Social Graph Restoration via Random Walk Sampling

Kazuki Nakajima、Kazuyuki Shudo

摘要

对第三方研究者来说，分析数据访问受限的社交图是困难的。为了解决这个挑战，提出了许多通过随机游走来估计结构属性的算法。但是，大部分算法局限于估计局部结构属性。我们提出了用由随机游走获得的少量样本来修复原图。我们提出的方法生成的图既能保持对局部属性的估计，又能保持随机游走采样到的子图的结构。我们将提出的方法跟使用爬取方法的子图采样和现有的通过随机游走生成与原图结构相似的方法进行了比较。我们的实验结果表明了我们提出的方法，相比于其他方法，更准确的再现了原图的局部和全局结构属性以及视觉表示。我们希望我们的方法有助于对数据访问有限的社交图进行详尽的分析。

关键词：social network analysis；social networks；social graphs；random walk；sampling；social graph restoration

1 引言

选题背景：研究社交网络对于理解社交网络中人们的关系和行为等社会特征来说式非常重要的。已经有很多对社交图的结构属性的研究了，在社交图中，节点表示用户，边表示在社交网络中用户之间的朋友关系。一般来说，研究者采样图数据用于分析，但是对于第三方研究者来说，是无法获取社交图中全部的数据。不过，爬取方法对于一个用户的邻居数据可以通过询问获得的在线社交网络是非常有效的。

选题依据：Gjoka 等人提出了基于重新加权随机游走的框架，可以通过随机游走来获取社交图结构属性的无偏估计。同时也解决了由爬取方法导致的偏向度高的节点的采样偏差。这个框架首先在图上进行随机游走得到一系列采样节点。然后对每一个采样节点进行重新加权来矫正采样偏差。但是，在实际采样阶段中，可用的询问次数是有限的。因此，目前也已经提出了很多基于这个框架并且使用少量询问来估计结构属性的算法。这些算法都有助于我们想法的实现。

选题意义：原则上，重新加权随机游走只能估计局部结构属性。因为，首先，当我们想要去估计全局结构属性的时候，这个框架需要采样大部分图数据去矫正采样偏差。其次，重新加权采样的数量对于预测原图的结构是不够的。另一方面，分析家对社交网络中感兴趣的特征通常是多样的。为了解决这个分歧，我们研究了图修复问题：通过爬取获得社交图少量的样本，生成一个图，这个图的结构属性尽可能的与原图相应属性接近。这个生成图能让我们估计原图的局部和全局结构属性以及预测原图的视觉表示。现有解决这个问题的方法有子图采样和 Gjoka 提出的图修复。在子图采样中，通过爬取方法构建子图，可以含蓄地认为这个子图是原图的一个具有代表性的样本。而 Gjoka 的方法生成的图保持了由重新加权随机游走得到的局部结构属性的估计，意在再现原图的结构属性。因此，我们扩展了叫做 dK-series 的图生成模型，尝试通过保持局部结构属性再现原图更多的结构属性。

2　相关工作

Gjoka 等人提出重新加权随机游走可以获得结构属性的无偏估计。在过去的几十年里，提出了很多基于这个框架并且使用少量的询问来准确地估计结构属性的算法。但是大部分的算法都局限于局部结构属性的估计。

我们将子图采样作为社交图修复问题的基准算法。早些时候的研究，子图是被含蓄地当作原图一个具有代表性的样本。但是 Gjoka 等人表明爬取方法通常会引入严重的对高度节点采样偏差。

Gjoka 等人提出通过重新加权随机游走生成一个保持联合度分布和度依赖聚类系数的估计的图的方法。他们表示生成的图不仅可以准确的再现局部结构属性，而且也能再现那些没有想保持的全局结构属性。

而我们提出了一个生成图的方法，生成的图不仅可以保持局部结构属性的估计，而且还能保持随机游走采样到的子图的结构。具体来讲，我们往通过随机游走采样的子图添加节点和边能确保最后的图能保持局部结构属性的估计。我们的想法是在生成图的过程中，更好地使用子图的原生结构信息。

许多研究已经发展了随机游走算法，提高了其评估器的准确性和询问的效率。Ribeiro 和 Towsley 提出了多维随机游走，提高了在简单的随机游走上估计的准确性。Lee 等人提出了非回溯的随机游走算法，提高了询问的效率并且保持了采样序列的马尔科夫特性。Nakajima 和 Shudo 最近提出了一个可以减少偏差的随机游走算法，这些偏差是由那些社交网络中邻居数据无法获取的私有节点导致的。

在这篇研究中，我们扩展了叫做 dK-series 的生成模型家族去生成一个既能保持局部结构属性的估计，又能保持由随机游走采样到的子图的结构属性的图。扩展包括 dK-series 在内的生成模型并不容易，生成模型认为所有的图数据是可以获取的，对于社交图修复问题有以下三个原因。第一：我们需要从原图的一个子图中估计模型的输入参数。第二：我们需要根据估计构建输入参数，使得参数满足所有实现目标图所需要的条件。第三：大部分的生成模型是添加节点和边到空图中，而我们是添加节点和边到采样子图中去修复原图[1]。

3 本文方法

3.1 本文方法概述

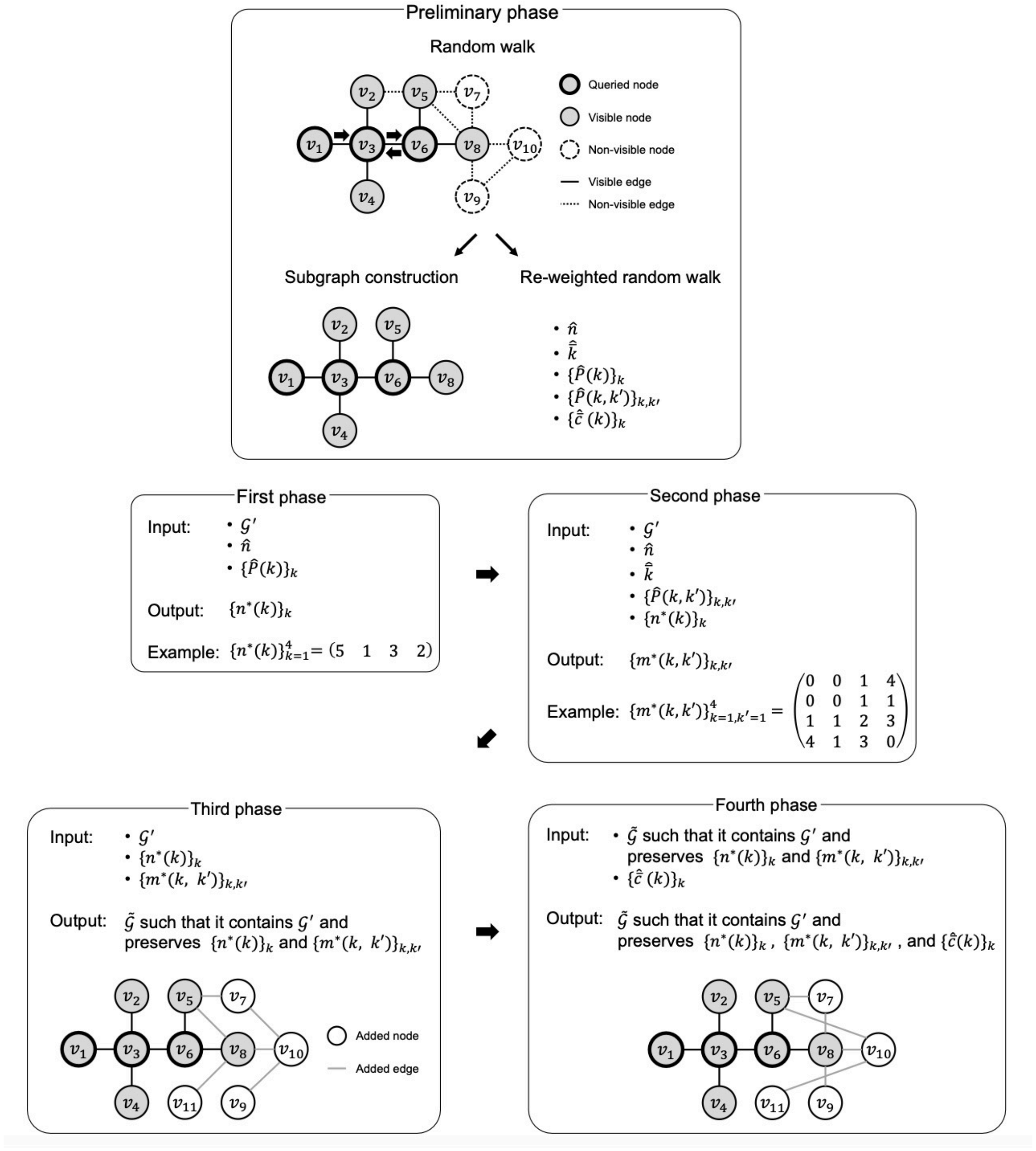


图 1: 方法示意图

3.2 阶段一

构建目标度向量，使其满足以下定理并且最小化目标度向量和原始图的度向量估计之间的误差。

1. $n^*(k)$ 是非负整数，对 $k = 1, ..., k^*_{max}$
2. $\sum_{k=1}^{k^*_{max}} kn^*(k)$ 是个偶数
3. $n^*(k) \geqslant n'(k)$，对 $k = 1, ..., k^*_{max}$

3.2.1 初始化

使用 $\hat{n}$ 和 $\hat{P}(k)$ 初始化 $n^*(k)$，根据等式 $n^*(k) = \hat{n}\hat{P}(k)$。

$$n^*(k) = \begin{cases} max(NearInt(\hat{n}\hat{P}(k)), 1) & if \quad \hat{P}(k) > 0 \\ 0 & if \quad \hat{P}(k) = 0 \end{cases}$$

3.2.2 调整

调整 $\{n^*(k)\}_k$ 使其满足定理 2，当度的总数是奇数时，选择一个度为奇数且能使 $\Delta_+(k)$ 最小的 k ，将 $n^*(k)$ 增加 1。

$$\Delta_+(k) = \begin{cases} \dfrac{|\hat{n}(k) - (n^*(k) + 1)|}{\hat{n}(k)} - \dfrac{|\hat{n}(k) - n^*(k)|}{\hat{n}(k)} & if \quad \hat{P}(k) > 0 \\ \infty & if \quad \hat{P}(k) = 0 \end{cases}$$

如果有两个以上不同的度，但 $\Delta_+(k)$ 相同的 k，则选择最小的 k 去最小化生成图边数的提升。该步骤不会破坏满足定理 1。

3.2.3 修改

为了生成具有目标度向量的图，我们需要将目标度分配给每个节点。用 d_i^* 表示给子图中节点 v_i' 分配的目标度，这个目标度受子图中这个节点度限制（定理 3）。引理：子图中所有访问节点的邻边都已经包含在子图中了，而可访问的节点的邻边并不都包含在子图中。

$$d_i = d_i' if \quad v_i' \in V_{qry}' \tag{1}$$

$$d_i \geqslant' d_i' if \quad v_i' \in V_{vis}' \tag{2}$$

对于访问节点

1. 按照任意顺序分配给访问节点目标度 $d_i^* = d_i'$
2. 计算当前子图中目标度为 k 的节点数量 $n'(k)$
3. 当 $n^*(k) < n'(k)$ 时，调整 $n^*(k) = n'(k)$

对于可访问节点

1. 选择还没分配目标度并且在子图中度最大的可访问节点 v_i'
2. 构建一个可以分配给 v_i' 的目标度的序列 $D_{seq}(i)$，其中度为 k 可以出现 $n^*(k) - n'(k)$ 次
3. 如果 $D_{seq}(i)$ 非空，随机选择一个度为 k 作为此节点的目标度
4. 如果 $D_{seq}(i)$ 为空，选择一个 $\Delta_+(k)$ 最小的 k 作为此节点的目标度。如果有两个以上不同的度，但 $\Delta_+(k)$ 相同的 k，则选择最小的 k
5. 分配给节点 v_i' 目标度 $d_i^* = k$ 之后，$n'(k)$ 加一
6. 因为 $n'(k)$ 发生变化，为了满足定理 3，需要调整 $n^*(k)$

3.3 阶段二

构建目标联合度矩阵，使其满足下述定理并且最小化目标联合度矩阵和原始图的联合度矩阵估计之间的误差。

1. $m^*(k,k')$ 是非负整数，对任意 $k=1,...,k^*_{max}$ 和 $k'=1,...,k^*_{max}$
2. $m^*(k,k')=m^*(k',k)$，对任意 $k=1,...,k^*_{max}$ 和 $k'=1,...,k^*_{max}$ 满足 $k\neq k'$
3. $\sum_{k'=1}^{k^*_{max}}\mu(k,k')m^*(k,k')=kn^*(k)$，对任意 $k=1,...,k^*_{max}$
4. $m^*(k,k')\geqslant m'(k,k')$，对任意 $k=1,...,k^*_{max}$ 和 $k'=1,...,k^*_{max}$

3.3.1 初始化

对任意 k 和 k' 的度，使用 $\hat{n}$，$\hat{\bar{k}}$ 和 $\hat{P}(k,k')$ 根据等式 $\hat{m}(k,k')=\hat{n}\hat{\bar{k}}\hat{P}(k,k')/\mu(k,k')$ 初始化 $m^*(k,k')$

$$m^*(k,k')=\begin{cases} max(NearInt(\hat{n}\hat{\bar{k}}\hat{P}(k,k')/\mu(k,k')),1) & if\quad \hat{P}(k,k')>0\\ 0 & if\quad \hat{P}(k,k')=0\end{cases}$$

3.3.2 调整

调整 $m^*(k,k')$ 使其满足定理 3，当且仅当 $s(k)$ 不能达到 $s^*(k)$ 的时候，只需要对多个 k' 调整 $m^*(k,k')$ 。调整过程中有三个限制：

- 对于任意 k 和 k'，$m^*(k,k')$ 不能低于输入的最小值 $m_{min}(k,k')$，$m_{min}(k,k')\geqslant 0$ ，该限制能防止破坏定理 1
- 对任意 k 和 k'，如果 $m^*(k,k')$ 增加或降低 1，相应的 $m^*(k,k')$ 也需要增加或降低 1，该限制能防止破坏定理 2
- 当我们调整 $s(k)$ 时，不改变 $s(k')=s^*(k')$ 的 $m^*(k,k')$，该限制能防止破坏已经已经调整好的 $s(k')$

按照 D 中 k 的递减顺序来调整 $s(k)$，原因如下：

- 越迟调整的 $s(k)$，可以调整的 $m^*(k,k')$ 就越少
- 度 k 越小，为了 $s(k)=s^*(k)$ 而添加的边的数量就越小

因此，当我调整 $s(k)$ 的时候，只改变 D 中且 $k'\leqslant k$ 的 k'。当我们调整度为 1 时，首先需要确保 $|s(1)-s^*(1)|$ 是偶数，原因：由于限制 3 和定理 3，调整 $s(1)$ 时，我们只能调整 $m^*(1,1)$。因此，如果 $|s(1)-s^*(1)|$ 是奇数，就没有办法使 $s(1)$ 达到 $s^*(1)$

当 $s(k)<s^*(k)$ 时 此时需要增加 $m^*(k,k')$。对于特定 k，定义可以调整的 k' 集合为 $D'_+(k)$

$$D'_+(k)=\begin{cases}\{k'|k'\in D\bigwedge k'\leqslant k\} & if\quad s(k)\neq s^*(k)-1\\ \{k'|k'\in D\bigwedge k'<k\} & if\quad s(k)=s^*(k)-1\end{cases}$$

独立出 $s(k) = s^*(k) - 1$ 是为了避免 $s(k)$ 增加 2。因为当 $k = k'$ 时，$m^*(k, k')$ 的项数 $\mu(k, k) = 2$。$D'_+(k)$ 至少会包含 $k' = 1$。

$$\Delta_+(k, k') = \begin{cases} \dfrac{|\hat{m}(k, k') - (m^*(k, k') + 1)|}{\hat{m}(k, k')} - \dfrac{|\hat{m}(k, k') - m^*(k, k')|}{\hat{m}(k, k')} & if \quad \hat{P}(k, k') > 0 \\ \infty & if \quad \hat{P}(k, k') = 0 \end{cases}$$

如果有两个以上不同的度 k'，但 $\Delta_+(k, k')$ 相同，则随机选择一个 k'。

当 $s(k) > s^*(k)$ 时 此时需要降低 $m^*(k, k')$，设置极限 $m_{min}(k, k') = 0$。对于特定 k，定义可以调整的 k' 集合为 $D'_-(k)$

$$D'_-(k) = \begin{cases} \{k' | k' \in D \bigwedge k' \leqslant k \bigwedge m^*(k, k') > m_{min}(k, k')\} & if \quad s(k) \neq s^*(k) + 1 \\ \{k' | k' \in D \bigwedge k' < k \bigwedge m^*(k, k') > m_{min}(k, k')\} & if \quad s(k) = s^*(k) + 1 \end{cases}$$

$$\Delta_-(k, k') = \begin{cases} \dfrac{|\hat{m}(k, k') - (m^*(k, k') - 1)|}{\hat{m}(k, k')} - \dfrac{|\hat{m}(k, k') - m^*(k, k')|}{\hat{m}(k, k')} & if \quad \hat{P}(k, k') > 0 \\ \infty & if \quad \hat{P}(k, k') = 0 \end{cases}$$

如果有两个以上不同的度 k'，但 $\Delta_-(k, k')$ 相同，则随机选择一个 k'。$D'_-(k)$ 可能是一个空集，因为 $m_{min}(k, k')$ 的存在。这种情况下，我们提高目标和 $s^*(k)$。

- 如果 $k > 1$，通过 $n^*(k)$ 增加 1 来使 $s^*(k)$ 提高 k
- 如果 $k = 1$，通过 $n^*(1)$ 增加 2 来使 $s^*(1)$ 增加 2，同时要保持 $|s^*(1) - s(1)|$ 是偶数

3.3.3 修改

根据定理 4，如果存在 $m^*(k1, k2) < m'(k1, k2)$，就需要调整 $m^*(k1, k2)$ 使其符合定理 4. 如果对 $m^*(k1, k2) < m'(k1, k2)$ 这一对 $k1$□$k2$ 直接强制满足定理 4，则会破坏多个已经满足定理 3 的 k，而且生成图的目标边总数会剧增。对于任意一对 $k1$ 和 $k2$（$1 \leqslant k1 \leqslant k^*_{max}$，$k1 \leqslant k2 \leqslant k^*_{max}$），均进行以下修改使得满足定理 4

$$D''_-(k) = \left\{1 \leqslant k' \leqslant k^*_{max} \bigwedge k' \neq k \bigwedge m^*(k, k') > m'(k, k')\right\}$$

1. $m^*(k1, k2)$ 增加 1，如果 $k1 \neq k2$，$m^*(k2, k1)$ 也需要增加 1。

2. 如果 $D''_-(k1)$ 不空，从 $D''_-(k1)$ 中找到一个 $k3 \neq k2$ 并且 $\Delta_-(k1, k3)$ 最小的 $k3$，使 $m^*(k1, k3)$ 减少 1，使得 $s(k1)$ 不变，$m^*(k3, k1)$ 也减少 1（当 $k1 \neq k3$）。如果有多个 $\Delta_-(k1, k3)$ 相同的 $k3$ ，则随机选择一个。

3. 因为 $m^*(k2, k1)$ 也增加了 1，所以如果 $D''_-(k2)$ 不空，从 $D''_-(k2)$ 中找到一个 $k4 \neq k2$ 并且 $\Delta_-(k2, k4)$ 最小的 $k4$，使 $m^*(k2, k4)$ 减少 1，使得 $s(k2)$ 不变，$m^*(k4, k2)$ 也减少 1（当 $k2 \neq k4$）。如果有多个 $\Delta_-(k2, k4)$ 相同的 $k4$ ，则随机选择一个。

4. 当找到的 $k3 \neq k4$ 时，$m^*(k3, k4)$□ $m^*(k4, k3)$ 均增加 1，确保 $s(k3)$ 和 $s(k4)$ 不变。

修改 $\{m^*(k, k')\}_{k,k'}$ 不会破坏定理 1 和 2，但可能会破坏定理 3，这时候只需要设置 $m_{min}(k, k') = m'(k, k')$ 再执行一次调整操作。

3.4 阶段三

添加节点和边到子图中，使生成的图满足目标度向量和目标联合度矩阵。如果从空图来生成目标属性的图的话，这个过程是很麻烦的。因此，我们从采样到的子图上进行扩展，生成满足条件的图。

1. 将生成图 $\tilde{G}$ 作为子图 G'
2. 添加 $(\sum_{k=1}^{k^*_{max}} n^*(k)) - n'$ 个节点到子图中
3. 任意分配给 V_{add} 中的节点目标度 k
4. 确保子图中每个节点有 $d_i^* - d_i'$ 条半边
5. 确保添加的节点有 d_i^* 条半边
6. 对任意 k 和 k' 均进行以下程序 $m^*(k,k') - m'(k,k')$ 次：随机连接目标度为 k 和 k' 的节点的自由半边

3.5 阶段四

通过大量的重新布线，使生成图满足目标度依赖聚类系数。

1. 随机选择 $\tilde{\varepsilon}_{rew}$ 中的一对边 $(\tilde{v}_i, \tilde{v}_j)$ 和 $(\tilde{v}_{i'}, \tilde{v}_{j'})$ ，其中 $d(\tilde{v}_i) = d(\tilde{v}_{i'})$
2. 替换 $(\tilde{v}_i, \tilde{v}_j)$ 和 $(\tilde{v}_{i'}, \tilde{v}_{j'})$ 两条边为 $(\tilde{v}_i, \tilde{v}_{j'})$ 和 $(\tilde{v}_{i'}, \tilde{v}_j)$，如果替换后的 D 有所降低，就接受此次替换
3. 如果此次替换被接受，就更新 $\tilde{\varepsilon}_{rew}$
4. 重复 $R = R_C|\tilde{\varepsilon}_{rew}|$ 次

重新布线的步骤不会破坏目标度向量和目标联合度矩阵的属性。

3.6 与已有开源代码对比

复现过程中参考了作者给出的源码 https://github.com/kazuibasou/social-graph-restoration

3.6.1 原图属性估计

对原图结构属性的估计功能，直接使用了作者给出的源码。

节点数量估计 在《Estimating sizes of social networks via biased sampling》[2]这篇论文中提出了通过随机游走的采样序列估计原图的节点数量。估计公式如下:

$$\hat{n} = \frac{\sum_{(i,j)\in I} \dfrac{d_{x_i}}{d_{x_j}}}{\sum_{(i,j)\in I} 1_{\{x_i = x_j\}}}$$

$$I = \left\{(i,j)|M \leqslant |i-j| \bigwedge 1 \leqslant i,j \leqslant r\right\}$$

```
def size_estimator(sampling_list):
   r = len(sampling_list)
   # parameter
   m = int(round(r * 0.025))
   phi = 0.0
   psi = 0.0
   for k in range(0, r - m):
      # 第k个节点的index
      v_k = sampling_list[k].index
```

```
        # 第k个节点的度
        d_k = len(sampling_list[k].nlist)
        for l in range(k + m, r):
            # 第l个节点
            v_l = sampling_list[l].index
            # 第l个节点的度
            d_l = len(sampling_list[l].nlist)
            # 节点k与节点l是同一个节点
            if v_k == v_l:
                # 发生碰撞的节点对，+2是因为分母是2C
                phi += 2
            # 所有采样节点度的和*所有采样节点度的倒数的和
            psi += (float(d_k) / d_l) + (float(d_l) / d_k)

    return float(psi) / phi
```

平均度估计　在《Walking in Facebook: A case study of unbiased sampling of osns》[3]这篇论文中提出了通过随机游走的采样序列估计原图的平均度。估计公式如下：

$$\hat{\bar{k}} = \frac{1}{\bar{\Phi}}$$

$$\bar{\Phi} = \frac{1}{r}\sum_{i=1}^{r}\frac{1}{d_{x_i}}$$

```
def average_degree_estimator(sampling_list):
    # 采样子集中每个节点度的倒数的总和
    est = 0.0
    for data in sampling_list:
        est += float(1) / len(data.nlist)
    return float(len(sampling_list)) / est
```

度分布估计　在《Walking in Facebook: A case study of unbiased sampling of osns》[3]这篇论文中提出了通过随机游走的采样序列估计原图的度分布。估计公式如下：

$$\hat{P}(k) = \frac{\Phi(k)}{\bar{\Phi}}$$

$$\Phi(k) = \frac{1}{kr}\sum_{i=1}^{r} 1_{\{x_i = x_j\}}$$

```
def degree_distribution_estimator(sampling_list):
    # P(k=i) = [度为i的节点度的倒数的总和] / [所有节点度的倒数的总和]
    est = defaultdict(float)
    x = 0
    for data in sampling_list:
        d = len(data.nlist)
        est[d] += float(1) / d
        x += float(1) / d

    for d in est:
        est[d] = float(est[d]) / x

    return est
```

联合度分布估计 在《2.5 k-graphs: from sampling to generation》[4]这篇论文中提出了通过随机游走的采样序列估计原图的联合度分布。估计公式如下：

$$\hat{P}(k,k') = \begin{cases} \hat{P}_{IE}(k,k') & ifk+k' >= 2\hat{\bar{k}} \\ \hat{P}_{TE}(k,k') & ifk+k' < 2\hat{\bar{k}} \end{cases}$$

$$\hat{P}_{IE}(k,k') = \hat{n}\hat{\bar{k}}\Phi(k,k')$$

$$\Phi(k,k') = \frac{1}{kk'|I|}\sum_{(i,j)\in I} 1_{\{d_{x_i}=k \wedge d_{x_j}=k'\}} A_{x_i x_j}$$

$$\hat{P}_{TE}(k,k') = \frac{1}{2(r-1)}\sum_{i-1}^{r-1}(1_{\{d_{x_i}=k \wedge d_{x_j}=k'\}} + 1_{\{d_{x_i}=k' \wedge d_{x_j}=k\}})$$

```
def jdd_estimator_induced_edges(sampling_list, est_n, est_aved):
   # 存放分子，度为k的所有节点和度为l的所有节点之间的边数占比
   phi = defaultdict(lambda: defaultdict(float))
   r = len(sampling_list)
   # 表示 margin 大小
   m = int(round(r * 0.025))

   # 公式中的分子计算
   for i in range(0, r - m):
      v = sampling_list[i].index
      # 节点i的度
      k = len(sampling_list[i].nlist)
      for j in range(i + m, r):
         # 节点j的度
         l = len(sampling_list[j].nlist)
         # 节点j和节点i之间的边数量
         c = sampling_list[j].nlist.count(v)
         # 度为k的节点i和度为l的节点j之间的边数的占比，（疑问：除以k*l是为了消除bais吗？）
         value = float(c) / (k * l)
         phi[k][l] += value
         phi[l][k] += value

   # 联合度分布的估计
   est = defaultdict(lambda: defaultdict(float))
   # 参加计算的 pairs 总数，公式中的分母
   num_sample = (r - m) * (r - m + 1)
   # 原图节点数的估计值，原图平均度的估计值，用来替换公式中的|Vk|*|Vl|
   sum_d = est_n * est_aved
   for k in phi:
      for l in phi[k]:
         value = float(phi[k][l]) / num_sample
         value *= sum_d
         est[k][l] = value

   return est


def jdd_estimator_traversed_edges(sampling_list):
   # 2.5 k-graphs: from sampling to generation, INFOCOM, 2013.
   # The original estimator does not correctly converge to a real value.
   # This is a modified unbiased estimator.

   est = defaultdict(lambda: defaultdict(float))
   r = len(sampling_list)
```

```
    # 按照随机游走采样的顺序
    for i in range(0, r - 1):
        k = len(sampling_list[i].nlist) # 节点i的度
        l = len(sampling_list[i + 1].nlist) # 节点i+1的度
        est[k][l] += 1 # 边数自增
        est[l][k] += 1

    for k in est:
        for l in est[k]:
            # 每个节点会重复算两次，所以要除以2，分母是采样边数（随机游走路过的边数，采样节点数
                -1)
            est[k][l] = float(est[k][l]) / (2 * (r - 1))

    return est


def JDD_estimator_hybrid(sampling_list, est_n, est_aved):
    # 2.5 k-graphs: from sampling to generation, INFOCOM, 2013.
    # The original estimator does not correctly converge to a real value.
    # This is a modified unbiased estimator.

    est_jdd_ie = jdd_estimator_induced_edges(sampling_list, est_n, est_aved)
    est_jdd_te = jdd_estimator_traversed_edges(sampling_list)

    est_jdd = defaultdict(lambda: defaultdict(float))

    for k in est_jdd_ie:
        for l in est_jdd_ie[k]:
            if (k + l) >= 2 * est_aved:
                est_jdd[k][l] = est_jdd_ie[k][l]
                est_jdd[l][k] = est_jdd_ie[l][k]

    for k in est_jdd_te:
        for l in est_jdd_te[k]:
            if (k + l) < 2 * est_aved:
                est_jdd[k][l] = est_jdd_te[k][l]
                est_jdd[l][k] = est_jdd_te[l][k]

    return est_jdd
```

度依赖聚类系数 在《Estimating clustering coefficients and size of social networks via random walk》[5]这篇论文中提出了通过随机游走的采样序列估计原图的度依赖聚类系数。估计公式如下：

$$\Phi_{\bar{c}}(k) = \frac{1}{(k-1)(r-2)} \sum_{i=2}^{r-1} 1_{\left\{d_{x_i}=k\right\}} A_{x_{i-1}x_{i+1}}$$

```
def degree_dependent_clustering_coefficient_estimator(sampling_list):
    # 度依赖聚类系数(平均聚类系数)：度为k的所有节点的聚类系数总和

    # fai
    phi = defaultdict(float)
    # 坡赛
    psi = defaultdict(float)
    r = len(sampling_list)

    for i in range(0, r):
        # 随机游走采样序列第i个节点的度
        d = len(sampling_list[i].nlist)
        psi[d] += float(1) / d
    # 第i-1个节点与第i+1个节点之间的边数
    for i in range(1, r - 1):
        d = len(sampling_list[i].nlist)
        # 如果第i个节点的度小于2的话，就一定不会有三角形
        if d == 0 or d == 1:
            continue

        # 采样到的第i-1个节点的index
        s = sampling_list[i - 1].index
        # 第i个节点的index
        v = sampling_list[i].index
        # 第i+1个节点的index
        t = sampling_list[i + 1].index

        # 连续采样的三个节点需要不同，才可能存在一这三个节点为顶点的三角形
        if s != v and v != t and t != s:
            # 节点s与节点t之间的边数
            c = sampling_list[i + 1].nlist.count(s)
            # f(v) = 1 / (d - 1)
            phi[d] += float(c) / (d - 1)

    for d in phi:
        phi[d] = float(phi[d]) / (r - 2)

    for d in psi:
        psi[d] = float(psi[d]) / r

    est_ddcc = defaultdict(float)
    for d in psi:
        est_ddcc[d] = float(phi[d]) / psi[d]

    return est_ddcc
```

3.6.2 阶段一核心算法

Procedure 1 Adjust the target degree vector to ensure that it satisfies condition (DV-2)

```
Input: Estimates: $\hat{n}$ and $\left\{\hat{P}(k)\right\}_k$
Input: Target maximum degree:$k^*_{max}$
Input: Target degree vector:$\{n^*(k)\}_k$
if $\sum_{k=1}^{k^*_{max}} kn^*(k)$ is an odd number then
    Select degree k such that k is an odd number and $\Delta_+(k)$ is the smallest
    $n(k) \leftarrow n(k) + 1$
end
return $k^*_{max}$
```

该算法参考了源码，在算法理解的基础上，自己重新实现了一遍。

```
# 调整目标度向量
def adjust_target_degree_vector(est_n, est_dd, tgt_degree_vector):
    # 满足 DV-2 边数应该是偶数

    # 求边的总数
    sum_deg = sum([k * tgt_degree_vector[k] for k in tgt_degree_vector])
    # 如果已经满足 DV-2 就不需要再调整了
    if sum_deg % 2 == 0:
        return tgt_degree_vector
```

```
degree_candidates = {}
for k in tgt_degree_vector.keys():
   # 选择需要调整的度应该为奇数
   if k % 2 == 0:
      continue
   # 度为k的节点数（估计值）
   est_k_num = est_dd[k] * est_n
   # 度为k的节点数（目标值）
   tgt_k_num = tgt_degree_vector[k]

   # 度为k的概率如果不为0，则计算提升度为k的节点数量之后的error
   if est_k_num != 0:
      delta_e = float(math.fabs(est_k_num - tgt_k_num - 1)) / est_k_num -
         float(math.fabs(est_k_num - tgt_k_num)) / est_k_num
   # 度为k的概率如果不为0，error无穷大
   else:
      delta_e = float("inf")

   # 提升度为k的节点数的候选这的误差
   degree_candidates[k] = delta_e

# 存在可以提升度为k的节点数量的候选者
if len(degree_candidates) > 0:
   # 选择error最小的候选者度k
   condidate_degree = select_min_key_with_smallest_value(degree_candidates)
   # 度为k的目标数量加一
   tgt_degree_vector[condidate_degree] += 1
else:
   # 不存在奇数的候选者，则就让度为1的节点数增加一
   tgt_degree_vector[1] += 1

return tgt_degree_vector
```

```
**Procedure 2** Modify the target degree vector to ensure that it satisfies condition (DV-3)
**Input:** Subgraph: $G' = (v', E')$
**Input:** Estimates: $\hat{n}$ and $\left\{\hat{P}(k)\right\}_k$
**Input:** Target maximum degree: $k^*_{max}$
**Input:** Target degree vector: $\{n^*(k)\}_k$
Calculate degree $d'_i$ of each node in the subgraph $v'_i \in V'$.
for each $v'_i \in V'_{qry}$ in arbitrary order do
    $d^*_i \leftarrow d'_i$
end
Calculate the present $n'(k)$ for each $k = 1, ..., k^*_{max}$.
for $k = 1, ..., k^*_{max}$ do
    $n^*(k) \leftarrow max(n^*(k), n'(k))$
end
for each $v'_i \in V'_{vis}$ in decreasing order of $d'_i$ do
    Construct the degree sequence $D_{seq}(i)$.
    if $D_{seq}(i)$ is not empty then
        Select degree $k$ uniformly randomly from $D_{seq}(i)$.
    end
    else
        Select degree $k$ such that $d'_i \leqslant k \leqslant k^*_{max}$ and $\Delta_+(k)$ is the smallest.
    end
    $d^*_i \leftarrow k$.
    $n'(k) \leftarrow n'(k) + 1$.
    $n^*(k) \leftarrow max(n^*(k), n'(k))$
end
return $\{n^*(k)\}_k$
```

该算法参考了源码，在算法理解的基础上，自己重新实现了一遍。

```python
# 修改目标度向量
def modify_target_degree_vector(subG: graph.Graph, est_n, est_dd,
    tgt_degree_vector):
    # 分配给每个节点目标度
    # 满足 DV-3 目标度向量不小于子图的度向量

    # 子图（生成图）的度向量
    subG_degree_vector = defaultdict(int)
    # 存放目标节点的度
    tgt_node_degree = {}
    # 遍历查询节点集合
    for v in subG.qry_nodes:
        # 节点v在子图中的度
        subG_degree = len(subG.nlist[v])
        # {节点: 度}
        tgt_node_degree[v] = subG_degree
        # 更新子图的度向量
        subG_degree_vector[subG_degree] += 1

    # 调整查询节点的度的节点数量
    for degree in subG_degree_vector:
        # 如果查询节点的度的目标数量小于子图（加入查询节点）中该度的数量
        if tgt_degree_vector[degree] < subG_degree_vector[degree]:
            # 增大目标度向量到与子图该度相同的数量
            tgt_degree_vector[degree] = subG_degree_vector[degree]

    # [[邻居可访问节点, 该节点的度],....]
    visible_node_pairs = []
```

```
# 遍历邻居可访问节点集合
for v in subG.vis_nodes:
    visible_node_pairs.append([v, len(subG.nlist[v])])
# 排序，度小的在前面，度相同时，节点index大的在前
visible_node_pairs.sort(key=cmp_to_key(cmp))

for visible_node_pair in visible_node_pairs:
    # 节点v
    v = visible_node_pair[0]
    # 节点v在子图（当前生成图）中的度
    subG_degree = visible_node_pair[1]

    # 候选度，当前还可以分配的度
    degree_candidates = []
    # 统计还可以分配给节点的度
    for k in tgt_degree_vector:
        # 从目标度向量中找到一个不小于节点v的度的度k，并且这个度k的目标节点数量要大于子图
            （当前生成图）这个度k的节点数量
        if k >= subG_degree and tgt_degree_vector[k] > subG_degree_vector[k]:
            # 可以存入（度k的目标数量-当前度k的数量）个度k到候选度list中
            for i in range(0, tgt_degree_vector[k] - subG_degree_vector[k]):
                degree_candidates.append(k)

    # 存在可分配的度
    if len(degree_candidates) > 0:
        # 从候选度list中随机选择一个度赋值给节点v
        tgt_node_degree[v] = np.random.choice(list(degree_candidates))
    # 不存在可分配的度
    else:
        # {度k: error}
        degree_to_add_candidates = {}
        # 从度分布估计中找
        for k in est_dd:
            # 找不小于节点v的度的度k
            if k < subG_degree:
                continue
            # 度k的节点数量的估计
            est_k_num = est_dd[k] * est_n
            # 度k的目标数量
            tgt_k_num = float(tgt_degree_vector[k])
            # 增加度k的目标数量之后的error
            if est_k_num != 0:
                delta_e = float(math.fabs(est_k_num - tgt_k_num - 1)) /
                    est_k_num - float(math.fabs(est_k_num - tgt_k_num)) /
                    est_k_num
            else:
                delta_e = float("inf")
            # 添加到候选度中
            degree_to_add_candidates[k] = delta_e

        # 存在可增加数量的度k
        if len(degree_to_add_candidates) > 0:
            # 选择增加度k节点数量之后error最小的度k，分配给节点v
            tgt_node_degree[v] = select_min_key_with_smallest_value(
                degree_to_add_candidates)
        # 不存在
        else:
            # 保持不变，节点v的度为子图（当前生成图）中的度
            tgt_node_degree[v] = subG_degree

        # 增加目标度k的节点数量+1
```

```
            tgt_degree_vector[tgt_node_degree[v]] += 1
        # 再+1
        subG_degree_vector[tgt_node_degree[v]] += 1

    # 改变目标度向量之后，需要对目标度向量进行重新调整
    tgt_degree_vector = adjust_target_degree_vector(est_n, est_dd,
        tgt_degree_vector)

    return [subG_degree_vector, tgt_degree_vector, tgt_node_degree]
```

3.6.3 阶段二核心算法

Procedure 3 Adjust the target joint degree matrix to ensure that it satisfies condition (JDM-3)

```
Input: Estimates: $\hat{n}$,$\hat{\bar{k}}$,and $\left\{\hat{P}(k,k')\right\}_{k,k'}$
Input: Target maximum degree: $k^*_{max}$
Input: Target degree vector: $\{n^*(k)\}_k$
Input: Target joint degree matrix: $\{m^*(k,k')\}_{k,k'}$
Input: Lower limits: $\{m_{min}(k,k')\}_{k,k'}$
for each $k \in D$ in decreasing order of $k$ do
    if $k = 1$ and $|s(1) - s^*(1)|$ is an odd number then
        $n^*(1) \leftarrow n^*(1) + 1$.
    end
    while $s(k) \neq s^*(k)$ do
        if $s(k) < s^*(k)$ then
            Select degree $k' \in D'_+(k)$ with the smallest $\Delta_+(k,k')$.
            $m^*(k,k') \leftarrow m^*(k,k') + 1$.
            if $k \neq k'$ then
                $m^*(k',k) \leftarrow m^*(k',k) + 1$.
            end
        end
        else
            if $D'\_(k)$ is not empty then
                Select degree $k' \in D'\_(k)$ with the smallest $\Delta_{(k,k')}$.
                $m^*(k,k') \leftarrow m^*(k,k') - 1$.
                if $k \neq k'$ then
                    $m^*(k',k) \leftarrow m^*(k',k) - 1$.
                end
            end
            else
                if $k = 1$ then
                    $n^*(1) \leftarrow n^*(1) + 2$.
                end
                else
                    $n^*(1) \leftarrow n^*(1) + 1$.
                end
            end
        end
    end
end
return $\{m^*(k,k')\}_{k,k'}$
```

该算法参考了源码，在算法理解的基础上，自己重新实现了一遍。

```
# 调整目标联合度矩阵
def adjust_target_joint_degree_matrix(est_n, est_aved, est_jdd,
    tgt_degree_vector, min_joint_degree_matrix, tgt_joint_degree_matrix):
    # 满足 JDM-3：目标联合度矩阵中所有度为k相关的边数总和=k*目标度k的数量

    # 目标度向量中值不为0的key集合
    degree_k1_set = set(tgt_degree_vector.keys())
    # 如果度为1不存在，则添加到key集合中
    if 1 not in degree_k1_set:
        degree_k1_set.add(1)
    # 对目标度向量中的key（度）进行排序，度大的放在前面
    degree_k1_set = sorted(list(degree_k1_set), reverse=True)

    # 从大到小遍历度k1
    for k1 in degree_k1_set:
        # 目标边总数，JDM-3 等式右侧
        target_sum = k1 * tgt_degree_vector[k1]
        # 当前边总数，JDM-3 等式左侧
        present_sum = sum(tgt_joint_degree_matrix[k1].values())
        # 差距
        different = target_sum - present_sum

        # JDM-3 条件满足，则无需调整
        if different == 0:
            continue

        # 小于等于度k1的度k2集合
        degree_k2_set = set([k2 for k2 in degree_k1_set if k2 <= k1])

        # 如果度k1是1，并且差距是奇数，就没有办法通过有限调整使得JDM-3成立
        if k1 == 1 and abs(target_sum - present_sum) % 2 != 0:
            # 度为1的目标向量增加一
            tgt_degree_vector[1] += 1
            # 目标总数+1
            target_sum += 1

        # 一直调整到JDM-3成立
        while target_sum != present_sum:
            # 如果当前比目标小，需要增大当前边的总数
            if target_sum > present_sum:
                # 可以用来调整的k2集合
                degree_k2_candidate = {}
                for k2 in degree_k2_set:
                    # 如果差距为一，没有办法通过调整m(k,k)使差距为0
                    if present_sum == target_sum - 1 and k2 == k1:
                        continue

                    # 联合度k的估计，k‘之间的边数=联合度分布估计*节点总数估计*平均度估计
                    est_k1_k2_edges_num = est_jdd[k1][k2] * est_n * est_aved
                    tgt_k1_k2_edges_num = float(tgt_joint_degree_matrix[k1][k2])

                    # 联合度k，k’概率估计为0
                    if est_k1_k2_edges_num == 0:
                        # error设为无穷
                        delta_e = float("inf")
                    else:
                        # 如果不是同度，计算将目标联合度m(k1,k2)增加1后的误差
                        if k2 != k1:
                            delta_e = float(math.fabs(est_k1_k2_edges_num -
                                tgt_k1_k2_edges_num - 1)) / est_k1_k2_edges_num - float
                                (math.fabs(est_k1_k2_edges_num - tgt_k1_k2_edges_num))
                                / est_k1_k2_edges_num
                        # 如果同度，计算将目标联合度m(k,k)增加2后的误差
                        else:
                            delta_e = float(math.fabs(est_k1_k2_edges_num -
                                tgt_k1_k2_edges_num - 2)) / est_k1_k2_edges_num - float
                                (math.fabs(est_k1_k2_edges_num - tgt_k1_k2_edges_num))
                                / est_k1_k2_edges_num
                    # 保存修改k1与k2的目标联合度的误差，k2候选集合
                    degree_k2_candidate[k2] = delta_e

                # 选择一个误差最小的k2，如果存在多个误差相同的k2，就随机选择一个k2
                k2 = select_random_key_with_smallest_value(degree_k2_candidate)

                # 修改相应的目标联合度矩阵
                tgt_joint_degree_matrix[k1][k2] += 1
                tgt_joint_degree_matrix[k2][k1] += 1

                # 修改当前边总数，如果k2与k1相同边数是增加2，不相同是增加1
                if k1 != k2:
                    present_sum += 1
                else:
                    present_sum += 2

            # 如果当前比目标大，需要减小当前边的总数
            else:
                degree_k2_candidate = {}
                for k2 in degree_k2_set:
                    # 因为需要降低目标联合度，所以可能会发生降低后的目标联合度低于下限
                    if tgt_joint_degree_matrix[k1][k2] <= min_joint_degree_matrix[k1
                        ][k2]:
                        continue
                    if present_sum == target_sum + 1 and k2 == k1:
                        continue

                    est_k1_k2_edges_num = est_jdd[k1][k2] * est_n * est_aved
                    tgt_k1_k2_edges_num = float(tgt_joint_degree_matrix[k1][k2])

                    if est_k1_k2_edges_num == 0:
                        delta_e = float("inf")
                    else:
                        if k2 != k1:
                            delta_e = float(math.fabs(est_k1_k2_edges_num -
                                tgt_k1_k2_edges_num + 1)) / est_k1_k2_edges_num - float
                                (math.fabs(est_k1_k2_edges_num - tgt_k1_k2_edges_num))
                                / est_k1_k2_edges_num
                        else:
                            delta_e = float(math.fabs(est_k1_k2_edges_num -
                                tgt_k1_k2_edges_num + 2)) / est_k1_k2_edges_num - float
                                (math.fabs(est_k1_k2_edges_num - tgt_k1_k2_edges_num))
                                / est_k1_k2_edges_num

                    degree_k2_candidate[k2] = delta_e

                if len(degree_k2_candidate) > 0:
                    k2 = select_random_key_with_smallest_value(degree_k2_candidate)
                    tgt_joint_degree_matrix[k1][k2] -= 1
                    tgt_joint_degree_matrix[k2][k1] -= 1

                    if k1 != k2:
                        present_sum -= 1
                    else:
                            present_sum -= 2
                      # 如果不存在可以调整的k2
                      else:
                         if k1 > 1:
                            target_sum += k1
                            tgt_degree_vector[k1] += 1
                         else:
                            target_sum += 2
                            tgt_degree_vector[1] += 2

               return [tgt_joint_degree_matrix, tgt_degree_vector]
```

Procedure 4 Modify the target joint degree matrix to ensure that it satisfies condition (JDM-4)

```
Input: Subgraph: $G' = (v', E')$
Input: Estimates: $\hat{n}$,$\hat{\hat{k}}$,and $\left\{\hat{P}(k,k')\right\}_{k,k'}$
Input: Target maximum degree: $k^*_{max}$
Input: Target degree of each node in the subgraph: $\{d^*_i\}_{v_i \in V^*}$
Input: Target joint degree matrix: $\{m^*(k,k')\}_{k,k'}$
Calculate $m'(k,k')$ for each $k = 1,...,k^*_{max}$ and $k' = 1,...,k^*_{max}$.
for $k_1 = 1,...,k^*_{max}$ do
    for $k_2 = k_1,...,k^*_{max}$ do
        while $m^*(k_1,k_2) < m'(k_1,k_2)$ do
            $m^*(k_1,k_2) \leftarrow m^*(k_1,k_2) + 1$.
            if $k_1 \neq k_2$ then
                $m^*(k_2,k_1) \leftarrow m^*(k_2,k_1) + 1$
            end
            if $D''_-(k_1)$ is not empty then
                Select degree $k_3 \in D''_-(k_1)$ with the smallest $\Delta_{(k_1,k_3)}$.
                $m^*(k_1,k_3) \leftarrow m^*(k_1,k_3) - 1$.
                if $k_1 \neq k_3$ then
                    $m^*(k_3,k_1) \leftarrow m^*(k_3,k_1) - 1$
                end
            end
            if $D''_-(k_2)$ is not empty then
                Select degree $k_4 \in D''_-(k_2)$ with the smallest $\Delta_{(k_2,k_4)}$.
                $m^*(k_2,k_4) \leftarrow m^*(k_2,k_4) - 1$.
                if $k_2 \neq k_4$ then
                    $m^*(k_4,k_2) \leftarrow m^*(k_4,k_2) - 1$
                end
            end
            if both degrees $k_3$ and $k_4$ have been found then
                $m^*(k_3,k_4) \leftarrow m^*(k_3,k_4) + 1$.
                if $k_3 \neq k_4$ then
                    $m^*(k_4,k_3) \leftarrow m^*(k_4,k_3) + 1$
                end
            end
        end
    end
end
return $\{m^*(k,k')\}_{k,k'}$
```

该算法参考了源码，在算法理解的基础上，自己重新实现了一遍。

```
# 修改目标联合度矩阵
```

```
def modify_target_joint_degree_matrix(subG: graph.Graph, est_n, est_aved,
   est_jdd, tgt_node_degree, tgt_degree_vector, tgt_joint_degree_matrix):
   # 满足 JDM-4：目标联合度都要不小于对应的估计值

   # 目标度向量中值不为0的key集合
   degree_set = set(tgt_degree_vector.keys())
   # 保证度集合中包含度为1
   if 1 not in degree_set:
      degree_set.add(1)
   # 对度k从小到大排列
   degree_set = set(sorted(list(degree_set)))

   # 初始化当前子图的联合度矩阵
   subG_joint_degree_matrix = defaultdict(lambda: defaultdict(int))
   # 遍历子图的节点集，得到当前子图的联合度矩阵
   for v in subG.nodes:
      # 节点v的目标度
      k1 = tgt_node_degree[v]
      # 遍历节点v的邻居
      for w in subG.nlist[v]:
         # 遍历节点v的邻居
         k2 = tgt_node_degree[w]
         # 度为k1的顶点v和度为k2的顶点w之间存在一条边
         subG_joint_degree_matrix[k1][k2] += 1


   # 遍历子图的联合度矩阵
   for k1 in subG_joint_degree_matrix:
      for k2 in subG_joint_degree_matrix[k1]:
         # 不满足 JDM-4 的时候
         while subG_joint_degree_matrix[k1][k2] > tgt_joint_degree_matrix[k1][
            k2]:
            # 目标联合度m(k1,k2)、m(k2,k1)都增加一
            tgt_joint_degree_matrix[k1][k2] += 1
            tgt_joint_degree_matrix[k2][k1] += 1

            # k3候选集，因为k2的联合度变大了，所以需要找一个k3使k2联合度变小
            degree_k3_candidates = {}
            for k3 in degree_set:
               # 找到一个不等于k1的k3，并且m(k2,k3)大于当前子图对应的联合度
               if k3 == k1 or tgt_joint_degree_matrix[k2][k3] <=
                  subG_joint_degree_matrix.get(k2, {}).get(k3, 0):
                  continue

               est_k2_k3_edges_num = est_jdd[k2][k3] * est_n * est_aved
               tgt_k2_k3_edges_num = tgt_joint_degree_matrix[k2][k3]

               if est_k2_k3_edges_num == 0:
                  delta_e = float("inf")
               else:
                  if k2 != k3:
                     delta_e = float(math.fabs(est_k2_k3_edges_num -
                        tgt_k2_k3_edges_num + 1)) / est_k2_k3_edges_num - float
                        (math.fabs(est_k2_k3_edges_num - tgt_k2_k3_edges_num))
                        / est_k2_k3_edges_num
                  else:
                     delta_e = float(math.fabs(est_k2_k3_edges_num -
                        tgt_k2_k3_edges_num + 2)) / est_k2_k3_edges_num - float
                        (math.fabs(est_k2_k3_edges_num - tgt_k2_k3_edges_num))
                        / est_k2_k3_edges_num
               degree_k3_candidates[k3] = delta_e

            k3 = -1
            # 如果存在k3，选择一个error最小的k3进行修改
            if len(degree_k3_candidates) > 0:
               k3 = select_random_key_with_smallest_value(degree_k3_candidates)
               tgt_joint_degree_matrix[k2][k3] -= 1
               tgt_joint_degree_matrix[k3][k2] -= 1

            # k2的联合度虽然通过k3使其保持不变，但是k3的联合度变大了，此时k1联合度也是变大
               的
            degree_k4_candidates = {}
            for k4 in degree_set:
               if k4 == k2 or tgt_joint_degree_matrix[k1][k4] <=
                  subG_joint_degree_matrix.get(k1, {}).get(k4, 0):
                  continue

               est_k1_k4_edges_num = est_jdd[k1][k4] * est_n * est_aved
               tgt_k1_k4_edges_num = tgt_joint_degree_matrix[k1][k4]

               if est_k1_k4_edges_num == 0:
                  delta_e = float("inf")
               else:
                  if k1 != k4:
                     delta_e = float(math.fabs(est_k1_k4_edges_num -
                        tgt_k1_k4_edges_num + 1)) / est_k1_k4_edges_num - float
                        (math.fabs(est_k1_k4_edges_num - tgt_k1_k4_edges_num))
                        / est_k1_k4_edges_num
                  else:
                     delta_e = float(math.fabs(est_k1_k4_edges_num -
                        tgt_k1_k4_edges_num + 2)) / est_k1_k4_edges_num - float
                        (math.fabs(est_k1_k4_edges_num - tgt_k1_k4_edges_num))
                        / est_k1_k4_edges_num

               degree_k3_candidates[k4] = delta_e

            # 如果存在k4，选择一个error最小的k3进行修改
            if len(degree_k4_candidates) > 0:
               k4 = select_random_key_with_smallest_value(degree_k4_candidates)
               tgt_joint_degree_matrix[k4][k1] -= 1
               tgt_joint_degree_matrix[k1][k4] -= 1

               if k3 > 0:
                  tgt_joint_degree_matrix[k3][k4] += 1
                  tgt_joint_degree_matrix[k4][k3] += 1

   # 修改目标联合度矩阵之后，可能会打破JDM-1，JDK-2的条件，所以需要重新调整一次
   [tgt_jnt_degree_matrix, tgt_degree_vector] =
      adjust_target_joint_degree_matrix(est_n, est_aved, est_jdd,
      tgt_degree_vector, subG_joint_degree_matrix, tgt_joint_degree_matrix)

   return [subG_joint_degree_matrix, tgt_jnt_degree_matrix, tgt_degree_vector]
```

3.6.4 阶段三核心算法

Procedure 5 Construct a graph that preserves the target degree vector and the target joint degree matrix from the subgraph

```
Input: Subgraph: $G'$
Input: Target degree vector:$\{n^*(k)\}_k$
Input: Target joint degree matrix: $\{m^*(k,k')\}_{k,k'}$
$\tilde{G} \leftarrow G'$.
 Add $(\sum_{k=1}^{k^*_{max}} n^*(k)) - n'$ nodes to a set of nodes in $\tilde{G}$
 Construct the degree sequeue $D_{seq}$ in which degree $k$ appears $n^*(k) - n'(k)$ times for $k = 1,..,k^*_{max}$.
 Randomly shuffle $D_{seq}$.
 for each added node $\tilde{v}_i \in V_{add}$ in arbitrary order do
     $k \leftarrow$ the last element in $D_{seq}$.
     Remove the last element in $D_{seq}$.
     $d^*_i \leftarrow k$.
end
for each node $\tilde{v}_i \in V_{qry} \cup V_{vis}$ do
    Attach $d^*_i - d'_i$ half-edges to $\tilde{v}_i$.
end
for each node $\tilde{v}_i \in V_{add}$ do
    Attach $d^*_i$ half-edges to $\tilde{v}_i$.
end
for $k = 1, ..., k^*_{nax}$ do
    for $k' = k, ..., k^*_{max}$ do
        for $i = 1$ to $m^*(k,k') - m'(k,k')$ do
            Uniformly and randomly select a free half-edge of the nodes with the target degree $k$ and a free
              halfedge of the nodes with the target degree $k'$ and connect them.
        end
    end
end
return $\tilde{G}$
```

该算法参考了源码，在算法理解的基础上，自己重新实现了一遍。

```
# 对子图进行构造，使得结构满足目标度向量和目标联合度矩阵
def reconstruct_genG_with_two_tgt(genG: graph.Graph, tgt_degree_vector,
    subG_degree_vector, tgt_node_degree, tgt_joint_degree_matrix,
    subG_joint_degree_matrix):
    # (4-1) 决定目标节点数量
    # 目标节点数
    tgt_N = sum(list(tgt_degree_vector.values()))
    # 当前子图的节点数
    subG_N = len(genG.nodes)
    # 添加节点，使当前子图的节点满足目标节点数
    for v in range(subG_N, tgt_N):
        genG.nodes.add(v)

    # (4-2) 给添加进来的节点分配目标度
    # 还可以分配的度
    degree_seq = []
    for d in tgt_degree_vector:
        for i in range(0, tgt_degree_vector[d] - subG_degree_vector[d]):
            degree_seq.append(d)

    # 当前子图的度向量副本
    cur_degree_vector = defaultdict(int)
    for d in tgt_degree_vector:
        cur_degree_vector[d] = subG_degree_vector[d]

    # 打乱可以分配的度
    random.shuffle(degree_seq)
    # 给刚添加的节点分配度
    for v in range(subG_N, tgt_N):
        # 取出可分配的第一个度d
        d = degree_seq.pop()
        # 分配给节点v
        tgt_node_degree[v] = d
        # 当前度d数量加一
        cur_degree_vector[d] += 1

    # (4-3) 统计自由边
    # 存放度为k的节点v还可以连出多少条边（自由边）
    free_edges = defaultdict(list)
    for v in genG.nodes:
        # 节点v的度
        tgt_degree = tgt_node_degree[v]
        # 节点v当前子图中的度
        subG_degree = len(genG.nlist[v])
        for i in range(0, tgt_degree - subG_degree):
            free_edges[tgt_degree].append(v)

    # 打乱自由边
    for degree in free_edges:
        random.shuffle(free_edges[degree])

    # (4-4) 随机连接度为k1的节点和度为k2的节点的自由边
    # 存档当前子图的联合度矩阵副本
    cur_joint_degree_matrix = defaultdict(lambda: defaultdict(int))
    for k1 in tgt_joint_degree_matrix:
        for k2 in tgt_joint_degree_matrix[k1]:
            cur_joint_degree_matrix[k1][k2] = subG_joint_degree_matrix[k1][k2]

    for k1 in tgt_joint_degree_matrix:
        for k2 in tgt_joint_degree_matrix[k1]:
            # 当前子图的联合度矩阵与目标联合度矩阵不等时
            while cur_joint_degree_matrix[k1][k2] != tgt_joint_degree_matrix[k1][
                k2]:
                # 从度为k1的自由边集合中选择一条边u
                u = free_edges[k1].pop()
                # 从度为k2的自由边集合中选择一条边v
                v = free_edges[k2].pop()
                # 将节点u与节点v加入到子图中
                graph.add_edge(genG, u, v)
                # 更新当前联合度矩阵
                cur_joint_degree_matrix[k1][k2] += 1
                cur_joint_degree_matrix[k2][k1] += 1

    return genG
```

3.6.5 阶段四核心算法

Procedure 6 Rewire edges to ensure that the generated graph preserves the estimate of the degree-dependent clustering coefficient

```
Input: Generated graph: $\tilde{G} = (\tilde{V}, \tilde{E})$
Input: Estimate: $\left\{\hat{c}(k)\right\}_k$
Input: Coefficient of the number of rewiring attempts: $R_C$
$\tilde{E}_{rew} \leftarrow$ a set of candidate edges to be rewired in $\tilde{G}$.
 $R \leftarrow R_C|\tilde{E}_{rew}|$.
 $\left\{\tilde{c}(k)\right\}_k \leftarrow$ the present degree-dependent clustering coefficient of $\tilde{G}$.
 $D \leftarrow L^1$ distance between $\{\tilde{c}(k)\}_k$ and $\left\{\hat{c}(k)\right\}_k$ for $r' = 1$ to $R$ do
    $(\tilde{v}_i, \tilde{v}_j), (\tilde{v}_a, \tilde{v}_b) \leftarrow$ random edge pair in $\tilde{E}_{rew}$.
     $\{\tilde{\tilde{c}}_{rew}(k)\}_k \leftarrow$ degree-dependent clustering coefficient when the selected edge pair is rewired.
     $D_{rew} \leftarrow L^1$ distance between $\{\tilde{\tilde{c}}_{rew}(k)\}_k$ and $\left\{\hat{c}_{rew}(k)\right\}_k \leftarrow$.
     if $D_{rew} < D$ then
        Remove edges $(\tilde{v}_i, \tilde{v}_j)$ and $(\tilde{v}_a, \tilde{v}_b)$.
         Add edges $(\tilde{v}_i, \tilde{v}_b)$ and $(\tilde{v}_a, \tilde{v}_j)$.
         Update $\tilde{E}_{rew}$.
         $D \leftarrow D_{rew}$
    end
end
return $\tilde{G}$
```

此部分完全借鉴源码。

```python
# 调整边的连接，使得更接近度依赖聚类系数的估计
def targeting_rewiring_for_clustering(genG: graph.Graph, tgt_ddcc,
    rewirable_edges, R_C=500):
    # 存放节点度（度要大于1）的字典:{节点v: 节点v的度}
    node_degree = defaultdict(int)
    # 存放度k的数量:{度k: 数量n}
    degree_number = defaultdict(int)
    # 遍历所有生成图中的所有节点
    for v in genG.nodes:
        # 节点v的度
        degree = len(genG.nlist[v])
        # 如果度大于1
        if degree > 1:
            node_degree[v] = degree
            degree_number[degree] += 1

    # 存放聚类系数相关的常量 2/[d*(d-1)] / n_d
    const_coeff = defaultdict(float)

    for degree in degree_number:
        # 度大于1才会有三角形（聚类系数）
        if degree > 1:
            const_coeff[degree] = float(2) / (degree * (degree - 1))
            const_coeff[degree] = float(const_coeff[degree]) / degree_number[
                degree]

    # 计算图的度依赖聚类系数和平均聚类系数
    graph.calc_clustering(genG)
    # 当前生成图的度依赖聚类系数的副本
    cur_ddcc = genG.ddcc.copy()

    # 计算当前的度依赖聚类系数的差距
```

```
[distance, normalization] = calc_L1_distance(tgt_ddcc, cur_ddcc)

# 调整次数
R = R_C * len(rewirable_edges)

for r in range(0, R):
   # 当前度依赖聚类系数的副本
   rewired_ddcc = cur_ddcc.copy()
   # 上个阶段修订完后的差距
   rewired_distance = distance

   # 修订边，同时保持联合度矩阵属性，num_tri_to_add 修订后度三角形数量变化，
      rewiring_case用于判断那种修订
   [num_tri_to_add, i_e1, i_e2, rewiring_case] =
      rewiring_random_edge_pair_preserving_joint_degree_matrix(genG,
      node_degree, rewirable_edges)

   # 重新计算生成图的聚类系数
   for degree in num_tri_to_add:
      # 度小于等一1肯定不会有三角形
      if degree > 1:
         # 修改当前度依赖聚类系数的副本
         rewired_ddcc[degree] += float(num_tri_to_add[degree] * const_coeff[
            degree])
         # 修订后的差距
         rewired_distance += math.fabs(tgt_ddcc[degree] - rewired_ddcc[
            degree]) - math.fabs(tgt_ddcc[degree] - cur_ddcc[degree])

   # 修订后的差距与修订前的差距的变化
   delta_dist = rewired_distance - distance

   # 变化没有变小，此次修订失效
   if delta_dist >= 0:
      continue

   # 以下是修订成功后生成图更新相应数据
   u = rewirable_edges[i_e1][0]
   v = rewirable_edges[i_e1][1]
   x = rewirable_edges[i_e2][0]
   y = rewirable_edges[i_e2][1]

   # (u,v),(x,y) 换 (u,y),(v,x)
   if rewiring_case == 0:
      graph.remove_edge(genG, u, v)
      graph.remove_edge(genG, x, y)
      graph.add_edge(genG, u, y)
      graph.add_edge(genG, v, x)

      tmp = rewirable_edges[i_e1][1]
      rewirable_edges[i_e1][1] = rewirable_edges[i_e2][1]
      rewirable_edges[i_e2][1] = tmp


   # (u,v),(x,y) 换 (u,x),(v,y)
   elif rewiring_case == 1:
      graph.remove_edge(genG, u, v)
      graph.remove_edge(genG, x, y)
      graph.add_edge(genG, u, x)
      graph.add_edge(genG, v, y)

      tmp = rewirable_edges[i_e1][1]
```

```
            rewirable_edges[i_e1][1] = rewirable_edges[i_e2][0]
            rewirable_edges[i_e2][0] = tmp

        cur_ddcc = rewired_ddcc.copy()
        distance = rewired_distance

    return genG
```

3.6.6 采样算法与评估指标

此部位源码中未给出，由自己独立实现。

```
import numpy as np
from littleballoffur import RandomWalkSampler, SnowBallSampler,
    ForestFireSampler, BreadthFirstSearchSampler
from collections import defaultdict
import networkx as nx
import igraph as ig
import sys


def seed_generation(nodes_num):
    return np.random.choice(nodes_num)


def l1_distance(sub_property, original_property):
    return np.abs(np.sum(sub_property - original_property)) / np.sum(
        original_property)


def l1_distance_for_distribution(sub_property, original_property):
    keys = set(sub_property.keys()) | set(original_property.keys())
    dist = 0
    norm = sum(list(original_property.values()))
    for key in keys:
        if key not in sub_property.keys():
            sub_property[key] = 0
        if key not in original_property.keys():
            original_property[key] = 0
        dist += np.fabs(original_property[key] - sub_property[key])
    return float(dist) / norm


def node_number_distance(subgraph, original_graph):
    original_graph_nodes_number = nx.number_of_nodes(original_graph)
    subgraph_nodes_number = nx.number_of_nodes(subgraph)
    return l1_distance(subgraph_nodes_number, original_graph_nodes_number)


def average_degree(original_graph):
    degree = dict(nx.degree(original_graph))
    num = original_graph.number_of_nodes()
    return sum(degree.values()) / num


def degree_distribution(original_graph):
    nodes_num = len(original_graph.nodes)
    d_distribution = defaultdict(float)
    degree_sum = 0
    for node_degree in original_graph.degree:
        d_distribution[node_degree[1]] += 1
        degree_sum += node_degree[1]
    for degree in d_distribution.keys():
        d_distribution[degree] = d_distribution[degree] / nodes_num
    return degree_sum / nodes_num, d_distribution


def degree_distribution_distance(subgraph, original_graph):
    original_ad, original_dd = degree_distribution(original_graph)
    sub_ad, sub_dd = degree_distribution(subgraph)
    return l1_distance(sub_ad, original_ad), l1_distance_for_distribution(sub_dd
        , original_dd)


def neighbor_connectivity(original_graph):
    degree_nodes = defaultdict(list)
    for node_v in original_graph.nodes:
        degree_v = original_graph.degree()[node_v]
        degree_nodes[degree_v].append(node_v)

    knn = defaultdict(float)
    for degree in degree_nodes:
        if degree * len(degree_nodes[degree]) == 0:
            continue
        for node_v in degree_nodes[degree]:
            for node_w in list(original_graph.neighbors(node_v)):
                knn[degree] += original_graph.degree()[node_w]
        knn[degree] = float(knn[degree]) / (degree * len(degree_nodes[degree]))

    return knn


def neighbor_connectivity_distance(subgraph, original_graph):
    original_neighbor_connectivity = neighbor_connectivity(original_graph)
    sub_neighbor_connectivity = neighbor_connectivity(subgraph)
    return l1_distance_for_distribution(sub_neighbor_connectivity,
        original_neighbor_connectivity)


def network_clustering_coefficient_distance(subgraph, original_graph):
    original_clustering_coefficient = nx.average_clustering(original_graph)
    sub_clustering_coefficient = nx.average_clustering(subgraph)
    return l1_distance(sub_clustering_coefficient,
        original_clustering_coefficient)


def degree_dependent_clustering_coefficient(original_graph):
    degree_num = defaultdict(int)
    lcc_degree_sum = defaultdict(float)
    nodes = original_graph.nodes
    node_degree = dict(original_graph.degree)
    nodes_num = len(nodes)
    sum_lcc = 0

    for node in nodes:
        degree = node_degree[node]
        degree_num[degree] += 1
        if degree == 0 or degree == 1:
            continue
        lcc = 0
        node_adj_list = list(original_graph[node].keys())
        for i in range(0, degree - 1):
            x = node_adj_list[i]
            for j in range(i + 1, degree):
                y = node_adj_list[j]
                if node != x and x != y and y != node:
                    x_adj_list = list(original_graph[x].keys())
                    lcc += 2 * x_adj_list.count(y)

        lcc = float(lcc) / (degree * (degree - 1))
        lcc_degree_sum[degree] += lcc
        sum_lcc += lcc

    ddcc = defaultdict(float)
    for degree in degree_num:
        if degree_num[degree] > 0:
            ddcc[degree] = float(lcc_degree_sum[degree]) / degree_num[degree]
    return sum_lcc / nodes_num, ddcc


def degree_dependent_clustering_coefficient_distance(subgraph, original_graph):
    original_acc, original_ddcc = degree_dependent_clustering_coefficient(
        original_graph)
    sub_acc, sub_ddcc = degree_dependent_clustering_coefficient(subgraph)
    return l1_distance(sub_acc, original_acc), l1_distance_for_distribution(
        sub_ddcc, original_ddcc)


def common_neighbor_distribution(original_graph):
    cnd = defaultdict(float)

    for node_i in original_graph.nodes:
        node_i_adj_list = list(original_graph[node_i].keys())
        for node_j in node_i_adj_list:
            if node_j <= node_i:
                continue
            m = 0
            for node_k in node_i_adj_list:
                if node_k == node_i and node_k == node_j:
                    continue
                node_j_adj_list = list(original_graph[node_j].keys())
                m += node_j_adj_list.count(node_k)
            cnd[m] += 1

    norm = sum(list(cnd.values()))
    for m in cnd:
        cnd[m] = float(cnd[m]) / norm

    return cnd


def common_neighbor_distribution_distance(subgraph, original_graph):
    original_cnd = common_neighbor_distribution(original_graph)
    sub_cnd = common_neighbor_distribution(subgraph)
    return l1_distance_for_distribution(sub_cnd, original_cnd)


def calc_shortest_path_properties(original_graph):
    original_edges = nx.to_pandas_edgelist(original_graph).values
    ig_original_graph = ig.Graph(original_edges)
    ig_original_path_length_hist = ig_original_graph.path_length_hist(directed=
        False)
```

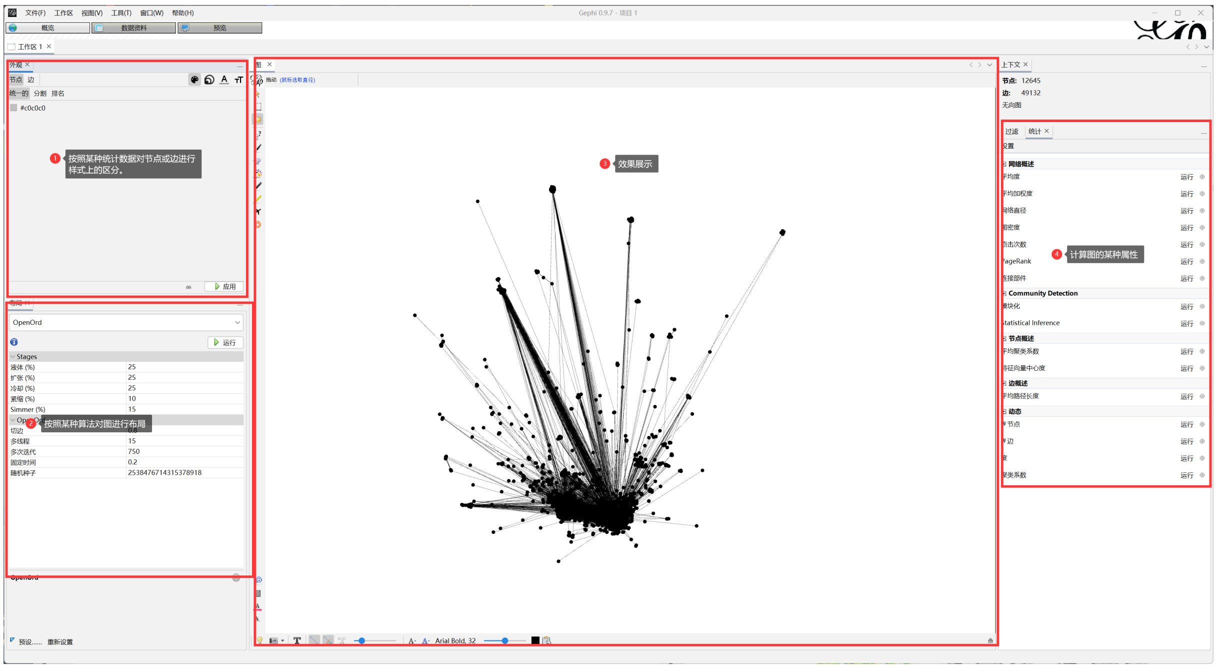
```
   spld = defaultdict(float)
   num_all = ig_original_path_length_hist.n
   bins = tuple(ig_original_path_length_hist.bins())

   for (i, j, k) in bins:
      if j != i + 1:
         print("Error.")
         exit(0)
      spld[i] = float(k) / num_all

   diameter = max(list(dict(spld).keys()))
   apl = sum([l * spld[l] for l in spld])
   return spld, diameter, apl


def shortest_path_length_distance(subgraph, original_graph):
   original_spld, original_diameter, original_apl =
      calc_shortest_path_properties(original_graph)
   sub_spld, sub_diameter, sub_apl = calc_shortest_path_properties(subgraph)
   spld_distance = l1_distance_for_distribution(sub_spld, original_spld)
   diameter_distance = l1_distance(sub_diameter, original_diameter)
   apl_distance = l1_distance(sub_apl, original_apl)
   return spld_distance, diameter_distance, apl_distance


def degree_dependent_betweeness_centrality(original_graph):
   original_edges = nx.to_pandas_edgelist(original_graph).values
   ig_original_graph = ig.Graph(original_edges)
   degrees = ig_original_graph.degree(list(range(0, len(original_graph.nodes)))
      )
   bc = ig_original_graph.betweenness(directed=False)
   n = int(ig_original_graph.vcount())

   ddbc = defaultdict(float)
   V_d = defaultdict(int)
   for i in range(0, len(degrees)):
      d = degrees[i]
      ddbc[d] += float(bc[i]) / ((n - 1) * (n - 2))
      V_d[d] += 1

   for d in ddbc:
      ddbc[d] = float(ddbc[d]) / V_d[d]

   return ddbc


def betweeness_centrality_distance(subgraph, original_graph):
   original_ddbc = degree_dependent_betweeness_centrality(original_graph)
   sub_ddbc = degree_dependent_betweeness_centrality(subgraph)
   return l1_distance_for_distribution(sub_ddbc, original_ddbc)


def largest_eigenvalue(original_graph):
   original_edges = nx.to_pandas_edgelist(original_graph).values
   ig_original_graph = ig.Graph(original_edges)
   L = ig_original_graph.laplacian(normalized=True)
   eigenvalues = np.linalg.eigvals(L)
   return float(max(eigenvalues))


def largest_eigenvalue_distance(subgraph, original_graph):
   original_L = largest_eigenvalue(original_graph)
   sub_L = largest_eigenvalue(subgraph)
   return l1_distance(sub_L, original_L)


def subgraph_modify(subgraph, original_graph):
   subgraph = nx.Graph(subgraph)
   subgraph_nodes_list = list(subgraph.nodes)
   for node in subgraph_nodes_list:
      node_neighbors_list = list(nx.all_neighbors(original_graph, node))
      for neighbor in node_neighbors_list:
         subgraph.add_node(neighbor)
         subgraph.add_edge(node, neighbor)
   return subgraph


def breadth_first_search(original_graph, start_node):
   breadth_first_search_sampler = BreadthFirstSearchSampler(
      number_of_nodes=int(original_graph.number_of_nodes() * 0.1))
   subgraph = breadth_first_search_sampler.sample(original_graph, start_node=
      start_node)
   subgraph = subgraph_modify(subgraph, original_graph)

   print("Normalized L1 distance of each property of breadth first search")

   n = node_number_distance(subgraph, original_graph)
   print("Number of nodes:", n)

   ad, dd = degree_distribution_distance(subgraph, original_graph)
   print("Average degree:", ad)
   print("Degree distribution:", dd)

   nc = neighbor_connectivity_distance(subgraph, original_graph)
   print("Neighbor connectivity:", nc)

   acc, ddcc = degree_dependent_clustering_coefficient_distance(subgraph,
      original_graph)
   print("Average local clustering coefficient:", acc)
   print("Degree-dependent clustering coefficient:", ddcc)

   cnd = common_neighbor_distribution_distance(subgraph, original_graph)
   print("Common neighbor distribution:", cnd)

   spld, diamter, apl = shortest_path_length_distance(subgraph, original_graph)
   print("Average shortest path length:", apl)
   print("Shortest path length distribution:", spld)
   print("Diameter:", diamter)

   bc = betweeness_centrality_distance(subgraph, original_graph)
   print("Degree-dependent betweenness centrality:", bc)

   # le = largest_eigenvalue_distance(subgraph, original_graph)
   # print("Largest eigenvalue:", le)
   print()


def snowball_sampling(original_graph, start_node):
   snow_ball_sampler = SnowBallSampler(number_of_nodes=original_graph.
      number_of_nodes() * 0.1, k=50)
   subgraph = snow_ball_sampler.sample(original_graph, start_node=start_node)
   subgraph = subgraph_modify(subgraph, original_graph)
```

```
    print("Normalized L1 distance of each property of snowball sampling")

    n = node_number_distance(subgraph, original_graph)
    print("Number of nodes:", n)

    ad, dd = degree_distribution_distance(subgraph, original_graph)
    print("Average degree:", ad)
    print("Degree distribution:", dd)

    nc = neighbor_connectivity_distance(subgraph, original_graph)
    print("Neighbor connectivity:", nc)

    acc, ddcc = degree_dependent_clustering_coefficient_distance(subgraph,
        original_graph)
    print("Average local clustering coefficient:", acc)
    print("Degree-dependent clustering coefficient:", ddcc)

    cnd = common_neighbor_distribution_distance(subgraph, original_graph)
    print("Common neighbor distribution:", cnd)

    spld, diamter, apl = shortest_path_length_distance(subgraph, original_graph)
    print("Average shortest path length:", apl)
    print("Shortest path length distribution:", spld)
    print("Diameter:", diamter)

    bc = betweeness_centrality_distance(subgraph, original_graph)
    print("Degree-dependent betweenness centrality:", bc)

    # le = largest_eigenvalue_distance(subgraph, original_graph)
    # print("Largest eigenvalue:", le)
    print()


def forest_fire_sampling(original_graph):
    forest_fire_sampler = ForestFireSampler(number_of_nodes=original_graph.
        number_of_nodes() * 0.1, p=0.7)
    subgraph = forest_fire_sampler.sample(original_graph)
    subgraph = subgraph_modify(subgraph, original_graph)

    print("Normalized L1 distance of each property of forest fire sampling")

    n = node_number_distance(subgraph, original_graph)
    print("Number of nodes:", n)

    ad, dd = degree_distribution_distance(subgraph, original_graph)
    print("Average degree:", ad)
    print("Degree distribution:", dd)

    nc = neighbor_connectivity_distance(subgraph, original_graph)
    print("Neighbor connectivity:", nc)

    acc, ddcc = degree_dependent_clustering_coefficient_distance(subgraph,
        original_graph)
    print("Average local clustering coefficient:", acc)
    print("Degree-dependent clustering coefficient:", ddcc)

    cnd = common_neighbor_distribution_distance(subgraph, original_graph)
    print("Common neighbor distribution:", cnd)

    spld, diamter, apl = shortest_path_length_distance(subgraph, original_graph)
    print("Average shortest path length:", apl)
    print("Shortest path length distribution:", spld)
    print("Diameter:", diamter)

    bc = betweeness_centrality_distance(subgraph, original_graph)
    print("Degree-dependent betweenness centrality:", bc)

    # le = largest_eigenvalue_distance(subgraph, original_graph)
    # print("Largest eigenvalue:", le)
    print()


def random_walk_sampling(original_graph, start_node):
    random_walk_sampler = RandomWalkSampler(number_of_nodes=original_graph.
        number_of_nodes() * 0.1)
    subgraph = random_walk_sampler.sample(original_graph, start_node=start_node)
    subgraph = subgraph_modify(subgraph, original_graph)

    print("Normalized L1 distance of each property of random walk sampling")

    n = node_number_distance(subgraph, original_graph)
    print("Number of nodes:", n)

    ad, dd = degree_distribution_distance(subgraph, original_graph)
    print("Average degree:", ad)
    print("Degree distribution:", dd)

    nc = neighbor_connectivity_distance(subgraph, original_graph)
    print("Neighbor connectivity:", nc)

    acc, ddcc = degree_dependent_clustering_coefficient_distance(subgraph,
        original_graph)
    print("Average local clustering coefficient:", acc)
    print("Degree-dependent clustering coefficient:", ddcc)

    cnd = common_neighbor_distribution_distance(subgraph, original_graph)
    print("Common neighbor distribution:", cnd)

    spld, diamter, apl = shortest_path_length_distance(subgraph, original_graph)
    print("Average shortest path length:", apl)
    print("Shortest path length distribution:", spld)
    print("Diameter:", diamter)

    bc = betweeness_centrality_distance(subgraph, original_graph)
    print("Degree-dependent betweenness centrality:", bc)

    # le = largest_eigenvalue_distance(subgraph, original_graph)
    # print("Largest eigenvalue:", le)
    print()


if __name__ == '__main__':
    file_name = "../data/syn10000.txt"
    graph = nx.read_edgelist(file_name, create_using=nx.Graph(), nodetype=int)
    graph.remove_edges_from(nx.selfloop_edges(graph))
    start_node = seed_generation(graph.number_of_nodes())
    breadth_first_search(graph, start_node)
    snowball_sampling(graph, start_node)
    forest_fire_sampling(graph)
    random_walk_sampling(graph, start_node)
```

3.7 实验环境搭建

- 语言：python3.9
- 所用库：numpy1.21.5；networkx2.7.1；igraph0.10.2；littleballoffur2.1.7
- 操作系统：windows11
- 集成开发环境：Python IDE
- 网络可视化：Gephi0.9.7

3.8 界面分析与使用说明



节点: 12645
边: 49132
无向图

OpenOrd

Stages
液体 (%) 25
扩张 (%) 25
冷却 (%) 25
紧缩 (%) 10
Simmer (%) 15

OpenOrd
切边 0.8
多线程 15
多次迭代 750
固定时间 0.2
随机种子 2538476714315378918

图 2: Gephi 操作界面示意

- 概览：展示图的基本结构
- 数据资料：保存图的原始数据
- 预览：对概况中的图自定义修改，预览
- 基本的编辑和设置：对于图的一般编辑操作都集中此处按照某种统计数据对节点或边进行样式上的区分。
- 布局：将图按照某种算法进行布局展示
- 统计与过滤：计算图的结构属性或过滤掉具有某些属性的节点或边

3.9 创新点

复现的部分没有对论文进行实质上的创新，但是在复现的过程中，对论文所提出的思想有了深刻的理解下，有一些创新的想法。不过由于时间原因，未能编写代码，将想法落地。

作者所提出的生成图的算法是基于随机游走采样得到的子图上进行生成的，而没有像 Gjoka 等人从空图的基础上重新去生成图。这样做可以保持采样子图的结构。之后的工作，作者是在对原图的一些结构属性估计的基础上，指导着生成一个图。

我在复现的结果上发现一些可以优化的方向，具体分析参见实验结果分析部分。

4 实验结果分析

4.1 基准算法的复现结果

- n：节点数量
- $\bar{k}$：平均度
- $\{P(k)\}_k$：度分布
- $\left\{\bar{k}_{nn}(k)\right\}_k$：邻域连通性
- $\bar{c}$：平均聚类系数
- $\{\bar{c}(k)\}_k$：度依赖聚类系数
- $\{P(s)\}_s$：公共邻居分布
- $\bar{l}$：平均最短路径长度
- $\{P(l)\}_l$：最短路径长度分布
- l_{max}：直径
- $\left\{\bar{b}(k)\right\}_k$：度依赖中介中心性

方法	n	$\bar{k}$	$\{P(k)\}_k$	$\left\{\bar{k}_{nn}(k)\right\}_k$	$\bar{c}$	$\{\bar{c}(k)\}_k$	$\{P(s)\}_s$	$\bar{l}$	$\{P(l)\}_l$	l_{max}	$\left\{\bar{b}(k)\right\}_k$
BFS	0.2454	0.0305	0.0768	0.1171	0.1006	0.2121	0.0767	0.0839	0.3574	0.4167	0.3907
Snowball	0.2414	0.0342	0.0778	0.1057	0.0629	0.1582	0.0871	0.0843	0.3531	0.3333	0.3753
FF	0.2356	0.0458	0.0681	0.0955	0.0219	0.1411	0.0963	0.0840	0.3528	0.3333	0.3761
RW	0.2434	0.0434	0.0675	0.0998	0.0159	0.1513	0.10177	0.0849	0.3538	0.25	0.3807

使用广度优先搜索、雪球采样、FF 采样和随机游走采样的基准采样算法对原图进行采样，将采样得到的子图与原图相应的结构计算其 L2 误差。

复现结果与作者在论文中所展示的结果基本吻合，考虑到算法具有很强的随机性，无法完全得到和作者一模一样的结果是属于正常的。

4.2 论文所提出的算法

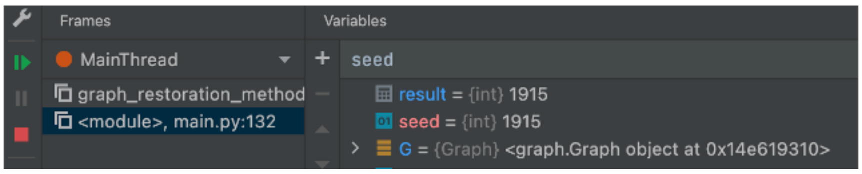
方法	n	$\bar{k}$	$\{P(k)\}_k$	$\left\{\bar{k}_{nn}(k)\right\}_k$	$\bar{c}$	$\{\bar{c}(k)\}_k$	$\{P(s)\}_s$	$\bar{l}$	$\{P(l)\}_l$	l_{max}	$\left\{\bar{b}(k)\right\}_k$
采样数量 1000	0.0764	0.1979	0.1127	0.7765	0.6588	3.1942	0.63812	0.1608	0.6776	0.2	0.6384
采样数量 10%	0.5341	0.6277	0.3675	0.1452	0.2789	0.1657	0.14206	0.1885	0.8270	0.0	0.5569
best	0.5738	0.1989	0.2118	0.1303	0.3931	0.1733	0.1357	0.1088	0.4506	0.0	0.3927

使用作者所提出的方法生成的图与的原图的结构属性的误差如上图所示。但是相较于作者论文中在 slashdot 这个数据集上的效果，复现的结果远远没有作者那么好。

究其原因，是因为该算法的一切都是在通过随机游走得到的采样节点序列的基础上进行的，包括对原图的各种属性的估计，更何况还有在这些属性的估计上对子图进行的调整。因此，最终生成图的效果好与坏依赖于采样节点序列的质量。

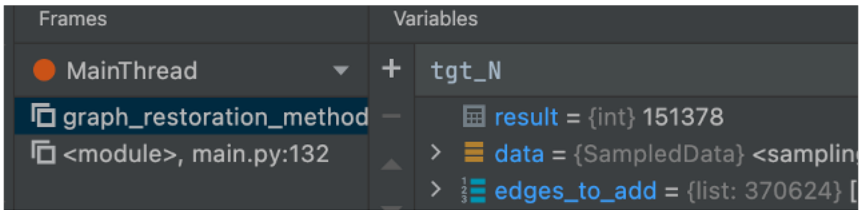
因此，为了进一步的确定我的想法和探究更本质的原因。我尝试选择度大的节点作为采样起始点进行采样。即使论文中是采用随机点，因为这更符合实际中的场景。得到的结果如下，以最终生成图

的节点数量为例（数量越接近原图节点数量 77360，结果越好）。



```
graph_restoration_method
<module>, main.py:132
result = {int} 1915
seed = {int} 1915
G = {Graph} <graph.Graph object at 0x14e619310>
```

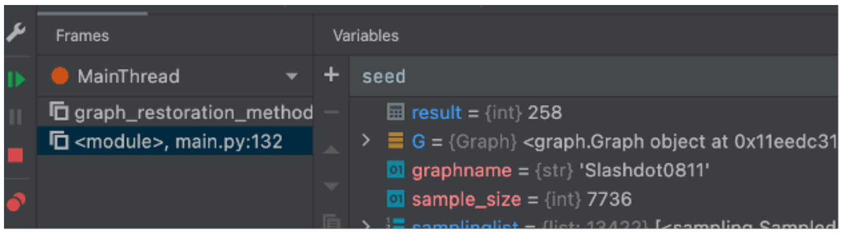
(a) 起始点



```
graph_restoration_method
<module>, main.py:132
result = {int} 151378
data = {SampledData} <samplin
edges_to_add = {list: 370624} [
```

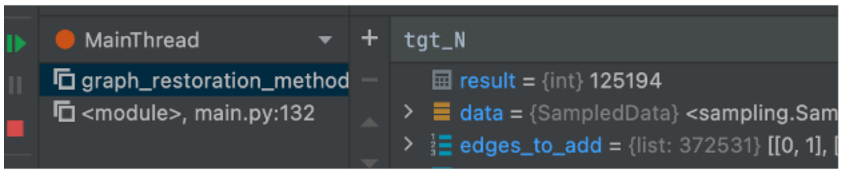
(b) 节点数量

图 3: 以 1915 为起始点的目标节点数量



```
graph_restoration_method
<module>, main.py:132
result = {int} 258
G = {Graph} <graph.Graph object at 0x11eedc31
graphname = {str} 'Slashdot0811'
sample_size = {int} 7736
samplinglist = {list: 13422} [<sampling.Sampled
```

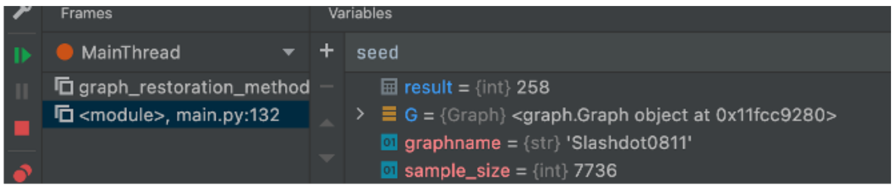
(a) 起始点



```
graph_restoration_method
<module>, main.py:132
result = {int} 125194
data = {SampledData} <sampling.Sam
edges_to_add = {list: 372531} [[0, 1], [
```

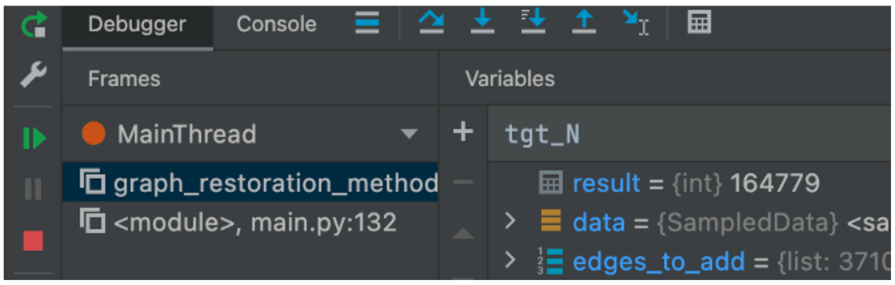
(b) 节点数量

图 4: 以 258 为起始点的目标节点数量 (第一次)



```
graph_restoration_method
<module>, main.py:132
result = {int} 258
G = {Graph} <graph.Graph object at 0x11fcc9280>
graphname = {str} 'Slashdot0811'
sample_size = {int} 7736
```

(a) 起始点



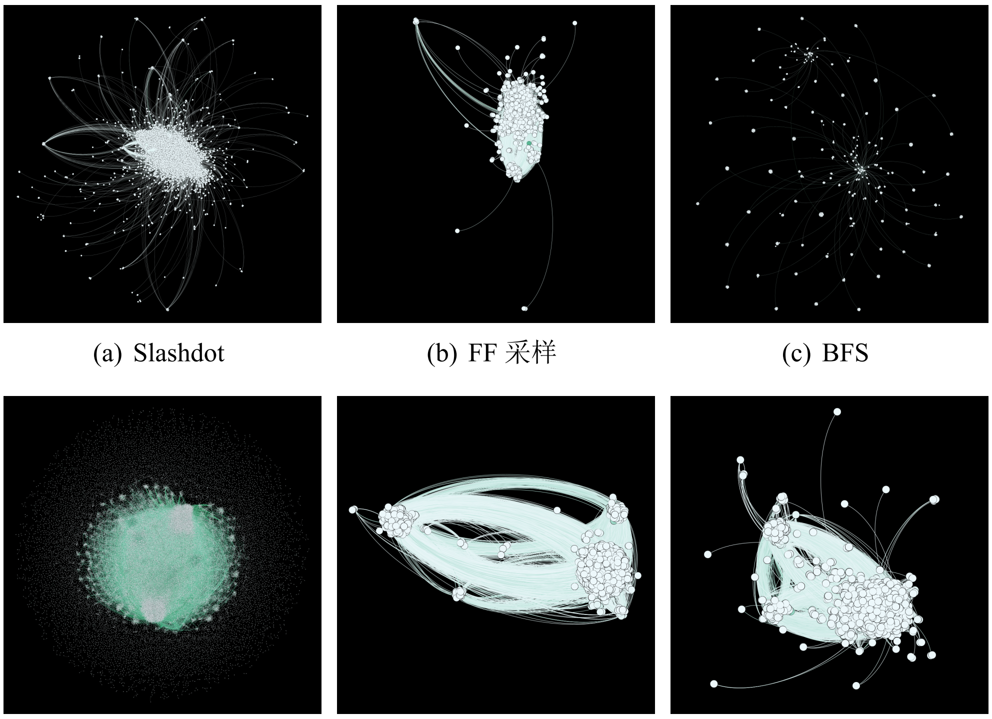
```
graph_restoration_method
<module>, main.py:132
result = {int} 164779
data = {SampledData} <sa
edges_to_add = {list: 3710
```

(b) 节点数量

图 5: 以 258 为起始点的目标节点数量 (第二次)

可以看的出来，度高的节点作为起始点，每次运行的结果也不尽相同，有好也有坏。只能说以度高的节点作为起始点最终得到好的结果概率会更大。

4.3 可视化与创新点


(a) Slashdot　(b) FF 采样　(c) BFS

(d) 生成图　(e) snowbal 采样　(f) randomwalk 采样

图 6: 图可视化

使用论文提出的方法生成的图跟原图结构相比，节点分布更加集中且均匀。没有零星的度小的节点散步在边缘。作者是使用了度分布、联合度分布以及度依赖聚类系数三个参数，生成的图虽然在这三个属性上接近于原图相应的属性，但仅限于按度进行的统计，而没有区分不同度之间的关系。

因此，我的想法是首先我们可以在作者基础上再添加度依赖联合最短路径的属性估计，对生成图再进一步进行调整。除此之外，还可以研究不同度之间节点数量、聚类系数的关系，再对图进行微调。

5 总结与展望

本部分对整个文档的内容进行归纳并分析目前实现过程中的不足以及未来可进一步进行研究的方向。

本文档可以分为引言、相关工作、本文方法、实验结果分析和总结与展望五个部分。在引言部分，我们介绍了论文的选题背景，选题依据以及选题意义。在相关工作部分，我们选题内容相关的工作进行简要的分类概括与描述。在本文方法部分，我们首先从四个阶段对论文所提算法进行分析与概括，然后将自己复现的代码与源码进行比较，最后介绍了实验环境、界面分析与使用说明和对论文未来工作的一些想法。在实验结果分析部分，我们先是对比了四个基准方法的实验结果，然后比较了论文提出的方法的实验结果，最后对生成的图进行了可视化，并对实验结果进行了分析，提出了自己的一些想法。

目前实验过程中第一点不足是除了基准方法达到了作者所实现的效果，使用作者所提出的算法实现的结果与作者所达到的效果还有一定的差距。原因可能就是如上面分析那样，也可能是因为复现过程中，有些许步骤代码编写错误。第二点不足是在复现过程中，还是非常依赖作者给出的源码，以我目前的实力单独去复现此算法还是有些困难的。第三点不足是由于完整地运行一次需要很长一段时间，所以我只在一部分的数据集上进行了测试，并有像论文中那样完备。

未来可进一步进行的研究方向是首先我们可以在作者基础上再添加度依赖联合最短路径的属性估计，对生成图再进一步进行调整。除此之外，还可以研究不同度之间节点数量、聚类系数的关系，再对图进行微调。

参考文献

[1] NAKAJIMA K, SHUDO K. Social graph restoration via random walk sampling[J]. 2022 IEEE 38th International Conference on Data Engineering (ICDE), 2022: 01-14.

[2] KATZIR L, LIBERTY E, SOMEKH O. Estimating sizes of social networks via biased sampling[J]. Proceedings of the 20th international conference on World wide web, 2011: 597-606.

[3] GJOKA M, KURANT M, BUTTS C T, et al. Walking in facebook: A case study of unbiased sampling of osns[J]. 2010 Proceedings IEEE Infocom, 2010: 1-9.

[4] GJOKA M, KURANT M, MARKOPOULOU A. 2.5 k-graphs: from sampling to generation[J]. IEEE, 2013.

[5] KATZIR L, HARDIMAN S J. Estimating clustering coefficients and size of social networks via random walk[J]. ACM Transactions on the Web (TWEB), 2015, 9(4): 1-20.