1. 主要问题描述

生活中的网络非常多，比如人际网络，由网页的超链接构成的网络，还有计算机网络等等，这些网络看起来与实际生活没有什么联系，但其实是非常重要的。比如在做推销的时候，一个朋友多的人，肯定比朋友少的人有更大的推销潜力，也可以称之为“人脉资源”，而如果能找到这种朋友多的人，让他来帮着做推销，就比随便找个人做的效果好很多。又比如，在搜索网页的时候，用户肯定希望热门网页（比如官方网页）排在前面，那怎么知道这个网页是否在网络上的“分量”呢？有一种办法就是比较有多少网页有到该网页的链接，这个数量越多，这个网页的“分量”也就越大。再比如，网络上如果出现谣言，相关机构应该先控制住谁，让谁先来澄清？肯定也是粉丝多的“大 V”，当然，“大 V”很少会传播谣言，因为他的影响力太大了。除此之外，我们在打击犯罪团伙的时候，逮捕了一团队的某一个成员时，也经常会试图找到与他联络的人（也就是网络里与他“相邻”的人），然后逐渐地把整个团伙都揪出来。

所以问题的关键往往在于，我们要如何找到那个网络中“重要”的结点，也就是给网络中的结点按重要程度排序，这样的需求催生了很多种排序模型，有基于度的，有基于概率的，还有基于信息熵的。但只排序还不够，我们还需要评价算法的结果，而 SIR 传播仿真算法，就从流行病的传播的角度给出了一个答案。

2. 方法描述

2.1. 基本概念

邻居：网络中一条边相连的结点互为邻居。

度：一个结点的邻居数量称为度

邻接矩阵：是一个 结点数量 x 结点数量 大小的矩阵，在无权图中，第 i 行第 j 列如果为 1，表示 i 结点和 j 结点相连，如果为 0 表示不相连。

聚集系数：集聚系数是图中的点倾向于集聚在一起的程度的一种度量。

结点间距离：通常来说，结点之间存在一条最短的路径（从一个点经过若干边到另一个结点），路径上边的数量成为路径的长度，结点间距离就等于这个最短路

径的长度。

2.2. SIR 传播仿真模型

SIR 模型是传染病模型中最经典的模型，其中 S 表示易感者，I 表示感染者，R 表示移出者。模型中把传染病流行范围内的人群分成三类：S 类，易感者（Susceptible），指未得病者，但缺乏免疫能力，与感病者接触后容易受到感染；I 类，感病者（Infective），指染上传染病的人，它可以传播给 S 类成员；R 类，移出者（Removal），指被隔离，或因病愈而具有免疫力的人。其中 S 被附近结点感染的概率为μ，如果 S 状态结点附近有 m 个 I 状态结点的话，S 在下个时间段被感染的概率就是$1-(1-\mu)^m$，如果本身就是 I 结点，那么下个阶段康复的概率为 β。

在处理网络问题的时候，对应的概念有一些变化，S 代表未接收到某种消息的人，I 是已经知道消息的人，R 是对消息厌倦了的人，被感染就对应着被传播消息，痊愈就表示对消息厌倦了，不会再传播了。

对于结点的评价，先固定一个传播轮数，然后选择一个起点开始传播，进行模拟，最后统计 I 或 R 状态的结点总数量作为起点的重要程度评估，也就是这个数量越大，起点结点就越重要。对所有结点进行模拟之后，就能得出重要性排序。因为模拟的时候是使用概率模拟的，所以一个结点最好是模拟若干次然后取平均权值。

2.3. 基于度的结点排序模型

度排序：统计所有结点的度数，将度数直接作为结点的重要程度，将结点按照度数从大到小排序，结果即为重要性排序结果。度排序太过简单粗暴，很多时候效果也并不好，偏差很大。

Kshell 排序：简单来说是通过递归地剥离网络中度数小于某个值的点来赋予结点 Kshell 值，意图是想度量一个结点有多靠近结点“中心”，具体的度量过程在算法流程小节中再叙述。

H-index 排序:给每个结点定义一个值，如果某个结点的 H-index 值为 K，那么它周围一定有 K 个度至少为 K 的点，而每个结点的 H-index 值，都会尽量取大。他和度排序有一点相似，但是条件更为苛刻，比度排序能更好的刻画结点的重要

程度

这三种方式有着千丝万缕的联系，假想如果将 H-index 的流程再做一遍，但是将 K 个度为 K 的结点变为 K 个 H-index 值为 K 的结点，这样获取得到 2 阶 H-index 值，这个流程还可以一直做下去，最终每个结点的 H-index 值会逼近 Kshell 值，也就是说，度数做一次 H-index 函数变为 H-index 值，再做很多次 H-index 函数最终会变为 Kshell 值。

2.4. 基于概率的结点排序模型

之前在介绍 SIR 模型的时候有提到，如果一个 S 结点周围有 m 个 I 结点，那么他下一次被不感染的概率为$1-(1-\mu)^{m}$，那么自然而然就会想到，如果我计算出以某个结点为起点时，所有点被感染的概率，并且加起来当作一个权值，那么更大的权值就会代表起点结点更重要。基于概率的结点排序模型就是使用这样的方法来进行排序的。并且由于μ一般不会很大，所以在经过几轮传播之后，被感染的概率就会很小，对权值的影响也就可以忽略不计，就意味着后面的情况我们都不用算了。也就表示，无论网络多大，每次需要计算的结点数量都不会很大，这保证了这个算法的时间复杂度会比较优秀。

2.5. 基于信息熵的结点排序模型

定义一个结点的信息熵大小：$h_i = \Sigma_{j\in T_i} - p_j \log(p_j)$，其中$p_j = \frac{k_j}{\Sigma_{l\in T_j} k_l}$，$k_i$就是 i 结点的度数。形象来理解就是附近的结点度数越大，信息熵会越大，越大的信息熵就代表结点更重要。

2.6. 相关性系数计算

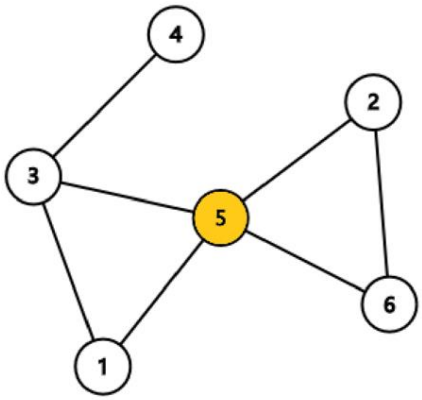
在 SIR 传播仿真和使用结点排序模型之后，我们会得到两个序列，分别是传播仿真和结点排序模型估计的重要性排序后的结点序列，那么如何来评价排序模型的优劣？这时候需要计算两个序列的相关性，相关性越大，说明排序模型的效果越好。具体的计算方法在下面会说明。

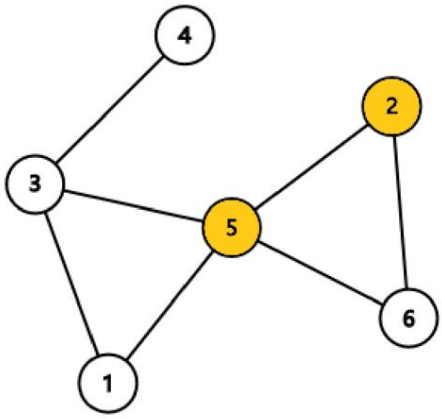
3. 算法流程

3.1. SIR 传播模型

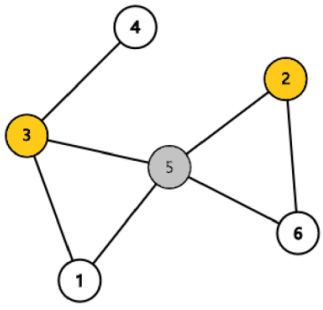
- 选择一个开始结点，开始传播
- 进行固定轮次的传播，每次传播中，枚举当前处于 I 状态的结点，以μ概率试图传播给相邻的 S 状态的结点，如果成功传播，将 S 状态结点改为 I 状态结点，在检查完所有相邻的 S 状态结点之后，当前结点有β的概率对消息厌倦，即变为 R 状态。
- 统计 I 和 R 状态的结点数量，越多表示开始结点更重要，将结点按照这个数量排序就可以得到重要性排序的结点列表。

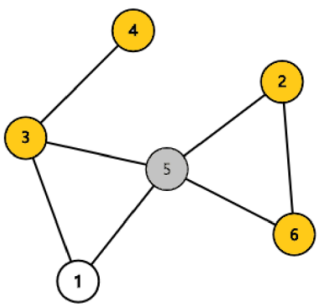
下面用两个网络来模拟一下：假如 5 号结点作为起点，模拟 4 轮，以 0.25 的概

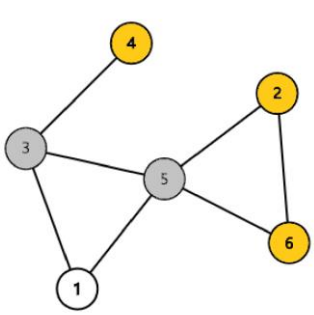




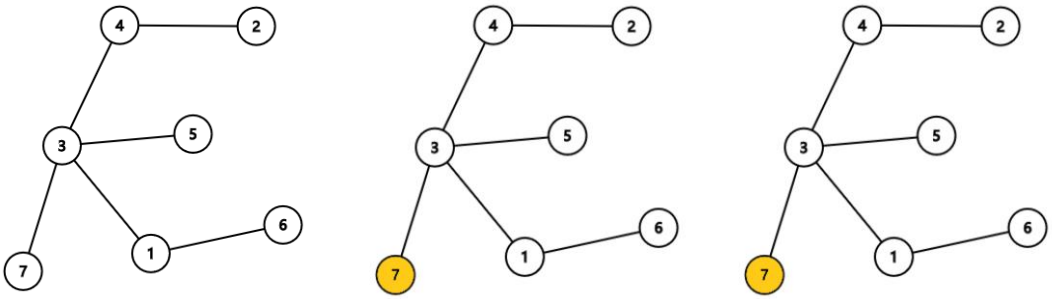
率传播到其他结点，第一轮：2 号结点被传播了，5 号结点仍处于 I 状态。第二轮：5 号结点将 3 号结点传播，转为 I 态，2 号结点未传播任何结点。第三轮：2 号结点传播 6 号结点，仍处于 I 状态，3 号结点传播 4 号结点，仍处于 I 态。第四轮：未出现新传播，3 号结点转为 R 态。四轮结束后，I 或 R 状态共有 5 个点。



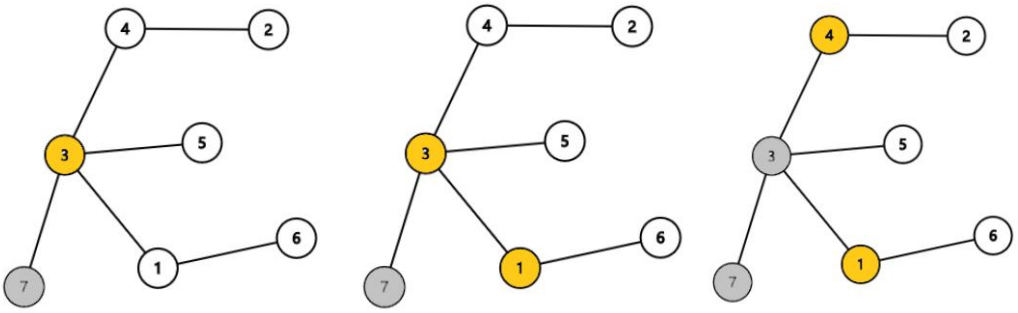




下面是第二个网络：从 7 号结点开始，其他数据和上面的一样。第一轮，7 号结点未能传播其他节点，第二轮，7 号结点将 3 号结点传播，转为 R 态。



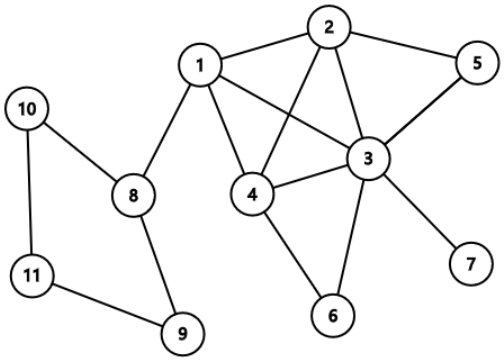
第三轮：3 号结点传播给 1 号结点，仍处于 I 态。第四轮：3 号结点传播给 4 号结点，转为 R 态。四轮结束共有 4 个结点是 I 或 R 状态。



3.2. 基于度的结点排序模型

度排序：

统计结点度数，然后按度数从大到小排序。以一个图作为示例来解释这个算法：



首先统计各个结点的度数。

结点	1	2	3	4	5	6	7	8	9	10	11
度数	4	4	6	4	2	2	1	3	2	2	2

再按照度数从大到小将结点排序，排序之后：

结点	3	1	4	2	8	5	6	9	10	11	7
度数	6	4	4	4	3	2	2	2	2	2	1

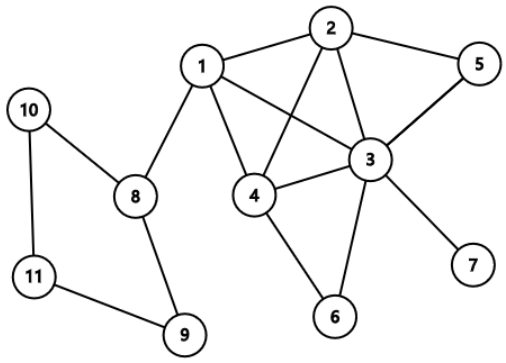
该算法认为 3 是最重要的结点，1 次之，以此类推。

Kshell 排序模型：

首先选择一个度为 1 的结点并将其删除，并将其的 Kshell 值设为 1，并且在删除之后检查是否还有度为 1 的结点，将其 Kshell 值设为 1，如此往复直到没有度为 1 的。

然后选择一个度为 2 的结点并将其删除，将其 Kshell 值设为 2，并且在删除之后检查是否还有度小于等于 2 的结点，将其 Kshell 值设为 2，如此往复直到所有结点度数都大于 2。

然后依次选择度为 3，4，5 的结点做这样的流程，直到所有点都被删除。下面以一个图作为例子：



首先找到度数为 1 的 7 号结点，删除，Kshell 值设为 1。没有度数为 1 的了，进入下一轮。

结点	1	2	3	4	5	6	7	8	9	10	11
Kshell							1				

找到度数为 2 的，10，11，9，5，6 号结点，删除之后发现 8 号结点度数变为 1，继续删除 8 号。没有度数小于等于 2 的了，进入下一轮。

结点	1	2	3	4	5	6	7	8	9	10	11
Kshell					2	2	1	2	2	2	2

找到度数为 3 的，1，2，3，4 号结点，删除后，所有结点都被删除。

结点	1	2	3	4	5	6	7	8	9	10	11
Kshell	3	3	3	3	2	2	1	2	2	2	2

按照 Kshell 值排序之后的结果：

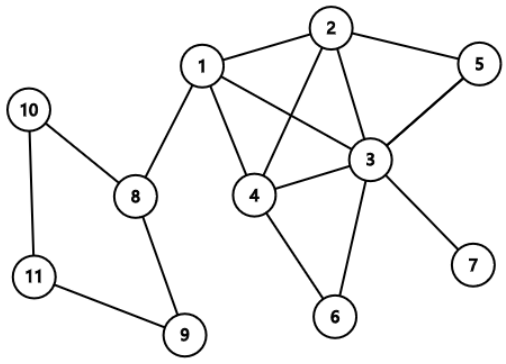
结点	1	2	3	4	5	6	8	9	10	11	7
Kshell	3	3	3	3	2	2	2	2	2	2	1

H-index 排序模型：

首先计算出每个结点的度数，然后依次处理每个结点。

对当前结点的 H-index 值进行二分，左边界初始为 0，右边界为 n，每次二分检查左右边界的中点（整除），假设为 mid，计算当前结点有多少个邻居的度数≥ mid，如果数量≥ mid，那么将左边界设置为 mid+1，并更新答案为 mid，否则将右边界设置为 mid-1，如此进行下去直到左右边界不满足左边界≤有边界的关系，得到最终答案。

下面以一个图作为例子：



首先统计度数：

结点	1	2	3	4	5	6	7	8	9	10	11
度数	4	4	6	4	2	2	1	3	2	2	2

然后计算每个点的 H-index 值，以 1 号点为例：首先 l=0,r=11。

第一轮 mid=5，1 号结点的邻居有 0 个度数大于等于 5，令 r=mid-1=4

第二轮 mid=2，1 号结点的邻居有 4 个度数大于等于 2，记录答案=2，l=mid+1=3

第三轮 mid=3，1 号结点的邻居有 3 个度数大于等于 3，记录答案=3，l=mid+1=4

第四轮 mid=4，1 号结点的邻居有 3 个度数大于等于 4，令 r=mid-1=3

此时不满足l ≤ r关系式，计算结束，答案为 3。按照这样的方法计算，结果如下：

结点	1	2	3	4	5	6	7	8	9	10	11
H-index	3	3	3	3	2	2	1	2	2	2	2

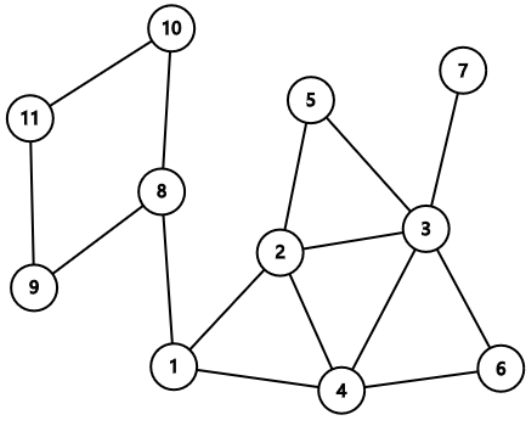
排序结果如下:

结点	1	2	3	4	5	6	8	9	10	11	7
H-index	3	3	3	3	2	2	2	2	2	2	1

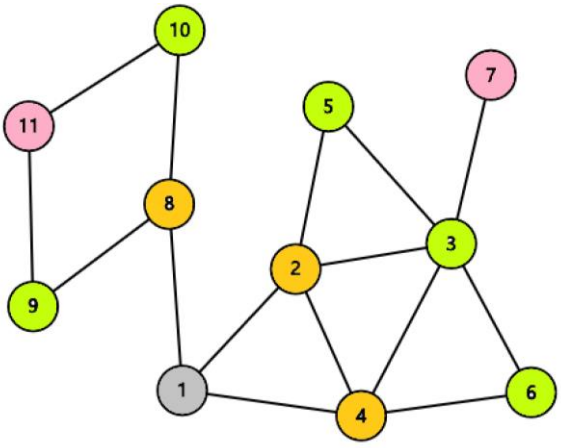
3. 3. 基于概率的结点排序模型

定义两个值$\mathrm{unif_s(p)}$表示 p 点不被传播到的概率，$\mathrm{score(p)}$表示 p 点被传播到的概率，显然$\mathrm{score(p)} = 1 - \mathrm{unif_s(p)}$。

首先枚举每个点，假设他以 1 的概率被传播到，然后进行 BFS 分层，每个结点的值都由与它相邻的上一层的结点决定，具体式子为$\mathrm{unif_s(p)} = \prod_{q \in T_{i-1}(u)}[1 - score(q) * \beta]$，$\mathrm{score(q)} * \beta$就是上一层的结点将其传播到的概率。下面通过一个例子来说明:



假设从 1 号点开始，首先分层。



结点	1	2	3	4	5	6	8	9	10	11	7
Score	1										

然后计算第二层的 score，假设β=0.25

2 号：unif_s(2)=1-score(1)*0.25=0.75, score(2)=1-unif_s(2)=0.25

4，8 号结点类似.

结点	1	2	3	4	5	6	8	9	10	11	7
Score	1	0.25		0.25			0.25				

第三层：

5 号：unif_s(5)=1-score(2)*0.25=0.9375, score(5)=0.0625

3 号：unif_s(3)=(1-score(2)*0.25)*(1-score(4)*0.25)=0.8789, score(3)=0.1211

6，9，10 号结点与 5 号结点类似

结点	1	2	3	4	5	6	8	9	10	11	7
Score	1	0.25	0.1211	0.25	0.0625	0.0625	0.25	0.0625	0.0625		

第四层：

11 号：unif_s(11)=(1-score(10)*0.25)*(1-score(9)*0.25)=0.9690, score(11)=0.0310

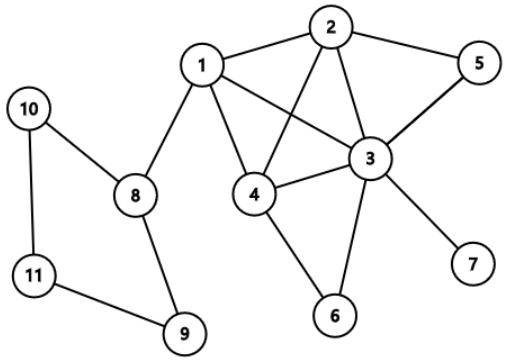
7 号: unif_s(7)=1-score(3)*0.25=0.9697, score(7)=0.0303

结点	1	2	3	4	5	6	8	9	10	11	7
Score	1	0.25	0.1211	0.25	0.0625	0.0625	0.25	0.0625	0.0625	0.031	0.0303

可以发现第四层的 score 已经比较小了，最多再计算三四层，score 就会小到可以忽略不计。此时 score 和为：2.1824，这个就是 1 号点的权值。还需要计算出其他的结点的权值，就可以进行排序了。

3.4. 基于信息熵的结点排序模型

如算法介绍里所说，结点的信息熵大小：$h_i = \Sigma_{j \in T_i} - p_j \log(p_j)$，其中$p_j = \frac{k_j}{\Sigma_{l \in T_j} k_l}$，$k_i$就是 i 结点的度数。首先计算出结点的度数，然后枚举每个点，按照这个公式计算p_j和h_i，最后按照h_i从大到小排序即可。以一个网络作为例子：



结点	1	2	3	4	5	6	7	8	9	10	11
度数	4	4	6	4	2	2	1	3	2	2	2

首先计算p_j，以结点 1 为例：

$$p_1 = \frac{k_1}{\Sigma_{l \in T_1} k_l} = \frac{4}{4+6+4+3} = 0.2353$$

全部的p_j列表如下

结点	1	2	3	4	5	6	7	8	9	10	11
p	0.2353	0.25	0.3529	0.25	0.2	0.2	0.1667	0.375	0.4	0.4	0.5

然后计算h_i，以结点 1 为例：

$$h_1 = -(0.25 * \log_2 0.25 + 0.3529 * \log_2 3529 + 0.25 * \log_2 0.25 + 0.375 * \log_2 0.375) = 2.0609$$

计算出所有h_i

结点	1	2	3	4	5	6	7	8	9	10	11
h	2.061	1.986	2.851	1.986	1.030	1.030	0.530	1.549	1.031	1.031	1.058

后按照h_i从大到小排序结点即可。

结点	3	1	2	4	8	11	9	10	5	6	7
h	2.851	2.061	1.986	1.986	1.549	1.058	1.031	1.031	1.030	1.030	0.530

该算法认为 3 号结点是最重要的结点，后面依次是 1,2,4,8,…

3.5. 相关性系数计算

假设两个序列是$X=(x_1,x_2,\dots,x_n),Y=(y_1,y_2,\dots,y_n)$，那么他们的相关性系数定义为$\tau=\frac{2(n_+-n_-)}{n(n-1)}\in[-1,1]$，其中$n_+$表示两个序列中满足如下关系的(i,j)对：$(x_i<x_j\ and\ y_i<y_j)\ or\ (x_i>x_j\ and\ y_i>y_j)$，简单来说就是大小关系一样的数对数量，而$n_-$表示大小关系不一样的数对数量，而如果有相等的数对的话，既不计入n_+也不计入n_-，而分母的 n(n-1)表示的是所有的数对的数量，形象理解的话就是大小关系正确就有一个正权值，错误就有一个负权值，然后除以所有数对数量。过这个方法的分母中并没有考虑相等的数对，所以可能会有一些偏差，为了消除这个偏差，有一种修正后的计算方法：$\tau=\frac{n_+-n_-}{\sqrt{(n_0-n_1)(n_0-n_2)}}$，其中$n_1$表示 x 序列中的相等数对，$n_2$表示 y 序列中的相等数对，具体计算方法:$n_1=\Sigma k_i(k_i-1)/2$，$k_i$表示的是第 i 种权值的数量，随机组合之后就是这么多对相等数对。

下面举两组例子来说明：

序列 X：4 2 11 3 6 8 11 7 6 8

序列 Y：10 13 1 4 8 3 3 4 7 4

$$n_1=\frac{(1*0+1*0+1*0+2*1+1*0+2*1+2*1)}{2}=3$$

$$n_2=\frac{1*0+2*1+3*2+1*0+1*0+1*0+1*0}{2}=4$$

$$n_+=3,n_-=35$$

$$\tau=\frac{3-35}{\sqrt{(45-3)(45-4)}}=-0.77$$

两个序列呈较强负相关性。

序列 X：11 2 4 5 5 8 4 9 13 12

序列 Y：9 8 10 13 3 3 12 4 12 5

$$n_1=\frac{1*0+2*1+2*1+1*0+1*0+1*0+1*0+1*0}{2}=2$$

$$n_2=\frac{2*1+1*0+1*0+1*0+1*0+1*0+1*0+1*0+1*0}{2}=1$$

$$n_+=21,n_-=20$$

$$\tau=\frac{21-20}{\sqrt{(45-2)*(45-1)}}=0.023$$

因为接近 0，所以两个序列没有什么相关性。

4. 核心算法代码

4.1. SIR 传播仿真模型

```
//对于单个结点模拟，返回固定回合后被感染的总节点数量
    private int run(T s,Graph<T> graph){
        int count=0;
        Queue<Pair<T,Integer>> q = new LinkedList<>();
        state.put(s,State.I);
        q.offer(new Pair<>(s,0));++count;//放入初始点，更新状态
        while(!q.isEmpty()){
            Pair<T,Integer> u = q.poll();
            if(u.getValue() > len) continue;
            for(T v : graph.to(u.getKey())){
                if(state.get(v)==State.S && r.nextDouble()<pu){//随
机结果为感染下一个 S 点
                    ++count;//计数
                    state.put(v,State.I);//修改状态
                    Pair<T,Integer> nxt = new Pair<>(v,u.getValue()
+1);
                    q.offer(nxt);//入队
                }
            }
            if(r.nextDouble()<pb){//如果变为 R 就不传播了
                state.put(u.getKey(),State.R);
            }else{//否则还会继续传播
                Pair<T,Integer> nxt = new Pair<>(u.getKey(),u.getVa
lue()+1);
                q.offer(nxt);
            }
        }
        return count;
    }

    //进行仿真，返回一个按重要度排序的结点和重要度数组
    public ArrayList<T> simulate(Graph<T> graph){
        Set<T> points = graph.getVertex();
        ArrayList<T> result = new ArrayList<>();
        Map<T,Double> sumMap = new HashMap<>();
        for(T v : points){
            double sum=0;
            for(int i=0;i<N;++i){
```

```
                for(T v2 : points) state.put(v2,State.S);//初始化状态全为 S
                    sum += run(v,graph);
                }
                sum/=N;
                sumMap.put(v,sum);
                result.add(v);
            }
            //从大到小排序
            result.sort((o1, o2) -> -sumMap.get(o1).compareTo(sumMap.get(o2)));
            return result;
        }
```

4.2. 基于度的结点排序模型

度排序：

```
//统计每个点的度数
public Map<T,Integer> getDegree(){
    Map<T,Integer> D = new HashMap<>();
    for(T v : getVertex()){
        D.put(v,to(v).size());//度数就是出边的数量
    }
    return D;
}

//单纯按照度数排序
public ArrayList<T> sortByDegree(Graph<T> graph){
    Map<T,Integer> D = graph.getDegree();
    ArrayList<T> v = new ArrayList<>(graph.getVertex());
    v.sort((o1, o2) -> -D.get(o1).compareTo(D.get(o2)));
    return v;
}
```

H-index 排序：

```
//H-index 排序
    public ArrayList<T> sortByHindex(Graph<T> graph){
        Map<T,Integer> D = graph.getDegree();//获取度数
        Map<T,Integer> H = new HashMap<>();
        for(T u : graph.getVertex()){//枚举每个点计算
            int l=1,r=graph.vertexSize(),mid,ans=0;//二分 H 值
```

```
            while(l<=r){
                mid = (l+r)/2;
                int count=0;
                for(T v : graph.to(u)){//计算 mid 是否符合要求
                    if(D.get(v) >= mid) ++count;
                }
                if(count >= mid) {ans=mid;l=mid+1;}
                else r=mid-1;
            }
            H.put(u,ans);
        }

        //使用 H 排序
        ArrayList<T> res = new ArrayList<>(graph.getVertex());
        res.sort((o1, o2) -> -H.get(o1).compareTo(H.get(o2)));
        return res;
    }
```

Kshell 排序：

```
//Kshell 值排序
    public ArrayList<T> sortByKshell(Graph<T> graph){
        Map<T,Integer> D = graph.getDegree();

        int maxD=0;//先统计出最大度数
        for(Map.Entry<T,Integer> entry : D.entrySet()){
            maxD = Math.max(maxD,entry.getValue());
        }

        Queue<T> q = new LinkedList<>();//保存因为度数减少而变为待处理的
点
        Set<T> vis = new HashSet<>();//标记点是否处理过

        Map<T,Integer> Kshell = new HashMap<>();
        for(int i=0;i<=maxD;++i)
        {//从小到大处理度数
            for(T u : graph.getVertex()) if(D.get(u)<=i && !vis.con
tains(u)){//枚举当前删除的点
                q.offer(u);
                vis.add(u);
                Kshell.put(u,i);
                D.put(u,D.get(u)-1);
                while(!q.isEmpty())
                {//还有待处理点的时候一直处理
                    T v = q.poll();
```

```
                    for(T t : graph.to(v)) {
                        D.put(t, D.get(t) - 1);//减少一度
                        if (D.get(t) <= i && !vis.contains(t)) {//
减少之后已经小于等于当前处理的度数，需要处理
                            q.offer(t);
                            vis.add(t);//标记为处理过，防止下次从 list
中重复处理
                            Kshell.put(t, i);
                        }
                    }
                }
            }
        }
        //使用 Kshell 数组排序
        ArrayList<T> res = new ArrayList<>(graph.getVertex());
        res.sort((o1, o2) -
Kshell.get(o1).compareTo(Kshell.get(o2)));

        return res;
    }
```

4.3. 基于概率的结点排序模型

```
//b 是感染概率
public Pair<Map<T,Double>,ArrayList<T>> sortByProb(Graph<T> graph,
double b){
    Queue<T> q = new LinkedList<>();
    Map<T,Double> sum = new HashMap<>();
    for(T s : graph.getVertex()){//枚举每一个点
        Map<T,Integer> deep = new HashMap<>();//标记层数，方便 BFS
        Map<T,Double> unif_s = new HashMap<>();
        for(T v : graph.getVertex()){
            unif_s.put(v,1d);//初始化
            deep.put(v,0);
        }
        unif_s.put(s,0d);//初始点不被感染的概率为 0，即一定被感染

        q.offer(s);deep.put(s,1);//起点第一层
        while(!q.isEmpty()){
            T u = q.poll();
            if(1-unif_s.get(u) < 1e-6){//剪枝，已经没有什么大影响了
                continue;
```

```
            }
            //枚举下一层的点，标记之后放入队列
            for(T v : graph.to(u)) if(deep.get(v)==0 || deep.get(v)
==deep.get(u)+1){
                if(deep.get(v)==0){
                    q.offer(v);
                    deep.put(v,deep.get(u)+1);
                }
                //计算下一层的 unif_s 值
                unif_s.put(v,unif_s.get(v)*(1-(1-
unif_s.get(u))*b));
            }
        }

        //统计概率和
        double ans = 0;
        for(T u : graph.getVertex()){
            ans += 1-unif_s.get(u);
        }
        sum.put(s,ans);
    }
    //按照概率排序
    ArrayList<T> res = new ArrayList<>(graph.getVertex());
    res.sort((o1, o2) -> -sum.get(o1).compareTo(sum.get(o2)));
    return new Pair<>(sum,res);
}
```

4.4. 基于信息熵的结点排序模型

```
public Pair<Map<T,Double>,ArrayList<T>> sortByComent(Graph<T> graph
){
    Map<T,Integer> D = graph.getDegree();
    Map<T,Double> p = new HashMap<>();
    for(T u : graph.getVertex()){
        double sum=0;
        for(T v : graph.to(u)){
            sum += D.get(v);
        }
        p.put(u,D.get(u)/sum);
    }

    Map<T,Double> h = new HashMap<>();
```

```
    for(T u : graph.getVertex()){
        double hi=0;
        for(T v : graph.to(u)){
            hi -= p.get(v)*MyMath.log2(p.get(v));
        }
        h.put(u,hi);
    }

    ArrayList<T> res = new ArrayList<>(graph.getVertex());
    res.sort((o1, o2) -> -h.get(o1).compareTo(h.get(o2)));
    return new Pair<>(h,res);
}
```

4.5. 序列相关性计算

```
//使用修正过的公式计算相关性，要求传入的数组长度相等
public static double CaculateCorrelation(int[] x, int[] y){
    if(x.length!=y.length) {
        System.out.println("传入数组长度不等");
        return Double.NaN;
    }
    //计算存在相等关系的 pair 数
    int n1=0,n2=0;
    Map<Integer,Integer> count = Count.mapCount(x);
    for(Map.Entry<Integer,Integer> entry : count.entrySet()){
        n1+=(entry.getValue()-1)*entry.getValue()/2;
    }
    count = Count.mapCount(y);
    for(Map.Entry<Integer,Integer> entry : count.entrySet()){
        n2+=(entry.getValue()-1)*entry.getValue()/2;
    }
    //计算逆序对和正序对数
    int equal=0,nx=0,n_=0;
    for(int i=0;i<x.length;++i){
        for(int j=i+1;j<y.length;++j){
            if((x[i]<x[j] && y[i]<y[j]) || (x[i]>x[j] && y[i]>y[j])
) ++nx;
            else if((x[i]<x[j] && y[i]>y[j]) || (x[i]>x[j] && y[i]<
y[j])) ++n_;
        }
    }
```

```
    return (nx-n_)/Math.sqrt((double)(x.length*(x.length-1)/2-
n1)*((y.length-1)*y.length/2-n2));
//        return 2*(nx-n_)/(double)x.length/(x.length-1);
}
```

5. 实验结果截图及分析

我选取了五个网络，获得的结果如下：

网络 1：top10 结点

Kshell：

17	18	22	29	42	43	45	47	59	63
35	35	35	35	35	35	35	35	35	35

概率模型

18	17	22	29	170	65	63	42	47	59
424.67	424.55	422.00	420.46	418.47	418.45	418.29	417.00	415.70	414.89

度数模型：

161	122	83	108	87	63	435	14	167	184
346	233	232	220	217	215	184	179	176	172

H-index 模型：

83	17	63	47	42	22	130	161	45	29
79	79	77	76	75	75	75	74	73	73

信息熵模型：

161	122	17	18	22	65	63	42	47	59
26.75	17.50	17.4	17.24	16.34	16.25	14.50	14.41	14.08	13.79

SIR 传播仿真：

18	17	65	63	42	45	161	130	29	122
196.1	193.6	189.3	186.5	183.9	183.8	183.2	181.0	180.5	179.9

相关性系数列表如下：

模型	相关系数
度数	0.47235
H-index	0.85264
Kshell	0.86524
概率	0.80253
信息熵	0.79672

网络 2：top10 结点

Kshell:

1	2	4	6	7	9	10	11	12	14
53	53	53	53	53	53	53	53	53	53

概率模型：

12	63	139	4	75	10	14	179	1	9
118.34	118.03	115.81	115.24	112.96	112.58	109.83	109.65	108.96	107.78

度数模型：

6	164	25	75	63	1	77	86	153	179
110	101	100	99	98	98	97	97	95	95

Hindex 模型：

2	4	6	9	10	11	22	23	25	35
66	66	66	66	66	66	66	66	66	66

信息熵模型：

1	4	6	9	63	139	14	179	534	12
9.55	9.45	8.99	8.90	8.88	8.54	8.28	8.20	8.15	8.06

SIR 传播仿真

6	4	9	179	10	12	4	139	534	63
931.1	930.6	930.57	930.57	930.53	930.2	930.17	930.1	930.07	929.87

相关性列表：

模型	相关性
度数	0.40314
H-index	0.84851
Kshell	0.83243
概率	0.70651
信息熵	0.85342

网络 3：top10 结点

Kshell

45	53	54	55	56	57	59	61	62	63
28	28	28	28	28	28	28	28	28	28

概率模型

449	469	452	478	221	2461	445	360	380	291
695.96	695.88	694.96	681.01	679.93	679.11	677.10	676.18	675.34	672.88

度数模型

53	65	59	56	259	297	411	308	366	57
242	218	211	172	170	169	168	166	160	159

Hindex 模型

53	59	65	57	218	223	225	366	69	220
75	73	73	71	69	68	68	66	65	65

信息熵

53	65	59	308	259	411	366	57	56	297
21.14	19.53	18.65	16.41	15.89	15.11	14.98	14.70	14.52	14.30

SIR 仿真

53	65	57	59	223	69	308	218	297	366
1361.1	1328.2	1306.4	1292.8	1266.6	1252.4	1245.1	1223.5	1217.5	1209.7

相关系数

模型	相关系数
度数	0.59963
H-index	0.63798
Kshell	0.64661
概率	0.81024
信息熵	0.32890

网络 4：top10 结点

Kshell

32	33	35	40	44	58	60	62	63	64
29	29	29	29	29	29	29	29	29	29

概率模型

136	174	88	96	69	111	70	170	83	194
82.82	81.81	81.13	81.10	80.63	80.34	79.63	78.87	78.75	78.54

度数模型

136	60	132	168	70	99	108	83	158	7
100	96	75	74	62	60	60	59	59	57

Hindex 模型

60	132	136	168	99	108	131	194	70	83
39	39	39	37	36	36	36	36	35	35

信息熵

136	60	168	132	70	149	83	158	108	99
13.35	12.42	9.77	9.58	8.15	8.01	7.77	7.76	7.74	7.70

SIR 仿真

136	174	194	88	70	18	178	158	192	60
183.4	177.5	177.2	177.0	176.7	174.7	176.6	174.6	174.5	174.2

相关系数

模型	相关系数
度数	0.70991
H-index	0.70127
Kshell	0.62249
概率	0.65245
信息熵	0.65546

网络 5：top10 结点

Kshell

120	121	122	126	131	132	133	134	135	136
10	10	10	10	10	10	10	10	10	10

概率模型

2022	120	302	784	492	566	1443	1817	549	252
247.3	228.1	226.9	226.7	224.1	216.3	211.8	211.1	209.2	207.5

度数模型

56	302	209	1443	784	147	492	120	508	644
65	64	63	63	62	57	56	55	55	55

Hindex 模型

252	302	492	61	138	167	425	120	126	133
20	20	20	19	19	19	19	18	18	18

信息熵

302	566	147	784	209	1443	508	549	492	120
12.36	12.12	11.86	11.79	11.77	11.44	10.86	10.38	10.38	10.25

SIR 仿真

302	492	784	549	120	252	566	199	209	1443
356.3	315.6	315.2	310.2	305.2	286.2	281.0	271.9	267.4	265.4

相关系数

模型	相关系数
度数	0.67380
H-index	0.71550
Kshell	0.71101
概率	0.87402
信息熵	0.56872

分析：五个网络对比

算法	相关系数
度数模型	较低(0.5 附近)
信息熵模型	较低(0.5 附近)
Kshell	较高（0.7 附近）
H-index	较高（0.7 附近）
概率模型	时高时低（0.3~0.8）

在筛选 top10 的时候，Kshell 和 H-index 由于很多点的值都一样，所以一个比较好的建议是关注所有 Kshell 或者 H-index 都是最大值的点，因为按照这两种算法，值都很大的点是没办法区分开的。想要区分开，只能换别的方法。

6. 总结

算法难度并不是很高，不过也并没有很顺利，有挺多 bug，而且在算相关系数的时候出了一点差错，是把排序之后的节点序列算相关系数，算出来就非常低，近乎于 0，每个算法都是 0，SIR 也找不出来错，非常的绝望。后来经同学提醒，才发现我算的有问题，改了之后数据就正常很多了。这时候我才想通，因为 Kshell 和 H-index 的很多点数值都是一样的，直接排序之后，必定会跟其他的有很大的差别。写这篇报告也花了很长时间，深刻的体会了写算法教材的不易。