金庸的江湖——金庸武侠小说中的人物关系挖掘

组长：韩畅，组员：李展烁、王一之、闫旭芃

2020 年 7 月 31 日

1 实验规划与设计

1.1 任务分配

171860551, 韩畅：组长,算法设计与实验规划，任务一、任务六，程序试运行与组织debug研讨
171860550, 王一之：算法设计与实验规划，任务四，参与debug，实验版本控制
171860549, 闫旭芃：算法设计与实验规划，任务五，参与debug与数据核对
171840565, 李展烁：算法设计与实验规划，任务一优化、任务二、任务三，参与debug并提出重要优化思路

1.2 任务要求

1.3 设计思路

2 实验实现

2.1 任务一：数据预处理

2.1.1 设计思路

数据输入：已经分词、分段的多篇中文文本文件

数据输出：每一段或几段中包含的所有人名，按顺序依次输出。

需要注意的是，由于我们的实验是对金庸全部人物关系的分析，因此无需特别地注意不同文件及文件名。

我们使用提供的名单列表，通过HashSet结构，快速地比对词语是否位于名单列表当中。

在装载名单列表时，由于姓名数量的有限性，且由于 configuration只能传输字符串，因此使用一个私有的将其全部装载到一个字符串中，填入configuration中，并在map的setup阶段将其提取出来，并存储于HashSet中。

特别地，为了提高效率，注意使用StringBuilder和String的相互配合。

2.1.2 程序分析

Main 主函数类：

为了传递名单列表，调用NameLoader类的load方法，从输入的文件名中获取名单，并使用configuration进行配置以备后用。其余大多为路径、类名的配置，不赘述。如图1

```java
public class GetCharaName {
    public static void main(String []args) throws Exception {
        Configuration myCfig = new Configuration(); // 配置初始化
        String[] argRemain = new GenericOptionsParser(myCfig, args).getRemainingArgs(); // 参数获取
        if (argRemain.length != 2 && argRemain.length != 3) {// 判断参数，错误判断，获取文件位置
            System.err.println("Usage: GetCharaName <in> <out> <<names>>");
            System.exit(2);
        }
        String pathNmFile = "../data/people_name_list.txt";// 给一个默认的文件路径
        if (argRemain.length == 3) {
            pathNmFile = argRemain[2];
        }
        NameLoader loader = new NameLoader();// 使用名单装载器装入名单
        String strAllName = loader.load(pathNmFile);

        Job job = Job.getInstance(myCfig, "GetCharaName");// 开始job，进行一系列设置
        job.setJarByClass(GetCharaName.class);
        job.setMapperClass(GetNameMapper.class);
        job.setReducerClass(GetNameReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        job.getConfiguration().set("Names", strAllName);// 使用configuration进行名单配置

        FileInputFormat.addInputPath(job, new Path(argRemain[0]));// 配置输入输出路径
        FileOutputFormat.setOutputPath(job, new Path(argRemain[1]));

        System.exit(job.waitForCompletion(true)? 0:1);

    }
}
```

图 1. 主函数类具体实现

NameLoader类：

使用FileInputStream获取文件内容，使用BufferReader进行缓冲区文件读入操作。

使用StringBuilder快速从单行读入中获取姓名，并快速构建字符串。如图2

```
class NameLoader {
    public String load(String filename) throws IOException {
        FileInputStream fIS = new FileInputStream(filename);// 使用文件名初始化文件输入流
        BufferedReader bR = new BufferedReader(new InputStreamReader(fIS));// 使用缓冲区读入文件流

        String ll = "";
        StringBuilder strAllName = new StringBuilder();// 使用StringBuilder快速构建名单
        while((ll = bR.readLine()) != null) {// 获取单行不为空
            strAllName.append(ll).append(" ");
        }

        fIS.close();
        bR.close();

        return strAllName.toString();// 将构建好的名单字符串转成字符串格式
    }
}
```

图 2. 名单装载器具体实现

Mapper类：

在setup中从configuration中获取之前装载的名单字符串，经过分词处理后，存入HashSet的名单表中。

在map的重载函数中，对每行也即每个value值以空格为分隔符分割，并依此使用HashSet的查找操作比对是否属于姓名。并将姓名归并为一个字符串输出。如图3

Reducer类：

不做操作，直接将获取的值输出即可。

2.2 任务二

2.2.1 设计思路

数据输入：每一段中包含的所有人名。

数据输出：对于所有的段落，统计所有人名对出现在同一段落中的次数。

使用IntWritable表示同现次数，因为同现次数在int可表示范围内。

2.2.2 程序分析

Main 主函数类：

这里使用了reducer作为combiner，在mapper之后进行一次合并，减少传输数据量，作为优化。如图4

Mapper类：

使用HashSet来对于同一段落中所有人名进行去重。再使用双重for循环来发射所有两个不同人名构成的人名对。因为是在HashSet内遍历，这里判断两个String不相等，不需要使用Java的!equals()来判断String内容，只需要判断引用不同即可。如图5

```java
class GetNameMapper extends Mapper<Object, Text, Text, NullWritable> {
    private HashSet<String> name_set = new HashSet<>();// 使用HashSet存储名单，以便快速比对单词与姓名
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        super.setup(context);
        String names = context.getConfiguration().get("Names");
        // 在setup中从configuration中获取名单字符串
        for (String name : names.split(" ")) {
            name_set.add(name);
        }
        // 使用空格分词后依次存入HashSet
    }

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        StringBuilder sb = new StringBuilder();// 使用stringBuilder快速对单行有效姓名进行连接
        for (String term : value.toString().split(" ")) {
            if (name_set.contains(term)) {
                sb.append(term).append(" ");
            }
        }
        // 使用空格分词，依次比对查找以后将有效姓名append到StringBuilder上

        if (sb.length() > 0) {
            context.write(new Text(sb.toString()), NullWritable.get());
        }
        // 把构建好的单行姓名列表输出
    }
}
```

图 3. mapper具体实现

```java
public class CooccurrenceCounting {
    Run | Debug
    public static void main(String []args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: CooccurrenceCounting <in> <out>");
            System.exit(2);
        }

        Job job = Job.getInstance(conf, "CooccurrenceCounting");
        job.setJarByClass(CooccurrenceCounting.class);
        job.setMapperClass(CooccurrenceMapper.class);
        // 使用reducer作为combiner，进行优化。
        job.setCombinerClass(CooccurrenceReducer.class);
        job.setReducerClass(CooccurrenceReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setNumReduceTasks(10);

        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

        System.exit(job.waitForCompletion(true)? 0:1);
    }
}
```

图 4. 主函数类具体实现

```java
class CooccurrenceMapper extends Mapper<Object, Text, Text, IntWritable> {
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] names = value.toString().split(" ");
        HashSet<String> names_set = new HashSet<>(Arrays.asList(names)); // 段落中所有人名的集合，目的是去重。
        // 双重for循环，生成所有不同人名对。
        for (String name1 : names_set) {
            for (String name2 : names_set) {
                if (name1 != name2) { // 这里不需要使用!equals()，只要!=就可以保证。
                    context.write(new Text("<" + name1 + "," + name2 + ">"), new IntWritable(1));
                }
            }
        }
    }
}
```

图 5. mapper具体实现

Reducer类：

对于key对应的人名对出现次数进行求和。如图6

```java
class CooccurrenceReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        //对人名对的同现次数进行求和统计。
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

图 6. reducer具体实现

2.3 任务三

todo

2.4 任务四:基于人物关系图的PageRank计算

2.4.1 PageRank算法介绍

PageRank，又称网页排名，名字源于google创始人之一的Larry Page，是Google公司所使用的对与网页重要性排序的算法。

PageRank通过网页之间的超链接评价网页重要性，它的基本思想是：

1) 如果一个网页被多个网页所指向，则该网页比较重要

2) 如果一个重要的网页指向另一个网页，则另一个网页也比较重要

该算法模拟一个上网者，随机打开一个网页，之后随机点击该网页的链接，统计上网者分布在每个网页的概率。

最初，每个网页的概率均等，每次跳转时，网页X将其PR(PageRank)均分到所指向的所有页

面，记链接数为L(X), 于是，经过一次跳转后：

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + ...$$

我们将每个网页抽象成一个节点，超链接抽象为有向边，共同构成一个图。则每次跳转可视为所有页面PR构成的特征向量R与该图的出度邻接矩阵M相乘，即：

$$R = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_n) \end{bmatrix} \quad M = \begin{bmatrix} p_1 \to p_1 & p_2 \to p_1 & \cdots & p_n \to p_1 \\ p_1 \to p_2 & p_2 \to p_2 & \cdots & p_n \to p_2 \\ \vdots & \vdots & \ddots & \vdots \\ p_1 \to p_n & p_2 \to p_n & \cdots & p_n \to p_n \end{bmatrix}$$

$$R_1 = MR_0$$

多次迭代后，PR值趋于稳定，即为最终的PR值。

2.4.2 设计思路

任务四的输入为任务三的输出，格式如下：
人物 [名字$_1$,影响$_1$—名字$_2$,影响$_2$—...—名字$_n$,影响$_n$]
影响$_i$ 为名字$_i$ 与该人物归一化后的同现次数，表示名字$_i$ 对该人物的影响权重。
每个人物视为图的一个节点，边权重为二人同现次数。
对于普通的PageRank计算，由于会存在自环边以及无出度的节点，为方式到达某一节点后陷入该点，会加入“随机浏览者”（random surfer）的概念，即到达某个节点后有一定概率直接跳转到任意一个节点，从而避免此情况。然而在此次任务中，首先没有自身与自身同现的情况，因此无自旋边；同时A与B同现，则B一定与A也同现，因此不考虑权重时所有边实际都为无向边，因此不存在出度为0的节点。所以此次任务无需引入“随机浏览者”。

MapReduce框架下，运算分布进行，因此不使用邻接矩阵，而采用邻接表的形式。算法大致分为三阶段：

阶段一：预处理首先要将输入格式化为供之后迭代处理的形式。采用如下格式： key：人物
value：PageRank#[名字$_1$,影响$_1$—名字$_2$,影响$_2$—...—名字$_n$,影响$_n$]
以概率为初始值，PageRank应设置为1/N，但N值较大，较小数字做乘法时误差较大，因此将初始PR设置为1来减小误差。

阶段二：迭代计算迭代计算PR值，直到PR收敛。
在Mapper中，首先输出如下键值对： key：人物
value：#出度表
此对目的在于维护出度表，value前加#使reducer便于区分。

之后计算PR值，记A的出度表集合为N,则计算过程如下式：

$$NewPR(A) = \sum_{x \in N} OldPR(x) * weight(x \to A)$$

计算得到新的PR值，再输出一组键值对： key：人物
value：新PageRank值
在Reducer中，首先查看value前是否有#号以区分该键值对类型。由于是一个迭代过程，将输出格式化，与Mapper的输入格式相同。

阶段三：处理结果在Mapper阶段去除结果中的出度表，只保留PR值。
利用Partition类进行排序，由于默认为升序，结果需要降序，因此重写DoubleWritable。因为只有1000余数据，未采用采样排序，使用了简单的全排序。
Reducer阶段整理输出即可

2.4.3 代码讲解

程序可分为三个模块：PageRank、RageResultSort以及调度模块。

模块一：PageRank
此模块实现了了阶段一与阶段二的任务。

```java
public class PageRank {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("PageRank neet 2 paras as input and output");
            System.exit(2);
        }
        String input = otherArgs[0];                // 输入目录
        String output = new String();
        for (Integer i = 0; i < 15; i++) {      // 迭代15轮
            if (i == 14) {
                output = otherArgs[1] + "final";    // 最终输出
            } else {
                output = otherArgs[1] + i.toString();    // 中间文件输出目录
            }
            Job job = Job.getInstance(conf, "PageRank");
            job.setJarByClass(PageRank.class);
            job.setMapperClass(PageRankMapper.class);
            job.setPartitionerClass(HashPartitioner.class);
            job.setReducerClass(PageRankReducer.class);
            job.setInputFormatClass(KeyValueTextInputFormat.class);// 以/t切割key与value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            job.setNumReduceTasks(5);                   // 设置五个Reducer节点
            FileInputFormat.addInputPath(job, new Path(input)); // 设置输入路径
            FileOutputFormat.setOutputPath(job, new Path(output));  // 设置输出路径
            job.waitForCompletion(true);    // 等待程序运行完毕
            input = output;
        }
    }
}
```

图 7. PageRankMain

图7为此类的main函数,使job运行15次，每次的结果储存在以运行次数为名的文件夹内，下一次迭代的输入为上一次的输出。最终结果储存在“final”文件夹下。由于程序中用不到行号，设置InputFormat为KeyValueTextInputFormat，这样会自动以分隔符(默认为\t)将一行的内容切割为key和value。

```java
class PageRankMapper extends Mapper<Text, Text, Text, Text> {
    private Text resKey = new Text();      // 要传入reduce的key和value
    private Text resValue = new Text();

    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        Double rank = 0.0;
        String relations = new String();
        if (value.toString().contains("#")) {// 不是第一次迭代
            rank = Double.parseDouble(value.toString().split("#")[0]); // 原本rank
            relations = value.toString().split("#")[1];       // 出度表 格式为[名字,影响|名字,影响|...]其中影响为对key中人物的影响
        } else {//第一次迭代，设置默认rank
            rank = 1.0;                // 初始化rank值为1
            relations = value.toString(); // 出度表
        }
        resKey.set(name);
        resValue.set("#" + relations);
        context.write(resKey, resValue);  // key为名字 value以#开头，后面为出度表，生成此键值对

        String[] slist = relations.split("\\[|\\]")[1].split("\\|");    // 分离出度表
        String target = new String();
        Double weight = 0.0;
        Double res = 0.0;
        for (String s : slist) {
            target = s.split(",")[0];   // 贡献影响的人名
            weight = Double.parseDouble(s.split(",")[1]);   // 影响值作为权重
            resKey.set(target);
            res = weight * rank;//得到此人对其造成的影响值
            resValue.set(res.toString());
            context.write(resKey, resValue);  // 发送键值对，key为受影响的人物名字，value为造成的影响值
        }

    }
}
```

图 8. PageRankMapper

图8为Mapper函数，此阶段,首先通过value中是否有#号来判断是否是第一次迭代。若为不为第一次，提取之前的RP，否则设置默认的PR值为1.0。通过value中的出度表，计算新PR值。向reducer发送“姓名 #出度表” 以及“姓名 PR值”两个键值对。

```java
class PageRankReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        Text resValue = new Text();
        String relations = new String();
        double sum = 0.0;
        for (Text t : values) {
            String value = t.toString();
            if (value.startsWith("#")) { // 以#开头的为出度表
                relations = value;
            } else {          // 否则为受影响值
                sum += Double.parseDouble(value);//; 累加得到其rank值
            }
        }
        resValue.set(String.format("%.6f", sum) + relations);
        context.write(key, resValue);
    }
}
```

图 9. PageRankReducer

图9为Reducer函数,通过#判断键值对类型，合并格式化后输出。

模块二：PageResultSort

此模块实现了阶段三的任务。

main函数中同样设置InputFormat为KeyValueTextInputFormat，同时设置reducer数量为1。

```java
class PageSortMapper extends Mapper<Text, Text, SortDoubleWritable, Text> {
    private SortDoubleWritable rKey = new SortDoubleWritable();
    private Text rValue = new Text();

    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        String rank = value.toString().split("#")[0];// 去除后面的关系表，只保留rank
        rank = rank.trim();// 去掉前后空白符
        rKey.set(Double.parseDouble(rank));// 简单的全排序，只有一个分区
        rValue.set(name);
        context.write(rKey, rValue);
    }
}
```

图 10. PageSortMapper

图10为Mapper函数，去除不需要的出度表。为进行排序，将PR值作为key，名字作为value。

```java
class SortDoubleWritable extends DoubleWritable { // 自定义DoubleWritable类 降序排列
    public SortDoubleWritable() {
        super();
    }

    public SortDoubleWritable(Double d) {
        super(d);
    }

    @Override
    public int compareTo(DoubleWritable o) { // 将返回值置反
        if (this.get() == o.get()) {
            return 0;
        } else if (this.get() > o.get()) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

图 11. SortDoubleWritable

```
class RankSortPartitioner extends HashPartitioner<SortDoubleWritable, Text> {// 重写HashPartitioner
    @Override
    public int getPartition(SortDoubleWritable key, Text value, int numReduceTasks) {// 重载getPartition函数，自定义的SortDoubleWritable类
        return super.getPartition(key, value, numReduceTasks);
    }
}
```

图 12. RankSortPartitioner

图12与图11为重载的Partitioner函数以及SortDoubleWritable类。重载Partitioner函数使得以DoubleWritable类型的key进行排序，由于默认DoubleWritable为升序，重载DoubleWritable为自定义SortDoubleWritable进行降序排列。

```
class PageSortReducer extends Reducer<SortDoubleWritable, Text, Text, Text> {
    @Override
    protected void reduce(SortDoubleWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for (Text name : values) {
            context.write(name, new Text(key.toString()));  // 将key和value置反
        }
    }
}
```

图 13. PageSortReducer

图13为Ruducer，将key与value互换位置，保证输出时名字在前。

2.5 任务五:在人物关系图上的标签传播

思路分析：在本实验里，我们拟采用 LPA(Label Propagation Algorithm) 标签传播算法对金庸小说中的人物进行聚类。 LPA 是一种半监督的图分析算法，能够为图中的每一个节点打上标签，对图的顶点进行聚类，从而在一张类似社交网络的图中完成社区发现。

2.5.1 LPA算法介绍

算法思路：

对于网络中的每一个节点，在初始阶段，Label Propagation算法对于每一个节点都会初始化一个唯一的一个标签。每一次迭代都会根据与自己相连的节点所属的标签改变自己的标签，更改的原则是选择与其相连的节点中所属标签最多的社区标签为自己的社区标签，这就是标签传播的含义了。随着社区标签不断传播。最终，连接紧密的节点将有共同的标签。

算法过程：

第一步：先给每个节点分配对应标签，即节点1对应标签1，节点i对应标签i；

第二步：遍历N个节点（for i=1：N），找到对应节点邻居，获取此节点邻居标签，找到出现次数最大标签，若出现次数最多标签不止一个，则随机选择一个标签替换成此节点标签；

第三步：若本轮标签重标记后，节点标签不再变化（或者达到设定的最大迭代次数），则迭代停止，否则重复第二步

具体伪代码如下图：

Input: Network $G = (V, E)$
Output: Label C(x) for each node
1: 初始化网络中所有的顶点，对于顶点 x，其标签 $C_0(x) = x$
2: **repeat**
3: 每一个顶点 x 向邻接的顶点发送自己标签 $C_t(x)$
4: 每一个顶点 x 从邻接顶点 $\{x_{n1}, x_{n2}, ..., x_{nk}\}$ 选择出现次数最多的标签作为自己新的标签 $C_{t+1}(x)$
5: **until** 每一个顶点的标签就是其邻接顶点中出现次数最多的标签

图 14. LPA算法伪代码

对于金庸小说中的人物关系的标签分析来说，其中人物的网络可以看成一个每条边带权值的有向图，因此我们的算法也要在此基础上做一些改变。首先我们还是对每一个顶点进行初始化，然后每一个顶点根据自己的出边向邻接顶点发送自己的标签。最后每个顶点从自己收到的标签里取权值和最大的标签作为自己新的标签（原算法的出现次数之和需要改成权值之和），相当于原算法中。当每一个顶点的标签不再改变时，算法终止。

具体分为3个部分进行:

2.5.2 第一部分： LabelInitial 社区初始化

第一个阶段，初始化每一个顶点，将每一个节点的人物姓名作为该节点的标签。(人名，(标签#邻接表))

Map：一行一行分析实验三的输出结果，直接将每个顶点的人名作为每个顶点的标签，输出

Reduce：不做任何处理直接输出 (人名，(标签#邻接表))

Mapper类：

一行行读入任务三输出的结果 (人名，邻接表)，然后将人名作为该任务的标签以 (人名，(标签，邻接表)) 的形式输出。初始时，每一个节点都是一个独立的社区，所以这里直接将标签赋值成该节点的人物姓名即可

```
class LabeliniMapper extends Mapper<Text, Text, Text, Text> {
    private Text rKey = new Text();
    private Text rValue = new Text();
    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        String Lablelist = value.toString();
        Lablelist=Lablelist.trim();
        rKey.set(name);
        rValue.set(name+"#"+Lablelist);
        context.write(rKey, rValue);
    }
}
```

图 15. LabelInitial Mapper

Reduce类：

直接输出即可，同时在main函数中设置job.setNumReduceTasks=1输出一个文件即可

2.5.3 第二部分：LabelAnalyse LPA社区聚类迭代

第二个阶段，读入第一部分处理好的数据，分别发送邻接表(节点，邻接表)和每一条边（即(节点的邻居人名，节点标签:二者所连边的权重)），之后利用相同key将同一个节点和不同标签的边分在同一个reducer的同一个组里面，之后选择出现权重和最大的标签作为新的标签即可。之后累次循环迭代直到标签稳定。

同时，LPA当遇到二分图的时候，会出现标签震荡，这里当迭代轮数超过 10 轮后就会开始震荡，因此我们这里可以选择迭代 15 -20轮。

伪代码：

Input: 人物关系网络权重图
Output: 每个人物的标签和人物关系网络权重图

```
1: function MAP(nid n, node N)
2:     emit(nid n, N.ADJACENCYLIST)
3:     for each nodeid m in N.ADJACENCYLIST do
4:         emit(m,N.label and weight(n→m))
5:     end for
6: end function
7:
8: function REDUCE(nid m, [p1, p2, ...])
9:     M ← ∅
10:    H ← HASHMAP
11:    for each s in [p1, p2, ...] do
12:        if ISNODE(s) then
13:            M ← s
14:        else
15:            if H.CONTAINS(s) then
16:                Update (s.label, s.oldWeight + s.newWeight)
17:            else
18:                Insert (s.label, s.weight)
19:            end if
20:        end if
21:    end for
22:    M.label ← The key which has the maximum value in H
23:    emit(nid m, node M)
24: end function
```

图 16. LabelAnalyse伪代码

具体程序：

1. Mapper类： 首先读入第一部初始化好的数据，首先发送(原节点人名，#邻接表)用于维护每一个节点的网络结构，以便下一次迭代使用

然后拆开邻接表中每一个邻居节点，将邻居节点姓名作为key，将该节点标签和权值作为value，发送(节点的邻居人名，节点标签:二者所连边的权重)

```java
class LabelAnaMapper extends Mapper<Text, Text, Text, Text> {
    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();//名字
        String list = value.toString();//value （标签+邻接表）
        String label = list.split("#")[0];//标签
        String relationlist = "#"+list.split("#")[1];//#+邻接表
        context.write(new Text(name), new Text(relationlist));//发送邻接表维护网络结构
        String[] slist = relationlist.split("\\[|\\]")[1].split("\\|");//将每个关系分离

        for (String s : slist) {
            String target = s.split(",")[0];//和名字主人产生影响的人名
            String weight = s.split(",")[1];//权值
            String LabelWeight=label+":"+weight;
            context.write(new Text(target), new Text(LabelWeight));
        }
    }
```

图 17. LabelAnalyse Mapper

1. Reducer类： 当读入 (人名, #邻接表)时，直接保存邻接表维护网络结构，以便下一次迭代使用。

当读入（节点的邻居人名，节点标签:二者所连边的权重）时，首先创建一个HashMap 数据结构，维护每一个（标签，权重），循环读入所有的这种输入。如果收到的该标签还没有出现过，那么以标签为key，权值为value插入哈希表中。如果标签已经出现过，则标签权值在原来的权值基础上累加上去。

然后循环比较，取权值和最大的标签作为新的标签，还是以原来的格式(节点名称，标签#邻接表) 输出作为结果

```java
class LabelAnaReducer extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
            InterruptedException {
        HashMap<String, Double> neighborMap = new HashMap<>();
        String list = "";
        for (Text value : values) {
            String tempstr = value.toString();
            if (tempstr.startsWith("#")) {
                list = tempstr.substring(1);//从#后面截取整个邻接表
            } else {
                String l = tempstr.split(":")[0];//标签
                Double w = Double.parseDouble(tempstr.split(":")[1]);//对应权值
                if (!neighborMap.containsKey(l)) {//如果哈希表中没有这个标签
                    neighborMap.put(l, w);//将标签，权值加入哈希表中
                } else {
                    Double m = neighborMap.get(l) + w;//将权值加上
                    neighborMap.put(l, m);
                }
            }
        }
        String label = "";
        Double maxWeight = 0.0;
        for (Map.Entry<String, Double> entry : neighborMap.entrySet()) {
            String keylable = entry.getKey();
            Double veight = entry.getValue();
            if (veight > maxWeight) {//循环比较找到权值最大的标签
                maxWeight = veight;
                label = keylable;
            }
        }
        String reslist = label + "#" + list;
        context.write(key, new Text(reslist));

    }
}
```

图 18. LabelAnalyse Reducer

1. Main函数： 设置一个循环即可，循环20次，每次的输出变成下一次的输入

```java
public class LabelAnalyse {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("PageRank neet 2 paras as input and output");
            System.exit(2);
        }
        String input = otherArgs[0];                // 输入目录
        String output = new String();
        int maxIteration = 15;
        for (Integer i = 0; i < maxIteration; i++) {
            if (i == maxIteration-1) {
                output = otherArgs[1] + "final";    // 最终输出
            } else {
                output = otherArgs[1] + i.toString();    // 中间文件输出目录
            }
            Job job = Job.getInstance(conf, "LabelAnalyse");
            job.setJarByClass(LabelAnalyse.class);
            job.setMapperClass(LabelAnaMapper.class);
            job.setReducerClass(LabelAnaReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            job.setInputFormatClass(KeyValueTextInputFormat.class);// 以/t切割key与value
            job.setNumReduceTasks(5);//5个reducer
            FileInputFormat.addInputPath(job, new Path(input));
            Path path = new Path(output);
            FileSystem fileSystem = path.getFileSystem(conf);// 根据path找到这个文件
            if (fileSystem.exists(path)) {
                fileSystem.delete(path, true);// 设置true
            }
            FileOutputFormat.setOutputPath(job, new Path(output));
            job.waitForCompletion(true);
            input = output;
        }
    }
}
```

图 19. LabelAnalyse Main函数

2.5.4 第三部分：LabelResult 结果整理（包括任务六的一部分）

这里因为第二部分输出的结果中还有邻接表的存在，而且结果比较杂乱，所以直接将任务六数据整理整合到这一部分。在mapper中读入(节点名，标签#邻接表)，去除邻接表，并且将标签和节点名反转，将标签设置成key，输出(标签，节点名)以便之后归类

其中重写了partition类，将首字母拼音在a-h，h-p，p-z分为三个区域，分三个文件存储，这个功能可加可不加，如果不加的话那么就输出一个文件，文件中相同标签会在一起，比较直观。

reducer部分直接输出((标签,人名),空)

```java
class LabelResultMapper extends Mapper<Text, Text, Text, Text> {
    // private Text resKey = new Text(); // 要传入reduce的key和value
    private Text rKey = new Text();
    private Text rValue = new Text();

    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        String lable = value.toString().split("#")[0];// 去除后面的关系表，只保留前面的标签
        lable = lable.trim();// 去掉前后空白符
        rKey.set(lable);// 将一个标签的人输入到一起
        rValue.set(name);
        context.write(rKey, rValue);
    }
}
```

(a) LabelResult Mapper

```java
class LabelResultReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for (Text name : values) {
            context.write(new Text(name.toString()+","+key.toString()),new Text(""));//中间加一个逗号
        }
    }
}
```

(b) LabelResult Reducer

```java
class ThreePartition extends Partitioner<Text,Text>{
    @Override
    public int getPartition(Text value1, Text value2, int i) {
        String part1="好"://首字母h
        String part2="跑"://首字母p
        Collator collator = Collator.getInstance(Locale.CHINA);
        CollationKey key1 = collator.getCollationKey(part1);
        CollationKey key2 = collator.getCollationKey(part2);
        CollationKey key = collator.getCollationKey(value1.toString());
        if(key.compareTo(key1)<=0){//首字母比较，分三个区
            return 0;
        }else if(key.compareTo(key2)<=0){
            return 1;
        }
        return 2;
    }
}
```

(c) LabelResult Partitioner

图 20. LabelResult 部分代码截图

2.5.5 总结：

LPA算法优点： LPA算法的最大的优点就是算法的逻辑非常简单，相对于优化模块度算法的过程是非常快的，不用pylouvain那样的多次迭代优化过程。

LPA算法利用自身的网络的结构指导标签传播，这个过程是无需任何的任何的优化函数，而且算法初始化之前是不需要知道社区的个数的，随着算法迭代最后可以自己知道最终有多少个社区利用MapReduce可以很好的利用其特性来进行迭代分析

LPA算法缺点： 划分结果不稳定，随机性强是这个算法致命的缺点。具体体现在：

1. 更新顺序： 节点标签更新顺序随机，但是很明显，越重要的节点越早更新会加速收敛过程；

2. 随机选择： 如果一个节点的出现次数最大的邻居标签不止一个时，随机选择一个标签作为自己标签。这种随机性可能会带来一个雪崩效应，即刚开始一个小小的聚类错误会不断被放大。不过如果相似邻居节点出现多个，可能是权重weight计算的逻辑有问题，需要回过头去优化weight抽象和计算逻辑；

2.6 任务六：基于PageRank的可视化

2.6.1 设计思路

金庸的全部武侠小说中总共包含人物一千余名，其存在的相互关联可能达到100K的等级，在这样庞杂的人物关系中，想要让全部的人物和关系呈现在受众面前，是不现实的。这一判断得到了很多前车之鉴的证实，例如使用gephi软件生成的关系图，如同一团乱麻，几乎看不清任何的姓名或关系，更枉谈“得到一些有趣的结论”。

在这样的前提下，我们选择了对可视化的内容进行取舍，将可视化的人物数量级从1000降低到了100的量级。同时对人物的重要性、人物关系的亲密度以PageRank及其排序的结果进行刻画，从而得到了比较好的视觉效果。

为了进行个性化的开发，我们选择了Qt作为可视化部分的开发平台，使用C++作为开发语言，编写了一个RelationPainter的程序，对数据进行个性化的、延拓性强的可视化操作。利用该程序，我们可以

- 通过文件装载按钮，使用txt文件作为输入，直接获得效果图；
- 通过关系曲线的粗细更直观地表示了人物关系的亲密程度；
- 通过人物姓名和标识点的尺寸，可以更直观地感受人物的重要性；
- 通过关系曲线的颜色，更清晰地辨别同一个人所具有的人物关系；
- 通过关系曲线的高亮，更快地辨别某一特定个体的关系网；
- 通过操控“显示最重要的n位人物”拖动条，控制显示在视野中的人数；
- 通过操控“调整显示尺寸”拖动条，控制人物关系显示的疏密程度，从而避免人物的重叠；
- 通过将人物名称和人物标识点围成圆环状，避免了姓名与人物关系线条的重叠。

该程序的上限很高，延拓性很广，我们将在后面相关章节中详细介绍其改进和优化思路。

```
// 这是一个刻画单行人物关系的类
class Relation
{
public:
    Relation(QString n, double r);
    // 添加新的关系
    void addNewRe(QString n, double w);

    //获取函数及简单的参数判断函数
    QString getName(){return name;}
    double getRank(){return rank;}
    QList<QPair<QString,double>*> getReList(){return reList;}

    int getListLen(){return reList.length();}
    bool isEmpty(){return reList.isEmpty();}
private:
    QString name; // 人物姓名
    double rank;// 人物的pageRank
    QList<QPair<QString,double>*> reList; // 与其他人的关系表
};
```

图 21. 单人物关系类具体实现

2.6.2 程序分析

Relation 类：

这是一个刻画单个人物及与之相关的人物关系的类。其中，通过QList¡QPair¡QString,double¿¿维护了一个人物关系表，从而可以对于该人物相关的人物关系进行操作。如图21

sigFig 类：

这是一个以单人物关系为基础的单图元类。在这个类中，维护了一个Relation类对象以存储人物关系。同时根据其中图元绘制参数常量的值，在绘制之前更新其图元的各项属性（如粗细、颜色、大小、角度、位置、透明度等），之后通过其中的绘制函数分别绘制人物标志点、人物名和人物关系曲线。一般来说，人物的PageRank值越高，人物名和标志点绘制得就越大越粗，而人物关系的权重越高，人物关系曲线就绘制得越粗。

特别的，由于我们将人物标志点和人物名绘制成圆盘状以避免重叠，而每个人物又在圆盘上占据不同的角度值，因此需要给每个人物图元在绘制前计算其在圆盘所占的角度。又因为每个单图元类没有其他图元的角度信息，因此在绘制关系曲线的时候，需要在更高层的类计算好角度对应表，并传入关系曲线的绘制函数。

由于类中大部分的方法为更新参数、获取或设置变量的方法，故不做展示，只集中展示genAll（更新所有参数），paintDot（绘制标志点），和paintLine（绘制标志曲线）三个函数的实现。

在图22中，我们看到，在对r（圆盘半径），midX、midY（屏幕中心X、y），mxRk（最大的PageRank值）设置过后，分别对CirWid（标志点粗细）、TxtWid（人物名粗细）、 DotR

```
// 汇集所有的生成函数
void sigFig::genAll(int r, int txtOff, double lasDeg, int lasR, double mxRk, int midX, int midY)
{
    setRXY(r,midX,midY);
    setMaxRank(mxRk);
    genCirWid();
    genTxtWid();
    genDotR();
    genDeg(r, lasDeg, lasR);
    genDotXY(r, midX, midY);
    genTxtXY(r + txtOff, midX, midY);
}
```

图 22. 单图元，全参数更新函数具体实现

（标志点大小）、Deg（标志点角度）、DotXY（标志点坐标）、 TxtXY（人物名坐标）进行了更新。 在图23可以看到，在更新了全部的绘制参数后，使用Qpainter、QPen、QFont等变量对画笔进行了设置，将画笔移动一定的角度和坐标，并依此绘制圆和Text文本。 在图24中，首先判断是否需要设置高亮色，并根据之前更新的参数设置画笔，此时注意，QPen的颜色设置内多了一个透明度，由于我们是通过绘制点集的方式绘制的曲线，因此透明度需要设置得足够低才有效果（当透明度低时，可以略去很多无效的曲线关系信息）。特别注意到，传入的degList是一个其他图元的角度表，通过对此表的查询，才能正确获得关系曲线的目标点位置，从而使用我们自己实现的简单贝塞尔曲线正确生成关系曲线。同时注意到，图元本身角度和目标点角度的比较，可以确认目标点与本身的排名先后，与限制位的比较，可以判断本身及目标点是否处于需要绘制的点集内，从而判断是否需要绘制曲线。

sigFigList 类：

作为一个统筹所有单图元的类，其主要的行为是对单图元的参数在宏观上进行调控，例如控制显示个数、疏密程度、分发颜色、更新角度表等操作。如图25

mainwindow 类：

对画布上产生的各类信号给出对应的槽进行处理，具体的任务交给 fgLs去做。如载入文件、设置显示个数以及疏密程度的滚动条、鼠标移动设置高亮的判断等。如图26

2.6.3 结果展示

如图27打开程序，导入文件并调整滚动条至合适位置，可以看到最重要的n位人物及其关系列表出现在屏幕中央。几条最粗的曲线彰显了郭靖与黄蓉、杨过与小龙女、胡斐与程灵素、慕容复与王语嫣、张无忌与周芷若等等人物之间的密切关系。通过不同的颜色，可以相对容易地查找与同一人物相关的人物关系。

必要时，可以调整人物关系颜色透明度，并拖动鼠标更改高亮人物，从而获取更高质量的信息。 如图28，通过调整透明度与高亮人物，可以清晰地看到杨过的人物关系。

3 优化与改进

3.1 任务二

- 数据格式中很多地方使用Text传输。可以考虑使用自定义Writable数据格式，来减少序列化时间和传输损耗。

```
// 画点和人名
void sigFig::paintDot(QMainWindow *q)
{
    QPainter pt(q);
//    QPen pen(Qt::darkBlue);

    if (highlight) lineColor = hlColor;
    QPen pen(lineColor);
//    if (highlight) pen.setColor(hlColor);

// 根据计算好的位置和角度，旋转画笔并绘图即可——此时的颜色不需要设置透明度

    pt.save();
    pt.translate(xDot, yDot);
    pen.setWidth(cirWid);
    pt.setPen(pen);
    pt.rotate(getDeg180()-45);
    pt.drawEllipse(0,0,dotR,dotR);
    pt.restore();

    pt.save();
    pt.translate(xTxt, yTxt);
    pt.rotate(getDeg180());
//    pen.setWidth(txtWid);
    QFont font("黑体", txtWid, QFont::Bold, false);
    pt.setFont(font);
    pt.setPen(pen);
    pt.drawText(0,0,rel->getName());
    pt.restore();
}
```

图 23. 单图元，绘制标志点函数具体实现

```
// 绘制曲线的函数
// 需要先访问map的角度字典，然后找到目标点的位置
// 以起点、圆心和目标点为基点，下压参数作为调整值，绘制贝塞尔曲线
// 此时的绘制时需要设置透明度，不然结果就没法看了
void sigFig::paintLine(QMainWindow *q, std::map<QString, double> degList, QString limitName)
{
    QPainter pt(q);
//    QPen pen(QColor(32,32,160,lineTransp));
//    QPen pen(lineColor, lineTransp);
    if (highlight) lineColor = hlColor;
    QPen pen(QColor(lineColor.red(),lineColor.green(), lineColor.blue(), lineTransp));
    for (QPair<QString,double>* qp: rel->getReList())
    {
        auto iter = degList.find(qp->first);
//        int limitPos =  min(degList.size(),vitalNum) - 1;
        double limitDeg = degList.find(limitName)->second;
        if (iter != degList.end() && iter->second > deg && iter->second <= limitDeg)
        {
            QPair<int,int> resXY = calDotXY(iter->second);
            QPoint a(xDot,yDot);
            QPoint b(xMid,yMid);
            QPoint c(resXY.first,resXY.second);
            b = (1-lowSize)*(a+c)/2.0 + lowSize*b;
            QList<QPoint> pts = genBesLine(a,b,c);

            pt.save();
            int lineWid = (baseLineWid + maxLineOff * qp->second/standardWei);
            pen.setWidth(lineWid);
            pt.setPen(pen);
            for(QPoint qpt:pts)
            {
                pt.drawPoint(qpt);
            }
            pt.restore();
        }
    }

}
```

图 24. 单图元，绘制关系曲线函数具体实现

```
// 原本想搞个随机化，想了想觉得没必要
// 向每个sigFig分发颜色
void sigFigList::genColor()
{
    for(int i = 0; i< sigLs.length();++i)
    {
        sigLs[i]->setLineColor(allColor[i%allColor.length()]);
    }
}

void sigFigList::genMaxRank()
{
    maxRank = sigLs[0]->getRel()->getRank();
}

// 这里如果不清空则会造成目标点无法调整的情况，因为map的性质使然
void sigFigList::genDegList()
{
//    if (!sigLs.isEmpty())
//        sigLs.clear();
    degList.clear();
    for(sigFig * ss: sigLs)
    {
        degList.insert(std::map<QString, double>::value_type(ss->getRel()->getName(), ss->getDeg()));
    }
}

// 必须依次访问sigFig成员并计算角度等参数，才能正常地全部更新
void sigFigList::genAll()
{
    genColor();
    double curDeg = 0;
    int curDotR = 0;

    genMaxRank();
    for(sigFig * s: sigLs)
    {
        s->setSigSize(sigSize);
        s->genAll(cirR, txtOff, curDeg, curDotR, maxRank, midX, midY);
        curDeg = s->getDeg();
        curDotR = s->getDotR();
    }
    genDegList();
}
```

图 25. 分发颜色、更新角度列表、更新疏密程度具体实现

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    void loadFile(); // 装入文件
//    void paintRe();

protected:

void paintEvent(QPaintEvent *); // 绘制图像

void mouseMoveEvent(QMouseEvent *event); // 捕捉鼠标事件

private slots:

    void on_FileBrowse_clicked();// 对装载文件的按钮的处理

    void on_vitalNumSlider_valueChanged(int value);// 对显示数量滚动条变化的处理

    void on_sigSizeSlider_valueChanged(int value);// 对疏密程度滚动条变化的处理

    void myMouseMoveHandler(QMouseEvent * e);// 对鼠标点击事件的处理

signals:
    void mouseMove(QMouseEvent *event);// 发射的鼠标信号

private:
    Ui::MainWindow *ui;
//    QList<Relation*> reLs;
    sigFigList fgLs;// 包含一个sigFigList类的成员，以对关系和显示进行处理
    bool loaded;// 已经装载文件的信号，应当放入sigFigList更好

    const int hlDistance = 10;//控制高亮显示的检测半径，也即灵敏度

};
```

图 **26.** mainwindow具体实现


韦小宝
令狐冲
张无忌
郭靖
黄蓉
杨过
段誉
汉子
袁承志
胡斐
陈家洛
小龙女
吴三桂
石破天
赵敏
岳不群
大汉
骆冰
徐天宏
文泰来
杨逍
狄云
周芷若
慕容复
王语嫣
虚竹
黄药师
余鱼同
林平之
周绮
岳灵珊
赵半山
陆菲青
张翠山
阿朱
丘处机
说不得
灭绝师太
方怡
福康安

图 27. 可视化结果

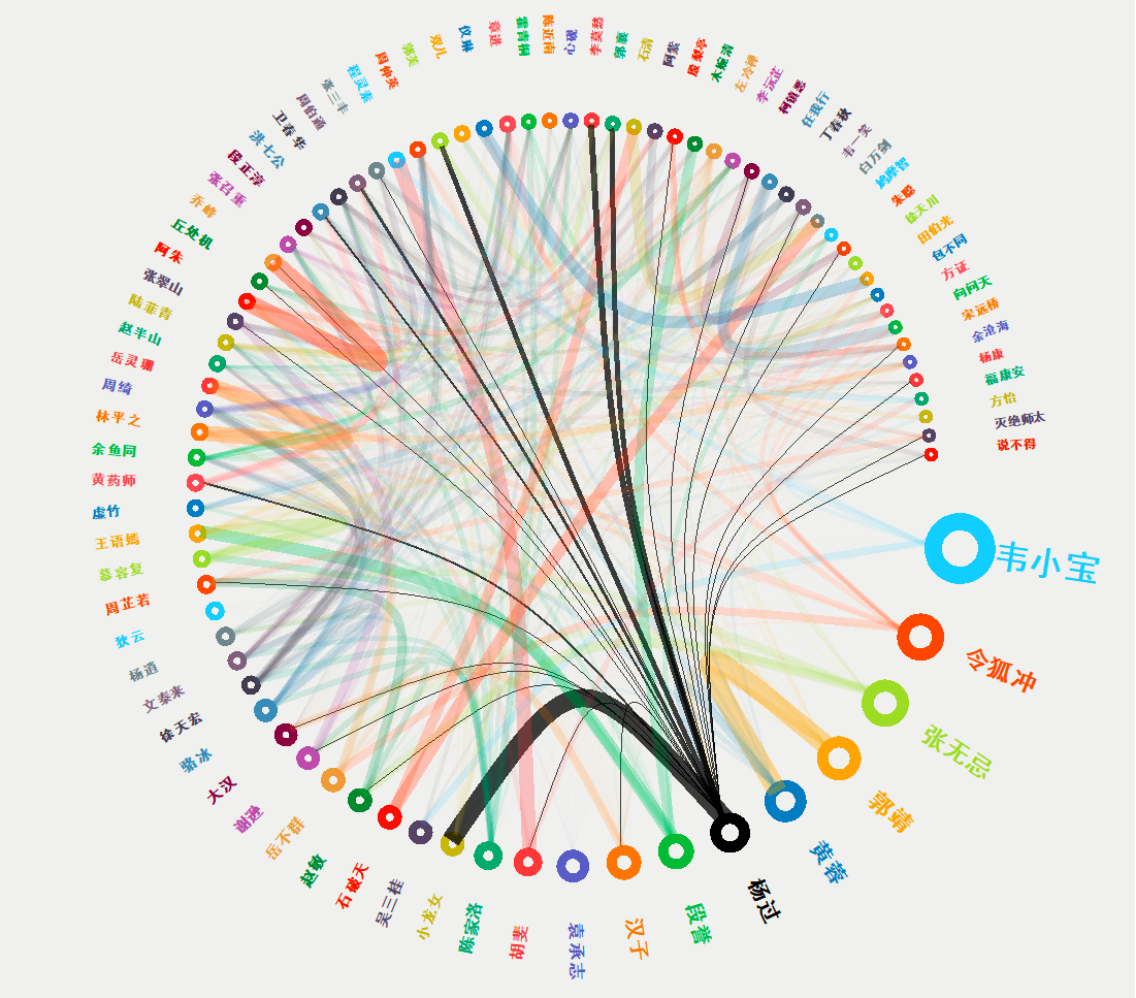

图 28. 透明度调整与高亮效果

- 统计人物同现时，可以考虑根据人物字母顺序，将¡a,b¿与¡b,a¿统一为¡a,b¿。这样任务二mapper和reducer之间减少一倍的传输损耗，任务二的输出结果将减少一倍占用空间，任务三mapper读、任务二reducer写也将减少一倍读写文件时间损耗。实现时可以将任务二mapper中HashSet用TreeSet代替来保证有序，在内部for循环中只发射在外部for循环当前值之后（或之前）的部分。当然任务三中mapper就要修改为：既发射第一个人名又发射第二个人名。之后保持不变。当然也要考虑到TreeSet比HashSet可能要消耗更多时间。

3.2 任务三

reducer中HashMap的使用是没有必要的。应该使用类似List¡Pair¡String, Integer¿¿代替，加快reducer端时间、空间效率。

3.3 任务四

1) 原本迭代次数为20次，在检查中间结果时发现在14次之后，结果变化不大，基本收敛，因此将迭代次数改为15次，节省时间开销。

2) 完整版的PageRank计算会引入随机跳转，避免自环节点以及无出度节点导致无法收敛。而本次数据实际不存在此类节点，因此无需引入，节省计算量。

3) PageRank算法原本RP值设置为访问概率1/N，实际上若PR值收敛，则结果与初始值无关。因此设置初始值为1.0以减小小数运算误差。

3.4 任务五

LPA当遇到二分图的时候，会出现标签震荡，这里当迭代轮数超过 10 轮后就会开始震荡，因此我们这里可以选择迭代 15 -20轮。通过对实验五的结果进行分析，我们可以看出，相同标签的人物都是在同一本书中的人物。但因为人物同现关系窗口选择的大小问题，会出现一些很小的簇或者孤岛

3.5 任务六

1) 【已经优化】由于QPainter的绘制特性缘故，因此人物的姓名有一半是倒着的，造成观感不佳，需要解决。

2) 【已经优化】由于曲线使用点集绘制，而单点的绘制被自动设置为方形，因此曲线的形状不佳，需要改进。

3) 【已经优化】由于曲线使用的颜色的亮度、深浅不同，造成人物关系密切程度的直观性降低，需要改进颜色组。

4) 采用数据结构效率较低，对于冗余的判断和循环没有进行优化，造成操作上的延迟，需要优化。

5) 由于采用的是高Rank值的人物向低Rank值的人物绘制曲线，因此造成低rank值人物快速无法快速辨认全部与其相关的人物关系。可以采用颜色的渐变进行优化。

6) 尚存在很多绘制参数没有在外部留出接口，例如透明度无法在用户界面设置，需要改进。