2024 年春季学期软件构造

Lab1 实验报告

班号	2211107	学号	2022212029	姓名	唐炜堤
开始	2024.5.25	截止	2024.6.7 第十四周周五 17:00		

目录

1. 实验目标概述 1
2. 实验环境配置 1
3. 实验过程（Poetic Walks） 1
3.1 Problem 1: Test Graph <String> 1
3. 2 Problem 2: Implement Graph <String> 3
3.2.1 Implement ConcreteEdgesGraph 3
3.2.2 Implement ConcreteVerticesGraph 7
3.3 Problem 3: Implement generic Graph<L> 10
3.3.1 Make the implementations generic 10
3.3.2 Implement Graph.empty() 10
3.4 Problem 4: Poetic walks 11
3.4.1 Test GraphPoet 11
3.4.2 Implement GraphPoet 12
3.4.3 Graph poetry slam 14

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：

（1） 针对给定的应用问题，从问题描述中识别所需的 ADT；
（2） 设计 ADT 规约（pre-condition、post-condition）并评估规约的质量；
（3） 根据 ADT 的规约设计测试用例；
（4） ADT 的泛型化；
（5） 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）；
（6） 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）；
（7） 测试 ADT 的实现并评估测试的覆盖度；
（8） 使用 ADT 及其实现，为应用问题开发程序；
（9） 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

（1） 安装 JDK21，并配置环境变量
（2） 安装 JetBrain IDEA
（3） 使用 Maven 创建项目
（4） 这里给出 GitHub 上 Lab1 实验的 URL：
https://github.com/2364058719/Lab1_2022212029.git

3. 实验过程（Poetic Walks）

本题需要编写一个实现图 Graph 的数据结构类，支持添加顶点、边，查询以某个顶点为起点/终点的终点集合、起点集合以及权重。然后使用该数据结构编写 GraphPoet 类，用来对输入的语句进行扩展。

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惯。并且在后面练习 ADT 的泛型化。

3.1 Problem 1: Test Graph <String>

1、主要是测试 Graph.empty（）函数的 testing stratrgy。其中，Graph.empty（）是一个返回一个空的 Graph<L>实现，所以此处的测试主要是测试在不同的 L 的情况下，只要 L 为 immutable 类型的数据就可以使用。所以测试策略就是使用不同的 immutable 类型的数据，此处选择 Integer 和 Long 以及接下来会使用的 String 进行测试，保证测试包括 set、add、remove、Vertices、等函数即可。

2、书写测试 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，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

Test add	/* Partition for inputs of graph.add(input) * graph: empty graph, graph with

	vertices ** input: new vertices, vertices existed in graph already* **/*
Test remove	*/* Partition for inputs of graph.remove(input)* ** graph: empty graph, graph with vertices* ** input: new vertices, vertices existed in graph already* ** without edges, vertices exited with edges.* **/*
Test set	*/* Partition for inputs of graph.set(source, target, weight)* ** graph: empty, graph with vertices* ** source: new vertex, vertex existed in graph already.* ** target: new vertex, vertex existed in graph already.* ** weight: 0, positive.* ** edge: new edge, edge existed in graph already.*
Test vertices	*/* Partition for graph.vertices()* ** graph: empty graph, graph with vertices.* **/*
Test source	*/* Partition for inputs of graph.source(input)* ** graph: empty graph, graph with vertices* ** input: new vertex, vertex without any edge point to,* ** vertex with edges point to* **/*
Test target	*/* Partition for inputs of graph.target(input)* ** graph: empty graph, graph with vertices* ** input: new vertex, vertex*

	without any edge start with, ** vertex with edges start with.* **/*

3. 2 Problem 2: Implement Graph <String>

这个部分要求实现 Graph<String>接口的两个具体实现，分别基于边为主和点为主来实现图的存储操作。并且实现 Abstraction function 和 Representation invariant 的记录，以及在每一个实现里面书写 checkRep 和重写 toString 函数。

3.2.1 Implement ConcreteEdgesGraph

本题要求实现以边存储图，具体来说，提供点集提供点集 `Set<String> vertices` 和边表 `List<Edge<String>> edges` 这两个 `rep`。

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

而且必须将 Edge 写成 Immutable 类型的类。

1、Edge 的 RI, AF 等如下所示：

```
class Edge<L>{

    // TODO fields
    private final L source;
    private final L target;
    private final int weight;

    // Abstraction function:
    // TODO
    // Represents a directed edge with weight from source vertex to
target vertex
    // that is different from the source.

    // Representation invariant:
    // TODO
    // Source is different from target and neither of them is empty
string.
    // Weight is an integer that is nonnegative.
    // Source != null, target != null.

    // Safety from rep exposure:
    // TODO
    // Each one of the fields are modified by keyword private and
final,
    // so they can't be accessed from outside the class or be
reassigned.
}
```

在接下来的方法中，不在 Edge 类中定义其余的 setter 方法，只在构造方法中一次

定义 source、target、weight 三个的值，而且由于 fields 中没有对象数据类型对象的使用，所以这个 Edge 是一个 immutable 类，避免了 Rep exposure。

另外需要设计 Edge 类中的 AF、RI 以及 Safety from Rep exposure. 对于 Abstraction function，Edge 类对应的即是“一个从 source 到 target 的带有 weight 权重的有向边”。

对于 Representation invariant 表示不变性，我们保证每一个 Edge 对象的 source 和 target 都不是空字符串或者 null，并且 weight 是一个非负数且允许自环的存在。

对于 Rep exposure 的安全性，由于我们使用的是 Private 和 final 关键字修饰的 filed，并且没有使用对象数据类型的 filed 成员和添加已有成员的 setter 方法，故外部用户无法修改内部的实现，保证数据不会外泄。基于以上几点，书写 checkRep 函数如下：

```
// TODO checkRep
    private void checkRep() {
        assert weight >= 0;
        assert source != null;
        assert target != null;
//        assert !source.equals(target);
        // 有没有自环?
//        assert !source.equals("");
//        assert !target.equals("");
        // 允许空串?
    }
```

2、ConcreteEdgesGraph 的 RI, AF 等如下所示：

```
public class ConcreteEdgesGraph<L> implements Graph<L> {
    private final Set<L> vertices = new HashSet<>();
    private final List<Edge<L>> edges = new ArrayList<>();

    // Abstraction function:
    // TODO
    // Represents a set of weighted directed edges from a source
    // vertex to a target vertex that is different from source in a
directed graph.

    // Representation invariant:
    // TODO
    // The weight of edge in edges is a positive integer.
    // Each vertex of edge in edges should also be in vertices set.
    // There is at most one directed edge between any source and
target.

    // Safety from rep exposure:
    // TODO
    // Each one of the fields are modified by keyword private and
final,
    // so they can't be accessed from outside the class or be
```

```
reassigned.
    // We've done some defensive copying in the return value of
vertices and edges.
}
```

在 ConcreteEdgesGraph 中写对于 Edge 类中的操作的测试，在 Edge 类中主要定义了四种特有操作和重写的 toString 方法。四种方法是 getSource、getTarget、getWeight 以及 cloneEdge，equals 和 hashCode 方法。 然后在 ConcreteEdgesGraph 中除了 hashCode 和 equals 之后书写 testing strategy 如下。

```
// Testing strategy for Edge
//   TODO
// Partition for edge.getSource()
//     has only one input, edge
//     has only one output, source
//
// Partition for edge.getTarget()
//     has only one input, edge
//     has only one output, target
//
// Partition for edge.getWeight()
//     has only one input, edge
//     has only one output, weight
//
// Partition for edge.cloneEdge()
//     has only one input, edge
//     has only one output, an Edge instance that has the same
fields with input
//
// Partition for edge.toString()
//     has only one input, edge
//     has only one output, edge.getSource() -> edge.getTarget() :
edge.getWeight()
```

以上的函数，由于没有参数，所以测试只有一种输出，测试的划分也十分简单。只需要测试也只需测试是否有合法的输出即可。

但对于 hashCode 和 equals 方法，测试策略书写如下：

```
// Partition for edge.equals(input)
//      input equal to edge, input not equal to edge
//      output true if input equal to edge, otherwise false
//
// Partition for edge.hashCode()
//      has only one input, edge
//      has only one output, the hash code of edge
```

而对于上面的由于 equals 函数可能有不同的输入，所以需要测试两种，相同边和不同边的测试。

3、完成以上步骤之后，接下来去完成 Edge 类和 ConcreteEdgesGraph 类。

对于 Edge 类函数实现比较简单，主要在于 equals 函数，我们认为两个有向边相等

只有且仅有 source、target 和 weight 都分别相等才可以相等。 对于 ConcreteEdgesGraph 主要是实现 Graph<L>接口里面的函数

①add 函数

直接利用 Set 中已有的 add 函数直接调用即可，便可以保证每一次调用后如果新增的点不在集合内，便可以返回 true 并将点加入集合，否则返回 false。

②set 函数

主要注意一点，每一次利用 set 函数都需要返回上一次的这个有向边的权值。而且由于 Edge 类是 immutable 类型，所以在更改的时候不能直接修改，而是应该选择将已有的 Edge 的对象删除，转而增加新的带有需要权值的 Edge 对象。

③remove 函数

在删除该点的时候，需要同时注意删除这个顶点所邻接的所有边。而且对于并未出现在 vertices 集合中的顶点应该返回 false 表示删除失败。

④vertices 函数

只需要返回一个新的 Set 对象，在里面包含所有顶点，保证 rep 不会被外部的 Clients 所使用。

⑤sources 函数

找到以 target 为 destination 的所有有向边的集合，并返回一个包括 source 和有向边权值对应关系的 Map 即可，遍历一遍边集找到所有符合条件的 Edge 加入 Map 后返回。

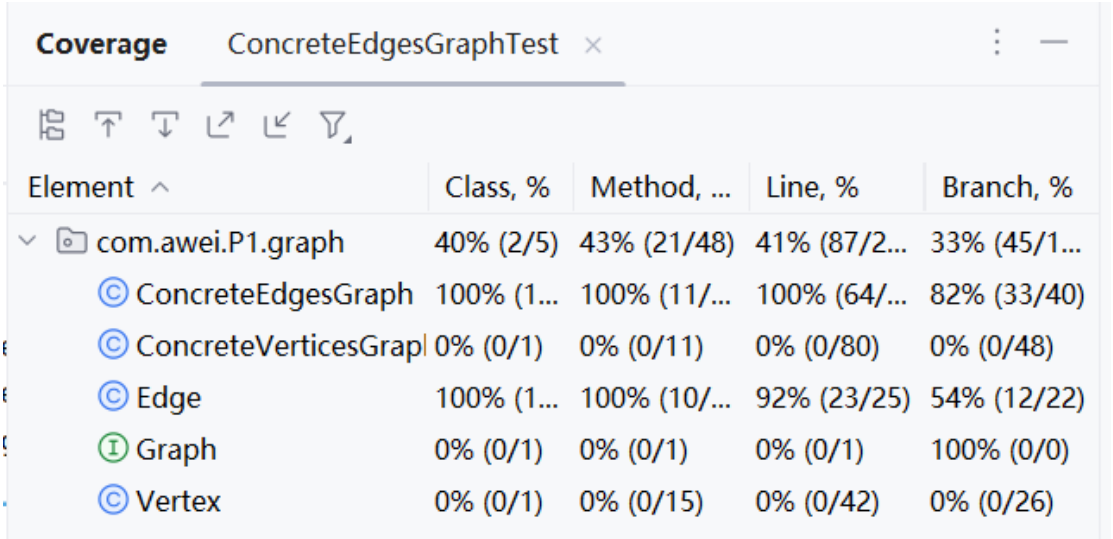
⑥target 函数

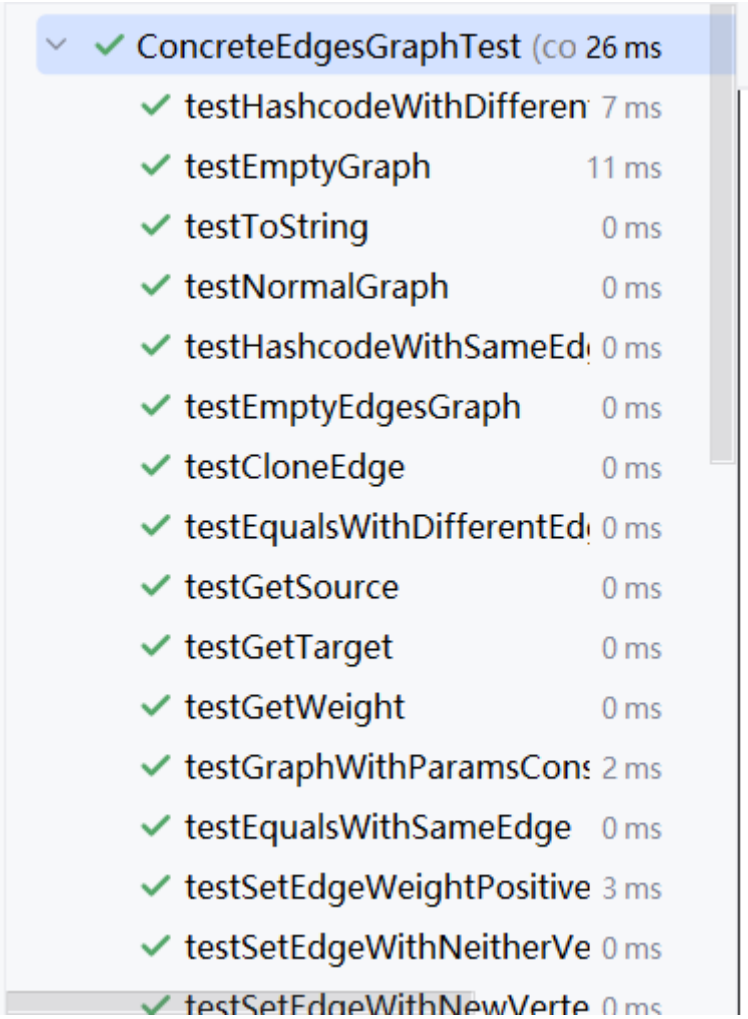
与 source 函数相对应，找到以 source 为源点的所有有向边的集合，并返回一个包括 target 和有向边权值对应关系的 Map 即可，同样遍历一遍边集找到所有符合条件的 Edge 加入 Map 后返回。

⑦toString 函数

Override 继承 Object 的 toString 函数，重写后的 toString 函数要尽量符合人的阅读习惯，所以返回的 String 中包含这个图中边数、顶点数、边集和顶点集中各个元素，并且排版尽量友好。

4、进行 ConcreteEdgesGraphTest 测试





3.2.2 Implement ConcreteVerticesGraph

必须使用下面的数据结构并且不能增加新的的的 fields

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

将 Vertex 设计成 mutable 类，需要有 2 个 field 分别为 label（表示这个 Vertex）和以这个顶点开始的有向边的集合一个 Map。并且由于是可变对象，所以需要增加一些 setter 方法和一些辅助方法。

1、Vertex 的 RI, AF 等如下所示：

```
class Vertex<L>{

    // TODO fields
    private final L name;
    private final Map<L, Integer> adjacent = new HashMap<>();
    // Abstraction function:
    // TODO
    // Represents a vertex which is one of two vertices of a
directed edge that
    // is from this to the other vertex with weight.

    // Representation invariant:
```

```
    // TODO
    // name != null.
    // Value in adjacent.getValue() is positive.
    // Target in adjacent.getKey() is not null or empty string.

    // Safety from rep exposure:
    // TODO
    // All fields are private and final.
    // We've done some defensive copying in the return value of
adjacent.
}
```

需要设计 Vertex 类中的 AF、RI 以及 Safety from Rep exposure. 对于 Abstraction function,Vertex 类对应的即是“在一条带有权值的有向边中的 Source 点”

在 Representation invariant 表示不变性中，Vertex 中的 name 不能是 null 而且在以其为 source 的边中不能出现 key 值为空的 entry。

在 Rep exposure 的防备下，所有的 filed 中的元素都使用了 Private 和 final 修饰,从外部不能直接接触到 field,并且在所有的需要返回 map 的地方均使用防御式拷贝，保证 safety from rep exposure。

基于以上已有的要求，设计 checkRep 如下：

```
// TODO checkRep
    private void checkRep() {
        assert name != null;
//        assert !name.isEmpty();
        if (!adjacent.isEmpty()) {
            for (Map.Entry<L, Integer> entry : adjacent.entrySet()) {
                assert !name.equals(entry.getKey());
                assert entry.getValue() > 0;
            }
        }
    }
```

2、ConcreteVerticesGraph 的 RI、AF 等如下所示：

```
public class ConcreteVerticesGraph<L> implements Graph<L> {
    private final List<Vertex<L>> vertices = new ArrayList<>();

    // Abstraction function:
    // TODO
    // Represents a set of weighted directed edges from a source
    // vertex to a target vertex that is different from source in a
directed graph.

    // Representation invariant:
    // TODO
    // Each vertex is different from each other.

    // Safety from rep exposure:
```

```
    // vertices is modified by keyword private and final.
    // so it can't be accessed from outside the class or be
reassigned.
}
```

3、在完成以上两步后，分别完成 Vertex 类和 ConcreteVerticesGraph 类

①add 函数

进行简单判断然后根据集合中是否已有需要加入的顶点。如果已有，则返回 false，否则将其加入并返回 true。

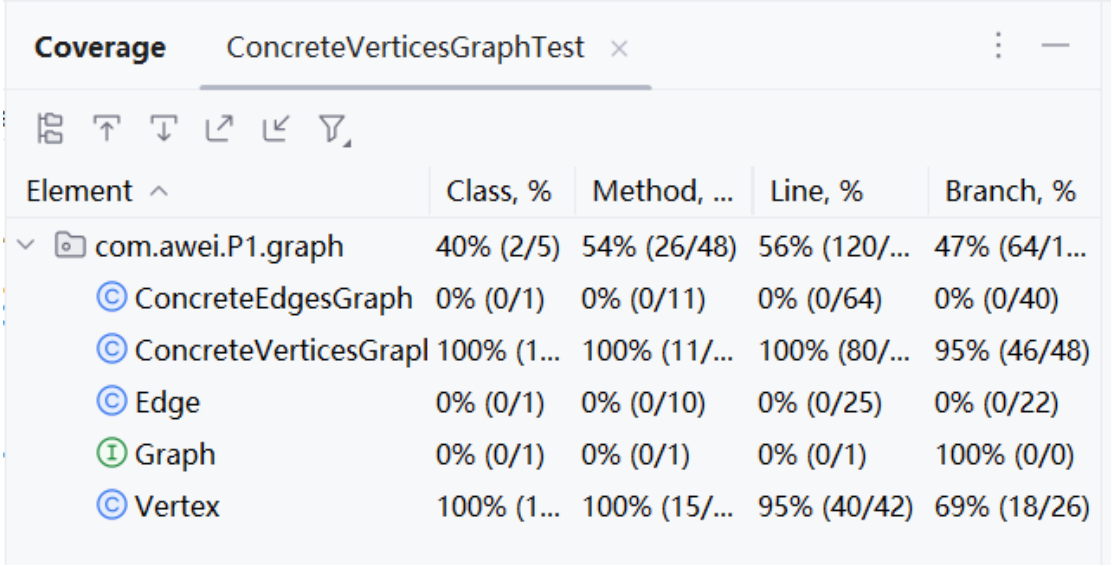
②set 函数

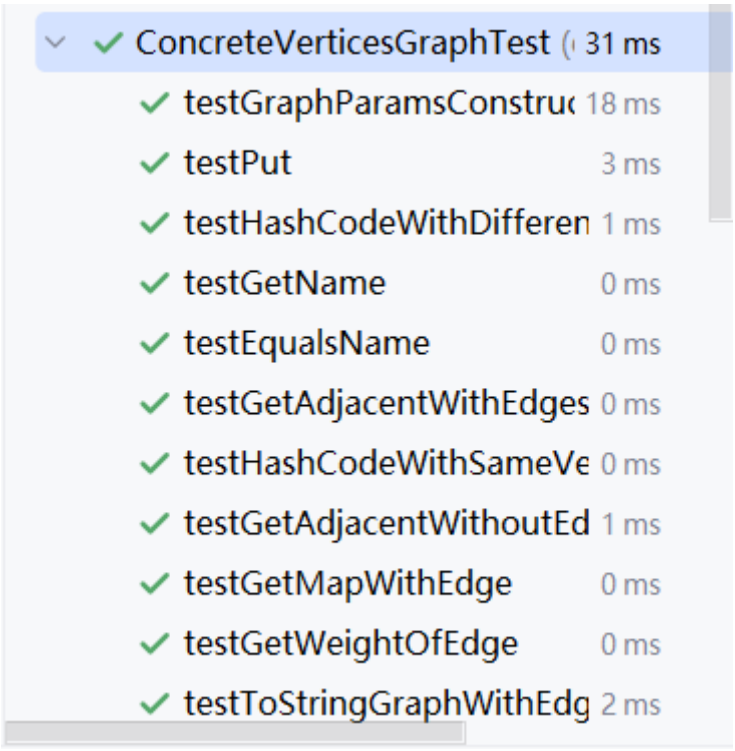
由于 Vertex 为 mutable 类型，所以与上面 ConcreteEdgesGraph 的写法不同，需要在设置为新的值之后，将已有的边直接进行更改即可。 ③remove 函数

注意在删掉顶点的时候需要将其邻接的边全部删除。 ④sources 函数和 targets 函数

在链表中扫一遍，将符合要求的对应关系加入返回的 map 中即可。

4、进行 ConcreteVerticesGraphTest 测试





3.3 Problem 3: Implement generic Graph<L>

本实验要求将 Graph 转换为真正的泛型数据结构。将 Graph<String>的实现改为基于 Graph<L>的实现。

3.3.1 Make the implementations generic

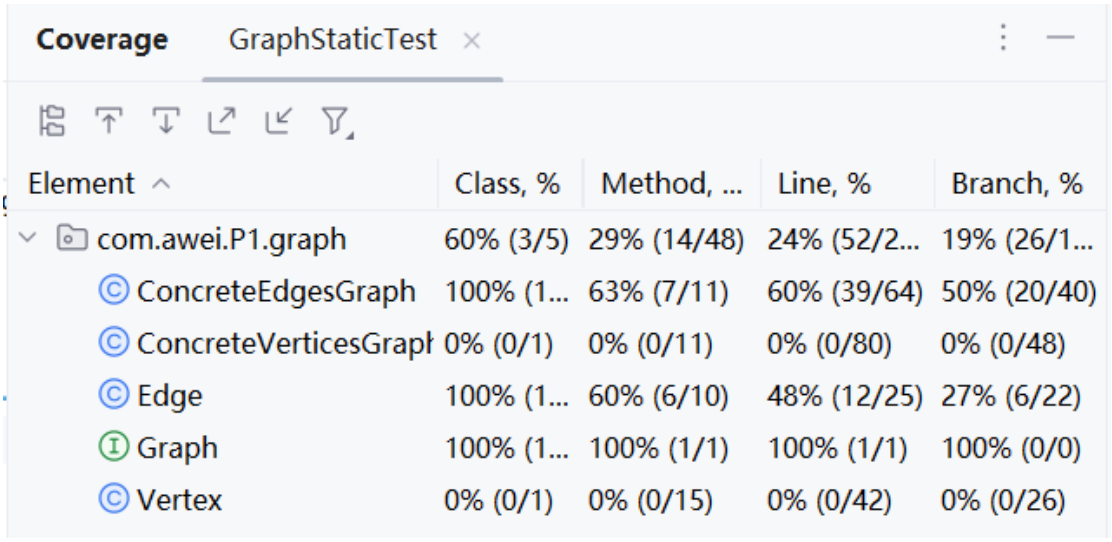
注意将所有的实现全部改为泛型实现即可，然后在更改结束后，重新测试 ConcreteVerticesGraphTest 和 ConcreteEdgesGraphTest 两个测试，可以测试通过 Graph<String>在两个泛型实现下仍然可以通过，表示更改成功。

3.3.2 Implement Graph.empty()

1、为了隐藏 Graph 的两个内部实现（生产环境中可能不需要这样做），Graph 的接口的静态 empty（）方法可以返回 Graph 的任意实现类的一个空实例，这里我们选择 ConcreteEdgesGraph。

```
/**
     * Create an empty graph.
     *
     * @param <L> type of vertex labels in the graph, must be
immutable
     * @return a new empty weighted directed graph
     */
    public static <L> Graph<L> empty() {
//      throw new RuntimeException("not implemented");
        return new ConcreteEdgesGraph<>();
    }
```

2、测试 GraphStaticTest



3.4 Problem 4: Poetic walks

“诗意之旅”这个任务主要利用已有的图的接口，根据一段诗生成其对应的图，然后根据这个生成的图再生成更多的诗，主要利用相同的搭配之间可能有的“bridge word”来修饰更改。

3.4.1 Test GraphPoet

测试 GraphPoet，主要是针对可能输入进行划分。在这里主要进行了以下的测试。主要测试的函数有 Constructor、poem、toString，采用“最少一次覆盖的策略进行策略”。

Construc tor 构造器	// Partition for constructor of GraphPoet // empty file, one line and several lines
Graph the poem 由诗所生成的图	// Partition for graph of poem // empty graph, a directed tree and a directed graph with rings
poem（in）	// Partition for graphPoet.poem(input) // empty string, one word, and several words
toString	// Partition for graphPoet.toString()

	`// empty graph, a directed tree and a directed graph with rings`

其中由于诗歌生成的图可能有多种情况，其中空图和有向树是比较容易处理的两种，而对于带环的有向图可能存在不同的 bridge word，需要在其中选择权值最大的一个。

测试文件如下所示：

```
# seven-words.txt
To explore strange new worlds
To seek out new life and new civilizations

# mugar-omni-theater.txt
This is a test of the Mugar Omni Theater sound system.
```

3.4.2 Implement GraphPoet

实现 GraphPoet 主要是实现 Constructor 和 Poem 两个函数。

1、对于 GraphPoet 的构造器

将文件中的诗歌按照行（hang）读入，利用空格将其分开，去掉空的字符串，在任何相邻的两个中间加入一条边，并按照边出现的数量更新权值建图即可。

```
/**
 * Create a new poet with the graph from corpus (as described
above).
 *
 * @param corpus text file from which to derive the poet's affinity
graph
 * @throws IOException if the corpus file cannot be found or read
 */
public GraphPoet(File corpus) throws IOException {
    BufferedReader in = null;
    in = new BufferedReader(new FileReader(corpus));
    List<String> list = new ArrayList<>();
    Map<String, Integer> map = new HashMap<>();
    String string;
    while ((string = in.readLine()) != null) {
        list.addAll(Arrays.asList(string.split(" ")));
    }
    in.close();
    Iterator<String> iterator = list.iterator();
    while(iterator.hasNext()){
        // remove empty string.
        if(iterator.next().length() == 0)
            iterator.remove();
    }
    for (int i = 0; i < list.size() - 1; i++) {
        String source = list.get(i).toLowerCase();
        String target = list.get(i+1).toLowerCase();
```

```
        int preweight = 0;
        if(map.containsKey(source+target)){
            preweight = map.get(source+target);
        }
        map.put(source+target, preweight+1);
        graph.set(source, target, preweight+1);
    }
    checkRep();
}
```

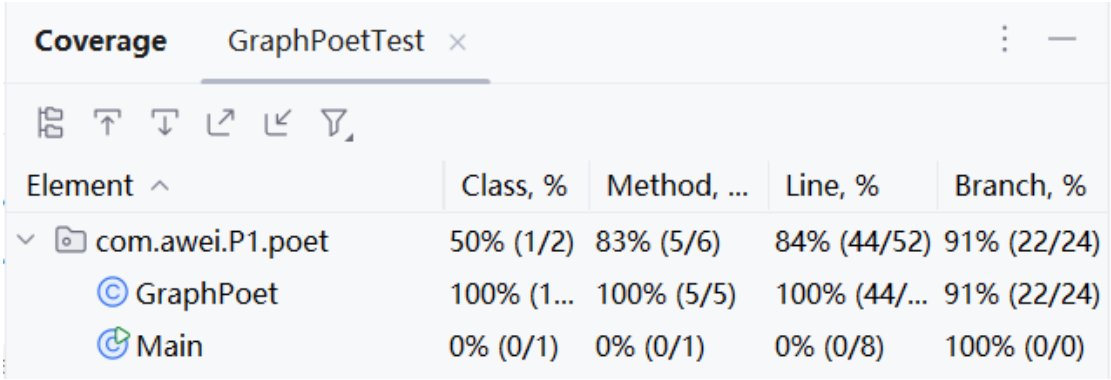
2、对于 poem 函数

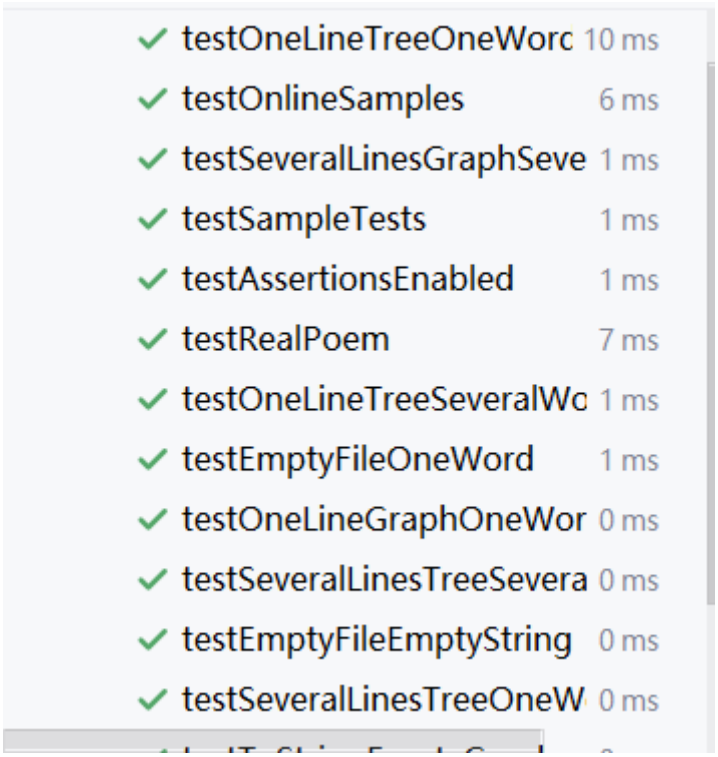
主要利用已有的诗歌生成的图，在输入的 input 字符串中增加 bridge word。根据 Graph<L>已有的 target 和 source 函数进行判断，前面的词的 target 和后面的词的 source 如果有重合的词，从中读取最大权值的一边作为 bridge word 加入即可。

```
/**
 * Generate a poem.
 *
 * @param input string from which to create the poem
 * @return poem (as described above)
 */
public String poem(String input) {
    StringBuilder stringBuilder = new StringBuilder();
    List<String> list = new ArrayList<>(Arrays.asList(input.split("
")));
    Map<String, Integer> sourceMap;
    Map<String, Integer> targetMap;
    for (int i = 0; i < list.size() - 1; i++) {
        String source = list.get(i).toLowerCase();
        String target = list.get(i + 1).toLowerCase();
        stringBuilder.append(list.get(i)).append(" ");
        targetMap = graph.targets(source);
        sourceMap = graph.sources(target);
        int maxWeight = 0;
        String bridgeWord = "";
        for (String string : targetMap.keySet()) {
            if (sourceMap.containsKey(string)) {
                if (sourceMap.get(string) + targetMap.get(string) >
maxWeight) {
                    maxWeight = sourceMap.get(string) +
targetMap.get(string);
                    bridgeWord = string;
                }
            }
        }
        if (maxWeight > 0) {
```

```
            stringBuilder.append(bridgeWord + " ");
        }
    }
    stringBuilder.append(list.get(list.size() - 1));
    return stringBuilder.toString();
}
```

3、在完成 GraphPoet 之后进行测试





3.4.3 Graph poetry slam

这个任务主要是利用我们上面已经写好的 GraphPoet 生成一个诗歌进行测试，在这里我选择的是在 1927 年由 Max Ehrmann 所写的 Desiderata（生命之所求）作为生成 Graph 的选择。然后在 input 中输入“This is a world”，经过我们已有的 Graph.poem(input) 得到的结果如下图，“This is still a beautiful world”

```
//Desiderata by Max Ehrmann, 1927
final GraphPoet graphPoet = new GraphPoet(new
```

```
File("src/main/java/com/awei/P1/poet/Desiderata.txt"));
final String inputString = "This is a world.";
final String output = graphPoet.poem(inputString);
System.out.println(inputString + "\n>>>\n" + output);
```