****

**运筹学与数学建模课程论文**

**班 级：数学与统计学院 数据科学与人工智能实验班**

**学生姓名 学号**

**黄河源 201800820087**

**徐潇涵 201800820149**

**日 期: 2020年12月30日**

**得 分:**

**全局最小割**

目录

[1 问题分析 1](#_Toc60230262)

[1.1 最小割问题 1](#_Toc60230263)

[问题定义 1](#_Toc60230264)

[P/NP分析与证明 1](#_Toc60230265)

[1.2 全局最小割问题 1](#_Toc60230266)

[问题定义 1](#_Toc60230267)

[P/NP分析与证明 2](#_Toc60230268)

[2 算法实现 2](#_Toc60230269)

[2.1 Karger算法 2](#_Toc60230270)

[算法原理 2](#_Toc60230271)

[实现步骤-以问题4-(1)中的数据为例 3](#_Toc60230272)

[2.2 Stoer-Wagner算法 5](#_Toc60230273)

[算法原理 5](#_Toc60230274)

[实现步骤-以问题4-(1)中的数据为例 5](#_Toc60230275)

[3 算法测试 8](#_Toc60230276)

[3.1 Karger算法 8](#_Toc60230277)

[3.2 Stoer-Wagner算法 9](#_Toc60230278)

[4 算法改进与推广 10](#_Toc60230279)

[Karger算法 10](#_Toc60230280)

[个人改进见解 10](#_Toc60230281)

[Stoer-Wagner算法 11](#_Toc60230282)

[5 总结 11](#_Toc60230283)

[6 引用 11](#_Toc60230284)

[7 附录 12](#_Toc60230285)

# 1 问题分析

对于一个只需要回答是或否的判定性问题，如果解决该问题存在一个复杂度为形式的算法，其中为该问题的实例规模，为常数，则定义该问题为Polynomial time多项式时间可解的。于是，P问题定义为所有多项式时间可解的问题的集合。

有些问题难以直接找出解决的算法，却可以在给定该问题的一个解的条件下，在多项式时间内判断这个解是否正确，这类问题称为多项式时间可验证的。NP问题定义为所有多项式时间可验证的问题的集合。多项式时间可解的问题必然在多项式时间内可验证，故，反之则暂未解决。

## 1.1 最小割问题

### 问题定义

定义一：s-t最小割问题

最小割问题指的是在给出一个连通图和两点的条件下，找出一个边权和最小的边集，使得删除该边集后不连通。

定义二：等同于全局最小割问题

在图论中，去掉其中所有边能使一张连通图分成两个独立的连通子图的边集称为图的割，一张图上最小的割称为最小割。

与最小割相关的问题称最小割问题，其变体包括带边权、有向图、包含源点与汇点，以及将原网络分为多于两个子图等问题[1]。

### P/NP分析与证明

对于定义一的s-t最小割问题，由于其与最大流问题等价，由求解最大流问题的增广路径算法(Edmonds-Karp, Dinitz)[2]复杂度形式为多项式函数可知s-t最小割问题为P问题，故也为NP问题。

对于定义二中的最小割问题，其分析与下文全局最小割问题相同。

## 1.2 全局最小割问题

### 问题定义

给定一个连通图，删去某些边使原图被划分两个独立的连通分量，在所有满足条件的边集中，边的权重之和最小的边集定义为全局最小割。全局最小割问题即是与在一个给定的连通图中，找出其全局最小割有关的问题。

* 无源汇的全局最小割问题

1.对于带有边权的无向图，可用Stoer-Wagner算法求解。

2.对于无边权的无向图，可用Karger算法求解，此时最小割等于图的边连通度[1]。

* 有源汇的全局最小割问题

3.对于带有边权的有向或无向图，该全局最小割问题被定义为切断所有边后能使源汇不连通且边权和最小的边集。（等价于1.1最小割问题中的定义一s-t最小割问题，此时的边集解不一定是图中最小的割）

4.将有源汇全局最小割问题加以推广可得到k端点最小割问题，即移除割边后形成k个连通分支的问题[1]。

### P/NP分析与证明

对于上述无源汇的全局最小割问题1，存在求解算法Stoer-Wagner算法，计算其复杂度为，由P问题定义可知，全局最小割问题为P问题，故也为NP问题。证明法二：由1.1中s-t最小割为P问题可知，Stoer-Wagner算法为运行n-1次s-t最小割即可得解，运行n-1次P问题表明S-W算法仍是P问题。对于问题2，同样存在多项式时间内可解的Karger算法，故其为P问题，也是NP问题。

对于有源汇的全局最小割问题3，如果是有向图，则根据其与最大流问题等价可知其为P问题，也为NP问题；如果是无向图，则可用最小割树求解，该数据结构以一棵带边权的树表示了所有源汇点对，可以以次最大流计算求解，故为P问题，也为NP问题。

对于推广的k割问题，其为NP难问题，无法在多项式时间内解决。

综上所述，全局最小割问题，除k割问题为NP难问题以外，均为P问题，也为NP问题。

# 2 算法实现

## 2.1 Karger算法

### 算法原理

Karger算法是一种计算连通图最小割值的随机算法。由大卫·卡格(David Karger)于1993年首次发表。该算法基于无向图中一条边收缩的概念，其基本思想是将无向图中随机选择的一条边两端的节点进行合并，直到剩下两个节点为止。

将Karger算法运行次，便能判断其最小值从而确定全局最小割。其次数基于对Karger算法正确的概率分析。一个图可能有多个最小割，计算得到某个特定最小割的概率，设最小割边的数目为，那么图中每个点的度数至少为。如果图有个节点，那么至少有条边。我们不能选特定的条边，否则就不是特定的割了。不选这 条边的概率是。每合并一次，点的数目减一。那么在整个过程中都不选那条边的概率

将算法运行多次。在每次运行中，没得到特定最小割的概率。那么运行m次没得到的概率。当时，失败的概率不到，当时，失败的概率更是不到，考虑到一般图的节点数量，可以忽略不计。

### 实现步骤-以问题4-(1)中的数据为例

**算法实现过程如下：**

已知：表示无向连通网络，V是顶点集，E是边集，n为顶点个数。

（1）随机选择边，当网络中节点数目大于2时，

合并边e，用新的节点 w 取代 u,v，

保持平行边，但删除自循环，

循环，直到网络图只有两个节点，求得s-t最小割。

（2）重复运行karger算法，保留每次最小割加权和结果，取最小加权和，

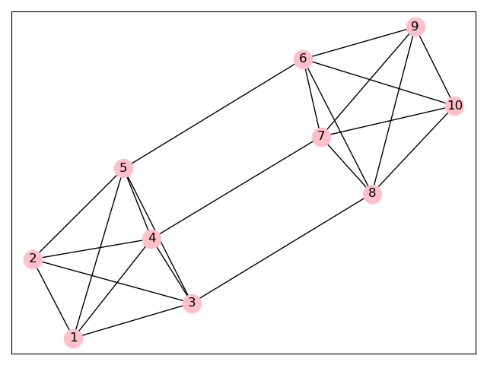
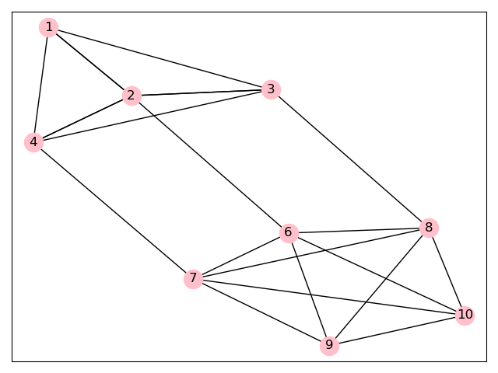
得全局最小割

Karger算法最终找到所有最小割集的时间复杂度为。

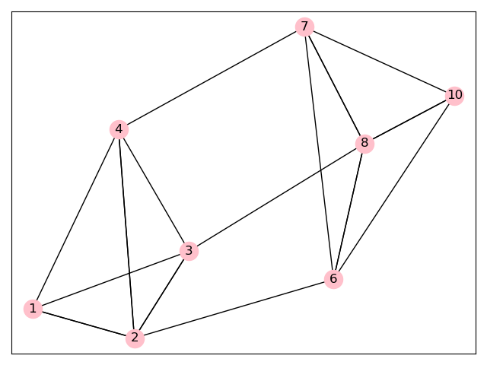
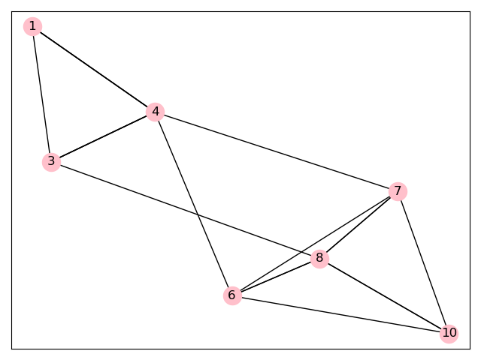
**求s-t最小割伪代码如下：**

1. **def** contract(ver, e):
2. **while** len(ver) > 2:
3. 随机选择边 e=（u,v）∈E
4. 合并边e，用新的节点 w 取代 u,v，
5. 保持平行边，但删除自循环。
6. **return** （保留边与割集的权值和）

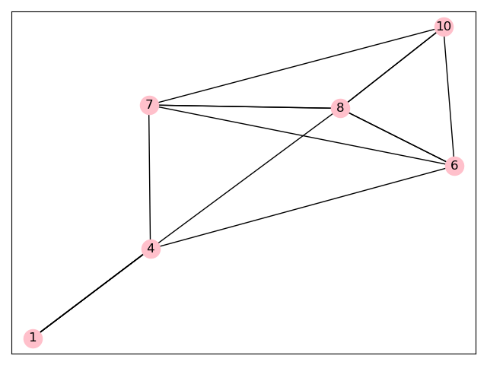
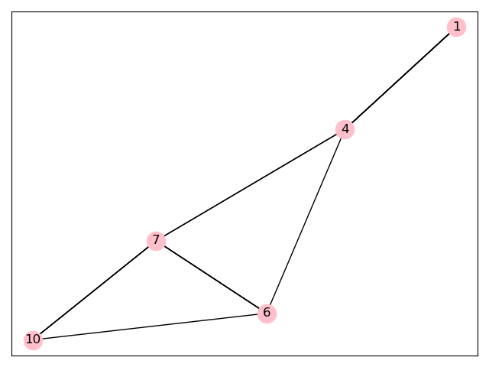
**之后将Karger算法运行次，判断其最小值来确定全局最小割。**

在Python中对问题4-(1)中的数据运行Karger算法（代码见附录karger.py），每一步可视化图像如下所示：

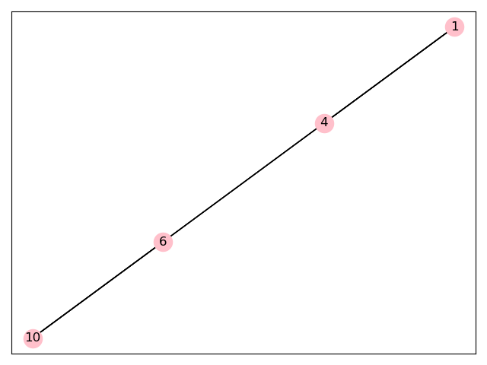
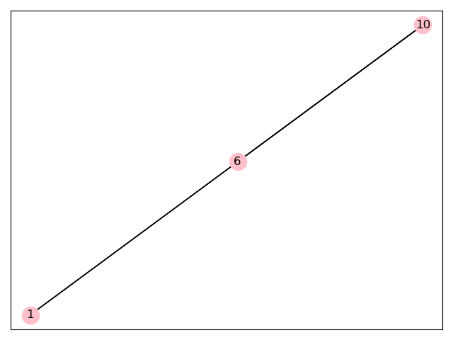
**图1.1 Karger初始图 图1.2 Karger一步合并图**



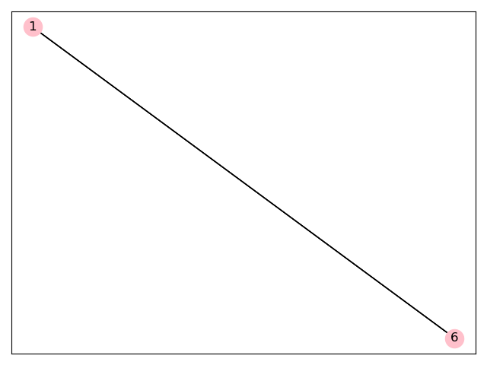
**图1.3 Karger二步合并图 图1.4 Karger三步合并图**



**图1.5 Karger四步合并图 图1.6 Karger五步合并图**



**图1.7 Karger六步合并图 图1.8 Karger八步合并图**



**图1.9 Karger最终合并图**

## 2.2 Stoer-Wagner算法

### 算法原理

给定一个正权无向图，对于图中任意两点，它们既有可能在全局最小割同侧，也可能在异侧。对于在同侧的情形，将合并成一个点并不会影响全局最小割的解；对于在异侧的情形，可证明该s-t最小割即为全局最小割。证明如下：

设全局最小割为边集，由全局最小割定义可知，对图任一割，权重之和。若两点在的异侧，则设s-t最小割为，权重之和。又因为两点在的异侧，可知也为s-t的一个割，故有权重之和。由此可得权重之和，由全局最小割定义可知为全局最小割。

由上述证明可推知，对于任意，的全局最小割必然等于原图的s-t最小割和将两点进行合并后的图的全局最小割二者中较小的一个。即。

Stoer-Wagner算法即是每次计算当前图中某两个点之间最小割，再将其合并为一个点，形成新的图并计算新的两点之间的最小割，再合并为一点，直至整个图收缩成一个点，全局最小割就是整个过程中计算过的s-t最小割中的最小值。

### 实现步骤-以问题4-(1)中的数据为例

Stoer-Wagner算法求正权无向连通图的全局最小割主要分为两部分：求s-t最小割和迭代更新全局最小割。

**求s-t最小割的步骤如下：**

1.设为一点集，初始时为空集。设为一长度为的列表，初始值均为0，存储并更新每个点的割值。

2.随机选一点加入点集。

3.令。

4.选择 中值最大的点加入集合,并对于中所有与相连的点,更新。

5.若，停止；否则，进入第4步。

6.令为倒数第二个加入的点，为最后一个加入的点，即为s-t最小割。

**迭代更新全局最小割的步骤如下：**

1.初始化全局最小割值。

2.在中求出s-t最小割。

3.迭代更新

4.合并点并得到新图。

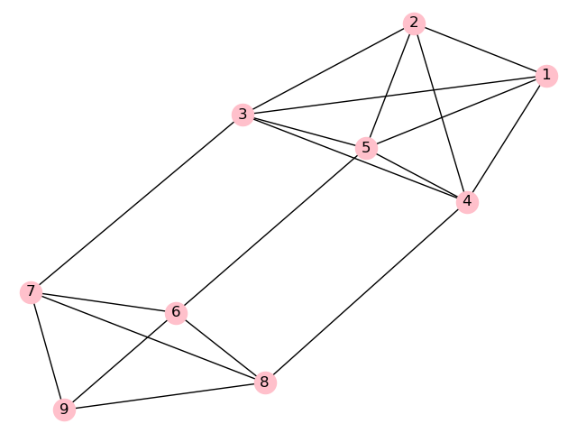
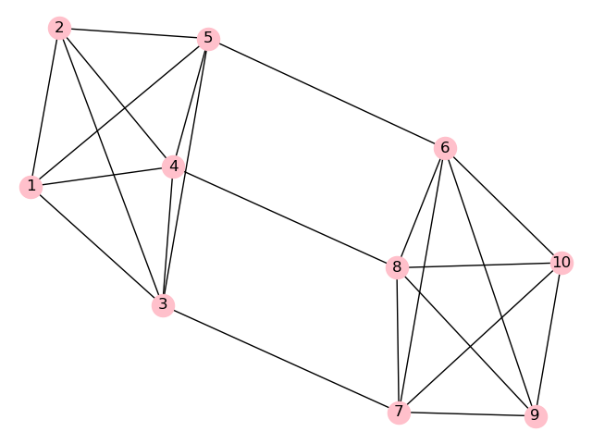
5.若则令，转入第2步；否则，输出。

第4步中，合并点为新点时，对于任意,令，若两点不相连，则设该边的权重值为0，从而得到新图。

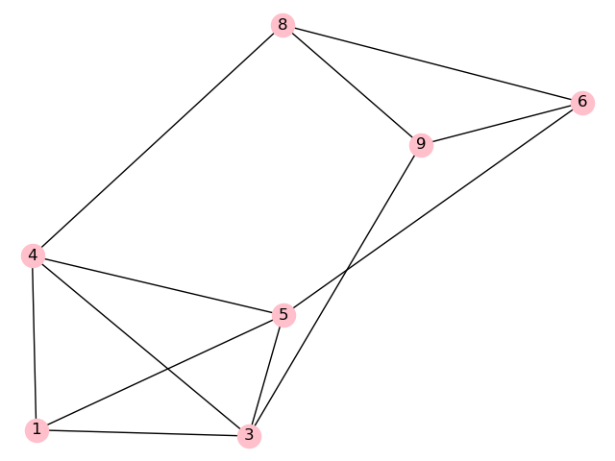
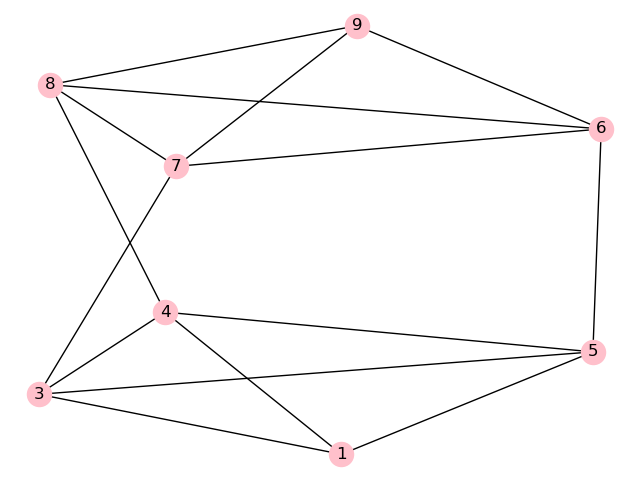
**Stoer-Wagner算法伪代码如下：**

1. **def** MinimumCutPhase(G, w, a):
2. A ← 任意一点{u}
3. wage=[0,0,0,…] 列表长度为G中点的个数
4. **while** A ≠ V:
5. 把与A联系最紧密即w(A,x)最大的点x加入A中，更新wage列表
6. cut-of-the-phase ← wage[t]   t为最后加入A的点
7. 合并最后两个加入到A的顶点s、t 并形成新图G=G’
8. **return** cut-of-the-phase
10. **def** SW-GlobalMinCut(G, w, a):
11. GlobalMinCut=+∞
12. **while** |V| > 1
13. cut-of-the-phase=MinimumCutPhase(G, w, a)
14. GlobalMinCut=min(cut-of-the-phase,GlobalMinCut)
15. **return** GlobalMinCut

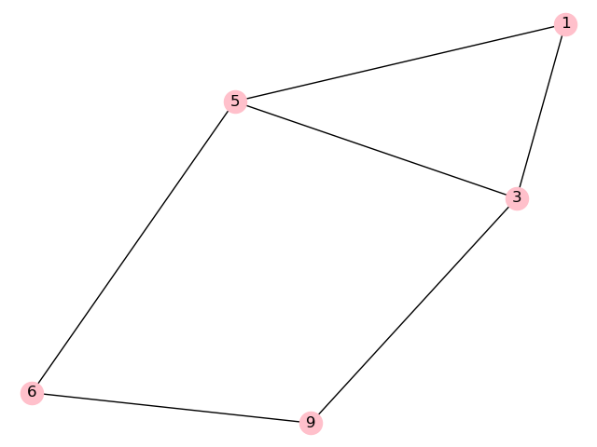
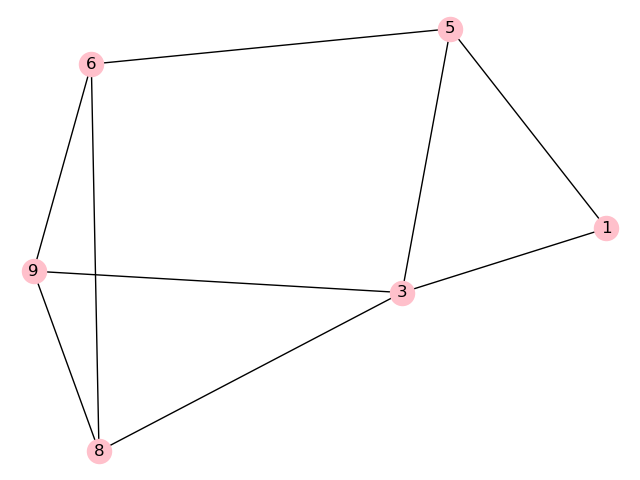
在Python中对问题4-(1)中的数据运行Stoer-Wagner算法（代码见附录SW-wenti4-1.py），每一步可视化图像如下所示：



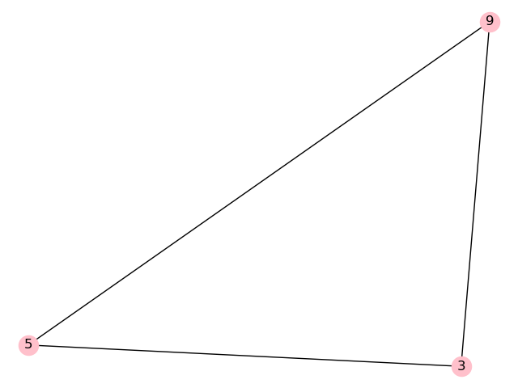
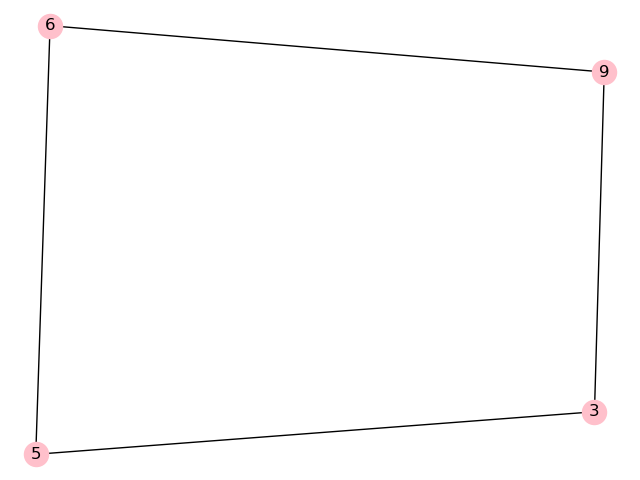
**图2.1 S-W初始图 图2.2 S-W一步收缩图**



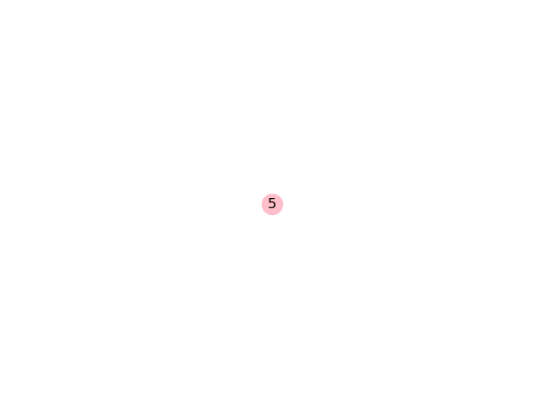
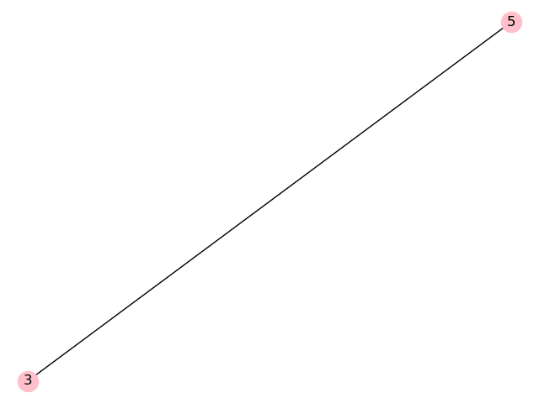
**图2.3 S-W两步收缩图 图2.4 S-W三步收缩图**



**图2.5 S-W四步收缩图 图2.6 S-W五步收缩图**



**图2.7 S-W六步收缩图 图2.8 S-W七步收缩图**



**图2.9 S-W八步收缩图 图2.10 S-W九步收缩图**

# 3 算法测试

由于全局最小割的边集不唯一，表格中仅写出一种全局最小割边集作为示例。数据中的各边权重均为1。

## 3.1 Karger算法

**表1 Karger算法测试数据表**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据来源 | 全局最小割 | 全局最小割(s,t) | 运行时间(秒) |
| 问题4-(1) | 3 | (1,6) | 0.00099 |
| BenchmarkNetwork | 2 | (65,50) | 5.37159 |
| Corruption\_Gcc | 1 | (19,201) | 319.49640 |
| Crime\_Gccsssff | 1 | (223,504) | 1277.47198 |
| PPI\_gcc | 1 | (1758,1672) | 4445.23003(n次运行) |
| RodeEU\_gcc | 1 | (130,533) | 1625.72945 |

## 3.2 Stoer-Wagner算法

对于正权连通无向图,Stoer-Wagner算法可得其全局最小割值、全局最小割所分图成的两个独立连通分量各自的点集，并可测定算法在不同数据集上的运行时间。

由于NetworkX库中有S-W算法函数，故我们额外测定了其封装函数在各数据集上的运行时间，并与自写的S-W算法代码运行时间进行对比，运行配置为英特尔i5处理器，数据记录如表2所示。

连通分量点集的获取方式为：设为包含全局最小割点的连通分量点集，则即是包含点的连通分量点集。将每次合并的两点以元组的格式记录在列表变量中，并将全局最小割出现在第几次收缩存储为变量。假设，表明在第次合并以后所得的新图的割都不会变小，而第次合并前的每次合并都有机会使当时的割值变小。由此可知第1次至次收缩均为有效收缩，收缩的点都在全局最小割同侧，第次收缩时，收缩的点在全局最小割异侧，即表明该对收缩点就是全局最小割的。故点为。进而通过列表绘制新的图，包含前条所有有效收缩的边，其中与连通的所有点即组成连通分量。

**表2 Stoer-Wagner算法测试数据表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据来源 | 全局最小割 | 连通分量点集T | 全局最小割对应的(s,t) | 运行时间(秒) | NetworkX运行时间(秒) |
| 问题4-(1) | 3 | {1,2,3,4,5} | (9,4) | 0.0 | 0.0 |
| BenchmarkNetwork | 2 | {'61', '79', '73', '67', '63', '70', '78', '72', '68', '71', '74', '80', '64', '77', '75', '62', '65', '76', '66', '69'} | (83,79) | 0.0781054 | 0.0937266 |
| Corruption\_Gcc | 1 | {'19'} | (307,19) | 2.3088777 | 1.4462323 |
| Crime\_Gcc | 1 | {'728'} | (440,728) | 35.388722 | 2.9468221 |
| PPI\_gcc | 1 | {'1290'} | (1822,1290) | 1235.2327 | 36.295177 |
| RodeEU\_gcc | 1 | {'910'} | (909,910) | 56.1527283 | 2.8260893 |

由运行时间对比可看出，自写的S-W算法在小数据集BenchmarkNetwork上运行效果较好，比NetworkX封装的函数运行速度更快，但在大数据集上的运行速率明显低于NetworkX封装好的函数。原因之一是NetworkX在求s-t最小割时加入了堆优化，使算法复杂度降低，运行更快。

# 4 算法改进与推广

## Karger算法

Karger-Stein's算法是Karger算法的扩展与改进，是由David Karger和Cliffor Stein 提出[3]，实现了数量级的改进，其具体算法过程如下：

1.对网络G，使用Karger算法直到节点数减少到，获得图与。

2.分别对与迭代使用Karger算法得到最小割集。

相比于Karger算法，该算法提高了一个数量级的运行时间，时间复杂度为。

### 个人改进见解

Karger-Stein's 算法重复选择和合并边两端的节点，直到节点数减少到，其主要思想是当图的尺寸变小时，运行Karger算法的多个独立副本。虽然提高了一个数量级的运行时间，却是Karger算法的不同使用，本质相同，时间复杂度无较大改进。

这里提出另一种改进方法，首先需要明确一个定义，我们知道在已知连通网络有最小割的情况下，其一定包含许多节点子集，当删除此节点子集时，可使源节点s与汇节点t不再连通，我们将此类节点子集定义为MCV。那么，MCV中的节点与V-MCV中节点之间的边就构成一个割集。

则可以通过**深度优先搜索（DFS）**方法来查找所有的MCV[4]。

已知条件：连通的网络，节点集V，边集E，源节点s，汇节点t。

求解：网络中的 MCV。步骤如下：

（1）。

（2）如果存在节点，且和相邻，那么是MCV，则转到步骤三，否则跳到步骤五。

（3）如果是连通网络，那么是MCV，转到步骤四执行，否则，不是MCV。

（4）令，转到步骤二。

（5）如果，结束。否则，删除中最后加入的节点，。 转到步骤二执行。

这种算法找出一个新的最小割时间复杂度仅需。并且其执行效率有较大的改进，极大地较少了查找和比较的次数，避免了大量不必要的比较和计算，能较快地找到最小割集。

## Stoer-Wagner算法

对于S-W算法，目前已有在编程技巧方面的改进，根据C++博客在测试平台POJ2914上的测试[5]，缩点合并标号的编程方式运行速度快，而不采用缩点、采用删点的编程方式思路清晰，但运行速度慢。对算法添加堆优化也可以降低算法复杂度[6]：求s-t最小割时，初始化空集合，从任选的一个起点开始不断向集合中加入新的点。这个过程类似prim算法，扩展出“最大生成树”。prim算法本身复杂度为，计算次最小割后算法复杂度为，添加堆优化后复杂度降为。

# 5 总结

全局最小割问题包含于最小割问题中。给定一个连通图，删去某些边使原图被划分两个独立的连通分量，在所有满足条件的边集中，边的权重之和最小的边集定义为全局最小割。与之相关的问题即为全局最小割问题。除k割问题为NP难问题以外，其余问题均为P问题且为NP问题。

使用Karger算法与Stoer-Wagner算法测试问题4-(1)、BenchmarkNetwork、Corruption\_Gcc、Crime\_Gcc、PPI\_gcc、RodeEU\_gcc中的数据所得最小割分别为3、2、1、1、1、1。

Karger-Stein’s算法提高了Karger算法一个数量级的运行时间，通过深度优先搜索寻找MCV确定全局最小割的方法降低了时间复杂度。添加堆优化后，Stoer-Wagner算法的复杂度由降为，且采用缩点而非删点的编程技巧可以提高算法的运行速度。

# 6 引用

1. <https://en.wikipedia.org/wiki/Minimum_cut>
2. Norman Zadeh. Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows.
3. D. R. Karger and C. Stein, “An O(n2) algorithm for minimum cuts”, in STOC., (1993)
4. 雷进军,张伯泉.查找无向图中所有最小割集的一种改进算法[J].现代计算机(专业版),2013(04):31
5. <https://blog.csdn.net/dingdi3021/article/details/101960508>

1. <https://www.cnblogs.com/zhang-qc/p/6516432.html>

# 7 附录

**（一）Karger算法代码：karger.py**

1. **import** random
2. **import** copy
3. **import** time
4. **from** math **import** \*
5. **import** networkx as nx
6. **import** matplotlib.pyplot as plt
8. **def** contract(ver, e):
9. **while** len(ver) > 2:
10. G = nx.MultiGraph()
11. G.add\_nodes\_from(ver)
12. G.add\_edges\_from(e)
13. nx.draw\_networkx(G,node\_color='pink')
14. plt.show()
16. ind = random.randrange(0, len(e))
17. [u,v] = e.pop(ind)  # pick a edge randomly
18. ver.remove(v)  # remove v from vertices
19. newEdge = []
20. **for** i **in** range(len(e)):
21. **if** e[i][0] == v:
22. e[i][0] = u
23. **elif** e[i][1] == v:
24. e[i][1] = u
25. **if** e[i][0] != e[i][1]: newEdge.append(e[i])  # remove self-loops
26. e = newEdge
27. # G = nx.MultiGraph()
28. # G.add\_nodes\_from(ver)
29. # G.add\_edges\_from(e)
30. # nx.draw\_networkx(G,node\_color='pink')
31. # plt.show()
32. **return** (len(e),e)  # return the number of the remained edges and the remained edges

35. **if** \_\_name\_\_ == '\_\_main\_\_':
36. # f = open('BenchmarkNetwork.txt')
37. # f = open('Corruption\_Gcc.txt')
38. # f = open('Crime\_Gcc.txt')
39. f = open('PPI\_gcc.txt')
40. # f = open('RodeEU\_gcc.txt')
41. # f = open('4data.txt')
43. \_f = list(f)
44. edges = []
45. vertices = []
46. **for** i **in** range(len(\_f)):
47. s = \_f[i].split()
48. vertices.append(int(s[0]))
49. vertices.append(int(s[1]))
50. vertices = list(set(vertices))
51. edges.append([int(s[0]), int(s[1])])
52. n=len(vertices)
53. n=int((n^2)\*log(n))  # 循环次数
54. **print**(n)
56. result1 = []
57. result2 = []
58. start = time.time()
59. **for** i **in** range(n):
60. v = copy.deepcopy(vertices)
61. e = copy.deepcopy(edges)
62. rl,re = contract(v, e)
63. result1.append(rl)
64. result2.append(re)
65. r=result1.index(min(result1))
66. end = time.time()
67. **print**("全局最小割值：",min(result1))
68. **print**("s,t:",result2[r])
69. **print**("运行时间:",end-start)

**（二）Stoer-Wagner算法代码**

总共有3个.py文件，第一个为测试问题4-(1)数据，第二个为测试五个txt数据，只需改动代码内的txt文件名即可读入不同数据，第三个为测试NetworkX封装函数在六种数据上的运行时间。

1. SW-wenti4-1.py (测试问题4-(1)数据)

1. **import** networkx as nx
2. **import** matplotlib.pyplot as plt
3. **import** numpy as np
4. **import** time
5. np.random.seed(100) #保持每次随机数相同
6. start\_time=time.time()
7. G = nx.Graph()
8. G.add\_edges\_from([(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(3,7),(4,5),(4,8),(5,6),(6,7),(6,8),(6,9),(6,10),(7,8),(7,9),(7,10),(8,9),(8,10),(9,10)],weight=1)

11. cut\_value=-1
12. originNodes=set(G)  #记录初始点集
13. originN=len(G)   #记录初始点的数量
14. n=len(G)
15. shousuo=[] #每一轮都收缩最后两个点 记下来每次收缩的两个点
16. globalMinCutN=-10
17. **while** n>1:
18. nodes=set(G)
19. # nx.draw(G,with\_labels=True,node\_color='pink')
20. # plt.savefig('第'+str(10-n)+'次')  #画出每次收缩两个点之后新图的图像
21. # plt.show()
22. **for** item **in** nodes:
23. G.nodes[item]['visit'] = 0
24. wage = {}
25. t = np.random.choice(G) #随机选一个起点
26. G.nodes[t]['visit']=1
27. **for** i **in** range(1,n): #i表示添加点的次数
28. p=-1
29. # print('i=',i,'t=',t,'wage=',wage)
30. **for** v, e **in** G[t].items():# v是和t相接的点，e是tv之间的权重字典
31. **if** G.nodes[v]['visit'] != 1:
32. wage[v] = wage.get(v,0)+e["weight"]
33. **if** p == -1 **or** wage[v]>wage.get(p,0):
34. p=v #找到当前wage最大的点v
35. #如果和t相接的点全都访问过了，那就寻找当前最大的wage[v]
36. **if** p==-1:
37. newWage={}
38. **for** item **in** nodes:
39. **if** G.nodes[item]['visit']!=1 **and** wage.get(item,0)>0:
40. newWage[item] =wage[item]
41. # print('newWage',newWage)
42. maxV=max(newWage,key=newWage.get)
43. p=maxV
45. # print('i=',i,'p=',p,type(p))
46. G.nodes[p]['visit']=1   #点p是当前找到的最大wage点加到A里
47. **if** i==n-1:  #表示添加了n-2次点 这次添加的是最后一个点T就是p 倒数第二个点S是t
48. **for** w, e **in** G[p].items():
49. **if** w != t:
50. **if** w **not** **in** G[t]:
51. G.add\_edge(t, w, weight=e["weight"])
52. **else**:
53. G[t][w]["weight"] += e["weight"]
54. G.remove\_node(p)
55. shousuo.append((t,p))
56. n=n-1
58. **if** cut\_value==-1:
59. cut\_value=wage[p]
60. globalMinCutN=originN-1-n
61. **else**:
62. **if** cut\_value>wage[p]:
63. cut\_value=min(cut\_value,wage[p])
64. globalMinCutN = originN - 1 - n
66. t=p
68. # nx.draw(G,with\_labels=True,node\_color='pink')
69. # plt.savefig('第'+str(10-n)+'次')  #画出最后收缩两个点之后新图的图像
70. # plt.show()
71. F=nx.Graph()
72. F.add\_edges\_from(shousuo[0:globalMinCutN])
73. Terminal=shousuo[globalMinCutN][1] #全局最小割的T点
74. Tset=set(nx.single\_source\_shortest\_path\_length(F, Terminal)) #从T点出发，根据之前两两收缩的点可以展开和T相连的边，从而还原出包含T点的连通分量
75. partition = (list(Tset), list(originNodes - Tset)) #V-T即为包含S点的连通分量
76. **print**('全局最小割所分成的两个连通分量点集为',partition)
77. end\_time=time.time()
78. time\_used=end\_time-start\_time
79. **print**('算法耗时',time\_used,'秒')

82. **print**('全局最小割值为',cut\_value)
83. **print**('全局最小割出现在第',globalMinCutN+1,'次合并点时','即此时的ST最小割就是全局最小割')
84. **print**('所有两两合并的点的列表,前者为S，后者为T：',shousuo)
85. **print**('故全局最小割为(s,t)=',shousuo[globalMinCutN],'的st最小割')

2. Stoer-Wagner-txtData.py (测试txt中的数据)

1. **import** networkx as nx
2. **import** matplotlib.pyplot as plt
3. **import** numpy as np
4. **import** time
5. np.random.seed(100) #保持每次随机数相同
6. #读取文件 构建图模型
7. G = nx.Graph()
8. edges=[]
9. with open('BenchmarkNetwork.txt','r') as f:
10. # with open('Corruption\_Gcc.txt', 'r') as f:
11. # with open('Crime\_Gcc.txt','r') as f:
12. # with open('PPI\_gcc.txt','r') as f:
13. # with open('RodeEU\_gcc.txt','r') as f:
14. **for** line **in** f.readlines():
15. edge = tuple(line.split())
16. edges.append(edge)
17. # print(edges)
18. **print**('边总数为',len(edges))
19. G.add\_edges\_from(edges,weight=1)
21. #定义SW函数
22. **def** Stoer\_W(G):
23. cut\_value = -1
24. originNodes = set(G)
25. originN = len(G)
26. n = len(G)
27. A = set()
28. shousuo = []  # 每一轮都收缩最后两个点 记下来每次收缩的两个点
29. globalMinCutN = -10
31. **while** n>1:
32. nodes=set(G)
33. # nx.draw(G,with\_labels=True)
34. # plt.show()
35. **for** item **in** nodes:
36. G.nodes[item]['visit'] = 0
37. wage = {}
38. t = np.random.choice(G) #随机选一个起点
39. G.nodes[t]['visit']=1
40. A.add(t)
41. **for** i **in** range(1,n): #i表示添加点的次数
42. p=-1
43. # print('i=',i,'t=',t,'wage=',wage,'n=',n)
44. # print('当前t相接的',G[t].items())
45. # print('A集合',A)
47. **for** v, e **in** G[t].items():# v是和t相接的点，e是tv之间的权重字典
48. # print('v=',v,'e=',e,'visit',G.nodes[v]['visit'])
49. **if** G.nodes[v]['visit'] != 1:
50. # print('当前t=',t,'相接的没访问过的',v)
51. wage[v] = wage.get(v,0)+e["weight"]
52. **if** p == -1 **or** wage[v]>wage.get(p,0):
53. p=v #找到当前wage最大的点v
54. #如果和t相接的点全都访问过了，那就寻找当前最大的wage[v]
55. **if** p==-1:
56. newWage={}
57. **for** item **in** nodes:
58. **if** G.nodes[item]['visit']!=1 **and** wage.get(item,0)>0:
59. newWage[item] =wage[item]
60. # print('当前未找过的点是',item)
61. # print('newWage',newWage)
62. maxV=max(newWage,key=newWage.get)
63. p=maxV
65. # print('i=',i,'p=',p,type(p))
66. G.nodes[p]['visit']=1   #点p是当前找到的最大wage点加到A里
67. A.add(p)
68. **if** i==n-1:  #表示添加了n-2次点 这次添加的是最后一个点T就是p 倒数第二个点S是t
69. **for** w, e **in** G[p].items():
70. **if** w != t:
71. **if** w **not** **in** G[t]:
72. G.add\_edge(t, w, weight=e["weight"])
73. **else**:
74. G[t][w]["weight"] += e["weight"]
75. G.remove\_node(p)
76. shousuo.append((t,p))
77. n=n-1
79. **if** cut\_value==-1:
80. cut\_value=wage[p]
81. globalMinCutN=originN-1-n
82. **else**:
83. **if** cut\_value>wage[p]:
84. cut\_value=min(cut\_value,wage[p])
85. globalMinCutN = originN - 1 - n
87. t=p
88. **return** cut\_value,originNodes,originN,shousuo,globalMinCutN,G
90. #开始计时并测试数据
91. start\_time=time.time()
92. cut\_value,originNodes,originN,shousuo,globalMinCutN,G\_changed=Stoer\_W(G)
94. Terminal=shousuo[globalMinCutN][1] #全局最小割的T点
95. F=nx.Graph()  #展开收缩列表中有效收缩的边，绘制新的图，获取连通分量T
96. **if** globalMinCutN==0:
97. F.add\_node(Terminal)
98. **else**:
99. F.add\_edges\_from(shousuo[0:globalMinCutN])
100. F.add\_node(Terminal)
101. # print(shousuo)
103. Tset=set(nx.single\_source\_shortest\_path\_length(F, Terminal)) #从T点出发，根据之前两两收缩的点可以展开和T相连的边，从而还原出包含T点的连通分量
104. partition = (list(Tset), list(originNodes - Tset)) #V-T即为包含S点的连通分量
105. **print**('全局最小割所分成的连通分量点集T为',partition[0])
106. end\_time=time.time()
107. time\_used=end\_time-start\_time
108. **print**('另一个连通分量点集为V-T')
109. **print**('算法用时',time\_used,'秒')
110. **print**('全局最小割值为',cut\_value)
111. **print**('全局最小割出现在第',globalMinCutN+1,'次合并点时','即此时的(s,t)=',shousuo[globalMinCutN],'这个st最小割就是全局最小割')
112. **print**('所有两两合并的点的列表,前者为S，后者为T：',shousuo)

3. networkxTest.py (测试NetworkX工具包的S-W算法在6组数据上的运行速度)

1. **import** networkx as nx
2. **import** matplotlib.pyplot as plt
3. **import** numpy as np
4. **import** time
5. np.random.seed(100) #保持每次随机数相同
6. # 问题4-(1)中十个点的图为G1
8. G1 = nx.Graph()
9. G1.add\_edges\_from([(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(3,7),(4,5),(4,8),(5,6),(6,7),(6,8),(6,9),(6,10),(7,8),(7,9),(7,10),(8,9),(8,10),(9,10)],weight=1)
10. time\_start1 = time.time()
11. ans1=nx.stoer\_wagner(G1)
12. time\_end1 = time.time()
13. time\_used1 = time\_end1 - time\_start1
14. **print**('问题4-(1)最小割数',ans1[0],'连通分量点集T',ans1[1][0])
15. **print**('G1用时',time\_used1,'秒')
17. # 问题4-(2)附件data中BenchmarkNetwork的图为G2
18. G2 = nx.Graph()
19. edges2=[]
20. with open('BenchmarkNetwork.txt','r') as f:
21. **for** line **in** f.readlines():
22. edge2 = tuple(line.split())
23. edges2.append(edge2)
24. # print(edges2)
25. **print**('G2边数',len(edges2))
26. G2.add\_edges\_from(edges2,weight=1)
28. time\_start2 = time.time()
29. ans2=nx.stoer\_wagner(G2)
30. time\_end2 = time.time()
31. time\_used2 = time\_end2 - time\_start2
32. **print**('BenchmarkNetwork最小割数',ans2[0],'连通分量点集T',ans2[1][0])
33. **print**('G2用时',time\_used2,'秒')
35. # nx.draw(G2,with\_labels=True)
36. # plt.title('BenchmarkNetwork')
37. # plt.show()
39. # 问题4-(2)附件data中Corruption\_Gcc的图为G3
41. G3 = nx.Graph()
42. edges3=[]
43. with open('Corruption\_Gcc.txt','r') as f:
44. **for** line **in** f.readlines():
45. edge3 = tuple(line.split())
46. edges3.append(edge3)
47. # print(edges3)
48. **print**('G3边数',len(edges3))
49. G3.add\_edges\_from(edges3,weight=1)
51. time\_start3 = time.time()
52. ans3=nx.stoer\_wagner(G3)
53. time\_end3 = time.time()
54. time\_used3 = time\_end3 - time\_start3
55. **print**('Corruption\_Gcc最小割数',ans3[0],'连通分量点集T',ans3[1][0])
56. **print**('G3用时',time\_used3,'秒')
58. # nx.draw(G3,with\_labels=True)
59. # plt.title('Corruption\_Gcc')
60. # plt.show()
61. # print('G3-19邻居',G3['19'])
63. # 问题4-(2)附件data中Crime\_Gcc的图为G4
65. G4 = nx.Graph()
66. edges4=[]
67. with open('Crime\_Gcc.txt','r') as f:
68. **for** line **in** f.readlines():
69. edge4 = tuple(line.split())
70. edges4.append(edge4)
71. # print(edges4)
72. **print**('G4边数',len(edges4))
73. G4.add\_edges\_from(edges4,weight=1)
75. time\_start4 = time.time()
76. ans4=nx.stoer\_wagner(G4)
77. time\_end4 = time.time()
78. time\_used4 = time\_end4 - time\_start4
79. **print**('Crime\_Gcc最小割数',ans4[0],'连通分量点集T',ans4[1][0])
80. **print**('G4用时',time\_used4,'秒')
82. # nx.draw(G4,with\_labels=True)
83. # plt.title('Crime\_Gcc')
84. # plt.show()
85. # print('G4-606邻点',G4['606'])
87. # 问题4-(2)附件data中PPI\_gcc的图为G5
89. G5 = nx.Graph()
90. edges5=[]
91. with open('PPI\_gcc.txt','r') as f:
92. **for** line **in** f.readlines():
93. edge5 = tuple(line.split())
94. edges5.append(edge5)
95. # print(edges5)
96. **print**('G5边数',len(edges5))
97. G5.add\_edges\_from(edges5,weight=1)
99. time\_start5 = time.time()
100. ans5=nx.stoer\_wagner(G5)
101. time\_end5 = time.time()
102. time\_used5 = time\_end5 - time\_start5
103. **print**('PPI\_gcc最小割数',ans5[0],'连通分量点集T',ans5[1][0])
104. **print**('G5用时',time\_used5,'秒')
106. # nx.draw(G5,with\_labels=True)
107. # plt.title('PPI\_gcc')
108. # plt.show()
109. # print('G5-1760邻居',G5['1760'])
111. # 问题4-(2)附件data中RodeEU\_gcc的图为G6
113. G6 = nx.Graph()
114. edges6=[]
115. with open('RodeEU\_gcc.txt','r') as f:
116. **for** line **in** f.readlines():
117. edge6 = tuple(line.split())
118. edges6.append(edge6)
119. # print(edges6)
120. **print**('G6边数',len(edges6))
121. G6.add\_edges\_from(edges6,weight=1)

124. time\_start6 = time.time()
125. ans6=nx.stoer\_wagner(G6)
126. time\_end6 = time.time()
127. time\_used6 = time\_end6 - time\_start6
128. **print**('RodeEU\_gcc最小割数',ans6[0],'连通分量点集T',ans6[1][0])
129. **print**('G6用时',time\_used6,'秒')
131. # nx.draw(G6,with\_labels=True)
132. # plt.title('RodeEU\_gcc')
133. # plt.show()
134. # print('G6-589邻居',G6['589'])