实现并评估测试的覆盖度；

使用 ADT 及其实现，为应用问题开发程序；

在测试代码中，能够写出 testing strategy 并据此设计测试用例。

2 实验环境配置

简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

特别是要记录配置过程中遇到的问题和困难，以及如何解决的。

实验所需的环境在以前的学习中就已经配好。

在这里给出你的 GitHub Lab2 仓库的 URL 地址 (Lab2-学号)。

<https://github.com/ComputerScienceHIT/HIT-Lab2-1190200717/tree/master>

3 实验过程

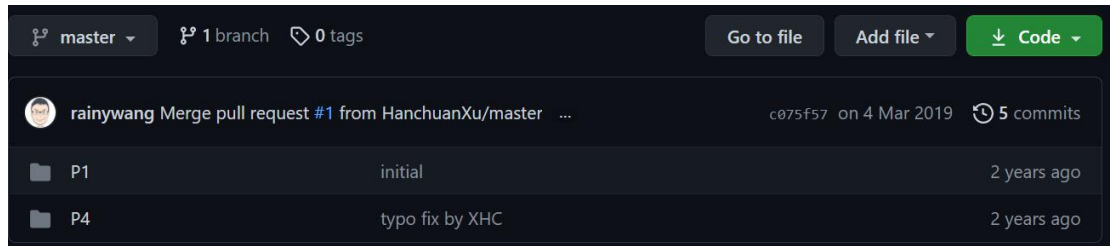
请仔细对照实验手册，针对三个问题中的每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

3.1 Poetic Walks

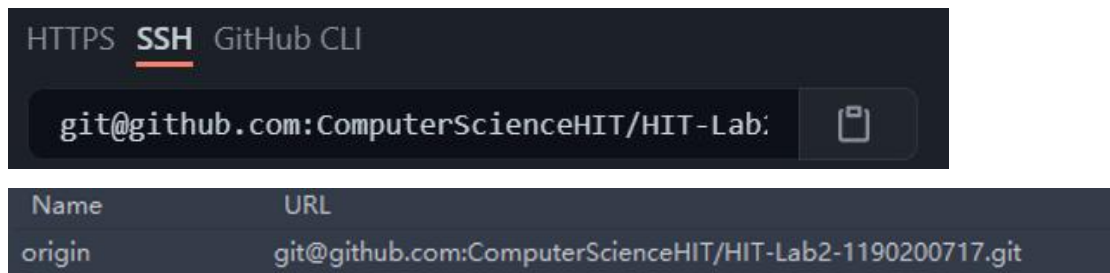
这个实验的主要目的是测试 ADT 的规约设计和 ADT 的多种不同的实现，并练习 TDD 测试优先编程的编程习，在后面练习 ADT 的泛型化。

3.1.1 Get the code and prepare Git repository

从下图处获取所需代码，创建新的项目后按照要求建立目录。



git init 初始化 Git 仓库，将已有的文件提交到本地仓库中，配置好远程仓库后，将本地仓库的内容 push 到远程仓库中。



3.1.2 Problem 1: Test Graph <String>

设计测试 Instance 方法的 testing strategy。主要对每一个需要测试的函数进行输入空间的划分，然后根据输入空间的划分以“最少一次覆盖”的策略进行测试。设计如下：

//boolean add(L vertex)

test add 方法:

graph: 1、graph 为空

2、graph 非空

参数 L vertex: 1、新的节点

2、已经存在 graph 中的点

//int set(L source,L target,int weight)

test set 方法:

graph: 1、graph 为空

2、graph 非空

参数 L source: 1、source 是新的节点

2、source 已经在 graph 中

参数 L target: 1、source 是新的节点

2、source 已经在 graph 中

参数 int weight: 1、weight = 0

2、weight > 0

//boolean remove(L vertex)

test remove 方法:

参数 L vertex: 1、vertex 为新的节点

2、vertex 已经在 graph 中,但没有节点和它相连

3、vertex 已经在 graph 中, 且有节点和它相连

//Set<L> vertices()

test vertices 方法:

graph: 1、graph 是空图

2、graph 非空

//Map<L,Integer> sources(L target)

test sources 方法:

graph: 1、graph 为空

2、graph 非空

参数 L target: 1、target 是新的节点

2、target 是 graph 中的节点, 但是没有边指向它

3、target 是 graph 中的节点, 有边指向它

//Map<L,Integer> targets(L source)

test targets 方法:

graph: 1、graph 为空

2、graph 非空

参数 L source: 1、source 是新的节点

2、source 是 graph 中的点, 但是没有边以它为起点

3、source 是 graph 中的点, 且有边以它为起点

3.1.3 Problem 2: Implement Graph <String>

以下各部分, 请按照 MIT 页面上相应部分的要求, 逐项列出你的设计和实现思路/过程/结果。

这一部分主要是将 Graph<String>实现两次, 分别基于边为主和点为主来实现图的存储和基本操作。

3.1.3.1 Implement ConcreteEdgesGraph

1) edge 类的设计

edge 类的成员变量如下图所示:

```
private final String source,target;
private final int weight;
```

方法如下:

Edge	含参构造函数, 利用传入的参数创建一条新的边
checkRep	检查不变量, 边的权值>0, 边的源顶点和目标顶点非空
getSource	返回边的源顶点
getTarget	返回边的目标顶点
getWeight	返回边的权值
toString	按照指定格式打印边, 比如"a->b 权值为:1"

对于 Abstraction function:

由 source, target, weight 组成的抽象数据类型表示从源顶点到目标顶点具有权重的有向边

```
// Abstraction function:
// 由source, target, weight组成的抽象数据类型
// 表示从源顶点到目标顶点具有权重的有向边
```

对于 Representation invariant:

weight > 0 source 和 target 不为空

```
// Representation invariant:
// weight > 0 source和target不为空
```

对于 Safety from rep exposure:

变量都由 private 和 final 关键字修饰, 类中并没有定义 set 方法, 成员变量都不能从类外部获取或者赋值, 保证了数组不会外泄。

```
// Safety from rep exposure:
// 变量都由private和final关键字修饰, 类中并没有定义set方法
// 它们都不能从类外部获取或者赋值
```

2) Testing strategy for Edge

```
// public String getSource() 返回边的String类型的源节点
// public String getTarget() 返回边的String类型的目标节点
// public int getWeight() 返回边的int类型的权值
// public String toString() 返回一个字符串 格式如下: 边的源节点->边的目标节点 权值为:x
```

3) 实现 Edge 类和 Edge 类的测试

截取部分的代码如下:

```

// TODO toString()
@Override
public String toString(){
    return(this.source+"->" +this.target+" 权值为:"+this.weight+"\n");
}

@Test
public void testEdgetoString(){
    final String source = "a";
    final String target = "b";
    final int weight = 1;
    Edge edge = new Edge(source,target, weight: 1);
    assertEquals( expected: source + "->" + target + " 权值为:" + weight + "\n",edge.toString());
}

```

4) ConcreteEdgesGraph 类的设计
成员变量和成员函数如图所示:

ConcreteEdgesGraph		
f	vertices	Set<String>
f	edges	List<Edge>
m	ConcreteEdgesGraph()	
m	add(String)	boolean
m	checkRep()	void
m	remove(String)	boolean
m	set(String, String, int)	int
m	sources(String)	Map<String, Integer>
m	targets(String)	Map<String, Integer>
m	toString()	String
m	vertices()	Set<String>

对于 Abstraction function:

vertices 表示图中的点 edges 表示图中的边, AF 是从这两种抽象数据类型到一个对应有向图的映射

```

// Abstraction function:
// vertices 表示图中的点 edges 表示图中的边
// AF是从这两种抽象数据类型到一个对应有向图的映射

```

对于 Representation invariant:

顶点不能为空, 边的权值要大于 0

```

// Representation invariant:
// 顶点不能为空
// 边的权值要大于0

```

对于 Safety from rep exposure:

vertices 和 edges 都用 private 和 final 关键字修饰, 返回时利用防御性拷贝

```

// Safety from rep exposure:
// vertices和 edges都用private和final关键字修饰
// 返回时利用防御性拷贝

```

5) 实现 ConcreteEdgesGraph 类和它的测试

截取部分的代码如下:

```
@Override
public int set(String source, String target, int weight) {
    //先看能不能找到对应的边
    int res = 0;
    Iterator<Edge> iter = edges.iterator();
    while(iter.hasNext()){ //执行过程中会执行数据稳定,性能稍差,若在循环过程中要去掉某个元素只能调用iter.remove()方法.
        Edge a = iter.next();
        if(a.getSource().equals(source) && a.getTarget().equals(target)){
            //如果找到了,如果weight = 0, 表示删除,从edges里删除这条边
            res = a.getWeight();
            if(weight==0){
                iter.remove();
                checkRep();
            }
            return res;
        }
    }
}
```

6) 运行测试

全部通过

ConcreteEdgesGraphTest (graph)	42 ms
testEdgeToString	5 ms
testEdgegetSource	0 ms
testEdgetarget	1 ms
testEdgesToString	2 ms
testEdgegetWeight	0 ms
testTargets	1 ms
testAddNull	0 ms
testAdd	0 ms
testSet	0 ms
testVertices	2 ms
testInitialVerticesEmpty	0 ms
testAssertionsEnabled	0 ms
testRemove	31 ms

覆盖率测试如下图所示, 覆盖率达 100%:

ConcreteEdgesGraph	100% (1/1)	100% (10/...	100% (76/...
ConcreteVerticesGraph	100% (1/1)	100% (10/...	100% (11/...
Edge	100% (1/1)	100% (7/7)	100% (15/...
Graph	0% (0/1)	0% (0/1)	0% (0/1)
Vertex	100% (1/1)	100% (10/...	100% (42/...

3.1.3.2 Implement ConcreteVerticesGraph

1) Vertex 类的设计

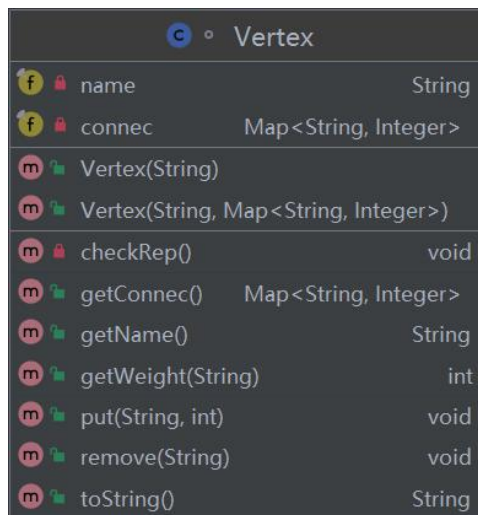
Vertex 类的成员变量如图所示:

```
private final String name; //记录顶点的名字
//String记录的是由这个点可以到达的别的点
//Integer记录的是这条边的权值
private final Map<String,Integer> connec = new HashMap<>();
```


Vertex 类的成员函数包括:

Vertex	含参构造函数, public Vertex(final String name)
Vertex	含参构造函数, public Vertex(final String name, final Map<String,Integer> connec)
checkRep	检查不变量, 顶点的 name 不为空或者"", 边的权值>0
getName	获取顶点名称
getConnec	获取顶点的边集
remove	删除这个源点的某条边
put	加边或者修改权值
getWeight	获取指定边的权值
toString	按指定格式打印

类的 UML 图如下图所示:



对于 Abstraction function:

代表有权值的有向边中的源点

```
// Abstraction function:
// 代表有权值的有向边中的源点
```

对于 Representation invariant:

顶点名不为空或者"", 边的权值大于 0

```
// Representation invariant:
// name != null && name != ""
// connec 中的 key(String) != null 和 ""
// connec 中的 value(Integer) > 0
```

对于 Safety from rep exposure:

name 和 connec 都用 private, final 关键字修饰, 函数返回时创建一个新的对象

```
// Safety from rep exposure:
// name和connec都用private, final 修饰
// 返回时创建一个新的对象
```

2) Testing strategy for Vertex

```
// Testing strategy for Vertex
// public String getName(): 返回顶点的名字
// public Map<String,Integer> getConnec(): 返回顶点的边的情况(目标定点, 权值)
// public void remove(final String target): target 顶点和该顶点间之前有边相连 target 顶点和该顶点间之前没有边相连
// public void put(final String target, final int weight) 为顶点添加新的边
// public int getWeight(final String target) 获得这条边的权值 这条边与该顶点相连(返回权值) 边本来就不相连(返回0)
// public String toString() 返回一个字符串 格式如下: source (target1,weight1) (target2,weight2).....
```

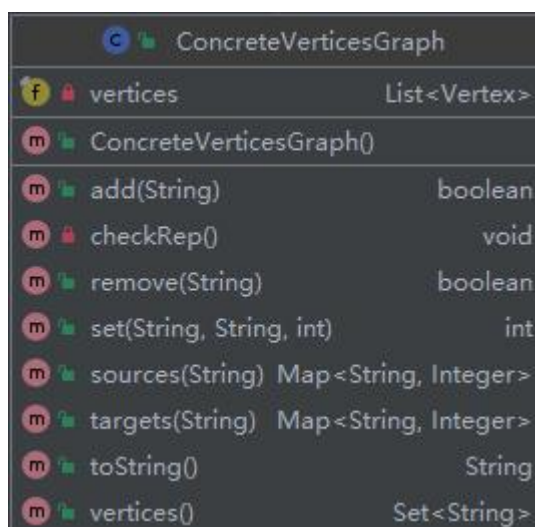
3) 实现 Vertex 类和它的测试

部分代码如下所示:

```
// TODO toString()
public String toString(){
    StringBuilder sb = new StringBuilder();
    sb.append(this.getName());
    sb.append("\t"+" \t"+" \t"+" \t");
    for(Map.Entry<String,Integer> entry : this.connec.entrySet()){
        sb.append('(' +entry.getKey()+',' +entry.getValue()+')');
        sb.append(" "+" ");
    }
    sb.append("\n");
    return new String(sb.toString());
}
```

4) ConcreteVerticesGraph 类的设计

UML 图如下:



对于 Abstraction function:

vertices 到有向图的映射

```
// Abstraction function:
// TODO
// 由vertices到有向图的映射
```

对于 Representation invariant:

vertices 之间不能重复 每个顶点的名字不为空 每条边的权值大于 0

```
// Representation invariant:
// TODO
// vertices之间不能重复 每个顶点的名字不为空 每条边的权值大于0
```

对于 Safety from rep exposure:

vertices 用 private, final 关键字修饰, 返回时用防御性拷贝

```
// Safety from rep exposure:
// TODO
// vertices用private, final 修饰
// 返回时用防御性拷贝
```

4) 实现 ConcreteVerticesGraph 类和它的测试

部分代码截图如下:

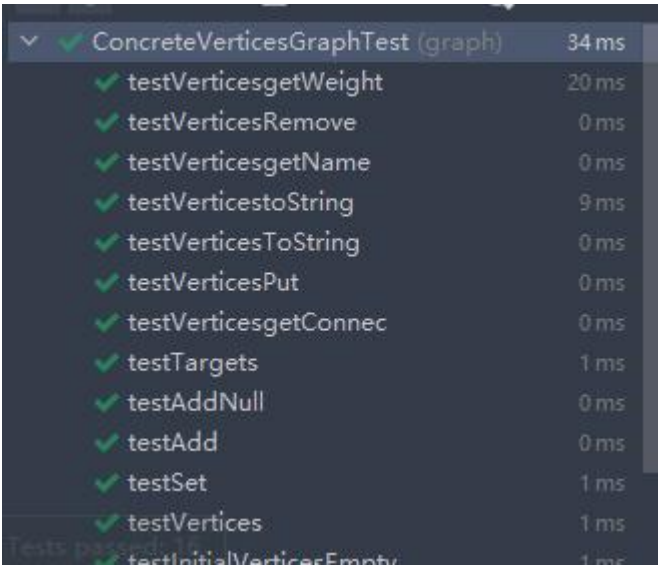
```
@Override
public int set(String source, String target, int weight) {
    int res = 0; // 代表函数的返回值
    // 先看能不能找到对应的源顶点
    for (Vertex temp : vertices) {
        // 如果找到了对应的源顶点, 遍历这个顶点的connec试着能不能找到对应的target
        // 如果两个都满足, 说明新加的这条边原来就在图中, 可以试着删除或者修改weight
        if (temp.getName().equals(source) && temp.getConnec().containsKey(target)) {
            // 找到了这条边, 返回值应为原来这条边的权值
            res = temp.getWeight(target);
            if (weight == 0) {
                // weight为0, 代表删除这条边, 意思就是temp的connec中的target对应的对要删除
                // 删除操作
                temp.remove(target);
                checkRep();
            }
            return res;
        }
    }
}
```

```
@Test
public void testVerticestoString(){
    Graph<String> graph = emptyInstance(); // 创建一个空图

    // 测试1: 如果是空图的情况
    // 打印的效果应该如下所示:
    /*
    源顶点为:          (目标顶点 , 权值)为:
    */
    String str1 = new String( original: "源顶点为: "+"\\t"+"\\t"+"(目标顶点 , 权值)为: "+"\\n");
    assertEquals(str1, graph.toString());
}
```

5) 运行测试

测试全部通过



代码覆盖率达 100%

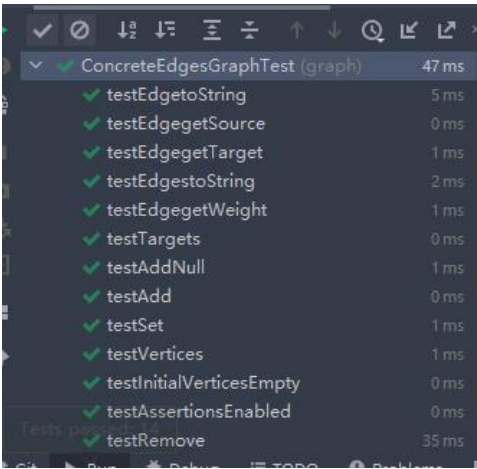
ConcreteVerticesGraph	100% (1/1)	100% (10...)	100% (1...
Edge	0% (0/1)	0% (0/7)	0% (0/15)
Graph	0% (0/1)	0% (0/1)	0% (0/1)
Vertex	100% (1/1)	100% (10...	100% (4...

3.1.4 Problem 3: Implement generic Graph<L>

将已有的两个 Graph<String>的实现改为 Graph<L>的泛型实现。

3.1.4.1 Make the implementations generic

将 ConcreteEdgesGraph 和 ConcreteVerticesGraph 都用泛型实现后，重新跑测试



ConcreteVerticesGraphTest (graph)	21 ms
Show Passed esgetWeight	10 ms
testVerticesRemove	0 ms
testVerticesgetName	0 ms
testVerticestoString	2 ms
testVerticesToString	1 ms
testVerticesPut	3 ms
testVerticesgetConnection	0 ms
testTargets	0 ms
testAddNull	0 ms
testAdd	0 ms
testSet	1 ms
testVertices	2 ms

覆盖率依旧是 100%

ConcreteEdgesGraph	100% (1/1)	100% (10...)	100% (78...)
ConcreteVerticesGraph	100% (1/1)	100% (10...)	100% (11...)
Edge	100% (1/1)	100% (7/7)	100% (15...)
Graph	0% (0/1)	0% (0/1)	0% (0/1)
Vertex	100% (1/1)	100% (10...)	100% (42...)

3.1.4.2 Implement Graph.empty()

1、选择 ConcreteEdgesGraph 作为返回对象实现 Graph.empty()函数:

```
public static <L> Graph<L> empty() {
    //java 8 中新增了接口的默认方法，简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。
    //接口实现方法一共有两种：
    //1. 在方法名前面加个 default 关键字。
    //2. 可以声明（并且可以提供实现）静态方法。
    //empty 方法即是采用了第二种。
    return new ConcreteEdgesGraph<L>();
}
```

2、测试 GraphStaticTest

尝试 label 为 long 的情况：


```
@Test
public void testGraphLong(){
    Graph<Long> graph = Graph.empty();
    long a = 1,b = 2,c = 3,d = 4;
    assertTrue(graph.add(a));
    assertTrue(graph.add(b));
    assertEquals( expected: 0,graph.set(a,b, weight: 10));
    assertEquals( expected: 10,graph.set(a,b, weight: 20));
    assertEquals( expected: 0,graph.set(a,c, weight: 11));
    assertEquals( expected: 0,graph.set(b,d, weight: 12));
    /
    图的状态应该为:
    /
    有向图的顶点包括:1 2 3 4
    /
    有向图的边有:
    /
    1->2 权值为:20
    /
    1->3 权值为:11
    /
    2->4 权值为:12
}
```

尝试 label 为 double 的情况：

```
@Test
public void testGraphDouble(){
    Graph<Double> graph = Graph.empty();
    double a = 1.1,b = 2.2,c = 3.3,d = 4.4;
    assertTrue(graph.add(a));
    assertTrue(graph.add(b));
    assertEquals( expected: 0,graph.set(a,b, weight: 10));
    assertEquals( expected: 10,graph.set(a,b, weight: 20));
    assertEquals( expected: 0,graph.set(a,c, weight: 11));
    assertEquals( expected: 0,graph.set(b,d, weight: 12));
    /
    图的状态应该为:
    /
    有向图的顶点包括:1.1 2.2 3.3 4.4
    /
    有向图的边有:
    /
    1.1->2.2 权值为:20
    /
    1.1->3.3 权值为:11
}
```

写完后运行测试，运行成功

GraphStaticTest (graph) 37 ms

testGraphLong 35 ms

testAssertionsEnabled 1 ms

testGraphDouble 1 ms

testEmptyVerticesEmpty 0 ms

Tests passed: 4, 1 failed: 0, 1 skipped: 0
"D:\xuexi\java_study\jdk_1.8\bin\java.exe" ...
有向图的顶点包括:1 2 3
有向图的边有:
1->2 权值为:20
1->3 权值为:11

查看一下覆盖率，依旧是 100%：

Element	Class, %	Method, %	Line, %
ConcreteEdgesGraph	100% (1/1)	100% (10/...	100% (78/...
ConcreteVerticesGraph	0% (0/1)	0% (0/10)	0% (0/110)
Edge	100% (1/1)	100% (7/7)	100% (15/...
Graph	100% (1/1)	100% (1/1)	100% (1/1)
Vertex	0% (0/1)	0% (0/10)	0% (0/42)

3.1.5 Problem 4: Poetic walks

这个问题一部分是用给的语料生成图，相邻的单词间用一条有向边连接，另一部分是给定一个输入字符串，通过在图中判断它们之间是否有 bridge 来对字符串进行扩充。

3.1.5.1 Test GraphPoet

Testing strategy:

对于构造函数:

- 1、传入不存在的文件
- 2、传入的文件为空
- 3、传入的语料文件只有一行单词
- 4、传入具有多行的语料文件

对于 poem 函数:

- 1、传入的字符串为 null
- 2、传入的字符串!=null，但只有一个单词
- 3、传入的字符串!=null，不止一个单词(还可以再设计根据权值选 bridge 的情形)

对于 toString 函数:

- 1、graph 为空
- 2、graph 不为空，但没有边
- 3、graph 有顶点有边

测试策略部分截图如下:

```
// Testing strategy
// 对于构造函数:
// 1、传入不存在的文件 2、传入的文件为空
// 3、传入的语料文件只有一行单词 4、传入具有多行的语料文件







// 对于poem函数:
// 1、传入的字符串为null 2、传入的字符串!=null，但只有一个单词
// 3、传入的字符串!=null，不止一个单词(还可以再设计根据权值选bridge的情形)

// 对于toString函数:
// 1、graph为空 2、graph不为空，但没有边 3、graph有顶点有边
```

3.1.5.2 Implement GraphPoet

1) GraphPoet 的结构分析

UML 图如下:

	GraphPoet	
	graph	<code>Graph<String></code>
	GraphPoet	<code>(File)</code>
	checkRep	<code>() void</code>
	poem	<code>(String) String</code>
	toString	<code>() String</code>

对于成员变量 graph:

```
private final Graph<String> graph = Graph.empty();
```

对于含参构造函数:

1、如果传入的文件不存在, catch 异常, 打印

```
}catch (FileNotFoundException e){
    System.out.println("传入的语料文件不存在!");
}
```

2、如果文件存在, 按行读入, 根据要求生成对应的图

```
initClose(), // 关闭文件流并释放资源
for (int i = 0; i < words.size() - 1; i++) {
    String source = words.get(i);
    String target = words.get(i + 1);
    int preWeight = 0; // 初始化为0方便后面统一起来
    if (this.graph.vertices().contains(source) && this.graph.vertices().contains(target)) {
        if (this.graph.sources(target).containsKey(source)) {
            preWeight = this.graph.sources(target).get(source);
        }
    }
    // 查看两个顶点是否已经有边了
    this.graph.set(source, target, weight: preWeight + 1);
}
```

对于 poem 方法:

关键在于寻找 bridge, 当有多个符合要求的节点时, 要选择权值大的。

```
//寻找有没有bridge的方法:
//显然没有的情况更多, 我们可以转化为什么情况下有bridge
//首先:source和target都在图中
//遍历source的目标节点temp, 如果有某个temp是target的源节点, 就说明这个点可以作为bridge
//当有很多bridge时, 取最大权值的bridge
//因此要记录下最大的权值max_Weight = 前一条边weight + 后一条边weight
if(graph.vertices().contains(source)&&graph.vertices().contains(target)){
    for(Map.Entry<String,Integer> entry: graph.targets(source).entrySet()){
        if(graph.sources(target).containsKey(entry.getKey())){
            int temp_Weight = entry.getValue()+graph.sources(target).get(entry.getKey());
            if(temp_Weight>max_Weight){
                bridge = entry.getKey();
                max_Weight = temp_Weight;
            }
        }
    }
}
//到这里就找到了bridge
```


对于 toString 方法:

沿用 ConcreteEdgesGraph.java 中的打印方法

2) 关于 AF, RI 和 rep exposure

对于 Abstraction function:

将文本文件转化成有向图, 图的顶点就是文件中的单词, 相邻单词在图中对应的顶点间有有向边

```
// Abstraction function:  
// 将文本文件转化成有向图, 图的顶点就是文件中的单词, 相邻单词在图中对应的顶点间有有向边
```

对于 Representation invariant:

图中的顶点不为空或者"", 边的权值要大于 0

```
// Representation invariant:  
// 图中的顶点不为空或者""  
// 边的权值要大于0
```

对于 Safety from rep exposure:

graph 用 private 和 final 关键字修饰, 返回参数时创建新的变量

```
// Safety from rep exposure:  
// graph用private和final关键字修饰  
// 返回参数时创建新的变量
```

3) 类的实现

部分代码截图如下:

```
StringBuilder sb = new StringBuilder();  
List<String> list = new ArrayList<>(Arrays.asList(input.split( regex: " ")));  
for(int i = 0; i < list.size()-1; i++) {  
    String source = list.get(i).toLowerCase();  
    String target = list.get(i+1).toLowerCase();  
    //source和target为了和语料库保持一致先转换为小写  
    sb.append(list.get(i)).append(" "); //原来的大写还是保持大写的样子  
    //开始寻找有没有bridge  
    String bridge = "";  
    int max_weight = 0;
```

4) 测试

对 poem 参数不同情况下的测试:

当传入的文件不存在时, 控制台打印“传入的语料文件不存在!”

```
@Test
public void testGraphPoet() throws IOException {
    System.out.println("这是测试传入不存在的文件后的打印结果:");
    GraphPoet graphPoem = new GraphPoet(new File( pathname: "test/P1/poet/notExist.txt"));
    //控制台将会打印: 传入的语料文件不存在!
}
```

当传入的语料库文件为空时, poem 后的字符串和以前一样

```
//如果传入的语料库是空的
//poem函数传入什么字符串都应该是返回和原来一样的字符串
GraphPoet graphPoem1 = new GraphPoet(new File( pathname: "test/P1/poet/empty.txt"));
assertEquals( expected: "abc efg", graphPoem1.poem( input: "abc efg"));
```

下面是语料库有一行或多行字符串时的情况

```
//如果传入的语料库只有一行
//下面的例子使用的和Main函数里一样的例子
// This is a test of the Mugar Omni Theater sound system.
GraphPoet graphPoem2 = new GraphPoet(new File( pathname: "test/P1/poet/singleLine.txt"));
assertEquals( expected: "Test of the system.", graphPoem2.poem( input: "Test the system.));

//如果传入的语料库不止一行, 这里用的是CMU的实验要求里举的例子
//To explore strange new worlds
//To seek out new life and new civilizations
//如果传入的input是:
GraphPoet graphPoem3 = new GraphPoet(new File( pathname: "test/P1/poet/multiline.txt"));
assertEquals( expected: "Seek to explore strange new life and exciting synergies!", graphPoem3.poem( input: "Seek to explo
```

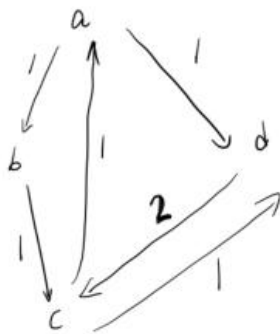
对 ToString 在空图、图非空（只有顶点）、图非空（既有顶点，也有边）的情况下的测试:

```
//图为空的情况下:
//打印状态如下:
//有向图的顶点包括:
//有向图的边有:
GraphPoet graph1 = new GraphPoet(new File( pathname: "test/P1/poet/toStringEmpty.txt"));
String rs = "有向图的顶点包括:\n" +
    "有向图的边有:\n";
assertEquals(rs,graph1.toString());
```

对 bridge 按照权值大小选取的测试:

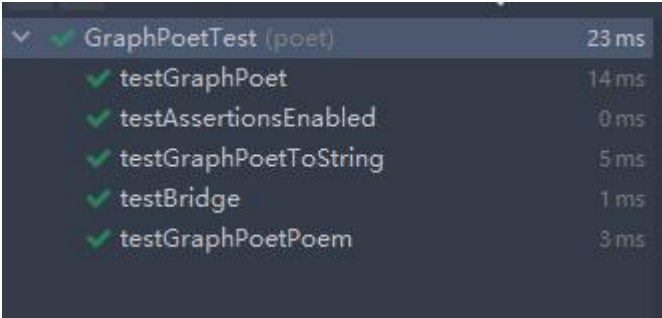
如下图所示, 构造一个特殊的图, 涉及到 bridge 的选择

a b c a d c d c



```
//可以看出来：
//a->b->c 权值为2
//a->d->c 权值为3
String input = "a c";
//最后扩充得到的应该是a d c
assertEquals( expected: "a d c",graph.poem(input));
```

所有的测试结果:



✓ GraphPoetTest (poet)	23 ms
✓ testGraphPoet	14 ms
✓ testAssertionsEnabled	0 ms
✓ testGraphPoetToString	5 ms
✓ testBridge	1 ms
✓ testGraphPoetPoem	3 ms

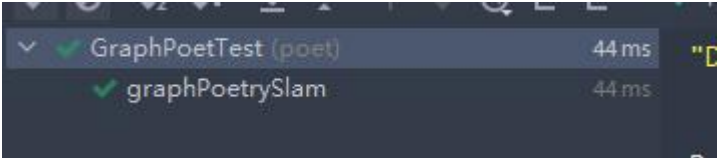
覆盖率测试如下:



Element	Class, %	Method, %	Line, %
GraphPoet	100% (1/1)	100% (5/5)	100% (54/...

3.1.5.3 Graph poetry slam

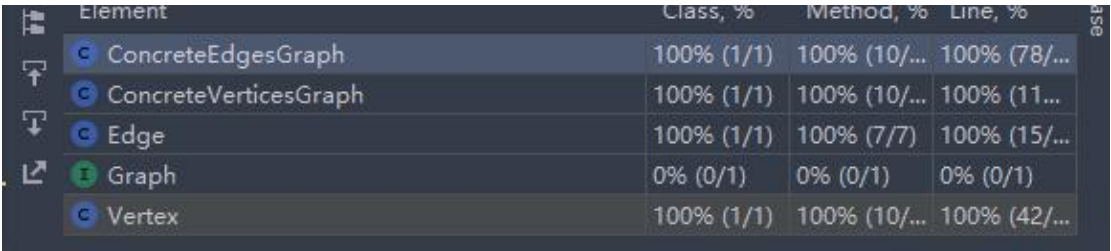
选取了 WILLIAM BUTLER YEATS 的 *When You Are Old*
输入字符串为"you old", 扩充后的结果为"you are old"



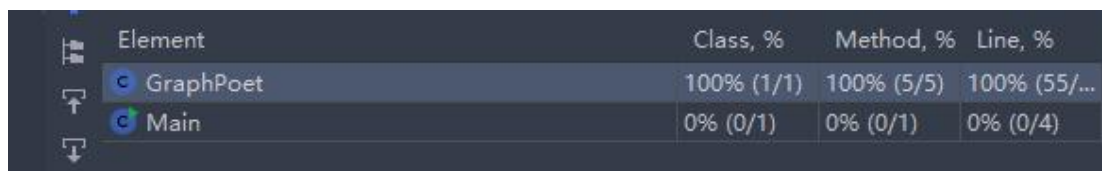
✓ GraphPoetTest (poet)	44 ms
✓ graphPoetrySlam	44 ms

3.1.6 使用 Eclemma 检查测试的代码覆盖度

代码覆盖度都是 100%



Element	Class, %	Method, %	Line, %
ConcreteEdgesGraph	100% (1/1)	100% (10/...	100% (78/...
ConcreteVerticesGraph	100% (1/1)	100% (10/...	100% (11/...
Edge	100% (1/1)	100% (7/7)	100% (15/...
Graph	0% (0/1)	0% (0/1)	0% (0/1)
Vertex	100% (1/1)	100% (10/...	100% (42/...



Element	Class, %	Method, %	Line, %
GraphPoet	100% (1/1)	100% (5/5)	100% (55/...
Main	0% (0/1)	0% (0/1)	0% (0/4)

3.1.7 Before you're done

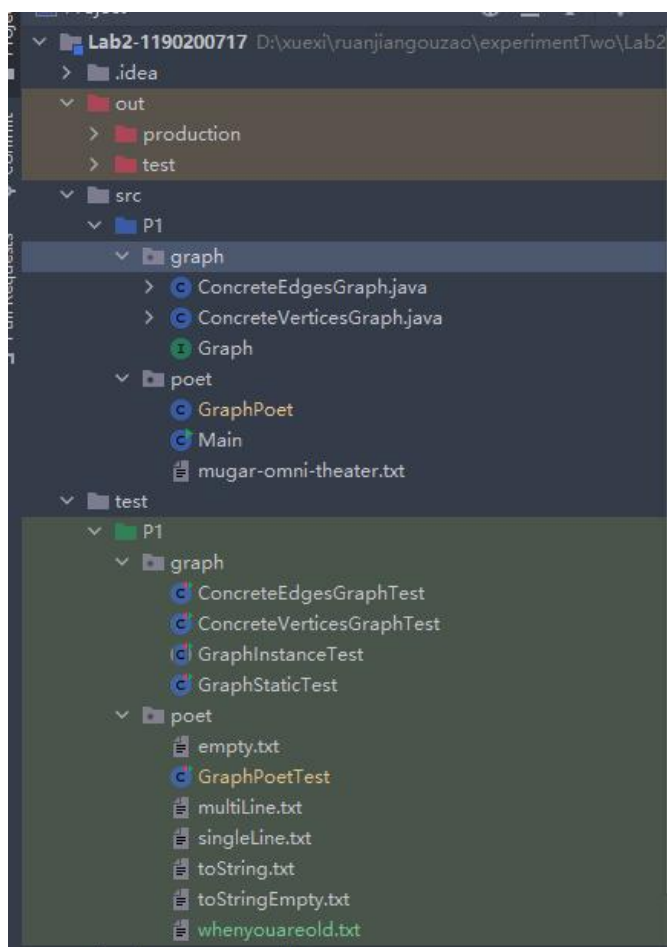
请按照 http://web.mit.edu/6.031/www/sp17/psets/ps2/#before_youre_done 的说明, 检查你的程序。

如何通过 Git 提交当前版本到 GitHub 上你的 Lab2 仓库。

```
git add --all
git commit -m "...."
git push -u origin master
```

直接通过 idea 来提交会更加方便

项目的目录结构树状示意图如下:

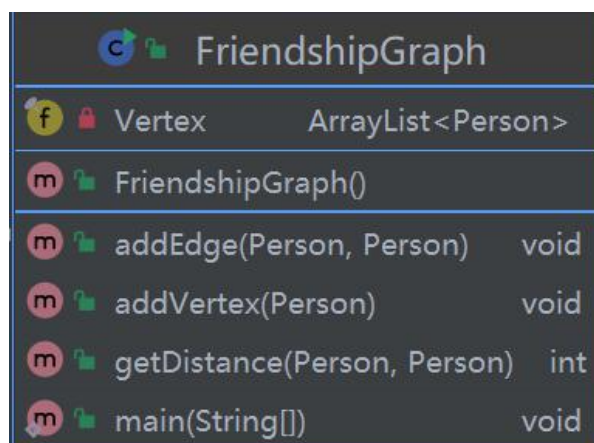


3.2 Re-implement the Social Network in Lab1

这个任务主要是重新实现 Lab1 中的 Social Network, 利用在 P1 中已经写好的 `Graph<L>` 接口来实现, 尽量重用已有的函数。

3.2.1 FriendshipGraph 类

这里给出 FriendshipGraph 类的 UML 图



包含成员变量 Vertex:

```
private final ArrayList<Person> Vertex = new ArrayList<>();
```

用来保存图中的各顶点。

对于 addVertex 方法:

充分使用 Graph<L>里面提供的函数 add, 即可实现增加顶点的功能。如果顶点名重复, 打印错误信息, 程序直接退出。

对于 addEdge 方法:

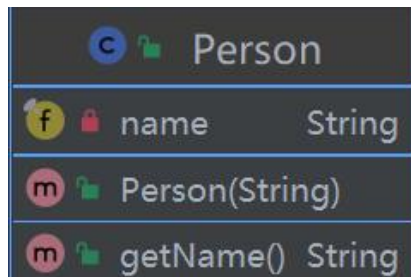
充分利用 Graph<L>里面提供的函数 set, 即可实现增加顶点的功能, 不过要先判断两个顶点是否相同。

对于 getDistance 方法:

利用 BFS 算法来求两个顶点之间的最短距离。

3.2.2 Person 类

下面给出 Person 类的 UML 图:



对于 Abstraction function:

```
// Abstraction function:
// 图中顶点
```

对于 Representation invariant:

```
// Representation invariant:
// name != null 且 name != ""
// 顶点之间不能有相同的name
```

对于 Safety from rep exposure:

```
// Safety from rep exposure
// name用private和final关键字修饰 只实现了get方法
// 用户不能从外界直接访问类的成员变量
```

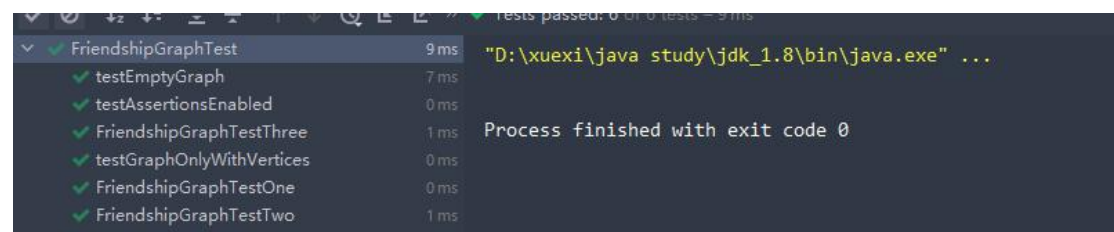
3.2.3 客户端 main()

利用 Lab1 中已有的 main 客户端即可。

3.2.4 测试用例

测试策略主要是根据 FriendshipGraph 中的图的类型进行测试，主要分为空图的测试、仅有顶点的图的测试、复杂图的测试。

测试结果如下图：



覆盖率测试如下图所示：

Class	Line Coverage	Branch Coverage	Method Coverage
FriendshipGraph	100% (1/1)	100% (3/3)	92% (36/39)
Person	100% (1/1)	100% (2/2)	100% (4/4)

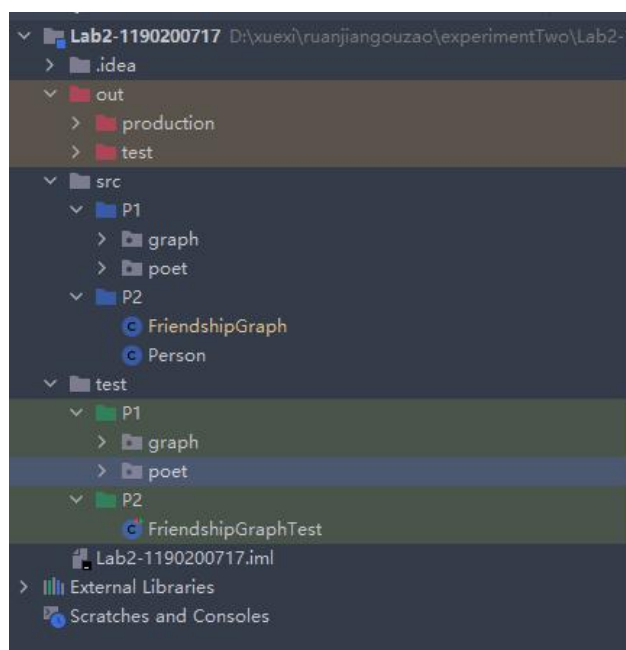
3.2.5 提交至 Git 仓库

如何通过 Git 提交当前版本到 GitHub 上你的 Lab3 仓库。

```
git add --all
git commit -m "...."
git push -u origin master
```

用 idea 的 git 工具来提交会更快。

在这里给出你的项目的目录结构树状示意图。



4 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

日期	时间段	计划任务	实际完成情况
2021/6/7	19:30-22:30	Implement ConcreteEdgesGraph	按时完成
2021/6/8	19:00-0:00	编写 ConcreteVerticesGraph 的方法	按时完成
2021/6/9	8:30-9:30	初步完成 ConcreteVerticesGraph.java	按时完成
2021/6/10	14:30-19:30	ConcreteEdgesTest 的编写	按时完成
2021/6/11	19:30-22:30	poetic walk	按时完成
2021/6/12	19:30-23:30	完成 P2	按时完成

5 实验过程中遇到的困难与解决途径

遇到的难点	解决途径
英文题目看不懂	Google 翻译，逐字逐句揣摩
Java 的泛型和接口概念不懂	菜鸟教程自学

6 实验过程中收获的经验、教训、感想

6.1 实验过程中收获的经验教训

代码的性能可以再优化, 对于成员变量的封闭性平时考虑的比较少。

6.2 针对以下方面的感受

(1) 面向 ADT 的编程和直接面向应用场景编程, 你体会到二者有何差异?

在面向 ADT 编程的时候更多的需要考虑代码的可复用性, 需要使整个的编程更加具有普适性; 而面向应用场景编程, 更加直接和方便, 但是适用的范围明显更小。

(2) 使用泛型和不使用泛型的编程, 对你来说有何差异?

使用泛型编程的话, 写起来不太习惯, 而不使用泛型编程的话, 写起来更加顺手。

(3) 在给出 ADT 的规约后就开始编写测试用例, 优势是什么? 你是否能够适应这种测试方式?

可以尽早发现错误, 可以逐渐适应。

(4) P1 设计的 ADT 在多个应用场景下使用, 这种复用带来什么好处?

可以将已有的代码进行复用来尽量节省时间。

(5) P3 要求你从 0 开始设计 ADT 并使用它们完成一个具体应用, 你是否已适应从具体应用场景到 ADT 的“抽象映射”? 相比起 P1 给出了 ADT 非常明确的 rep 和方法、ADT 之间的逻辑关系, P3 要求你自主设计这些内容, 你的感受如何?

没做 P3。

(6) 为 ADT 撰写 specification, invariants, RI, AF, 时刻注意 ADT 是否有 rep exposure, 这些工作的意义是什么? 你是否愿意在以后编程中坚持这么做?

提前发现错误, 避免数据外泄, 愿意。

(7) 关于本实验的工作量、难度、deadline。

工作量比较大, 时间紧凑。

(8) 《软件构造》课程进展到目前, 你对该课程有何体会和建议?

和计算机系统一起同时做实验, 有点累人。