BUAA-OO-Unit3总结

本单元的主题是**JML规格化**设计，在三次作业中，我们需要根据课程组给定的JML规格完成相应的方法，同时要考虑到实现功能的性能优化，最后还要使用juint来对指定的方法进行测试。

本单元的测试过程

黑箱测试&白箱测试

- **黑箱测试**又称为**功能测试**，主要检测软件的每一个功能是否能够正常使用。在测试过程中，将程序看成不能打开的黑盒子，不考虑程序内部结构和特性的基础上通过程序接口进行测试，检查程序功能是否按照设计需求以及说明书的规定能够正常打开使用。
- **白箱测试**又称为**结构测试**，这种方法是把测试对象看做一个打开的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序的所有逻辑路径进行测试，通过在不同点检查程序状态，确定实际状态是否与预期的状态一致。

总的来说，黑箱测试更关注于外部行为和功能，而白箱测试更关注于内部实现和代码执行路径。我们OO的中测和强测都属于黑箱测试，只需要通过测试点即可；而互测属于白箱测试，我们可以通过查看对方的代码结构，找出其中的缺陷，然后进行针对性的hack。为了提高自己代码的正确性，我们应当全面地进行黑箱和白箱测试，对自己的的代码进行查漏补缺。

单元测试、功能测试、集成测试、压力测试、回归测试

- **单元测试（Unit Testing）**是针对软件中的最小可测试单元（通常是函数或方法）进行的测试。旨在验证每个单元的功能是否按预期工作。
- **功能测试（Functional Testing）**是对软件的功能进行测试，确保软件按照需求规格的要求正常工作，它并不关心代码的内部结构，只关心代码是否能够正常行使功能。
- **集成测试（Integration Testing）**是将不同的模块或组件整合在一起，并测试它们之间的交互和接口，目标是确保整个系统的各个部分能够正确地协同工作。
- **压力测试（Stress Testing）**旨在评估系统在负载增加的情况下的性能表现。通过模拟高负载情况下的用户访问或数据处理，以确定系统的性能极限和稳定性。
- **回归测试（Regression Testing）**是在对软件进行修改或更新后执行的测试，以确保已有功能未受影响。它涉及重新运行先前的测试用例，以确保软件在进行更改后仍然能够正常运行。

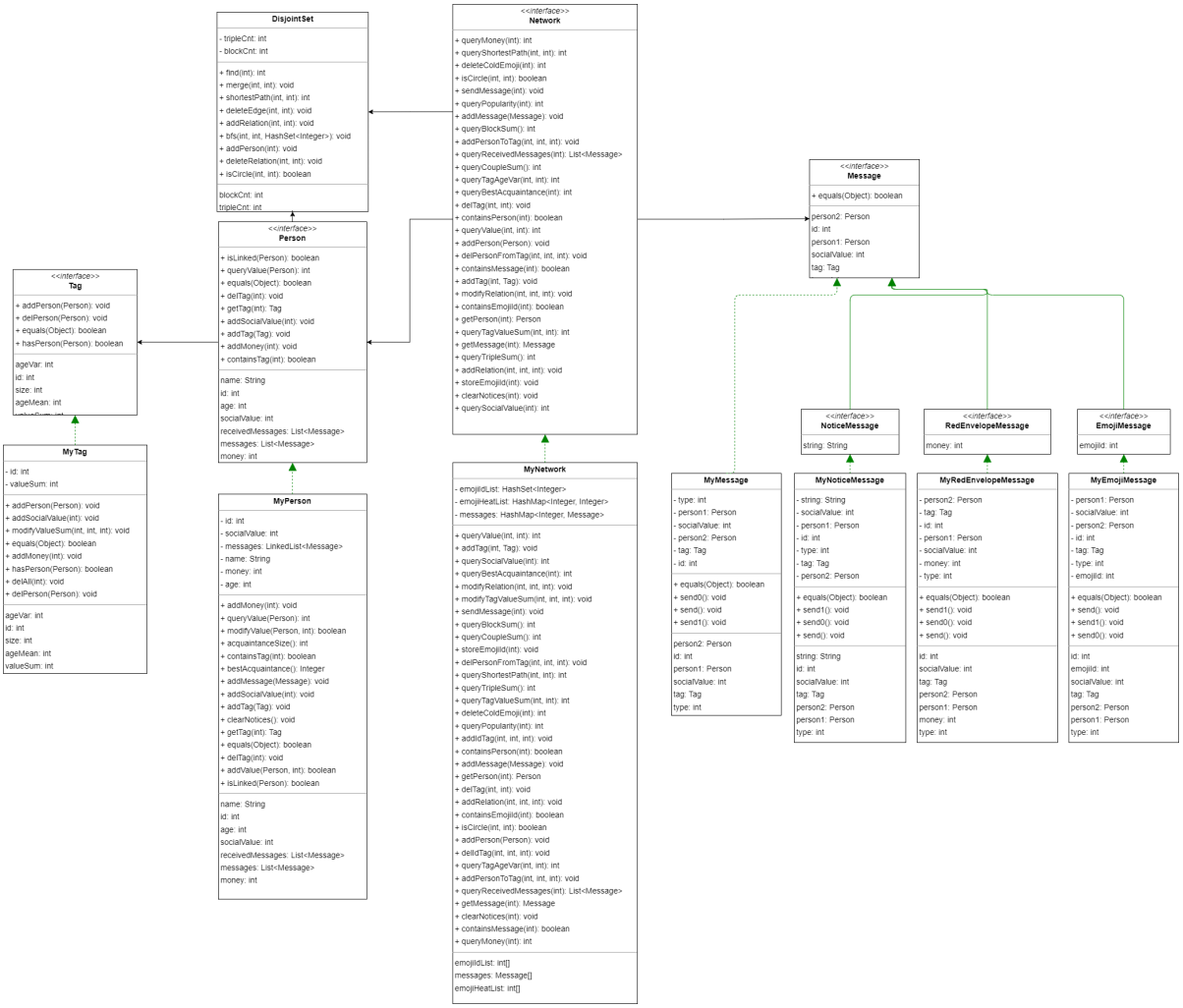
这些测试方法在完成作业的过程中几乎都有涉及。当需要针对某个方法进行测试的时候，我们会使用单元测试，比如junit；当我们要确保程序能行使正确的功能的时候，会使用功能测试；当我们要确保程序中不同的类之间能够正确进行协作或者能够正确传递数据的时候，会使用集成测试；当我们要确保程序会不会因为性能问题而超时的时候，会使用压力测试；在我们找到并修改bug的时候，我们会使用回归测试确保修改不会影响到程序原先的正确功能。

数据构造策略

- **边界值分析。**通过构造0，1，2^31-1等这些边界值，我们有可能会发现程序潜在的问题。比如，第二次作业中涉及的对id的比较，是不能够直接相减的，可能会超出int范围。
- **等价类划分。**这种策略将输入数据划分为等价类，即具有相同功能和行为的输入数据集合。然后从每个等价类中选择一个代表性的数据进行测试，以覆盖整个等价类。在我们的程序中，指令可以大致分为**person,tag,message**几类，当然他们之间可能会有交叉，我们可以对它们做一些针对性的测试。

- **随机数据生成。**随机数据生成可以帮助发现系统对于不同类型和范围的输入数据的行为。这是评测机通常使用的方法，可以覆盖掉比较常见的一些数据。
- **特殊情况测试。**这种策略针对特定的边缘情况或异常情况构造测试数据，以验证系统对这些情况的处理能力。此类数据一般针对于那些评测机难以生成和覆盖的数据。随机数据+特殊数据一般能够覆盖掉程序的绝大部分情况。

架构设计



这个单元的架构几乎都是助教给好的，我们几乎只需要思考如何在完全满足JML语言的基础之上去提升性能。几乎每次作业中也只有那一两个地方需要思考一下性能的问题。

hw9

第九次作业中，比较复杂的问题是`isCircle`，`queryBlockSum`和`queryTripleSum`方法，我们需要思考如何快速地查询两个人之间是否有联系。把问题抽象出来就是需要去批量地查找无向图中某两个点是否连通。

最终选用的方法是**并查集**，并查集的特点在于有一个`parent`容器维护每个节点的父节点，用集合中的一个元素来代指这一个集合。在并查集中进行增加节点，增加一条边等操作的时候都非常方便，为了提升性能，我还进行了**路径压缩和按秩合并**。然而，在并查集中进行删边的时候可能会比较麻烦。如果我想要删除A和B之间的一条边，首先将A的parent设置为A，然后对A进行遍历，将遍历到的节点的parent全部设置成A；然后查看有没有遍历到B，如果遍历到了，说明AB还在同一个连通分支下，如果没有，则对B也进行上述操作，此时整个图的连通分支增加了一个。

在这样的并查集中，上述三个方法就很容易实现了：

- `isCircle`：只需要比较两个节点的根节点是否是同一个就可以了。

- `queryBlockSum`：在并查集中动态维护了一个`blockCnt`。增加一个人的时候，或者上述并查集中可能出现的删边情况中，也就是增加连通分支的时候，让`blockCnt++`；增加两个人之间的关系导致两个连通分支合并的时候，`blockCnt--`。
- `quaryTripleSum`：在并查集中动态维护了一个`tripleCnt`。在增加AB关系和删除关系的时候，遍历查找所有和AB都有关系的节点，进行动态维护。

虽然这种方法会降低增加或者修改两人之间关系的性能（最多o(n)），但是能使上述三个方法达到o(1)的复杂度。

有一个比较重要的点是在对图进行遍历的时候，不管是dfs还是bfs，都不要使用递归的方法，否则有可能爆栈，可以使用队列来遍历。

hw10

第十次作业中，`queryTagValueSum、queryTagAgeVar、queryBestAcquaintance、queryCoupleSum、queryShortestPath`这五个方法比较复杂。

queryTagValueSum

这个方法要查找一个tag中所有联系之间的value的和，最好的方法肯定是动态维护。问题在于，当两个人之间的关系值发生变化，或者一个人从tag中被删除后，该怎么维护。我的解决方法是用一个容器存储每个人被包含在哪些tag中，从人能够直接找到相应的tag，这样的话，在相应属性发生变化的时候，可以直接进入有关的tag，修改valueSum的值。

queryTagAgeVar

这个方法比较容易解决，我们可以对方差公式进行化简，然后发现只需要维护一下age的总和和平方和就可以了。不过需要注意的一点是，必须要严格按照ML给的式子化简，不然可能会存在误差。

queryBestAcquaintance

解决这个方法，我们可以维护每个人的bestAcquaintance，可以采用TreeMap容器，firstKey对应的值就是所需要的。

```
this.acquaintance = new TreeMap<>((id1, id2) -> {
            int cmp = value.get(id2).compareTo(value.get(id1));
            if (cmp == 0) {
                return id1.compareTo(id2);
            }
            return cmp;
        });
```

在增加关系的时候，TreeMap会动态维护，唯一不同的一点是在修改关系的时候不能直接修改，必须先将原来的键值对删掉，然后加上新的，TreeMap才能正确更新。

queryCoupleSum

直接遍历一遍persons就好，o(n)的复杂度也可以接受。

queryShortestPath

这个方法我选择了使用bfs进行遍历，没有使用复杂的算法，复杂度也可以满足条件。

hw11

第十一次作业中没有什么需要着重去优化的地方，`queryReceivedMessages`方法使用了LinkedList容器来解决。`deleteColdEmoji`直接用迭代器去遍历删除就可以了，o(n)的复杂度可以接受。

性能问题及其修复情况

性能问题

在第九次作业中，我出现了一个性能bug。

```
public void bfs(int id, int parent, HashSet<Integer> visited) {
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(id);
    while (!queue.isEmpty()) {
        int now = queue.poll();
        visited.add(now);
        for (int link : this.edge.get(now)) {
            if (!visited.contains(link)) {
                queue.offer(link);
                this.parent.put(link, parent);
            }
        }
    }
}
```

在遍历到结点之后，我没有第一时间将节点加入到visited队列中，这样导致bfs的速度非常慢，可能会有很多节点重复遍历，再加上压力测试做的不充分，导致出现bug。修改后如下（只是多加了一行）：

```
public void bfs(int id, int parent, HashSet<Integer> visited) {
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(id);
    while (!queue.isEmpty()) {
        int now = queue.poll();
        visited.add(now);
        for (int link : this.edge.get(now)) {
            if (!visited.contains(link)) {
                visited.add(link);            //<--
                queue.offer(link);
                this.parent.put(link, parent);
            }
        }
    }
}
```

规格与实现分离

在本次作业中，我感受到了规格的巨大好处。规格严格要求了程序中每个函数的作用，可以改变的属性，在此基础上，规格对函数做出了明确的限制，而具体的实现过程则交给代码的编写者来完成。

规格与实现的分离，大大减轻了编写代码时的困难。规格一旦给定，相当于程序的框架就已经确定了，在这种情况下，我们只需要按照要求把代码补全就可以了，而不用一边构思一边写，大大降低了编写代码时遗漏某些特殊情况的可能，提高了代码的质量。

Junit测试方法

本单元我们需要使用junit来进行测试。它有点类似于一个对拍工具，需要我们自己去构造样例，然后比对运行的结果。

数据构造

在构造数据的时候，我们需要尽可能保证测试样例能够覆盖到所有的代码所有的分支。我采用了随机生成多组数量大小合适的数据，基本能够覆盖常见的一些bug。

与规格的一致性

在编写Junit程序的时候，我们要严格对每一条 requires、assignable、ensures 涉及到的数据进行检查。对于signals，我们要确保能够正确抛出异常。对于pure方法，我们要确保调用方法前后，对象中的各种相关属性是否被修改掉，这其中就有可能出现深浅拷贝的问题。总之就是确保应该改变的属性被正确修改，不应该改变的属性不能发生变化。

学习体会

本单元的主题是基于规格的层次化设计，不得不说，在有了规格的规范作用下，编写代码的效率极大地得到了提升。在本单元中，我们需要正确读懂课程组给定的JML语言，写出相应的代码。规格虽然写起来复杂，但是确实能够提高代码的正确性和安全性，对于一些要求比较高的代码来说还是十分有必要的。

这个单元比较遗憾的地方在于第九次和第十一次作业中均出现了一些比较低级的错误，这说明在强测之前的测试做的还不够好，希望最后一个单元能够再接再厉。