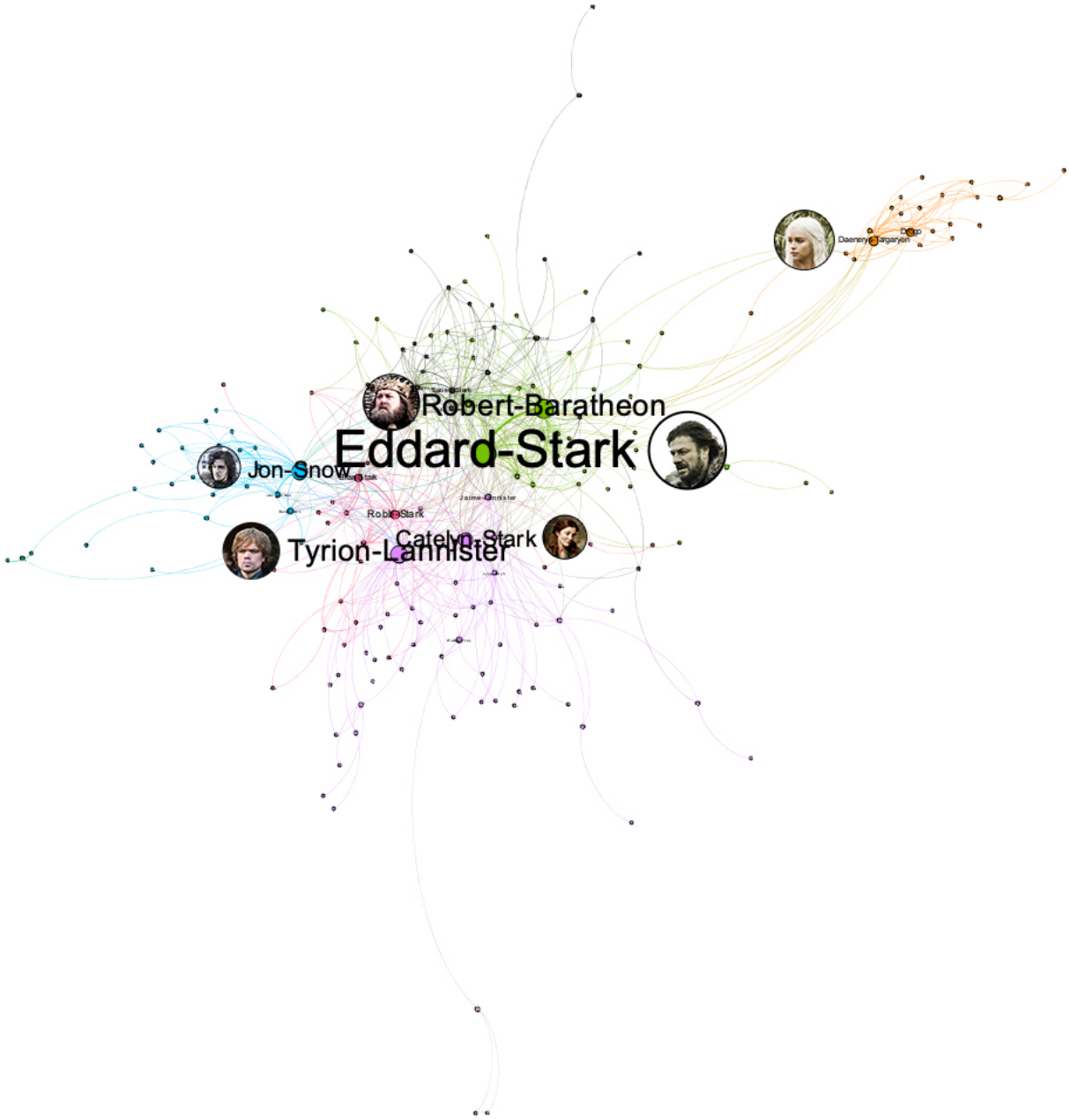


我们都知道《权利的游戏》在全世界都很多忠实的粉丝，除去你永远不知道剧情下一秒谁会挂这种意外“惊喜”，当中复杂交错的人物关系也是它火爆的原因之一，而本文介绍如何通过 NetworkX 访问开源的分布式图数据库 Nebula Graph，并借助可视化工具—— Gephi 来可视化分析《权力的游戏》中的复杂的人物图谱关系。

上篇

数据集

本文的数据集来源：冰与火之歌第一卷(至第五卷)[1]

- 人物集 (点集）：书中每个角色建模为一个点，点只有一个属性：姓名
- 关系集（边集）：如果两个角色在书中发生过直接或间接的交互，则有一条边；边只有一个属性：权重，权重的大小代表交互的强弱。

这样的点集和边集构成一个图网络，这个网络存储在图数据库 Nebula Graph [2]中。

社区划分——Girvan-Newman 算法

我们使用 NetworkX [3] 内置的社区发现算法 Girvan-Newman 来为我们的图网络划分社区。

以下为「社区发现算法 Girvan-Newman」解释：

> 网络图中，连接较为紧密的部分可以被看成一个社区。每个社区内部节点之间有较为紧密的连接，而在两个社区间连接则较为稀疏。社区发现就是找到给定网络图所包含的一个个社区的过程。

> Girvan-Newman 算法即是一种基于介数的社区发现算法，其基本思想是根据边介数中心性（edge betweenness）从大到小的顺序不断地将边从网络中移除直到整个网络分解为各个社区。因此，Girvan-Newman 算法实际上是一种分裂方法。

> Girvan-Newman 算法的基本流程如下： （1）计算网络中所有边的边介数； （2）找到边介数最高的边并将它从网络中移除； （3）重复步骤 2，直到每个节点成为一个独立的社区为止，即网络中没有边存在。

概念解释完毕，下面来实操下。

1. 使用 Girvan-Newman 算法划分社区。NetworkX 示例代码如下

```
comp = networkx.algorithms.community.girvan_newman(G)
k = 7
limited = itertools.takewhile(lambda c: len(c) <= k, comp)
communities = list(limited)[-1]
```

2. 为图中每个点添加一个 community 属性，该属性值记录该点所在的社区编号

```
community_dict = {}
community_num = 0
for community in communities:
    for character in community:
        community_dict[character] = community_num
        community_num += 1
        nx.set_node_attributes(G, community_dict, 'community')
```

节点样式——Betweenness Centrality 算法

下面我们来调整下节点大小及节点上标注的角色姓名大小，我们使用 NetworkX 的 Betweenness Centrality 算法来决定节点大小及节点上标注的角色姓名的大小。

> 图中各个节点的重要性可以通过节点的中心性（Centrality）来衡量。在不同的网络中往往采用了不同的中心性定义来描述网络中节点的重要性。Betweenness Centrality 根据有多少最短路径经过该节点，来判断一个节点的重要性。

1. 计算每个节点的介数中心性的值

```
betweenness_dict = nx.betweenness_centrality(G) # Run betweenness
centrality
```

2. 为图中每个点再添加一个 betweenness 属性

```
nx.set_node_attributes(G, betweenness_dict, 'betweenness')
```

边的粗细

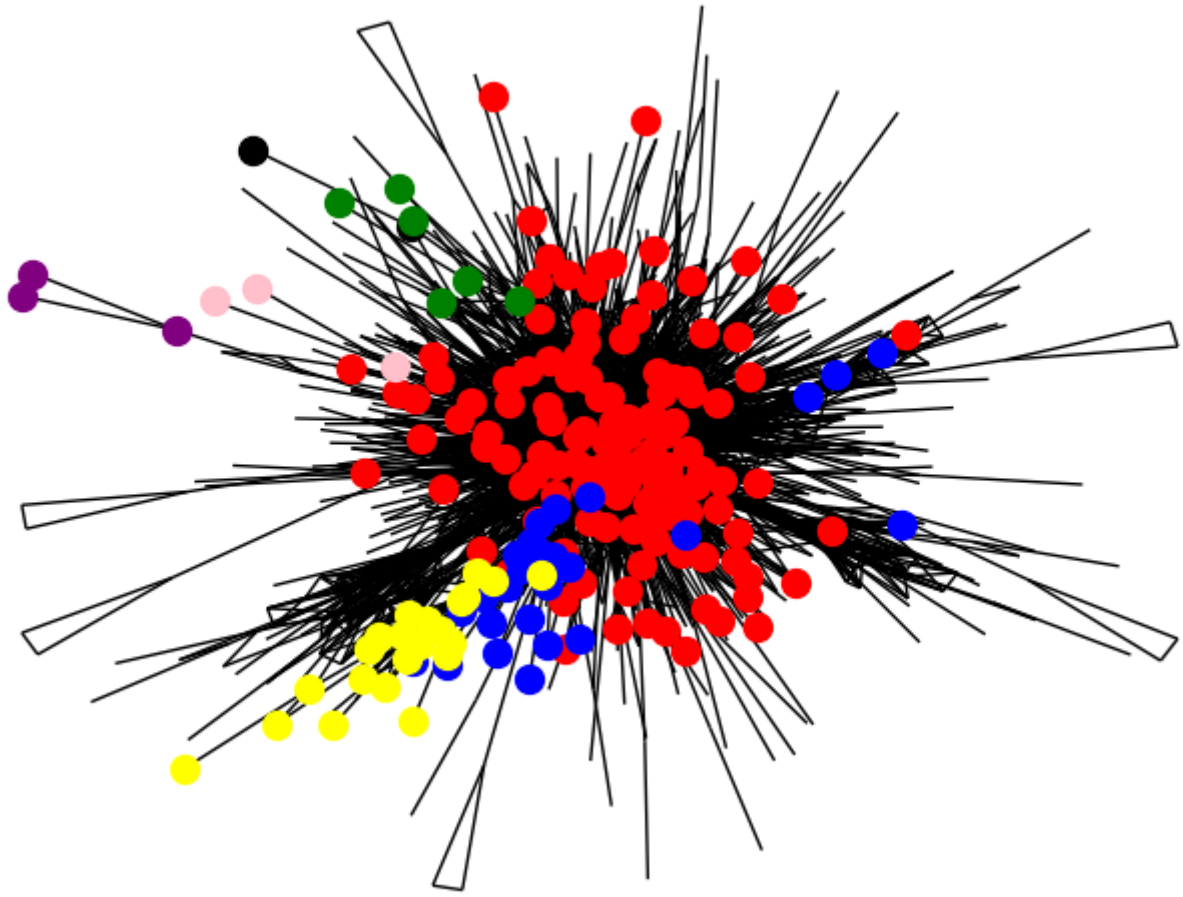
边的粗细直接由边的权重属性来决定。

通过上面的处理，现在，我们的节点拥有 name、community、betweenness 三个属性，边只有一个权重 weight 属性。

下面显示一下：

```
import matplotlib.pyplot as plt
color = 0
color_map = ['red', 'blue', 'yellow', 'purple', 'black', 'green', 'pink']
for community in communities:
    nx.draw(G, pos = nx.spring_layout(G, iterations=200), nodelist =
community, node_size = 100, node_color = color_map[color])
    color += 1
plt.savefig('./game.png')
```

emmm，有点丑...

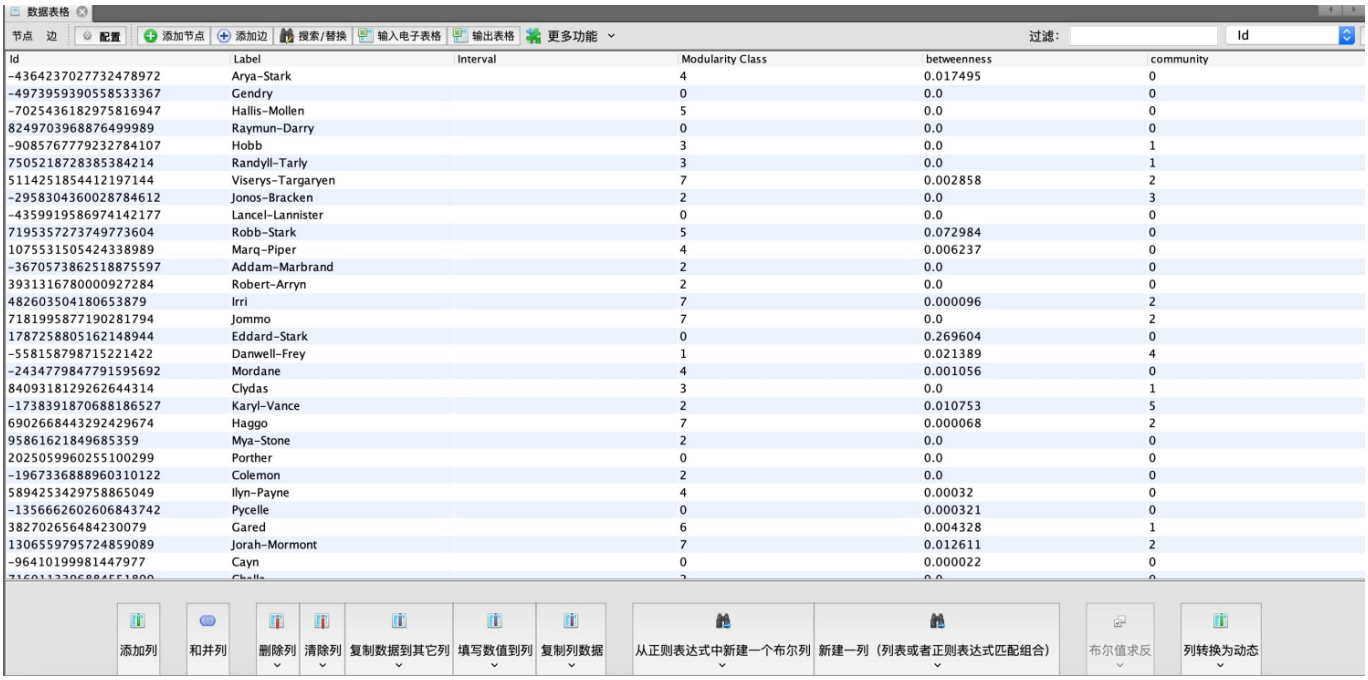


虽然 NetworkX 本身有不少可视化功能，但 Gephi [4] 的交互和可视化效果更好。

接入可视化工具 Gephi

现在将上面的 NetworkX 数据导出为 game.gephi 文件，并导入 Gephi。

```
nx.write_gexf(G, 'game.gexf')
```

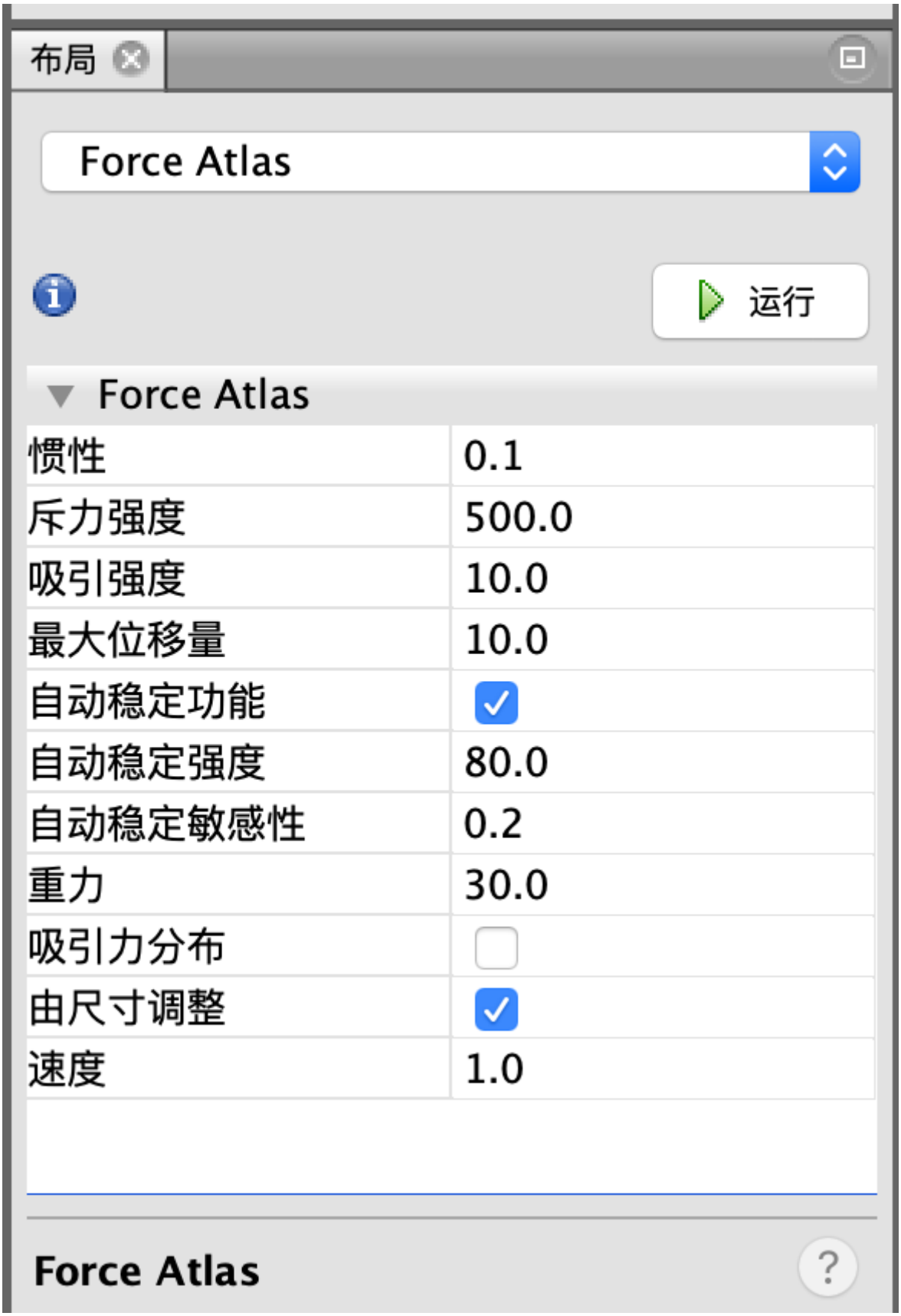


Gephi 可视化效果展示

在 Gephi 中打开刚才导出的 `game.gephi` 文件，然后微调 Gephi 中的各项参数，就以得到一张满意的可视化：

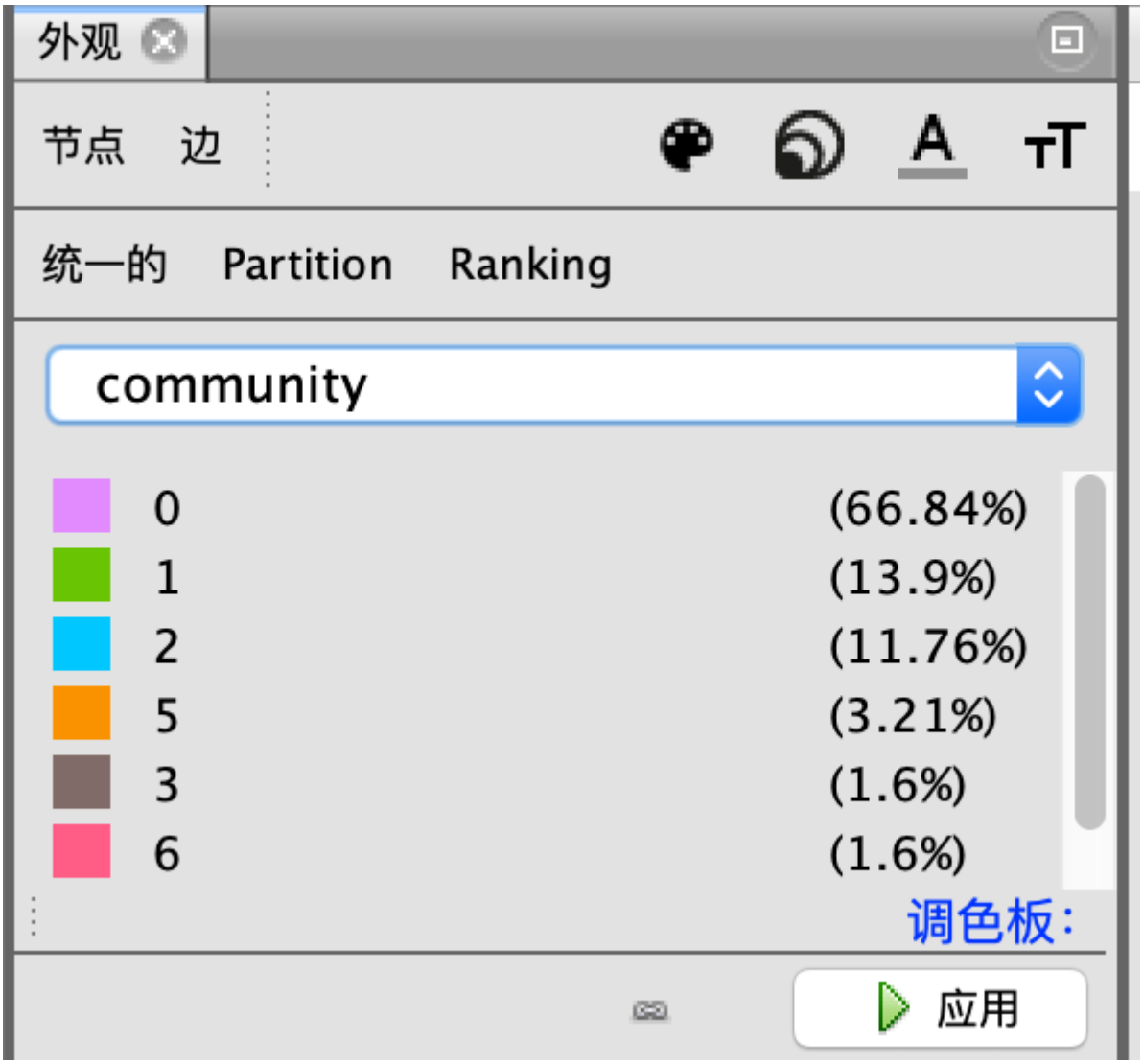
1. 将布局设置为 Force Atlas, 斥力强度改为为 500.0， 勾选上 由尺寸调整 选项可以尽量避免节点重叠：

Force Atlas 为力引导布局，力引导布局方法能够产生相当优美的网络布局，并充分展现网络的整体结构及其自同构特征。力引导布局即模仿物理世界的引力和斥力，自动布局直到力平衡。



2. 给划分好的各个社区网络画上不同的颜色：

在外观-节点-颜色-Partition 中选择 community（这里的 community 就是我们刚才为每个点添加的社区编号属性）



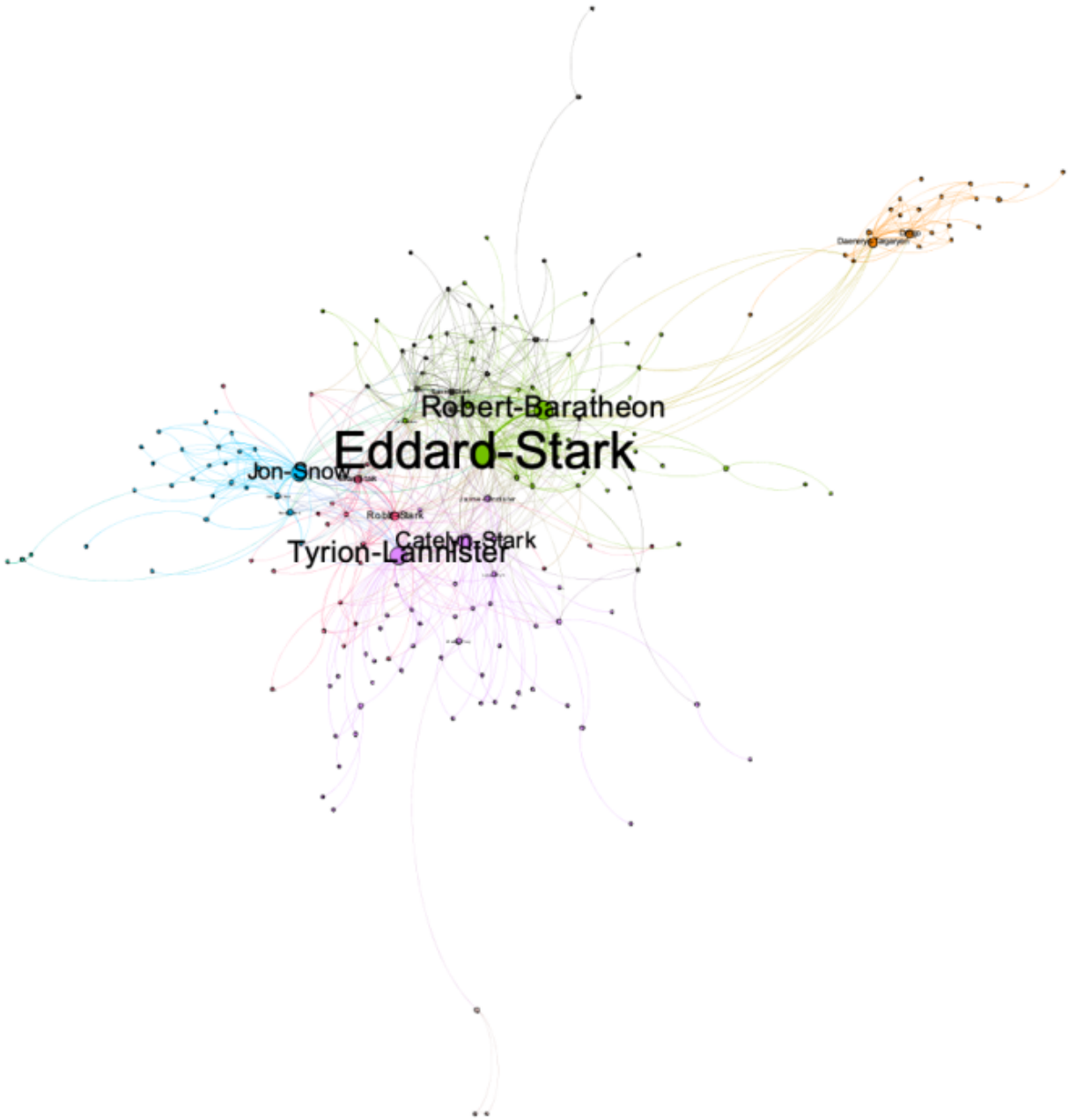
3. 决定节点及节点上标注的角色姓名的大小：

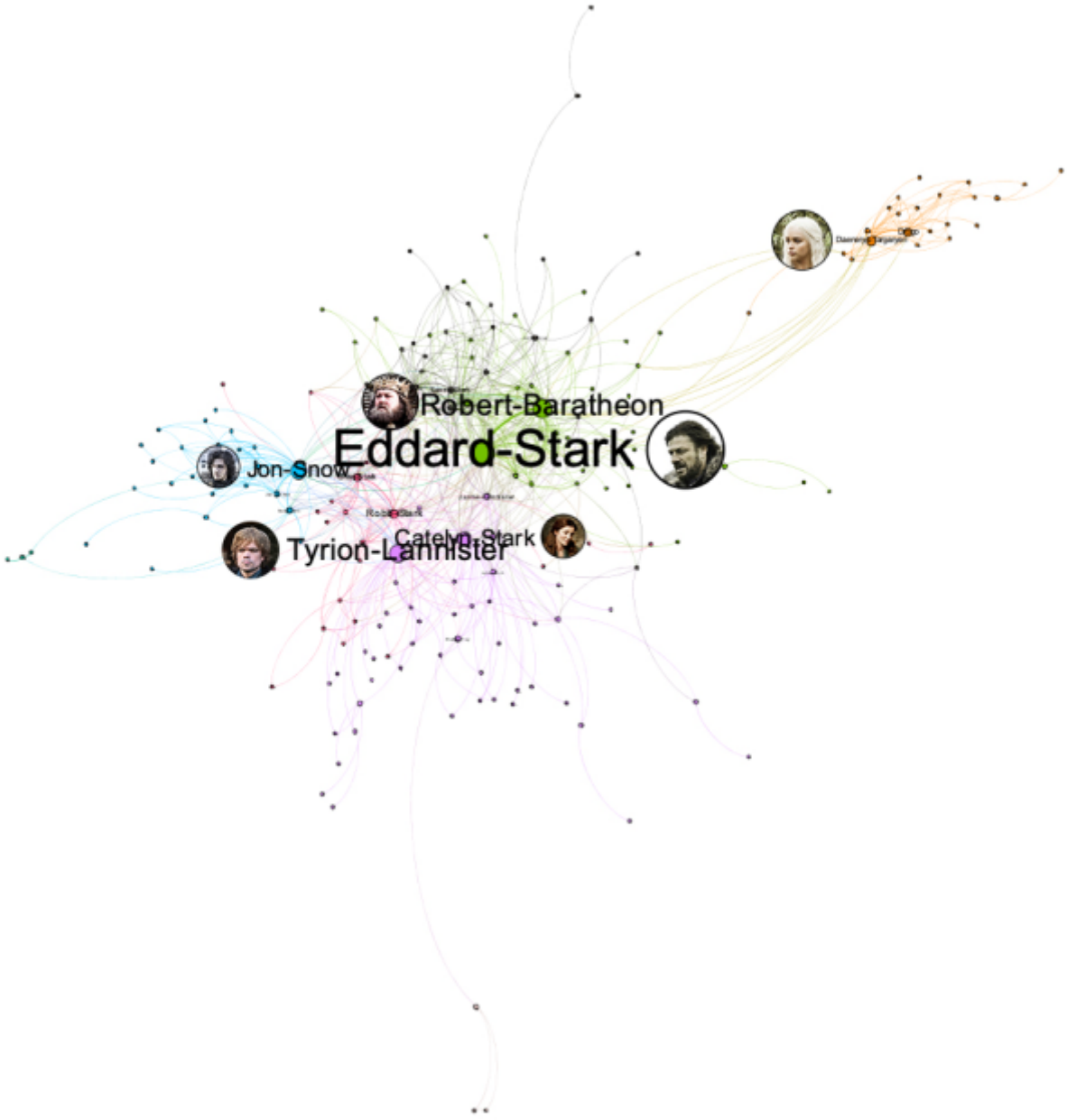
在外观-节点-大小-Ranking 中选择 betweenness（这里的 betweenness 就是我们刚才为每个点添加的 betweenness 属性)

4. 边的粗细由边的权重属性来决定：

在外观-边-大小-Ranking 中选择边的权重

5. 导出图片再加个头像效果


Robert-Baratheon
Eddard-Stark
Jon-Snow
Catelyn-Stark
Tyrion-Lannister



大功告成，一张权力游戏的关系谱图上线 😃 每个节点可以看到对应的人物信息。

本篇主要介绍如何使用 NetworkX，并通过 Gephi 做可视化展示。下篇将介绍如何通过 NetworkX 访问图数据库 Nebula Graph 中的数据。

上篇代码可以访问[5]。

致谢：本文受工作 [6] 的启发

下篇

在上一篇[7]中，我们通过 NetworkX 和 Gephi 展示了<权力的游戏>中的人物关系。在本篇中，我们将展示如何通过 NetworkX 访问图数据库 Nebula Graph。

NetworkX

NetworkX [8] 是一个用 Python 语言开发的图论与复杂网络建模工具，内置了大量常用的图与复杂网络分析算法，可以方便地进行复杂网络数据分析、仿真建模等工作，功能丰富，简单易用。

在 NetworkX 中，图是由顶点、边和可选的属性构成的数据结构。顶点表示数据，边是由两个顶点唯一确定的，表示两个顶点之间的关系。顶点和边也可以拥有更多的属性，以存储更多的信息。

NetworkX 支持 4 种类型的图：

- Graph：无向图
- DiGraph: 有向图
- MultiGraph: 多重无向图
- MultiDiGraph: 多重有向图

在 NetworkX 中创建一个无向图：

```
import networkx as nx
G = nx.Graph()
```

添加顶点：

```
G.add_node(1)
G.add_nodes_from([2,3,4])
G.add_node(2,name='Tom',age=23)
```

添加边：

```
G.add_edge(2,3)
G.add_edges_from([(1,2),(1,3)])
g.add_edge(1, 2, start_year=1996, end_year=2019)
```

在上一篇文章（一）中，我们已经演示了 NetworkX 的 Girvan-Newman 社区发现算法。

图数据库 Nebula Graph

NetworkX 通常使用本地文件作为数据源，这在静态网络研究的时候没什么问题，但如果图网络经常会发生变化——例如某些中心节点已经不存在(Fig.1)或者引入了重要的网络拓扑变化(Fig.2)——每次生成全新的静态文件再加载分析就有些麻烦，最好整个变化过程可以持久化在一个数据库中，并且可以实时地直接从数据库中加载子图或者全图做分析。本文选用 Nebula Graph [9]作为存储图数据的图数据库。

Fig. 1

Fig. 2

Nebula Graph 提供了两种方式来获取图结构：

1. 编写一个查询语句，拉取一个子图；
2. 全量扫描底层存储，获取一个完整的全图。

第一种方式适合在一个大规模的图网络中通过精细的过滤和剪枝条件来获取符合需求的若干个点和边。第二种方式更适合于全图的分析，这通常是在项目前期对全图进行一些启发式探索，当有进一步认知后再用第一种方式做精细的剪枝分析。

分析完 Nebula Graph 两种获取图结构方式后，下面来查看 Nebula Graph 的 Python 客户端代码，nebula-python/nebula/ngStorage/StorageClient.py 与 nebula-python/nebula/ngMeta/MetaClient.py 就是和底层存储交互的 API, 里面有扫描点、扫描边、读取一堆属性等等一系列丰富的接口。

下面两个接口可以用来读取所有的点、边数据：

```
def scan_vertex(self, space, return_cols, all_cols, limit, start_time,
end_time)
def scan_edge(self, space, return_cols, all_cols, limit, start_time,
end_time)
```

1. 初始化一个客户端，和一个 scan_edge_processor。scan_edge_processor 用来对读出来的边数据进行解码：

```
meta_client = MetaClient([('192.168.8.16', 45500)])
meta_client.connect()
storage_client = StorageClient(meta_client)
scan_edge_processor = ScanEdgeProcessor(meta_client)
```

2. 初始化 scan_edge 接口的各项参数：

```
space_name = 'nba' # 要读取的图空间名称
return_cols = {} # 要返回的边（或点）及其属性列
return_cols['serve'] = ['start_year', 'end_year']
return_cols['follow'] = ['degree']
allCols = False # 是否返回所有属性列，当该值为 False 时，仅返回在 returnCols 里指
定的属性列，当为 True 时，返回所有属性列
limit = 100 # 最多返回的数据条数
start_time = 0
end_time = sys.maxsize
```

3. 调用 scan_part_edge 接口，该接口会返回一个 scan_edge_response 对象的迭代器：

```
scan_edge_response_iterator = storage_client.scan_edge(space_name,
return_cols, all_cols, limit, start_time, end_time)
```

4. 不断读取该迭代器所指向的 scan_edge_response 对象中的数据，直到读取完所有数据：

```
while scan_edge_response_iterator.has_next():
    scan_edge_response = scan_edge_response_iterator.next()
    if scan_edge_response is None:
        print("Error occurs while scaning edge")
        break
    process_edge(space, scan_edge_response)
```

其中，process_edge 是自定义的一个处理读出来边数据的函数，该函数可以先使用 scan_edge_processor 对 scan_edge_response 中的数据进行解码，解码后的数据可以直接打印出来，也可以做一些简单处理，另作他用，比如：将这些数据读入计算框架 NetworkX 里。

5. 处理数据。在这里我们将读出来的所有边都添加到 NetworkX 中的图G 里：

```
def process_edge(space, scan_edge_response):
    result = scan_edge_processor.process(space, scan_edge_response)
    # Get the corresponding rows by edge_name
    for edge_name, edge_rows in result.rows.items():
        for row in edge_rows:
            srcId = row.default_properties[0].get_value()
            dstId = row.default_properties[2].get_value()
            print('%d -> %d' % (srcId, dstId))
            props = {}
            for prop in row.properties:
                prop_name = prop.get_name()
                prop_value = prop.get_value()
                props[prop_name] = prop_value
            G.add_edges_from([(srcId, dstId, props)]) # 添加边到 NetworkX 中
的图G
```

读取顶点数据的方法和上面的流程类似。

此外，对于分布式的一些图计算框架[10]来说，Nebula Graph 还提供了根据分片 (partition) 并发地批量读取存储的功能，这会在之后的文章中演示。

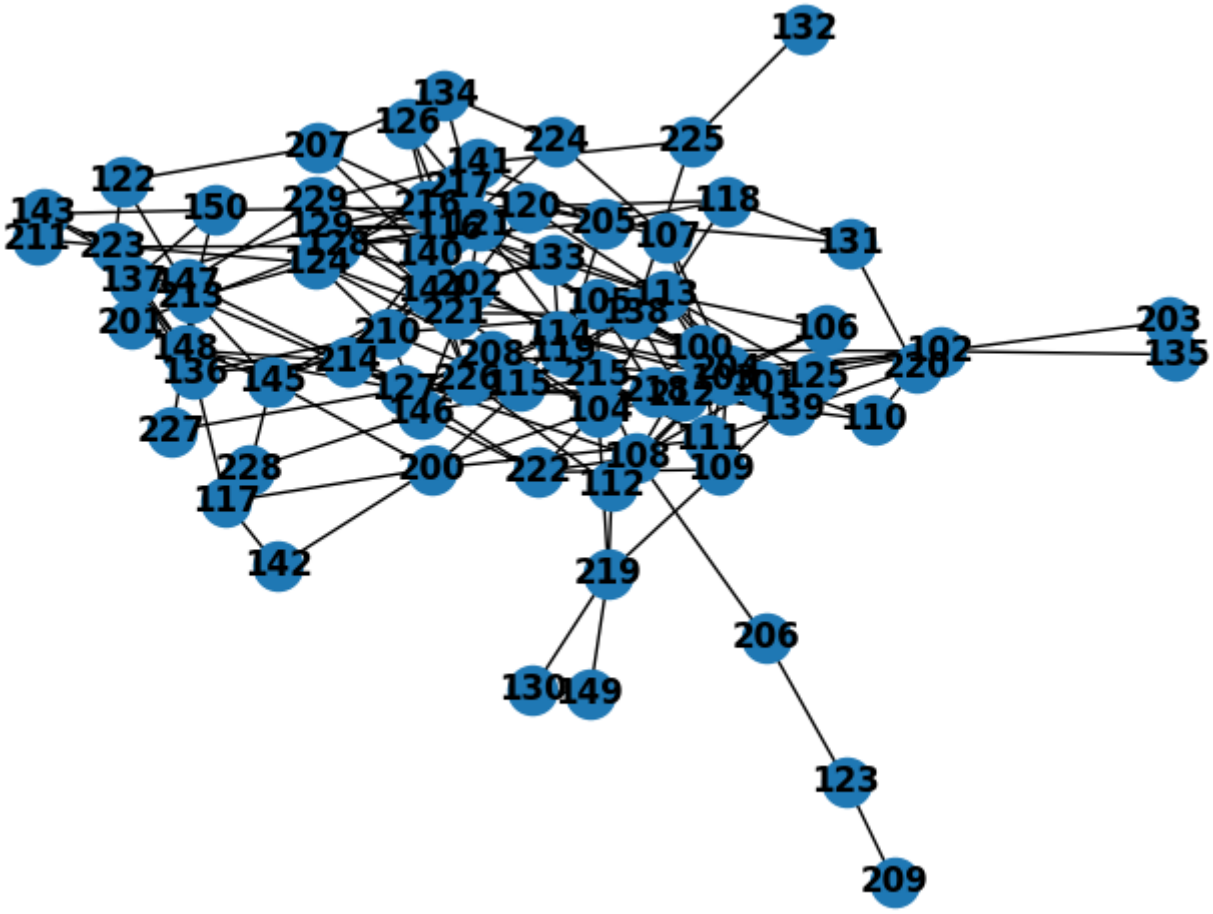
在 NetworkX 中进行图分析

当我们把所有点和边数据都按照上述流程读入 NetworkX 后，我们还可以做一些基本的图分析和图计算：

1. 绘制图：

```
nx.draw(G, with_labels=True, font_weight='bold')
import matplotlib.pyplot as plt
plt.show()
plt.savefig('./test.png')
```

绘制出来的图：



2. 打印出图中的所有点和边:

```
print('nodes: ', list(G.nodes))
print('edges: ', list(G.edges))
```

输出的结果:

```
nodes:  [109, 119, 129, 139, 149, 209, 219, 229, 108, 118, 128, 138, 148,
208, 218, 228, 107, 117, 127, 137, 147, 207, 217, 227, 106, 116, 126, 136,
146, 206, 216, 226, 101, 111, 121, 131, 141, 201, 211, 221, 100, 110, 120,
130, 140, 150, 200, 210, 220, 102, 112, 122, 132, 142, 202, 212, 222, 103,
113, 123, 133, 143, 203, 213, 223, 104, 114, 124, 134, 144, 204, 214, 224,
105, 115, 125, 135, 145, 205, 215, 225]
edges:  [(109, 100), (109, 125), (109, 204), (109, 219), (109, 222), (119,
200), (119, 205), (119, 113), (129, 116), (129, 121), (129, 128), (129,
216), (129, 221), (129, 229), (129, 137), (139, 138), (139, 212), (139,
218), (149, 130), (149, 219), (209, 123), (219, 130), (219, 112), (219,
104), (229, 147), (229, 116), (229, 141), (229, 144), (108, 100), (108,
101), (108, 204), (108, 206), (108, 214), (108, 215), (108, 222), (118,
120), (118, 131), (118, 205), (118, 113), (128, 116), (128, 121), (128,
201), (128, 202), (128, 205), (128, 223), (138, 115), (138, 204), (138,
210), (138, 212), (138, 221), (138, 225), (148, 127), (148, 136), (148,
```

```
137), (148, 214), (148, 223), (148, 227), (148, 213), (208, 127), (208,
103), (208, 104), (208, 124), (218, 127), (218, 110), (218, 103), (218,
104), (218, 114), (218, 105), (228, 146), (228, 145), (107, 100), (107,
204), (107, 217), (107, 224), (117, 200), (117, 136), (117, 142), (127,
114), (127, 212), (127, 213), (127, 214), (127, 222), (127, 226), (127,
227), (137, 136), (137, 213), (137, 150), (147, 136), (147, 214), (147,
223), (207, 121), (207, 140), (207, 122), (207, 134), (217, 126), (217,
141), (217, 124), (217, 144), (106, 204), (106, 212), (106, 113), (116,
141), (116, 126), (116, 210), (116, 216), (116, 121), (116, 113), (116,
105), (126, 216), (136, 210), (136, 213), (136, 214), (146, 202), (146,
210), (146, 215), (146, 222), (146, 226), (206, 123), (216, 144), (216,
105), (226, 140), (226, 112), (226, 114), (226, 144), (101, 100), (101,
102), (101, 125), (101, 204), (101, 215), (101, 113), (101, 104), (111,
200), (111, 204), (111, 215), (111, 220), (121, 202), (121, 215), (121,
113), (121, 134), (131, 205), (131, 220), (141, 124), (141, 205), (141,
225), (201, 145), (211, 124), (221, 104), (221, 124), (100, 125), (100,
204), (100, 102), (100, 113), (100, 104), (100, 144), (100, 105), (110,
204), (110, 220), (120, 150), (120, 202), (120, 205), (120, 113), (140,
114), (140, 214), (140, 224), (150, 143), (150, 213), (200, 142), (200,
104), (200, 145), (210, 124), (210, 144), (210, 115), (210, 145), (102,
203), (102, 204), (102, 103), (102, 135), (112, 204), (122, 213), (122,
223), (132, 225), (202, 133), (202, 114), (212, 103), (222, 104), (103,
204), (103, 114), (113, 104), (113, 105), (113, 125), (113, 204), (133,
114), (133, 144), (143, 213), (143, 223), (203, 135), (213, 124), (213,
145), (104, 105), (104, 204), (104, 215), (114, 115), (114, 204), (134,
224), (144, 145), (144, 214), (204, 105), (204, 125)]
```

3. 常见的，可以计算两个点之间的最短路径:

```
p1 = nx.shortest_path(G, source=114, target=211)
print('顶点 114 到顶点 211 的最短路径: ', p1)
```

输出的结果:

```
顶点 114 到顶点 211 的最短路径:  [114, 127, 208, 124, 211]
```

4. 也计算图中每个点的 PageRank 值，来看各自的影响力:

```
print(nx.pagerank(G))
```

输出的结果:

```
{109: 0.011507076520104863, 119: 0.007835838669313514, 129:
0.015304593799331218, 139: 0.007772926737873626, 149:
0.0073896601012629825, 209: 0.0065558926178649985, 219:
0.014100908598251508, 229: 0.011454115940170253, 108: 0.01645334474680034,
118: 0.01010598371500564, 128: 0.01594717876199238, 138:
0.01671097227127263, 148: 0.015898676579503977, 208: 0.009437234075904938,
218: 0.0153795416919104, 228: 0.005900393773635255, 107:
0.009745182763645681, 117: 0.008716335675518244, 127:
0.021565565312365507, 137: 0.011642680498867146, 147:
0.009721031073465738, 207: 0.01040504770909835, 217: 0.012054472529765329,
227: 0.005615576255373405, 106: 0.007371191843767635, 116:
0.020955704443679106, 126: 0.007589432032220849, 136:
0.015987209357117116, 146: 0.013922108926721374, 206:
0.008554794629575304, 216: 0.011219193251536395, 226:
0.013613173390725904, 101: 0.016680863106330837, 111:
0.010121524312495604, 121: 0.017545503989576015, 131:
0.008531567756846938, 141: 0.014598319866130227, 201:
0.0058643663430632525, 211: 0.003936285336338021, 221:
0.009587911774927793, 100: 0.02243017302167168, 110: 0.007928429795381916,
120: 0.011875669801396205, 130: 0.0073896601012629825, 140:
0.01205992633948699, 150: 0.010045605782606326, 200: 0.015289870550944322,
210: 0.017716629501785937, 220: 0.008666577509181518, 102:
0.014865431161046641, 112: 0.007931095811770324, 122:
0.008087439927630492, 132: 0.004659566123187912, 142:
0.006487446038191551, 202: 0.013579313206377282, 212: 0.01190888044566142,
222: 0.011376739416933006, 103: 0.013438110749144392, 113:
0.02458154500563397, 123: 0.01104978432213578, 133: 0.00743370900670294,
143: 0.008011123394996112, 203: 0.006883198710237787, 213:
0.020392557117890422, 223: 0.012345866520333572, 104:
0.024902235588979776, 114: 0.019369722463816744, 124:
0.017165705442951484, 134: 0.008284361176173354, 144:
0.019363506469972095, 204: 0.03507634139024834, 214: 0.015500649025348538,
224: 0.008320315540621754, 105: 0.01439975542831122, 115:
0.007592722237637133, 125: 0.010808523955754608, 135:
0.006883198710237788, 145: 0.014654713389044883, 205:
0.014660118545887803, 215: 0.01337467974572934, 225: 0.009909720748343093}
```

此外，也可以和上一篇中一样，接入Gephi [11]来得到更好的图可视化效果。

本文的代码可以参见[12].

Reference

[1] https://www.kaggle.com/mmmarchetti/game-of-thrones-dataset

[2] https://github.com/vesoft-inc/nebula

[3] https://networkx.github.io/

[4] https://gephi.org/

[5] https://github.com/jievince/nx2gephi

[6] https://www.lyonwj.com/2016/06/26/graph-of-thrones-neo4j-social-network-analysis/

[7] https://nebula-graph.com.cn/posts/game-of-thrones-relationship-networkx-gephi-nebula-graph/

[8] https://networkx.github.io/

[9] https://github.com/vesoft-inc/nebula

[10] https://spark.apache.org/graphx/

[11] https://gephi.org/

[12] https://github.com/vesoft-inc/nebula-python/pull/31