

**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 1](#_Toc510291757)

[2 实验环境配置 1](#_Toc510291758)

[3 实验过程 1](#_Toc510291759)

[3.1 Poetic Walks 1](#_Toc510291760)

[3.1.1 Get the code and prepare Git repository 1](#_Toc510291761)

[3.1.2 Problem 1: Test Graph <String> 2](#_Toc510291762)

[3.1.3 Problem 2: Implement Graph <String> 3](#_Toc510291763)

[3.1.3.1 Implement ConcreteEdgesGraph 3](#_Toc510291764)

[3.1.3.2 Implement ConcreteVerticesGraph 6](#_Toc510291765)

[3.1.4 Problem 3: Implement generic Graph<L> 9](#_Toc510291766)

[3.1.4.1 Make the implementations generic 9](#_Toc510291767)

[3.1.4.2 Implement Graph.empty() 9](#_Toc510291768)

[3.1.5 Problem 4: Poetic walks 10](#_Toc510291769)

[3.1.5.1 Test GraphPoet 10](#_Toc510291770)

[3.1.5.2 Implement GraphPoet 11](#_Toc510291771)

[3.1.5.3 Graph poetry slam 11](#_Toc510291772)

[3.1.6 Before you’re done 12](#_Toc510291773)

[3.2 Re-implement the Social Network in Lab1 13](#_Toc510291774)

[3.2.1 FriendshipGraph类 13](#_Toc510291775)

[3.2.2 Person类 13](#_Toc510291776)

[3.2.3 客户端main() 14](#_Toc510291777)

[3.2.4 测试用例 14](#_Toc510291778)

[3.2.5 提交至Git仓库 15](#_Toc510291779)

[3.3 The Transit Route Planner（选做，额外给分） 16](#_Toc510291780)

[3.3.1 UML图最终设计 16](#_Toc510291781)

[3.3.1.1 TripSegment BusSegment WaitSegment的实现 17](#_Toc510291782)

[3.3.1.2 Itinerary的实现 17](#_Toc510291783)

[3.3.1.3 RoutePlanner 以及 RoutePlannerBuilder的实现 18](#_Toc510291784)

[3.3.1.4 Main以及实际测试效果 19](#_Toc510291785)

[4 实验进度记录 21](#_Toc510291786)

[5 实验过程中遇到的困难与解决途径 21](#_Toc510291787)

[6 实验过程中收获的经验、教训、感想 22](#_Toc510291788)

# 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

# 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>

# 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！）。

## 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。

### 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。

### 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 | |  |
| test remove | |  |
| test set | |  |
| test vertices | |  |
| test source | |  |
| test target |  | |

### 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。

#### Implement ConcreteEdgesGraph

必须使用下面的数据结构来实现ConcreteEdgesGraph



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

|  |
| --- |
|  |

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。具体书写如下：

|  |
| --- |
|  |

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

|  |
| --- |
|  |

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。

|  |
| --- |
|  |

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：

|  |
| --- |
|  |

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

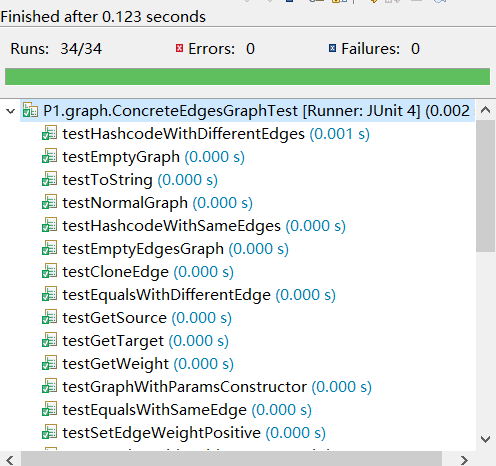
与source函数相对应，找到以source为源点的所有有向边的集合，并返回一个包括target和有向边权值对应关系的Map即可，同样遍历一遍边集找到所有符合条件的Edge加入Map后返回。

⑦toString函数

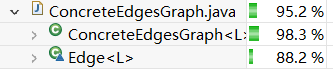
Override继承Object的toString函数，重写后的toString函数要尽量符合人的阅读习惯，所以返回的String中包含这个图中边数、顶点数、边集和顶点集中各个元素，并且排版尽量友好。

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：



#### Implement ConcreteVerticesGraph

必须使用下面的数据结构并且不能增加新的的的fields



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

|  |
| --- |
|  |

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：

|  |
| --- |
|  |

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

|  |
| --- |
|  |

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：

|  |
| --- |
|  |

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

|  |
| --- |
|  |

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：

|  |
| --- |
|  |

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

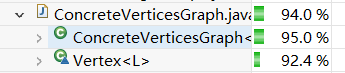
注意在删掉顶点的时候需要将其邻接的边全部删除。

④sources函数和targets函数

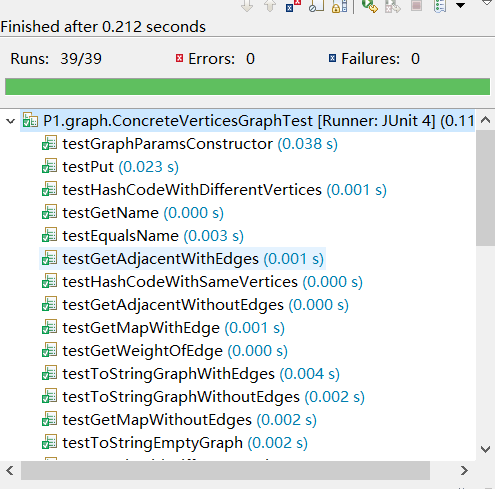
在链表中扫一遍，将符合要求的对应关系加入返回的map中即可。

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：



### 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。

#### 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。

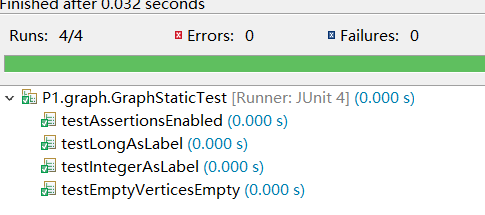
#### Implement Graph.empty()

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

|  |
| --- |
|  |

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：



### 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。

#### 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 |  |
| Graph the poem  由诗所生成的图 |  |
| poem(in) |  |
| toString |  |

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。

#### 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

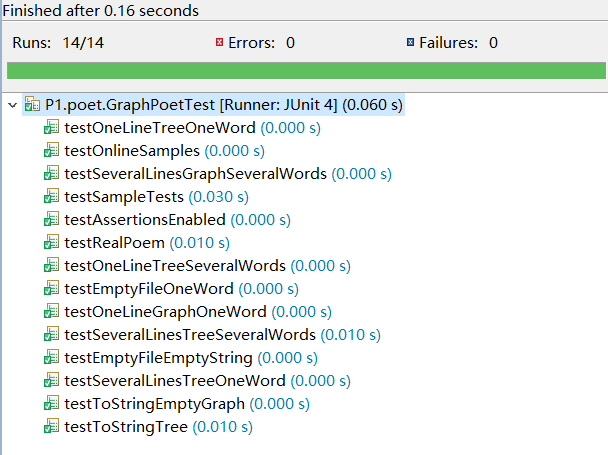
将文件中的诗歌按照行（hang）读入，利用空格将其分开，去掉空的字符串，在任何相邻的两个中间加入一条边，并按照边出现的数量更新权值建图即可。

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

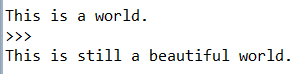


代码覆盖度如下



#### 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”



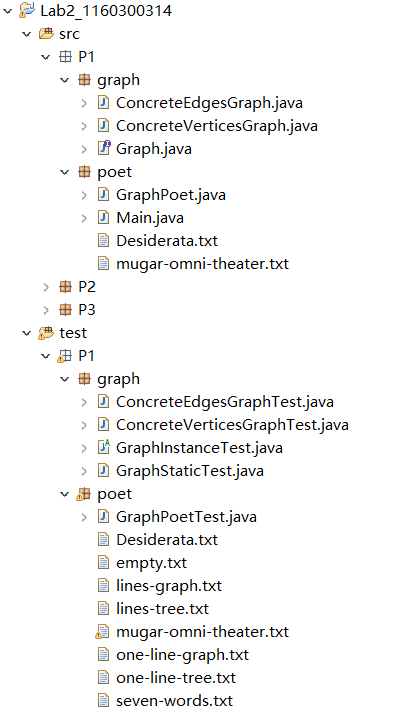
### Before you’re done

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。

|  |
| --- |
|  |

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。



## 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.

### 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。

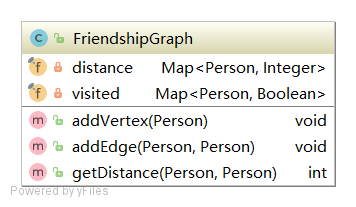
2、实现addEdge函数

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

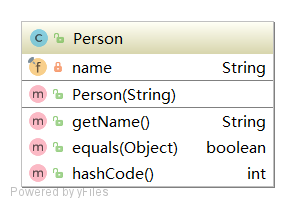
仍然是利用BFS来实现找最短路，大部分实现仍然利用已有的函数来实现。在寻找某一个顶点邻接的顶点，只需要利用targets函数即可。

UML图见下



### 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：

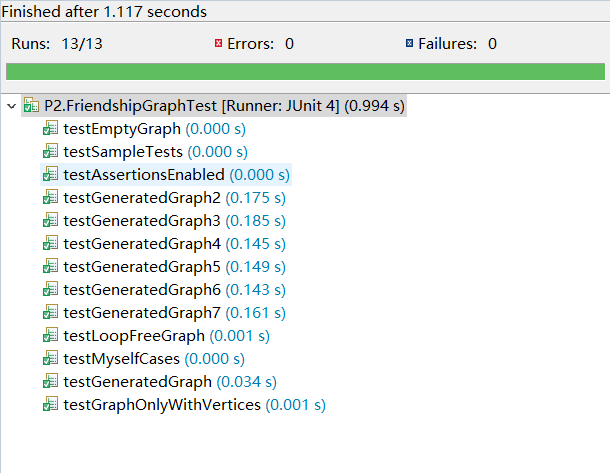


### 客户端main()

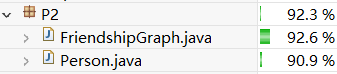
重新利用Lab1中已有的main客户端即可，由于所有Lab2中的所有实现都是没有改变addVertex、addEdge和getDistance三个函数，只需要重新使用即可使用。

### 测试用例

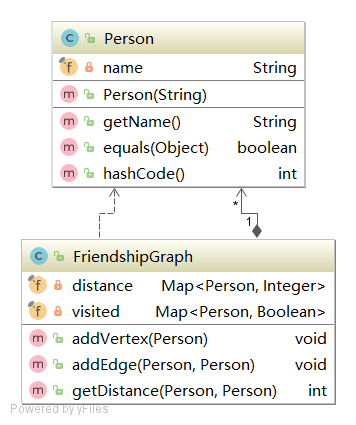
测试策略主要是根据FriendshipGraph中的图的类型进行测试，主要分为空图的测试、仅有顶点的图的测试、无环图的测试和有环图的测试。除了以上的新增加的测试用例，还有在Lab1中使用的测试用例，来验证在不改变客户端仍然可以使用。测试结果如下图：



代码覆盖度如下：



最终Person和FriendshipGraph的依赖关系的UML图如下

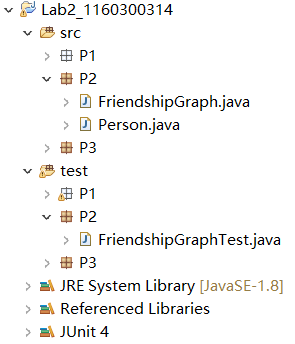


### 提交至Git仓库

|  |
| --- |
|  |

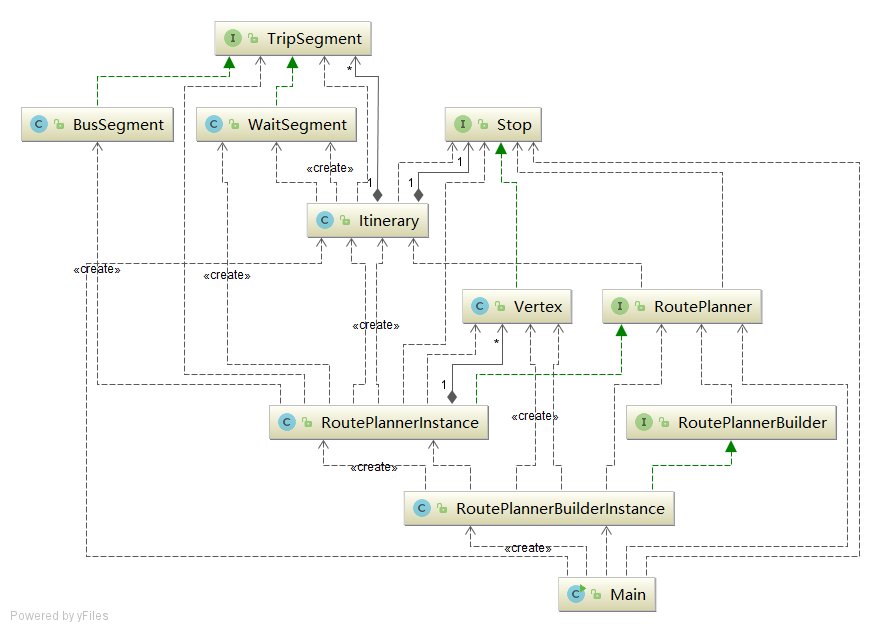
运行以上命令即可件项目增加至GitHub仓库中。

在这里给出你的项目的目录结构树状示意图。如下图



## The Transit Route Planner（选做，额外给分）

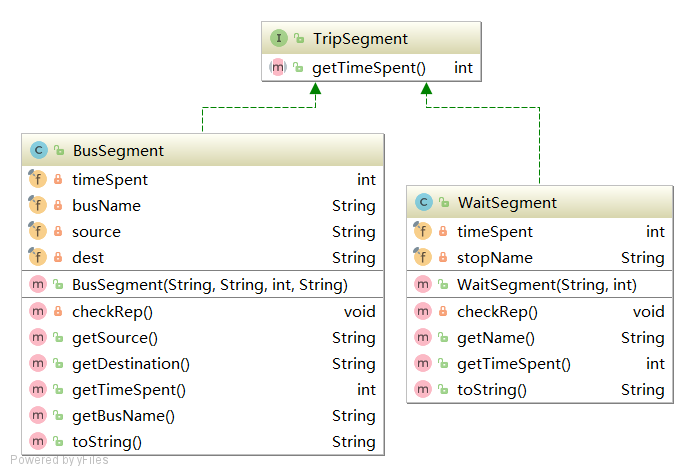
### UML图最终设计



最终设计UML图如上，通过以上设计来实现P3的任务。

#### TripSegment BusSegment WaitSegment的实现

对于TripSegment接口的设计采用以下方式

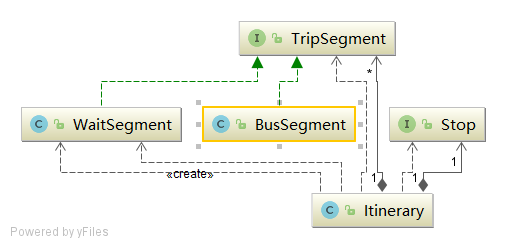


对于BusSegment表示在某一段乘坐公交车线路中的具体实现，里面记录了车辆行驶的时间和出发地和目的地以及乘坐公交车的名字。

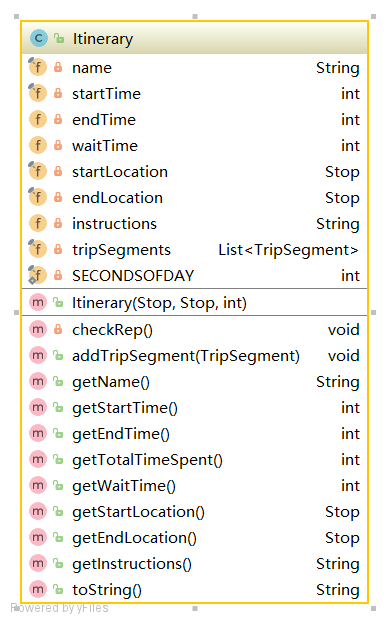
对于WaitSegment表示在某地等待公交车的具体实现里面记录了Rider在某地的等待时间和公交车站的名字。

#### 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:



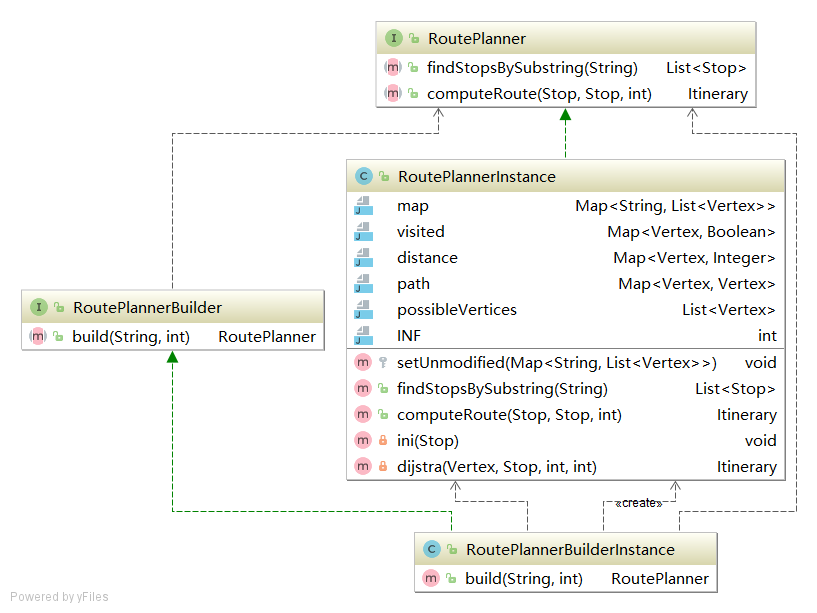
而对于Itinerary的具体实现如下图所示



#### RoutePlanner 以及 RoutePlannerBuilder的实现

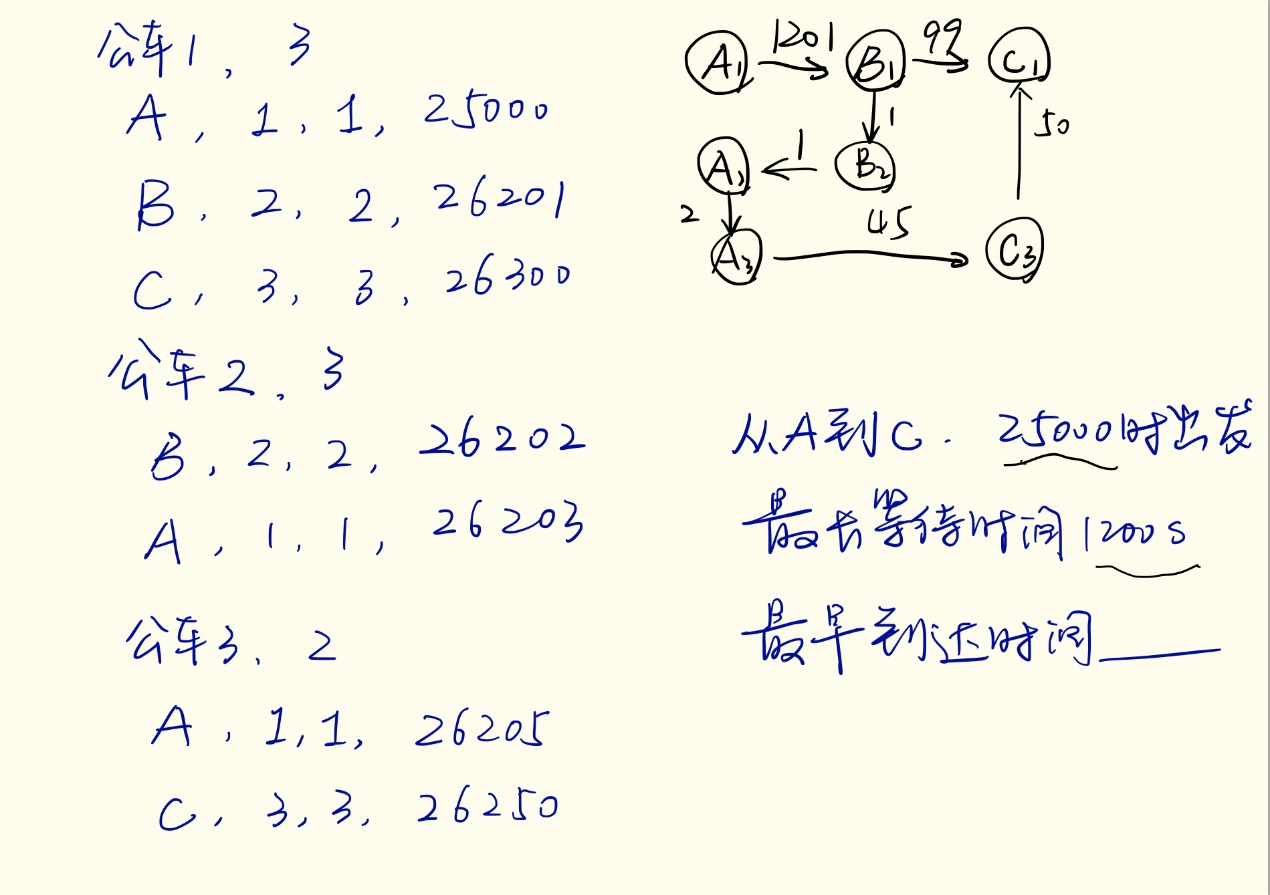
对于RoutePlanner来说，这是整个任务的核心函数，在其中要计算具体的乘坐公交车的路径。在这个任务中我选择将所有的**公交车所经过的车站按照到达时间进行划分**，说起来比较拗口，简单的来说就是**认为不同公交车经过的相同的车站是不同的顶点**。并且使用Dijstra来寻找最短路。建图的时候使用相同的车站名，在不同的公交车之间建立的有向边，其中权值表示等待时间，方向表示先后次序；在不同的车站相同的公交车之间建立有向边，权值代表该公交车的行驶时间，方向代表行驶方向。具体的实现见下图。

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。



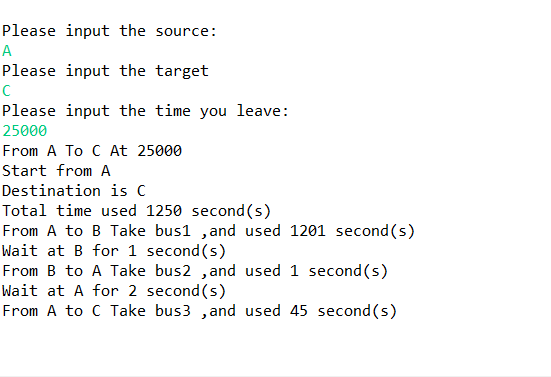
#### Main以及实际测试效果

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：

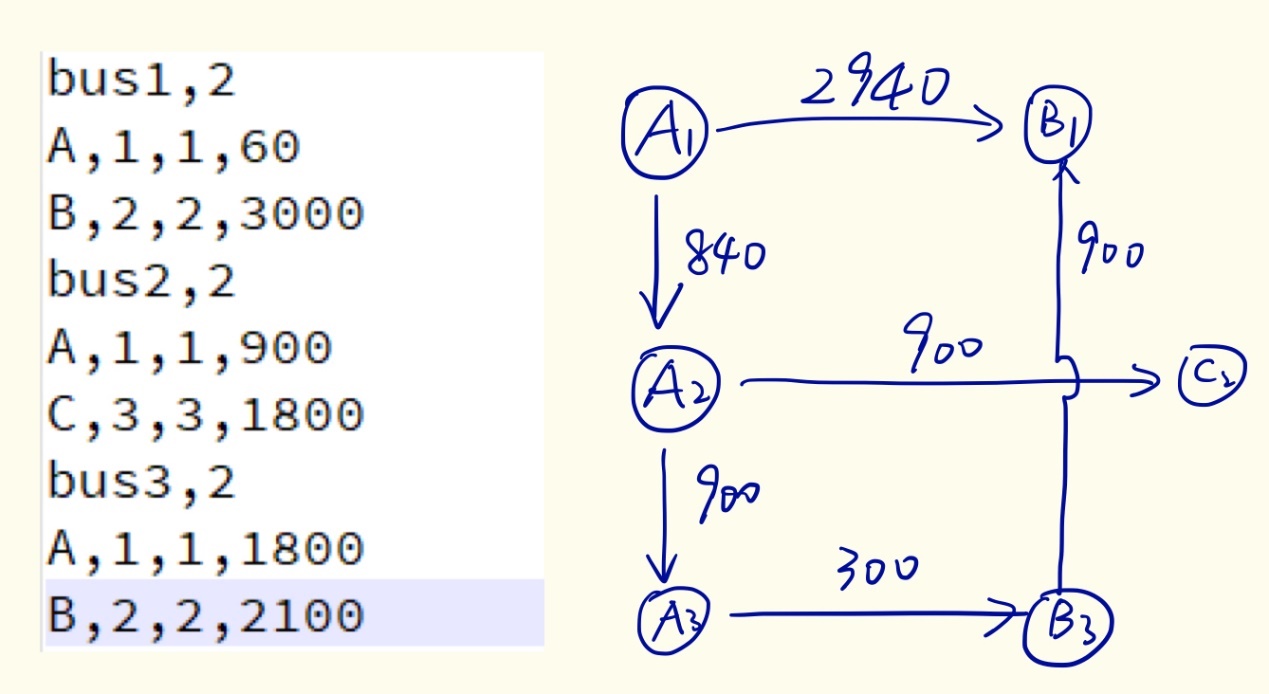


在其中有一条非常特殊的路线可以比较好的测试完成情况，从A出发去往C地，在25000时出发，最短的路线由于20分钟的等待时间限制，所以采用的方法应该是从A去往B，然后从B回到A，再从A到达C。

测试结果如下图所示：



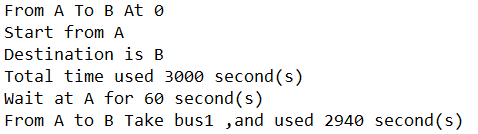
另外由于最长等待时间为20分钟的限制，所以在图中跑出来的最短时间不一定为真正可行的最短时间，比如对于下边这个图：



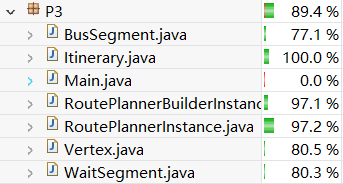
从A地出发在0时去往B地，如果直接利用最短路得到的结果为A1->A2->A3->B3实际由于不能再某一站连续等待的时间超过20分钟，这种最短路的结果实际为非法结果。

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：



以及代码测试覆盖度：



# 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 |

# 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。

# 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。

1. 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。

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。

1. 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。

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？

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。

1. 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。

1. 《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。