力导向图报告

1.基本信息和运行方法

数据集：课件提供的论文-作者数据集，我只采用了其部分数据

编程环境：Windows 10，python 3.8.8

依赖库：

- numpy==1.20.1

运行方法：

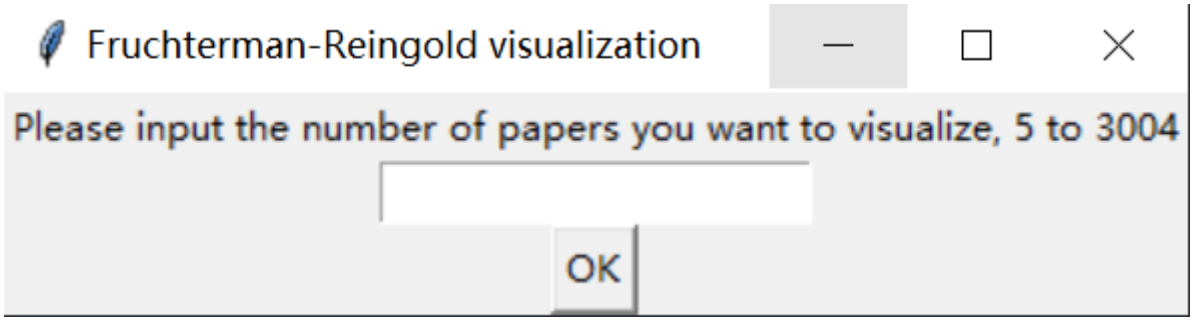
首先安装依赖

```
cd src
pip install -r requirements.txt
```

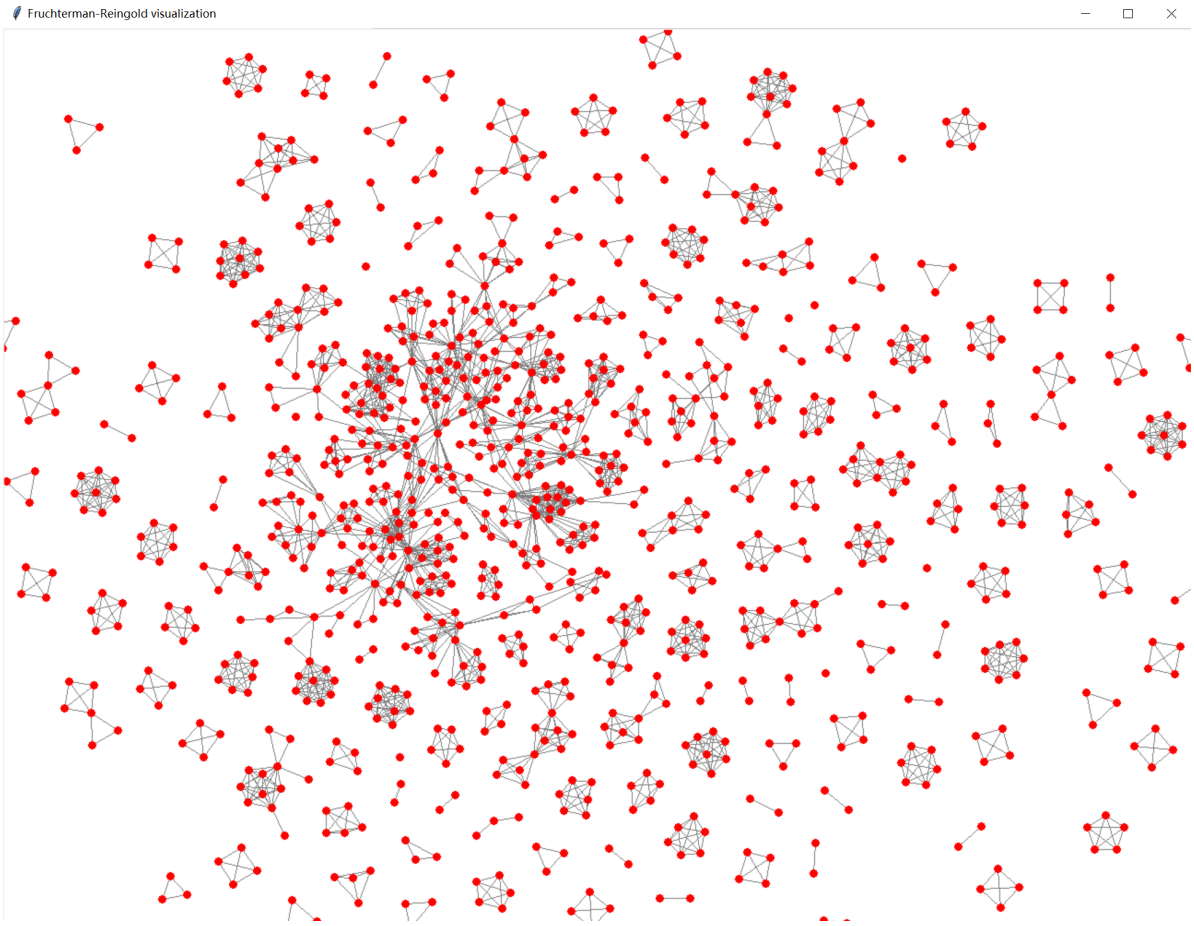
运行可视化交互界面

```
python interaction.py
```

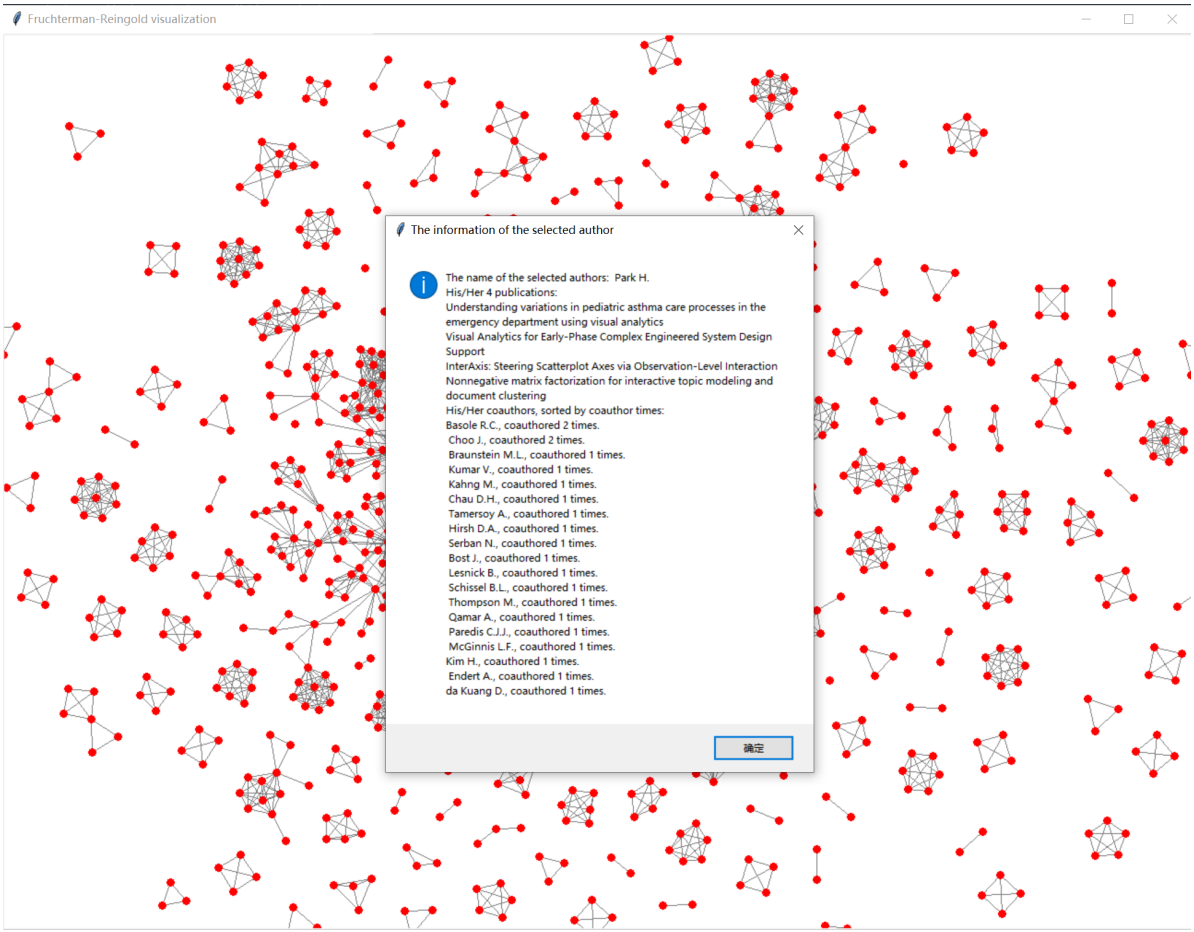
你需要首先按照提示输入要可视化的论文个数，之后就会进入可视化界面。



图中，红色圆形是节点，代表论文作者；灰色线是边，代表合作关系。



力导向图会不断更新，点会不断运动。你可以用鼠标左键点击点，查看对应的作者姓名、发表论文和合作者。退出时候有些小问题，无视即可。



具体运行情况可以运行代码或者查看video.mp4视频文件。

2.基本原理

我实现的是Fruchterman-Reingold算法

其基本原理如下：把每个点当作一个电荷，每条边当作一个弹簧，每两个点之间都有库仑斥力，每两个有边的点之间都有胡克弹力。两者的大小如下：

$$常数k = \sqrt{(\frac{area}{|V|})}，其中|V|为节点个数$$

$$弹簧弹力f_a(d) = \frac{d^2}{k}，其中d为两点间欧氏距离$$

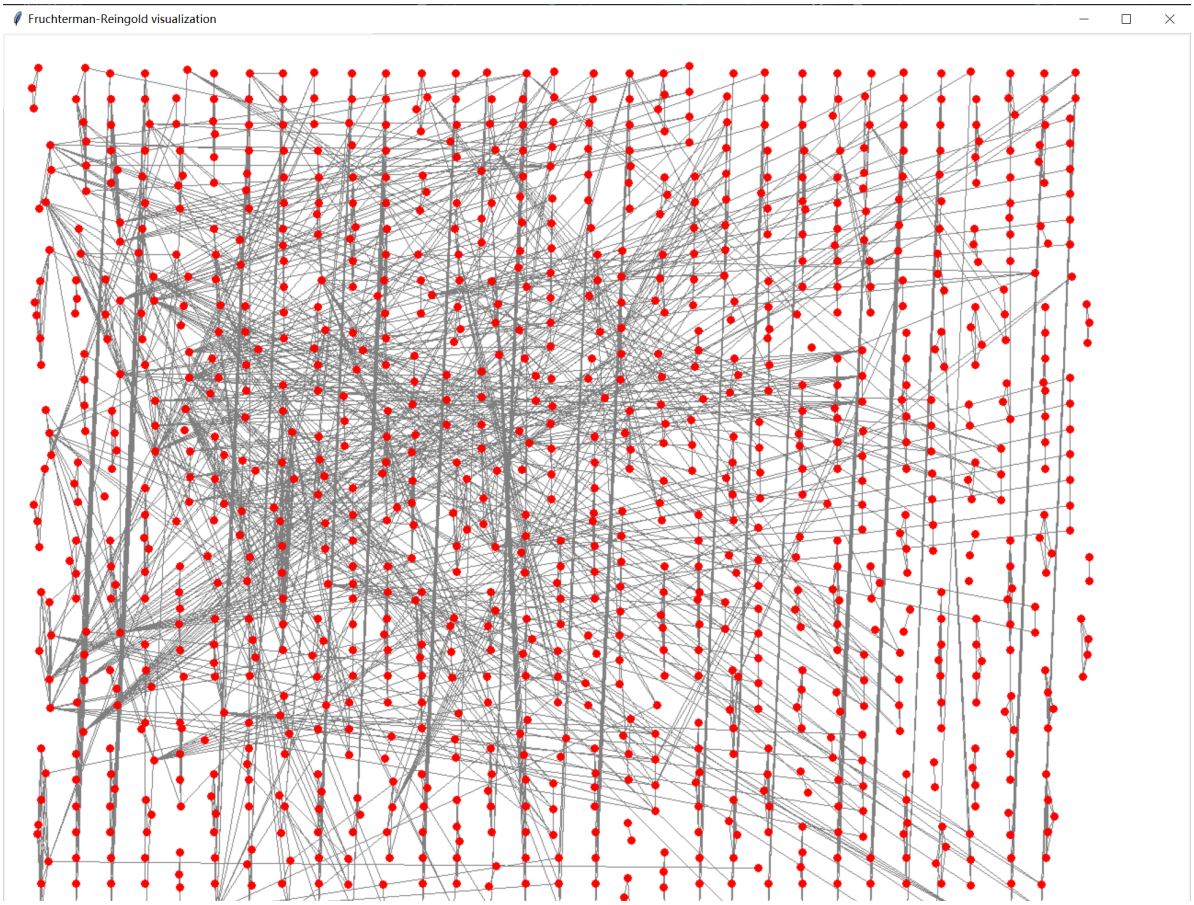
$$库仑斥力f_r(d) = -\frac{k^2}{d}$$

之后通过迭代的方式更新每个点的位置，用一个参数t限制迭代步长，其中t随着迭代不断变小，最终整个图就能收敛。

3.实现细节

我使用python的numpy库实现上述算法。我用一个N * 2维矩阵分别存储每个点的位置，用一个N * N维矩阵存储两点之间的边个数（如果vi,vj无边，则矩阵[i, j]和[j, i]值都是0，如果vi, vj有三条边，则则矩阵[i, j]和[j, i]值都是3，以此类推）。

我的具体流程是这样：首先，我将节点按照编号均匀初始化在整个屏幕中。



之后，我迭代足够多次（10000次），每次迭代分为四步：计算每两个节点的距离和方向向量、计算库仑斥力、计算胡克弹力、更新节点位置。我使用numpy库，利用python的矩阵运算加速大大提高了迭代运算效率，核心代码如下：

计算距离和方向向量模块：

```
self.dist_x = [] #N * N矩阵，dist_x[i, j]代表点j到点i的方向向量的x分量，已经归一化了
```

```
self.dist_y = [] #N * N矩阵，dist_y[i, j]代表点j到点i的方向向量的y分量，已经归一化了
self.dist = [] #N * N矩阵，dist[i, j]代表点i与点j的欧式距离
for i in range(self.nodes):
    x = self.position[i, 0]
    y = self.position[i, 1]
    dist_x = -(self.position[:, 0] - x) #N * 1
    dist_y = -(self.position[:, 1] - y) #N * 1
    dist = np.sqrt(dist_x ** 2 + dist_y ** 2) #N * 1
    dist[i] = 1.0
    dist_x = dist_x / dist #N * 1
    dist_y = dist_y / dist #N * 1
    dist[i] = 0.0
    self.dist_x.append(dist_x.reshape(1, self.nodes)) #1 * N
    self.dist_y.append(dist_y.reshape(1, self.nodes)) #1 * N
    self.dist.append(dist.reshape(1, self.nodes)) #1 * N

self.dist_x = np.concatenate(self.dist_x, axis=0) #N * N
self.dist_y = np.concatenate(self.dist_y, axis=0) #N * N
self.dist = np.concatenate(self.dist, axis=0) #N * N
```

计算库仑斥力模块：

```
self.repulsive = [] #N * 2矩阵，代表每个点受到的库仑斥力
mask = self.dist <= self.threshold_repulsive #为了防止斥力整体过大，我设定只有距离小于等于50像素距离的点互相才有斥力。斥力只是为了让点与点保持合理距离而已，这样做很合理
dist_x = self.dist_x * mask
dist_y = self.dist_y * mask

for i in range(self.nodes):
    self.dist[i, i] = 1.0 #avoid / 0
    repulsive_x = np.sum(dist_x[i] / self.dist[i] * self.k * self.k)
    repulsive_y = np.sum(dist_y[i] / self.dist[i] * self.k * self.k)
    repulsive = np.array([repulsive_x, repulsive_y],
dtype=np.float32).reshape(1, 2)
    self.dist[i, i] = 0.0
    self.repulsive.append(repulsive)
self.repulsive = np.concatenate(self.repulsive, axis=0) #N * 2
```

计算胡克弹力模块

```
self.attractive = [] #N * 2矩阵，代表每个点受到的胡克弹力
mask = self.adjacency
dist_x = self.dist_x * mask
dist_y = self.dist_y * mask

for i in range(self.nodes):
    attractive_x = np.sum(-dist_x[i] * self.dist[i] * self.dist[i] / self.k)
    attractive_y = np.sum(-dist_y[i] * self.dist[i] * self.dist[i] / self.k)
    attractive = np.array([attractive_x, attractive_y],
dtype=np.float32).reshape(1, 2)
    self.attractive.append(attractive)
self.attractive = np.concatenate(self.attractive, axis=0) #N * 2
```

更新模块

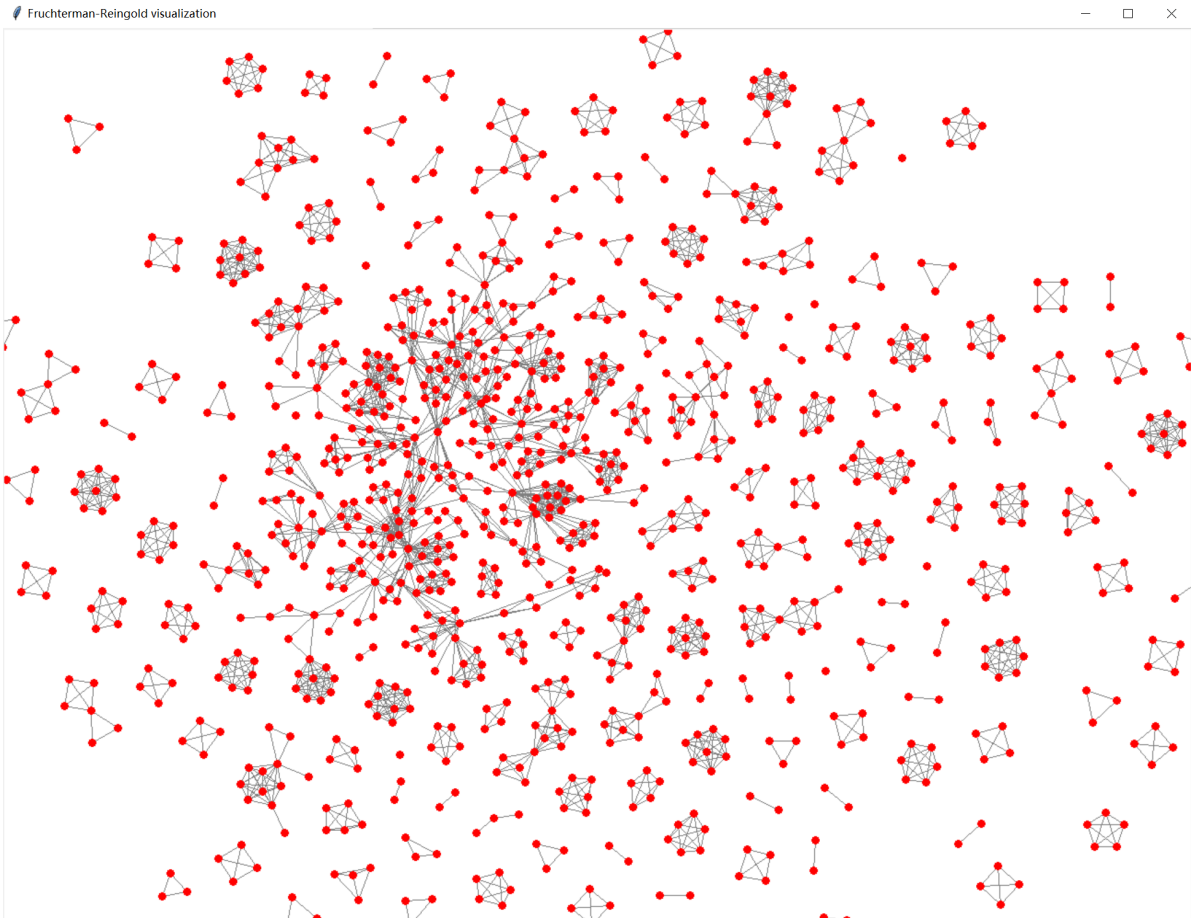
```
t = 1.0 - (self.iter / self.max_iters) #最大步长t，来限制更新量
delta = self.attractive + self.repulsive #N * 2矩阵，代表距离更新量
delta_length = np.sqrt(np.sum(delta ** 2, axis=1)) #N
mask = (delta_length < t)
learning_rate = (mask * delta_length + (~mask) * t).reshape(self.nodes,
1).repeat(2, axis=1)
length = delta_length.reshape(self.nodes, 1).repeat(2, axis=1)
length = length + (delta <= 0)
move = delta / length * learning_rate #N * 2
self.position = self.position + move
```

我使用python自带的tkinter库进行可视化。tkinter库的canvas可以显示圆（点）和线（边），让我们可以显示节点运动的动画信息，动画信息储存在video.mp4中。

在数据读取阶段，我就存储了每个作者的姓名、论文数量和名称、合作者姓名和合作次数（按照合作次数降序排列）。在点击选择节点之后，便可以查看每个作者的相应信息。

4.结果分析

首先，查看结果信息，每个连通子图都均匀分布在屏幕上，形成三角型、四边形、五边形等均匀结构，较大的图也以一些关键节点为边界分为几个子部分，可以看出结果的正确性。



其次，由于我们使用了numpy矩阵运算进行加速，我们的运行速度也有保障，具体结果如下：

论文个数	100	300	500	1000	2000	3004
点个数	378	1045	1644	2964	4991	6726
边个数	882	2625	4376	8238	16828	23828
每次迭代大致运行时间	0.03s	0.1s	0.2s	0.6s	1.8s	2.6s

可以看出，当点个数不到1000之时，可视化代码能运行的非常流畅。当点个数达到几千，接近10000的时候会略有卡顿但是也勉强可以使用，还是比较快速的。

不过我们实现的可视化算法也有一些小问题：首先，由于斥力问题部分节点会被推到屏幕之外无法显示，当点非常多的时候也因为屏幕大小原因难以查看细节，需要用一些显示方法提高显示范围，让显示范围超过平面。其次，加速不够充分，可以用四叉树近邻方法加速斥力求解，以及使用cuda并行化加速矩阵运算。