smart local moving algorithm

智能局部搜索算法

目录

CONTENTS

01
part

算法和启发式算法

1.1算法

算法可以理解为由基本运算及规定的运算顺序所构成的完整的解题步骤。或者看成按照要求设计好的有限的确切的计算序列，并且这样的步骤和序列可以解决一类问题。还必须满足算法的特征。

例子：

一个有趣的问题——“人、狼、羊过河”问题。有个人带着三只狼、三只羊，要过河去。有一条小船。船上除了运载一个人外，至多再载狼或羊中的任意两只。但难点是：当人不在场时，如果狼的数量大于等于羊的数量，那么羊会被狼吃掉。为了安全过河，你有什么办法呢?

解决它的算法有多个，其中一个解决方案是这样的：

开始，运一只狼过河，空船回来；

接着，运一只狼和一只羊再过河，到对岸后，再运两只狼回来；

然后，运两只羊过河，空船回来；

最后，分两次将狼全部运过河；

由此，过河问题就得以解决了。

1.2启发式算法

启发式算法是算法在实际应用过程中的技术性算法，因为在实际计算过程中，很有可能第一：无确定的目标函数；第二：无法找到真实的最优解；第三：计算代价非常大，现有的计算机技术都无法解决此类问题。所以人们就找到另一种情况，使得在可接受的计算成本内去搜寻最好的解，但求出来的解可能不是最好的，只能说是相对较好的，但是这个相对程度就不敢保证了。

02
part

模块度

社区：是网络中紧密连接的节点的集群。
模块度：社区结构强度。

求解模块度的公式： $Q = \frac{1}{2m}\sum_{i,j}\left(A_{ij} - \frac{k_i k_j}{2m}\right)\delta(c_i, c_j)$ $\quad \frac{1}{2m}\sum_{c}[\Sigma in - \frac{(\Sigma tot)^2}{2m}]$

公式涵义：社区内节点的连边数与随机情况下的边数之差

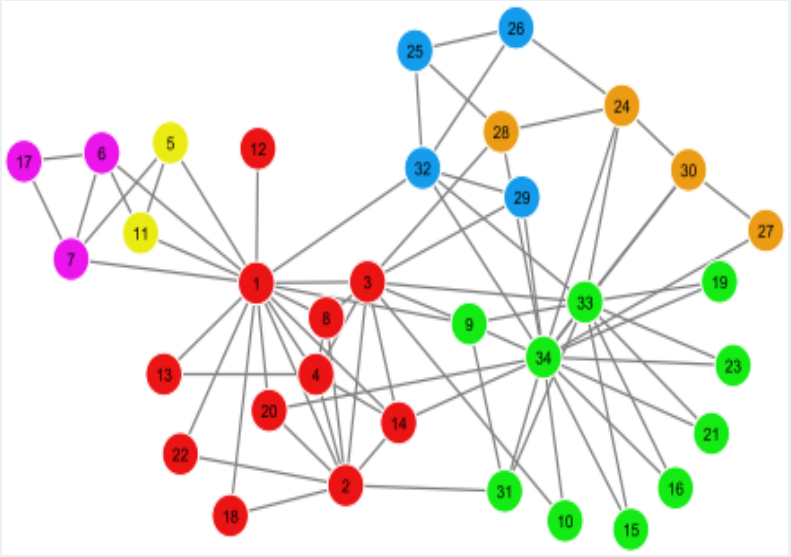
对公式的理解：Aij节点i和节点j之间边的权重，网络不是带权图时，所有边的权重可以看做是1。ci表示节点i所属的社区

$k_i = \sum_j A_{ij}$：表示所有与节点i相连的边的权重之和（度数）

$m = \frac{1}{2}\sum_{ij} A_{ij}$：表示所有边的权重之和（边的数目）

$$A_{ij} - \frac{k_i k_j}{2m} = A_{ij} - k_i \frac{k_j}{2m}$$

$$\Delta Q = [\frac{\sum_{in} + k_{i,in}}{2m} - (\frac{\sum_{tot} + k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2]$$



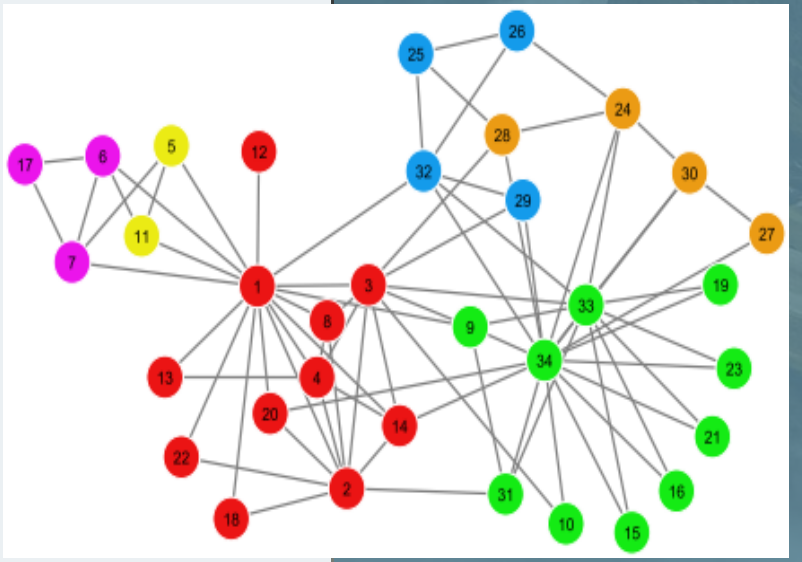
：Σin表示社区c内的边的权重之和，Σtot表示与社区c内的节点相连的边的权重之和

03
part

算法举例

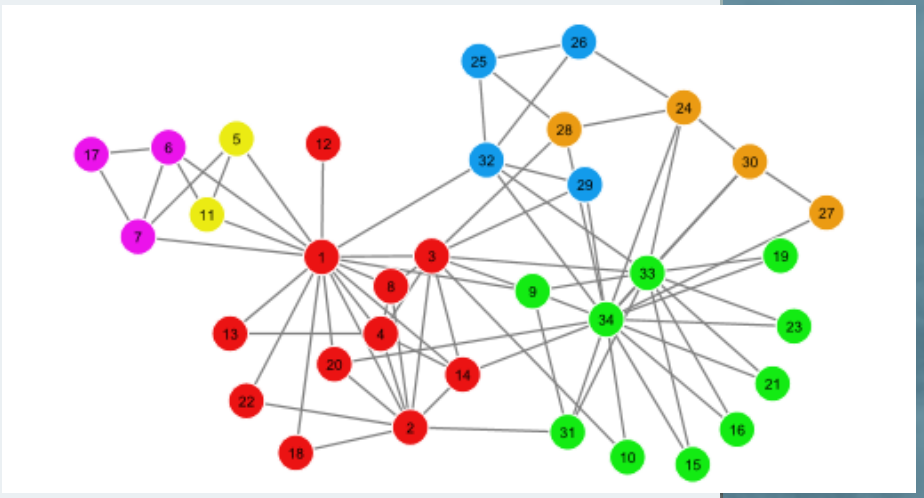
局部搜索启发式算法

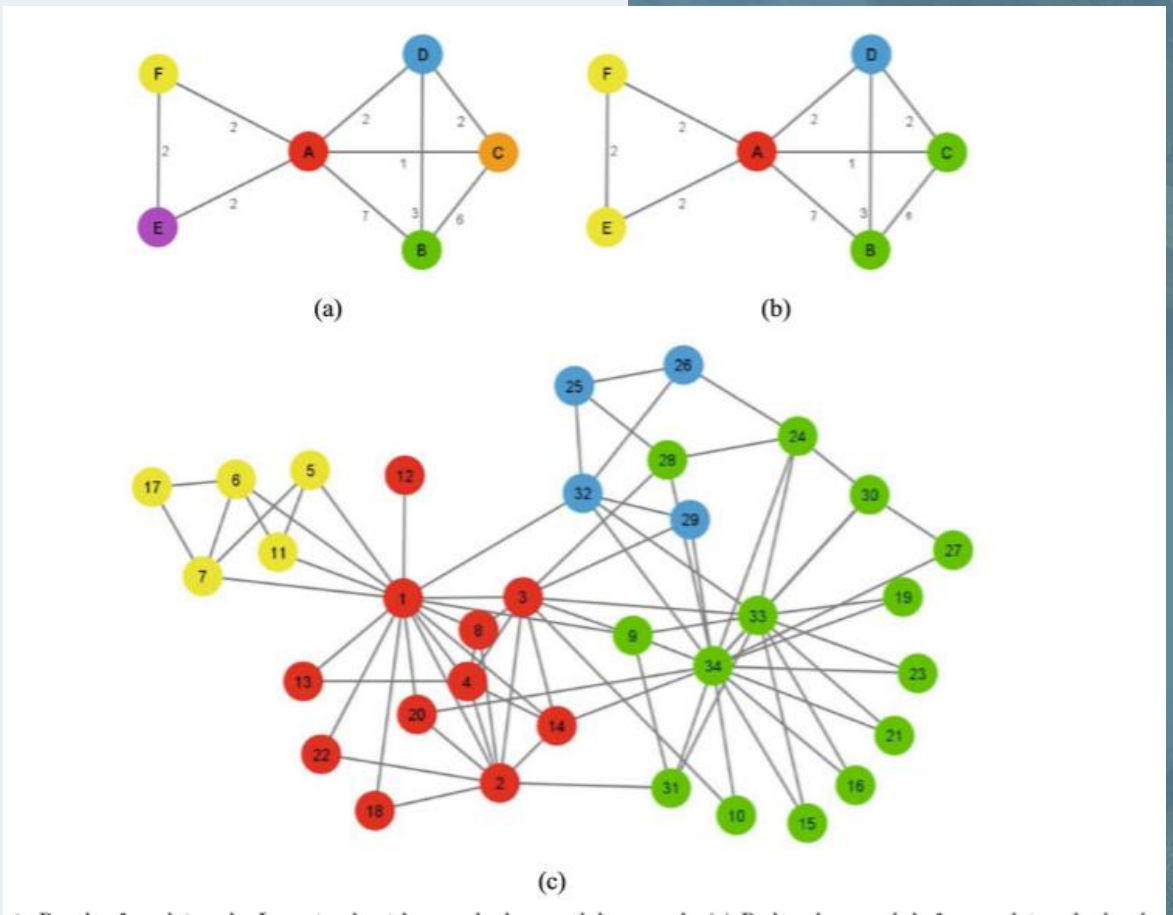
局部搜索启发法的思想是
将图中的每个节点看成一个独立的社区，次数社区的数目与节点个数相同，重复地将**单个节点**从一个社区移动到邻居节点所在的社区，这样每个节点的移动都会导致**模块性**的增加。局部移动启发式算法以随机顺序迭代网络中的节点。对于每个节点，确定是否可以通过将节点从其当前社区移动到不同的(可能是空的)社区来增加模块性。如果增加模块性确实是可能的，那么节点将被转移到社区，从而获得最大的模块性收益。局部移动启发式算法不断移动节点，直到达到这样一种情况，即没有进一步的可能性通过单独的节点移动来提高模块性。



Louvain算法思想

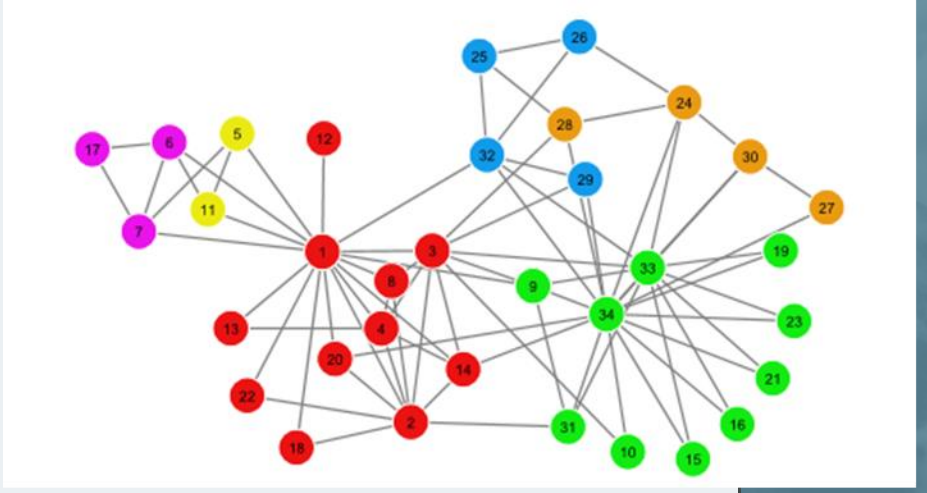
1）将图中的每个节点看成一个独立的社区，次数社区的数目与节点个数相同；
2）对每个节点i，依次尝试把节点i分配到其每个邻居节点所在的社区，计算分配前与分配后的模块度变化ΔQ，并记录ΔQΔQ最大的那个邻居节点，如果maxΔQ>0，则把节点i分配ΔQ最大的那个邻居节点所在的社区，否则保持不变；
3）重复2），直到所有节点的所属社区不再变化；
4）对图进行压缩，将所有在同一个社区的节点压缩成一个新节点，社区内节点之间的边的权重转化为新节点的环的权重，社区间的边权重转化为新节点间的边权重；
5）重复1）直到整个图的模块度不再发生变化。

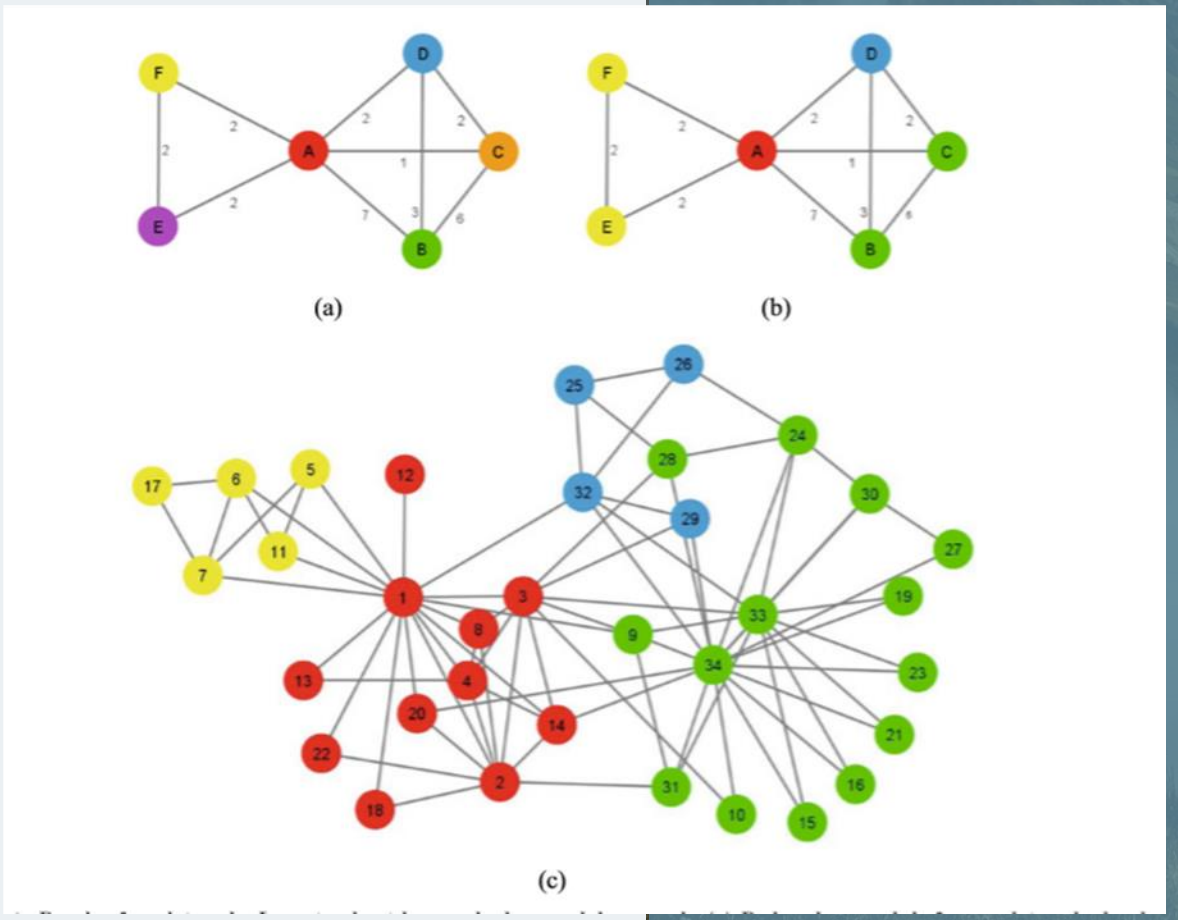




多级细化Louvain算法思想

1）将图中的每个节点看成一个独立的社区，次数社区的数目与节点个数相同；
2）对每个节点i，依次尝试把节点i分配到其每个邻居节点所在的社区，计算分配前与分配后的模块度变化ΔQ，并记录ΔQΔQ最大的那个邻居节点，如果maxΔQ>0，则把节点i分配ΔQ最大的那个邻居节点所在的社区，否则保持不变；
3）重复2），直到所有节点的所属社区不再变化；
4）对图进行压缩，将所有在同一个社区的节点压缩成一个新节点，社区内节点之间的边的权重转化为新节点的环的权重，社区间的边权重转化为新节点间的边权重；重复1)
5）对每个节点i，重复1直到整个图的模块度不再发生变化。





Louvain算法和多级细化迭代

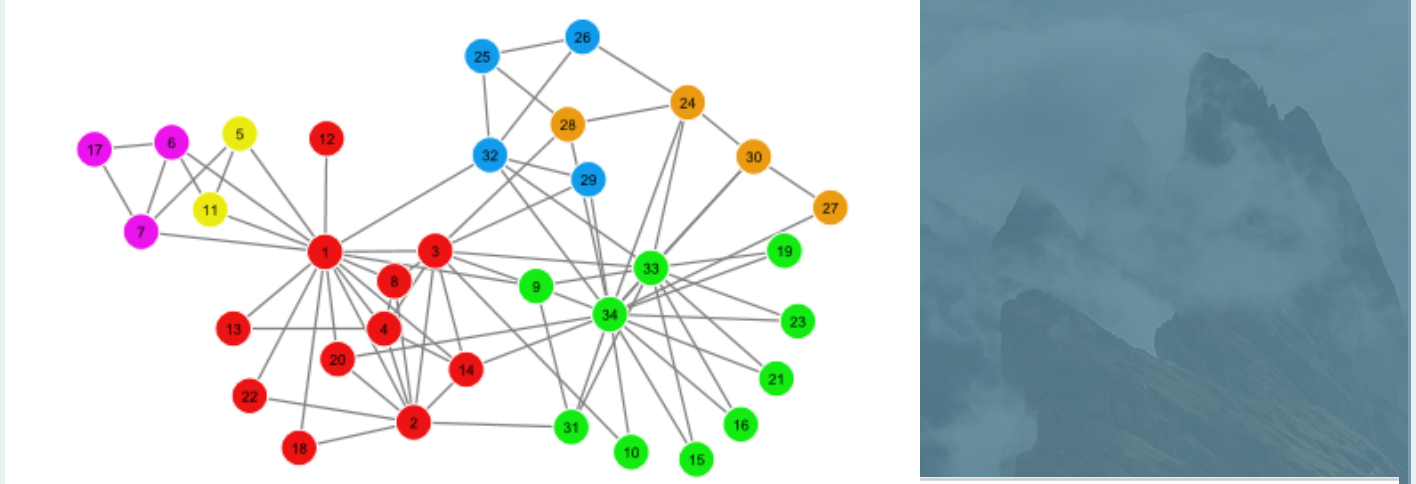
该算法包括两个阶段，这两个阶段重复迭代运行，直到网络社区划分的模块度不再增长。第一阶段合并社区，算法将每个节点当作一个社区，基于模块度增量最大化标准决定哪些邻居社区应该被合并。经过一轮扫描后开始第二阶段，算法将第一阶段发现的所有的社区重新看作节点，构建新的网络，在新的网络上迭代的进行第一阶段。当模块度不再增长时，得到网络的社区近似最优划分。 算法的基本步骤如下： 1）.初始化，将每个节点划分在不同的社区中。 2）.逐一选择各个节点，根据公式计算将它划分到它的邻居社区中得到的模块度增益。如果最大增益大于0，则将它划分到对应的邻居社区；否则，保持归属于原社区。 3）.重复步骤2，直到节点的社区不再发生变化。 4）.构建新图。新图中的点代表上一阶段产生的不同社区，边的权重为两个社区中所有节点对的边权重之和。重复步骤2，直到获得最大的模块度值。

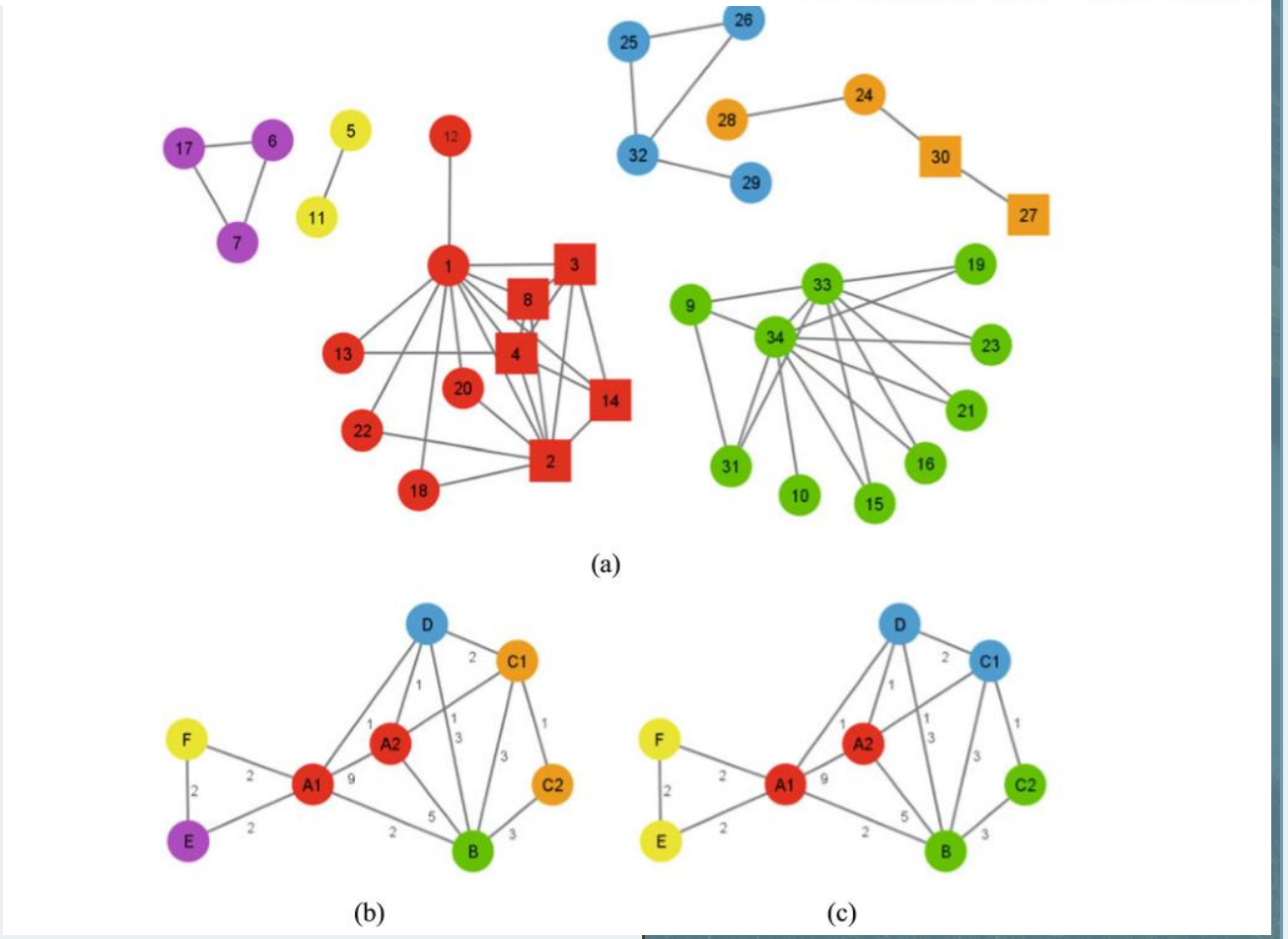
04
part

smart local moving算法

算法思想

1）将图中的每个节点看成一个独立的社区，次数社区的数目与节点个数相同；
2）对每个节点i，依次尝试把节点i分配到其每个邻居节点所在的社区，计算分配前与分配后的模块度变化ΔQ，并记录ΔQΔQ最大的那个邻居节点，如果maxΔQ>0，则把节点i分配ΔQ最大的那个邻居节点所在的社区，否则保持不变；
3）重复2），直到所有节点的所属社区不再变化；
4）在算法的第一次迭代中，我们从一个初始情况开始，在这个初始情况中，网络中的每个节点被分配给它自己的社区。在第二次迭代中，我们从将节点分配给在第一次迭代中获得的社区开始。在第三次操作中，在第二次迭代中获得的社区是我们的起点，以此类推。直到 ΔQ和节点所属社区不再变化为止。





05
part

Louvain算法程序实现

```
# coding=utf-8
import collections
import random
# 导入包括节点和节点间边权重的txt文件。然后放入字典中（运行G-view文件查看G形
式)
def load_graph(path):
    G = collections.defaultdict(dict)
    with open(path) as text:
        for line in text:
            vertices = line.strip().split()
            v_i = int(vertices[0])
            v_j = int(vertices[1])
            w = float(vertices[2])
            G[v_i][v_j] = w
            G[v_j][v_i] = w
    return G
class Vertex():
    def __init__(self, vid, cid, nodes, k_in=0):
        self._vid = vid
        self._cid = cid
        self._nodes = nodes
        self._kin = k_in  # 节点内部的边的权重
```

```
#Louvain算法
class Louvain():
    #给每个节点都分别划定一个不同的社区，并计算整个网络中的边数
    def __init__(self, G):
        self._G = G
        self._m = 0  # 边数量
        self._cid_vertices = {}  # 需维护的关于社区的信息(社区编号,其中包含的节点编号的集合)
        self._vid_vertex = {}  # 需维护的关于节点的信息(结点编号，相应的Vertex实例)
        for vid in self._G.keys():
            self._cid_vertices[vid] = set([vid])
            self._vid_vertex[vid] = Vertex(vid, vid, set([vid]))
            self._m += sum([1 for neighbor in self._G[vid].keys() if neighbor > vid])   # 总边数
```

#Louvain算法

```
#局部搜索算法：
    def first_stage(self):
        mod_inc = False  # 用于判断算法是否可终止
        visit_sequence = self._G.keys()
        random.shuffle(list(visit_sequence))
        while True:
            can_stop = True  # 第一阶段是否可终止
            #权重
            for v_vid in visit_sequence:
                v_cid = self._vid_vertex[v_vid]._cid
                k_v = sum(self._G[v_vid].values()) + self._vid_vertex[v_vid]._kin
                cid_Q = {}
                for w_vid in self._G[v_vid].keys():
                    w_cid = self._vid_vertex[w_vid]._cid
                    if w_cid in cid_Q:
                        continue
                    else:
                        tot = sum(
                            [sum(self._G[k].values()) + self._vid_vertex[k]._kin for k in self._cid_vertices[w_cid]])
                        if w_cid == v_cid:
                            tot -= k_v
                        k_v_in = sum([v for k, v in self._G[v_vid].items() if k in self._cid_vertices[w_cid]])
                        delta_Q = k_v_in - k_v * tot / self._m  # 由于只需要知道delta_Q的正负，所以少乘了1/(2*self._m)
                        cid_Q[w_cid] = delta_Q
                cid, max_delta_Q = sorted(cid_Q.items(), key=lambda item: item[1], reverse=True)[0]
                if max_delta_Q > 0.0 and cid != v_cid:
                    self._vid_vertex[v_vid]._cid = cid
                    self._cid_vertices[cid].add(v_vid)
                    self._cid_vertices[v_cid].remove(v_vid)
                    can_stop = False
                    mod_inc = True
            if can_stop:
                break
        return mod_inc
```

#Louvain算法

```
#把社区压缩为一个节点进行第一过程的运算
    def second_stage(self):
        cid_vertices = {}
        vid_vertex = {}
        for cid, vertices in self._cid_vertices.items():
            if len(vertices) == 0:
                continue
            new_vertex = Vertex(cid, cid, set())
            #重新查看社区节点和权重
            for vid in vertices:
                new_vertex._nodes.update(self._vid_vertex[vid]._nodes)
                new_vertex._kin += self._vid_vertex[vid]._kin
                for k, v in self._G[vid].items():
                    if k in vertices:
                        new_vertex._kin += v / 2.0
            cid_vertices[cid] = set([cid])
            vid_vertex[cid] = new_vertex
        G = collections.defaultdict(dict)
        for cid1, vertices1 in self._cid_vertices.items():
            if len(vertices1) == 0:
                continue
            for cid2, vertices2 in self._cid_vertices.items():
                if cid2 <= cid1 or len(vertices2) == 0:
                    continue
                edge_weight = 0.0
                for vid in vertices1:
                    for k, v in self._G[vid].items():
                        if k in vertices2:
                            edge_weight += v
                if edge_weight != 0:
                    G[cid1][cid2] = edge_weight
                    G[cid2][cid1] = edge_weight
        self._cid_vertices = cid_vertices
        self._vid_vertex = vid_vertex
        self._G = G
```

```
#Louvain算法
  def get_communities(self):
        communities = []
        for vertices in self._cid_vertices.values():    #遍历社区中的不为空的节点（社区压缩而来）
            if len(vertices) != 0:
                c = set()
                for vid in vertices:                #社区中：遍历被压缩社区的节点
                    c.update(self._vid_vertex[vid]._nodes)
                communities.append(c)               #按照遍历顺序在列表中加入压缩社区中的节点编号
        return communities
    def execute(self):
        iter_time = 1
        while True:
            iter_time += 1
            mod_inc = self.first_stage()
            if mod_inc:
                self.second_stage()
                self.first_stage()
            else:
                break
        return self.get_communities()
```

```
#主程序
if __name__ == '__main__':
    G = load_graph(r'D:\E盘\可视化\text.txt')
    algorithm = Louvain(G)
    communities = algorithm.execute()
    # 按照社区大小从大到小排序输出
    communities = sorted(communities, key=lambda b: -len(b))  # 按社区大小排序
    count = 0
    for communitie in communities:
        count += 1
        print("社区", count, " ", communitie)
```

#演示文件形式和结果

```
1 2 0.5
1 4 1
2 3 0.7
2 5 0.2
3 5 1
1 2 0.5
6 7 0.9
8 3 0.3
3 7 0.4
5 6 0.2
4 3 0.3
4 6 0.8
```

```
D:\ruanjian\pycharm\luvain\venv\Scripts\python.exe "D:/ruanjian/pycharm/luvain/louvain  algorithm.py"
社区 1   {8, 2, 3, 5}
社区 2   {1, 4, 6, 7}

Process finished with exit code 0
```