演员的小世界实验报告

杨佳宇　1800011409

1　代码和数据分析

1.1　图的建立

图的建立代码如图1。

```
153 def buildGraph():  # 用于建图，返回值为一个Graph
154     f = open('Film.json', 'r')
155     allfilm = json.load(f)
156     f.close()
157     graph = Graph()
158     for film in allfilm:  # 对每个电影进行逐一处理
159         actors = film['actor'].split(',')
160         for actor in actors:  # 先将电影的演员创建Vertex加入到图中
161             if actor not in graph:
162                 graph.addVertex(actor)
163             graph.addFilm_V(actor, film)  # 将这部电影添加到每个演员对应的Vertex中
164         num = len(actors)
165         if num >= 2:  # 演员不少于2人时在演员之间两两构建一条边
166             for i in range(num - 1):
167                 for j in range(i + 1, num):
168                     # 在addFilm_E函数中，如果actors[i]和actors[j]之间没有边则会先建立边，如果有边则会直接在边中添加电影film
169                     graph.addFilm_E(actors[i], actors[j], film)
170     return graph
```

图 1: 图的建立

总体思路是逐个电影进行处理，先保证所有演员都在图中，再给顶点中添加电影、添加边、给边添加电影。这样只有在添加顶点时需判断某演员是否已在图中，提高了效率。添加边的时候加入对演员是否一致的判断，避免出现自环，如图2。

```
130     def addEdge(self, actor1, actor2, film=None):  # actor1:str, actor2:str, film:dict, 在actor1和actor2对应的顶点之间添加一条边
131         # film为actor1和actor2共同出演的电影，给这条边也附加这个共同出演的电影
132         if actor1 != actor2:  # 为了避免出现自环
133             newEdge = Edge(self.vertices[actor1], self.vertices[actor2])
134             if film:
135                 newEdge.addFilm(film)
136             self.vertices[actor1].addNbr(self.vertices[actor2], newEdge)
137             self.vertices[actor2].addNbr(self.vertices[actor1], newEdge)
```

图 2: 添加 Edge

1.2　得到连通分支

代码如图3。该图为无向图，因此使用 BFS 选定某个起始顶点，所能搜索到的顶点就构成一个连通分支，所以只需要对图中的顶点依次 BFS 即可。使用一个集合来记录剩余还没搜索到的顶点，已经被归入某个连通分支的顶点无需再进行 BFS。

```python
def getComponents(graph):  # 将图中所有联通分支输出
    # 返回值components为一个list，其中元素为set，每一个set为一个联通分支中的顶点的集合
    # components的元素按照各自集合的大小降序排列
    components = []
    restVertex = set(graph.getAllVertex())
    # 因为一次bfs所能搜索到的顶点就构成一个连通分支，所以将还没搜索到的节点记录在集合中，可以避免在获取连通分支时的重复搜索
    while len(restVertex) != 0:
        curComponent = bfs(restVertex.pop(), Compnt=True)  # 为一个得到的连通分支
        components.append(curComponent)
        # print('getComponents:', len(components), ' finished')
        # 用于监视获取连通分支的进程
        restVertex = restVertex - curComponent  # 利用差集将已得到连通分支中的顶点从剩余顶点中剔除
    components.sort(key=lambda x: len(x), reverse=True)  # 根据连通分支的规模降序排列
    # components.sort(key=lambda x: (len(x), len(getEdges(x))), reverse=True)
    # 顶点数相同时按照边数排序，但实际运行时会大大降低效率，并且该二级排序意义并不显著
    return components
```

图 3: 获取连通分支

其中 BFS 的代码如图4，在这里只需要得到包含起始顶点的连通分支，连通分支中顶点并不分先后顺序，也没有重复，所以将连通分支中的顶点放在一个集合中返回。

```python
def bfs(start, Compnt=False, Dist=False, ColorReverse=False):
    # start:Vertex，为进行广度优先搜索的起始顶点
    # Compnt:bool，为True时返回值为set，为广度优先搜索搜索到的顶点的集合
    # Dist:bool，为True时返回值为int，为广度优先搜索能搜到的顶点中与起始点的最远距离，
    # ColorReverse:bool，广度优先搜索进行第一次后，会将能搜到的节点的color全部变为'black'，
    # 因此ColorReverse为True的时候规定起始颜色为'black'，搜索中为'gray'，搜索完成后为'white'，即黑白互换
    currentComponent = set()
    d = 0
    white = 'black' if ColorReverse else 'white'  # 这里的white代表bfs之前起始节点和能搜到的节点的color
    black = 'white' if ColorReverse else 'black'  # 这里的black代表bfs之后起始节点和能搜到的节点的color
    start.setDistance(0)
    vertexQueue = Queue()  # bfs过程中借助队列这一数据结构
    vertexQueue.enqueue(start)
    while not vertexQueue.isEmpty():
        current = vertexQueue.dequeue()
        for nbr in current.getNbr():
            if nbr.getColor() == white:  # 未搜索时color为white
                nbr.setColor('gray')  # 搜索中color为'gray'
                nbr.setDistance(current.getDistance() + 1)
                vertexQueue.enqueue(nbr)
        current.setColor(black)  # 搜索完成后color为black
        if Compnt:
            currentComponent.add(current)  # 将搜索到的顶点加入集合
        if Dist and d < current.getDistance():
            d = current.getDistance()  # 将最远的dist记录下来
    if Compnt:
        return currentComponent
    if Dist:
        return d
```

图 4: 广度优先搜索

最后将获取的所有连通分支进行排序，排序的依据为包含的节点数。对于本题中的数据，会有很多连通分支包含顶点数是相同的。起初考虑过相同时比较边的个数，但这样极大地降低了排序过程的效率，另外注意到实际上前 20 名的并列并没有显著影响，而后 20 名规模均为 1，边的个数均为 0，即使考虑了边的个数也没办法区分。最终相同同规模完全按照 BFS 的顺序输出，因此并列情形的输出具有不确定性。

1.3 连通分支相关属性的计算

连通分支规模只用计算集合的大小即可。电影平均星级和类别前三名代码如图5，先将连通分支中的所以电影统计出来，再对这些电影进行处理。

```python
def getFilms(component):  # component:set, 集合中的元素为Vertex
    # 返回值film_dic:dict, key:str为电影的id, value:dict为电影信息对应的集合
    # 返回的film_dic包含集合中所有演员出演的电影
    film_dic = {}
    for vertex in component:
        films = vertex.getFilms()
        for film in films:
            film_dic[film['_id']['$oid']] = film
            # 使用字典可以很好地避免因为电影重复出现带来的麻烦，虽然set也可以实现，但是film为dict, 是unhashable类, 因此不能使用set
    return film_dic


def getType(film_dic):  # film_dic:dict, 与getFilms返回值的数据类型相同
    # 返回值为list, 为film_dic中所有电影类别排行前3, 如果不足3种类别则全部输出
    types = {}  # key:str为电影类别, value:int为该类别出现的次数
    for film in film_dic.values():
        type_lst = film['type'].split(',')
        for t in type_lst:
            if t in types:
                types[t] += 1
            else:
                types[t] = 1
    result = list(types.keys()).copy()
    result.sort(key=lambda x: (types[x], x), reverse=True)  # 在类别出现次数相同时按字符串大小排序
    return result[:3:]


def getAveStar(film_dic):  # film_dic:dict, 与getFilms返回值的数据类型相同
    # 返回值为float, 为film_dic中所有电影的平均星级数, 保留到小数点后2位
    total_star = 0
    for film in film_dic.values():
        total_star += film['star']
    return round(total_star/float(len(film_dic)), 2)
```

图 5: 连通分支中电影数据统计

其中类别如果不足三个则全部输出，如果存在并列则按字符串大小优先输出较大者。

连通分支直径的计算如图6。在 BFS 中记录与起始顶点最远的距离，对连通分支中的每个节点都进行 BFS，这些最远距离中的最大值就是直径。其中 BFS 后顶点颜色均为黑色时，BFS 的发现过程变更为黑 → 灰 → 白。

```python
def getDiameter(component):  # component:set, 集合中的元素为Vertex, 为一个连通分支
    # 返回值为int, 为该连通分支的直径
    diameter = 0
    for vertex in component:  # 对每个顶点都作为起始点进行bfs, 返回最大距离 这些最大距离的最大值为该连通分支的直径
        reverse = (vertex.getColor() == 'black')  # 用于判断联通分支内是否均为黑色，如果是则需要在bfs时黑白互换
        d = bfs(vertex, Compnt=False, Dist=True, ColorReverse=reverse)
        if d > diameter:
            diameter = d
    return diameter
```

图 6: 连通分支直径计算

主函数中相关代码如图7。

```
    bottoms = components[-20::]  # 连通分支中规模最小的20个
    top_bottom = components[:20:]  # 规模最大的20个
    top_bottom.extend(bottoms)  # 将规模最小的20个接到规模最大的20个之后
    scale = []
    star = []
    types = []
    diameter = []
    for i in range(len(top_bottom)):  # 依次处理这40个连通分支
        scale.append(len(top_bottom[i]))
        films = getFilms(top_bottom[i])
        types.append(getType(films))
        star.append(getAveStar(films))
        if i != 0:
            diameter.append(getDiameter(top_bottom[i]))
        else:  # 最大的连通分支直径设为-1
            diameter.append(-1)
    draw_bar(scale, diameter, star)  # 将结果绘制成柱形图
    print('scale:', scale)
    print('types:', types)
    print('star:', star)
    print('diameter:', diameter)
```

图 7: 主函数中与连通分支计算相关代码

1.4 与周星驰相关的计算

这里使用连通分支计算中的代码，只用将周星驰和共同出演者放在一个集合中作为输入即可。该过程代码如图8。

```
def getPartners(graph, actor):  # graph:Graph, actor:str
    # 返回值为set, 集合中元素均为Vertex, 且为演员actor和actor曾经的共同出演者们构成的集合
    vertex = graph.getVertex(actor)
    partners = {vertex}
    for partner in vertex.getNbr():
        partners.add(partner)
    return partners
```

图 8: 得到演员与共同出演者的集合

主函数中相关代码如图9。

```
    vertex_zxc = g.getVertex('周星驰')  # 获得周星驰对应的顶点
    zxc = {vertex_zxc}
    print('周星驰的电影的平均星级数: ', getAveStar(getFilms(zxc)))
    print('周星驰和共同出演者: ')
    z_partners = getPartners(g, '周星驰')  # 为周星驰和共同出演者构成的顶点集合
    z_films = getFilms(z_partners)  # 为周星驰和共同出演者一共演过的电影构成的dict
    print("总人数: ", len(z_partners))
    print("总电影数: ", len(z_films))
    print("所演电影平均星级数: ", getAveStar(z_films))
    print("所演电影类别前三名: ", getType(z_films))
```

图 9: 周星驰相关计算主函数

1.5 最小生成树的探索

先求出 2-19 名连通分支包含的边，如图10，再对无权图使用简化版 Prim 算法得到最小生成树，返回生成树中包含的边，如图11。

```python
def getEdges(component):  # component:set, 集合中的元素为Vertex, 为一个连通分支
    # 返回值为int, 为该连通分支中的所有边的集合
    total_edges = set()
    for vertex in component:
        edges = set(vertex.getNbr().values())
        total_edges = total_edges | edges
    return total_edges
```

图 10: 计算联通分支边的个数

```python
def prim(start):  # start:Vertex, 为Prim算法的起始顶点
    # 因为这里的图中的边权重均为1, 所以Prim算法可以极大地简化
    # 对这种特殊情形, 只需要队列即可, 无需优先队列
    # 返回值为set, 为最小生成树中包括的边的集合
    vertex_q = Queue()
    min_edges = set()
    if start.getColor() == 'white':
        white = 'white'
        black = 'black'
    else:
        white = 'black'
        black = 'white'
    vertex_q.enqueue(start)
    start.setColor(black)
    while not vertex_q.isEmpty():
        current = vertex_q.dequeue()
        nbrs = current.getNbr()
        for nbr in nbrs:
            if nbr.getColor() == white:
                vertex_q.enqueue(nbr)
                nbr.setColor(black)
                min_edges.add(nbrs[nbr])
    return min_edges
```

图 11: 简化 Prim 算法

对所有顶点逐个使用 Prim 算法，观察最小生成树中的边数是否依赖起点选取。并将其和总边数进行对比，如图12。

```python
print('对规模前20的连通分支的最小生成树的探究: ')
# 并没有计算最大规模的连通分支, 因为对2-19名连通分支的研究已经足够得出结论
total_edge_nums = []  # 该列表中的元素为连通分支的总边数
min_edge_nums = []
# 该列表中的元素为集合, 每个集合对应这个连通分支以不同顶点为起点得到的最小生成树的边数的集合
# 使用集合是为了探究最小生成树的边数是否与顶点的选取有关
for i in range(1, 20):
    total_edge_nums.append(len(getEdges(top_bottom[i])))
    edgenums = set()
    for vertex in top_bottom[i]:
        edgenums.add(len(prim(vertex)))
    min_edge_nums.append(edgenums)
print('总边数: ', total_edge_nums)
print('最小生成树中边数: ', min_edge_nums)
```

图 12: 主函数中对最小生成树的探究

2 运行结果说明

代码运行输出结果如图13。

```
/usr/local/bin/python3.7 /Users/yangjiayu/PycharmProjects/DSA/FIN/FIN.py
连通分支总个数: 4577
scale: [84687, 45, 45, 31, 30, 28, 27, 23, 22, 22, 22, 21, 21, 21, 21, 19, 18, 18, 17, 17, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
types: [['剧情', '喜剧', '爱情'], ['恐怖', '战争', '惊悚'], ['纪录片', '科幻', '短片'], ['剧情', '犯罪', '爱情'], ['剧情', '爱情'], ['战争', '历史', '动作'], ['剧情', '纪录片', '恐怖'], ['音乐', '剧情'],
 ['剧情', '悬疑'], ['短片', '喜剧', '剧情'], ['剧情', '爱情', '犯罪'], ['剧情', '战争', '惊悚'], ['爱情', '情色', '剧情'], ['历史', '动作', '剧情'], ['喜剧', '剧情'], ['同性', '剧情', '爱情'], ['剧情',
 ['剧情', '同性', '短片'], ['剧情', '黑色电影', '鬼怪'], ['剧情'], ['短片', '家庭', '喜剧'], ['剧情'], ['惊悚', '恐怖'], ['剧情'], ['歌舞', '家庭'], ['短片', '爱情', '同性'], ['短片', '喜剧', '奇幻'], ['
 纪录片', '传记'], ['纪录片', '爱情', '传记'], ['爱情'], ['运动', '纪录片', '传记'], ['纪录片', '传记'], ['惊悚', '恐怖', '剧情'], ['纪录片', '传记'], ['西部', '短片', '爱情'], ['音乐', '歌舞', '悬疑'],
 ['纪录片', '历史'], ['纪录片', '历史'], ['喜剧'], ['惊悚', '恐怖']]
star: [6.87, 5.08, 8.13, 5.59, 7.94, 7.2, 7.78, 8.0, 7.62, 8.45, 7.3, 7.0, 5.75, 5.95, 7.15, 6.93, 7.83, 7.1, 6.65, 8.1, 7.9, 7.6, 6.4, 7.5, 7.1, 7.1, 6.2, 9.1, 8.9, 5.9, 9.2, 8.6, 6.2, 9.2,
  7.2, 8.8, 8.0, 7.5, 7.4, 6.0]
diameter: [-1, 4, 3, 5, 3, 1, 3, 1, 3, 2, 4, 2, 2, 2, 2, 2, 2, 3, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
周星驰的电影的平均星级数: 7.19
周星驰和共同出演者:
总人数: 302
总电影数: 3132
所演电影平均星级数: 6.26
所演电影类别前三名: ['动作', '剧情', '喜剧']
对规模前20的连通分支的最小生成树的探究:
总边数: [239, 826, 137, 201, 378, 142, 253, 125, 123, 55, 118, 174, 122, 126, 113, 123, 122, 73, 136]
最小生成树中边数: [{44}, {44}, {30}, {29}, {27}, {26}, {22}, {21}, {21}, {21}, {20}, {20}, {20}, {18}, {17}, {17}, {16}, {16}]

Process finished with exit code 0
```

图 13: 代码运行结果

连通分支总数： 4577

连通分支相关属性： 见表1和图14

表 1: 连通分支的属性

规模的名次	演员数	电影类别前三名	直径	规模的名次	演员数	电影类别前三名	直径
1	84687	剧情, 喜剧, 爱情	-1	4558	1	短片, 家庭, 喜剧	0
2	45	恐怖, 战争, 惊悚	4	4559	1	剧情	0
3	45	纪录片, 科幻, 短片	3	4560	1	惊悚, 恐怖	0
4	31	剧情, 犯罪, 爱情	5	4561	1	剧情	0
5	30	剧情, 爱情	3	4562	1	歌舞, 家庭	0
6	28	战争, 历史, 动作	1	4563	1	短片, 爱情, 同性	0
7	27	剧情, 纪录片, 恐怖	3	4564	1	短片, 喜剧, 奇幻	0
8	23	音乐, 剧情	1	4565	1	纪录片, 传记	0
9	22	剧情, 悬疑	3	4566	1	纪录片, 爱情, 传记	0
10	22	短片, 喜剧, 剧情	2	4567	1	爱情	0
11	22	剧情, 爱情, 犯罪	4	4568	1	运动, 纪录片, 传记	0
12	21	剧情, 战争, 惊悚	2	4569	1	纪录片, 传记	0
13	21	爱情, 情色, 剧情	2	4570	1	惊悚, 恐怖, 剧情	0
14	21	历史, 动作, 剧情	2	4571	1	纪录片, 传记	0
15	21	喜剧, 剧情	2	4572	1	西部, 短片, 爱情	0
16	19	同性, 剧情, 爱情	2	4573	1	音乐, 歌舞, 悬疑	0
17	18	剧情	2	4574	1	纪录片, 历史	0
18	18	剧情, 同性, 短片	3	4575	1	纪录片, 历史	0
19	17	剧情, 黑色电影, 鬼怪	2	4576	1	喜剧	0
20	17	剧情	1	4577	1	惊悚, 恐怖	0

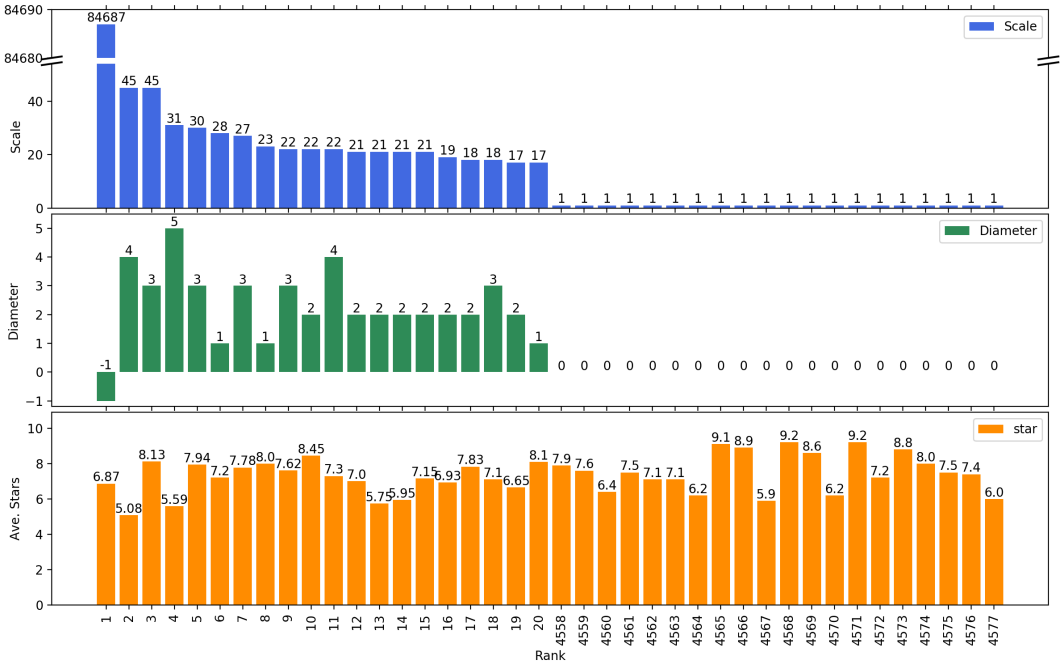

图 14: 连通分支相关属性柱状图

周星驰相关计算：

周星驰出演电影的平均星级数：7.19

周星驰和共同出演者总人数：302(包括周星驰)

一共出演电影数：3132

所演电影平均星级数：6.26

所演电影类别前三名：动作, 剧情, 喜剧

最小生成树相关计算： 首先可以从图13中看出最小生成树的边数与起点的选取无关。计算结果如表2。

表 2: 连通分支的属性

规模的名次	顶点数	总边数	最小生成树中边数	规模的名次	顶点数	总边数	最小生成树中边数
1	84687	\	\	11	22	55	21
2	45	239	44	12	21	118	20
3	45	826	44	13	21	174	20
4	31	137	30	14	21	122	20
5	30	201	29	15	21	126	20
6	28	378	27	16	19	113	18
7	27	142	26	17	18	123	17
8	23	253	22	18	18	122	17
9	22	125	21	19	17	73	16
10	22	123	21	20	17	136	16

可以看出最小生成树包含边的个数 = 顶点数 -1，这是因为生成树中根节点入度为 0，其余节点入度为 1，这也就解释了边的个数与起始点选取无关。同时可以看出，连通分支的规模和边的个数整体上有相关性，体现在规模最大的那几个边的数目也很多，但在规模比较接近的时候也没有绝对的相关性。