

**2021年春季学期  
计算学部《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 傅浩东 |
| 学号 | 1190202105 |
| 班号 | 1903002 |
| 电子邮件 | [1091288450@qq.com](mailto:1091288450@qq.com) |
| 手机号码 | 13881165621 |

**目录**

[1 实验目标概述 1](#_Toc73817591)

[2 实验环境配置 1](#_Toc73817592)

[3 实验过程 2](#_Toc73817593)

[3.1 Poetic Walks 2](#_Toc73817594)

[3.1.1 Get the code and prepare Git repository 2](#_Toc73817595)

[3.1.2 Problem 1: Test Graph <String> 2](#_Toc73817596)

[3.1.3 Problem 2: Implement Graph <String> 3](#_Toc73817597)

[3.1.3.1 Implement ConcreteEdgesGraph 3](#_Toc73817598)

[3.1.3.2 Implement ConcreteVerticesGraph 7](#_Toc73817599)

[3.1.4 Problem 3: Implement generic Graph<L> 10](#_Toc73817600)

[3.1.4.1 Make the implementations generic 10](#_Toc73817601)

[3.1.4.2 Implement Graph.empty() 10](#_Toc73817602)

[3.1.5 Problem 4: Poetic walks 11](#_Toc73817603)

[3.1.5.1 Test GraphPoet 11](#_Toc73817604)

[3.1.5.2 Implement GraphPoet 12](#_Toc73817605)

[3.1.5.3 Graph poetry slam 13](#_Toc73817606)

[3.1.6 使用Eclemma检查测试的代码覆盖度 13](#_Toc73817607)

[3.1.7 Before you’re done 14](#_Toc73817608)

[3.2 Re-implement the Social Network in Lab1 15](#_Toc73817609)

[3.2.1 FriendshipGraph类 15](#_Toc73817610)

[3.2.2 Person类 17](#_Toc73817611)

[3.2.3 客户端main() 17](#_Toc73817612)

[3.2.4 测试用例 17](#_Toc73817613)

[3.2.5 提交至Git仓库 18](#_Toc73817614)

[4 实验进度记录 19](#_Toc73817615)

[5 实验过程中遇到的困难与解决途径 19](#_Toc73817616)

[6 实验过程中收获的经验、教训、感想 20](#_Toc73817617)

[6.1 实验过程中收获的经验和教训 20](#_Toc73817618)

[6.2 针对以下方面的感受 20](#_Toc73817619)

# 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。

# 实验环境配置

Java环境实验一以及配置好，各种工具如Git也在实验一安装好。本实验要求安装配置的EclEmma（用于统计Junit测试代码覆盖度的plugin）在最新版的Eclipse中自带。

在这里给出你的GitHub Lab2仓库的URL地址（Lab2-学号）。

<https://github.com/ComputerScienceHIT/HIT-Lab2-1190202105>

# 实验过程

## Poetic Walks

该任务主要是训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象编程（OOP）技术实现ADT，最后ADT泛型化。

* 撰写测试用例：设计Graph（黑箱检测）、编写异常操作。
* 两个类ConcreteEdgesGraph，ConcreteVerticesGraph 实现Graph接口：add、set、remove、vertices、sources、targets，两个类中都有各自实现的类Edge和Vertex，这些需要自己设计。
* 运用泛型的思想，将String拓展为泛型L类。
* 利用实现的Graph，实现GraphPoet类，根据输入产生新的诗歌。

### Get the code and prepare Git repository

在 Get the code 步骤中，从以下地址获取初始代码：

Git clone http://github.com/rainywang/Spring2021\_HITCS\_SC\_Lab2.git

由于代码文件夹存在混乱，将P1置换到src和test目录下。

Prepare Git Repository步骤中，git init。

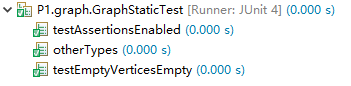
### Problem 1: Test Graph <String>

目前只测试顶点标签为String的图形。之后将扩展到其他类型的标签。

1. 使用静态方法的测试策略在文件GrapStaticTest.java中进行测试，主要是测试函数Grap.empty()。由于该方法是静态的，并且此时L类型只是String，因此将只有一个实现，并且只需要运行这些测试一次。根据提供的这些测试，暂且不进行更改，后续过程将添加进L为不同类型的测试。
2. 在测试文件GraphInstanceTest.java中，根据编写的testing strategy来实现测试方法，测试包括：add、set、remove、vertices、sources、targets。对于每一个测试函数需要获得全新的空图，测试设计、实现思路和过程如下。

|  |  |
| --- | --- |
| 测试函数 | 测试设计、实现思路和过程 |
| testAdd( ) | 往空图中添加新的点、已经存在的点，查看是否add成功。若是add重复的点，返回false。 |
| testSet( ) | 在图中添加进一些点，然后设置有向边，除了不同点之间的正权重边，考虑其他特殊情况：  边的source或target不存在，则add顶点；  若边权重为0，删除该边；  边可以指向自己。 |
| testRemove( ) | 删除一个顶点，同时要删除这个顶点作为source或target的那些边。因此除了检查该顶点是否还存在，还要检测相关边是否存在。 |
| testVertices( ) | 略 |
| testSourcesAndTargets( ) | 若有A作为source指向B的边，那么就有B作为target由A来的等权重的边；添加边后的某顶点的返回值与预期比较。 |

测试结果：



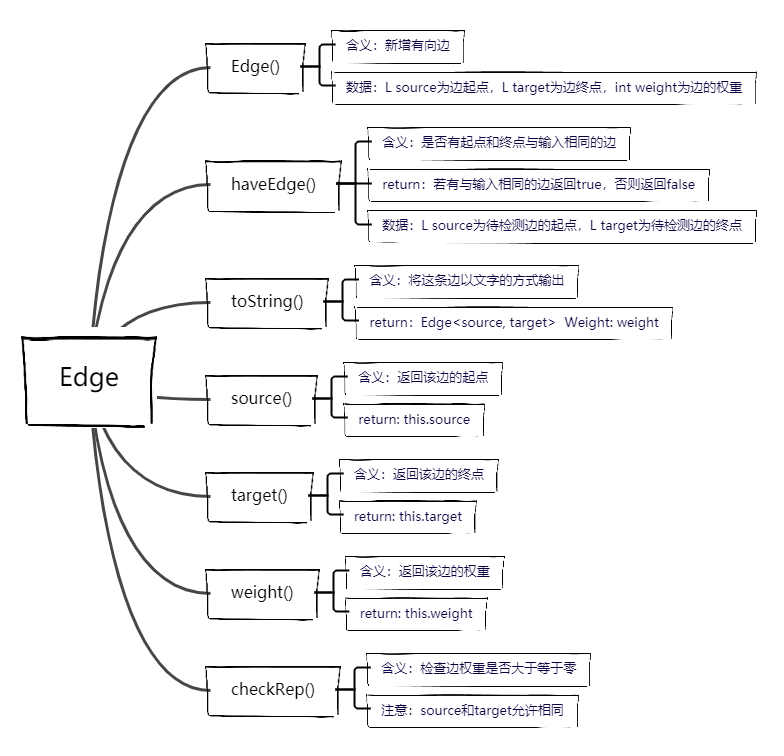
### Problem 2: Implement Graph <String>

对于Graph <String>的实现，主要是基于边和点两种方法来储存实现的。由此，在实现ConcreteEdgesGraph和ConcreteVerticesGraph之前还需要实现Edge和Vertex两种自定义类。除了实现上述问题中接口Graph的方法，每个方法都需要有documented specification，还需要实现AF(Abstraction Function)、RI(Representation Invariant)和防止表示暴露，书写checkRep方法以及重写toString。

#### Implement ConcreteEdgesGraph

1. 首先设计Edge类：从source到target权重为weight的有向边。

主要定义如下4中方法：haveEdge、source、target、weight，当然还重写了toString，写了private方法checkRep，它们的含义、参数、返回值以及注意事项等如下图所示：



首先对于Edge类里面的fields

|  |
| --- |
| **private** **final** L source;  **private** **final** L target;  **private** **final** **int** weight; |

对于Abstraction function，Edge类中的source表示有向边起点，target为该边的终点，weight为有向边的权重。

|  |
| --- |
| // Abstraction function:  // AF(source) = the source vertex of the edge  // AF(target) = the target vertex of the edge  // AF(weight) = the weight of the edge |

对于Representation invariant，保证每一个Edge对象的权重都要大于等于零，若是权重 weight == 0，那么该边之后会被删去。

|  |
| --- |
| // Representation invariant:  // vertex can point to itself  // the weight have to be positive ( >= 0)  // if weight is 0, it will be removed later. |

对于Safety from rep exposure，首先用private和final来表示fields，其次要使数据不会泄露，外部无法修改内部实现。

|  |
| --- |
| // Safety from rep exposure:  // Check the rep invariant is true. That means setting  // source, target and weight unchangeable / private / final  // add return value MUST be immutable |

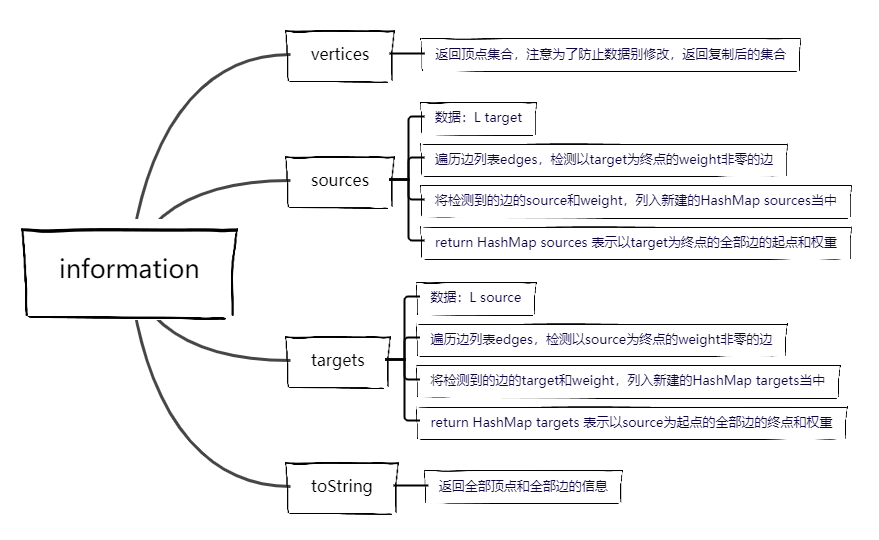
1. 实现ConcreteEdgesGraph

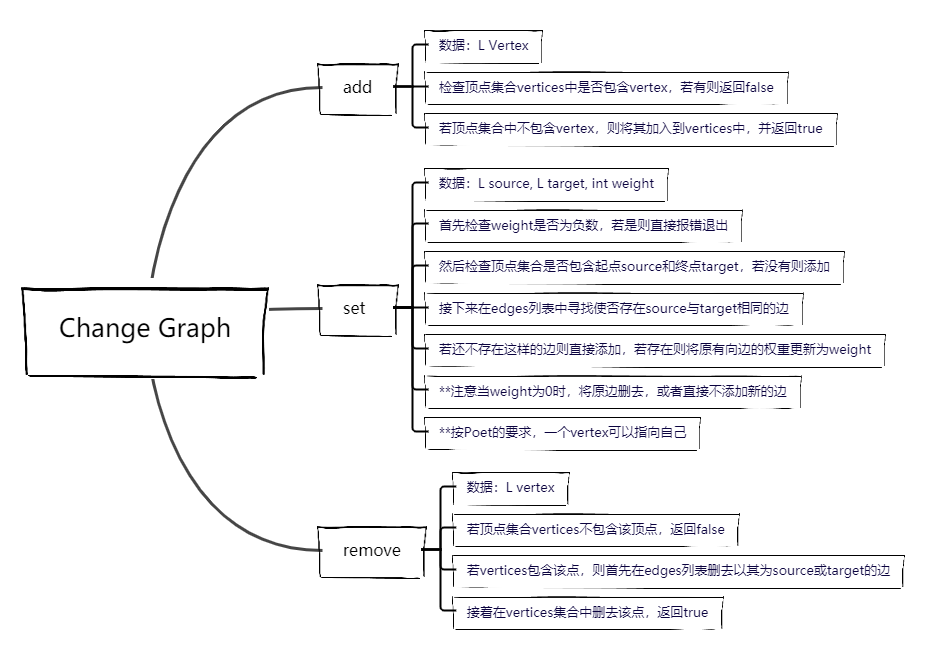
首先在Graph.java中我们以及清楚了重写methods的要求和含义，按要求有以下数据结构，那么基于此我们有AF、RI以及实现方法：

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

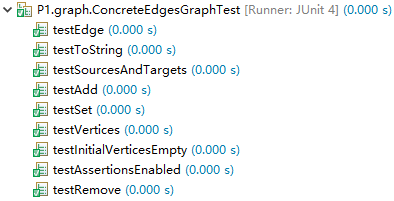
|  |
| --- |
| // Abstraction function:  // AF(vertices) = the vertices set of this graph.  // AF(edges) = the edges list of this graph  // Representation invariant:  // one or no edge between the same source and target  // the source and target of any edge in the edge list must be in the vertices set    // Safety from rep exposure:  // Check the rep invariant is true, but list and set are mutable  // All fields MUST be private  // So make defensive copies instead of just return mutable data |

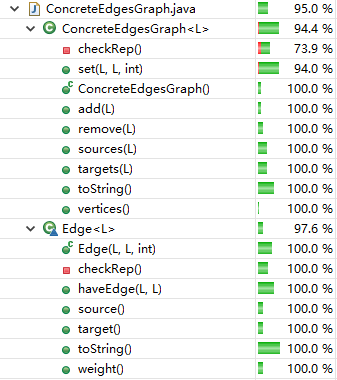
该类以类Edge为基础重写Graph<L>，如下两张图所示：





1. ConcreteEdgesGraphTest测试结果以及覆盖率：

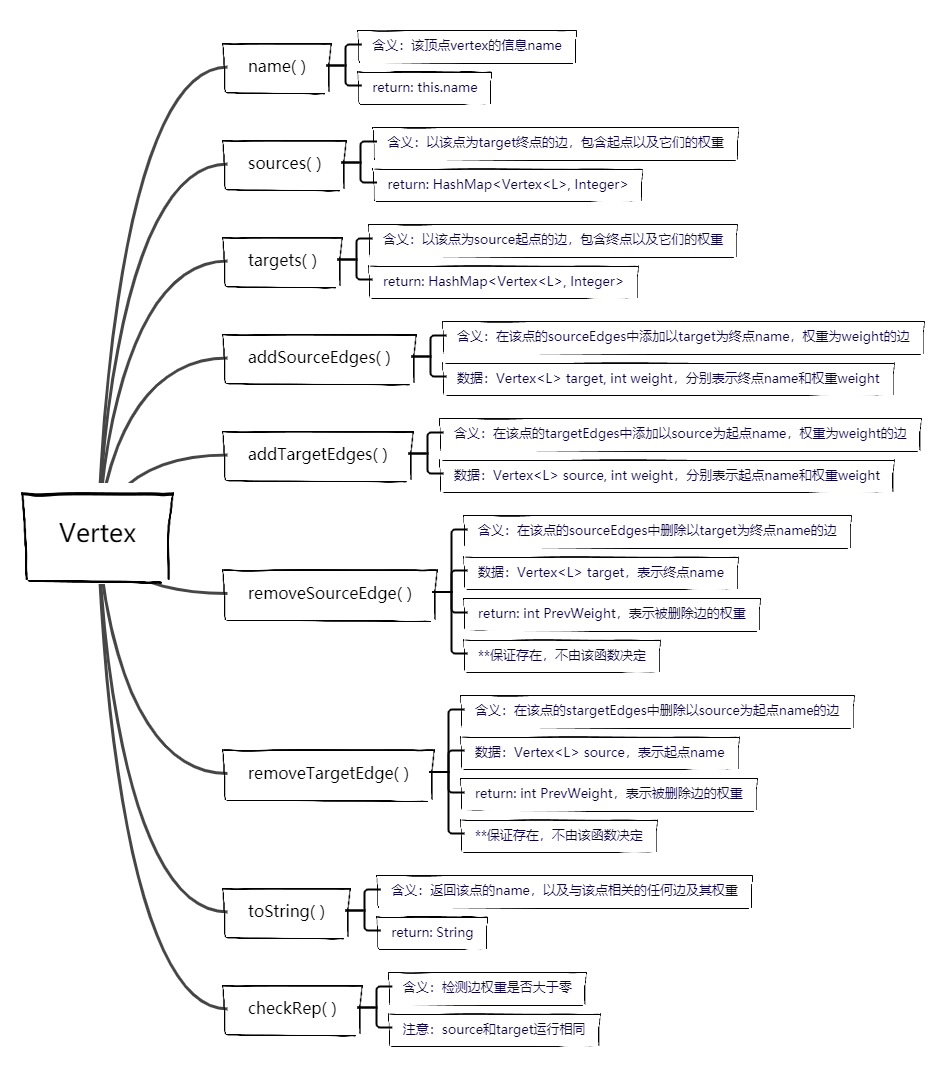




#### Implement ConcreteVerticesGraph

1. 首先设计Vertex类：每个Vertex中保存有L name，以Vertex为起点的边targets HashMap，以Vertex为终点的边sources HashMap，储存有其他点和有向边的权重weight。

主要定义如下几种方法：name、sources、targets、addSourceEdges、addTargetEdges、removeSourceEdges、removeTargetEdges，当然还重写了toString，写了private方法checkRep，它们的含义、参数、返回值以及注意事项等如下图所示：



首先对于Vertex类里面的fields

|  |
| --- |
| **private** **final** L name;  // this vertex as source  **private** **final** Map<Vertex<L>, Integer> sourceEdges = **new** HashMap<>();  // this vertex as target  **private** **final** Map<Vertex<L>, Integer> targetEdges = **new** HashMap<>(); |

对于Abstraction function，Vertex类中的sourceEdges表示以name为起点的边，存放有targets点以及有向边的权重weight；targetEdges表示以name为终点的边，存放有sources点以及有向边的权重weight；而那么表示该点的信息。

|  |
| --- |
| // Abstraction function:  // AF(sourceEdges) = vertecies to the target and thier weights  // the edges to the vertex  // AF(targetEdges) = vertecies from the source and thier weight  // the edges from the vertex |

对于Representation invariant，保证每一个Vertex对象的每一条相关有向边的权重都要大于零，并且A存有A到B的边，那么B中也必须要存有相同weight的边。

|  |
| --- |
| // Representation invariant:  // the weight have to be positive ( > 0)  // if vertex A has a edge to vertex B, then B must  // have an edge from A with the SAME weight |

对于Safety from rep exposure，首先用private和final来表示fields，其次要使数据不会泄露，外部无法修改内部实现，对于返回的HashMap、Set等内容，一定要是返回复制后的数据。

|  |
| --- |
| // Safety from rep exposure:  // Check the rep invariant is true, all fields are private  // return copied Maps |

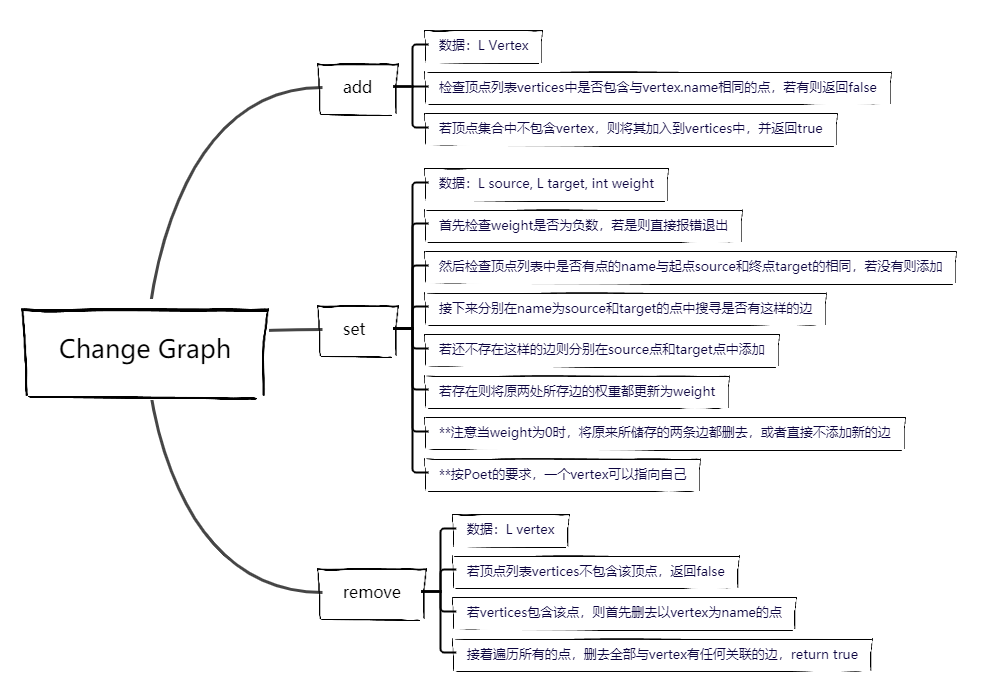
1. 实现ConcreteVerticesGraph

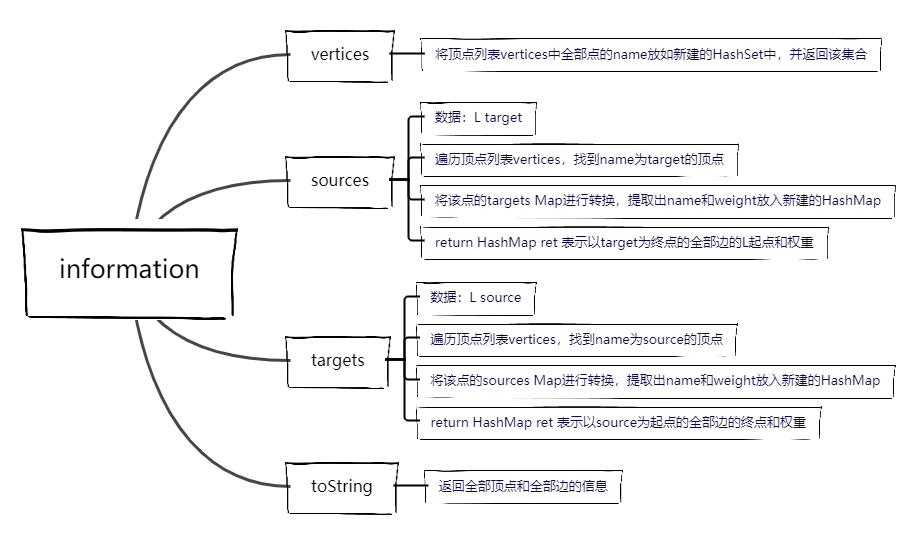
首先在Graph.java中我们以及清楚了重写methods的要求和含义，按要求有以下数据结构，那么基于此我们有AF、RI以及实现方法：

|  |
| --- |
| **private** **final** List<Vertex<L>> vertices = **new** ArrayList<>(); |

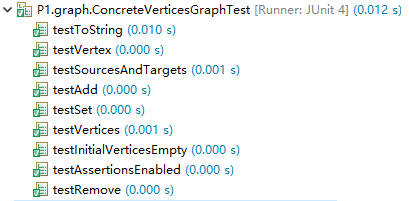
|  |
| --- |
| // Abstraction function:  // AF(vertices) = the vertices of the graph  // Representation invariant:  // Each vertex's name(key) MUST be unique, and cant be null  // Safety from rep exposure:  // List is mutable, return copied data, and dont use dangerously  // All fields MUST be private final |

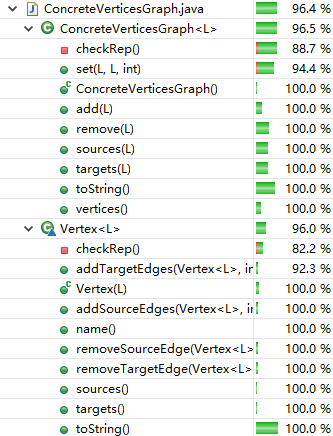
该类以类Vertex为基础重写Graph<L>，如下两张图所示：





1. ConcreteVerticesGraphTest测试结果以及覆盖率：





### Problem 3: Implement generic Graph<L>

在以上的实现中，任何一项都依赖于标签是特定类型String，即认定L为String。接下来将已有的实现改为基于L泛型的即可。

#### 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， Graph在两个泛型实现下仍然可以通过，表示更改成功。



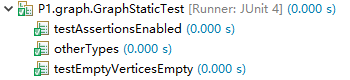
#### Implement Graph.empty()

empty返回一个Graph接口的具体实现即可：

|  |
| --- |
| **public** **static** <L> Graph<L> empty() {  **return** **new** ConcreteEdgesGraph<L>();  } |

更改GraphStaticTest，使用不同泛式，例如基本数据类型（Immutable）中的Double和Integer类型进行检测，增加检测类型代码和结果如下：

|  |
| --- |
| // test other vertex label types in Problem 3.2  @Test  **public** **void** otherTypes() {  *assertEquals*("expected empty() graph to have no vertices",  Collections.*emptySet*(), Graph.<String>*empty*().vertices());  *assertEquals*("expected empty() graph to have no vertices",  Collections.*emptySet*(), Graph.<Integer>*empty*().vertices());  *assertEquals*("expected empty() graph to have no vertices",  Collections.*emptySet*(), Graph.<Double>*empty*().vertices());  } |



### Problem 4: Poetic walks

根据语库corpus生成基于Words的有向图，每条边上的权重表示该两个单词联系出现的次数。例如：A—2 ––>B，表示AB这种形式出现过两次。值得注意的是，允许出现AA连续形式。根据生成的Graph，若有input，则对其自动补全，即检查两个单词之间是否在Graph中找得到另一个单词插入。即相邻单词A和B，在Graph中有A到C和C到B，那么就将C插入到input中等待输出，若存在多个C则选取两边weight之和最大的C。插入到最后得到输出output。

#### Test GraphPoet

在测试文件GraphPoetTest.java中，根据编写的testing strategy来实现测试方法，测试包括：corpus类型（nothing、一行、多行、存在空行），input类型（nothing、有无插入），output类型（0/>=1 bridge word、choose heavier bridge word）。测试设计、实现思路和过程如下：

|  |  |
| --- | --- |
| 测试函数 | 测试思路设计 |
| testExample() | deal with the example |
| testMyExample() | //corpus filename: seven-words.txt  //corpus: free to add additional methods.  //input: I want to add methods. |
| testEmptyInput() | //corpus filename: inputNothing.txt  //corpus(I dont care):  //input nothing |
| testEmptyCorpus() | //corpus filename: corpusNothing.txt  //corpus nothing  //input: I want to add methods. |
| testOneLineTreeOneWord() | //corpus filename: severalLines.txt  //corpus:  Hugo is one of the most popular open-source static site generators.  With its amazing speed and flexibility, Hugo makes building websites fun again.  //input: The popular hugo can makes websites with amazing speed. |
| testOneLineTreeSeveralWords() | //corpus filename: corpusToSelf.txt  //corpus: Hello, hello, hello, Hello, my name is Fu Haodong.  //input: I want to add methods. |

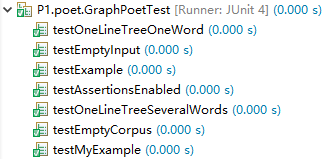
#### Implement GraphPoet

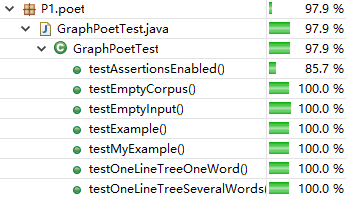
GrapPoet()方法：对于语库corpus，先一行一行地读取，将内容全部利用toLowerCase()方法转化为小写字母，再用split(“ ”)方法分为一个个的单词，若单词后有标点符号，则标点符号也算入该单词，根据每一行的String[] words生成加权有向图。

Poem()方法：将input用split(“ ”)方法分为一个个的单词，依次在Graph中查询是否在inputWords[i]. toLowerCase()和inputWords[i+1]. toLowerCase()之间的第三个单词，若有则插入，直到inputWords中的每一个单词对都被查询完。

Private void checkRep()：首先保证图不是空的、顶点表中的顶点不是空的且全是小写字母，最后保证每条边权重至少为1.

测试结果及覆盖率：





#### Graph poetry slam

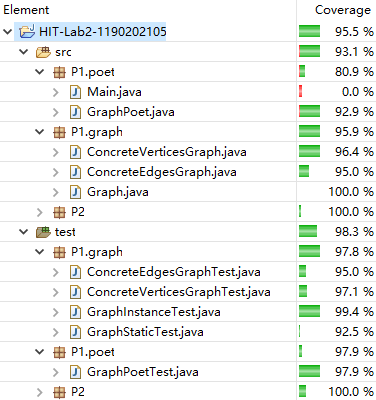
在GraphPoet.java的main函数中添加一个自己的测试，搜索一首英文诗。

|  |
| --- |
| **final** GraphPoet myNimoy = **new** GraphPoet(**new** File("src/P1/poet/jellicles-song.txt"));  **final** String myInput = "Jellicle are quit in some time.";  System.***out***.println(myInput + "\n>>>\n" + myNimoy.poem(myInput)); |

运行结果如下所示：



### 使用Eclemma检查测试的代码覆盖度



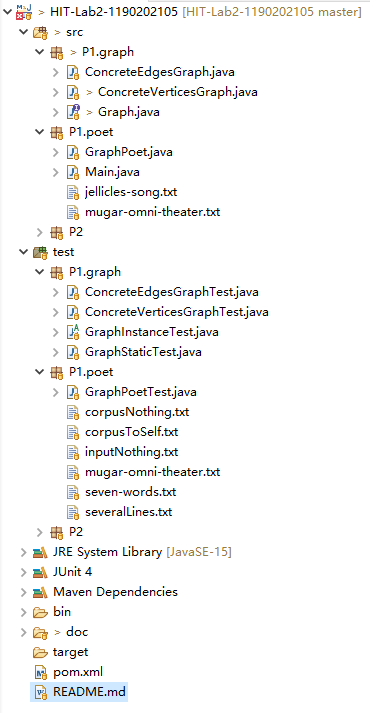
### Before you’re done

请按照[http://web.mit.edu/6.031/www/sp17/psets/ps2/#before\_youre\_done](http://web.mit.edu/6.031/www/sp17/psets/ps2/" \l "before_youre_done)的说明，检查你的程序。

通过Git提交当前版本到GitHub上你的Lab2仓库。

|  |
| --- |
| Git add .  Git commit -m “update”  Git push -u origin master |

项目的目录结构树状示意图：

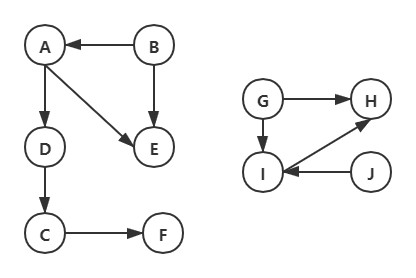


## Re-implement the Social Network in Lab1

利用上述实现的ADT，重新实现实验一中的FriendshipGraph，令L泛型为实验一中的Person，改变两个java类，通过本实验中的ConcreteEdgeGraph或者ConcreteVertexGraph来快速实现。

### FriendshipGraph类

首先创建一个新的Graph，将Person添加到vertices中，添加朋友关系，形成有向边。接下来按照给出的例子，要对Person、addVertex、addEdge、getDistance进行分析。



#### Method addVertex()

创建节点的时候，直接使用Graph的add操作，与本操作十分吻合。

// add a new convex

**public** **boolean** addVertex (Person newPerson) {

**return** graph.add(newPerson);

}

#### Method addEdge()

addEdge的功能是为不同的顶点之间添加边，由于类要保留拓展到有向图的功能，所以每一次运行addEdge都只会添加一条单向边。利用Graph中的set操作，只不过要注意set是允许相同顶点的指向。而且此时不必使用太过复杂的权重，用0和1来表示有无边即可。

// add a new edge between two vertexes

**public** **int** addEdge (Person personA, Person personB) {

**if** (personA.name().equals(personB.name())) **return** -1;

**int** prevWeight;

prevWeight = graph.set(personA, personB, 1);

**return** prevWeight;

}

#### Method getDistance()

BFS算法：从原始顶点开始一层一层地进行搜索，每经过一次顶点标记为visited（这里设置一个HashSet<Person> visited），下经过就直接跳过，并且把它的非终点全部入队；每搜索完一层则distance++；若达到终点则返回层数。若最终队列里面不存在任何顶点则表示没有从原点到终点的路径。程序最开始还对两点的相同性进行检测，若是相同直接返回0.

// get the distance between the two vertexes, BFS

**public** **int** getDistance(Person personA, Person personB) {

// nobody is a friend of oneself

HashSet<Person> visited = **new** HashSet<Person>();

// the distance is 0, when they are the same person

**if** (personA == personB)

**return** 0;

**int** temp, distance = 0;

visited.add(personA);

LinkedList<Person> queue = **new** LinkedList<Person>();

queue.addAll(graph.targets(personA).keySet());

**int** levelSize = queue.size();

**do** {

temp = 0;

distance ++;

**for** (**int** i = 0; i < levelSize; i++) {

**if** (queue.getFirst() == personB)

**return** distance;

**else** **if** (visited.contains(queue.getFirst()))

queue.removeFirst();

**else** {

// all the friends of the guy enter the queue

queue.addAll(graph.targets(queue.getFirst()).keySet());

temp += graph.targets(queue.getFirst()).size();

// set the guy as visited

visited.add(queue.getFirst());

queue.removeFirst();

}

}

// all the reachable are visited, but person 2 not found

**if** (temp == 0) **return** -1;

levelSize = temp;

} **while**(**true**);

}

### Person类

每个人对应到一个Person对象，且至少包含人的名字，定义方法返回名字name()。

**package** P2;

**public** **class** Person {

**private** **final** String name;

// Abstraction function:

// AF(name) = the name of the person

// Safety from rep exposure:

// all fields are final and private

// constructor

**public** Person(String str) {

**this**.name = str;

}

// metheds

**public** String name() {

**return** name;

}

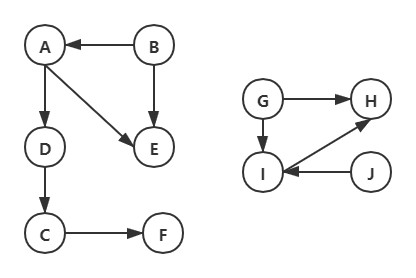
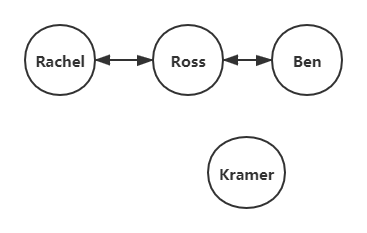
}

### 客户端main()

由于只是改变了methods的实现方法，直接用实验一中的main客户端。

### 测试用例

给出两个测试，首先按照下图进行创建关系：

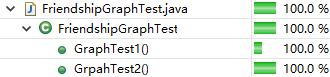


它们的测试用例分别如下：

|  |
| --- |
| *assertEquals*(1, graph.getDistance(rachel, ross));  *assertEquals*(2, graph.getDistance(rachel, ben));  *assertEquals*(0, graph.getDistance(rachel, rachel));  *assertEquals*(-1, graph.getDistance(rachel, kramer)); |
| *assertEquals*(3, graph.getDistance(a, f));  *assertEquals*(-1, graph.getDistance(a, b));  *assertEquals*(-1, graph.getDistance(e, f));  *assertEquals*(2, graph.getDistance(d, f));  *assertEquals*(-1, graph.getDistance(a, j));  *assertEquals*(0, graph.getDistance(i, i));  *assertEquals*(-1, graph.getDistance(g, j));  *assertEquals*(1, graph.getDistance(i, h)); |

检测结果及覆盖率：



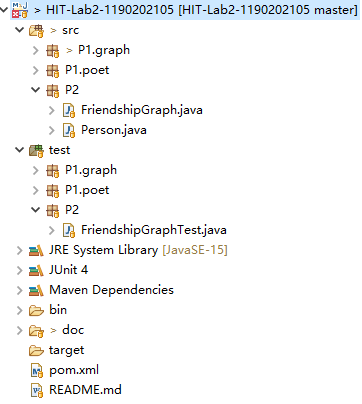


### 提交至Git仓库

通过Git提交当前版本到GitHub上Lab2仓库。

|  |
| --- |
| Git add .  Git commit -m “update”  Git push -u origin master |

给出项目的目录结构树状示意图。



# 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 |
| 2021.05.30 | 晚 | 初始化项目、了解项目内容等 | 遇到困难，未完成 |
| 2021.05.31 | 晚 | 3.1 Problem 1: Test Graph<String> | 遇到困难，未完成 |
| 2021.05.31 | 晚 | 3.1 Problem 2: ConcreteEdgesGraph | 超时完成 |
| 2021.06.03 | 晚 | 3.1 Problem 2: ConcreteVerticesGraph | 按时完成 |
| 2021.06.03 | 早 | 3.1 Problem 3: Generic Graph<L> | 按时完成 |
| 2021.06.03 | 晚 | 3.1 Problem 4: Poet and Poet Test | 按时完成 |
| 2021.06.04 | 早 | 3.1 Problem 1: Test Graph<String> | 未完成 |
| 2021.06.04 | 晚 | 3.1 Problem 1: Test Graph<String> | 几乎完成 |
| 2021.06.05 | 早 | 3.2 Re-implement Social Network | 完成 |
| 2021.06.05 | 午 | 撰写报告 | 未完成 |
| 2021.06.05 | 晚 | 撰写报告 | 完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 在实现List的选择删除时，不能使用增强for循环遍历list，删除元素后会有异常抛出。 | 改用方法removeIf |
| 对于泛型的不理解 | 查阅资料后有了一定的理解，但还是一知半解 |
| 对于基于Vertex实现的图，在进行remove操作时，难以实现在删除点之前就将相关边给删去。由此对HashMap的实现存疑。 | 没有解决，但是我先remove该点，之后进一步清理图中该删去的边。 |
| 对于equals和HashCode是否重写不清楚 | 了解后续课程的内容 |
| 不习惯于先写测试用例，在写的过程中对于不是自己定义的方法使用不习惯 | 在完成相关类的实现之后进一步修改测试用例 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

1. 掌握了一定从应用场景到ADT的抽象映射的能力；
2. Java语言的有些类方法的使用不够熟练，甚至很多类方法都不知道，导致在实验过程中浪费了很多时间，有些代码写得过于冗长；

## 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编程的时候需要考虑代码的可复用性，需要使整个编程更加具有普适性；面向应用场景可以直接根据要求来编写程序，更易于操作，但是程序复用性很低，难以在其他场景进行复用。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

对我来说，不使用泛型的编程更加顺手和便于理解，但是使用泛型能够兼容所指定的类型，所以应用范围会更加广。

1. 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？

优势是能够尽早找到程序中的错误，避免错误累积到后续难以修改；但是不太能适应这种测试方法，首先是方法规约于测试内容联系不上，再来就是测试用例有时不太能考虑到全部情况。

1. 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？

在抽象编程的基础上，利用已有代码，避免重复工作，节省编程时间，减少开发成本。

1. 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？

对于应用场景到ADT的抽象映射，其实难度不大，但是还是不熟悉rep方法等。

1. 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？

意义在于避免数据外泄、指导程序编写的准确性和健壮性的提高，规避错误。愿意坚持注意是否有rep exposure，由此来保护内容的私密性。

1. 关于本实验的工作量、难度、deadline。

实验与课程进度不相符合，实验往往要落后课程一段时间，由此如果要按时完成实验在读懂要求上消耗的时间量是巨大的。由此导致工作量等等都很大，但是deadline也还能接受。

1. 《软件构造》课程进展到目前，你对该课程有何体会和建议？

课程到目前为止，绝大多数内容都显得很抽象，在此之前对于Java的了解也没有那么多，所以短时间理解起来还是比较难的。

建议课程任务降低一点点，实验内容比例升高。