课程设计2—人物关系挖掘

> 小组成员：张涵之@191220154 林芳麒@191220057
>
> 日期：2022.7.4

任务1：数据预处理

- 从原始的西游记小说的文本中，抽取出与人物互动相关的数据。 需要屏蔽与人物关系无关的文本内容，为后面的基于人物共现的分析做准备。
- 数据输入：西游记系列小说文集（未分词）；西游记系列小说中的人名列表。
- 数据输出：分词后保留人名。

主要流程

Mapper:

1. 在`setup`中读入`xiyouji_name_list.txt`，存入`ArrayList`备用。
2. 默认的`LineRecordReader`每次读取文件的一行（以'\n'划分），即一个“段落”，调用第三方JAR包对这一行（一段）进行分词，分词结果逐个与`ArrayList`中人名进行比对，匹配成功的保留。

本阶段不需要Reducer。

主要算法

使用第三方的 JAR 包结巴分词器`jieba`来辅助分析（进行分词）

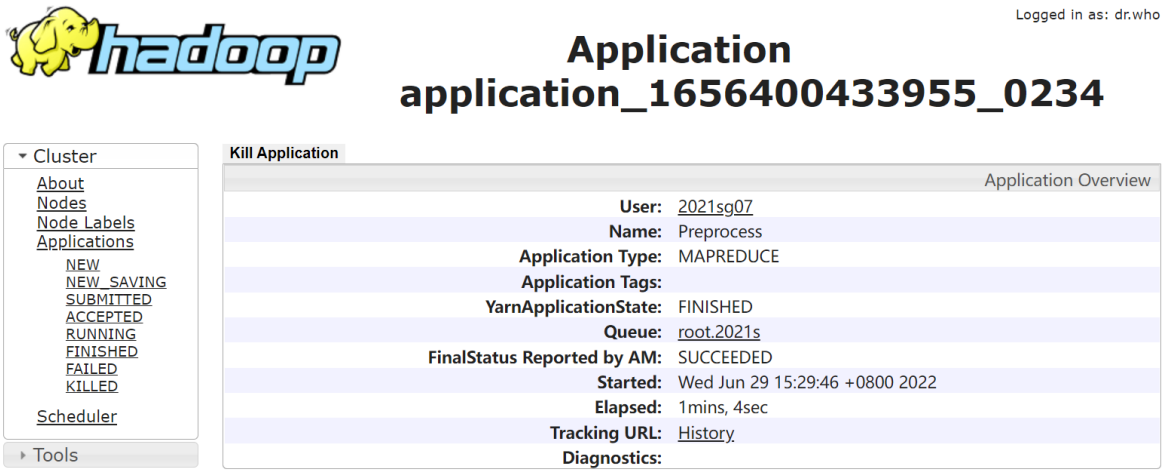
Mapper:

```
public class PreprocessMapper extends Mapper<>
{
    ArrayList<String> nameList = new ArrayList<>(); // 人物（角色）名单

    protected void setup(Context context)
            throws IOException {
        // 从西游记系列小说中的人名列表读取人名存入nameList
        ......
        Text line = new Text();
        while (lineReader.readLine(line) > 0) {
            String name = line.toString();
            nameList.add(name);
        }
    }

    protected void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        // 从文件中读取一行（一段）内容
        String line = value.toString();
        String names = "";
        //  使用JiebaSegmenter进行分词
        JiebaSegmenter segmenter = new JiebaSegmenter();
```

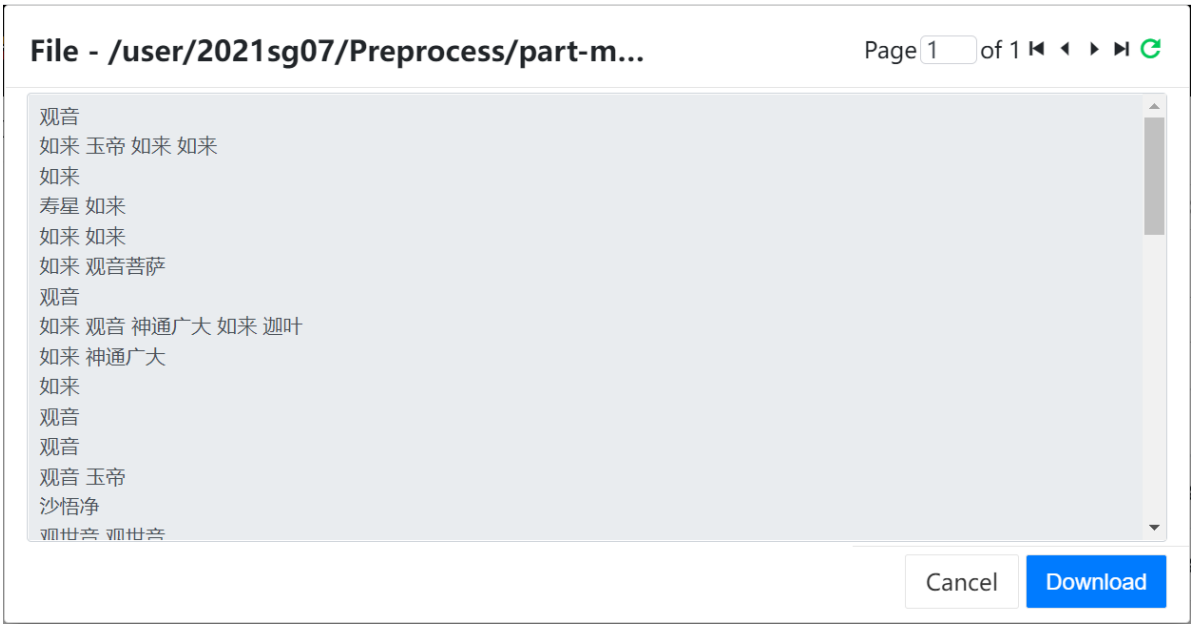
```
        List<String> segments = segmenter.sentenceProcess(line);
        for (String segment : segments) {
            // 逐个与nameList中人名进行比对，匹配的加入结果
            if (nameList.contains(segment))
                names += (segment + " ");
        }
        if (names.length() > 0) {
            // 去掉结果一行末尾多于的空格，写文件
            Text text = new Text(names.trim());
            context.write(text, NullWritable.get());
        }
    }
}
```

程序运行截图

```
hadoop jar Preprocess.jar /data/2022s/relation/xiyouji
/data/2022s/relation/xiyouji_name_list.txt /user/2021sg07/Preprocess
```



对于输入数据中的每个文件（章节）单独生成一个输出文件，例如：



任务2：人物同现统计

- 在人物同现分析中， 如果两个人在原文的同一段落中出现，则认为两个人发生了一次同现关系。我们需要对人物之间的同现关系次数进行统计， 同现关系次数越多，说明两人之间的关系越密切。
- 数据输入：任务1的输出
- 数据输出：人物之间的同现次数

主要流程

Mapper:

1. 根据任务1的输出，遍历一个段落的人名，判断某个人名是否是主要人物别名。若是，替换成主要人物名后再加入`HashSet`集合；若不是直接加入`HashSet`集合。由于`HashSet`数据结构的特性，重复元素不会被重复加入，从而起到了一个去重的作用。
2. 对`HashSet`里的元素两两配对，输出`<(人名1，人名2)，1>`

Reducer:

对于人物的同现关系次数进行统计、累加，最后输出`<(人名1，人名2)，n>`，`n`为同现次数。

主要算法

单词同现算法

Mapper:

```
class Mapper
{
    map(LongWritable key, Text value, Context context) {
        HashSet<String> roles = new HashSet<>();    // 该段落的人名集合
        // 1.统计该段人名，去重并替换别名
        for name in 段落 {
            if name is 主要人物别名 {
                name => 主要人物 character_1;
                roles.add(character_1);
            }
            else
                roles.add(name);
        }
        for (String c1: roles) {
            for(String c2: roles){
                if(c1.equals(c2))
                    continue;
                context.write<(c1,c2),1>;   // 2.两两配对,输出
            }
        }
    }
}
```

Reducer:

```
class Reducer
{
    reduce(key,values) {
        int sum = 0;
        //累加人物同现次数
        for (IntWritable it: values) {
            sum += it.get();
        }
        context.write(key,new IntWritable(sum));
    }
}
```

优化工作

在单词同现算法算法的基础上增加`Combiner`对同现人物对同现次数进行累加。

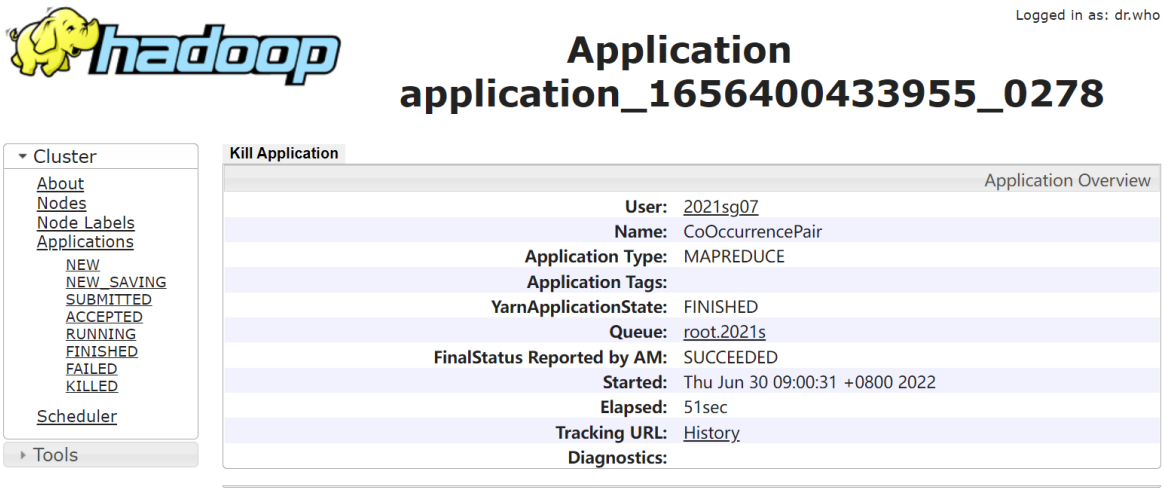
遇到问题

> 在`MapReduce`编程中发现Reduce中的迭代器只能使用一次，第二次使用迭代出的数据即为空。

解决方法：在第一次迭代中就把要迭代的数据保存到一个容器中（即`HashSet`中），之后遍历该容器，就可以实现在Reduce中多次遍历。

程序运行截图

```
hadoop jar Cooccurance.jar /user/2021sg07/Preprocess /user/2021sg07/Cooccurance
```



任务3：人物关系图构建与特征归一化

- 数据输入：任务2的输出。
- 数据输出：人物关系图。
- 注意：为了使后面的分析方便，需要对共现次数进行归一化处理：将共现次数转换为共现概率。

主要流程

Mapper:

1. 处理任务2输出的数据，划分字符串，获取同现人物对信息。
2. 根据map的输出结构`<name1,(name2, times)>`发送，其中，`name1`和`name2`是一对同现人物，`times`是同现次数。

Reducer:

1. 输入`Key`类型为`String`(人物1)，输入`Value`类型为`(String,Int)`（以`key`为分组的的`ValuePair`集合）。
2. 遍历统计与key同现的人物总次数`count`和人物数`sum`
3. 二次遍历与key同现的人物，`同现次数/sum`
4. 输出`<key, ([name1 , value1], [name2 , value2], ....., [name_n , value_n])>`

主要算法

数据归一算法

Mapper:

```
class NormalizationMapper extends Mapper<>
{
    void map() {
        // 1.获取一行数据
        String line = value.toString();
        String[] splits = line.split("\t");

        // 2.处理输入数据，分割字符串
```

```
        int times = Integer.parseInt(splits[1]);
        String[] names = splits[0].split("<|, |>");
        String key_name = names[1];
        String value_name = names[2];

         // 3.输出，key_name和value_name是一对同现人物，value_times是同现次数
        context.write<key_name, (value_name,value_times)>;
    }
}
```

Reducer:

```
class NormalizationReducer extends Reducer<>
{
    protected void reduce(key, values, context) {
        // 1.统计与key同现的人物总次数和人物数
        for (ValuePair it: values) {
            sum += it.getValue();//所有边权和
            ++num;//共现人物数量
        }

        // 2.遍历与key同现人物，归一化处理
        for (ValuePair it :values) {
            pair <= {it.getName() , it/sum}
        }

        // 3.输出
        context.write(key,([name1 , value1],[name2 , value2],.....,[name_n ,
value_n]);
    }
}
```

程序运行截图

```
 hadoop jar Normalization.jar /user/2021sg07/Cooccurance
/user/2021sg07/Normalization
```

Logged in as: dr.who

Application application_1656400433955_0279

- Cluster
  - About
  - Nodes
  - Node Labels
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Kill Application

Application Overview

User:	2021sg07
Name:	Normalization
Application Type:	MAPREDUCE
Application Tags:	
YarnApplicationState:	FINISHED
Queue:	root.2021s
FinalStatus Reported by AM:	SUCCEEDED
Started:	Thu Jun 30 09:03:18 +0800 2022
Elapsed:	17sec
Tracking URL:	History
Diagnostics:	

任务4：基于人物关系图的`PageRank`计算

- 计算`PageRank`，定量分析小说的主角。
- 数据输入：任务3的输出
- 数据输出：各人物的`PageRank`值
- 注意：该任务默认的输出是杂乱的，从中无法直接得到分析结论。需要对`PageRank`值进行全局排序，确定`PageRank`值最高的任务。

主要流程和算法

后来经过验收，我们知道了不需要使用随即浏览模型，但.....

使用`PageRank`的随机浏览模型，其中`PageRank`公式为：

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

通过迭代计算得到所有节点的`PageRank`值。

此处`d`取0.85，`N`取任务3输出行数（人数），由于任务3已经得到每个人物与“邻居”们的同现频率，则公式右边第二项应使用加权平均而非算术平均，修改后的公式为：

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} PR(p_j) \times Cooccurance(p_i, p_j)$$

此处的`Coocurance(pi, pj)`即对应为任务3输出中`<人名, [(邻居1, 同现频率1), (邻居2, 同现频率2), ...]>`的`同现频率`，其中`pi`对应`人名`，`pj`对应`邻居`。

Phase 1: Graph Builder

原始数据集：任务3的输出。每行包含一个人名、与之同现的所有人物和相应的同现频率。

`GraphBuilder`目标：各人物之间的同现关系任务3已完成，故此处只需加入`PageRank`初始值即可。

Mapper:

在每个`<key, value>`对之间插入`PageRank`初始值`PR_init`，取值为`1/N`，在`PageRankDriver`中作为参数输入`GraphBuilder`。输出形如`<人名, PR, [[(邻居1, 同现频率1), ...]>`。

```
public class GraphBuilderMapper extends Mapper<>
{
    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {
        Configuration config = context.getConfiguration();
        // PR_init从外界传入
        String pageRank = config.get("PR_init") + "\t";
        String[] tmpStr = value.toString().split("\t");
        Text name = new Text(tmpStr[0]);
        pageRank += tmpStr[1];
        Text pair = new Text(pageRank);
        context.write(name, pair);
    }
}
```

本阶段不需要Reducer。

Phase 2: Page Rank Iterator

自定义数据格式`PageRankBean`:

```
public class PageRankBean implements Writable
{
    private Hashtable<String, Double> link_list;
    private int list_length;
    private double page_rank;
    private String type;        // 记录是Mapper产生的哪一种键值对
}
```

Mapper:

Map对上阶段的`<人名, PR, [[(邻居1, 同现频率1), ...]>`产生两种键值对：

1. For each `邻居i` in `link_list`，输出 `<邻居i, PR * 同现频率i>` ，其中`邻居i`代表当前一个和人物同现的角色，作为key；`PR * 同现频率i`作为value，即实现上面所说的加权平均。
2. 同时为了完成迭代，传递人物的同现关系以维护图的结构，还要输出`<人名, link_list>`。

```
public class PageRankIterMapper extends Mapper<>
{
    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {
        // 人名 \t PR \t [邻居1，同现频率1|邻居2，同现频率2|...]
        String[] tmpStr = value.toString().split("\t");
        String name = tmpStr[0];    // 人名
        double cur_rank = Double.parseDouble(tmpStr[1]);    // PR
        String str = tmpStr[2];     // link_list
        Hashtable<String, Double> link_list = new Hashtable<>();
        str = str.substring(1, str.length() - 1);
```

```
        String[] pairs = str.split("\\|");
        for (String pair : pairs) {
            String[] split = pair.split(", ");
            String u = split[0];
            double o = Double.parseDouble(split[1]);
            link_list.put(u, o);    // <邻居i, 同现频率i>
            // 输出形如<邻居i, PR * 同现频率i>的键值对
            PageRankBean prb = new PageRankBean();
            prb.setPage_rank(cur_rank * o);
            context.write(new Text(u), prb);
        }
        // 输出形如<人名, link_list>的键值对
        PageRankBean prb = new PageRankBean();
        prb.setLink_list(link_list);
        context.write(new Text(name), prb);
    }
}
```

Combiner:

对Map产生的第一类键值对合并处理，即对同一key对应的`PR * 同现频率`进行累加。

```
public class PageRankIterCombiner extends Reducer<>
{
    protected void reduce(Text key, Iterable<> values, Context context)
        throws IOException, InterruptedException {
        double val = 0;
        for (PageRankBean bean : values) {
            // <邻居, PR * 同现频率i>键值对的value通过累加进行合并
            if (bean.getType().equals("page_rank"))
                val += bean.getPage_rank();
            // <人名, link_list>键值对直接输出
            else if (bean.getType().equals("link_list"))
                context.write(key, bean);
        }
        // 输出合并的结果
        PageRankBean prb = new PageRankBean();
        prb.setPage_rank(val);
        context.write(key, prb);
    }
}
```

Reducer:

Reduce继续合并同一key的`PR * 同现频率`，根据上面修改后的浏览公式，计算当前key对应人物的更新后的PR。此外还要继续维护图的结构，输出形如`<人名, 新的PR, [[(邻居1, 同现频率1), ...]>`。

```
public class PageRankIterReducer extends Reducer<>
{
    // 设置阻尼因子d为0.85
    private static final double d = 0.85;

    protected void reduce(Text key, Iterable<> values, Context context)
        throws IOException, InterruptedException {
        // 用于格式化输出link_list
```

```
        StringBuilder link_list = new StringBuilder();
        Hashtable<String, Double> ll;
        double val = 0;
        Configuration config = context.getConfiguration();
        // 这里的PR_init实际上就是1/N的值
        double pr_init = Double.parseDouble(config.get("PR_init"));
        for (PageRankBean bean : values) {
            // 继续累加合并PR * 同现频率i
            if (bean.getType().equals("page_rank"))
                val += bean.getPage_rank();
            // 获得link_list及其格式化的字符串表示
            else if (bean.getType().equals("link_list")) {
                ll = bean.getLink_list();
                ......
            }
        }
        // 代入公式计算新的PageRank，输出<人名，新PR，link_list>
        // PR_init = 1/N, val = 所有PR(j) * Cooccurance(i, j)的累加
        double page_rank = (1 - d) * pr_init + (d * val);
        context.write(key, new Text(String.valueOf(page_rank) + link_list));
    }
}
```

Phase 3: Page Rank Viewer

将最终结果排序输出。 从最后一次迭代的结果读出人物和其PR，按PR值从大到小的顺序输出。

排序过程利用框架自身的排序处理，使其经过shuffle和sort后反序（从大到小）输出。

Mapper:

读入`<人名，PR，link_list>`，抛弃`link_list`，以PR取负为key，人名为value输出。

```
public class PageRankViewerMapper extends Mapper<>
{
    protected void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String[] tmpStr = value.toString().split("\t");
        String name = tmpStr[0];
        double page_rank = Double.parseDouble(tmpStr[1]);
        // 输出<-PR，人名>
        context.write(new DoubleWritable(-page_rank), new Text(name));
    }
}
```

Partitioner

采用`Hadoop`提供的`TotalOrderPartitioner`。

Reducer:

将key再次取负获得原来的PR作为value，value的人名作为key输出。

```
public class PageRankViewerReducer extends Reducer<>
{
```

```
    DecimalFormat DF = new DecimalFormat("#.0000000000");   // 格式化输出

    reduce(DoubleWritable key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException
    {
        for (Text value : values) {
            // 完成排序后再次取负获得原来的PR，输出<人名, PR>
            Text formated_key = new Text(DF.format(-key.get()));
            context.write(value, formated_key);
        }
    }
}
```

`PageRank` 迭代终止条件

迭代至固定次数（默认10次，用户也可以手动输入）。

多趟 `MapReduce` 的处理

`PageRankDriver` 负责以上三个阶段的调配。

```
public class PageRankDriver
{
    private static int times = 10;

    public static int getNameCnt(String filePath) throws IOException {
        ......  // 读任务3输入文件确定人数N
        int nameCnt = 0;
        for (int i = 0; i < fileStatus.length; i++) {
            inputStream = fileSystem.open(fileStatus[i].getPath());
            lineReader = new LineReader(inputStream, configuration);
            while (lineReader.readLine(line) > 0)
                nameCnt++;
        }
        return nameCnt;
    }

    public static void main(String[] args) throws IOException
    {
        int nameCnt = getNameCnt(args[0]);
        // 设置PR_init = 1/N，通过configuration传入GraphBuilder和PageRankIter
        double pr_init = 1.0 / nameCnt;
        // 用户可选参数，手动输入迭代次数，否则默认迭代10轮
        if (args.length > 2)
            times = Integer.parseInt(args[2]);
        String[] forGB = {"", args[1] + "/Data0"};
        forGB[0] = args[0];
        // 初始化图，在任务3输出同现频率基础上添加初始PR信息
        GraphBuilder.GraphBuilder.main(forGB, pr_init);
        String[] forItr = {"Data" , "Data"};
        // 迭代计算和更新每个人物的PR
        for (int i = 0; i < times; i++) {
```

```
            forItr[0] = args[1] + "/Data" + (i);
            forItr[1] = args[1] + "/Data" + (i + 1);
            PageRankIter.PageRankIter.main(forItr, i + 1, pr_init);
        }
        String[] forRV = {args[1] + "/Data" + times, args[1] + "/FinalRank"};
        // 根据PR从大到小排序和格式化输出
        PageRankViewer.PageRankViewer.main(forRV);
    }
}
```

优化工作

在课件介绍 `PageRank` 算法的基础上增加了Combiner对 `PR * 同现频率` 进行累加。

遇到问题

1. 网页排名图算法 `PageRank` 中的“网页排名图”是严格的有向图，分为出边（链接到其他网页）和入边（有网页链接到它）。且对于一个网页链接到的所有网页，认为它们被用户点击的概率是相等的。这里需要进行一个思维上的类比和转化，因为人物在段落之间的“同现”关系其实可以看作一个无向图，而这个“同现频率”其实可以理解为用户浏览某一网页时，随机点击它提供的各个链接的概率。一旦解决了这个思路问题，实际上就知道了无向图也可以当作有向图来处理，即对于任务3输出结果中的 `<人物，邻居列表>`，每一个 `<人物，邻居>` 对可以看作有向图中的一条出边，也可以确认在计算加权平均的过程中，同现频率应该在map阶段就乘。起初我觉得这有点反直觉，即每个reducer最后收到的 `PR * 同现频率`，这些 `同现频率` 加起来并不一定（实际上绝大多数情况下都不会）等于1。然而，这种“直觉”完全想反了，网页迭代更新PR时，并不是按照每个网站链入链接的数量，每个链入链接为该网站均匀地贡献它们的PR，而是按照每个网站链出链接的数量，即该网站均匀地为每个链出网站贡献它自己的PR。虽然“同现”的关系是双向的，但不妨也这样去理解。
2. 由于自定义数据格式 `PageRankBean` 中定义了一个 `Hashtable` 用于维护link_list，在读写、赋值、传递时常遇到传值和传引用混淆不清的问题，再加上reducer中迭代器values的特殊性质，往往出现难以定位的bug。索性每次set或get这个link_list时，都新建的 `Hashtable`，通过逐个put原 `Hashtable` 中的 `<key，value>` 对来拷贝一个 `Hashtable` 的副本用于赋值。
3. 由于任务三输出是人物同现关系，这里实际上已经没有了“没有外出链接的独立的网页”，即原始名单中没有和其他如何人物同现（每次都一个人一段）的人物已经被剔除了，所以其实不需要考虑rank leak和rank sink问题，因此也就不需要“随机访问”，可以直接使用简化模型。虽然还是按照随机访问模型写了，但这里 `PR_init` =1/N中的人数N取任务三输出的行数而非初始name list的行数其实也是出于这一考虑，即没有同现的人物已经不在了，也不必计算他们的PR。

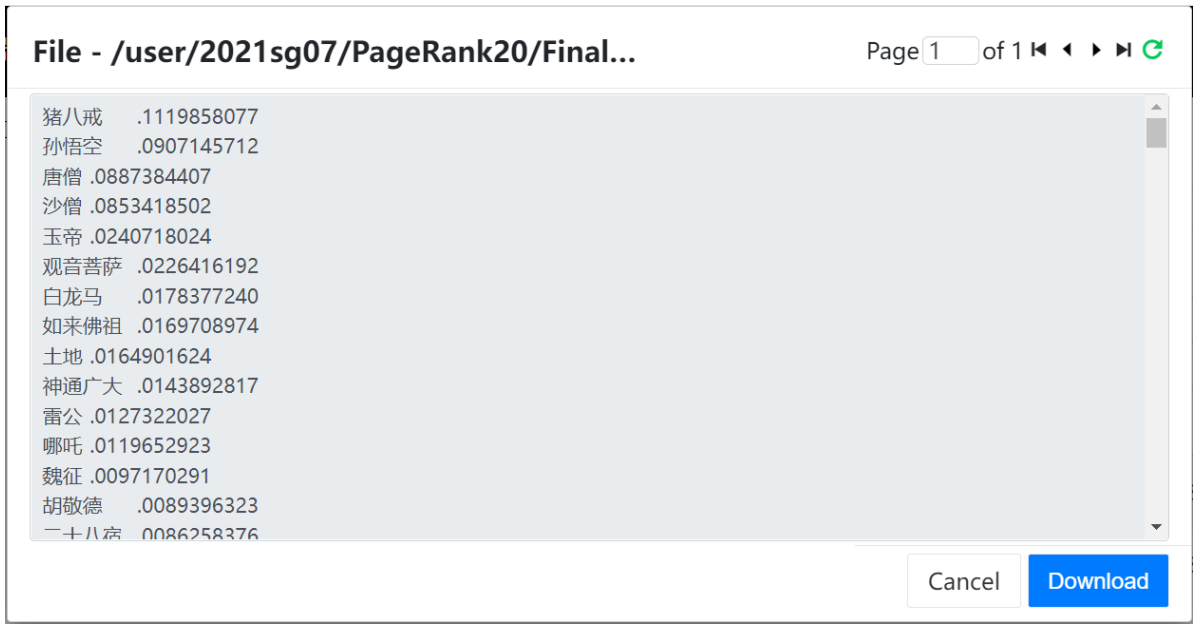
程序运行截图

```
hadoop jar PageRank.jar /user/2021sg07/Normalization /user/2021sg07/PageRank
```

application_1656400433955_0313	2021sg07	PageRankViewer	MAPREDUCE	root.2021s	Thu Jun 30 09:21:10 +0800 2022	Thu Jun 30 09:21:27 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_0312	2021sg07	PageRankIter20	MAPREDUCE	root.2021s	Thu Jun 30 09:20:52 +0800 2022	Thu Jun 30 09:21:08 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_0311	2021sg07	PageRankIter19	MAPREDUCE	root.2021s	Thu Jun 30 09:20:33 +0800 2022	Thu Jun 30 09:20:49 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_0310	2021sg07	PageRankIter18	MAPREDUCE	root.2021s	Thu Jun 30 09:20:15 +0800 2022	Thu Jun 30 09:20:30 +0800 2022	FINISHED	SUCCEEDED

- 迭代10轮获得的最终排名和PR：

- 迭代20轮获得的最终排名和PR：



排名前十乃至前二十的人物不变，则可视为近排名似收敛了。师徒四人是毋庸置疑的“主角”，他们的PR至少是其他人的三到四倍。此外玉帝、观音菩萨、白龙马、如来佛祖等也是出镜率比较高的配角。

任务 5：人物关系图上的标签传播（选做）

- 实现标签传播算法。标签传播是一种半监督图分析算法，通过在图上顶点打标签，进行图顶点的聚类分析，从而在一张社交网络图中完成社区发现。
- 数据输入：任务3的输出
- 数据输出：人物标签信息

主要流程：

根据实验指南中所列举的参考文献：Phys. Rev. E 76, 036106 (2007) - Near linear time algorithm to detect community structures in large-scale networks 中的内容，总结标签传播算法主要流程如下：

Step 1: initialize every node with unique labels

先给N个节点分配独一无二的标签，不妨设为1-N，即节点1对应标签1，节点i对应标签i。

Step 2: let the labels propagate through the network

迭代更新每个节点的标签。在每一轮迭代中，遍历N个节点，对于每个节点所有邻居对应的标签，找到出现次数最多的替换成当前节点的新标签；若出现次数最多的标签不止一个，则随机选择一个替换。

Step 3: the stopping criteria

1. 本轮标签更新后，节点标签与上一轮相比没有变化（即收敛了）；
2. 达到用户手动设定的最大迭代次数；
3. 本轮标签更新后，每个节点在社区内的邻居数量不少于社区外的。

主要算法：

1. Initialize the labels at all nodes in the network. For a given node x, `C(x0) = x`.
2. Set `t = 1`.
3. Arrange the nodes in the network in a random order and set it to X.
4. For each x in X chosen in that specific order, let

$$C_x(t) = f(C_{x_{i1}}(t), \ldots, C_{x_{im}}(t), C_{x_{i(m+1)}}(t-1), \ldots, C_{x_{ik}}(t-1))$$

   f here returns the label occurring with the highest frequency among neighbors.

5. If every node has a label most of its neighbors have, stop. Else, set `t = t + 1` and go to iii.

论文里提出了几个问题，在我们的实验中都有涉及：

标签传播方式：同步更新和异步更新

- synchronous updating 同步更新

  概念：在第t次迭代中，每个节点必须只能依赖邻居节点上一次（第t-1次）迭代获得的社区标签。

  迁移公式：其中`f()`表示取出现频率最高的标签。

$$C_x(t) = f(C_{x_1}(t-1), \ldots, C_{x_k}(t-1))$$

  问题：如果图含有二部图或类似二部图结构的子图，标签将出现震荡（后续将举例说明）。

- asynchronous updating 异步更新

  概念：在第t次迭代中，节点可以依赖邻居节点在本次（第t次）迭代中刚刚更新的社区标签。

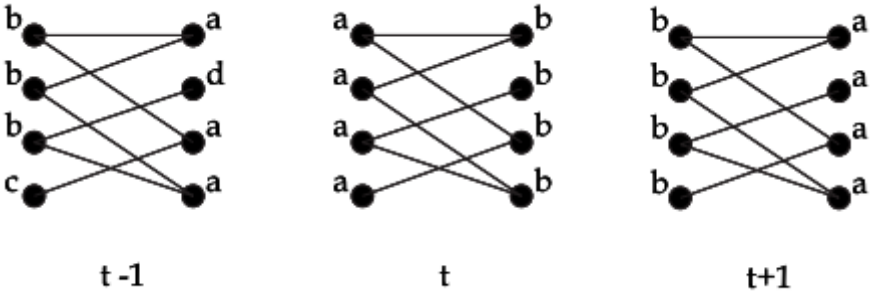
  迁移公式：其中`x_i1`到`x_im`是本轮已经更新标签的邻居，`x_i(m+1)`到`x_ik`是尚未更新的。

$$C_x(t) = f(C_{x_{i1}}(t), \ldots, C_{x_{im}}(t), C_{x_{i(m+1)}}(t-1), \ldots, C_{x_{ik}}(t-1))$$

  注意：每一轮迭代中，节点更新标签的顺序应该是随机的。

同步更新中的二部图震荡问题

- 如果图中含有二部图和近似二部图结构的子图，使用同步更新会出现震荡现象，如下图所示，即两个"部"的标签每迭代一轮就进行一次互换，始终无法合并到同一个标签下，算法也就无法收敛。



- 后面将进一步讨论`MapReduce`并行化和同步更新的适配性，以及如何解决震荡问题。

迭代的终止条件

- 在理想情况下，迭代一定轮数后，各节点标签都将不再变化，即算法收敛。此时剩余标签的数量即是社区的数目。如果算法确实收敛，那么甚至不必每轮迭代完判断一下是否有节点的标签改变了，用户可以输入一个足够大的固定迭代轮数，或者多尝试几个数字，来测试收敛所需的大致轮数。
- 然而，对于某些特定的图来说，一个节点可能在两个或者更多个社区里都有数量相当的邻居。由于我们的算法总是随机选取出现频率最高的标签来替换当前的，就可能造成这样的节点在好几个社区之间跳来跳去，也就是说，算法也可能永远不会收敛。在这种情况下，用户仍然可以输入足够大的固定迭代轮数，从而强行打破死循环，或者改以"每个节点在当前社区内的邻居不少于社区外的邻居"（也即和这个节点同一个标签的邻居不少于其他标签的邻居）作为迭代终止条件。

算法结果的不唯一性

- 由于算法总是随机选取出现频率最高的标签来替换当前的，再加上异步更新顺序的随机性，社区之间的连结总是随机地被打破，随机性在标签传播的迭代过程中不断累积，则同一个图很有会可能得到满足迭代终止条件的不同结果。具体随机性有多大，可以通过多运行几次来观察。

论为什么初始化时要给每个节点分配独一无二的标签

- 这学期学过的其他一些聚类划分算法，如K-Means，会先设置一些初始节点，然后再开始迭代。如果能够预先知道一些比较可能会是社区中心的节点，并且给它们独一无二的标签，其他节点则不给初始标签，会不会比给所有的节点初始化不同的标签更容易减小随机性、缩短运行所需的时间呢？理论上是可以的。然而，在一个图中，很难预先判断哪些节点比较可能是社区的中心，所以平等的给每个节点初始化一个不同的标签也不失为一种办法。

同步实现

基本思路:

由于上一任务实现了迭代更新的`PageRank`，很容易想到仿照去写，仍然分为三个阶段，分别负责初始建图和分配标签、迭代更新标签、以及将节点按标签分类输出。

Phase 1: Graph Builder

Mapper:

为N个节点依次分配1~N的初始标签。

```
public class LabelGraphBuilderMapper extends Mapper<>
{
    private static int index;

    protected void setup(Context context) {
        index = 0;
    }

    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {
        IntWritable label = new IntWritable(++index);
        context.write(label, value);
    }
}
```

本阶段不需要Reducer。

Phase 2: Label Propagation Iterator

自定义数据格式`LabelPropBean`:

```
public class LabelPropBean implements Writable
{
    private int label;      // 标签
    private String name;    // 名字
    private int old_label;  // 旧标签
    private Hashtable<String, Double> link_list;    // 邻居
    private int list_length;    // 邻居数量
    private String type;    //类型
}
```

Mapper:

Map对上阶段的`<人名, 标签, [[(邻居1, 同现频率1), ...]>`产生两种键值对:

1. For each `邻居i` in `link_list`，输出 `<邻居i, (标签, 人名)>` ，即把上一轮的标签传给邻居们。
2. 同时为了完成迭代，需要传递每个人物的同现关系以维护图的结构，输出`<人名, (link_list, 标签)>`，即把邻居列表和上一轮更新的旧标签传给自己（旧标签的用途在Reducer中会解释）。

```
public class LabelPropIterMapper extends Mapper<>
{
    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {
        ......
        for (String pair : pairs) {
            ......
            link_list.put(u, o);
            LabelPropBean lpb = new LabelPropBean();
            lpb.setLabel_Name(label, name);
            context.write(new Text(u), lpb);
        }
        LabelPropBean lpb = new LabelPropBean();
        lpb.setLink_list(link_list, label);
        context.write(new Text(name), lpb);
    }
}
```

Reducer:

Reduce根据上面的同步更新公式，计算key对应人物的新标签。这里有一个小设计，即设置一个更新概率，通过生成随机数来决定到底要不要替换标签。通过这种随机更新或继续保持上一轮结束时的旧标签，希望扰动二部图震荡中的"互换"，从而达到伪异步的效果。概率的设置需要权衡，理论上设得越小，异步效果越好，越能够打破二部图震荡的格局，然而每轮更新的标签太少，算法效率就会很低，收敛所需的迭代轮数也会很多；设得越大，每轮正常更新的标签越多，对算法效率的影响越小，然而如果二部图非常庞大和复杂，那么"扰动"因素不够，可能还是没法使二部图合并。对于这个题目（及其测试数据来说），二部图主要是个别一对一情况（两个几乎只和彼此同现的角色），因此概率不需要设得很小，0.85就够了。此外，比起论文中提到的算法，"邻居中出现最多的标签"不是按照邻居人数，而是同现频率的和来比较。输出形如`<人名, 标签, [[(邻居1, 同现频率1), ...]>`。

```
public class LabelPropIterReducer extends Reducer<>
{
    // 设置更新的概率为0.85
    private static final double d = 0.85;

    protected void reduce(Text key, Iterable<> values, Context context)
            throws IOException, InterruptedException {
        // 用于计算每个标签对应邻居按贡献频率计分的结果
        HashMap<Integer, Double> label_score = new HashMap<>();
        // 用于保存每个邻居上一轮迭代得到的标签
        Hashtable<String, Integer> name_label = new Hashtable<>();
        // 用于保存每个邻居及其同现频率
        Hashtable<String, Double> link_list = new Hashtable<>();
        StringBuilder neighbors = new StringBuilder(key.toString());
        int chosen = 0;
        for (LabelPropBean bean : values) {
            // 把邻居上一轮迭代获得的标签保存到name_label
            if (bean.getType().equals("label")) {
                name_label.put(bean.getName(), bean.getLabel());
            }
            else if (bean.getType().equals("link_list")) {
                // 得到自己上一轮的旧标签，格式化link_list字符串
                chosen = bean.getOld_label();
                ......
```

```
            }
        }
        double random = new Random().nextDouble();
        // 按照一定的概率更新标签，否则继续保持上一轮的
        if (random < d) {
            // 计算每个标签对应的分值
            for (String neighbor : link_list.keySet()) {
                int label = name_label.get(neighbor);
                double score;
                if (!label_score.containsKey(label))
                    score = link_list.get(neighbor);
                else
                    score = link_list.get(neighbor) + label_score.get(label);
                label_score.put(label, score);
            }
            double score, max_score = 0;
            // 找出最大的分值
            ArrayList<Integer> labels = new ArrayList<>();
            for (int label : label_score.keySet()) {
                score = label_score.get(label);
                if (score >= max_score) {
                    max_score = score;
                }
            }
            // 找出所有分值（并列）最大的标签
            for (int label : label_score.keySet()) {
                score = label_score.get(label);
                if (score == max_score) {
                    labels.add(label);
                }
            }
            // 随机取一个来替换标签
            int randInt = new Random().nextInt(labels.size());
            chosen = labels.get(randInt);
        }
        // 格式化输出<人名，标签，[[(邻居1，同现频率1)，...]>
        context.write(new IntWritable(chosen), new Text(neighbors.toString()));
    }
}
```

Phase 3: Label Propagation Viewer

将最终结果归并输出。 从最后一次迭代的结果读出人物和标签，按标签输出属于该社区的人物。

具体实现比较简单，这里不展开了。

迭代终止条件

迭代至固定次数（默认20次，用户也可以手动输入）。

多趟`MapReduce`的处理

`LabelPropDriver`负责以上三个阶段的调配，与`PageRankDriver`类似，也不展开了。

遇到问题

- 二部图震荡效应：完全按照同步更新，用本地sample数据测试时，注意到`<龙女，龙婆>`分别都只和对方同现，即图中是含有二部图的，而输出结果中这两个人物标签一直在互换，无论迭代多少轮都不会合并到同一个社区。一种解决方法是设置一个随机概率决定是否更新标签，从而使每个人物的标签在一定程度上交替更新，来模拟异步的效果。具体原理和效果上面已经解释过了。

优化工作

用异步实现:

很容易想到改写成异步更新。`MapReduce`的设计决定了Mapper或者Reducer节点之间是并列的，很难相互通信，因此在一个节点上更新的标签无法立即同步给其他节点，不同节点之间也无法使用其他节点上本轮更新的标签。解决方法就是只用一个Reducer来完成一轮迭代中的全部更新任务。（这里放到cleanup里面做了，验收的时候想到只要Reducer维护一个本轮已经更新过的人物标签，那么在reduce里做也是可以的，只是不知道按照数据到达Reducer的顺序，是否能够保证异步更新顺序的随机性。）

这里主要修改了同步方法中的Phase 2，Phase 1和3都和同步实现相同。

Phase 2: Label Propagation Iterator

自定义数据格式`LabelPropBean`:

因为无需记录旧标签，去掉了old_label，其他同上。

Mapper:

Map对上阶段的`<人名，标签，[[(邻居1，同现频率1)，...]>`产生两种键值对：

1. 输出`<null，(标签，人名)>`，即把上一轮更新的标签传给reducer。
2. 输出`<null，(link_list，人名)>`，即把邻居列表传给reducer。

可见mapper什么都没有做，输出的key是`NullWritable`类型，所有键值对都发射到同一个reducer。

```
public class LabelPropIterMapper extends Mapper<>
{
    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {
        ......
        for (String pair : pairs) {
            ......
            double o = Double.parseDouble(split[1]);
            link_list.put(u, o);
        }
        //
        LabelPropBean lnb = new LabelPropBean();
        lnb.setLabel_Name(label, name);
        context.write(NullWritable.get(), lnb);
        LabelPropBean llb = new LabelPropBean();
        llb.setLink_list(link_list, name);
        context.write(NullWritable.get(), llb);
```

```
    }
}
```

Reducer:

Reduce计算key对应人物的新标签。name_label仍然用于保存人名对应的标签，在迭代过程中可以修改；link_lists保存每个人物的邻居同现频率列表；neighbor_list保存邻居列表的格式化输出字符串。在reduce中保存mapper发来的旧标签和邻居列表到`Hashtable`，待cleanup中再统一计算。

```
public class LabelPropIterReducer extends Reducer<>
{
    Hashtable<String, Integer> name_label = new Hashtable<>();
    Hashtable<String, Hashtable<String, Double>> link_lists = new Hashtable<>();
    Hashtable<String,  String> neighbor_list = new Hashtable<>();

    reduce(NullWritable key, Iterable<> values, Context context)
    {
        Hashtable<String, Double> link_list;
        for (LabelPropBean bean : values) {
            // 保存上一轮得到的标签到name_label
            if (bean.getType().equals("label")) {
                name_label.put(bean.getName(), bean.getLabel());
            }
            // 保存邻居列表到link_lists，格式化的字符串到neighbor_list
            else if (bean.getType().equals("link_list")) {
                StringBuilder neighbors = new StringBuilder();
                String keyrepr = bean.getName();
                link_list = bean.getLink_list();
                link_lists.put(keyrepr, link_list);
                ......
                neighbor_list.put(keyrepr, neighbors.toString());
            }
        }
    }

    protected void cleanup(Context context)
            throws IOException, InterruptedException {
        // 打乱人名列表，从而实现随机的标签更新顺序
        List<String> list = new ArrayList<>(link_lists.keySet());
        Collections.shuffle(list);
        for (String name : list) {
            // 对每个人依次计算新的标签
            int chosen;
            Hashtable<String, Double> link_list = link_lists.get(name);
            Hashtable<Integer, Double> label_score = new Hashtable<>();
            ......
            int randInt = new Random().nextInt(labels.size());
            chosen = labels.get(randInt);
            // 计算出的新标签不仅要输出，还要更新到name_label
            name_label.put(name, chosen);
            Text repr = new Text(neighbor_list.get(name);
            context.write(new IntWritable(chosen), repr));
        }
    }
}
```

迭代终止条件

迭代至固定次数（默认20次，用户也可以手动输入）。

优化取得的效果

- 收敛需要的迭代轮数减少了，且每一轮迭代所需的时间没有增加，因此总的算法效率提升了。可见并不是所有的问题都适合使用`MapReduce`进行并行处理，这里实际上完全是串行的作法，但是更适合异步更新的需求。(实际上不完全串行也可以实现，我们之前没有想到。)

进一步优化

迭代终止条件的判断:

考虑可能不收敛的情况，添加"每个节点在当前社区内的邻居不少于社区外的邻居"的判断条件。在异步更新的基础上进行修改，在cleanup里完成所有人物的标签更新后，可以遍历计算每个人物是否符合社区内的邻居同现频率之和不低于社区外的，全部符合的话，就想办法通知外部提前停止迭代。

可是怎样"通知外部"呢？首先想到`context.getConfiguration().set()`，然而，很不幸，context无论从外界传递给job，还是由job分配给诸mapper和reducer，都用的是值传递而非引用传递。这也是为了维持context的全局统一，简而言之，configuration可以在job外set，mapper和reducer里get，即使只用到一个reducer，也不能指望在这个reducer里set一个值，再到job外面去get。这个"传不出reducer的configuration"耗费了我们大量时间去定位，最后转而在专门建一个文件来存放这个值。

Label Propagation Reducer:

cleanup中统一计算和更新所有人物的标签后，进行是否达到终止条件的判断并将结论写入文件。

```
public class LabelPropIterReducer extends Reducer<>
{
    protected void cleanup(Context context)
            throws IOException, InterruptedException {
        ......  // 其他同上（异步方法）
        boolean finished = true;
        // 对每个人物计算当前标签下的邻居同现频率之和
        for (String name : list) {
            Hashtable<String, Double> link_list = link_lists.get(name);
            int the_label = name_label.get(name);
            double score = 0;
            for (String neighbor : link_list.keySet()) {
                int label = name_label.get(neighbor);
                if (label == the_label)
                    score += link_list.get(neighbor);
            }
            // 若有人这个值小于0.5，则尚未达到终止条件
            if (score < 0.5)
                finished = false;
        }
        // 将是否达到终止条件写入给定的filePath
```

```
        Configuration configuration = context.getConfiguration();
        String filePath = configuration.get("filePath");
        Path path = new Path(filePath);
        FileSystem fileSystem = path.getFileSystem(configuration);
        FSDataOutputStream outputStream = fileSystem.create(path, true);
        outputStream.write(String.valueOf(finished).getBytes());
        outputStream.close();
    }
}
```

Label Propagation Iterator:

这个模块的main函数返回值是一个布尔值，即达到迭代终止条件与否。

```
public class LabelPropIter
{
    public static boolean main(String[] args, int itrCnt, String filePath)
    {
        try {
            // 设置这个保存判断结论的文件路径
            Configuration conf = new Configuration();
            conf.set("filePath", filePath);
            ......
            // job结束后，从文件中读出是否迭代终止条件的结论并返回
            Text line = new Text();
            if (lineReader.readLine(line) > 0) {
                String finished = line.toString();
                lineReader.close();
                return finished.equals("true");
            }
            return true;
        }
        ......
    }
}
```

Label Propagation Driver

当`LabelPropagationIter.LabelPropIter.main()`返回值为true时，跳出循环。

```
public class LabelPropDriver
{
    public static void main(String[] args) throws IOException
    {
        ......
        while (true) {
            forItr[0] = args[1] + "/Round" + (i);
            forItr[1] = args[1] + "/Round" + (i + 1);
            if (LabelPropIter.main(forItr, i + 1, filePath))
                break;
            i++;
        }
```

```
        String[] forRV = {args[1] + "/Round" + (i + 1), args[1] +
"/Communities"};
        ......
    }
}
```

最终迭代轮数

根据反复测试，实际收敛所需的迭代轮数往往在3-5之间，故之前即使手动输入迭代10轮也是太浪费了。

程序的性能分析

效率上："同步+随机更新伪异步" < "手动输入迭代轮数的异步" < "自行判断迭代终止条件的异步"

程序运行截图

application_1656400433955_2525	2021sg07	LabelPropViewer	MAPREDUCE	root.2021s	Mon Jul 4 15:50:17 +0800 2022	Mon Jul 4 15:50:32 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_2524	2021sg07	LabelPropIter3	MAPREDUCE	root.2021s	Mon Jul 4 15:49:59 +0800 2022	Mon Jul 4 15:50:15 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_2523	2021sg07	LabelPropIter2	MAPREDUCE	root.2021s	Mon Jul 4 15:49:41 +0800 2022	Mon Jul 4 15:49:56 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_2522	2021sg07	LabelPropIter1	MAPREDUCE	root.2021s	Mon Jul 4 15:49:23 +0800 2022	Mon Jul 4 15:49:39 +0800 2022	FINISHED	SUCCEEDED
application_1656400433955_2521	2021sg07	LabelGraphBuilder	MAPREDUCE	root.2021s	Mon Jul 4 15:49:10 +0800	Mon Jul 4 15:49:21 +0800 2022	FINISHED	SUCCEEDED

- 同步+随机更新迭代20轮：

- 异步更新迭代10轮：

- 异步更新迭代至“每个人物社区内的邻居不少于社区外的”：

可见三种作法最后都收敛了，且共同点是都有一个非常巨大的社区，一个中等规模的社区，`<虎将，熊师>`和`<千里眼，顺风耳>`这两对构成的两个小社区。有些时候，`<急如火，掀烘兴，云里雾>`这三位会单独构成一个社区，还有时候是`<狻猊，雪狮，伏狸，黄狮，白泽，青脸儿>`，更多时候则全部并入最大的社区里。这就验证了算法具有一定的随机性，然而在这个题目中，随机性总的来说并不是很大。

在非常偶然的情况下，`<广智，广谋>`这二位也会单独构成一个社区，但它们绝大多数情况下都在最大的社区里。从任务3输出的同现频率看，这一现象其实也非常容易理解：

```
广智  [广谋，0.5|白龙马，0.5]
广谋  [广智，0.5|白龙马，0.5]
```

所以，广智和广谋被分到一个单独的社区需要满足以下条件：它们在之前的轮次里都还没有被随机更新成和白龙马相同的标签——只要有一个人已经这样了，另一个就只能也加入白龙马所在的社区，最后也就是最大的这个社区；它们在这一轮里先更新的那个人碰巧随机到了对方，而不是白龙马——这些条件很难全都满足。所以在三种方法的大量测试中，只有极少几次`<广智，广谋>`单独构成一个社区。

测试结果在集群上的存储位置

HDFS分布式存储 模拟超级

	LabelPropStop	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:50:21 GMT+0800 (中国标准时间)
	LabelPropAsync	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:44:10 GMT+0800 (中国标准时间)
	LabelPropSync	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:37:40 GMT+0800 (中国标准时间)
	PageRank20	0B	2021sg07	supergroup	drwxr-xr-x	Thu Jun 30 2022 09:21:15 GMT+0800 (中国标准时间)
	PageRank	0B	2021sg07	supergroup	drwxr-xr-x	Thu Jun 30 2022 09:08:37 GMT+0800 (中国标准时间)
	Normalization	0B	2021sg07	supergroup	drwxr-xr-x	Thu Jun 30 2022 09:03:35 GMT+0800 (中国标准时间)
	Cooccurance	0B	2021sg07	supergroup	drwxr-xr-x	Thu Jun 30 2022 09:01:22 GMT+0800 (中国标准时间)
	Preprocess	0B	2021sg07	supergroup	drwxr-xr-x	Wed Jun 29 2022 15:30:50 GMT+0800 (中国标准时间)

其中从下往上依次是任务1、2、3、4，以及任务5的三种实现。

文件名	大小	用户	用户组	权限	上次修改
Communities	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:50:32 GMT+0800 (中国标准时间)
Round3	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:50:14 GMT+0800 (中国标准时间)
finished	4B	2021sg07	supergroup	-rw-r--r--	Mon Jul 04 2022 15:50:14 GMT+0800 (中国标准时间)
Round2	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:49:56 GMT+0800 (中国标准时间)
Round1	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:49:38 GMT+0800 (中国标准时间)
Round0	0B	2021sg07	supergroup	drwxr-xr-x	Mon Jul 04 2022 15:49:20 GMT+0800 (中国标准时间)

任务5的文件夹中是每一轮的迭代结果和最终的社区，finished用于Reducer输出迭代是否终止。