哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2018 年春季学期

计算机学院大二软件构造课程

Lab 2 实验报告

姓名	朱明彦
学号	1160300314
班号	1603003
电子邮件	1160300314@stu.hit.edu.cn
手机号码	18846082306

目录

1 实验目标概述

实验的主要目标是训练 ADT 的设计、规约、测试和使用 OOP 技术实现 ADT。主要有下面几个方面：

- 针对给定的 应用问题，从描述中识别所需 ADT
- 设计 ADT 的规约并评估规约的质量
- 根据 ADT 的规划设计测试用例，并写出 testing strategy
- 实现 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 representition、rep invariant 和 abstraction function。
- 使用 OOP 实现 ADT 并判定表示不变性是否违反已经是否有 rep exposure

2 实验环境配置

Java 的环境在以前就已经配置好。Git 的环境在以前也已经配置好。

在这里给出你的 GitHub Lab2 仓库的 URL 地址（Lab2-学号）。
https://github.com/ComputerScienceHIT/Lab2-1160300314

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惯。并且在后面练习 ADT 的泛型化。

3.1.1 Get the code and prepare Git repository

从https://github.com/rainywang/Spring2018_HITCS_SC_Lab2/tree/master/P1 将这个项

目的代码Clone到本地，并建立.git文件夹，并将其push到我的私人仓库中即可。

3.1.2 Problem 1: Test Graph <String>

1、主要是测试 Graph.empty()函数的 testing strategy。其中 Graph.empty()是返回一个空的 Graph<L>实现，所以此处的测试主要就是测试在不同的 L 的情况下，只要 L 为 immutable 类型的数据类型就可以使用。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. */

3.1.3 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<L>实现两次，分别基于边为主和点为主来实现图的存储和操作。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。

3.1.3.1 Implement ConcreteEdgesGraph

必须使用下面的数据结构来实现 ConcreteEdgesGraph

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

而且在必须将 Edge 写成 Immutable 类型的类。

1、下面先写出相对应的 Edge 类里面的 fields

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

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”。

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

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。具体书写如下：

```
// Representation invariant:
// Source is different from target and neither of them is empty string.
// Weight is an integer that is nonnegative.
// Source != null, target != null and source is not equal to target either.
```

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

```
private void checkRep() {
        assert weight >= 0;
        assert source != null;
        assert target != null;
    }
```

2、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
// 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

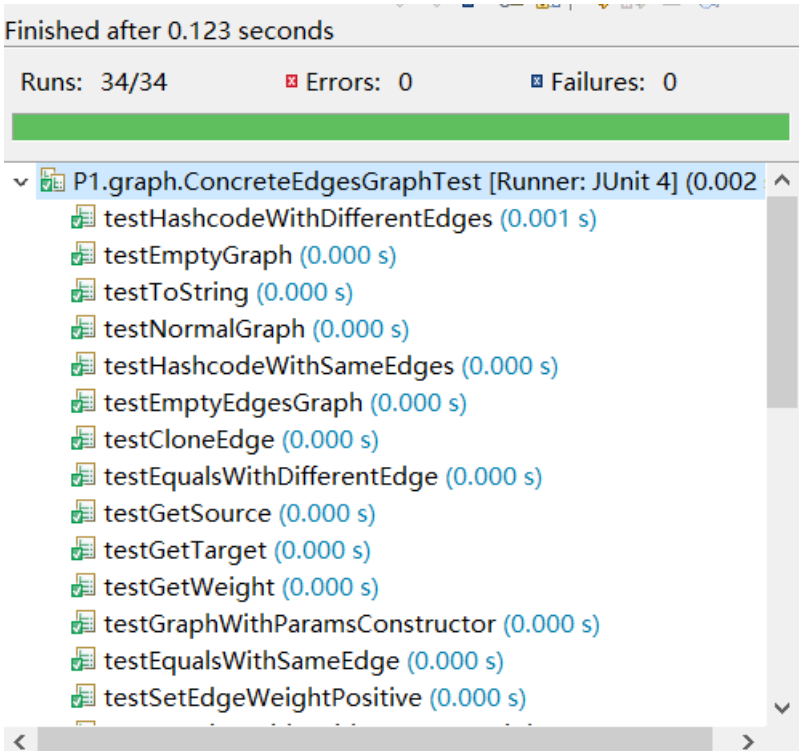
找到以 target 为 destination 的所有有向边的集合，并返回一个包括 source 和有向边权值对应关系的 Map 即可，遍历一遍边集找到所有符合条件的 Edge 加入 Map 后返回。
⑥target 函数

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

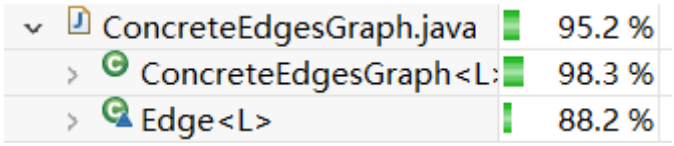
Override 继承 Object 的 toString 函数，重写后的 toString 函数要尽量符合人的阅读习惯，所以返回的 String 中包含这个图中边数、顶点数、边集和顶点集中各个元素，并且排版尽量友好。
4、进行 ConcreteEdgesGraphTest 测试

直接运行 Junit 测试，并利用 EclEmma 进行覆盖率的测试，运行结果如下

图所示



覆盖率测试如下图：



3.1.3.2 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> 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、写出 `Vertex` 中的 `fields`

```
private final L name;
private final Map<L, Integer> adjacent = new HashMap<>();
```

另外需要设计 `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”具体书写如下：

```
    // Abstraction function:
    // Represents a vertex which is one of two vertices of a directed edge that
    // is from this to the other vertex with weight.
```

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，具体书写如下

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

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：

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

2、在 ConcreteVerticesGraphTest 需要测试 Vertex 中定义的辅助操作，测试策略如下：

对于只有一种输入的测试我们划分输入如下

```
    // Partition for vertex.getName()
    //     only has one input, vertex
    //     only has one output, the name of vertex
    //
    // Partition for vertex.getAdjacentName()
    //     input vertex without edges, and vertex with edges
    //     only has one output, the number of edges adjacent to this vertex
    //
    // Partition for vertex.getMap()
    //     input vertex without edges, and vertex with edges
    //     only has one output, the copy of the adjacent map
```

总的来说，以上 Vertex 中的方法主要是 getter 方法，注意防御式拷贝即可。
对于有参数输入的函数，则需要根据输入进行划分，具体的划分如下：

```
    // Partition for vertex.equalsName(input)
    //     input has the same name with vertex, input has different name
    //     output: true if input has the same with vertex, otherwise false
    //
    // Partition for vertex.getWeightOfEdge(input)
    //     input the target of some directed edge
    //     output: if there an edge from this vertex to input return the weight of that,
    //             otherwise 0.
    //
    // Partition for vertex.put(target, weight)
    //     input the target of the edge and the weight
    //
    // Partition for vertex.adjacnetTo(input)
    //     input the target of edge
    //     output: true if this vertex is connected to target, otherwise false
    //
    // Partition for vertex.equals(input)
    //     input equal to vertex, input not equal to vertex
    //     output true if input's name equal to name of this, otherwise false
    //
    // Partition for vertex.hashCode()
    //     has only one input, vertex
    //     has only one output, the hash code of vertex
```

根据以上的划分，采用“最少一次覆盖”策略设计 Tests 样例即可。

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

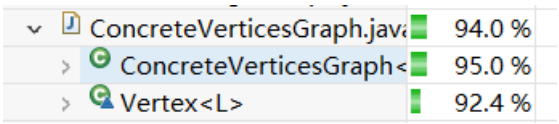
注意在删掉顶点的时候需要将其邻接的边全部删除。

④sources 函数和 targets 函数

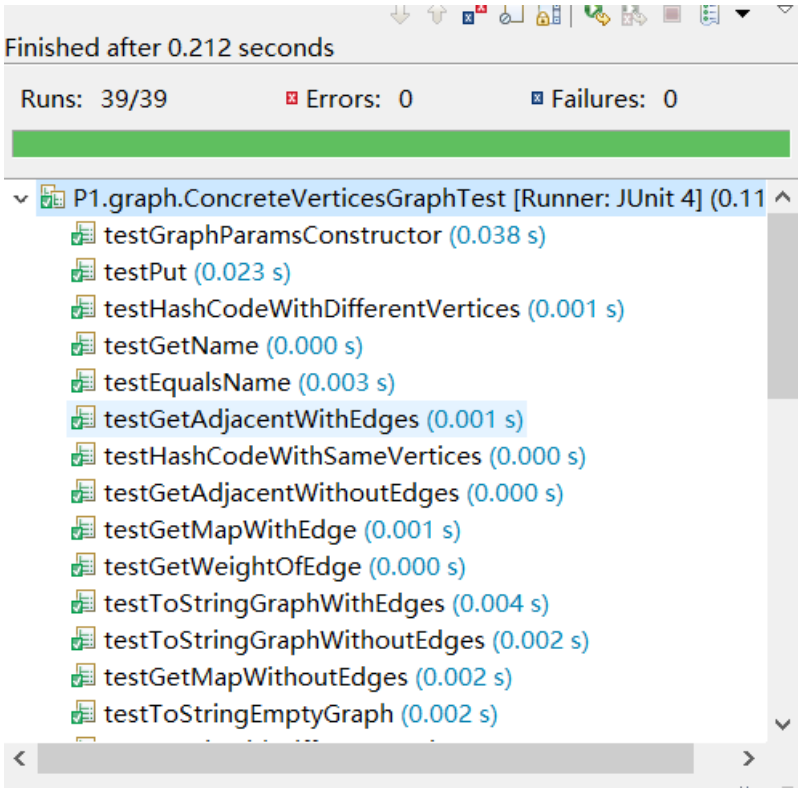
在链表中扫一遍，将符合要求的对应关系加入返回的 map 中即可。

4、进行 ConcreteVerticesGraphTest 测试

直接运行 Junit，并利用 EclEmma 测试覆盖率。覆盖率结果如下



对于测试结果如下：



3.1.4 Problem 3: Implement generic Graph<L>

将已有的两个 Graph<String>的实现改为基于 Graph<L>的实现，充分利用各种 ide 中提供的替换工具即可。

3.1.4.1 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。

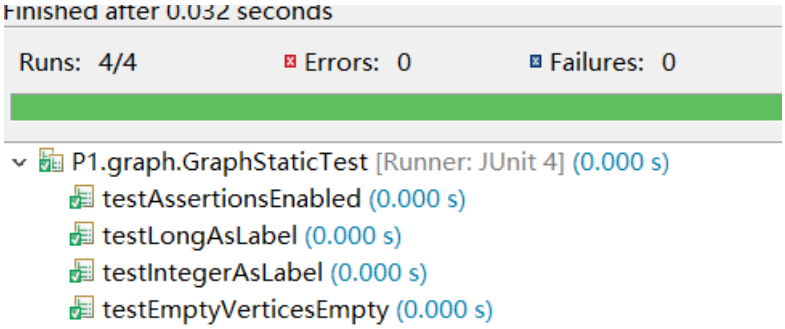
3.1.4.2 Implement Graph.empty()

1、graph.empty 返回一个 Graph 接口的具体实现即可，比如可以选择 ConcreteEdgesGraph 作为返回对象实现 graph.empty 函数如下

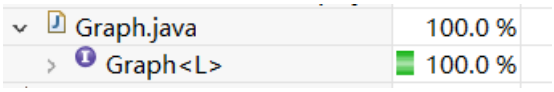
```
public static <L> Graph<L> empty() {
    return new ConcreteEdgesGraph<>();
}
```

2、测试 GraphStaticTest

使用不同的 L 泛型进行测试，由于基本数据类型均是 Immutable 类的，主要可以利用这个进行测试。此处我采用 Integer 和 Long 类型进行了测试，具体的测试结果如下：



代码覆盖测试如下：



3.1.5 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。

3.1.5.1 Test GraphPoet

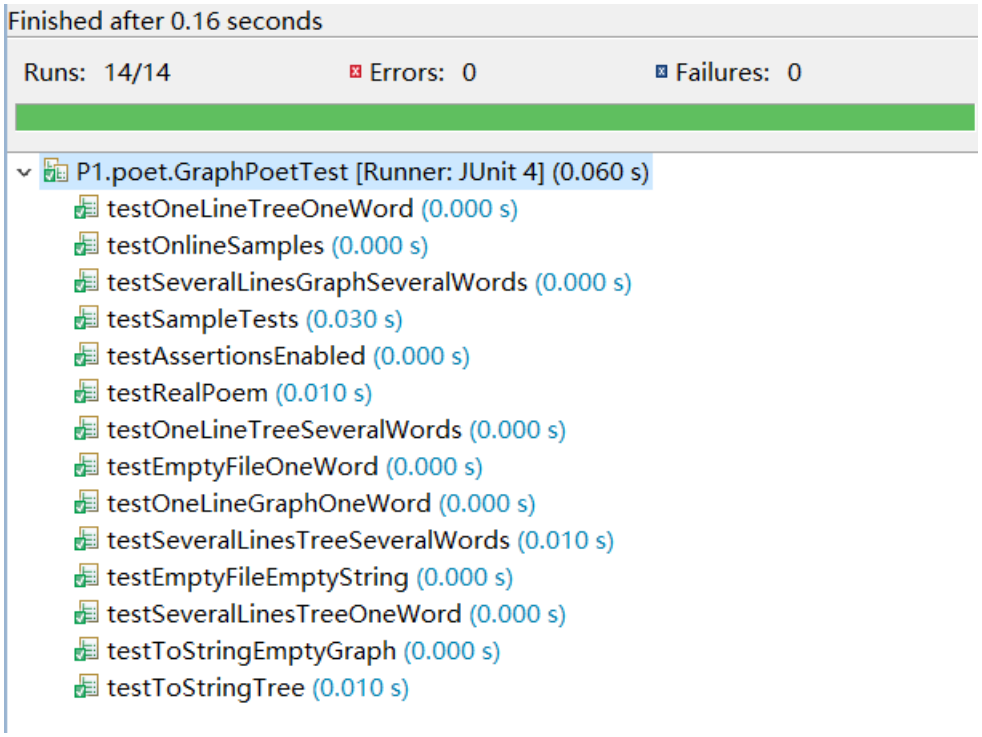
测试 GraphPoet，主要是针对可能输入进行划分。在这里主要进行了以下的测试。主要测试的函数有 Constructor、poem、toString，采用“最少一次覆盖的策略进行策略”。

Constructor 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

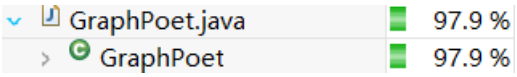
其中由于诗歌生成的图可能有多种情况，其中空图和有向树是比较容易处理的两种，而对于带环的有向图可能存在不同的 bridge word，需要在其中选择权值最大的一个。

3.1.5.2 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。
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。
3、在完成 GraphPoet 之后进行测试
测试结果如下



代码覆盖度如下



3.1.5.3 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”

```
This is a world.
>>>
This is still a beautiful world.
```

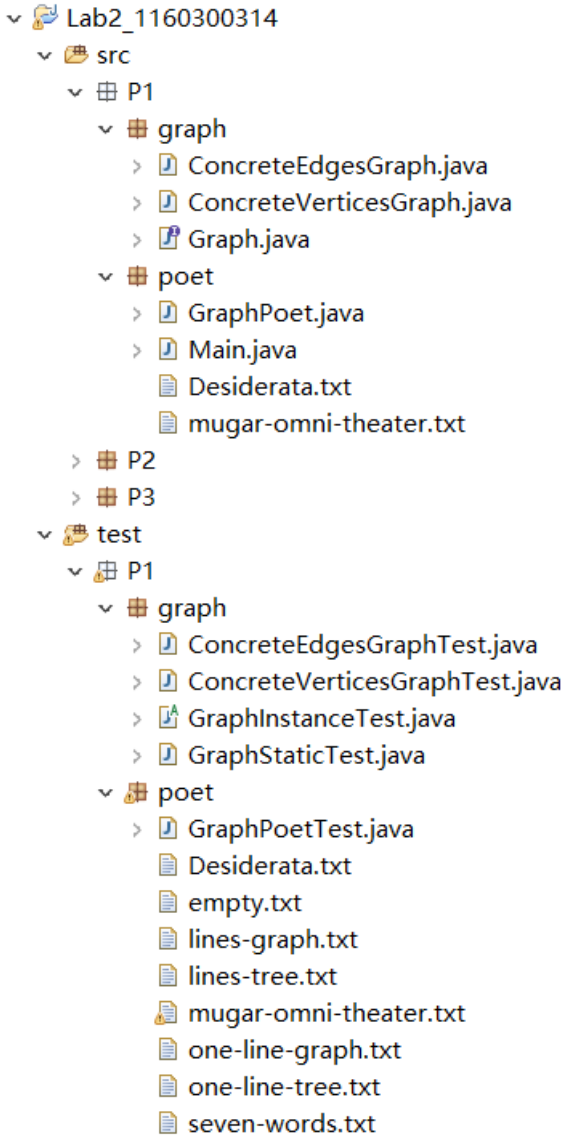
3.1.6 Before you're done

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。

```
git commit -m "finish P1"
git add -A
git push
```

运行以上命令即可

在这里给出你的项目的目录结构树状示意图。



3.2 Re-implement the Social Network in Lab1

在这里简要概述你对该任务的理解。

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，而不是重写一遍 Lab1.

3.2.1 FriendshipGraph 类

FriendShipGraph 类主要是完成建立一个朋友圈的建立，目的和 Lab1 相同，实现 addVertex、addEdge 和 getDistance 三个函数。

1、实现 addVertex 函数

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。只需要在 addVertex 中在增加对于 distance 和 visited（两个用在 BFS 的数据结构）的初始化即可。

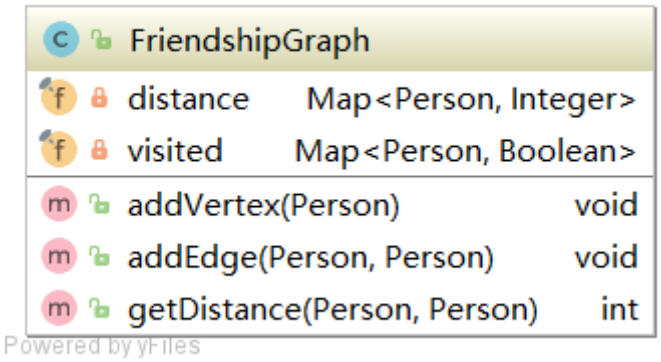
2、实现 addEdge 函数

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，只需要在 addEdge 中开始判断一下是否是源点和目的点是否相同即可，再利用 set(person1, person2,1)增加边即可。

3、实现 getDistance 函数

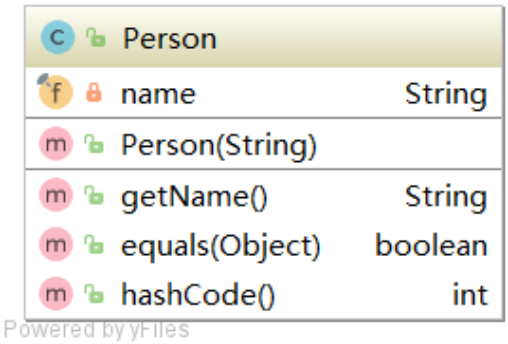
仍然是利用 BFS 来实现找最短路，大部分实现仍然利用已有的函数来实现。在寻找某一个顶点邻接的顶点，只需要利用 targets 函数即可。

UML 图见下



3.2.2 Person 类

Person 类中只需要有一个 field 即可，就是某一个人的名字即可。主要注意需要重写 equals 和 hashCode 两个函数，用在 FriendshipGraph 中对于 Person 类的判断。UML 图见下：

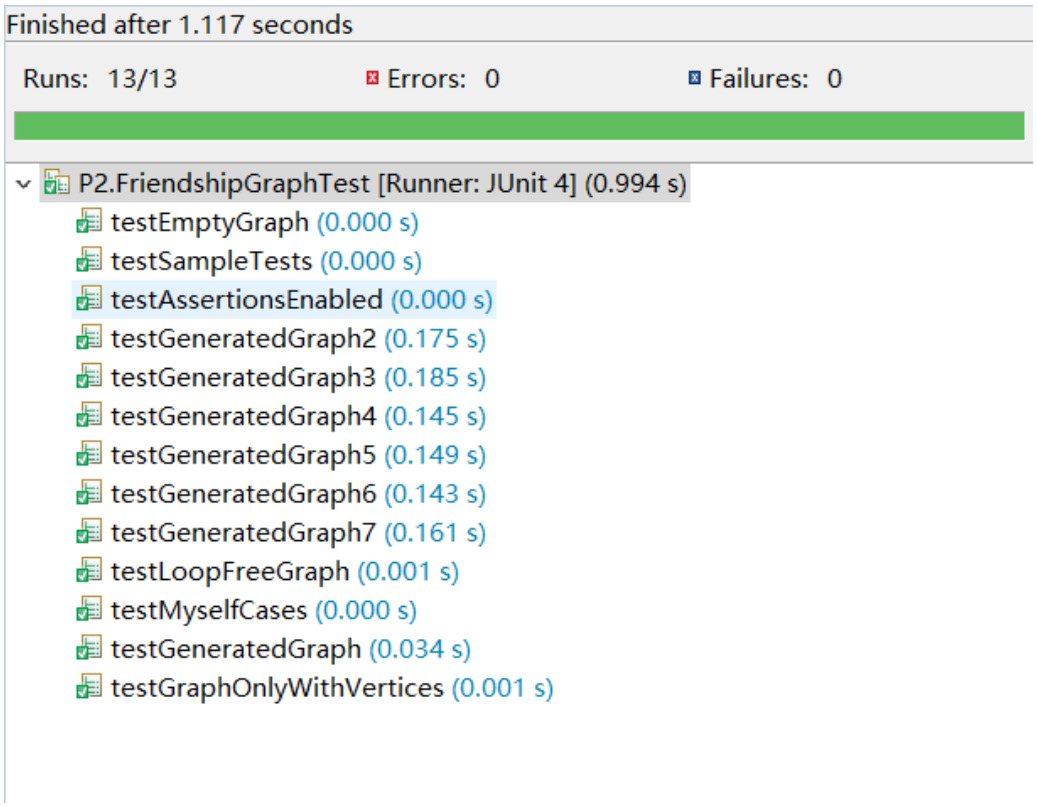


3.2.3 客户端 main()

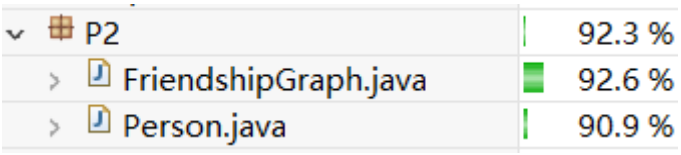
重新利用 Lab1 中已有的 main 客户端即可，由于所有 Lab2 中的所有实现都是没有改变 `addVertex`、`addEdge` 和 `getDistance` 三个函数，只需要重新使用即可使用。

3.2.4 测试用例

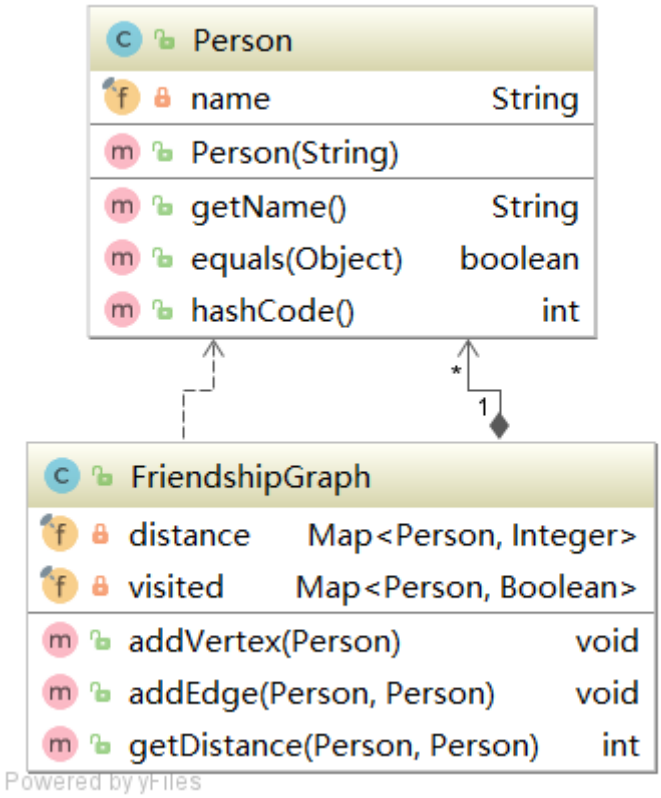
测试策略主要是根据 `FriendshipGraph` 中的图的类型进行测试，主要分为空图的测试、仅有顶点的图的测试、无环图的测试和有环图的测试。除了以上的新增加的测试用例，还有在 `Lab1` 中使用的测试用例，来验证在不改变客户端仍然可以使用。测试结果如下图：



代码覆盖度如下：



最终 Person 和 FriendshipGraph 的依赖关系的 UML 图如下

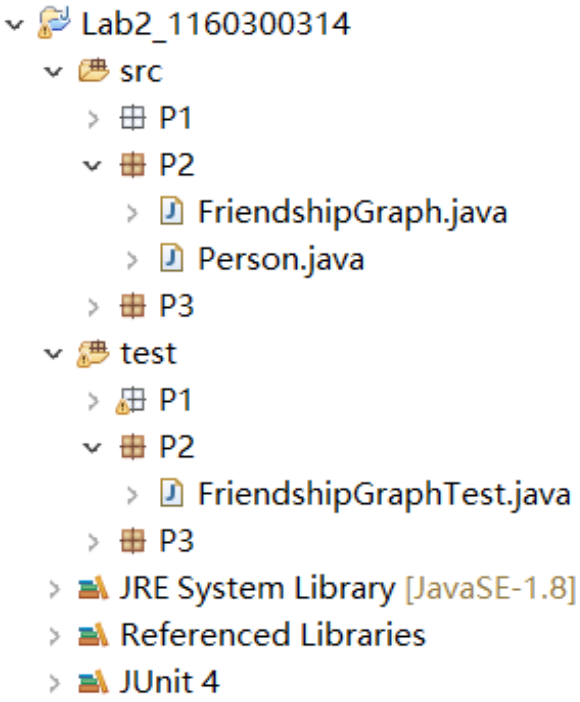


3.2.5 提交至 Git 仓库

```
git commit -m “finish P1”
git add -A
git push
```

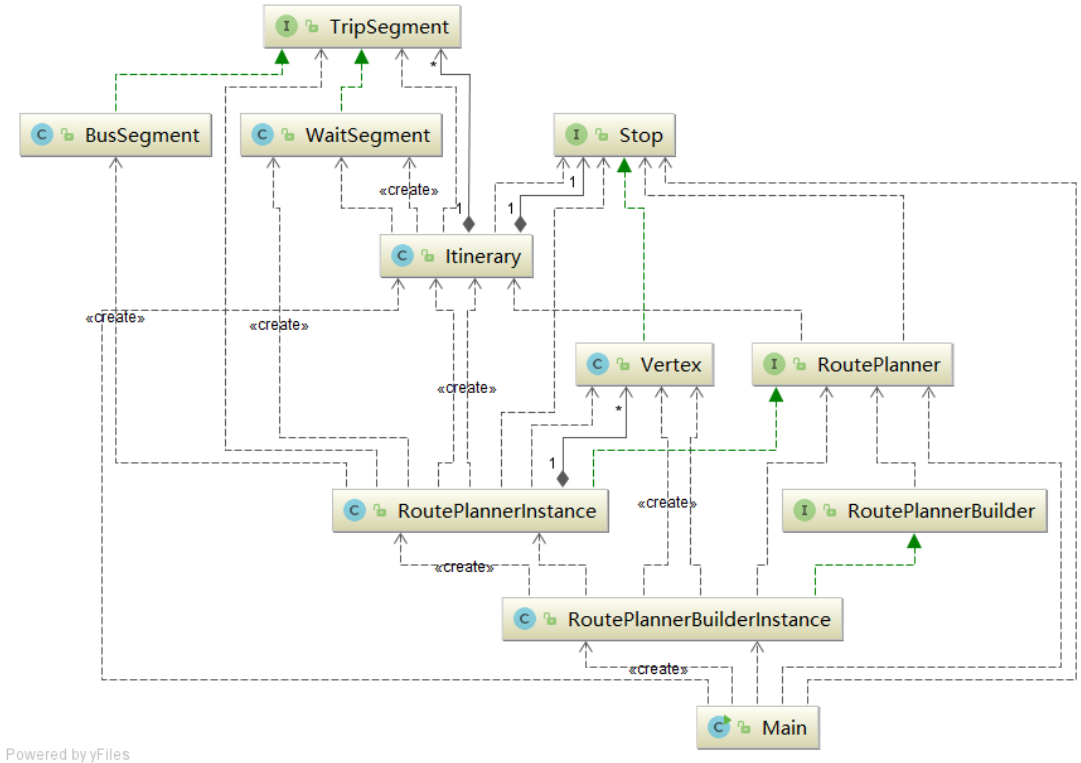
运行以上命令即可件项目增加至 GitHub 仓库中。

在这里给出你的项目的目录结构树状示意图。如下图



3.3 The Transit Route Planner（选做，额外给分）

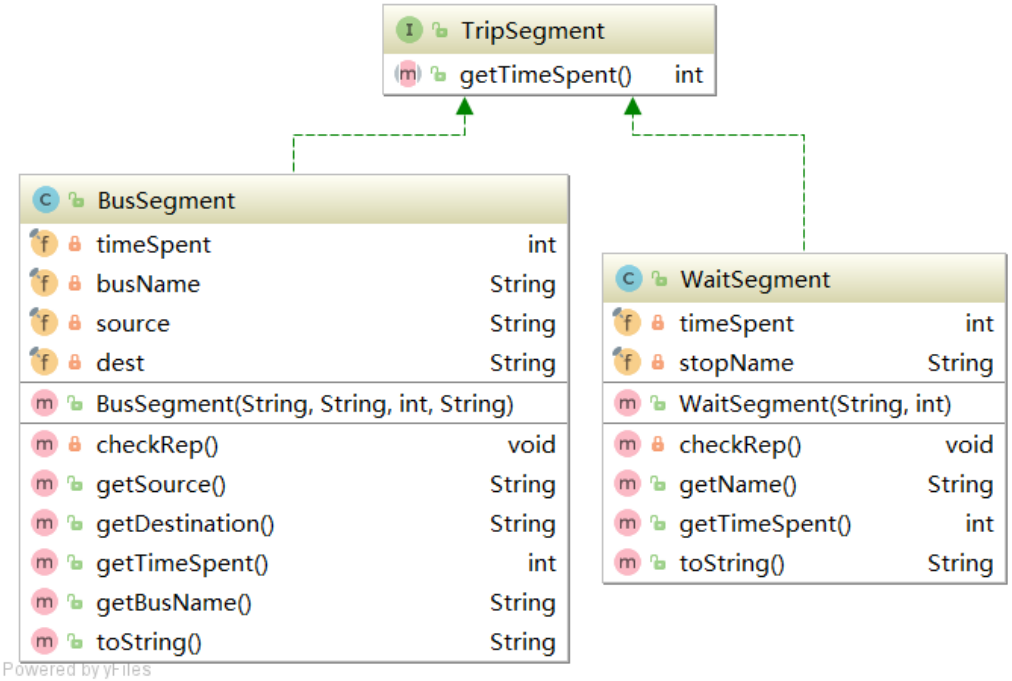
3.3.1 UML 图最终设计



最终设计 UML 图如上，通过以上设计来实现 P3 的任务。

3.3.1.1 TripSegment BusSegment WaitSegment 的实现

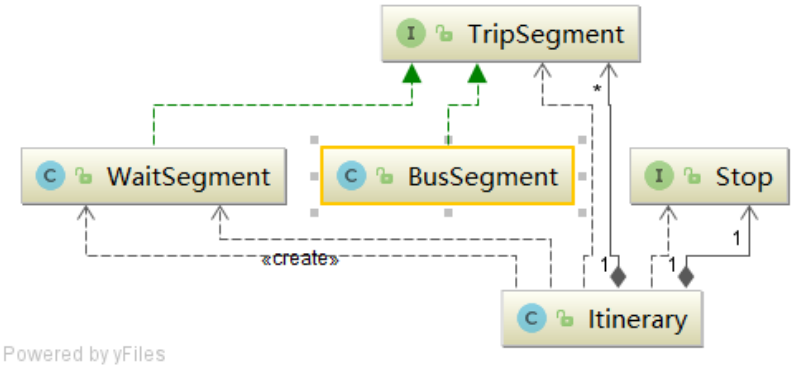
对于 TripSegment 接口的设计采用以下方式



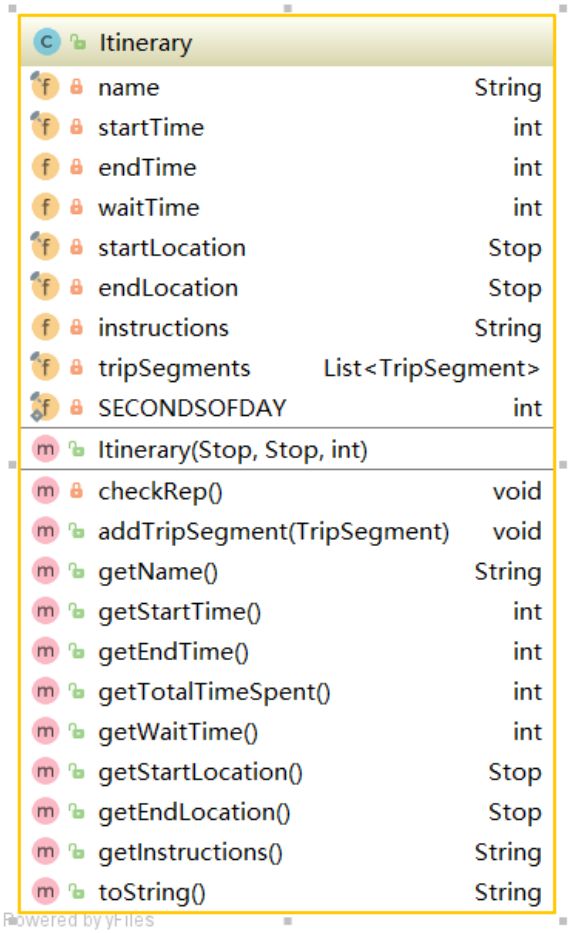
对于 BusSegment 表示在某一段乘坐公交车线路中的具体实现，里面记录了车辆行驶的时间和出发地和目的地以及乘坐公交车的名字。
对于 WaitSegment 表示在某地等待公交车的具体实现里面记录了 Rider 在某地的等待时间和公交车站的名字。

3.3.1.2 Itinerary 的实现

对于 Itinerary 类是表示一个人具体从某地出发去往某地的路线问题，所以在其中有一个 TripSegment 的序列表示某个人在其中的等待和乘坐公交车的具体次序和时间，另外还有出发地、目的地、出发时间等等 fields。并且由于 Itinerary 的实现是依赖于 Stop 接口和 TripSegment 接口，所以具体的依赖关系见下图：



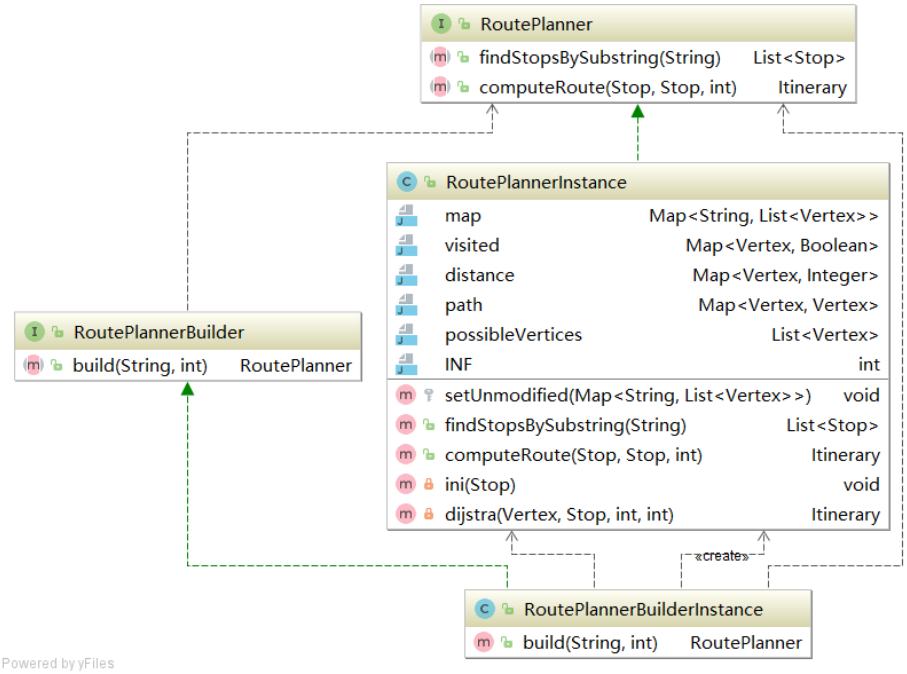
而对于 `Itinerary` 的具体实现如下图所示



3.3.1.3 `RoutePlanner` 以及 `RoutePlannerBuilder` 的实现

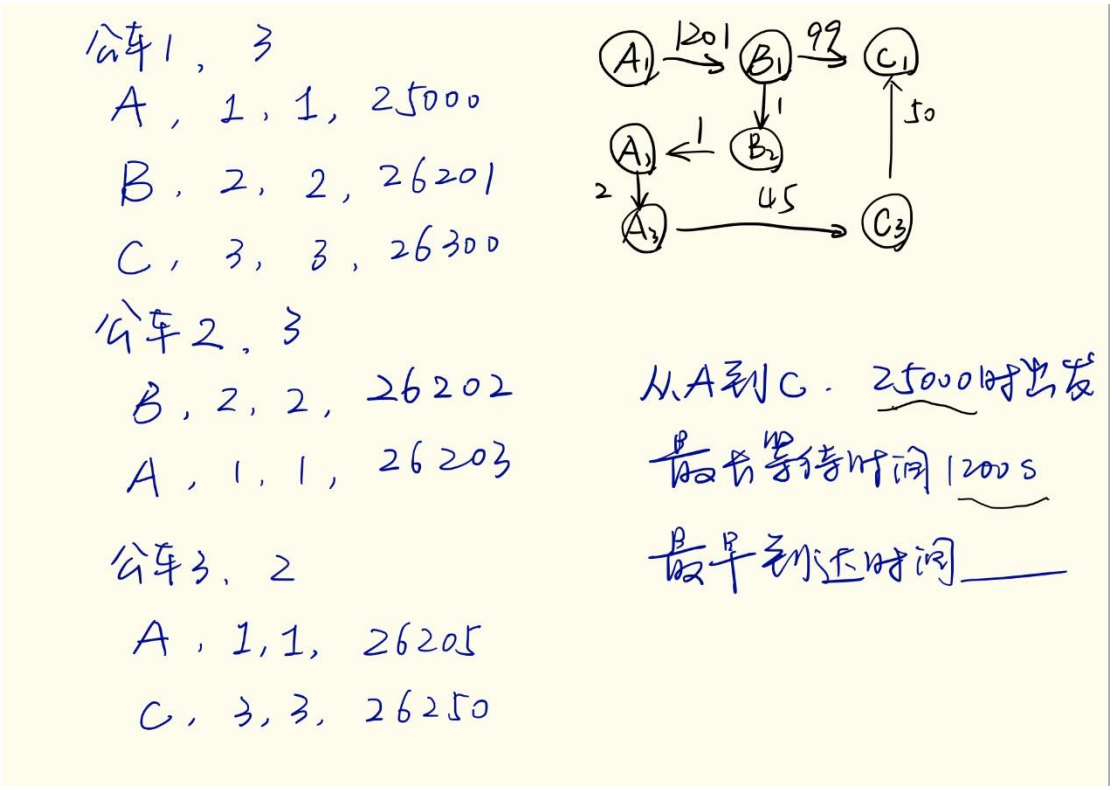
对于 `RoutePlanner` 来说，这是整个任务的核心函数，在其中要计算具体的乘坐公交车的路径。在这个任务中我选择将所有的**公交车所经过的车站按照到达时间进行划分**，说起来比较拗口，简单的来说就是**认为不同公交车经过的相同的车站是不同的顶点**。并且使用 `Dijstra` 来寻找最短路。建图的时候使用相同的车站名，在不同的公交车之间建立的有向边，其中权值表示等待时间，方向表示先后次序；在不同的车站相同的公交车之间建立有向边，权值代表该公交车的行驶时间，方向代表行驶方向。具体的实现见下图。

其中对于 `Dijstra` 来说，由于不同的出发时间相同的出发地点可能代表了不同的图中的顶点（由于乘坐公交车的不同），而 `Dijstra` 是解决单源最短路问题。所以在 `computeRoute` 中需要将所有可能的乘坐的公交车对应的顶点均跑一遍最短路，并从其中找到一个最短用时的 `Itinerary` 来作为最终答案返回给用户。



3.3.1.4 Main 以及实际测试效果

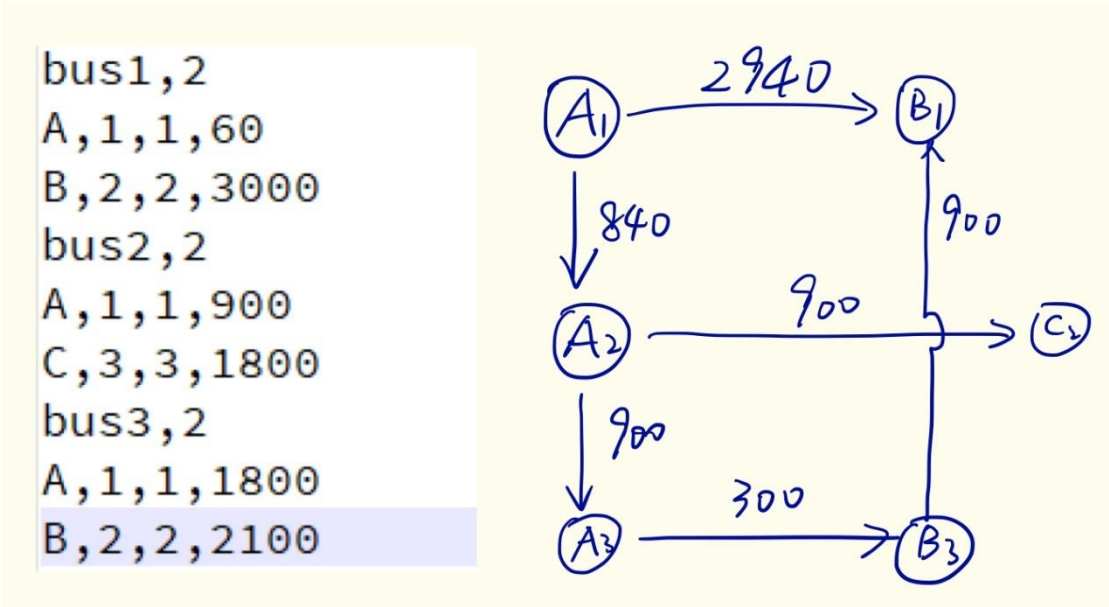
在最终的测试的时候使用的是一个比较特殊的运输图，如下：



在其中有一条非常特殊的路线可以比较好的测试完成情况，从 A 出发去往 C 地，在 25000 时出发，最短的路线由于 20 分钟的等待时间限制，所以采用的方法应该是从 A 去往 B，然后从 B 回到 A，再从 A 到达 C。
测试结果如下图所示：

```
Please input the source:
A
Please input the target
C
Please input the time you leave:
25000
From A To C At 25000
Start from A
Destination is C
Total time used 1250 second(s)
From A to B Take bus1 ,and used 1201 second(s)
Wait at B for 1 second(s)
From B to A Take bus2 ,and used 1 second(s)
Wait at A for 2 second(s)
From A to C Take bus3 ,and used 45 second(s)
```

另外由于最长等待时间为 20 分钟的限制，所以在图中跑出来的最短时间不一定为真正可行的最短时间，比如对于下边这个图：



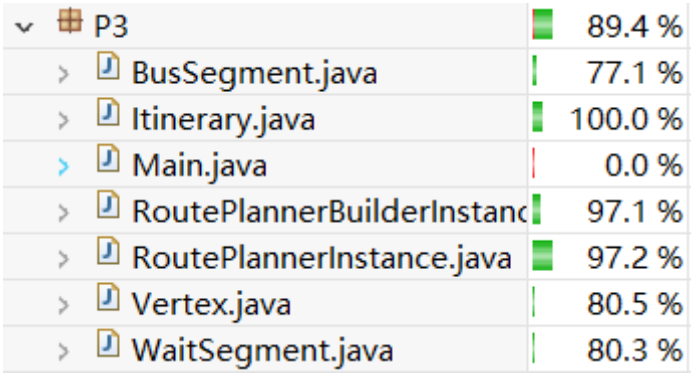
从 A 地出发在 0 时去往 B 地，如果直接利用最短路得到的结果为 A1->A2->A3->B3 实际由于不能再某一站连续等待的时间超过 20 分钟，这种最短路的结果实际为非法结果。

所以在使用 Dijkstra 的时候，需要在更新最短路径的收缩时，不能将连续两个顶点的名字相同，即有相同时间。因此需要特殊处理才可。

对于上面的测试用例，最终得到的结果如下图所示：

```
From A To B At 0
Start from A
Destination is B
Total time used 3000 second(s)
Wait at A for 60 second(s)
From A to B Take bus1 ,and used 2940 second(s)
```

以及代码测试覆盖度：



4 实验进度记录

请尽可能详细的记录你的进度情况。

日期	时间段	计划任务	实际完成情况
2018/3/12	13:45-17:00	书写 GraphInstanceTest	延时 2 小时左右
2018/3/12	20:00-23:00	完成 ConcreteEdgesGraph	按时完成
2018/3/13		完成 ConcreteEdgesGraphTest	按时完成
2018/3/14		完成 ConcreteVerticesGraph	按时完成
2018/3/15		写注释	延期
2018/3/16		写完已完成类的注释	完成
2018/3/17		泛型编程 Problem3	完成
2018/3/18		初步完成 P1	完成
2018/3/19		完成 P1 和 P2	完成
2018/3/20		完成 P3 的 UML 设计	延期
2018/3/21		完成 P3 的 UML 设计	延期
2018/3/25		完成 P3	完成
2018/3/26		写报告	延期至 3/27 完成

5 实验过程中遇到的困难与解决途径

1、 泛型编程的问题。由于以前没有写过有关泛型编程的 Java 程序，所以在转换为泛型的时候，特别谨慎。通过浏览其他博主写过的一些文章来解决这个问题。

2、 P3 中对于建图的问题。在最开始的时候认为只需要在每一个顶点中记录该地的 Stop 信息和所有经过此地的公交车信息即可，但最后发现这样将过多的工作推迟到搜索最短路的时候去做，而且如果使用 Dijstra 进行寻找最短的时间的话，不能回到已经经过的顶点。这对于一下问题来说是很难解决的，比如倒车的问题。后来决定采用拆点的方式，来区分不同的公

交车经过的不同的车站，将新实现 Stop 接口的类作为图中的节点。避免一些特殊情况无法处理。

6 实验过程中收获的经验、教训、感想

本节除了总结你在实验过程中收获的经验和教训，也可就以下方面谈谈你的感受（非必须）：

(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，通过 Lab2 中的 P2 任务有了更好的理解，就是我们可以在 Lab1 中根据已有的 Spec 和我们的需要来书写数据结构，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，主要需要考虑可能有的数据类型，对于尖括号的处理也不是很习惯。而不使用泛型编程的话，是以往编程中十分常用的方法，对于我来说写起来也更加顺手。

(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？
优势主要是我们实现“Fail fast”，也就是我们可以尽早发现错误，这样可以尽量在 Build Time 来发现错误，避免将错误带入更靠后的阶段造成更大的危险。
对于先写测试的方式，其实坦白说还是不是很适应。首先在不写的具体实现的时候，对于一些错误意识的还不是很清晰。一般我先写的测试用例在后期还需要继续向里面添加新的测试用例来补充没有覆盖到的情况，但是先写测试用例确实可以避免一些简单的错误出现，但对于更加细节的东西还是需要在完成部分实现后来补充。

(4) 本实验设计的 ADT 在三个应用场景下使用，这种复用带来什么好处？
避免重复造轮子，可以将已有的代码进行复用来尽量节省时间，而且对于问题的抽象能力的培养有了更好的锻炼。编程的时候不再只注意某一个应用场景，而是从很多相似的应用场景中发现共性和已经编写好的代码之间的相似之处。

(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？
避免表示外泄。其实表示外泄的影响，会让 Clients 对于内部的实现有了一些窥探，这种窥探有可能使客户在使用已经编写好的 ADT 的时候对于内部的一些书结构产生依赖，那么以后我再进行升级的时候，就可能不能

兼容其已有的使用代码；其次，如果用户端对于代码的目的不纯，可以利用 rep exposure 做一些超过客户权限的操作，影响更大范围的情况；再其次，内部的 rep 如何实现是对于一个程序员来说他的“饭碗”，如果被同行看到，甚至可能导致我的失业，所以 rep 的保护也像一种知识产权的保护。

(6) 关于本实验的工作量、难度、deadline。

本次实验 P1 和 P2 的工作量适中，难度适中；但对于 P3 的难度较大，相比前两个来说难度上升过大，而且由于缺少测试数据和部分要求也不够细致，导致了反复的情况，其实在无意中又放大了这个实验的难度。

Deadline 可以提前一周。

(7) 《软件构造》课程进展到目前，你对该课程有何体会和建议？

先说体会吧，在软件构造课程进行到现在，讲到了最核心的部分，确实是一些从未接触过的知识，但是很大部分的知识的抽象程度也很大，对于短时间理解还是有比较大的压力。

建议，对于 P3 这种叙述不是很清楚的作业，可以给出一组样例来更好地解释和说明 P3 所要求的任务，便于更好的了解 Spec，以及希望使用 MIT 的作业。