面向对象JML系列第三次代码作业指导书

摘要

本次作业，需要完成的任务为实现容器类`Path`，地铁系统类`RailwaySystem`，学习目标为**JML规格进阶级的理解和代码实现、设计模式和单元测试的进阶级实战。**

前排提醒：建议动手开始写之前，好好琢磨清楚，并最好先看一下最后的提示部分~~，巨佬除外~~。

问题

基本目标

本次作业最终需要实现一个简单地铁系统：

- 可以像`PathContainer`一样，通过各类输入指令来进行基于路径的增删查改管理。
- 还可以将内部的`Path`构建为无向图结构，进行基于无向图的一些查询操作。
- 再可以构建一个简单的`RailwaySystem`地铁系统，进行一些基本的查询操作。

基本任务

本次作业的程序主干逻辑我们均已经实现，只需要同学们完成剩下的部分，即：

- **继承`Path`、`RailwaySystem`两个官方提供的接口，通过设计实现自己的`Path`、`RailwaySystem`容器，来满足接口对应的规格。**

需要特别说明的是，`RailwaySystem`继承了上次作业的`Graph`接口，为其拓展了一些本次作业新增的功能。**`Graph`接口、`PathContainer`接口和上次相比均没有变化。**

每个模块的**接口定义源代码和对应的JML规格**都已在接口源代码文件中给出，各位同学需要**准确理解JML规格**，然后使用Java来实现相应的接口，并保证**代码实现严格符合对应的JML规格。**

具体来说，各位同学需要新建两个类`MyPath`和`MyRailwaySystem`（仅举例，具体类名可自行定义并配置），并实现相应的接口方法，每个方法的代码实现需要严格满足给出的JML规格定义。

当然，还需要同学们在主类中通过调用官方包的`AppRunner`类，并载入自己实现的`MyPath`、`MyRailwaySystem`类，来使得程序完整可运行，具体形式下文中有提示。

地铁系统相关概念定义

换乘

两个结点的路径中每更换一次线路（Path）即视为换乘一次。

举例来说，假设有以下两条地铁线路：

- 1号线：`1 -- 2 -- 3`
- 2号线：`4 -- 3 -- 5`

则路线`2 --> 3 --> 5`中，经过3号地铁站时，则需要进行一次换乘（从1号线换至2号线）。

票价

- 设从某一站，到另一站，需要经过x条边，以及y次换乘。
- 则走此路线所需要的票价为$T_{ticket} = x + 2y$。
- 以上述的例子来说
  - 路线`2 --> 3 --> 5`中
  - 经过了2条边
  - 进行过一次换乘
  - 则$x = 2, y = 1$，可得$T_{ticket} = 2 + 2 \times 1 = 4$，票价为4

不满意度

- 由于这些地铁站，有不同的拥挤程度，所以用户在经过一些边的时候，也会产生一定的不满意度。
- 设函数$F(x) = (x\%5 + 5)\%5$（即对5取模，且保证结果为正数）
- 设函数$H(x) = 4^x$
- 则边$E(u, v)$的不满意度定义为$U_e(E) = H(\max(F(u), F(v)))$
- 此外，每进行一次换乘行为，会产生$U_s = 32$点不满意度（固定值）
- 对于一条路径而言，设其所经过的边的集合为S_E，共进行过y次换乘，则总不满意度为
  $U = \sum\limits_{E \in S_E} U_e(E) + yU_s$
- 以上述的例子来说
  - 路线`2 --> 3 --> 5`中
  - 共计进行过一次换乘，$y = 1$
  - 对于边$E_1(2, 3)$和边$E_2(3, 5)$

$$
\begin{aligned}
U_e(E_1) &= H(\max(F(2), F(3))) \\
&= H(\max(2, 3)) \\
&= H(3) = 4^3 = 64.
\end{aligned}
$$

$$
\begin{aligned}
U_e(E_2) &= H(\max(F(3), F(5))) \\
&= H(\max(3, 0)) \\
&= H(3) = 4^3 = 64.
\end{aligned}
$$

故

$$
\begin{aligned}
U &= \sum_{E \in S_E} U_e(E) + yU_s \\
&= U_e(E_1) + U_e(E_2) + 1 \cdot U_s \\
&= 64 + 64 + 1 \times 32 \\
&= 160.
\end{aligned}
$$

总不满意度为160。

测试模式

针对本次作业提交的代码实现，课程将使用公测+互测+bug修复的黑箱测试模式，具体测试规则参见下文。

类规格

Path类

`Path`的具体接口规格见官方包的开源代码，此处不加赘述。

除此之外，`Path`类必须实现一个构造方法

```
public class MyPath implements Path {
    public MyPath(int[] nodeList);
}
```

或者

```
public class MyPath implements Path {
    public MyPath(int... nodeList);
}
```

构造函数的逻辑为按照`nodeList`数组内的结点序列生成一个`Path`对象。

请确保构造函数正确实现，且类和构造函数均定义为`public`。`AppRunner`内将自动获取此构造函数进行`Path`实例的生成。

（注：这两个构造方法实际本质上等价，不过后者实际测试使用时的体验会更好，想要了解更多的话可以百度一下，关键词：`Java 变长参数`）

RailwaySystem类

`RailwaySystem`类的接口规格见官方包的开源代码，此处不加赘述。

除此之外，`RailwaySystem`类必须实现构造方法

```
public class MyRailwaySystem implements RailwaySystem {
    public MyRailwaySystem();
}
```

构造函数的逻辑为生成一个空的`RailwaySystem`对象。

请确保构造函数正确实现，且类和构造函数均定义为`public`。`AppRunner`内将自动获取此构造函数进行`RailwaySystem`实例的生成。

输入输出

本次作业将会下发输入输出接口和全局测试调用程序，前者用于输入输出的解析和处理，后者会实例化同学们实现的类，并根据输入接口解析内容进行测试，并把测试结果通过输出接口进行输出。

输出接口的具体字符格式已在接口内部定义好，各位同学可以阅读相关代码，这里我们只给出程序黑箱的字符串输入输出。

规则

- 输入一律在标准输入中进行，输出一律在标准输出。
- 输入内容以指令的形式输入，一条指令占一行，输出以提示语句的形式输出，一句输出占一行。
- 输入使用官方提供的输入接口，输出使用官方提供的输出接口。

指令相关概念

- **结点**：一个结点是一个int范围（32位有符号）的整数
- **结点序列**：一个结点序列由n个结点组成，每个整数之间以一个空格分隔，其中$2 <= n <= 80$
- **路径id**：任意int范围（32位有符号）整数（每一个成功加入容器的路径都有一个唯一的id，且按照加入的先后顺序从1开始递增）

指令格式一览

添加路径

输入指令格式：`PATH_ADD 结点序列`

举例：`PATH_ADD 1 2 3`

输出：

- `Ok, path id is x.` x是调用返回的路径id，如果容器中已有相同的路径，则返回已有的路径id，否则返回添加后的路径id

删除路径

输入指令格式：`PATH_REMOVE 结点序列`

举例：`PATH_REMOVE 1 2 3`

输出：

- `Failed, path not exist.` 容器中不存在结点序列对应的路径
- `Ok, path id is x.` 容器中存在路径，x是被删除的路径id

根据路径id删除路径

输入指令格式：`PATH_REMOVE_BY_ID 路径id`

举例：`PATH_REMOVE_BY_ID 3`

输出：

- `Failed, path id not exist.` 容器中不存在路径id对应的路径
- `Ok, path removed.` 容器中存在路径，x是被删除的路径id

查询路径id

输入指令格式：`PATH_GET_ID 结点序列`

举例：`PATH_GET_ID 1 2 3`

输出：

- `Failed, path not exist.` 容器中不存在结点序列对应的路径
- `Ok, path id is x.` 容器中存在路径，x是被查询的路径id

根据路径id获得路径

输入指令格式：`PATH_GET_BY_ID 路径id`

举例：`PATH_GET_BY_ID`

输出：

- `Failed, path id not exist.` 容器中不存在路径id对应的路径
- `Ok, path is x.` 容器中存在路径，x是用英文圆括号包起来的结点序列，例如：`Ok, path is (1 2 3).`

获得容器内总路径数

输入指令格式：`PATH_COUNT`

举例：`PATH_COUNT`

输出：`Total count is x.` x是容器内总路径数

根据路径id获得其大小

输入指令格式：`PATH_SIZE 路径id`

举例：`PATH_SIZE 3`

输出：

- `Failed, path id not exist.` 容器中不存在路径id对应的路径
- `Ok, path size is x.` 容器中存在路径，x是该路径的大小

根据路径id获取其不同的结点个数

输入指令格式：`PATH_DISTINCT_NODE_COUNT 路径id`

举例：`PATH_DISTINCT_NODE_COUNT 5`

输出：

- `Failed, path id not exist.` 容器中不存在路径id对应的路径
- `Ok, distinct node count of path is x.` 容器中存在路径，x是路径中不同的结点个数

根据结点序列判断容器是否包含路径

输入指令格式：`CONTAINS_PATH 结点序列`

举例：`CONTAINS_PATH 1 2 3`

输出：

- `Yes.` 如果容器中包含该路径
- `No.` 如果容器中不包含该路径

根据路径id判断容器是否包含路径

输入指令格式：`CONTAINS_PATH_ID 路径id`

举例：`CONTAINS_PATH_ID 3`

输出：

- `Yes.` 如果容器中包含该路径
- `No.` 如果容器中不包含该路径

容器内不同结点个数

输入指令格式：`DISTINCT_NODE_COUNT`

举例：`DISTINCT_NODE_COUNT`

输出：`Ok, distinct node count is x.` x是容器内不同结点的个数

根据字典序比较两个路径的大小关系

输入指令格式：`COMPARE_PATHS 路径id 路径id`

举例：`COMPARE_PATHS 3 5`

输出：

- `Ok, path x is less than y.` 字典序x小于y，x是输入的第一个路径id，y是第二个路径id
- `Ok, path x is greater than y.` 字典序x大于y，x是输入的第一个路径id，y是第二个路径id
- `Ok, path x is equal to y.` 字典序x等于y，x是输入的第一个路径id，y是第二个路径id
- `Failed, path id not exist.` 输入的两个路径id中任意一个不于容器中存在

路径中是否包含某个结点

输入指令格式：`PATH_CONTAINS_NODE 路径id 结点`

举例：`PATH_CONTAINS_NODE 3 5`

输出：

- `Failed, path id not exist.` 容器中不存在路径id对应的路径
- `Yes.` 容器中存在该路径，且包含此结点
- `No.` 容器中存在该路径，但不包含此结点

容器中是否存在某个结点

输入指令格式：`CONTAINS_NODE 结点id`

举例：`CONTAINS_NODE 3`

输出：

- `Yes.` 容器中存在结点id对应的结点
- `No.` 容器中不存在结点id对应的结点

容器中是否存在某一条边

输入指令格式：`CONTAINS_EDGE 起点结点id 终点结点id`

举例：`CONTAINS_EDGE 3 5`

输出：

- `Yes.` 容器中存在以起点结点id和终点结点id为两个端点的一条边
- `No.` 容器中不存在以起点结点id和终点结点id为两个端点的一条边

两个结点是否连通

输入指令格式：`IS_NODE_CONNECTED 起点结点id 终点结点id`

举例：`IS_NODE_CONNECTED 3 5`

输出：

- `Yes.` 容器中的各路径构成的图上，起点节点和终点节点之间连通（即在同一连通块上）
- `No.` 容器中的各路径构成的图上，起点节点和终点节点之间不连通（即不在同一连通块上）
- `Failed, node id not exist.` 输入的两个结点id中任意一个不于容器中存在

两个结点之间的最短路径

输入指令格式：`SHORTEST_PATH_LENGTH 起点结点id 终点结点id`

举例：`SHORTEST_PATH_LENGTH 3 5`

输出：

- `Ok, length is len.` len为最短路径的长度
- `Failed, node id not exist.` 输入的两个结点id中任意一个不于容器中存在
- `Failed, node not connected with each other.` 输入的两个结点id不相连

整个图中的连通块数量

输入指令格式：`CONNECTED_BLOCK_COUNT`

举例：`CONNECTED_BLOCK_COUNT`

输出：

- `Ok, connected block count is x.` x是整个图中的连通块数量(连通块的定义请参考图论)

两个结点之间的最低票价

输入指令格式：`LEAST_TICKET_PRICE 起点结点id 终点结点id`

举例：`LEAST_TICKET_PRICE 5 9`

输出

- `Ok, least price is x.` x是两个结点之间的最低票价
- `Failed, node id not exist.` 输入的两个结点id中任意一个不于容器中存在
- `Failed, node not connected with each other.` 输入的两个结点id不相连

两个结点之间的最少换乘次数

输入指令格式：`LEAST_TRANSFER_COUNT 起点结点id 终点结点id`

举例：`LEAST_TRANSFER_COUNT 5 9`

输出

- `Ok, least transfer count is x.` x是两个结点之间的最少换乘次数
- `Failed, node id not exist.` 输入的两个结点id中任意一个不于容器中存在
- `Failed, node not connected with each other.` 输入的两个结点id不相连

两个结点之间的最少不满意度

输入指令格式：`LEAST_UNPLEASANT_VALUE 起点结点id 终点结点id`

举例：`LEAST_UNPLEASANT_VALUE 5 9`

输出

- `Ok, least unpleasant value is x.` x是两个结点之间的最少不满意度
- `Failed, node id not exist.` 输入的两个结点id中任意一个不于容器中存在
- `Failed, node not connected with each other.` 输入的两个结点id不相连

样例

#	标准输入	标准输出
1	PATH_ADD 1 2 2 3 PATH_REMOVE 1 2 3 CONNECTED_BLOCK_COUNT SHORTEST_PATH_LENGTH 1 3 LEAST_TICKET_PRICE 1 3	Ok, path id is 1. Failed, path not exist. Ok, connected block count is 1. Ok, length is 2. Ok, least price is 2.
2	PATH_ADD -3 0 5 PATH_ADD -2 0 PATH_REMOVE 0 -2 SHORTEST_PATH_LENGTH -3 -2 LEAST_TRANSFER_COUNT -3 -2 CONNECTED_BLOCK_COUNT	Ok, path id is 1 Ok, path id is 2. Failed, path not exist. Ok, length is 2. Ok, least transfer count is 1. Ok, connected block count is 1.
3	PATH_ADD 1 2 3 5 PATH_ADD 1 2 4 3 4 PATH_ADD 1 2 3 5 DISTINCT_NODE_COUNT CONNECTED_BLOCK_COUNT SHORTEST_PATH_LENGTH 2 3 LEAST_UNPLEASANT_VALUE 2 3	Ok, path id is 1. Ok, path id is 2. Ok, path id is 1. Ok, distinct node count is 5. Ok, connected block count is 1. Ok, length is 1. Ok, least unpleasant value is 64.
4	PATH_ADD 1 2 3 4 5 PATH_ADD 2 5 7 8 9 PATH_ADD 3 6 8 10 11 PATH_ADD 12 14 15 SHORTEST_PATH_LENGTH 1 11 LEAST_TICKET_PRICE 5 10 LEAST_TRANSFER_COUNT 1 15 PATH_ADD 4 14 CONNECTED_BLOCK_COUNT LEAST_TRANSFER_COUNT 1 15 LEAST_UNPLEASANT_VALUE 1 15	Ok, path id is 1. Ok, path id is 2. Ok, path id is 3. Ok, path id is 4. Ok, length is 6. Ok, least price is 5. Failed, node not connected with each other. Ok, path id is 5. Ok, connected block count is 1. Ok, least transfer count is 2. Ok, least unpleasant value is 912.

关于判定

数据基本限制

- 输入指令数
  - 总上限为6000条
  - 其中，下述指令被称为线性指令，总数不超过1000条
    - `PATH_ADD`
    - `PATH_REMOVE`
    - `PATH_REMOVE_BY_ID`
    - `PATH_GET_ID`
    - `PATH_GET_BY_ID`

        - `CONTAINS_PATH`
        - `COMPARE_PATHS`
    - 下述指令被称为图结构变更指令，总数不超过50条
        - `PATH_ADD`
        - `PATH_REMOVE`
        - `PATH_REMOVE_BY_ID`
- 输入指令满足标准格式。
- 输入结点序列最长为80，最短为2，其中结点编号可重复。
- 输入的路径id必须是int范围（32位有符号）的整数。
- 本次由路径构成的图结构，在任何时候，**总节点个数（不同的节点个数，即`DISTINCT_NODE_COUNT`的结果），均不得超过120。**
- 满足以上基本限制的输入数据都是合法数据。也即互测数据需要满足的基本条件。

测试模式

公测和互测都将使用指令的形式模拟容器的各种状态，从而测试各个接口的实现正确性，即是否满足JML规格的定义。**可以认为，只要代码实现严格满足JML，就能保证正确性。**

任何满足规则的输入，程序都应该保证不会异常退出，如果出现问题即视为未通过该测试点。

程序的最大运行cpu时间为35s，虽然保证强测数据有梯度，但是还是**请注意时间复杂度的控制。**

其他

- 本次所涉及到的图结构
    - **由所有的容器内的路径构成**（具体来说，可以把每条路径视为一条地铁线路。相同的节点编号代表同一地铁站，由此构成的地铁交通网络图）

提示

- 如果还有人不知道标准输入、标准输出是啥的话，那在这里解释一下
    - 标准输入，直观来说就是屏幕输入
    - 标准输出，直观来说就是屏幕输出
    - 标准异常，直观来说就是报错的时候那堆红字
    - 想更加详细的了解的话，请去百度
- 本次作业中可以自行组织工程结构。任意新增`java`代码文件。只需要保证两个类的继承与实现即可。
- **关于本次作业容器类的设计具体细节，本指导书中均不会进行过多描述，请自行去官方包开源仓库中查看接口的规格，并依据规格进行功能的具体实现**，必要时也可以查看AppRunner的代码实现。关于官方包的使用方法，可以去查看开源库的`README.md`。
- 开源库地址：传送门
- 推荐各位同学在课下测试时使用Junit单元测试来对自己的程序进行测试
    - Junit是一个单元测试包，**可以通过编写单元测试类和方法，来实现对类和方法实现正确性的快速检查和测试。**还可以查看测试覆盖率以及具体覆盖范围（精确到语句级别），以帮助编程者全面无死角的进行程序功能测试。

    - Junit已在评测机中部署（版本为Junit4.12，一般情况下确保为Junit4即可），所以项目中可以直接包含单元测试类，在评测机上不会有编译问题。
    - 此外，Junit对主流Java IDE（Idea、eclipse等）均有较为完善的支持，可以自行安装相关插件。推荐两篇博客：
        - Idea下配置Junit
        - Idea下Junit的简单使用
    - 感兴趣的同学可以自行进行更深入的探索，百度关键字：`Java Junit`。
- 本系列作业中
    - 其实不难发现，每一次的功能是单调递增的，而且具备一定的继承性。
    - 所以**建议大家采用继承的方式，将每一次的逻辑进行独立封装**，以提高程序的工程性、可维护性。
    - 此外，实际上这个系列作业，从`PathContainer`，到`Graph`再到`RailwaySystem`，这个**很形象的描述了实际OOP开发中，一个功能模块的演化过程。**从低层次抽象，逐步形成一个面向实际需求的模块。大家可以好好感受一下
- 本次作业中
    - 请求数看似很多，实际上写指令很少，图结构变更指令更少。
    - 这毫无疑问意味着需要充分优化各类读指令。
    - 此外，由于图结构相对静态，所以可以考虑以下的策略
        - 将时间复杂度分散到本就无法降低复杂度的写指令以及线性复杂度指令中。
        - 将之前计算出来的部分中间结果保存下来，以减少后续的计算复杂度。
        - 一言以蔽之，应当使用类似缓存的思想，**将大量的计算任务按照一定的策略进行均摊，以减少重复劳动。**
        - 具体来说，如果你使用的是单源最短路系算法的话（spfa、dijkstra等）
            - 假设你本次求解的是x到y的最短路
            - 那么，**实际上你半道上求出来的最短路结果，可远不止(x,y)这么一组。**具体来说，最起码包含了其他全部从x开头的结果，以及一些其他的中间结果（因算法而异）。
            - 然后，你需要做的，就是充分利用好这些本次暂时用不上的中间结果，使得以后的最短路运算被不断加速（准确的说，**这样的架构下，求解次数越多，后期速度越快**）。
            - 比如，你可以考虑**单独开一个图，初始只包含初始的边**，对于**已经求出来的全部最短路结果**（无论本次是否用得上），都**作为边加入到这张图中**（后续经过此段的时候，不再需要一步步算），并**做好标记**（对于访问已有结果的情况，可以直接出解），使得这张图不断滚雪球，运算速度越来越快。
    - 此外，由于本次**涉及到大量图相关的计算**，并且很多逻辑实际上很类似
        - 所以**建议将图相关的计算进行单独的开类封装，并进行单独维护。**
        - 同理，**上文所说的缓存系统，也最好和图一样，进行单独维护**（或者就以图的形态存在，不过还是建议尽可能降低耦合）。
        - 不仅如此，本次还存在较多相通的图建模形态（即不再是简单的裸建模，但是形态结构基本类似），所以可以**将类似的建模形态，基于图类进行一层封装，充分运用组合模式和工厂模式，优化工程性。**
    - 我知道你们想吐槽这个太数据结构了，但是实际上在工程领域中，各类数据结构最常见的存在形式，恰恰就是面向对象的形式，你们使用的HashMap等数据结构就是最好的例子。而**对于数据结构对象而言，正确性、复杂度性能、工程性的平衡折中，则是极为重要的，也是最能体现设计水准的。**
    - 综上，虽然是本作业的主题是规格化设计。不过还是建议各位同学们，**在架构设计上多思考，也多感悟感悟规格化设计的思想精髓。**

- **不要试图通过反射机制来对官方接口进行操作**，我们有办法进行筛查。此外，在互测环节中，如果发现有人试图通过反射等手段hack输出接口的话，请邮件niuyazhe@buaa.edu.cn进行举报，**经核实后，将直接作为无效作业处理。**