目录

1. 实验说明

1. 影响最大化介绍

社交网络可以抽象的用图 G(V,E,P)来描述，V 是结点集，E 是边集，P 是边的概率集。一个用户就是一个节点 v，用户与用户之间的关系就是边 e，每条边都有一条概率 p，感染会根据边上的概率进行传播。

在此描述下，影响最大化问题可以分为两类：

1. 给定节点数，选择出 k 个节点作为种子集，使得种子集能影响的节点数尽可能多。
2. 给定所要求产生的影响力，找到满足条件的最小节点集合

在本实验中，研究的问题为第一类，在实验种，分别选取 5、10、15、20 个节点，使得选取的节点，在给定的节点个数中，可以影响尽可能多的节点数。

1.2 独立级联模型

独立级联模型（Independent Cascade Model）简称 IC 模型，是一种概率传播模型，基本原理描述如下：

在社交网络 G=(V,E)中，点集 V 中的节点具有两种状态，一种是已感染状态，一种是未感染状态。每一个处于已感染状态的结点，都有按边权的概率感染每个与之相连的结点的机会。每个节点只有一次被相邻节点尝试感染的机会。在初始状态时，仅有种子节点集合中的点处于已感染状态，这些感染的点按照概率尝试感染与之相邻的点，直至集合中所有已感染的点，都尝试感染过与之相邻的所有点，并且无新增的已感染点。

1.3 线性阈值模型

线性阈值模型与独立级联模型的区别表现在，在线行阈值模型中，每个节点 v 受到相邻已感染节点的影响的叠加，当超过某个阈值时，该节点被感染。

2. 实验方法

2.1 寻找自洽排名

本次实验中采用的是独立级联模型。

实验中首先采用迭代的方式，通过寻找自适应的排名来计算给定图的影响最大化的点集[1]。 具体算法如算法 1 和算法 2。

算法 1：

过程：更新边界影响力 Calculate Mr(r)

输入：Mr(r)，图 G(V,E)，其中边的权重 $p(v_{rj},v_{ri})$ 表示从 v_{rj} 直接感染 v_{rj} 的概率

输出：更新之后的 Mr(r)

1. for i = 1 to n do
2. $\quad$ Mr(v_{ri}) ← 1
3. end for
4. for i = 2 to n do
5. $\quad$ for j = 1 to i do
6. $\quad\quad$ Mr(v_{rj}) ← Mr(v_{rj}) + p(v_{rj},v_{ri}) * Mr(v_{ri})
7. $\quad\quad$ Mr(v_{ri}) ← (1 - p(v_{rj},v_{ri})) * Mr(v_{ri})
8. $\quad$ end for
9. end for
10. output Mr

算法 2：

过程：输出边界影响序列 IMRank(r)

输入：图 G(V,E)，初始排名序列 r

输出：边界影响力排名 r

1. $r^{(0)}$ ← r
2. t ← 0
3. 设置初始时所有节点的边界影响力均为 1
4. 设置初始时所有节点的排名 $r^{(0)}$(全为 1)
5. repeat
6. $\quad$ t ← t + 1

7. 调用更新边界影响力过程，将边界影响力更新为 Mr(t)

8. 通过对边界影响力进行排序，生成新的边界影响力的排名 $r^{(t)}$

9. until $r^{(t)} = r^{(t-1)}$

10. output $r^{(t)}$

其中采用的符号表如图 2.1 所示

Notation	Description
v_i	a node with index i
r_i	the index of node with rank i with respect to a given ranking r
$S = \{v_1, v_2, \ldots, v_n\}$	a set of nodes
$I(S)$	expected number of nodes eventually activated by set S
$M(v\|S)$	marginal influence spread by adding node v into a seed set S
$M_r(v_{r_i})$	ranking-based marginal influence, short for $M(v\|\{v_{r_1}, v_{r_2}, \ldots, v_{r_{i-1}}\})$
$p(v_i\|\{v_1, v_2, \ldots, v_{i-1}\})$	probability that v_i is activated given that a collection of nodes $\{v_1, v_2, \ldots, v_{i-1}\}$ are already activated
$\eta_r(v_i, v_j)$	influence score that node v_i sends to node v_j with respect to a given ranking r
$d(v_j, v_i)$	a simple path starting from v_j and ending at v_i, i.e., $\{w_1 = v_j, w_2, \ldots, w_n = v_i\}$
$d_r(v_j, v_i)$	influence path, which is a simple path where v_j is the only node ranked higher than v_i on the path
$\rho_r(v_i, v_j)$	probability that v_i is activated by v_j through any influence path, with respect to a given ranking r
l	maximal length of all influence paths to account into

图 2.1 符号表

2.2 加上贪婪算法的改进

在实验过程中，初始排名不同时，计算所得到的排名可能不一致。在实验中采用贪婪算法解决这个问题。实验中采用两个完全相反的排名初始序列，根据者两个不同的初始排名，可能得到两个不完全相同的 top k 点序列。对于这两个序列中相同的前一部分，认为对给定数据集而言，这重合的一部分一定在 top k 点序列中。对于后面不相同的部分，则考虑是因为数据中边的权重设置，使得该算法不能有效区分这几个点的边界影响力。因此对于这些点，采用模拟爆发一定次数，来计算该节点的影响力，然后将较大影响力的点，加入 top k 点序列，直至序列长度达到 k。具体操作为：

有两个根据不同初始排名得到的两个不同的序列 t1、t2。采用两个指针 i、j 分别扫描 t1、t2。若指向相同的结点，则说明此结点排序正确，则将 i 和 j 都指向列表中下一个元素。若指向不同的结点，则分别采用模拟爆发的方法来计算将元素加入已经正确排序的结点的序列的影响力，然后选择较大的影响力对应的节点，认为该节点更适合加入 top k 点序列。

3. 实验过程

3.1 读取文件

首先从磁盘中读取数据文件，将其用邻接矩阵的形式保存下来

```
def load_file(file_name="DUNF with Weights.txt"):  # 读取文件，文件形式为三列，分别为父节点、子节点、权重，返回图的邻接矩阵
    with open(file_name, 'r') as fp:
        lines = fp.readlines()
        num = int(max(max([float(_) for _ in line.split()]) for line in lines))  # 获取节点的个数
        data = np.zeros((num+1, num+1), np.float32)
        for line in lines:
            data[int(line.split()[0])][int(line.split()[1])] = float(line.split()[2])
    return data
```

3.2 IMRank 算法实现

采用 last to first 策略，算法 1 的实现代码如下

```
def last_to_first(mi_rank, data):  # 输入数据和现有排名，计算下一次迭代产生的边界影响力
    mi_list = [1 for _ in range(len(mi_rank))]  # 存放每个结点对应的 Mr(r)，表示该点的边界影响力
    for i in range(len(mi_rank)-1, -1, -1):  # 取出排名为 i 的结点
        for j in range(i):
            mi_list[mi_rank[j]] = mi_list[mi_rank[j]] + data[mi_rank[j]][mi_rank[i]] * mi_list[mi_rank[i]]
            mi_list[mi_rank[i]] = (1 - data[mi_rank[j]][mi_rank[i]]) * mi_list[mi_rank[i]]
    return mi_list
```

算法 2 的实现代码如下

```
def im_rank(data, init_rank, top_k=20, iter_num=40):  # 返回的边界影响力是全部点的影响力

    mi_rank_1 = [_ for _ in init_rank]  # 第零次迭代产生的排名
    iter_count = 0  # 迭代次数
    equal = False
    while not equal and iter_count <= iter_num:  # 终止迭代条件（满足其一）：1.迭代不导致结果变动  2.达到一定迭代次数
        tmp = last_to_first(mi_rank_1, data)
        mi_rank_2 = np.argsort(np.array(tmp))[::-1].tolist()
```

```
        equal = (np.array(mi_rank_1) == np.array(mi_rank_2)).all()  # 如果两次迭代得到的边界影响最大的 k 个点相同，则停止迭代
        mi_rank_1 = [_ for _ in mi_rank_2]
        iter_count = iter_count + 1
    return mi_rank_2[:top_k:]
```

3.3 模拟爆发

采用 IC 模型模拟爆发，返回感染的字典

```
def ic_model(data, seed_list, total=100):  # 输入带权图、爆发种子结点、爆发次数，返回键为感染点、值为感染几率的字典
    active_dict ={}
    for break_num in range(total):
        active_set = np.zeros(len(data), np.int)  # 标记点的感染状态，为 0 表示未感染，为 1 表示感染
        child_queue = queue.Queue()  # 待考察点
        for _ in seed_list:
            child_queue.put(_)
        while not child_queue.empty():  # 当队列不为空时，即存在已感染的点的孩子结点尚未考察
            i = child_queue.get()  # 本次考察的点
            for j in range(len(data)):  # 对于该点的每个孩子
                if j != i and data[i][j] != 0:  # 不为自己，也不为零
                    if active_set[i] != 1 and data[i][j] > random.random():  # 在此次传染中，由未感染变为感染
                        if j not in active_dict:
                            active_dict[j] = 1
                        else:
                            active_dict[j] += 1
                        active_set[i] = 1  # 设置为被感染的状态
                        child_queue.put(j)  # 加入待考察队列
    for _ in active_dict:
        active_dict[_] = active_dict[_] / total
    return active_dict
```

3.4 算法改进

采用贪婪算法，对于两个不同的序列，不断的从这两个序列中挑选影响最大的结点，直到已经挑选了 k 个点时停止。

```python
def greedy_mi_influence(order1, order2, k, data):  # 传入两次迭代产生的结果，返回模拟爆发之后的序列
    result = []
    i = 0
    j = 0
    while len(result) < k:
        if order1[i] in result:
            i = i + 1
            continue
        if order2[j] in result:
            j = j + 1
            continue
        if order1[i] == order2[j]:  # 如果两处排名相同，则说明排名没有波动，则加入序列
            result.append(order1[i])
            i = i + 1
            j = j + 1
        else:
            tmp1 = sum(ic_model(data, result+order1[i:i+1:1]).values())  # 计算 order1[i]的边界影响力
            tmp2 = sum(ic_model(data, result+order2[j:j+1:1]).values())  # 计算 order2[j]的边界影响力
            if tmp1 > tmp2:
                result.append(order1[i])
                i = i + 1
            else:
                result.append(order2[j])
                j = j + 1
    return result
```

结合两种方法，计算排名

```python
def combine_rank(data, top_k=20, iter_num=30):  # 综合采用两种方法计算影响力排名
    init_rank_1 = [len(data) - i - 1 for i in range(len(data))]  # 生成的初始排名 1
    init_rank_2 = [i for i in range(len(data))]  # 生成的初始排名 2
    mi_rank_1 = im_rank(data, init_rank_1, top_k, iter_num)
    mi_rank_2 = im_rank(data, init_rank_2, top_k, iter_num)
    mi_order = greedy_mi_influence(mi_rank_1, mi_rank_2, top_k, data)  # 迭代可能未收敛，采用贪婪算法再次处理
    return mi_order
```

调用上面函数，计算最大影响力的函数

```
1.
2.  def max_influence(data, top_k=5, iter_num=30, break_num=10):  # 返回影响力最大的 k 个点，和这 k 个点在模拟爆发时的平均感染点数
3.      begin_time = time()  # 开始计算边界影响力的时间
4.      order = combine_rank(data, top_k, iter_num)
5.      end_mi_cal = time()  # 边界影响力计算完成的时间
6.      duration1 = end_mi_cal - begin_time
7.      break_result = ic_model(data, order, break_num)  # 传入用于爆发的种子时，应该从 order 列表中选取前 k 个，k 的选取为计算边界影响最大的的个数
8.      duration2 = time() - end_mi_cal  # 计算模拟爆发所用时长
9.      print("计算影响力最大的%d 个点用时：%fs\n 计算这%d 点模拟爆发用时：  %fs" % (top_k, duration1, top_k, duration2))
10.     active_count = 0
11.     for _ in break_result:
12.         active_count += break_result[_]
13.     return order, break_result
```

3.5 主函数

运行主函数。计算的结果不方便

```
1.  if __name__ == "__main__":
2.      Data = load_file("DUNF with Weights.txt")
3.      seedSetList = []  # 存放每次计算得到影响力最大的 k 个点
4.      activeSetList = []  # 存放感染的
5.      activeCountList = []  # 存放平均感染的点的个数
6.      iterNum = 30  # 计算边界影响力时的迭代次数上界
7.      breakNum = 100  # 模拟爆发点的次数
8.      for topK in list(range(5, 25, 5)):
9.          seedSet, activeSet = max_influence(Data, topK, iterNum, breakNum)
10.         seedSetList.append(seedSet)
11.         activeSetList.append(activeSet)
12.         activeCountList.append(sum(activeSet.values()))
13.
14.     print("end")
```

4. 实验结果

4.1 中间变量

在调试模式下，查看运行的中间变量，调用贪婪算法之后得到的结果如图 4.1 到图 4.4 所示。其中 init_rank_1 和 init_rank_2 表示两个不同的初始排

名，mi_list_1 和 mi_list_2 分别对应这两个初始排名对应的 IMRank 算法得到的结果，mi_order 对应调用贪婪算法之后得到的排名。

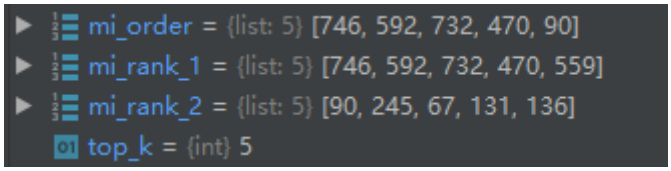

图 4.1 Top 5 个结点

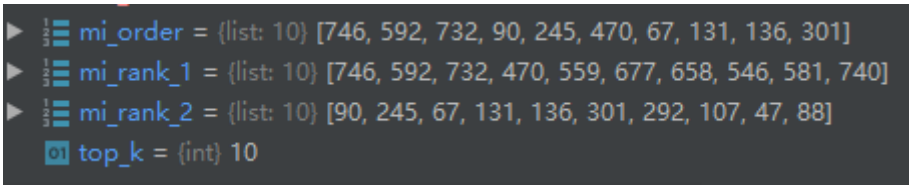

图 4.2 Top 10 个结点

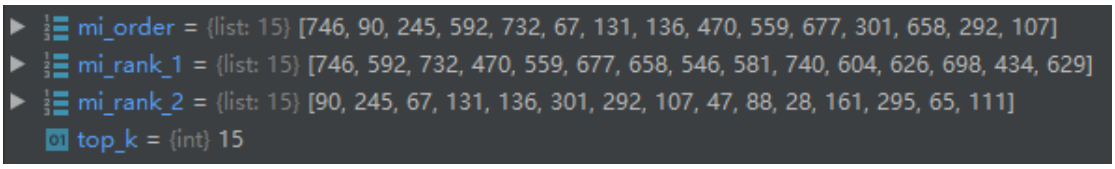

图 4.3 Top 15 个结点

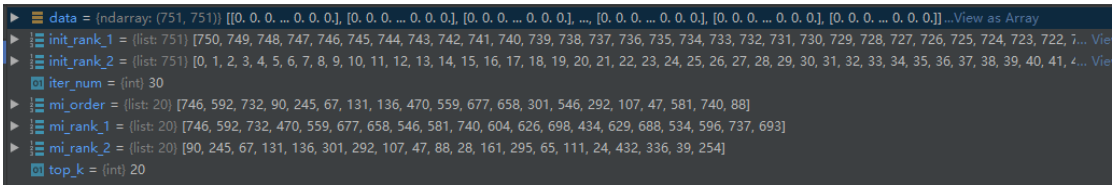

图 4.4 Top 20 个结点

4.2 最终结果

分别计算 top 5、10、15、20 个点集时的感染情况，采用如下记录方式：初始感染的种子节点为计算所得的 top 节点。计算结果不包含初始被选中的节点

$$\sum \frac{\text{每个点在模拟爆发中感染的次数}}{\text{模拟爆发的总次数}}$$

某次运行所得结果如图 4.5 所示，与之对应的时间如图 4.6 所示。

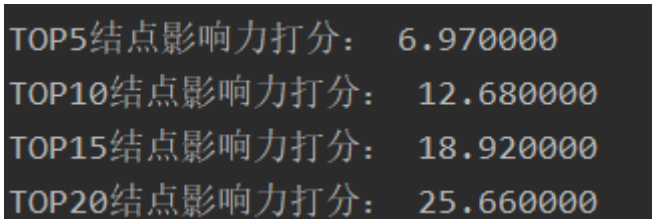

图 5.5　实验结果

图 4.6　运行时间