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 3 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

这一次的实验覆盖课程的第 3、5、6 章的内容，目标是编写具有可复用性和可维护性的软件，主要使用以下软件构造技术：

- 子类型、泛多态重写载
- 继承、代理组合
- 常见的 OO 设计模式
- 语法驱动的编程 、正则表达式
- 基于状态的编程
- API 设计

2 实验环境配置

Java 环境在以前的实验中就已经配置好。
Git 环境也在以前的实验中配置好。

在这里给出你的 GitHub Lab3 仓库的 URL 地址（Lab3-学号）。
https://github.com/ComputerScienceHIT/Lab3-1160300314

3 实验过程

请仔细对照实验手册，针对每一项任务，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 待开发的四个应用场景

3.1.1 GraphPoet

对于 GraphPoet 这个应用场景来说，由于其不需要直接在文本上生成出某个图，而是选择根据固定的图的输入格式来形成 GraphPoet，所以对于这个应用场景主要的需求在于组织一个单模有向并且带权的图。

3.1.2 SocialNetwork

对于 SocialNetwork 这个应用场景，主要是维持一个比较特殊的 Spec，即所有边的权值之和为 1，这个条件在所有的该应用的场景下均可以维持。另外这个应用场景允许多重边的存在。总的来说，SocialNetwork 需要维持一个单模、有向、带权的多重图。

3.1.3 NetworkTopology

对于 NetworkTopology 这个应用场景主要是需要维持一个网络拓扑关系，与上面两个场景的不同就在于每一个网络节点可能有多个不同的类型，即该图为多模图。并且由于任意两个网络节点之间的联系都是双向的，所以这是一个无向图。另外我们使用边上的权值记录网络带宽，又是一个带权图。

3.1.4 MovieGraph

对于 MovieGraph 来说其不同于以上几个图的点在于这是一个带有超边的图。并且边的种类也有很多种，允许多个顶点之间有多种边。所以 MovieGraph 需要维持一个带有超边的单重图多模图。

3.2 基于语法的图数据输入

基于语法的图的输入，其实主要在于如何将输入的文本合理解析。在所有的输入文本中主要有四种不同的类型。

3.2.1 GraphType、GraphLabel、VertexType、EdgeType

对于上面四种情况的输入，都属于用等号分开的两部分：前半部分表示的是这个操作的具体类型，比如是输入的节点类型等等。但对于其中的具体表示，比如后半部分是不是真的是一种顶点的类型，如果也选择在此处进行判断会使这部分的程序过于臃肿。因此，基于以上的考虑，选用了

```
^(GraphType|VertexType|EdgeType|GraphName) =
        (\"[\w]+\")(, ?\"[\w]\")*$
```

进行匹配以上四种类型的输入语句。

3.2.2 Vertex

对于顶点的信息输入主要考虑的是，是否符合

$$Vertex = <Label_m, type_m, <attr_1, ..., attr_k>>$$

这种形式；换句话说就是，是否包含了 Label、type 这种必要的信息。以及对于一些没有属性值的简化到下面这种形式

```
Vertex = <Label_m, type_m>
```

基于以上的两点考虑，选择使用

```
^Vertex = <\"[\w]+\", ?\"[A-Za-z]+\"(, ?<(\"[\w.]+\")(, \"[\w.]+\")*>)?>
```

进行匹配，对于 Type 类型肯定是基于纯字母的一个表示，对于所有的参数仅仅判断是不是由[a-zA-Z_.]这几种符号组成（小数点是为了保证在网络拓扑图中所有的 IP 地址都可以合法的输入，但具体对于 IP 地址合法性的检测作为 Vertex 内部的检查属性进行下放）。

3.2.3 Edge

对于一般的边的输入信息主要考虑的是，是否符合下面这种形式：

```
Edge = <Label, type, Weight, StartVertx, EndVertex, Yes|No>
```

这种简单边的表示，主要保证所有需要的信息均有输入即可。另外需要注意最后的一个属性保持仅可以是 Yes 或者 No 即可。所以使用的正则匹配是

```
^Edge = <\"[\w]+\", \"[a-zA-Z]+\", \"([+-]?[0-9]*\.?[0-9]+|[0-
    9]+\.?[0-9]*)\", \"[\w]+\", \"[\w]+\", \"(Yes|No)\">$
```

其中主要处理麻烦的一点是对于 Weight 这个属性，需要保证这是一个非负的浮点数或者整数或者-1，仅仅有这三种情况。但此处如果直接卡死这三种情况可能会使正则表达式的长度更长，以及为了遵循责任单一原则，将责任尽可能的分配小而且仔细。所以仅仅匹配所有的合法的浮点数和整数。对于其余的情况下放到 Edge 中再进行判断。

3.2.4 HyperEdge

对于超边来说，主要就在于所有的超边输入时不同于简单边的输入。需要满足：

```
HyperEdge = <Label, Type, {Vertex_1,…,Vertex_n}
```

即超边需要满足为顶点集的一个非空子集的性质。
最终使用的正则表达式如下：

```
^HyperEdge = <\"[\w]+\", \"[a-zA-Z]+\", \{\"[\w]+\"(, \"[\w]+\")*\}>$
```

3.3 面向复用的设计：Graph<L, E>

3.3.1 Graph<L, E>

对于 Graph 接口的设计，其实本质是从 Lab2 中进一步抽象，将顶点和边同时作为泛型参数。另外提供有一些操作，由于只需要定义方法的 Signature，并且没有增加新的方法除了实验手册中已经提到的方法进入 Graph 接口，所以整体的设计还是比较轻松的。

3.3.2 ConcreteGraph

对于实现 Graph 接口的 ConcreteGraph 类，选用了方案 2 作为最终的实现方式。

```
public class ConcreteGraph<L extends Vertex, E extends Edge>
                                implements Graph<L, E>
```

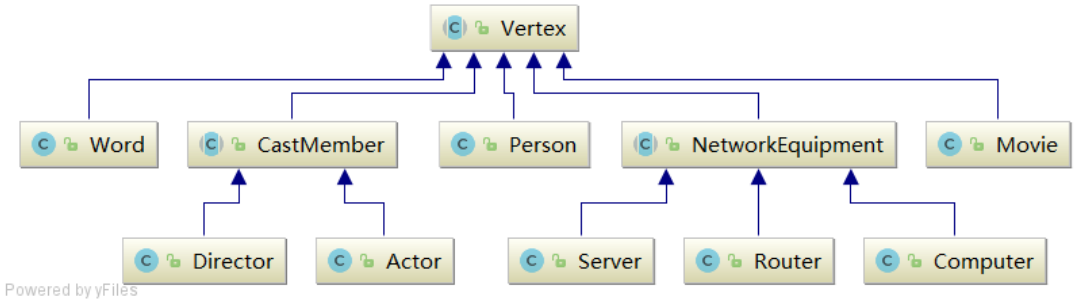
并使用顶点集合和边的集合来维护整个图中的顶点信息和边的信息，所以 ConcreteGraph 属性信息如下表示：

```
    private final Set<L> vertexSet = new HashSet<>();
    private final Set<E> edgeSet = new HashSet<>();
    private String name = "";
```

并且针对这种属性来实现接口中已经定义好的操作，此处主要说一下比较复杂的 removeVertex(L v)函数，在这个函数里需要在删除顶点的时候将与其邻接的边一并删除。但是对于超边，我们删除掉超边当且仅当该超边中只有这一个顶点。

3.4 面向复用的设计：Vertex

根据实验手册中提到的各种基于 Vertex 的具体类之间具有的共性，进行了再一次的抽象，具体的抽象后 vertex 包中的 UML 图如下：



分开来看，分别针对四个不同的应用场景分别来说。
对于 Vertex 来说，其中的 Label 属性和 getLabel 操作都是所有 Vertex 的子类中所必须的，对于 fillVertexInfo 来说，则需要在子类中更加具体的重写。

3.4.1 GraphPoet

在这个应用场景中只有一种顶点类型即 Word，并且 Word 类型的顶点没有除了 Label 之外的其他属性，所以只需要将 Vertex 中 Override 的 Object 的那几个方法再次复写即可以及将必须复写的 fillVertexInfo 写成没有函数体的即可。

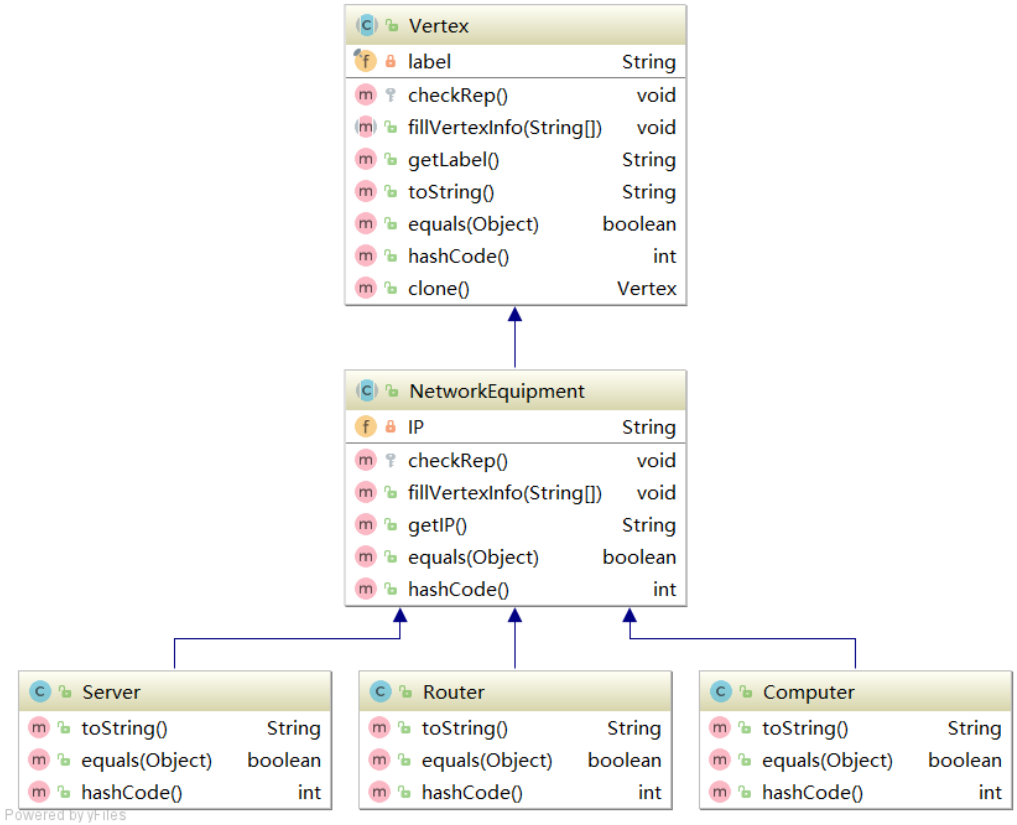
3.4.2 SocialNetwork

在这个应用场景中，也仅仅有一种 Person 类型的顶点，但是其有两种不同的属性，而且与后面将会提到的 MovieGraph 中的 Director 和 Actor 仅有顺序不同。所以我们在这里，不仅仅需要将其继承来自 Object 父类中的 hashCode 重写，最为重要的是我们需要在 fillVertexInfo 中保证输入的参数是符合顺序的

3.4.3 NetworkTopology

在网络拓扑关系中，由于其是一个多模图，并且所有可能出现在 NetworkTopology 中的顶点类型都是符合有 Label 和一个属性 IP 地址组成，所有我们在已有的抽象上增加一层的抽象，将 Computer、Router 和 Server 共同抽象到一个新的抽象类 NetworkEquipment 中。这样在 NetworkEquipment 中我们将 getIP 操作和 IP 属性定义好并且重写好 fillVertexInfo 这样在 Computer 等子类中仅仅需要重写 equals 等来自 Object 父类中的函数即可。

继承关系的 UML 图如下：

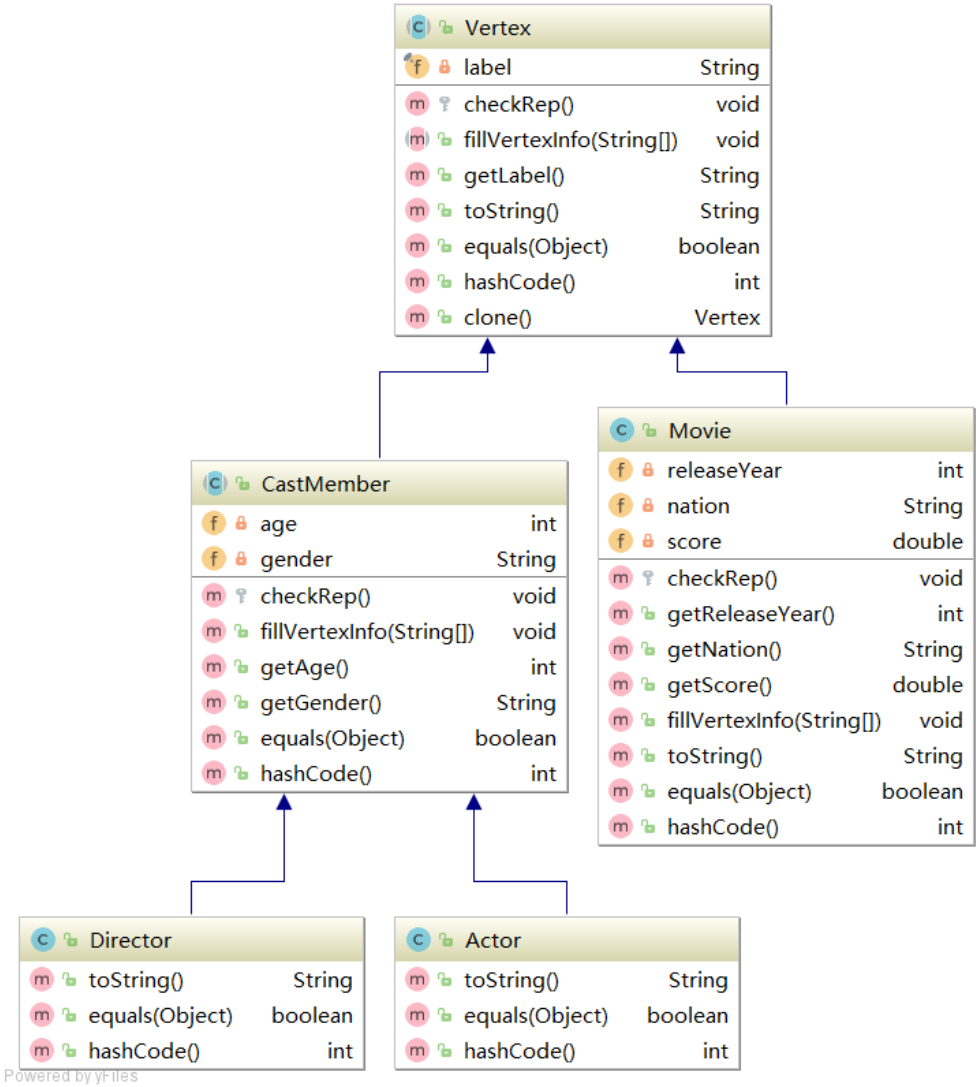


3.4.4 MovieGraph

在最后一个应用场景中，一共有三种不同类型的顶点，其中对于 Director 和 Actor 来说都是有两种属性，age 和 gender，所以我们在 Director 和 Actor

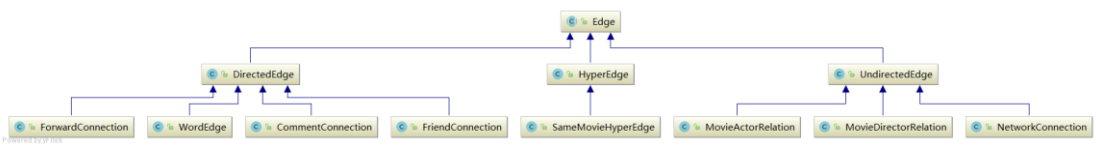
上增加一层的抽象，形成抽象类 CastMember，在这里面写好 gender 和 age 的属性及其相关的操作。对于 Movie 类而言，直接继承 Vertex 重写相关的函数和定义自身属性即可。

继承关系的 UML 图如下图：



3.5 面向复用的设计：Edge

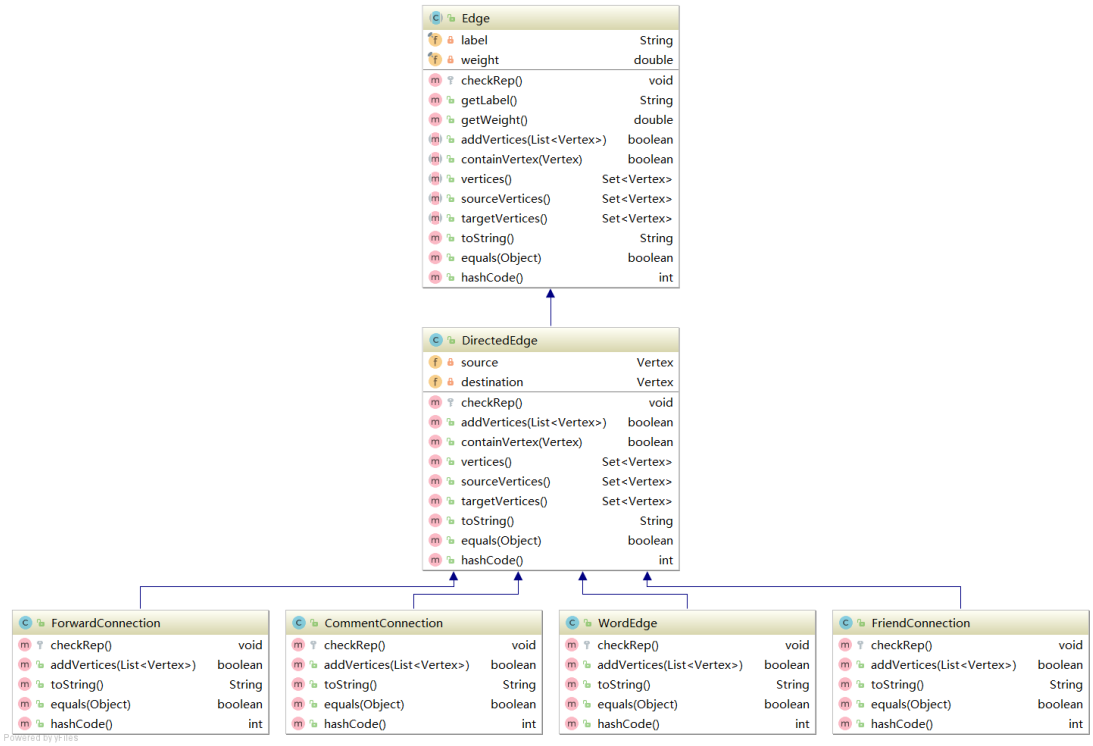
根据实验手册中提到的各种边的类型，对于抽象类 Edge 类而言，主要功能由其 DirectedEdge、UndirectedEdge 和 HyperEdge 类所完成。对于具体应用中使用的边的类型都是有上面提到的三个具体的类所派生出来的。对于 edge 包中的继承关系用 UML 图表示如下：



分别来看各个直接继承 Edge 抽象类的子类。

3.5.1 DirectedEdge

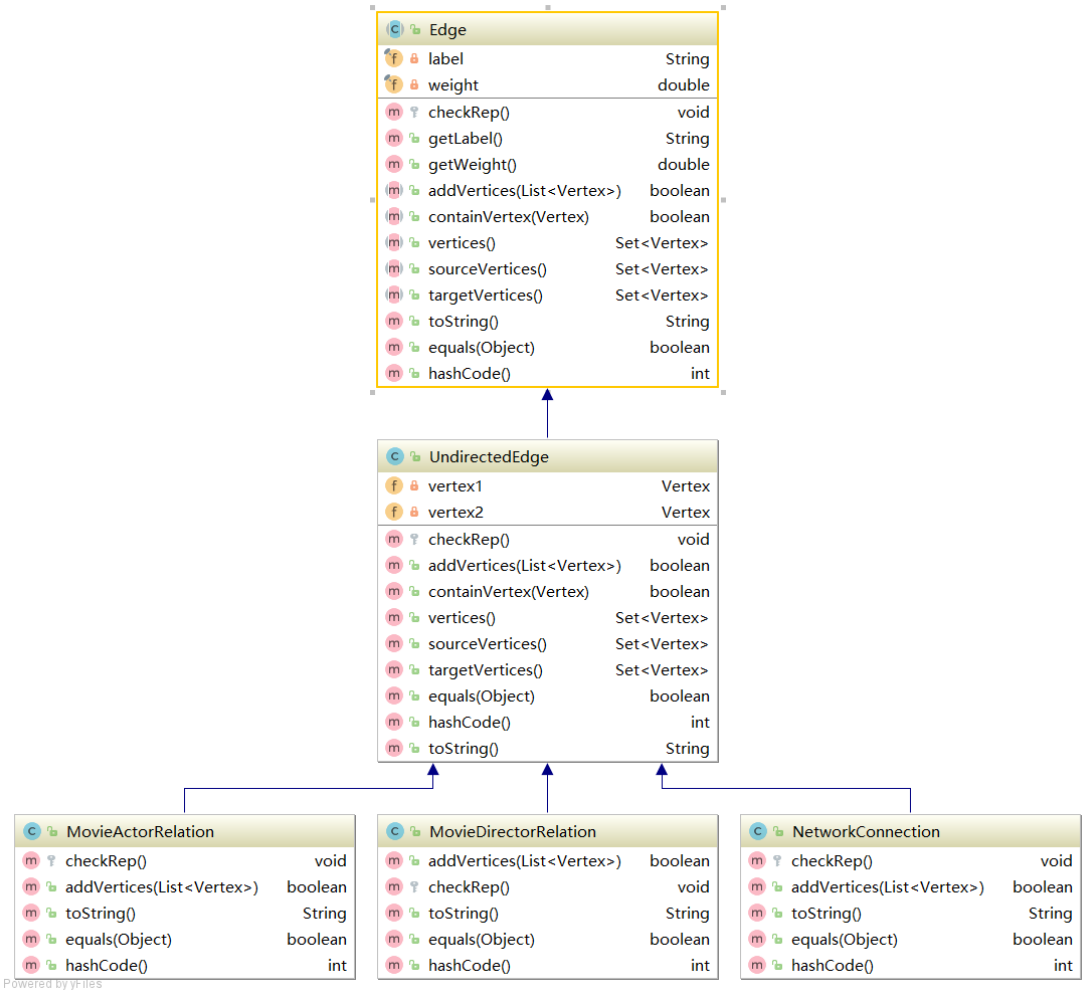
对于有向边，属于仅有起始点和目标点两个顶点组成的简单边，所以组织其内部的属性可以直接用两个顶点 Vertex 来记录。具体的继承关系如下图：



可以看到在 DirectedEdge 这个类中我们已经定义好了所有在 Edge 抽象类中没有实现的抽象方法。这样在继承于 DirectedEdge 的子类中，仅仅重写了来自 Object 的 hashCode 等用于判断相等的函数；另外由于所有子类都有具体的应用场景，并不是所有类型的顶点都可以加入任意一条边中。这种对于边所邻接的顶点的限制通过在继承 DirectedEdge 的子类中增加判断条件进行限制。

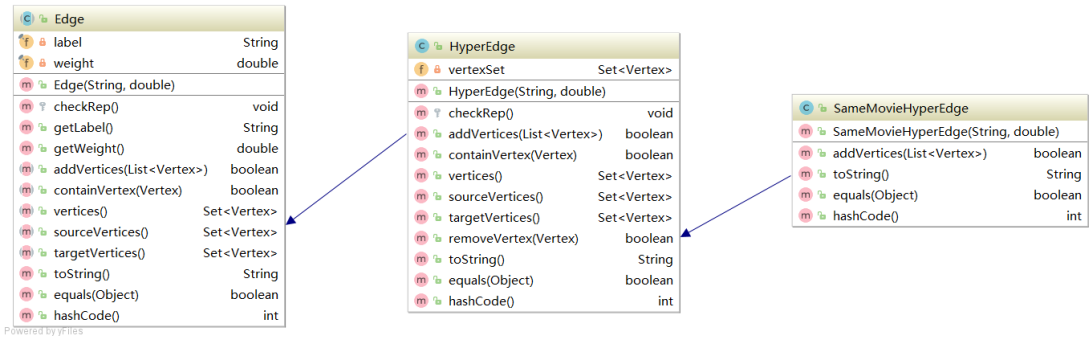
3.5.2 UndirectedEdge

对于无向边，其本质是与 DirectedEdge 相同的。仅仅在于 DirectedEdge 更加一般化，所以 UndirectedEdge 本质上也可以直接继承于 DirectedEdge。但为了区分三种不同的边的类型，在这里仍然选择将 UndirectedEdge 作为 Edge 的子类。其内部的组织顶点的形式，由于无向边不区分出发点与目的地，仅仅需要两个没有顺序关系的两个顶点即可，具体的继承关系见下方的 UML 图：



3.5.3 HyperEdge

对于超边来说，其不同于简单边，其中的顶点个数一般为多个，所以其属性中采用 Set<Vertex>来组织包含于其中的顶点。具体来看，在超边中我们是允许直接在其中删除某些顶点，只要在删除之后仍然可以保持一种合法的超边存在形式即可。所以需要在其中增加一个新的方法，以满足在超边中的特殊操作。具体的 UML 图如下（为了排版，此处的 UML 图没有采用严格的上下关系）：



3.6 可复用 API 设计

根据 façade 模式面向 ConcreteGraph<L, E>设计可复用的 API。主要就是将多个针对图的计算统一的划分到一个类中实现，下面说明一下其具体的代码实现。

3.6.1 degreeCentrality 的计算

对于度中心性而言，与其余两种中心性不同，此处不光需要实现针对图中的某个顶点的度中心性的计算，还需要计算图的度中心性。此处需要利用方法的重载，设计两个参数列表不同的函数即可。

另外对于有向图而言，如果直接使用 degreeCentrality 来计算顶点的中心性，设计为无论是出度还是入度在计算的时候均直接作为度进行计算，但是不进行累加计算。具体的代码如下（仅用顶点度中心性计算来说明）

```
public static <L extends Vertex, E extends Edge>
   double degreeCentrality(Graph<L, E> g, L v) {
        long degree = 0;
        Set<E> edges = g.edges();
        for (E edge : edges) {
            if (edge.sourceVertices().contains(v) ||
                    edge.targetVertices().contains(v)) {
                degree++;
            }
        }
        return degree;  // As wiki defined
    }
```

3.6.2 ClosenessCentrality 的计算

计算图中某顶点的 ClosenessCentrality，就是计算该顶点到所有剩余顶点的最短路径之和的倒数，用 Wiki 上的定义表示就是：

$$C(x) = \frac{1}{\sum_y d(y,x)}$$

对于非连通图，这个公式的定义没有具体说明。这此处的 post-condition 中进行限制，如果需要计算 ClosenessCentrality 的顶点与其余顶点之间没有路径的

情况，那么将返回 0 作为该顶点的 ClosenessCentrality。

3.6.3 betweennessCentrality 的计算

对于 betweennessCentrality 的计算，这里没有再重写一个计算的轮子，而是借用已有的 Jung 中提供的计算方法。将已有的图转化为 Jung 中所定义的图，然后直接利用其进行计算。这样的复杂度是一个图的复制复杂度 O(|V| + |E|)和 Jung 计算 betweennessCentrality 的复杂度 $O(|V|^2 + |V||E|)$相比我直接写这个的复杂度还是有比较大的优化。

3.6.4 indegreeCentrality 和 outDegreeCentrality 的计算

对于有向图中这两个入度中心性和出度中心性的计算，本质上是和度中心性的计算一致的。所以在计算的时候仅仅需要稍微更改一下判断方法即可（直接统计以该顶点为目标点或者初始点的边的数量即可）。

3.6.5 distance 的计算

对于两个顶点之间的最短路的计算，只需要利用 Dijkstra 算法来实现。但是在为了更好描述在 API 中如果两个顶点直接没有直接的可以到达的路径，即两点不连通。distance 函数会返回-1 表示两个顶点之间没有最短路径存在。

3.6.6 Eccentricity 的计算

对于计算图中某个顶点的离心率，同样对于不连通图而言，如果直接计算可能有两个顶点之间的距离是正无穷，所以当存在由两个顶点之间没有可达路径的时候返回 INF 表示该顶点的离心率。

3.6.7 radius 和 diameter 的计算

对于图中的半径和直径的计算，是建立在顶点离心率的基础之上的。对于半径的

计算，如果任意一点的离心率都是不可达的话，返回-1 表征图的半径大小；对于顶点的计算，如果存在由顶点的离心率是超过 INF 的情况，仍然选择返回-1 表征图的直径。

3.7 图的可视化：第三方 API 的复用（选做）

在最后的实现图的可视化的时候对于其中的一些参数进行了设置。

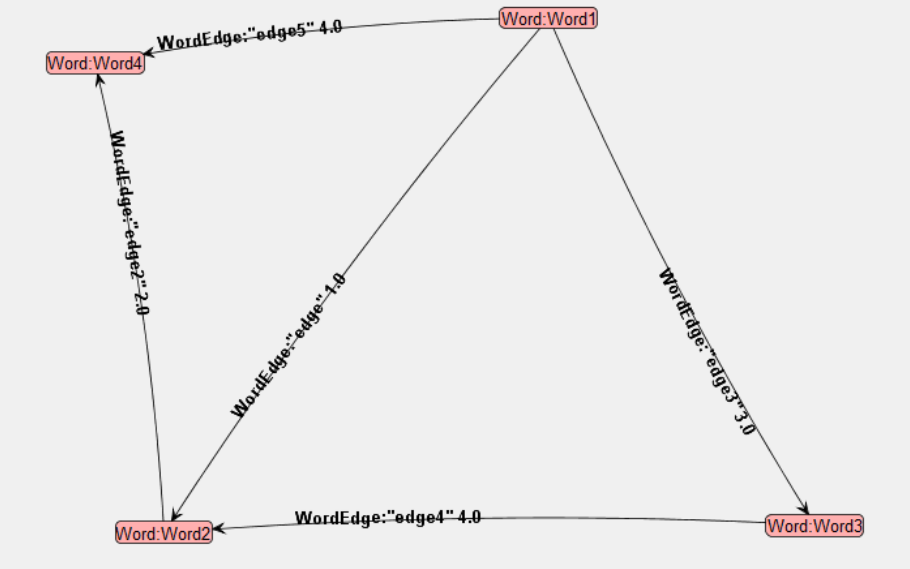
1、 顶点和边标签的设计，由于在默认的实现中对于所有的标签都是直接调用 Vertex 和 Edge 继承 Object 的 toString()方法来实现的，由于对于其中的每一个顶点和边在 toString()中展示的信息远比标签中需要的要多，所以在其中增加了两个辅助类 VertexLabelHelper 和 EdgeLabelHelper 用于适配需要的标签。

2、 对于所有的顶点，采用不同的颜色来实现其中的不同的类的顶点来标示其类，比如对于 GraphPoet 中使用的 Word 类采用了粉色，对于 GraphMovie 中使用的 Actor 类使用了蓝色等等。

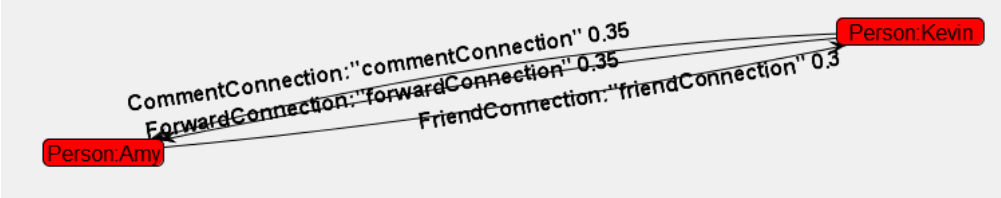
3、 对于超边的可视化，在 Jung 中没有提供使用的统一的 layout 所以退而求其次选择在 APP 中使用文字的方式来实现展示其中的信息。

最终效果展示：

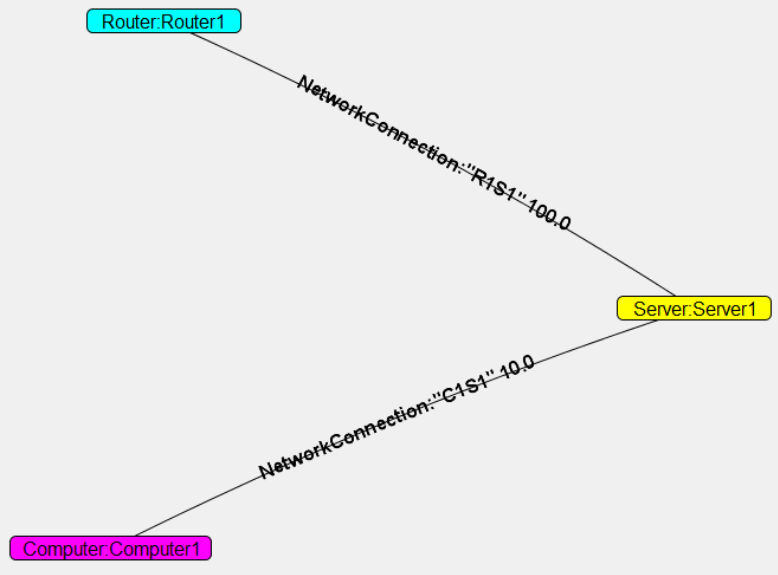
1、GraphPoet 的图可视化展示：



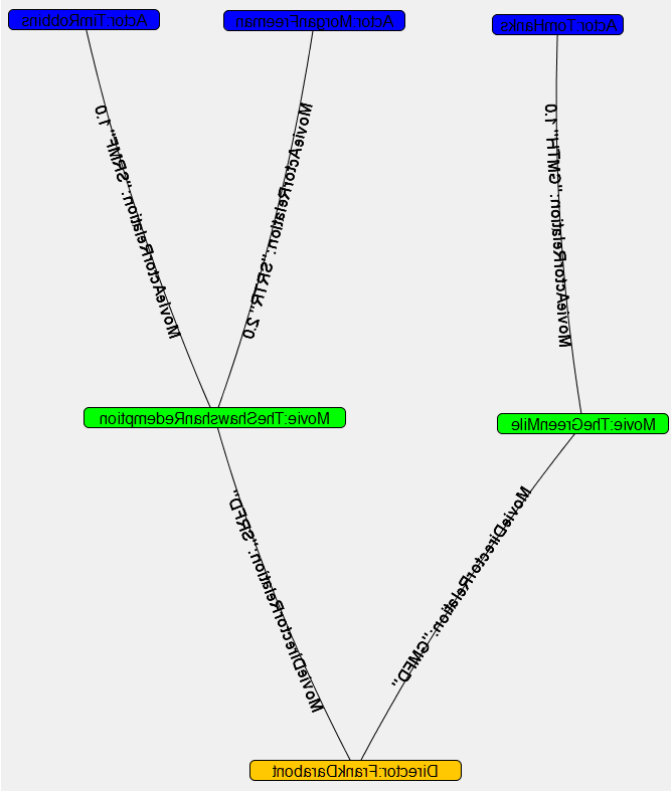
2、SocialNetwork 图的可视化展示：



3、NetworkTopology 图的可视化展示：



4、GraphMovie 图的可视化展示（此处没有展示超边）



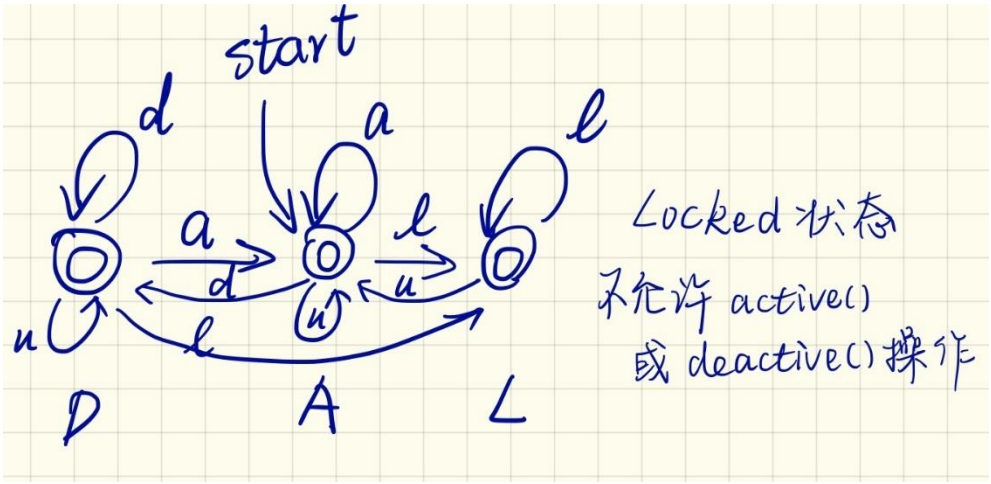
3.8 设计模式应用

3.8.1 使用 State/Memento 模式进行 Vertex 的状态管理（选做）

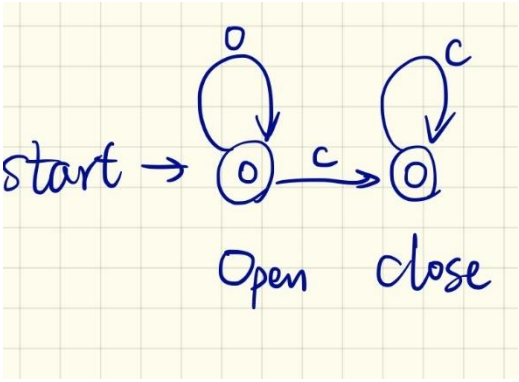
对于 Vertex 的状态管理其实只有两个顶点类需要实现，分别是在 NetworkTopology 中使用的各类顶点和在 SocialNetwork 中使用的各类顶点。对于所有的 State 设计一个接口作为所有 State 必须继承的模版，并在 Memento 中作为其中的一个属性。而在所有的需要使用记录状态的顶点类中增加一个属性 Caretaker 作为记录者维护这个顶点属性的变化过程。

并附上对于状态转换中形成的 DFA（Deterministic Finite Automaton）

对于 NetworkEquipment 顶点的状态转换如下图：



对于 Person 顶点的状态转换如下图：



3.8.2 使用 factory method 模式构造 Vertex 对象

使用 factory method 模式来构造 Vertex，在实现中设计了一个 VertexFactory 作为最终的工厂类，在这里由于使用了 type 参数所以只构造了一个 factory 类作为（虽然这样的可维护性不好，不满足 OCP 原则，但是实验手册相比于进度的修改没看到，只好这样了）。

3.8.3 使用 factory method 模式构造 Edge 对象

使用 factory method 模式构造 Edge，在实现中使用了和 Vertex 类似的方法，使用 type 来实现不同的 edge 类对象的构造。另外对于 HyperEdge 有着和简单边不同的参数列表（超边中没有权重）所以，在这里使用了另一个工厂类，HyperEdgeFactory 来构造 HyperEdge 的对象。

3.8.4 使用 abstract factory 或 builder 模式构造 Graph 对象

使用 abstract factory 来构造 Graph 对象，就是在 GraphFactory 中分别使用已经设计好的 VertexFactory 和 EdgeFactory 来实现对与 Graph 对象的组装。

3.8.5 使用 Strategy 模式调用 centrality 度量算法

遵循实验手册中的要求，对于不同的计算 Centrality 的方法，均放在将所有 API 放置在一个 helper 类 GraphMetrics 当中。计算的时候只需要直接调用静态的相应计算 Centrality 方法即可。

3.8.6 使用 Composite 模式设计超边对象（选做）

3.8.7 使用 decorator 模式构造不同特征的 Edge 对象（选做）

3.8.8 使用其他设计模式（选做）

3.9 图操作指令的输入和处理（选做）

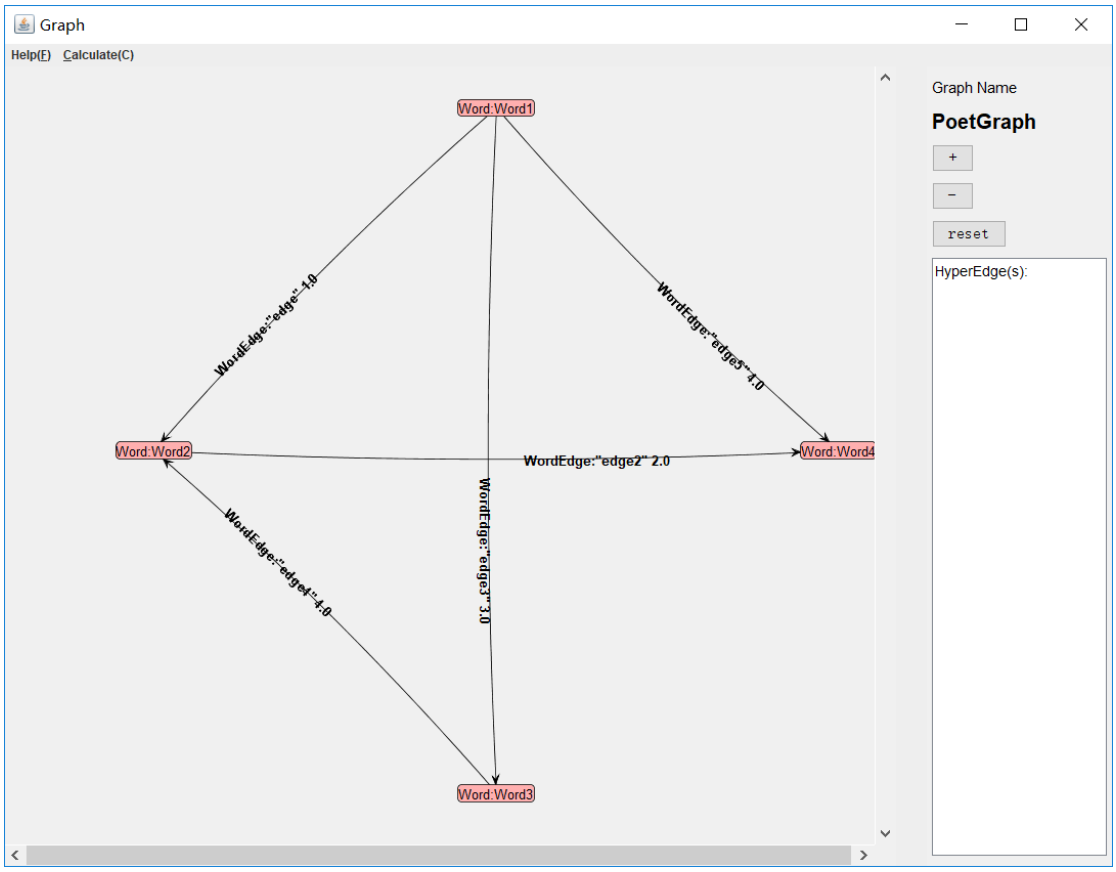
使用 façade 设计模式，完善 ParseCommandHelper 类

3.10 应用设计与开发

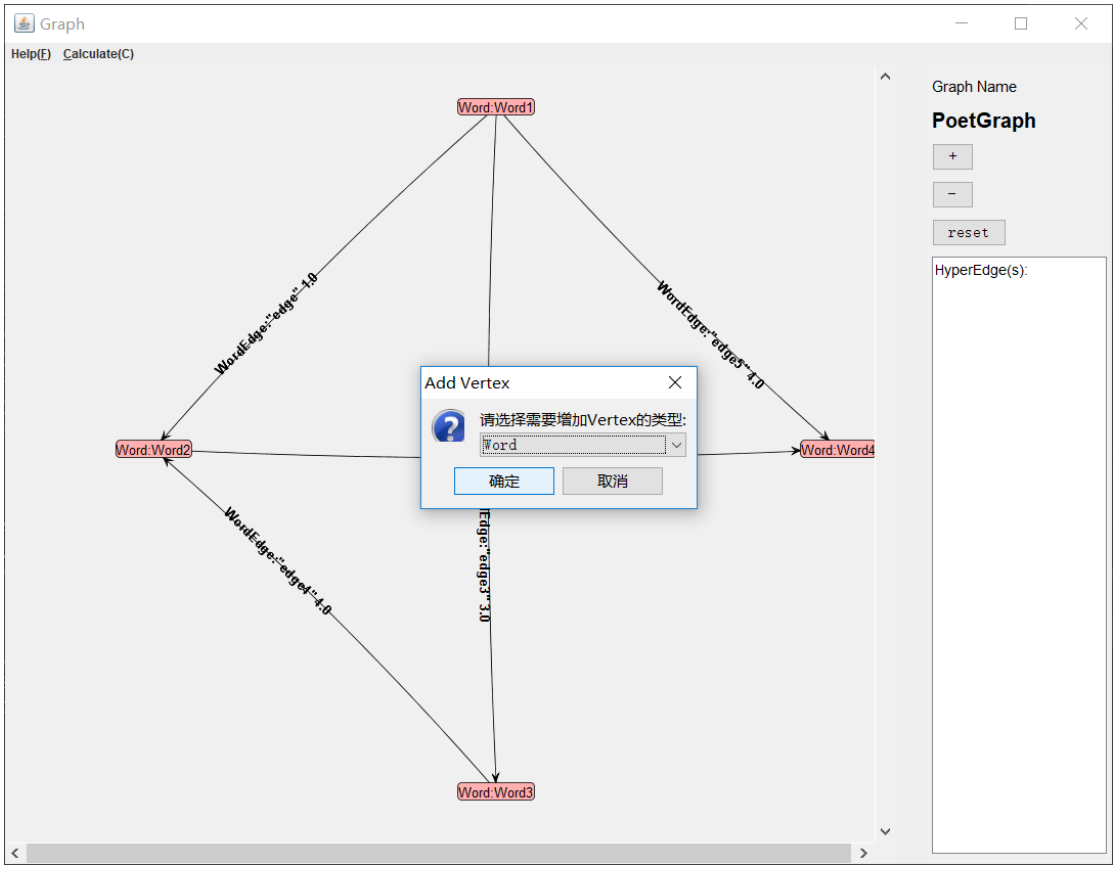
利用上述设计和实现的 ADT，实现手册里要求的各项功能。

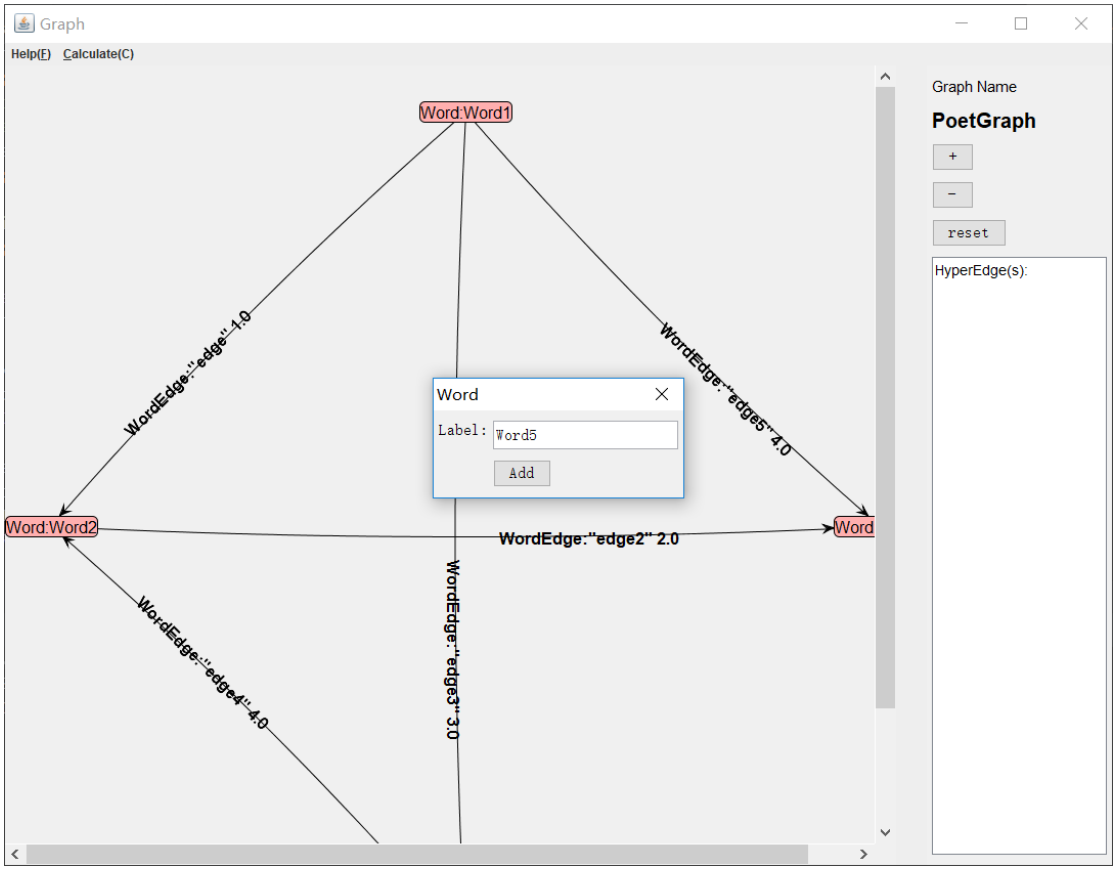
3.10.1 单词网络 GraphPoet

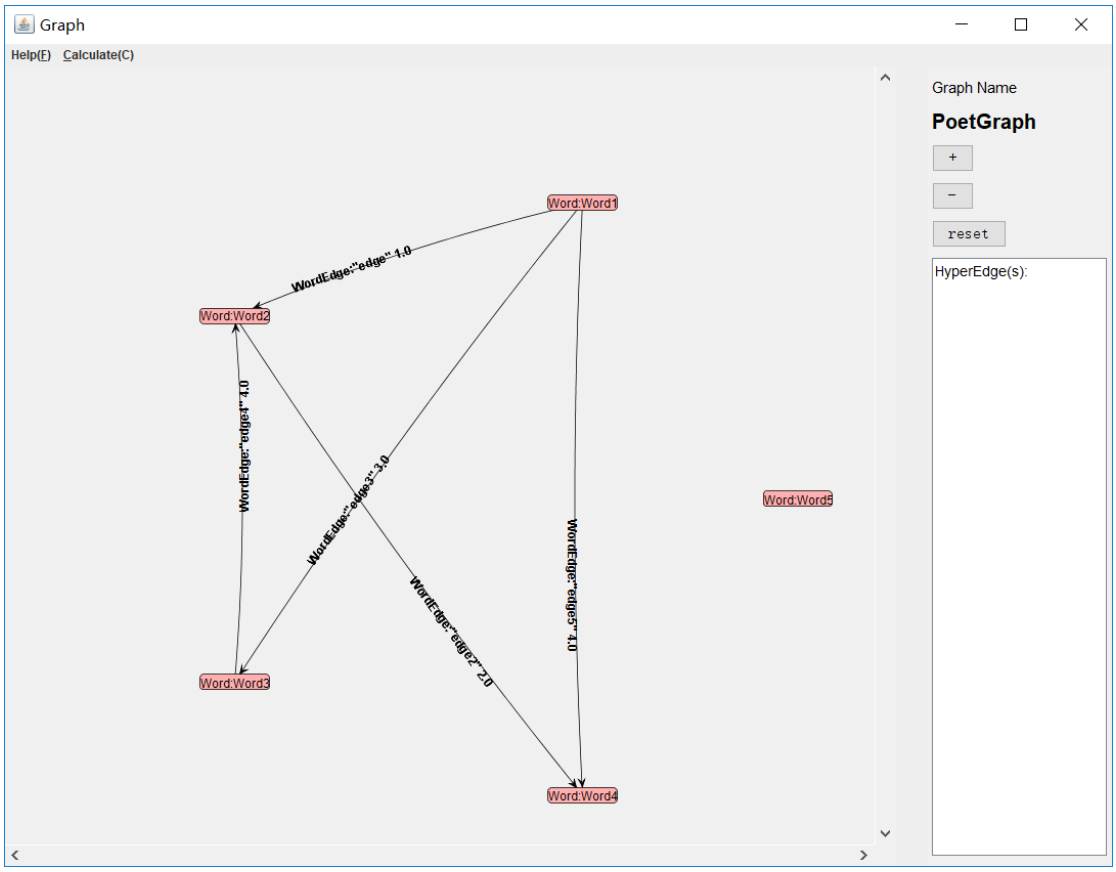
初始界面：



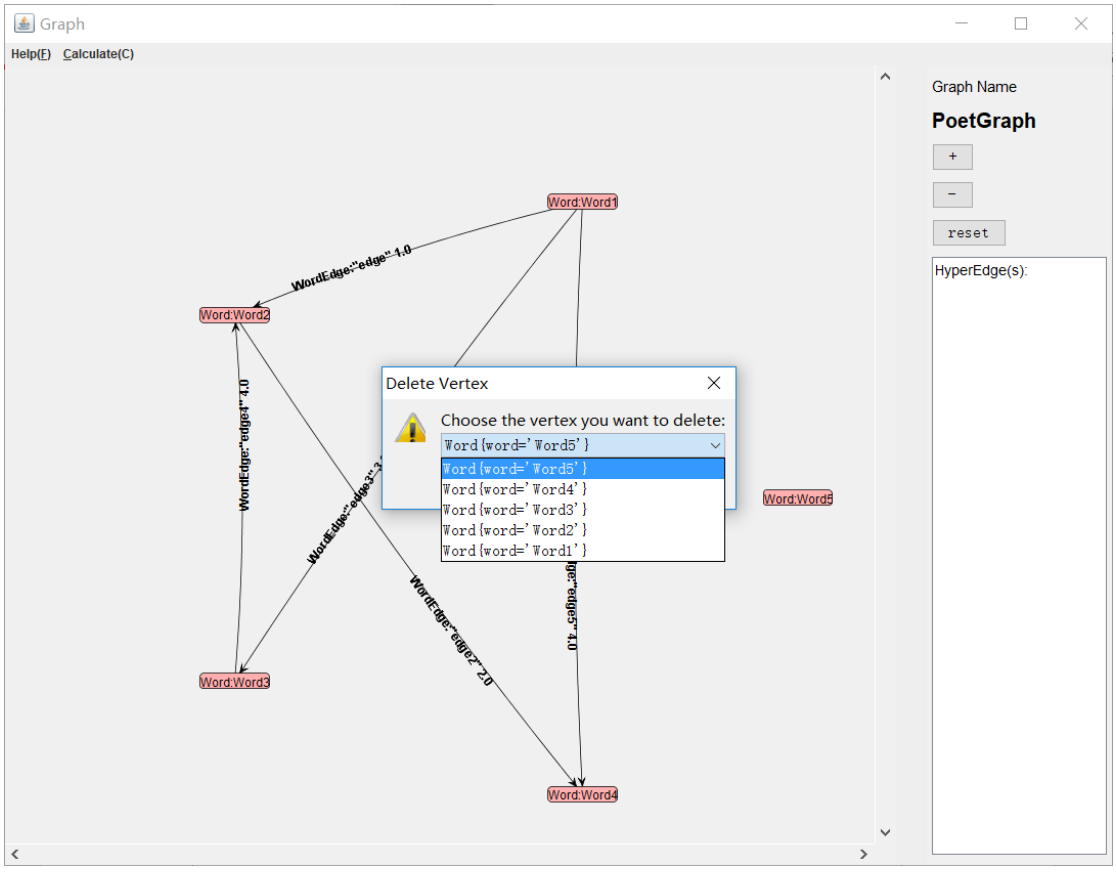
增加节点（点击 Help，选择 Add Vertex）：

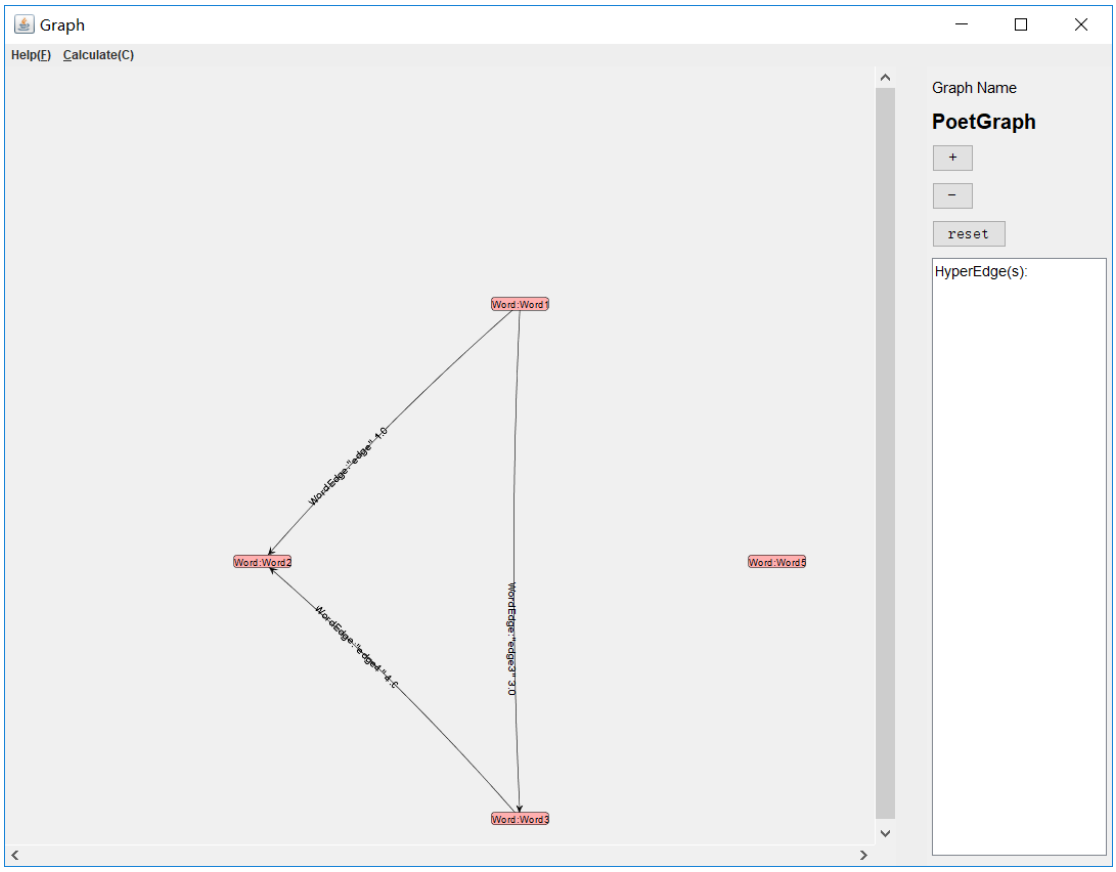




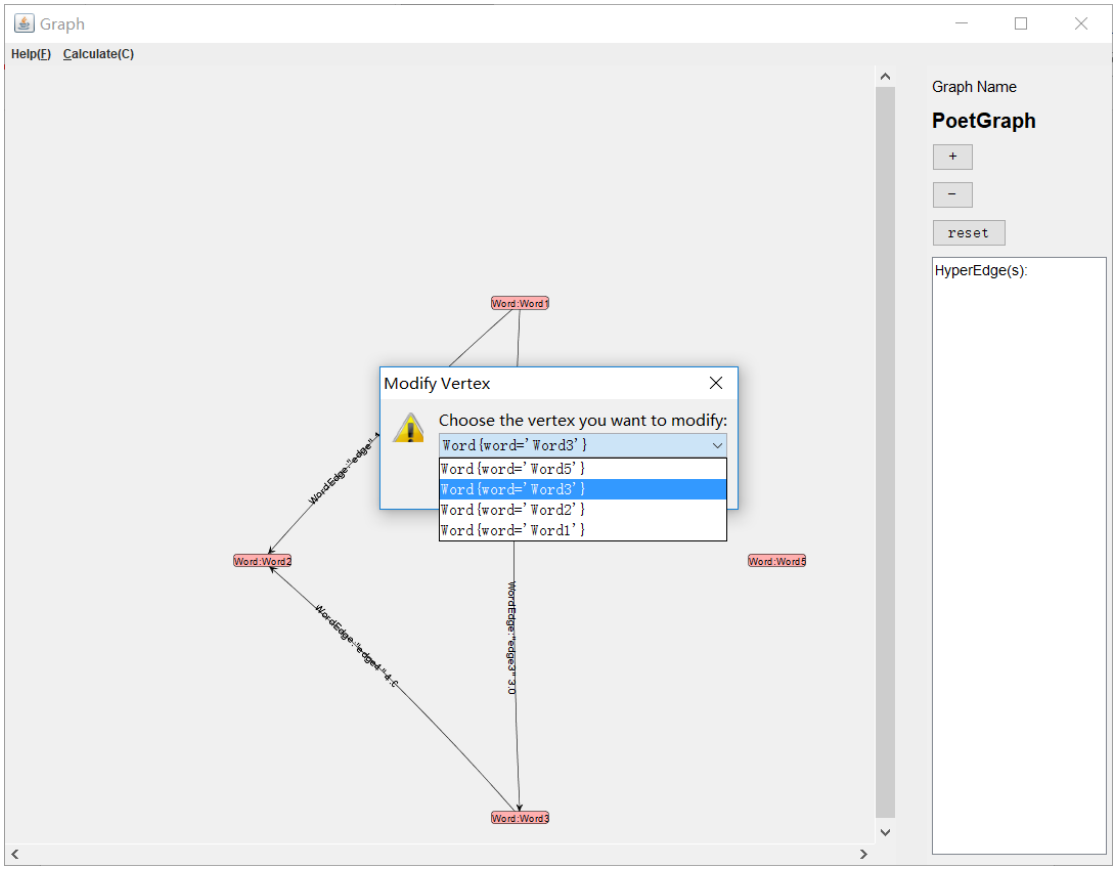


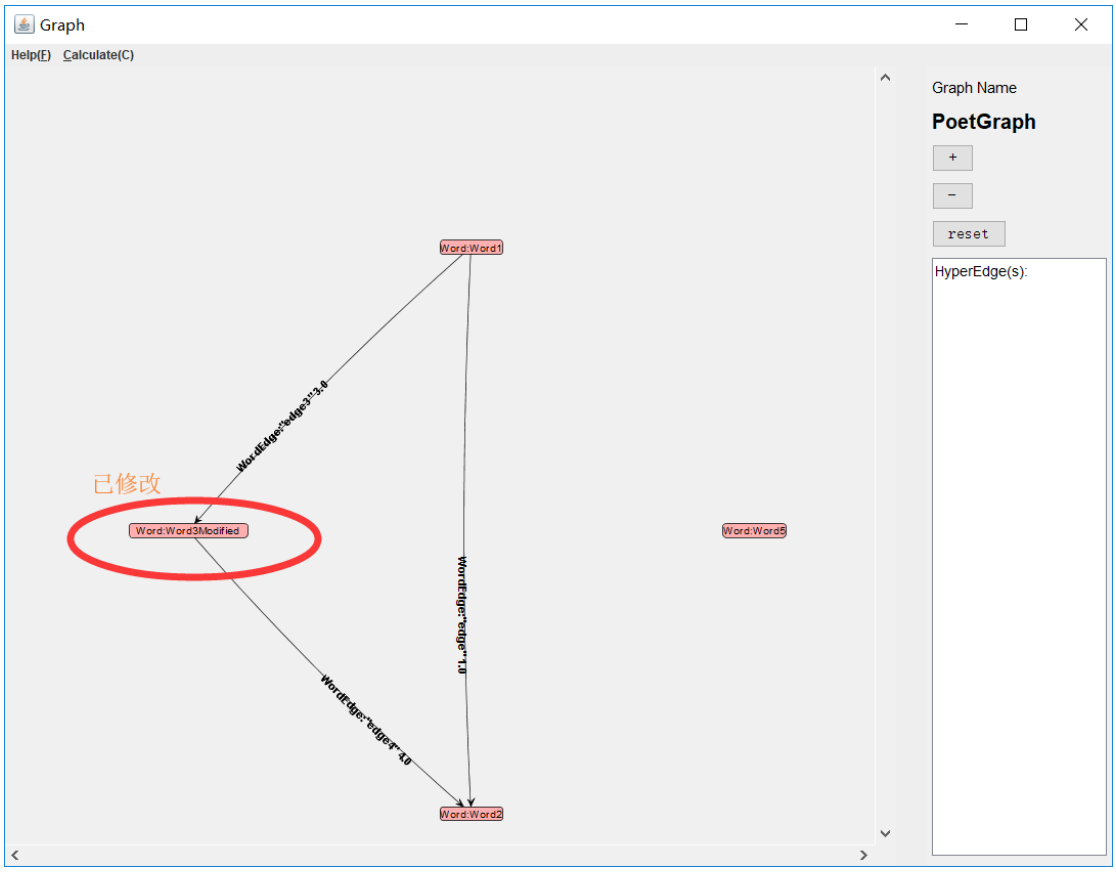
删除节点（点击 Help，选择 Delete Vertex）：



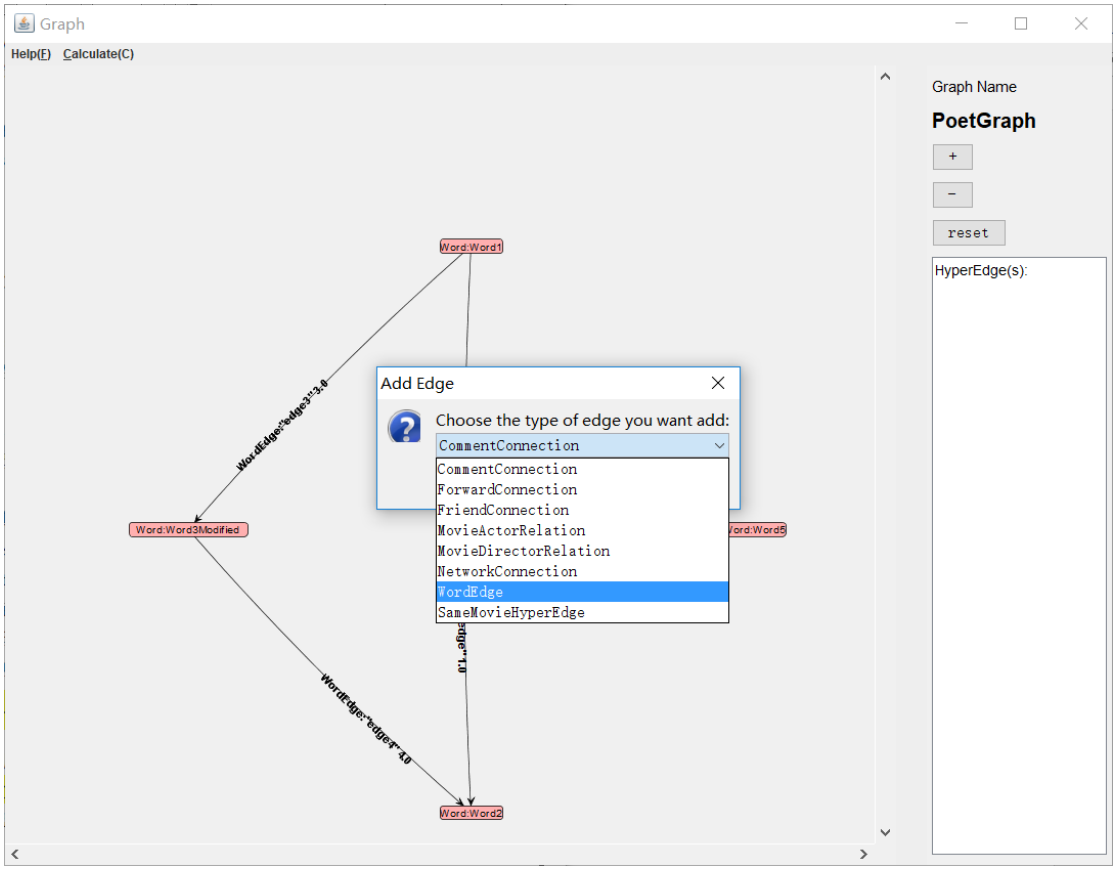


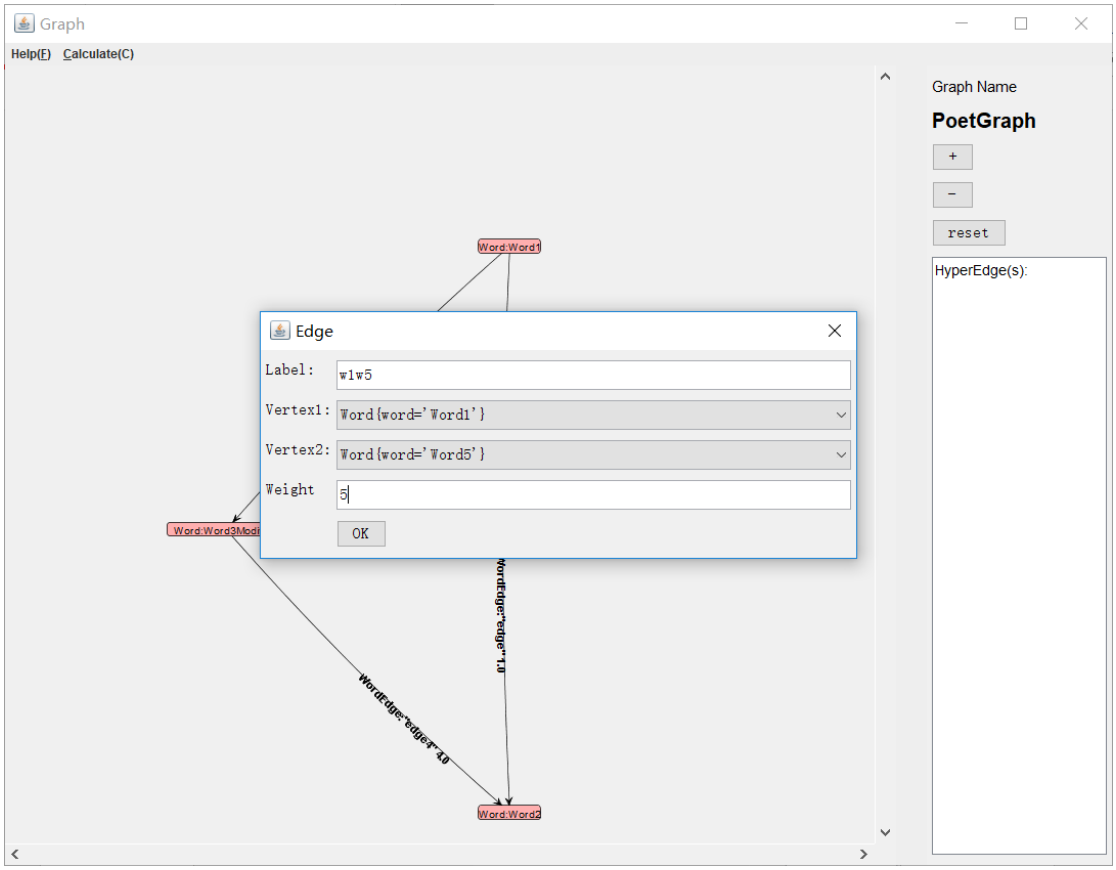
修改节点（点击 Help 按钮，选择 Modify Vertex）：

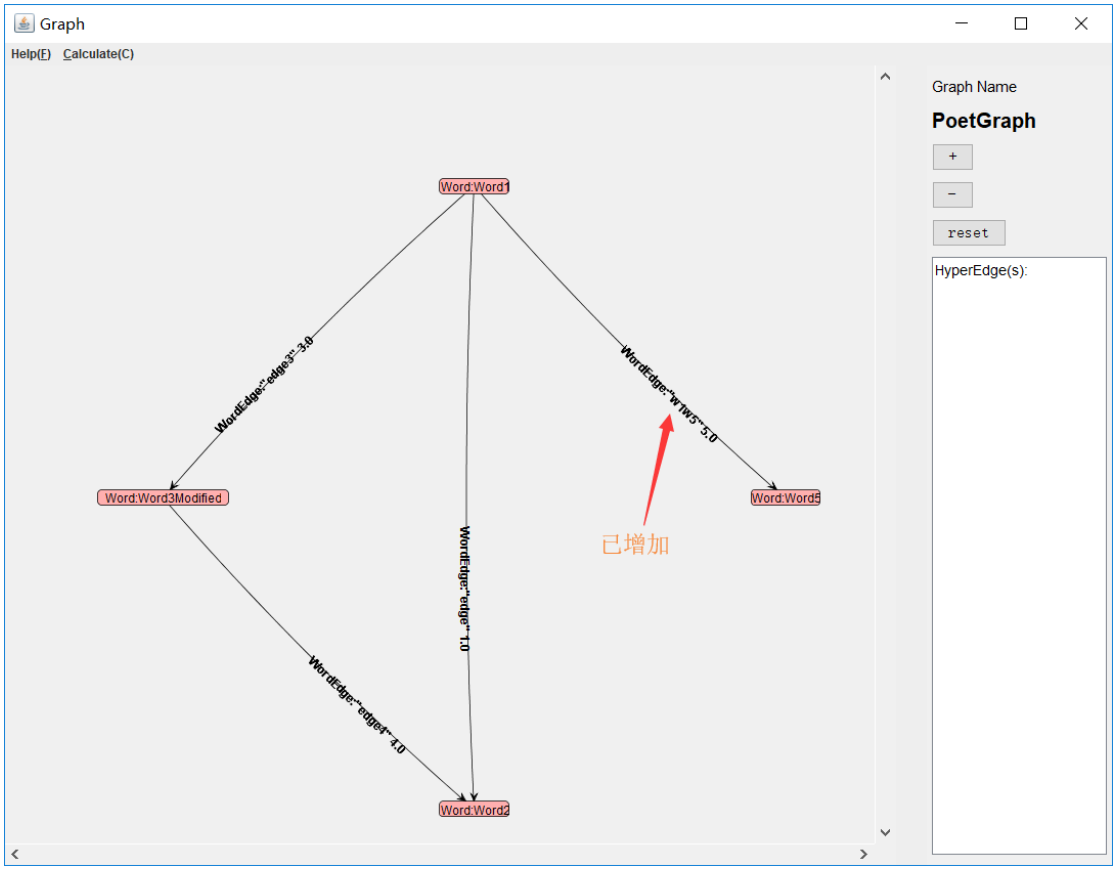




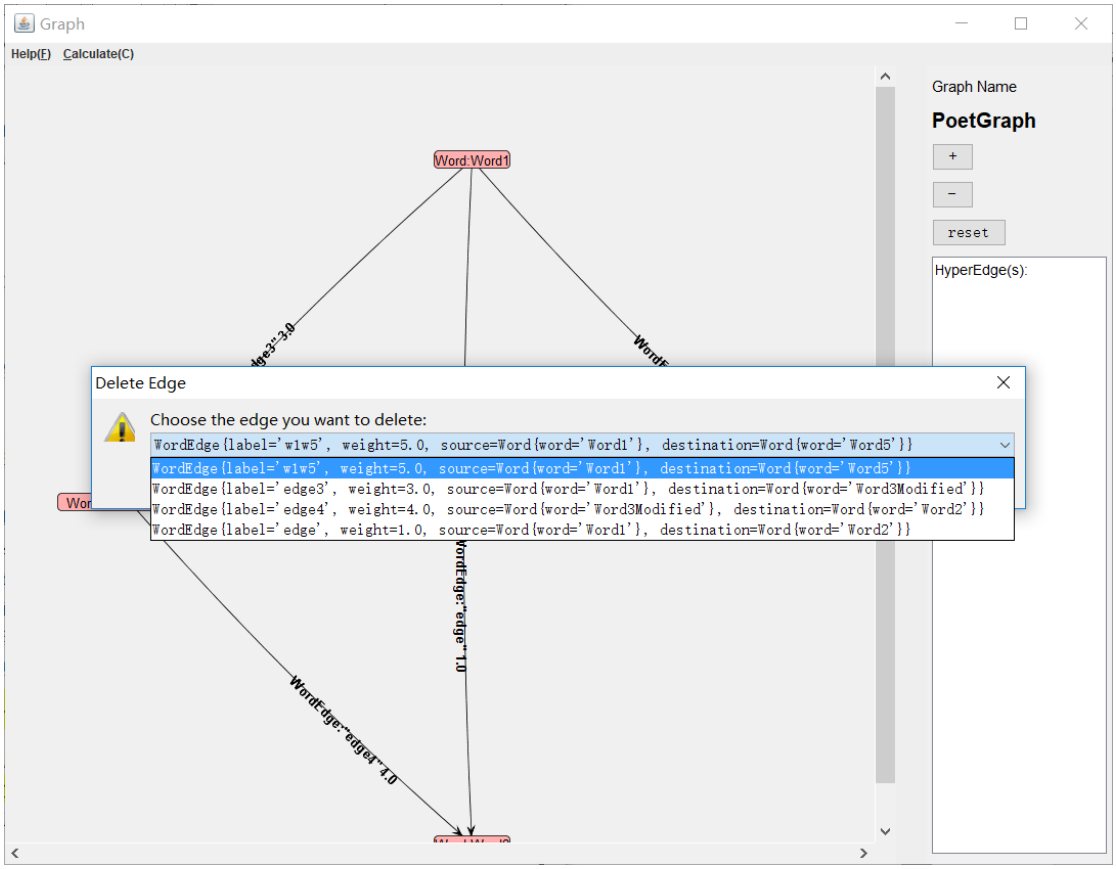
增加边（点击 Help，选择 Add Edge）：

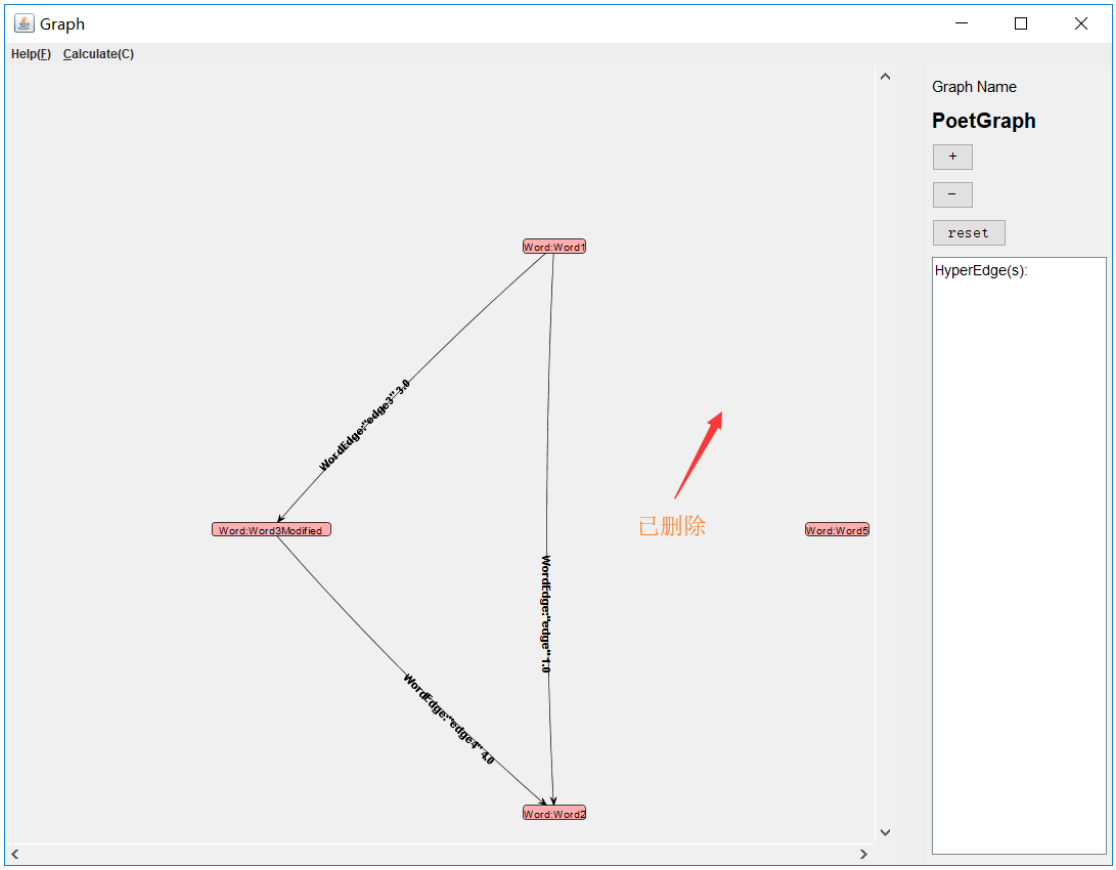



Graph
Help(F) Calculate(C)
Graph Name
PoetGraph
+
-
reset
HyperEdge(s):
Word:Word1
Edge
Label:
w1w5
Vertex1:
Word{word='Word1'}
Vertex2:
Word{word='Word5'}
Weight
5
OK
Word:Word3Modi
Word:Word2
WordEdge:"edge1":1.0
WordEdge:"edge4":4.0

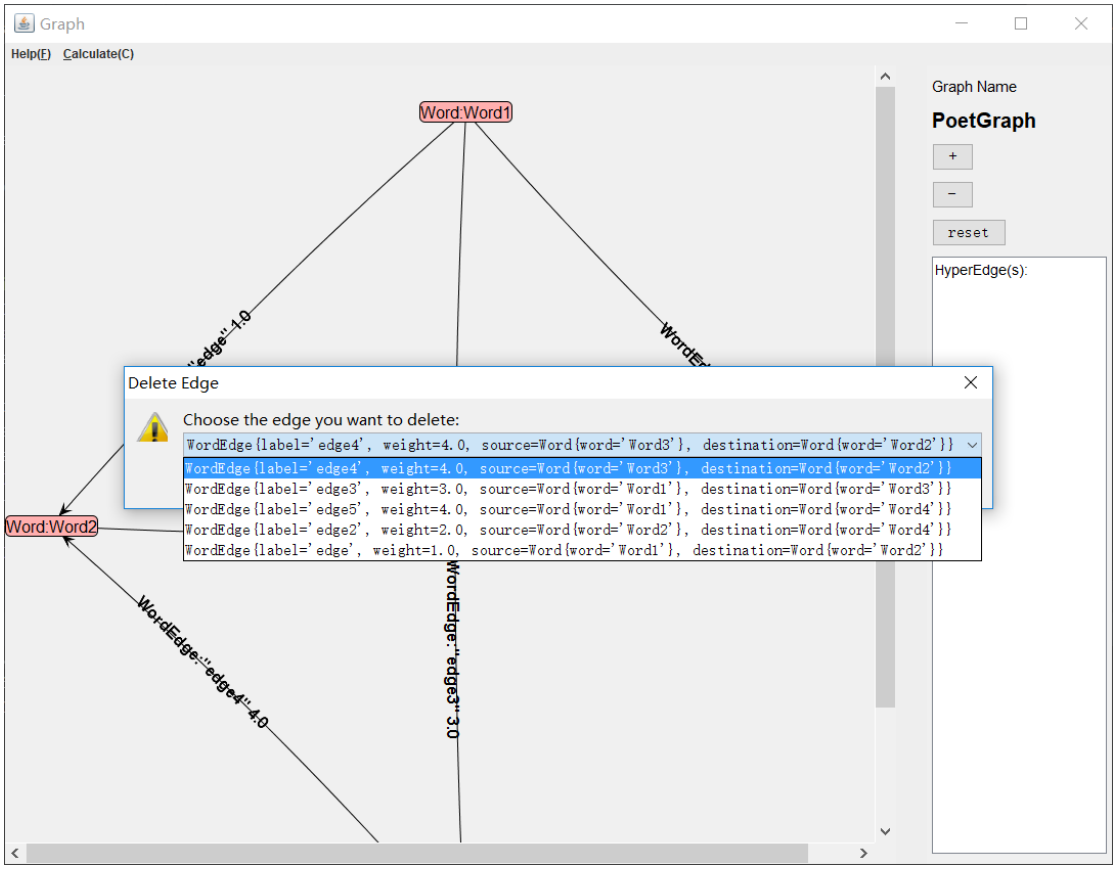

Graph
Help(F) Calculate(C)
Graph Name
PoetGraph
+
-
reset
HyperEdge(s):
Word:Word1
Word:Word3Modified
Word:Word5
Word:Word2
WordEdge:"edge3":3.0
WordEdge:"w1w5":5.0
WordEdge:"edge1":1.0
WordEdge:"edge4":4.0
已增加

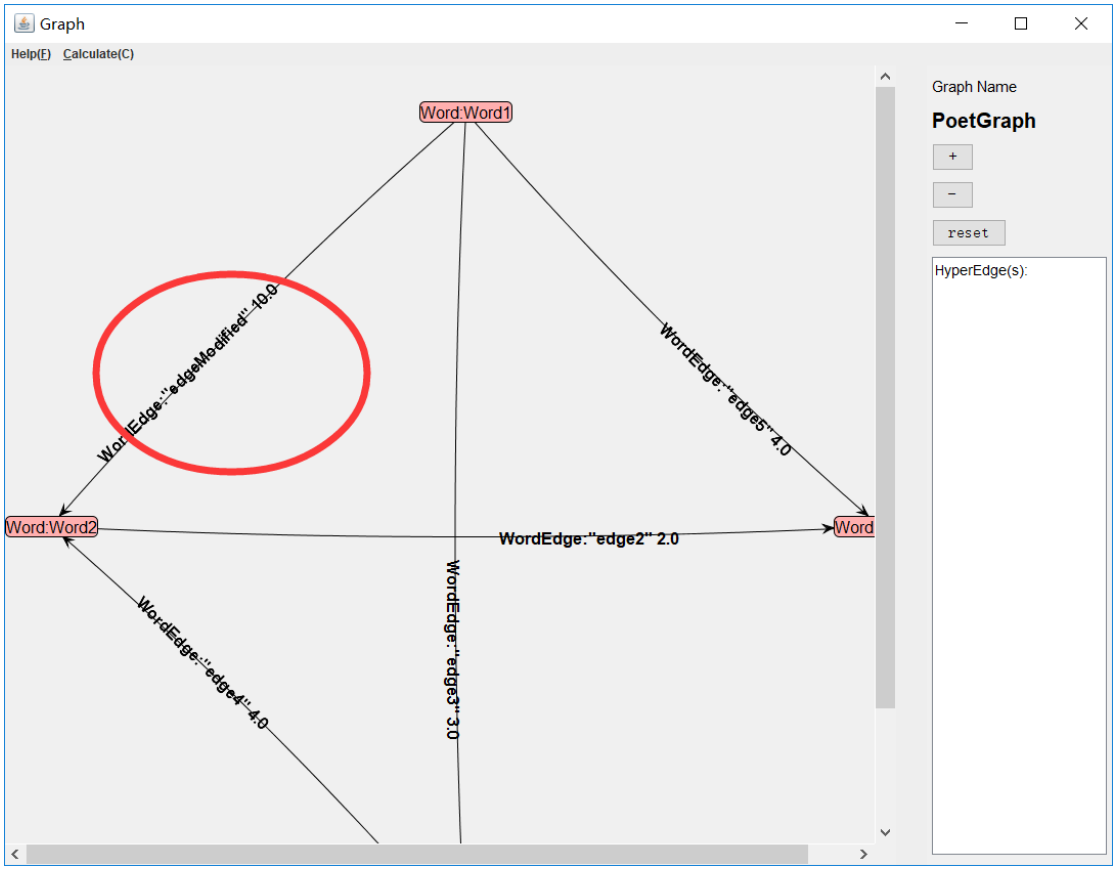
删除边（点击 Help 选择 Delete Edge）：



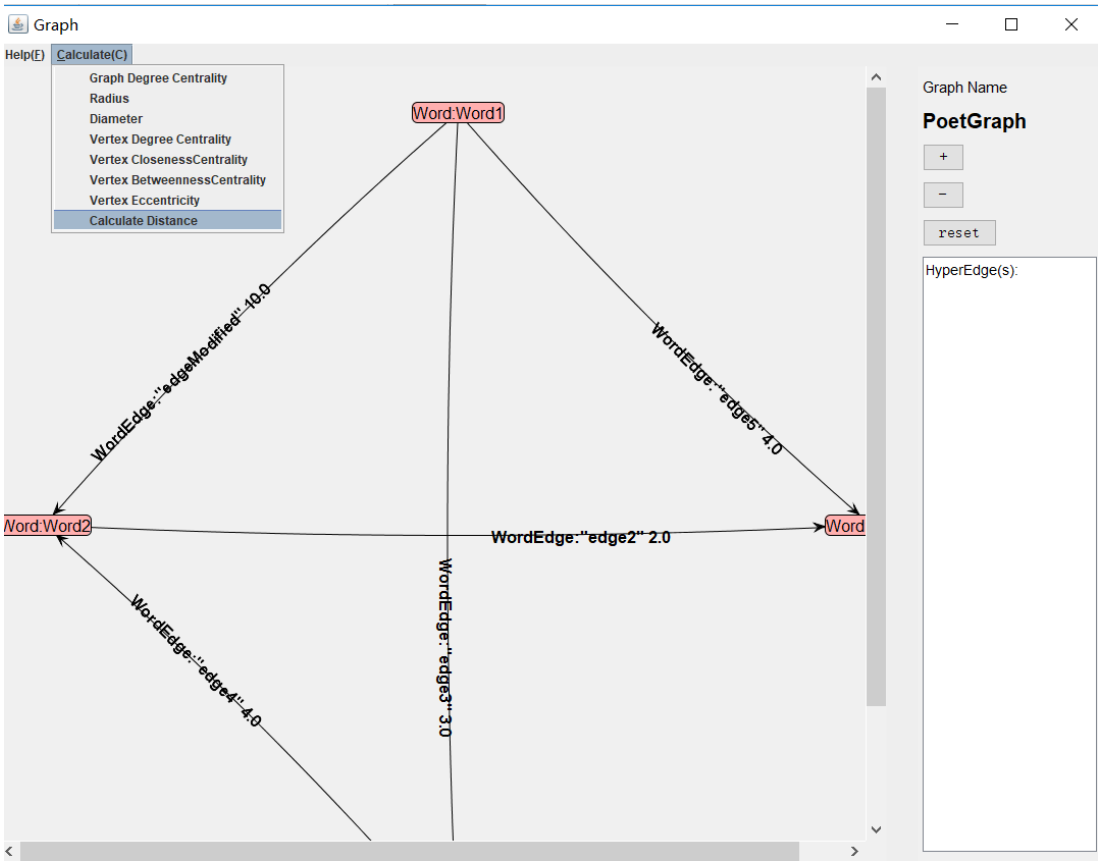


修改边（点击 Help 选择 Modify Edge）：

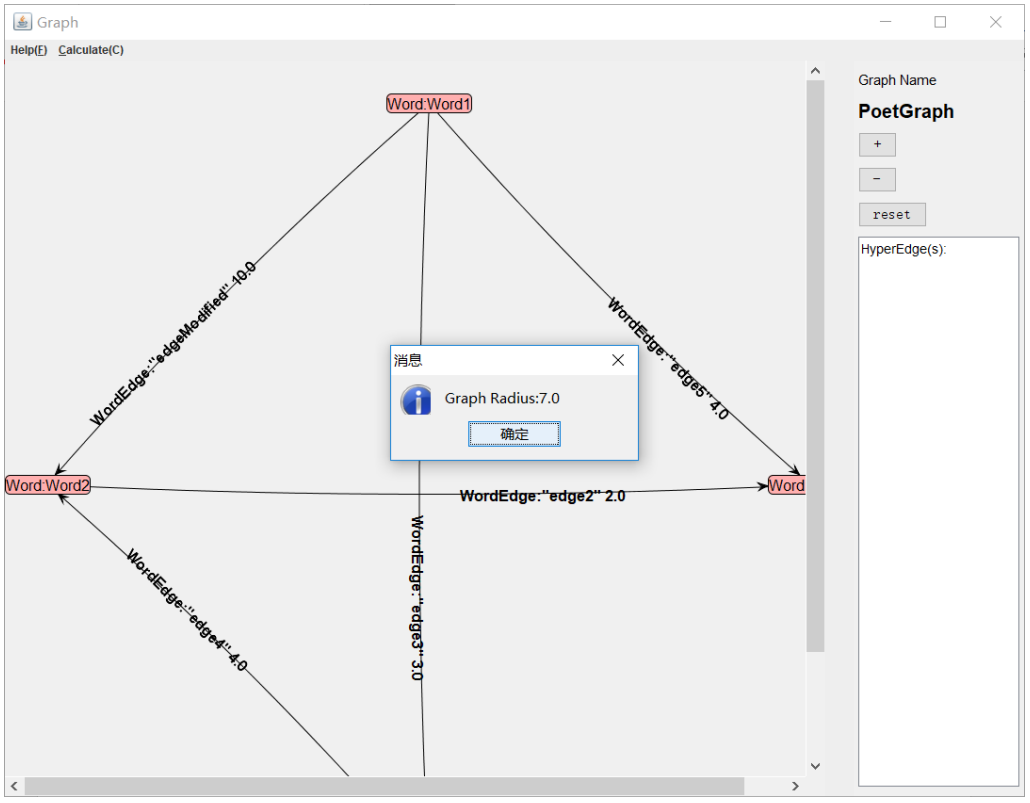




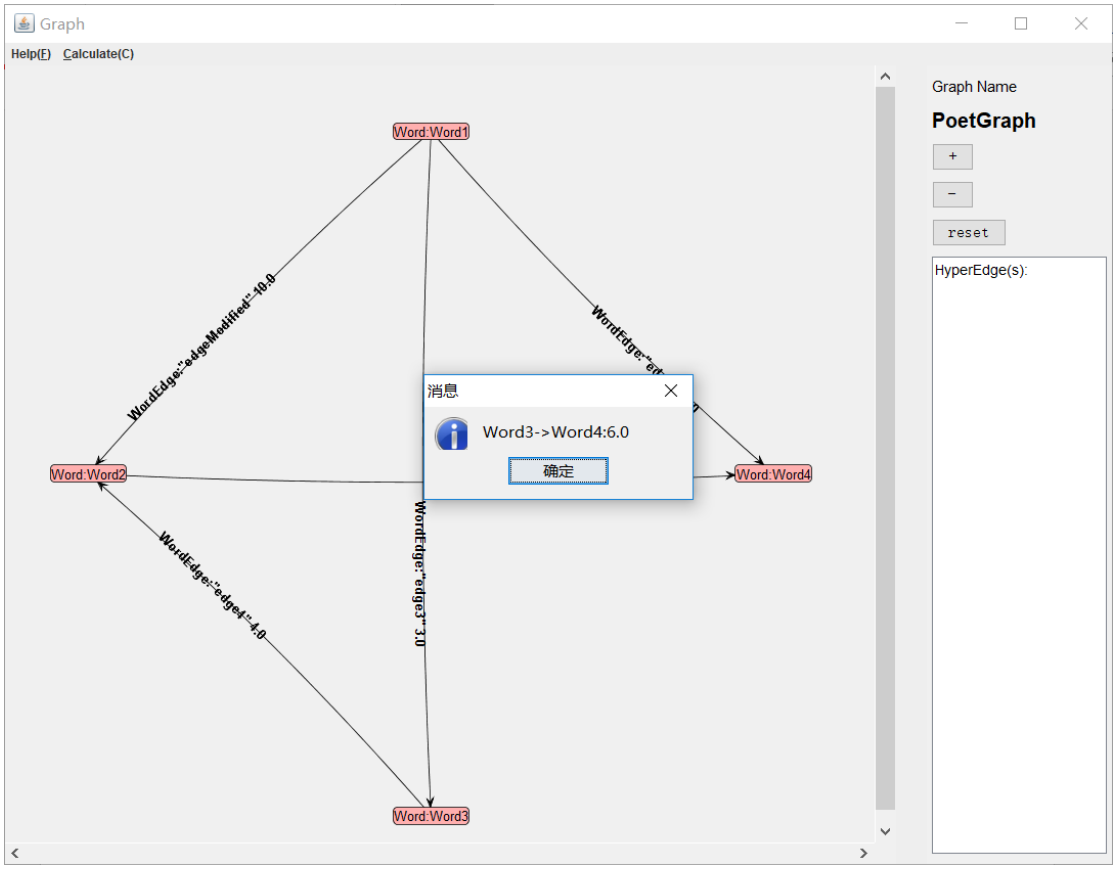
计算各类 Centrality（点击 Calculate 选择相应的计算项即可）



计算 Radius：

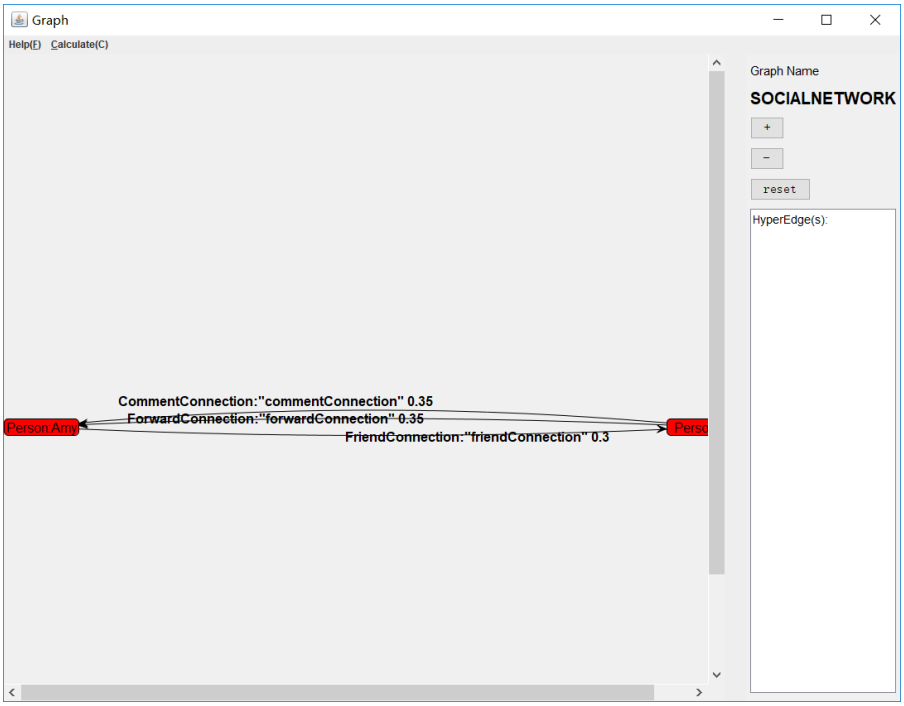


计算 distance：

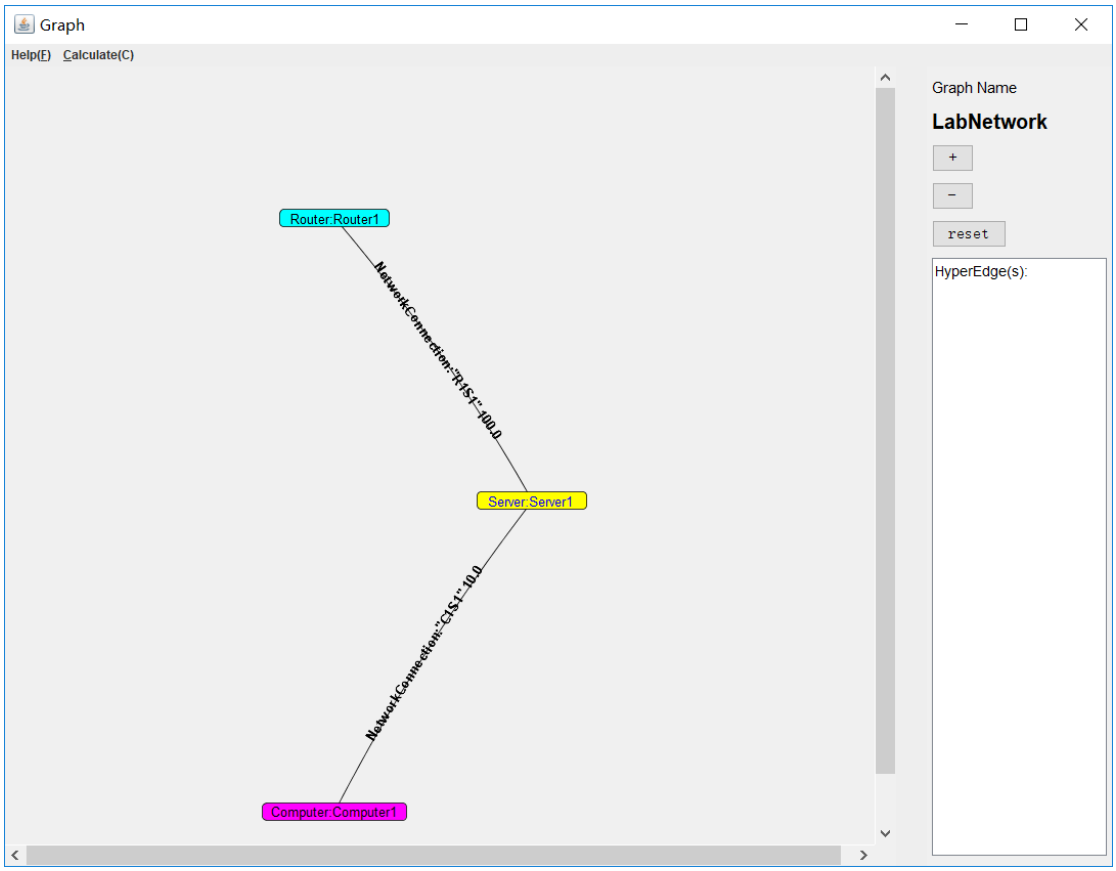


3.10.2 微博社交网络 SocialNetwork

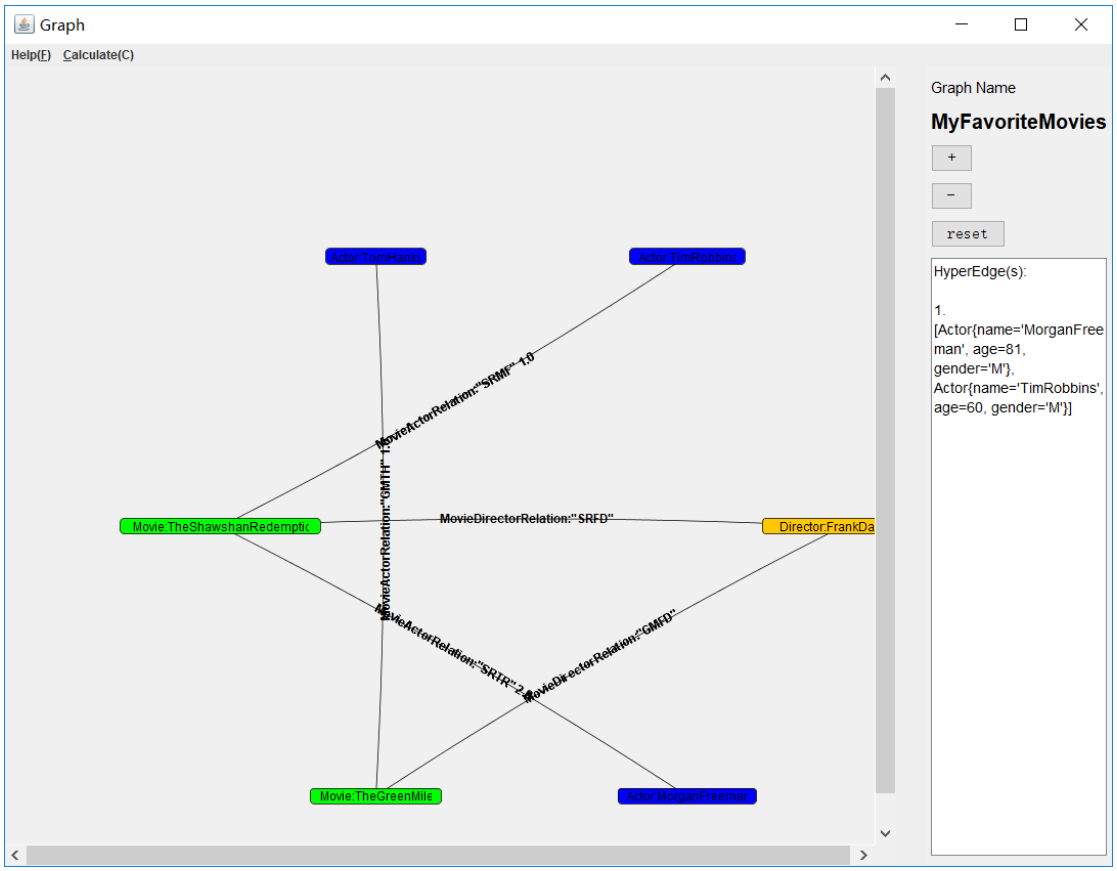
由于操作类似，所以不再演示操作，只展示界面的不同。



3.10.3 网络拓扑图 NetworkTopology



3.10.4 电影网络 MovieGraph

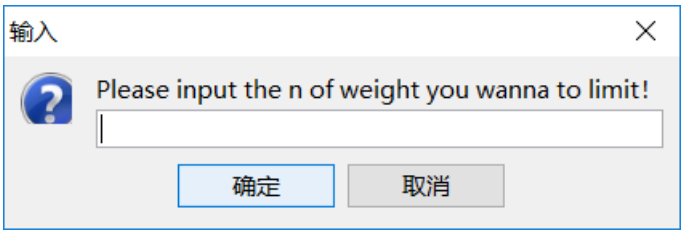


3.11 应对四个应用面临的新变化（任选两个）

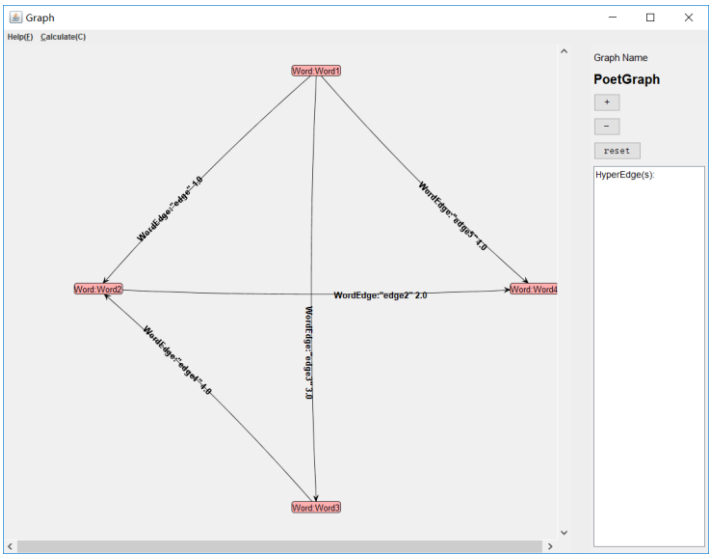
选择增加两种变化，在 GraphPoet 中增加一种对于边权的限制功能，在 NetworkTopology 中增加一种新的顶点类型 WirelessRouter。最终所有的新增变化得到的增删的代码量如下，具体增删的细节请见 Lab3-1160300314 仓库（https://github.com/ComputerScienceHIT/Lab3-1160300314/commit/c3856b0bc59b7c6115ba7ab56cdf58d499bcb402?diff=unified）：

3.11.1 单词网络 GraphPoet

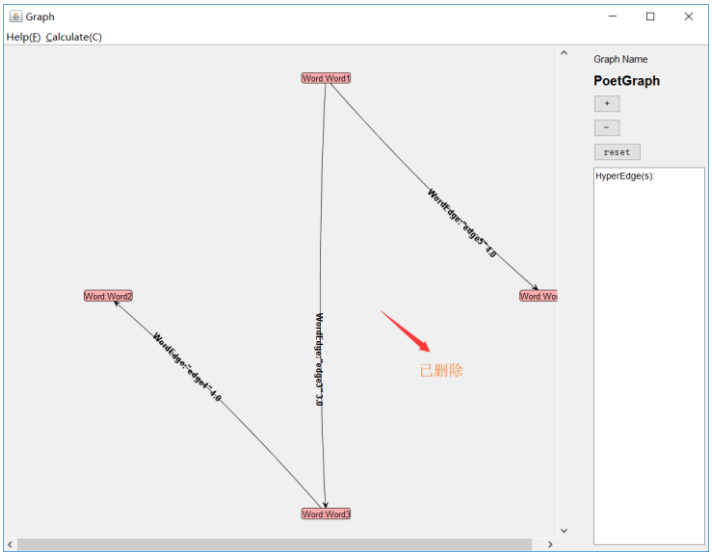
在增加限制之后，每一次打开 APP 都需要先进行设置 n，对其中的边权进行限制，并且如果输入不合法，会重新提示进行输入。



此处使用与上面相同的输入文件，并且输入限制 n = 3，得到的结果如下图：



（原图）

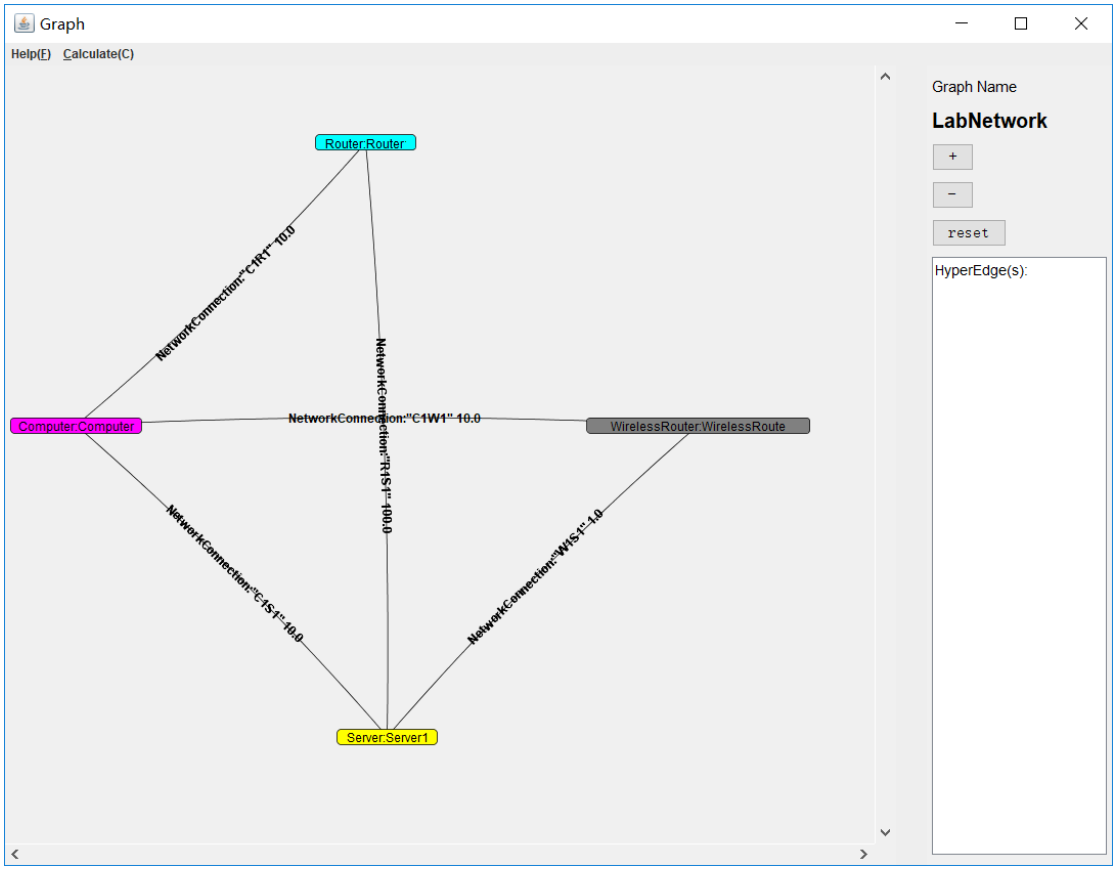


（增加限制后的图）

3.11.2 微博社交网络 SocialNetwork

3.11.3 网络拓扑图 NetworkTopology

增加一种新的顶点类型 WirelessRouter，增加后在图中的显示变为如下：



3.11.4 电影网络 MovieGraph

4 实验进度记录

请尽可能详细的记录你的进度情况。

更详细的见 commit 记录

日期	时间段	计划任务	实际完成情况
4.2	13：45-15:30	形成初步框架	完成
4.2	16:00-19:00	完成 vertex 文件里对于顶点的要求	完成
4.2	19:00-	初步完成 vertex 文件夹里的测	完成

	22:00	试	
4.2	22:00-23:00	增加边集	延期
4.3	8:00-10:00	完成 edge 文件夹里的测试	延期
4.3	？-18:00	修复遗留问题	完成
4.3	19:00-23:00	完善已有 toString 的测试	完成
4.5	全天	编写 Graph	延期
4.6	全天	测试 Graph 编写 API	延期
4.7	全天	修复 bug	完成
4.8-4.15		GUI	完成
--至今		完善报告	完成

5 验过程中遇到的困难与解决途径

1、计算 betweenness centrality 的困难，由于平常经常写的 Dijkstra 不能跑出多个最短路，而且对于很多情况处理并不够好。所以在思考书写难度之后，改成采用 Jung API 中已经提供好的计算 API 直接进行计算，而且调用其的性能也远比直接去写得到的性能要好的多。

2、对于 Jung 中很多功能，在可视化的时候如何处理其中一些滚动条之类的处理并不够了解，而且在它的文档中也没有书写，所以去 GitHub 发现 Jung 官方将所有它写过的 example 都在 GitHub 上面开源了（而且发现所有的例子都使用的是 Guava 的 Graph API 来实现图），然后通过发现其中的细节再去实现自己的可视化。

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) 重新思考 Lab2 中的问题：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？本实验设计的 ADT 在四个图应用场景下使用，你是否体会到复用的好处？

(2) 重新思考 Lab2 中的问题：为 ADT 撰写复杂的 specification, invariants, RI, AF，时刻注意 ADT 是否有 rep exposure，这些工作的意义是什么？你是否愿意在以后的编程中坚持这么做？

(3) 之前你将别人提供的 API 用于自己的程序开发中，本次实验你尝试着开发给别人使用的 API，是否能够体会到其中的难处和乐趣？

(4) 在编程中使用设计模式，增加了很多类，但在复用和可维护性方面带来了收益。你如何看待设计模式？

(5) 你之前在使用其他软件时，应该体会过输入各种命令向系统发出指令。本次实验你开发了一系列命令行指令，使用语法和正则表达式去解析它们并映射到对后台程序的调用。你对语法驱动编程有何感受？

(6) 关于本实验的工作量、难度、deadline。

(7) 到目前为止你对《软件构造》课程的评价。

（1）对于面向 ADT 编程与面向应用场景编程的最大区别就在于，我编程的目的是什么。或者说我考虑问题的方式不同，面向 ADT 编程我需要了解所有应用场景的共性，从中做便利性与性能的折中，寻找最适合的方法适应尽可能多的场景；而直接面向应用场景编程，我可以找到相比前者更加适合的数据结构和算法，可以设计更加便利的操作也不需要和已有的 ADT 相匹配。

但就我设计的 ADT 而言，还是有很多不够方便的地方，对于复用而言，可以减少代码量进而减少 bug 的出现，这是直接面向应用编程所不能相比的。但仅仅就目前所写出的 ADT，在粗略学习过 Guava 中提供的图的集合类之后感觉需要改善的地方还有很多。

（2）感觉所有注释的作用都可以叫做“Remind”，就是提醒自己也提醒别人。因为我们在看到其他人写的代码的时候，如果写的很晦涩并且没有注释我们常常会抱怨“连个注释都不写”，但到了我们自己写代码的时候又不懒到不想写注释。其实无论是 AF、RI 或者是 Safety for Rep Exposure 都是对自己的一种提醒，将自己的设计思路记录下来，这样不仅仅便于以后自己在看自己写的代码时候有看不懂的尴尬，也可以便于不同程序员之间的交流。

这一点我在这次实验中感触还是比较深的，这一次的实验我开始写的时候没有及时记录 RI 和 AF 等注释性的内容，导致后来我再次需要用到其中的某些类和方法的时候需要再看一遍代码，明确一些细节，虽然看一次花费的时间要比直接记录 AF 等内容要短不少，但是我反复看浪费的时间也是相当的。基于自己的体验感觉自己以后应该有动力及时书写注释。

（3）使用 API 的感受是好爽，这都不用再写了；设计 API 的感受是麻烦，啥都得记下来说清楚。设计 API 实际是将自己的代码给别人使用的一种方式，当然是提供的是黑盒；需要将每一个方法的 pre-condition 和 post-condition 都写下来明确好，还需要保持正确性和健壮性。

（4）设计模式的存在价值感觉在实验中的体现并不明显，因为整体而言实验

还应该算是一个小型的项目，所以我们在考虑使用设计模式的时候需要增加几个类，就会感觉把简单的事情变得很复杂，而且带来的收益也不大。但是我感觉既然设计模式被提出来，而且被发扬到今天一定有其中的意义，就是它带来的可复用性和可维护性的价值应该远超增加几个类的代价，才会用人们愿意去使用设计模式。

（5）对于正则匹配的相关问题，其实在学到了形式语言与自动机理论的时候有了一些别样的理解。对于实验要求中的正则匹配来说，其实由于一些要求还是没有明确所以在使用的时候，会带来一些问题。比如说在 edge 中的命令的解析，对于其中的最后一个参数，有向边或者无向边来说，使用“Yes”或者“No”，但是由于前面的参数已经明确了这个边的类型，所以其是否有为有向边就已经确定了，增加这个参数的作用就表现不出来，甚至在一些的特殊情况中还需要特殊的处理。其余的命令行的设计来说，解析起来还是比较方便的，对于正则语法的学习还是很有好处的。

（6）对于 Lab3 来说，感觉整体的实验难度不大，但是设计的量特别大，尤其是在与 Lab4 的结合做的并不是很好，其中很多部分在 Lab3 中并没有增加 throws Exception 的设计，导致在以后的设计中的可维护性其实并不高。并且由于中间的一些设计要求的反复（第一次开课的阵痛），写完之后再去修改的代价还是很高，总的来说 deadline 的设计还是合理的，只是这个实验写起来感觉太挣扎了。

（7）《软件构造》课程到现在已经结束了一半多了，对于其中的一些概念来说可能还是感觉很抽象，虽然想将一些学到的设计模式和方法用到实验的书写中，但是由于实验本身并不算一个太大的项目，在其中应用设计模式的代价远比成本来的要大不少，而且感觉最近的可维护性没有前面 OOP 学习起来更有感觉。