



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	沈城有
学号	1190200526
班号	
电子邮件	
手机号码	

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	2
3.1 Poetic Walks	2
3.1.1 Get the code and prepare Git repository	2
3.1.2 Problem 1: Test Graph <String>	3
3.1.3 Problem 2: Implement Graph <String>	3
3.1.3.1 Implement ConcreteEdgesGraph	3
3.1.3.2 Implement ConcreteVerticesGraph	6
3.1.4 Problem 3: Implement generic Graph<L>	9
3.1.4.1 Make the implementations generic	9
3.1.4.2 Implement Graph.empty()	9
3.1.5 Problem 4: Poetic walks	10
3.1.5.1 Test GraphPoet	10
3.1.5.2 Implement GraphPoet	10
3.1.5.3 Graph poetry slam	12
3.1.6 使用 Eclemma 检查测试的代码覆盖率	12
3.1.7 Before you're done	13
3.2 Re-implement the Social Network in Lab1	14
3.2.1 FriendshipGraph 类	14
3.2.2 Person 类	16
3.2.3 客户端 main()	17
3.2.4 测试用例	17
3.2.5 提交至 Git 仓库	17
4 实验进度记录	19
5 实验过程中遇到的困难与解决途径	19
6 实验过程中收获的经验、教训、感想	19
6.1 实验过程中收获的经验教训	19
6.2 针对以下方面的感受	20

1 实验目标概述

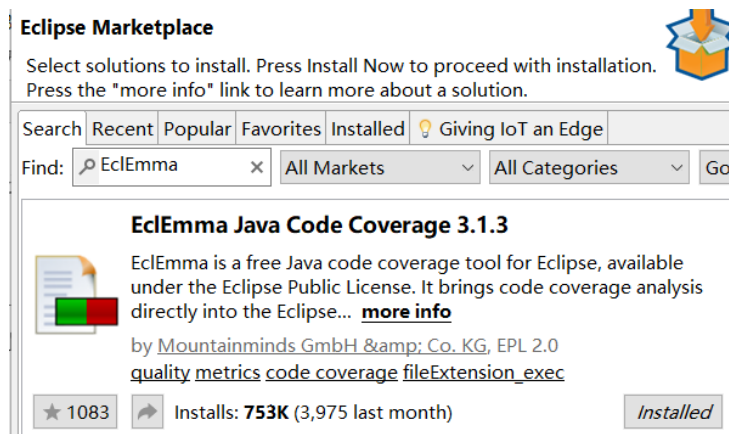
本次实验训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象编程（OOP）技术实现 ADT。具体来说：

- 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（abstraction function）
- 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

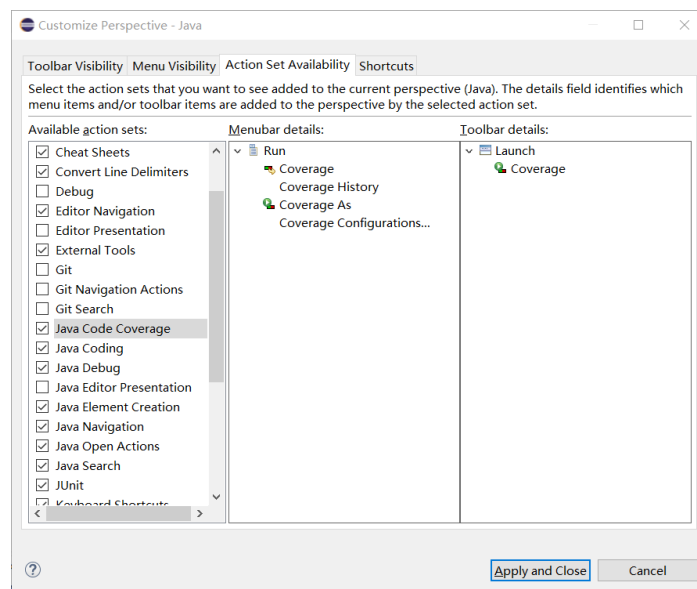
本次实验需要在 Eclipse IDE 中配置安装 EclEmma，我通过浏览 EclEmma 官网找到了安装教程。

首先在 Eclipse 中找到 Help-Eclipse Marketplace，在其中搜索 EclEmma，结果如下图：



显示已经安装，但我没有在工具栏看到其图标。打开 Window-Perspective-

Customize Perspective, 弹出一个页面并选择 Action Set Availability, 勾选 Java Code Coverage (如下图) 并保存设置, 发现工具栏出现图标 , 配置完成。



GitHub Lab2 仓库的 URL 地址:

<https://github.com/ComputerScienceHIT/HIT-Lab2-1190200526>

3 实验过程

3.1 Poetic Walks

该任务要求首先实现并测试 $\text{Graph}\langle\text{String}\rangle$, 其包括边图、点图两种具体实现, 随后从具体扩展到泛型 $\text{Graph}\langle L \rangle$, 最后使用编写的 ADT 完成简单的应用。

3.1.1 Get the code and prepare Git repository

我首先通过 `git clone` 命令 (SSH 方式) 将 GitHub 上用于实验提交的代码库克隆至本地, 这同时也完成了本地库与远程库的关联。随后使用 GitHub 网页端下载了包含任务代码的 ZIP 包, 调整结构后放入本地 git 仓库。最后使用 Eclipse IDE 在此目录创建新工程, 调整工程 `buildpath` 及部分 `import` 后将初始化的项目推送至 GitHub。

注: 此步完成后的项目结构见 GitHub 描述为 “Lab2 init.” 的提交。

3.1.2 Problem 1: Test Graph <String>

要求我们针对 Graph<String>设计和实现对应的测试并写明测试策略,规定在 GraphInstanceTest 中的测试用例必须是符合 Graph 规约的客户端代码。

我的思路是按照等价类划分情况,并针对每个情况设置至少一个测试用例,具体测试策略如下:

```
/** Testing strategy (测试策略)
 * 测试用例设计基本按照等价类划分来进行,且保证每个等价类有不少于1个测试用例
 * observers在测试中被调用,相当于一同测试
 * testAdd(): 添加同一个点 / 不同的点,
 *             检查返回值并使用observer--vertices()检查结果
 * testSet(): weight = 0 / > 0 / < 0 (非法), 不存在的边 / 存在的边,
 *             点不存在 / 存在,检查返回值
 * testRemove(): 不存在 / 存在的点 (存在的点也要测试边移除是否正确),
 *             检查返回值
 * testVertices(): 空 / 非空 (空情况已经被测试)
 *             检测返回的集合是否与真实情况一致
 * testSourceTargets(): 同时测试两个函数,空 / 非空情况,
 *             检查返回的集合是否与真实情况一致
 */
```

注: 保留了 testInitialVerticesEmpty()用于测试 testVertices()未测试的空情况。

测试代码较长,此处略。

3.1.3 Problem 2: Implement Graph <String>

此部分要求重写 Graph 里的方法,分别实现以点为基础、以边为基础的图。

3.1.3.1 Implement ConcreteEdgesGraph

要求使用给定的私有属性完成 ConcreteEdgesGraph 的编程,且 Edge 类要求必须是不可变(immutable)的。

首先设计实现 Edge 类,过程如下:

- 1) Edge 类应保存有向边的始点、终点和边权信息,故定义以下私有属性:

```
// fields
private final String start, end;
private final int weight;
```

- 2) 需要实现的方法如下:

- Edge(): 构造方法
- checkRep(): 检查表示不变性
- getStart()、getEnd()、getWeight(): 获取边信息(始点、终点、边权)
- isSameEdge(): 判断两边是否为同一条(用于辅助图实现,简单比较两边的始、终顶点是否分别相同)
- toString(): 返回有向边的可读表示

3) AF、RI 和 Safety from rep exposure:

```
// Abstraction function:
/**
 * AF(start) = 边的始点
 * AF(end) = 边的终点
 * AF(weight) = 边权
 */
// Representation invariant:
/**
 * 1) 边的始点和终点均不为空, 且不应相同
 * 2) 边权应大于0
 */
// Safety from rep exposure:
// 将类中属性均设置为private final
```

4) 设计 Edge 测试策略并编写测试:

测试策略:

```
/**
 * 测试Edge类中的各方法
 * 使用空顶点、负权和始终点相同情况检测checkRep()是否正常（在构造函数里被调用）
 * 使用合法输入，通过observer检查构造函数是否正常
 * 相同边 / 不同边检测isSameEdge()是否正确
 * 合法输入检查toString返回值是否与预期相同
 */
```

具体测试此处略去。

5) 实现 Edge 类（此处略，见源代码文件 ConcreteEdgesGraph.java）。

然后设计实现 ConcreteEdgesGraph 类，具体过程如下:

1) 使用实验要求的私有属性:

```
private final Set<String> vertices = new HashSet<>();
private final List<Edge> edges = new ArrayList<>();
```

2) AF、RI 与 Safety from rep exposure:

```
// Abstraction function:
/**
 * AF(vertices) = 图中顶点
 * AF(edges) = 图中边
 */
// Representation invariant:
/**
 * 1) 边的起始点和终点均在顶点集中
 * 2) 顶点不能为空(null)
 * 3) 始点与终点不能一致
 * 注: 部分RI在Edge类中检测
 */
// Safety from rep exposure:
/**
 * 1) 类的属性均声明为private final
 * 2) 由于vertices和edges均为mutable, 方法中进行了defensive copy
 */
```

3) 实现各方法，主要思路:、

除 toString() 方法外，均根据 AF、RI、Safety from rep exposure 及 Graph.java 文件中的规约来实现具体的方法，在源代码文件 ConcreteEdgesGraph.java 中有较详细的注释说明，故代码此处略去；对于 toString() 方法，我的思路为输出图顶点数、边数，然后输出每条边的信息（调用 Edge 类的 toString 方法，

并连接字符串), 具体实现见下图:

```
// toString()
/**
 * 返回图的可读表示
 * 注: 顶点数、边数、顶点信息及每条边具体信息
 * @return 图的可读表示
 */
@Override public String toString() {
    String base = "V:" + Integer.toString(vertices.size()) + ", E:"
        + Integer.toString(edges.size()) + ", Vs:";
    if (vertices.size() == 0) {
        checkRep();
        return base + " empty, Es: empty";
    }
    for (String v : vertices) {
        base += " " + v;
    }
    if (edges.size() == 0) {
        checkRep();
        return base + ", Es: empty";
    }
    else {
        for (Edge e : edges)
            base += " " + e.toString();
    }
    checkRep();
    return base;
}
```




4) 编写测试策略, 设计测试用例: 由于其他方法的测试在 Problem 1 中已经进行了编写, 此处仅编写 toString()方法的测试策略并实现其测试, 如下图:

```
// Testing strategy for ConcreteEdgesGraph.toString()
// 使用几个合法的简单边图检查返回的字符串是否正确
// 空图、有顶点无边图、有顶点有边图

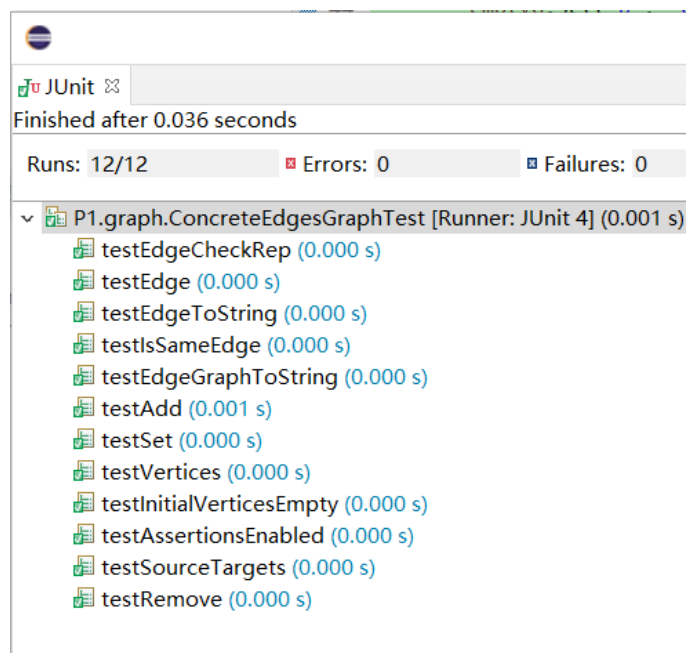
// tests for ConcreteEdgesGraph.toString()
@Test
public void testEdgeGraphToString() {
    Graph<String> emptyG = emptyInstance();
    assertEquals("V:0, E:0, Vs: empty, Es: empty", emptyG.toString());
    emptyG.add("a");
    emptyG.add("b");
    assertEquals("V:2, E:0, Vs: a b, Es: empty", emptyG.toString());
    emptyG.set("a", "b", 1);
    emptyG.set("b", "c", 2);
    assertEquals("V:3, E:2, Vs: a b c, Es: a->b:1 b->c:2", emptyG.toString());
}
```

最后运行测试, 得到的结果如下图:

代码覆盖率情况:

ConcreteEdgesGraph.java		95.8 %
ConcreteEdgesGraph		94.9 %
Edge		100.0 %

测试用例通过情况:



3.1.3.2 Implement ConcreteVerticesGraph

要求使用给定的私有属性完成 ConcreteVerticesGraph 的编程, 且 Vertex 类要求必须是可变(mutable)的。

首先设计 Vertex 类, 过程如下:

1) Vertex 类应保存顶点及顶点间边关系的相关信息, 故定义以下私有属性:

```
// fields
private final String vertexName; // 顶点名
private final Map<String, Integer> inMap; // 入边信息
private final Map<String, Integer> outMap; // 出边信息
```

2) 需要实现的方法如下:

- Vertex(): 构造方法
- checkRep(): 检查不变性
- getName()、getIns()、getOuts(): 访问私有属性信息
- setIn()、setOut(): 设置顶点入边/出边信息 (整合了新增、修改与删除), 用于辅助 ConcreteVerticesGraph 类 set()、remove()等方法的实现
- toString(): 返回 Vertex 类对象可读表示

这里展示较关键的方法 setIn(), setOut()与其类似:

```
/**
 * 操作以该顶点为终点的边信息
 * 边权大于0则修改或新增边
 * 等于0则删除边 (不存在不作任何修改)
 * 小于0非法情况, 返回 -1, 不作任何修改
 * @param sourceName 始点名
 * @param weight 边权
 * @return 原边权值, 不存在则为0, 非法输入为 -1
 */
```



```

public int setIn(L sourceName, int weight) {
    Integer lastWeight = -1;
    if (weight > 0) {
        lastWeight = inMap.put(sourceName, weight);
    } else if (weight == 0) {
        lastWeight = inMap.remove(sourceName);
    }
    if (lastWeight == null) {
        lastWeight = 0;
    }
    checkRep();
    return lastWeight.intValue();
}

```

3) AF、RI 和 Safety from rep exposure:

```

// Abstraction function:
/**
 * AF(vertexName) = 顶点名
 * AF(inMap) = 指向这个顶点的所有顶点及边权（数据对）
 * AF(outMap) = 以此顶点为始点的所有顶点及边权（数据对）
 */

// Representation invariant:
/**
 * 1) 顶点名不能为空(null)
 * 2) 边权均大于0
 * 3) 顶点的边对应的始点和终点不能为本身
 *    (inMap和outMap中没有key值为此顶点名的元素)
 */

// Safety from rep exposure:
/**
 * 1) 将类中属性均声明为private final
 * 2) 由于inMap、outMap均为mutable, 返回时进行defensive copy
 */

```

4) 设计 Vertex 测试策略并编写测试:

测试策略:

```

// Testing strategy for Vertex
/**
 * 测试Vertex类中的各方法
 * 使用违反RI的测试检查checkRep()是否正确
 * 使用合法输入, 通过observer检测构造函数, 同时也测试了observer
 * 分别用顶点名非空, weight > 0 / = 0 / < 0 (非法) 的情况测试setIn()、setOut()
 * (顶点名为空的情况违反RI之前, 检查checkRep()时测试过)
 * 使用合法输入 (顶点无边/有边) 检查toString()方法返回值是否与预期相同
 */

```

具体测试代码略。

5) 按照设计实现 Vertex 类:

见源代码文件。

随后设计实现 ConcreteVerticesGraph 类, 具体过程如下:

1) 使用实验要求的私有属性:

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

2) AF、RI 与 Safety from rep exposure:

```
// Abstraction function:
// AF(vertices) = 图中所有顶点及顶点间有向边信息

// Representation invariant:
/**
 * 1) vertices中无重复点
 * 2) 出边和入边数目相同
 * 注：部分RI在Vertex类中检测
 */

// Safety from rep exposure:
/**
 * 1) vertices声明为private final
 * 2) 无返回vertices的方法
 */
```

- 3) 实现各方法，主要思路：除 toString()方法外，均根据 AF、RI、Safety from rep exposure 及 Graph.java 文件中的规约来实现具体的方法，在源代码文件 ConcreteVerticesGraph.java 中有较详细的注释说明，对于 toString()方法，实现与 ConcreteEdgesGraph 中类似。故代码此处略去；

- 4) 编写测试策略，设计测试用例：




测试策略如下（与 ConcreteEdgesGraph 类似）：

```
// Testing strategy for ConcreteVerticesGraph.toString()
// 使用几个合法的简单点图检查返回的字符串是否正确
// 空图、有顶点无边图、有顶点有边图
```

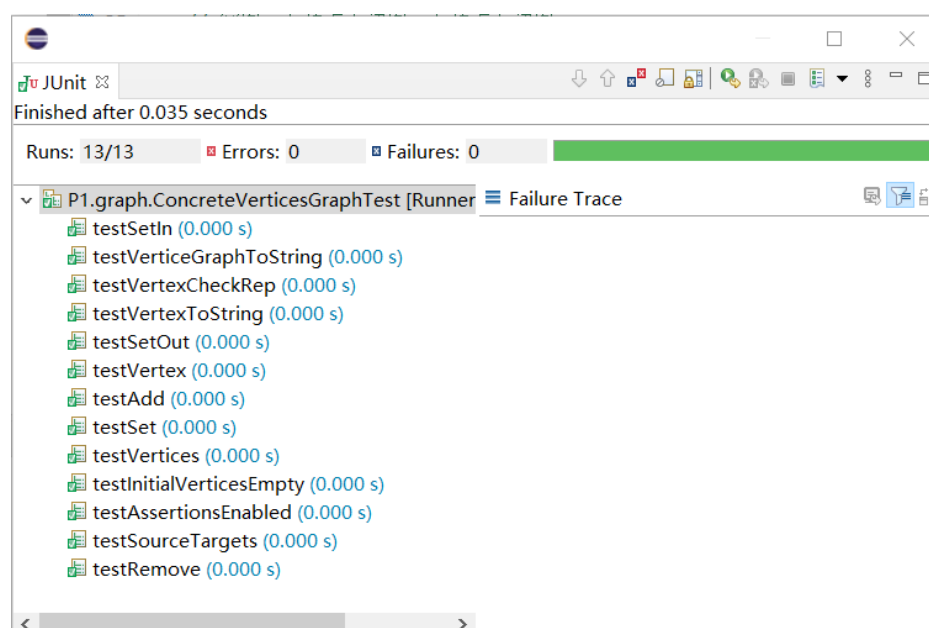
具体测试代码略。

最后运行测试，得到的结果如下图：

代码覆盖率情况：

ConcreteVerticesGraph.java		97.4 %
ConcreteVerticesGraph		95.1 %
Vertex		100.0 %

测试用例通过情况：



3.1.4 Problem 3: Implement generic Graph<L>

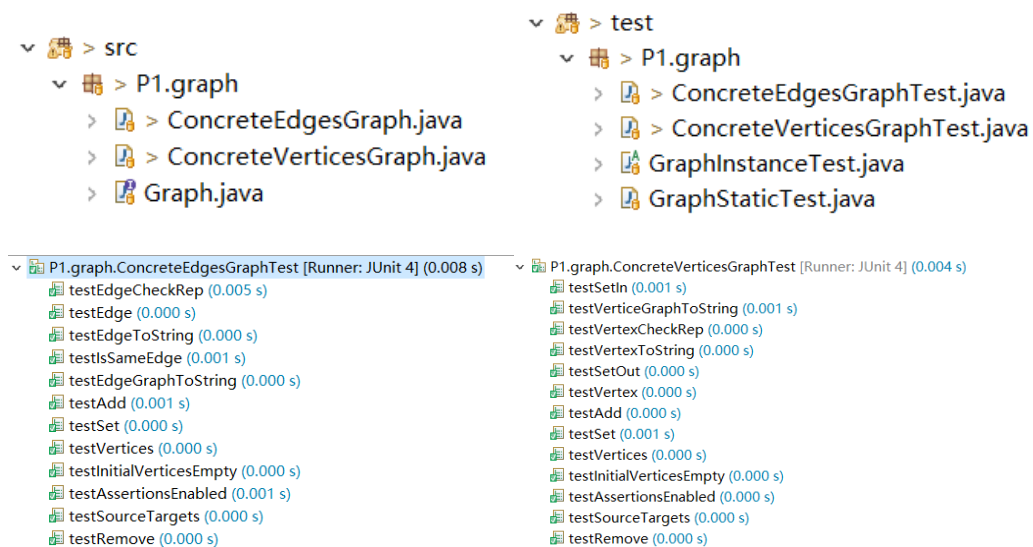
3.1.4.1 Make the implementations generic

要求将之前实现的 ConcreteEdgesGraph、ConcreteVerticesGraph 及相关的 Edge 类和 Vertex 类转换为泛型实现。

按照实验要求，先修改几个类的声明，随后可按照 Eclipse IDE 的错误提示进行对应的修改。

首先修改声明和私有属性，随后按照错误提示修改类内部的类型，最后按照警告提示修改对应的测试文件，具体过程略去，可查看 GitHub 仓库中描述为“P1 Problem 3.1 completed.”的提交中的代码变化。

完成全部修改后，所有相关源代码文件均不再显示警告或错误，且能通过测试，如下图：



3.1.4.2 Implement Graph.empty()










这里我选择 ConcreteEdgesGraph 供客户端使用，首先实现 Graph.empty()，如下图：

```
public static <L> Graph<L> empty() {
    return new ConcreteEdgesGraph<L>();
}
```



随后在 GraphStaticTest.java 中补充对不同 immutable 类型 L 的测试，这里我添加了对 Integer 和 Character（分别是 int 和 char 的封装对象类型）的测试。具体测试代码此处略，测试策略类似于简化版的 GraphInstanceTest，初始化相应类型的对象并进行了一些简单的方法操作，检测结果是否正确。



最后运行测试，结果如下图：




代码覆盖率情况：

▼ P1.graph		97.1 %
▼ ConcreteEdgesGraph.java		96.8 %
> ConcreteEdgesGraph<L>		96.0 %
> Edge<L>		100.0 %
▼ ConcreteVerticesGraph.java		97.4 %
> ConcreteVerticesGraph<L>		95.1 %
> Vertex<L>		100.0 %
▼ Graph.java		100.0 %
> Graph<L>		100.0 %

测试用例通过情况:

 JUnit  Finished after 0.031 seconds

Runs: 29/29	 Errors: 0	 Failures: 0
-------------	---	---

- >  P1.graph.ConcreteEdgesGraphTest [Runner: JUnit 4] (0.004 s)
- >  P1.graph.ConcreteVerticesGraphTest [Runner: JUnit 4] (0.006 s)
- >  P1.graph.GraphStaticTest [Runner: JUnit 4] (0.001 s)

3.1.5 Problem 4: Poetic walks

该任务要求使用 `Graph` 类来实现 `GraphPoet` 类, 此类使用读入的语料库 (corpus) 初始化图结构, 并运用这一图结构搜索词之间的关系, 自动扩充输入的语句 (使之诗歌化)。

3.1.5.1 Test GraphPoet

这一步要求在 `GraphPoetTest.java` 中设计对于 `GraphPoet` 类的测试策略并编写测试用例。允许在 `GraphPoet` 类中添加额外的方法及加强所需方法的规约, 但不能修改所要求的方法的签名或减弱其规约强度。

测试策略如下:

```
// Testing strategy
/**
 * GraphPoet类功能较少且相互依赖, 故在同一测试函数中进行测试。
 * 使用不存在文件/空文件/一行内容文件/多行内容文件测试构造方法,
 * 使用toString()结果是否正确来判断构造方法正确性,
 * 通过无扩充/简单扩充 (仅有一个可能的bridge word) /较复杂扩充 (多个位置, 多选择) 检测poem()正确性。
 */
```

具体测试代码略。

3.1.5.2 Implement GraphPoet

这一步要求实现 `GraphPoet` 类, 我按照实验要求实现了类中的各方法, 没有修改原来的规约, 也没有增加新的方法。具体过程如下:

1) 使用要求的私有属性:

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

2) AF、RI 及 Safety from rep exposure

```

// Abstraction function:
/**
 * AF(graph) = 从文件读取的语料库(corpus)
 * 图中顶点是单词，边是两个单词在文本中的相邻关系
 * 边A->B的权代表AB在文本中出现的次数
 */
// Representation invariant:
/**
 * 1) 顶点信息(单词)不能为空(null)
 * 2) graph对象不能为null
 */
// Safety from rep exposure:
/**
 * 将类中graph设置为private final
 * 不提供返回其引用或修改它的公共方法
 */

```

3) 实现各方法:

需实现的方法及简要思路:

- GraphPoet(): 构造函数，读取文件，构建图结构。文件操作与 Lab1 类似，使用 String 示例的 split 方法得到单词，将单词转换为小写表示，并对每对单词调用两次图结构的 set 方法来分别获取原权值和更新权值。一个关键点是两个单词不能相等，完成编程后测试时才发现问题，这样会违反图的 RI 从而导致程序异常终止。

具体代码如下:

```

public GraphPoet(File corpus) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(corpus));
    String line; // 暂存读到的行
    String[] words; // 暂存一行按空格分割后的一行内容(单词)
    List<String> wordList = new ArrayList<>();
    while ((line = br.readLine()) != null) {
        words = line.split(" "); // 空格分割
        for (String w : words) {
            wordList.add(w.toLowerCase()); // 转换为小写
        }
    }
    br.close();
    for (int i = 0; i < wordList.size() - 1; ++i) {
        String left = wordList.get(i);
        String right = wordList.get(i + 1);
        if (left.equals(right)) // 避免在始点与终点相同
            continue;
        int before = graph.set(left, right, 0); // 获取原边权或添加顶点并返回0
        graph.set(left, right, before + 1); // 更新边权
    }
    checkRep();
}

```

- checkRep(): 检查不变性;
- poem(): 生成新文本(poetry)，利用图结构的 sources 和 targets 方法，查找输入文本中每两个相邻单词间可能的 bridge word，并在有多个可能的 bridge word 时选择最早出现、边权最大的一个。核心部分代码如下:
(见下页图)

```

for (int i = 0; i < wordList.size() - 1; ++i) {
    sb.append(wordList.get(i)).append(" "); // 加入输入的一个单词
    // 转换为小写便于查找图中两点间bridge word
    String source = wordList.get(i).toLowerCase();
    String target = wordList.get(i + 1).toLowerCase();
    targetMap = graph.targets(source); // 得到前一个点的所有终点
    sourceMap = graph.sources(target); // 得到后一个点的所有源点
    // 在这两个map中查找边权最大的公共点即为bridge word
    int max = 0;
    String bridgeWord = "";
    for (String tar : targetMap.keySet()) {
        if (sourceMap.containsKey(tar) && sourceMap.get(tar) + targetMap.get(tar) > max) {
            max = sourceMap.get(tar) + targetMap.get(tar);
            bridgeWord = tar;
        }
    }
    if (max > 0) {
        sb.append(bridgeWord + " "); // 假如找到则加入
    }
}
sb.append(wordList.get(wordList.size() - 1)); // 加入最后一个单词

```

- toString(): 将类信息转换为可读字符串返回, 主要使用了 graph 类的 toString 方法。

注: 具体实现见源代码文件 GraphPoet.java。

- 4) 进行测试:
测试结果如下:

Runs: 2/2 Errors: 0 Failures: 0

▼ P1.poet.GraphPoetTest [Runner: JUnit 4] (0.000 s)

- testGraphPoet (0.000 s)
- testAssertionsEnabled (0.000 s)

3.1.5.3 Graph poetry slam

在 Main.java 中添加了一个简单的例子, 如下图:

// 以下为自行添加的一个简单测试

```

final GraphPoet myPoet = new GraphPoet(new File("src/P1/poet/lines.txt"));
final String myInput = "It the of software construction";
System.out.println(myInput + "\n>>>\n" + myPoet.poem(myInput));

```

运行结果如下图所示:




```




Test the system.
>>>
Test of the system.
It the of software construction
>>>
It was the season of software construction

```

3.1.6 使用 Eclemma 检查测试的代码覆盖度

运行 P1 部分的所有测试, 代码覆盖度结果如下:

▼ P1.poet		84.1 %
> Main.java		0.0 %
> GraphPoet.java		96.7 %

▼ P1.graph		97.4 %
> ConcreteEdgesGraph.java		96.7 %
> ConcreteVerticesGraph.java		98.0 %
> Graph.java		100.0 %

注: Main.java 是客户端代码, 不进行 JUnit 测试, 可直接运行输出结果。

3.1.7 Before you're done

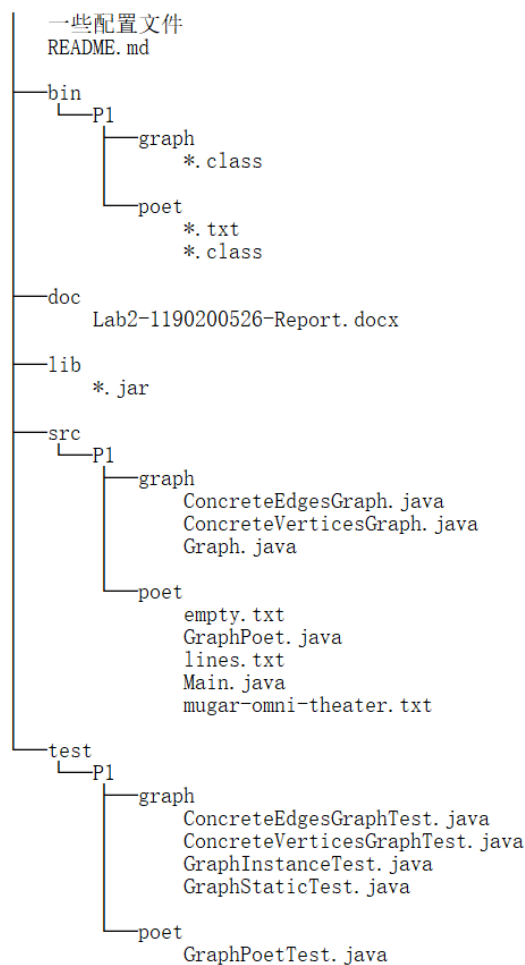
请按照 http://web.mit.edu/6.031/www/sp17/psets/ps2/#before_youre_done 的说明, 检查你的程序: 2021.6.1 完成检查及相应的代码修改完善工作。

如何通过 Git 提交当前版本到 GitHub 上你的 Lab2 仓库:

```
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-119020052
6 (master)
$ git add .
warning: LF will be replaced by CRLF in src/P1/graph/ConcreteEdgesGraph.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/P1/graph/ConcreteVerticesGraph.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/P1/poet/GraphPoet.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/P1/poet/Main.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/graph/ConcreteEdgesGraphTest.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/graph/ConcreteVerticesGraphTest.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/poet/GraphPoetTest.java.
The file will have its original line endings in your working directory
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-119020052
6 (master)
$ git commit -m "P1 completed"
[master a56025a] P1 completed
10 files changed, 169 insertions(+), 50 deletions(-)
create mode 100644 src/P1/poet/empty.txt
create mode 100644 src/P1/poet/lines.txt
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-119020052
6 (master)
$ git push
Enter passphrase for key '/c/Users/86189/.ssh/id_rsa':
Enumerating objects: 37, done.
Counting objects: 100% (37/37), done.
Delta compression using up to 8 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (21/21), 155.87 KiB | 457.00 KiB/s, done.
Total 21 (delta 7), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (7/7), completed with 6 local objects.
To github.com:ComputerScienceHIT/HIT-Lab2-1190200526.git
c0b037b..a56025a master -> master
```

在这里给出你的项目的目录结构树状示意图。

(见下页)



3.2 Re-implement the Social Network in Lab1

该任务要求基于在 3.1 节 Poetic Walks 中定义的 $\text{Graph}\langle L \rangle$ 及其两种实现（这里的 L 即为 `Person`），重新实现 Lab1 中 3.3 节的 `FriendshipGraph` 类，并通过 Lab1 中编写的测试用例。

3.2.1 FriendshipGraph 类

我选择使用 3.1 中实现的边图来完成 `FriendshipGraph` 类的实现。
补充了 AF、RI 和 Safety from rep exposure:

```

// Abstraction function:
/**
 * graph代表社交网络图
 * 其中顶点为社交网络中的人
 * 边为人之间的关系，边权默认均为1
 */

// Representation invariant:
// 按照实验要求，沿用ConcreteEdgeGraph的RI

```



```
// Safety from rep exposure:
/**
 * 将graph设置为private final
 * 仅能通过公共方法对其进行修改
 * 无返回其引用的方法
 */
```

需要实现的方法的思路及具体实现如下：

1) addVertex():

```
public void addVertex(Person p) {
    // 直接调用Graph类的add()方法
    graph.add(p);
}
```

2) addEdge():

```
public void addEdge(Person p1, Person p2) {
    // 直接调用Graph类的set方法来设置有向边
    graph.set(p1, p2, 1);
}
```

3) getDistance():

使用与 Lab1 实现相同的 BFS 思路，只需调整队列的元素类型、记录顶点是否访问过的方式和循环体，具体实现如下：

```
if (p1 == p2)
    return 0;
if (!graph.vertices().contains(p1) || !graph.vertices().contains(p2)) {
    throw new RuntimeException("Error:At least one vertex is not in the graph!");
}
```

注：以上部分检查两点是否相同或有一点不在图中。

```
// 使用广度优先搜索的思想求单点对最短路径
Queue<Person> queue = new LinkedList<Person>(); // 队列用于广度优先搜索
Map<Person, Integer> disMap = new HashMap<>(); // 保存中间结果并用于顶点判断是否访问过
Person curP = p1;
Person dstP = p2;
queue.add(curP);
disMap.put(curP, 0);
// BFS过程
while (!queue.isEmpty()) {
    curP = queue.poll();
    int curDis = disMap.get(curP);
    Map<Person, Integer> nextPList = graph.targets(curP);
    for (Person p : nextPList.keySet()) {
        if (!disMap.containsKey(p)) {
            disMap.put(p, curDis + 1);
            if (p == dstP) { // 已经找到目标点并计算出距离
                queue.clear();
                break;
            }
            queue.add(p);
        }
    }
}
```

注：以上部分是该方法的核心，运用 BFS 求距离。

```
// 两点之间无通路
if (!disMap.containsKey(dstP))
    return -1;
return disMap.get(dstP);
```

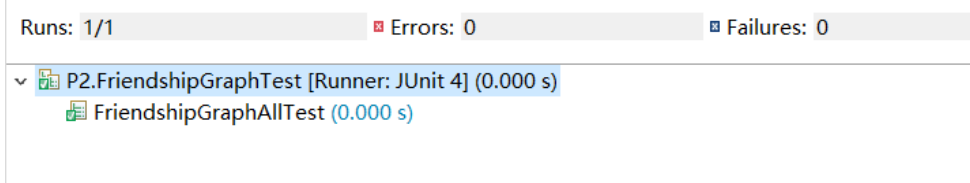
注：以上部分为结果的返回，若 BFS 不可达，disMap 中则无对应记录，返回-1，若可达，输出 disMap 中记录的结果。

至此, 要求的三个函数已经全部实现, 运行客户端 `main()` 代码, 结果与预期结果一致, 如下图:

```
System.out.println(graph.getDistance(rachel, ross));  
// should print 1  
System.out.println(graph.getDistance(rachel, ben));  
// should print 2  
System.out.println(graph.getDistance(rachel, rachel));  
// should print 0  
System.out.println(graph.getDistance(rachel, kramer));  
// should print -1
```

```
1  
2  
0  
-1
```

JUnit 测试用例通过, 结果如下图:



3.2.2 Person 类

实现与 Lab1 相同, 定义一个私有属性 `name` 保存人的姓名, 构造函数初始化姓名, 公共 `getName()` 方法获取姓名:

```
public class Person {  
  
    /* 私有属性 */  
    private final String name;  
  
    /* 构造函数 */  
    public Person(String name) {  
        this.name = name;  
    }  
  
    /**  
     * 获取姓名  
     *  
     * @return 姓名  
     */  
    public String getName() {  
        return name;  
    }  
}
```

3.2.3 客户端 main()

保留了 Lab1 中原有的客户端代码, 运行结果与预期一致 (见 3.2.1 节)。

3.2.4 测试用例

基本沿用了 Lab1 中的测试用例, 但对于 addVertex()和 addEdge()方法, 新实现不再抛出异常 (在 Graph 类中进行了检测), 故删去了对这些非法抛出异常的测试 (根据 Lab1 中 CMU 的实验指导, 对于违反 spec 的非法情况的处理可以自行决定, 故这样的修改不会影响程序正确性)。

具体测试代码见 FriendshipGraphTest.java。

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。

过程与 3.1.7 节类似 (git add + git commit -m “描述” + git push), 如下图:

第一步:

```
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-1190200526 (master)
$ git add .
warning: LF will be replaced by CRLF in test/P1/graph/ConcreteEdgesGraphTest.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/graph/ConcreteVerticesGraphTest.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/graph/GraphInstanceTest.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test/P1/graph/GraphStaticTest.java.
The file will have its original line endings in your working directory
```

查看暂存情况:

```
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-1190200526 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   doc/Lab2-1190200526-Report.docx
    new file:   src/P2/FriendshipGraph.java
    new file:   src/P2/Person.java
    modified:   test/P1/graph/ConcreteEdgesGraphTest.java
    modified:   test/P1/graph/ConcreteVerticesGraphTest.java
    modified:   test/P1/graph/GraphInstanceTest.java
    modified:   test/P1/graph/GraphStaticTest.java
    new file:   test/P2/FriendshipGraphTest.java
```

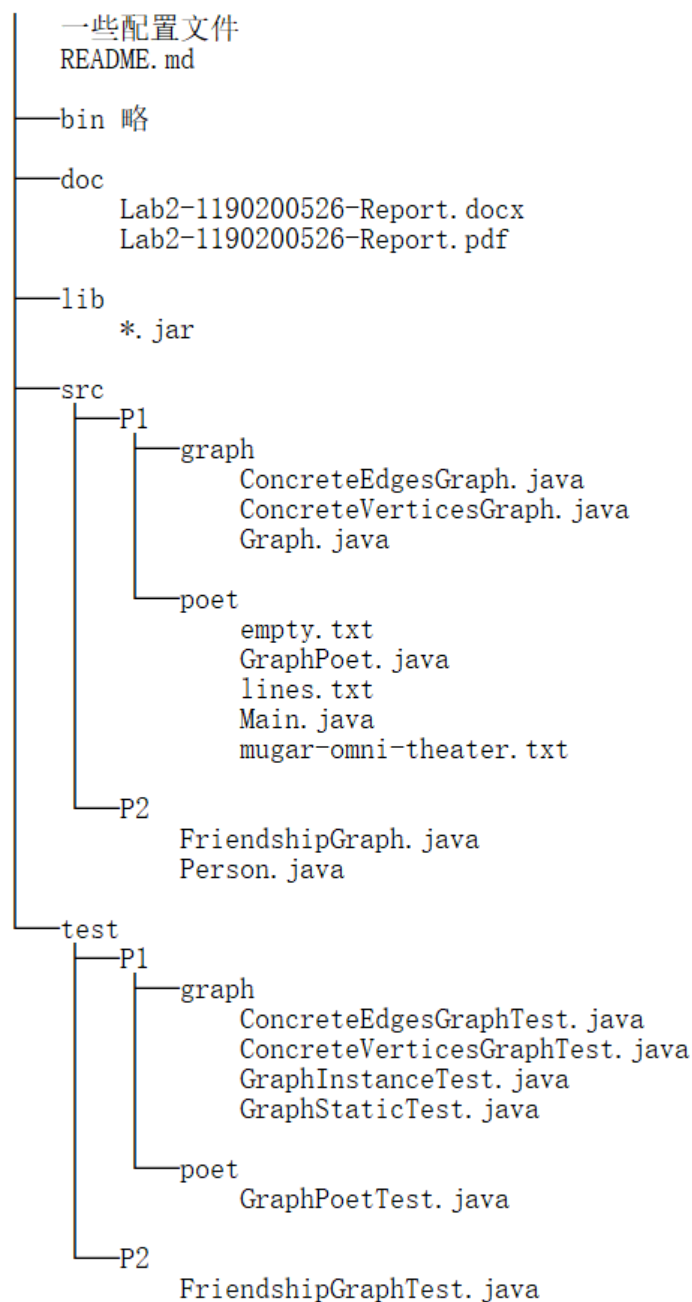
第二步:

```
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-1190200526 (master)
$ git commit -m "P2 completed."
[master d5abb2a] P2 completed.
 8 files changed, 225 insertions(+), 6 deletions(-)
 create mode 100644 src/P2/FriendshipGraph.java
 create mode 100644 src/P2/Person.java
 create mode 100644 test/P2/FriendshipGraphTest.java
```

第三步：

```
86189@DESKTOP-CUTTINGEDGE191 MINGW64 /d/Code/Eclipse/HITCS-SC/HIT-Lab2-1190200526 (master)
$ git push
Enter passphrase for key '/c/Users/86189/.ssh/id_rsa':
Enumerating objects: 27, done.
Counting objects: 100% (27/27), done.
Delta compression using up to 8 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (17/17), 233.85 KiB | 760.00 KiB/s, done.
Total 17 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), completed with 5 local objects.
To github.com:ComputerScienceHIT/HIT-Lab2-1190200526.git
 a56025a..d5abb2a master -> master
```

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
2021.5.25	18:00 - 20:00	完成 P1 Problem 1	按计划完成
2021.5.27	14:30 - 15:30	初步设计 P1 ConcreteEdgesGraph	按计划完成
2021.5.27	18:30 - 22:30	完成 P1 ConcreteEdgesGraph 的编程及对应部分的报告	按计划完成
2021.5.29	14:40 - 17:00	完成 P1 ConcreteVerticesGraph 的编程	按计划完成
2021.5.29	19:00 - 20:40	完成 P1 Problem 2 所有内容	按计划完成
2021.5.30	11:50 - 12:40	完成 P1 Problem 3 的 3.1	按计划完成
2021.5.30	14:40 - 15:30	完成 P1 Problem 3 所有内容	按计划完成
2021.5.31	16:00 - 17:20	完成 P1 Problem 4 的 4.1	按计划完成
2021.5.31	18:00 - 20:30	完成 P1 全部内容，完善代码	按计划完成
2021.6.1	15:00 - 17:00	修改 P1 部分代码，完善报告，完成 P2	按计划完成
2021.6.1	17:50 - 18:30	总结实验，在线构建，提交	按计划完成

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
实验初期对 AF、RI 和表示泄露等概念理解不清，不知道如何说明和编写相关代码。	课堂听课、课后复习课件并与其他同学探讨。
不适应测试优先的编程方式，在没有具体实现前不知如何设计测试。	只能在完成实验的过程中逐渐适应。现在能做到根据规约设计测试用例，再进行具体实现，但测试有时还要进行调整和补充。
不了解如何设计实现 checkRep() 检查 RI。	在实验课上请教老师。

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

1. 深入理解了 AF、RI、表示泄露等概念；
2. 理解了设计 ADT、将 ADT 泛型化的过程；
3. 学会了如何避免表示泄露及如何检查表示不变性；
4. 认识到养成测试优先编程习惯的重要性和意义，锻炼了编写规约及根据规约设计测试用例的能力；
5. 学会了如何检查和提高代码覆盖度及如何使用 ADT 进行简单的 OOP 编程。

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?

答: 面向 ADT 的编程更底层、更抽象、更通用化, 不如直接面向应用场景编程具体, 但 ADT 是面向应用场景编程的基础。

- (2) 使用泛型和不使用泛型的编程, 对你来说有何差异?

答: 总体上差异不是很大, 但泛型更加灵活, 有时也要考虑更多情况。

- (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?

答: 优势是能增强编程的目的性, 更好地保证 ADT 在编写过程中的正确性, 减少错误和漏洞积累, 便于及时修正和调整; 个人不是很适应这种测试方式, 会在今后的学习和实践中继续适应。

- (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?

答: 减少重复开发的工作量, 提高开发效率。

- (5) 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?

答: 意义是提高程序的正确性、健壮性和安全性, 保证程序质量, 避免编程的实际工作脱离目标, 且便于用户使用; 我愿意在以后编程中坚持这样做, 因为这对于程序开发、维护和多人协作具有重要作用。

- (6) 关于本实验的工作量、难度、deadline。

答: 工作量适中, 难度适中, deadline 较晚也保证了比较充足的时间。

- (7) 《软件构造》课程进展到目前, 你对该课程有何体会和建议?

答: 收获了很多 Java 编程的相关知识, 认识到编写程序与实际开发的巨大差异, 学习到了很多实际开发软件的重要知识、方法和技能。建议增强课件及课堂讲解内容的逻辑性、层次性, 便于同学们进行预习和复习。