雪堆博弈最小节点覆盖

1 问题描述

验证结论：当雪堆博弈满足 r<1/kmax 时 (kmax 为网络节点的最大度)，网络博弈的纳什均衡中的采用合作策略的节点构成极小节点覆盖。网络结构可自定，节点数目不少于 10。

节点的初始状态可随机定为 Cooperator 或者 Defector, 按照某种给定顺序 (例如 1,2,⋯,10) 依次检查每个节点，是否改变其状态可以获得更大收益，如果是则改变状态，否则不改变，直到所有节点都不再改变状态为止。验证合作的节点集合是否是极小节点覆盖。

2 问题背景

2.1 极小节点覆盖

网络节点最小覆盖问题 (MVCP) 是一个著名组合优化问题，其目的在于找出给定网络的最小节点集合以覆盖所有的边。其中，若节点集合中去掉任何一个点，就不能覆盖网络所有边，则称此时为极小节点覆盖。

2.2 雪堆博弈

雪堆博弈所描述的情景是：在一个风雪交加的夜晚，两人相向而来，被一个雪堆所阻。他们可以选择下车铲雪 (合作 C)，或者都不铲雪 (背叛 D)。如果两人都不铲雪，两人就都无法通行；如果一人铲雪一人不铲，则铲雪者付出了劳动，背叛者白占了成果。

设铲雪所需的劳动价值为 r，通行所获得的收益为 1，则雪堆博弈的收益矩阵收益矩阵如下表所示[1]：

表 1: 雪堆矩阵收益矩阵表

参与者	参与者 B	
A	C（合作）	D（背叛）
C（合作）	(1,1)	(1-r,1+r)
D（背叛）	(1+r,1-r)	(0,0)

2.3 纳什均衡

纳什均衡指的是在给定别人策略的情况下，没有人愿意单方面改变自己的策略，从而打破这种均衡。

3 算法思路

为了寻找到当前网络博弈的纳什均衡，需完成以下几个步骤:

1. 随机给每个节点初始状态设置为 C 或 D(C 表示合作，D 表示背叛)
2. 依次对每个节点计算其采用 C 或者 D 的收益，改变其策略使其收益最大
3. 重复 2 过程到每个节点的状态不在改变

算法流程如图 1 所示：

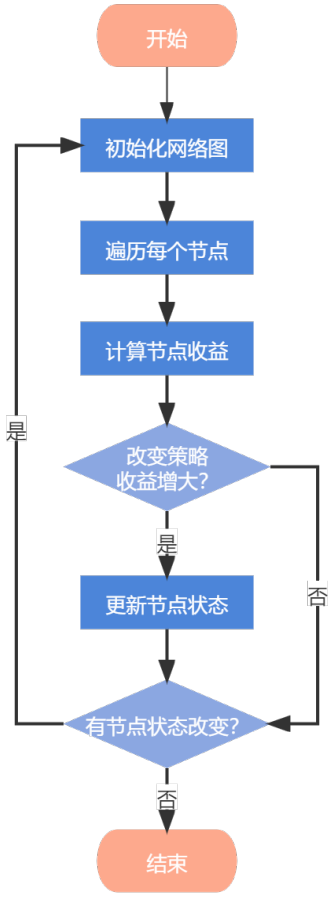


图 1: 算法流程图

4 实验步骤

4.1 网络结构图

本实验网络设置了 15 个节点，其结构图如图 2 所示：

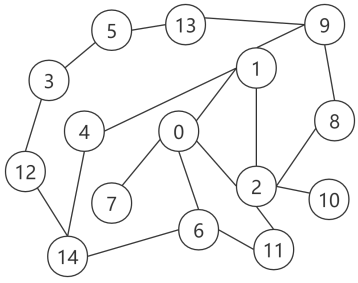
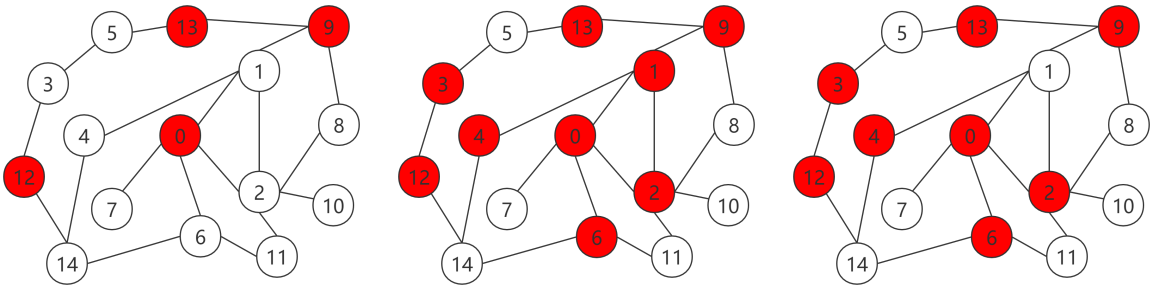


图 2: 网络结构图

该网络中 kmax 为 5，定理存在条件 r<1/kmax，因此实验设定 r 为 0.1。

4.2 第一次求解

首先初始化网络，每个节点都有 50% 的概率选择合作 (C) 或背叛 (D)，之后根据算法流程进行更新，初始状态，中间过程，最终结果如下图所示：



(a) 初始状态　　(b) 中间状态　　(c) 最终状态

图 3: 第一次求解网络过程图

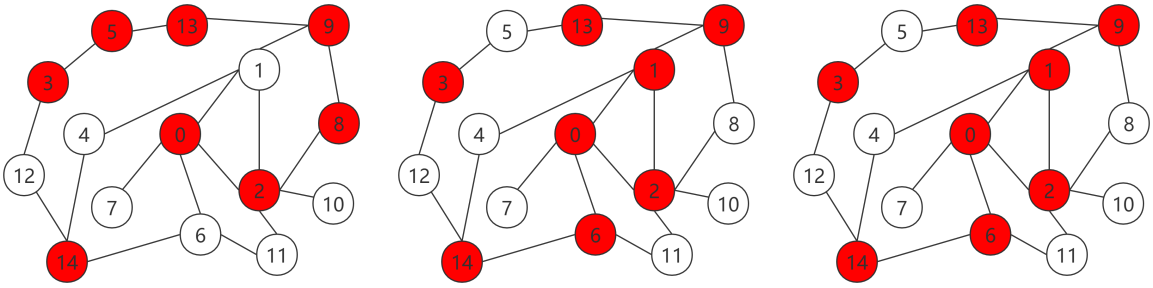
图中，红色代表决策 C，白色代表决策 D。可以发现，最终结果实现了极小节点覆盖。各节点的决策收益如表 2 所示：

表 2: 各节点决策收益表

节点	决策	收益	节点	决策	收益	节点	决策	收益
0	C	0.950	5	D	1.100	10	D	1.100
1	D	1.100	6	C	0.933	11	D	1.100
2	C	0.920	7	D	1.100	12	C	0.950
3	C	0.950	8	D	1.100	13	C	0.900
4	C	0.900	9	C	0.900	14	D	1.100

4.3 第二次求解

第二次求解过程和第一次类似，初始状态，中间过程，最终结果如下图所示：



(a) 初始状态　　(b) 中间状态　　(c) 最终状态

图 4: 第二次求解网络过程图

可以发现，由于初始生成情况较好，此次求解很快，中间状态和最终状态一致。同样，和第一次求解对比，可以发现极小节点覆盖的解不唯一，存在多解情况。各节点的决策收益如表 3 所示：

表 3: 各节点决策收益表

节点	决策	收益	节点	决策	收益	节点	决策	收益
0	C	0.975	5	D	1.100	10	D	1.100
1	C	0.975	6	C	0.967	11	D	1.100
2	C	0.940	7	D	1.100	12	D	1.100
3	C	0.900	8	D	1.100	13	C	0.900
4	D	1.100	9	C	0.950	14	C	0.933

5 总结

本次实验验证了结论：当雪堆博弈满足 r<1/kmax 时 (kmax 为网络节点的最大度)，网络博弈的纳什均衡中的采用合作策略的节点构成极小节点覆盖。与此同时，该验证过程还反映出个体在博弈中寻求最大利益的属性，会使得整个群体在数次迭代后趋向于一个对整体较为有利的情况，使得整个群体得到最大的利益。

参考文献

[1] 吴建设, 焦李成, 皎魁, 等. 基于雪堆博弈进化的复杂网络节点覆盖方法:,CN105050096A[P]. 2015.

6 程序代码

python 代码：

```python
import random

# r<1/kmax
r = 0.1

# 收益矩阵
rewardMat = {
'C': {'C': (1, 1), 'D': (1 - r, 1 + r)},
'D': {'C': (1 + r, 1 - r), 'D': (0, 0)}
}


# 节点类，每个节点保存节点状态，邻居节点列表和邻居节点个数
class Node:
def __init__(self):
if random.random() < 0.5:
self.state = 'C'
else:
self.state = 'D'
self.value = 0
self.all_value = 0
```

```
22  self.neighbour_number = 0
23  self.nb = list()
24
25
26  # 博弈网络  迭代更新节点状态，直至收敛
27  class Net:
28  def __init__(self, n):
29  self.numbers = n
30  self.nodes = list()
31  self.edges = list()
32  self.reward = 0
33  self.initNode()
34
35  # 打印状态
36  def printState(self):
37  self.getAllReward()
38  print('各个节点的决策和其收益：')
39  for i in range(self.numbers):
40  print('Noede%d:%c ==> reward: %f ' %
41  (i, self.nodes[i].state, self.nodes[i].value))
42  print("博弈网络总体回报为:", self.reward)
43
44  # 生成n个结点
45  def initNode(self):
46  for _ in range(self.numbers):
47  tmp_node = Node()
48  self.nodes.append(tmp_node)
49  print(tmp_node.state)  # 打印初始状态
50
51  # 根据传入的边的列表构建网络
52  def buideNet(self, es):
53  for e in es:
54  self.edges.append(e)
55  self.updateNb()
56
57  # 更新每个节点的邻居
58  def updateNb(self):
59  for a, b in self.edges:
60  a.nb.append(self.nodes.index(b))
61  b.nb.append(self.nodes.index(a))
62  for i in range(self.numbers):
63  self.nodes[i].neighbour_number = len(self.nodes[i].nb)
64
65  # 计算每个结点的平均收益,每条边的收益和/边数
```

```
66  def calValue(self):
67  for i in range(self.numbers):
68  self.nodes[i].all_value = 0
69  for a, b in self.edges:
70  a.all_value += rewardMat[a.state][b.state][0]
71  b.all_value += rewardMat[a.state][b.state][1]
72  for i in range(self.numbers):
73  self.nodes[i].value = self.nodes[i].all_value / self.nodes[i].
        neighbour_number
74
75  # 每次改变一个参与人的策略，增加自己的收益
76  # 根据是否有节点发生变化返回是否发生改变的标志
77  def updateState(self):
78  self.calValue()
79  flag = False
80  for i in range(self.numbers):
81  if self.nodes[i].state == 'C':
82  reward1 = self.getReward(i)
83  self.nodes[i].state = 'D'
84  reward2 = self.getReward(i)
85  if reward2 <= reward1:
86  self.nodes[i].state = 'C'
87  continue
88  flag = True
89  elif self.nodes[i].state == 'D':
90  reward1 = self.getReward(i)
91  self.nodes[i].state = 'C'
92  reward2 = self.getReward(i)
93  if reward2 <= reward1:
94  self.nodes[i].state = 'D'
95  continue
96  flag = True
97  return flag
98
99  # 计算单个结点的收益，某节点的所有边的收益/邻居个数
100 def getReward(self, i):
101 all_value = 0
102 for s in self.nodes[i].nb:
103 all_value += rewardMat[self.nodes[i].state][self.nodes[s].state][0]
104 value = all_value / self.nodes[i].neighbour_number
105 return value
106
107 # 获得博弈网络的总收益
108 def getAllReward(self):
```

```
109  for a, b in self.edges:
110  self.reward += sum(rewardMat[a.state][b.state])
111  return self.reward
112
113
114  if __name__ == '__main__':
115  net = Net(15)
116  # 博弈网络中边的集合
117  edge_list = [(0, 1), (0, 2), (0, 6), (0, 7),
118  (1, 2), (1, 4), (1, 9),
119  (2, 8), (2, 10), (2, 11),
120  (3, 5), (3, 12),
121  (4, 14),
122  (5, 13),
123  (6, 11), (6, 14),
124  (8, 9),
125  (12, 14)
126  ]
127
128  net.buideNet(({net.nodes[a], net.nodes[b]} for a, b in edge_list))
129
130  # 循环更新节点状态，直至每个人都不愿改变自己的决策
131  while net.updateState():
132  net.printState()  # 打印中间状态
133  net.printState()
```