摘　要

图与网络作为可视化的数学工具，在实际问题中有广泛的映射，因此，图论具体问题是否具有多项式时间的算法、时间复杂度是否可以优化对实际问题的解决有至关重要的作用。

本文基于最小割问题、全局最小割问题与其在加权图、有向图、划分为两个以上集合的变体，给出了定义，并分析了各种定义下的时间复杂度。我们还使用 python 实现了两个全局最小割算法——Karger 算法和 Stoer-Wagner 算法，并用六组实际数据进行验证，得到其最小割值与最小割点。最后，基于对以上两种算法原理的理解，我们分别给出了算法改进方案，减小了时间复杂度，提升了运行速度。

关键词：最小割　全局最小割　Karger 算法　Stoer-Wagner 算法

1　引言

图区分问题是一种组合优化问题，最小割问题作为其中的一种，可以将图划分为两个或多个部分，并实现不同分区间的连接权重最小。最小割问题在数据挖掘、并行计算通信等领域中有广泛的应用。同时，最小割问题根据现实问题中的不同需求增加了不同的限制，泛生出了 k 最小割、无源汇的最小割等定义。本文第一部分中基于最小割问题，分析了不同定义下问题的时间复杂度。第二部分中，用 python 实现了 Karger 算法和 Stoer-Wagner 算法。第三部分中，将第二部分的算法应用于实例，便于读者理解。

2　最小割与全局最小割

2.1　最小割问题

2.1.1　s-t 最小割

图或网络中，给定源点与汇点，若从中删去一个边的集合后使源点与汇点间没有通路，那么此集合为图像的割。所有割中边权值和最小的即为最小割（若是无权图，默认每边权重为 1）。如图 1 中，{S–1,S–T} 为切断 S 到 T 的通路的最小割集。

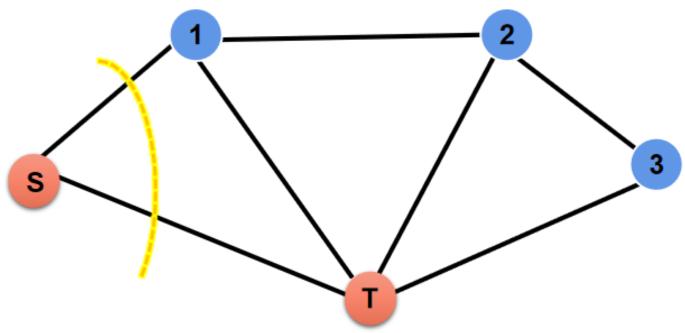


图 1: s-t 最小割实例

2.1.2　多端子切割

在无向图中，根据实际情况的复杂性，最小割问题可以泛化为多端子切割问题，即给定一多个端点的集合，计算将端点分开的最小权值和的切割。如图 2 中，{S1–S2,S1–S3, S2–S3,S3–1} 为切断端子间通路的最小割集。

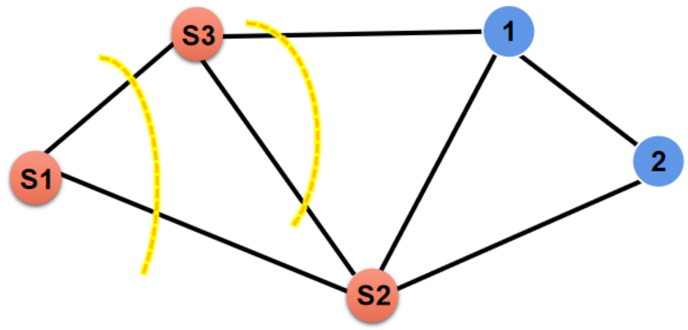


图 2: 三端子切割实例

2.1.3 时间复杂度

s-t 最小割: 根据最大流最小割定理，最大流的值等于 s-t 最小割的容量，因此 s-t 最小割问题可以转换为最大流问题。解决最大流问题有 Ford-Fulkerson 算法、Edmonds-Karp 算法、Dinic 算法等，时间复杂度分别为 $O(E|f_{max}|)$、$O(VE^2)$、$O(V^2E)$，均可以在多项式时间内解决，因此 s-t 最小割问题为 p 问题，后面会详细讲解 Ford-Fulkerson 算法。

增加约束的 s-t 最小割 s-t 最小割问题也可以通过增加约束扩大其定义范围，此时时间复杂度会发生变化。比如，负析取约束 (某一对边不能同时具有非零流量) 和正析取约束 (某一对边至少有一个具有非零流量)，负析取约束下，此问题成为 np-hard 问题，正析取约束下，允许分数流则是 p 问题，流只能为整数时可能是很强的 np-hard 问题。

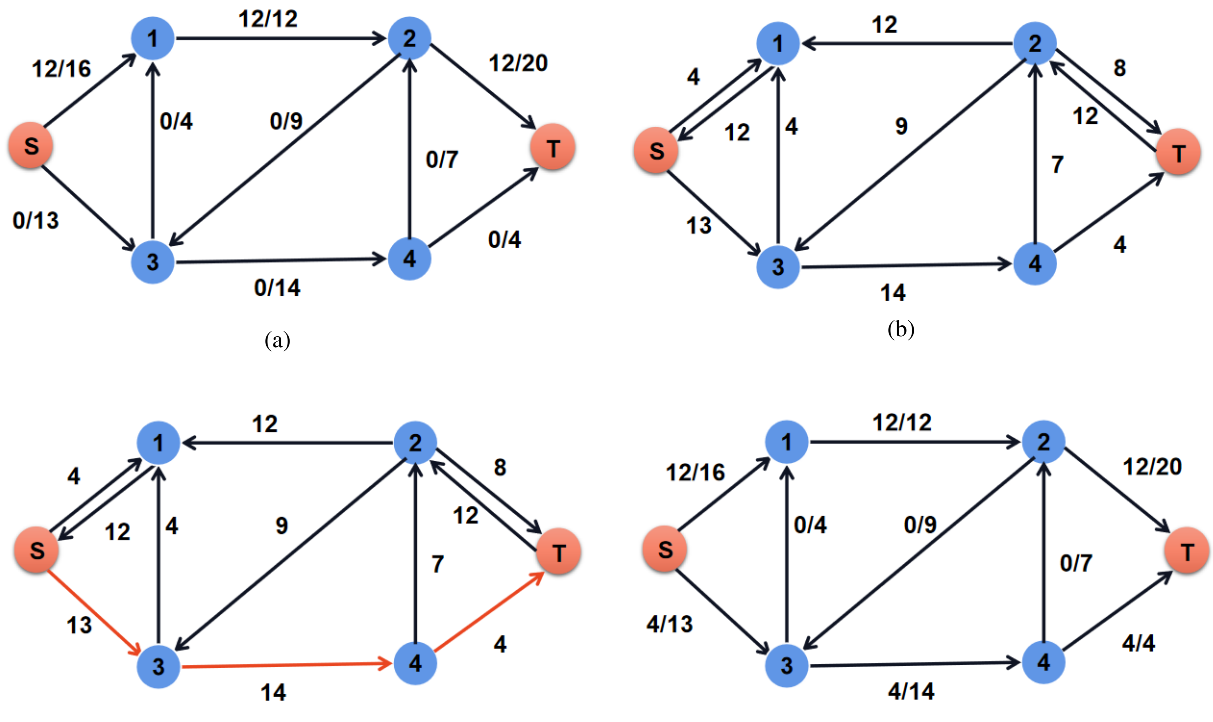
多端子切割: 当端数为 2 时，即为 s-t 最小割。当端数大于等于 3 时，由 [1] 可知，对于平面图，根据其对偶图的性质，当 k 固定时可以在多项式时间内解决。但对于任意图，或 k 不固定时，多端子切割为 np-hard 问题。

2.1.4 Ford-Fulkerson 算法

此算法是基于残差网络的贪婪算法，通过不断寻找源点到汇点之间的增广路径而增加流值，直到找到所有增广路径，得到最大流。残存容量定义如下：

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(u,v) & (v,u) \in E \\ 0 & otherwise \end{cases}$$

图 3 为算法的过程，(a) 中找到了一条由 S 到 T 的路径，(b) 为其残差网络，(c) 中的红色路径是一条增广路径，将此路径添加后得到了 (d) 图。按此步骤继续，找不到增广路径时便达到最大流。



(a) (b) (c) (d)

图 3: Ford-Fulkerson 算法

2.2 全局最小割问题

2.2.1 全局最小割

相对于 s-t 最小割问题，全局最小割的区别在于没有给出源点、汇点，而是计算将无向图分成 2 个不相交部分具有最小权值和的切割。

2.2.2 k 割

与多端子切割类似，k 割是全局最小割的泛化情况，不给定端子，计算将无向图分为 k 部分具有最小权值和的切割。

2.2.3 时间复杂度

全局最小割: 在无向无权图中，Karger 算法可以在 $O(n^2 m l n n)$ 时间内得到全局最小割，在无向加权图中，Stoer-Wagner 算法可以在 $O(mn + n^2 \log n)$ 时间内得到全局最小割，为多项式时间，因此全局最小割为 p 问题。

k 割: k=2 时，k 割为全局最小割，是 p 问题。k>=3 时, 由 [2] 可知，当 k 固定时，问题可在 $O(|V|^{k^2})$ 时间内解决，是 p 问题。但当 k 不固定时，由 [3] 可知，k 割为 np 完全问题。

其他变体: 上述所说 Stoer-Wagner 算法解决的是权重为正的无向加权图中的全局最小割问题，但当权重为负时，可以通过翻转权重的符号将其转换为加权最大割问题，因此此时全局最小割问题是 np-hard 问题。

3 全局最小割算法实现

3.1 Karger 算法

3.1.1 算法描述

1. 在图中随机取一条边，将边的两个端点合并，同时消除所有由于合并而形成自环的边

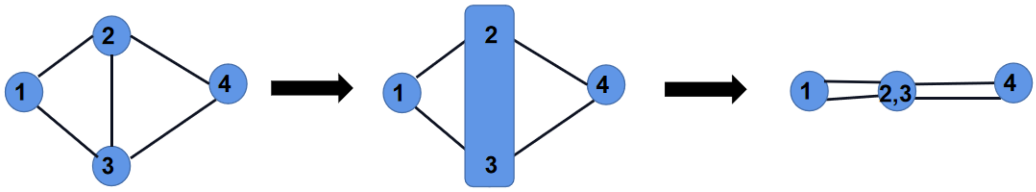


图 4: 端点合并实例

2. 重复上述步骤直到图中仅剩下两个点
3. 将最终两点之间的变作为找回的割返回

3.1.2 算法分析

这样一次运算的复杂度为 $O(m)$，我们可以看到，这样随机的过程返回的结果是不确定的，找到的割并不一定是最小的，事实上可以证明，一次运行找到最小割的概率最低为 $\frac{1}{n^2}$，那么，将上述算法重复执行 $n^2 lnn$ 次，我们可以以低于的 $\frac{1}{n}$ 的失败概率获得最小割，这就是 Karger 全局最小割算法的基本思想，时间复杂度为 $O(n^2 mlnn)$。

3.1.3 核心代码

以下为代码的核心部分，对应了算法描述中的三个步骤，详细代码于附录中给出

```
def karger(G):
    while(len(G.nodes())>2):
    ## 随机选取一条边（以两个点来表示）
    u = choice(list(G.nodes()))
    v = list(G[u])[0]
    ## 进行合并
    neighbours_v = dict(G[v])
    G.remove_node(v)
    del neighbours_v[u]
    for i in neighbours_v:
        if(G.has_edge(u,i)):
            G[u][i]['weight'] += neighbours_v[i]['weight']
```

```
        else:
            G.add_edge(u,i,weight=neighbours_v[i]['weight'])
    ## 返回最后两点，及权值
    return G[list(G.nodes())[0]][list(G.nodes())[1]]['weight'],G
```

3.1.4 测试

我们将图 5(a) 输入程序，测试 Karger 算法。可见点 1、点 6、点 2、点 7、点 8、点 3、点 9、点 5 依次被合并，最终剩下点 4 和点 10，虽然图 5(i) 中仅显示一条边，但此边的权重为 3，因此最小割的值为 3。以下为程序输出值与效果图 (边的颜色深浅代表权重大小)：

```
# 最小割，最小割点
3 ['5', '9']
```

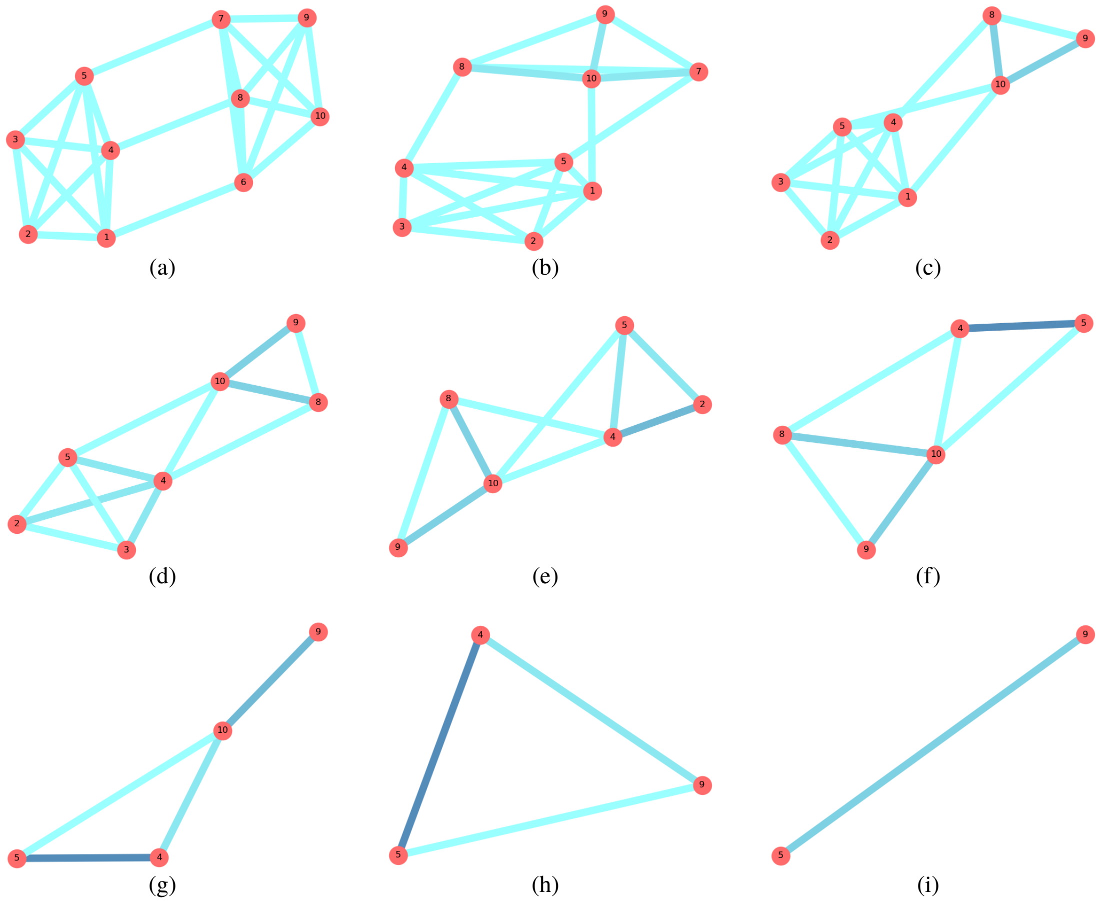


图 5: 测试结果图

3.2 Stoer-Wagner 算法

3.2.1 算法思想

算法主要基于以下事实：

两个顶点 s、t，要么在全局最小割的同一个集合中，要么在不同的集合中。那么全局最小割只可能是 st 最小割，或者是合并 s、t 后的的新图的全局最小割。

通过不断寻找 s-t 最小割、合并 s-t 最小割的过程，可以最终得到全局最小割。依据这个思想，在图 G 中任意选择 s，t 点，循环寻找最小的 s-t-cut，就可以找到图 G 的最小割。

3.2.2 算法描述

1. 初始化一张图，随机选择一个点
2. 根据权值逐步合并临近点，直至剩两点，得到一个 s,t 最小割
3. 在原图中合并 s,t 点，重复步骤 2，直至原图剩两个点
4. 取所有 s,t 最小割中的最小者即全局最小割

3.2.3 核心代码

```
def stoer_wagner(G,global_cut,u_v,s):
    if(len(G)>2):
        clo = 0
        # 求 st 最小割
        u,v,w = min_cut(deepcopy(G),s,clo)
        # 在原图中合并 st
        merge(G,u,v)
        # 比较 st 最小割与当前最小割
        if(w<global_cut):
            global_cut = w
            u_v = (u,v)
        return stoer_wagner(G,global_cut,u_v,s)
    else:
        # 剩两点时记录最后一次最小割
        last_cut = list(dict(G[s]).values())[0]['weight']
        if(last_cut<global_cut):
            global_cut = last_cut
            u_v = (s,list(G[s])[0])
        # 返回全局最小割及割点
        return global_cut,u_v
```

3.2.4 测试

我们使用 Karger 算法同样的例子测试 Stoer-Wagner 算法，程序输出值与图片如下：

```
# 最小割及最小割点
3 (5, 8)
```

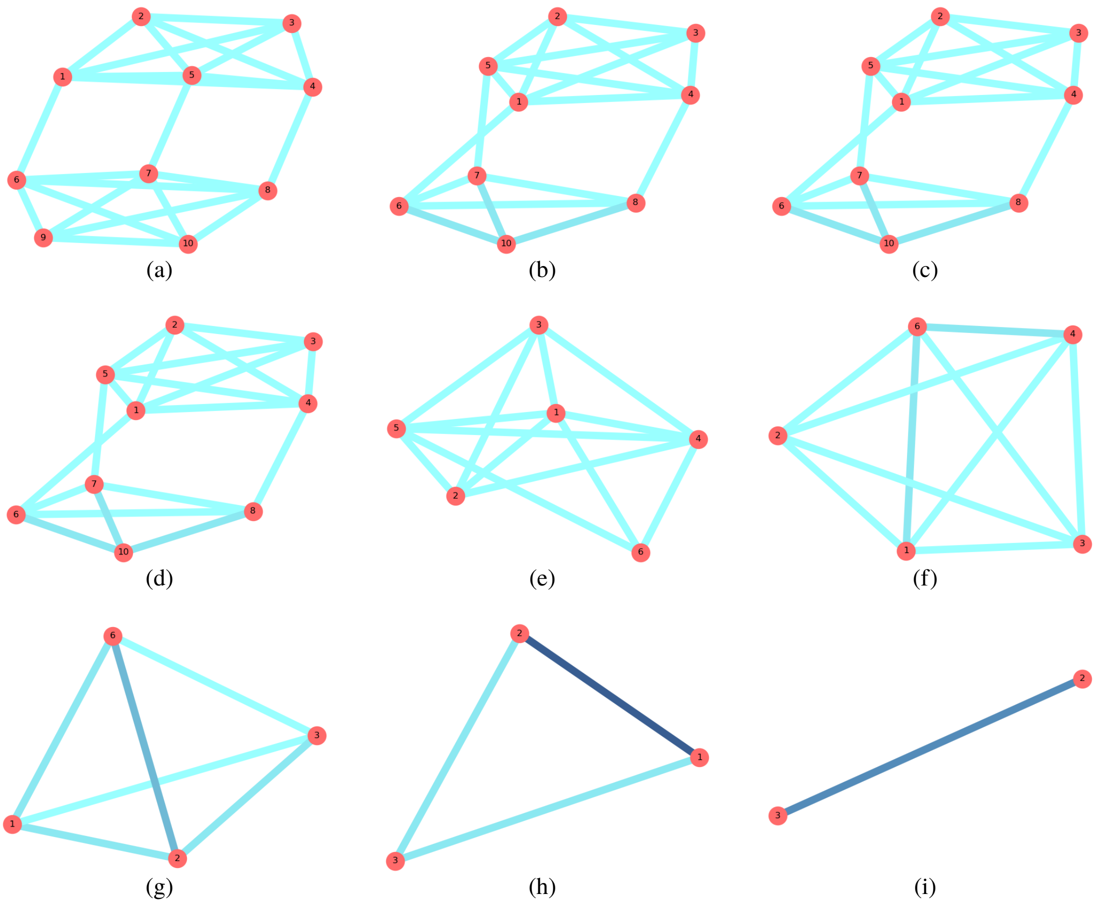
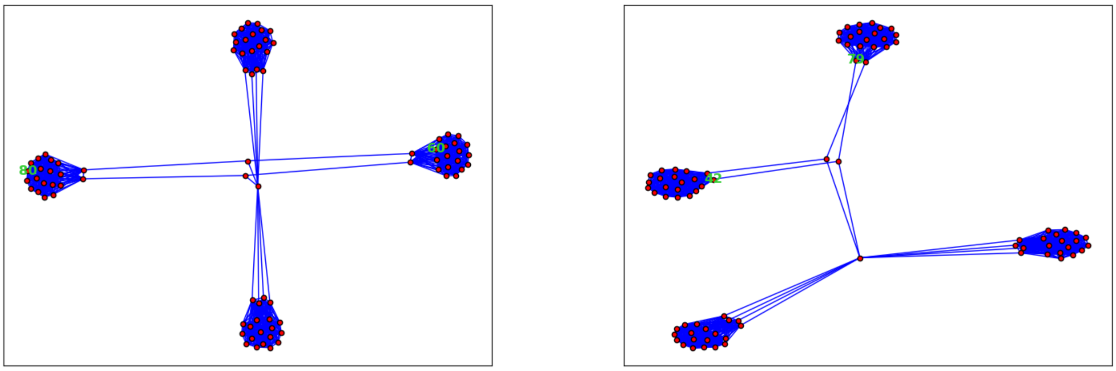


图 6: 测试结果图

4 算法实例

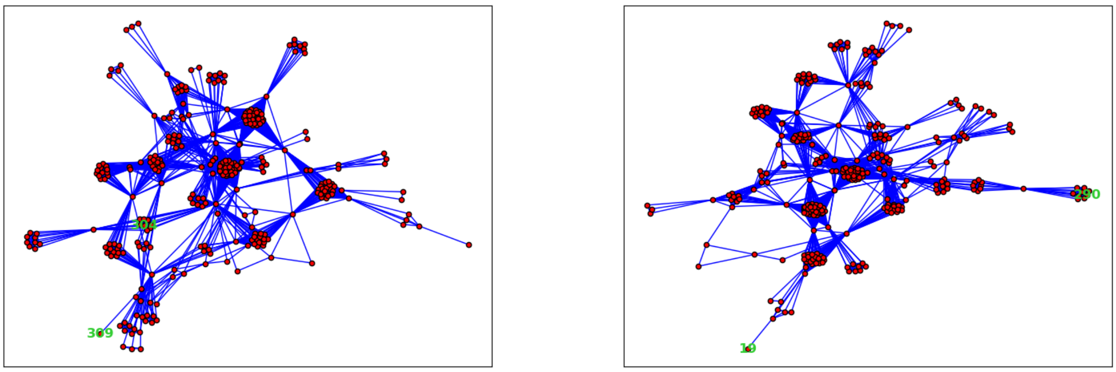
分别用 Karger 算法和 Stoer-Wagner 算法运行附件中的五组数据，得到了最小割的效果图与结果如下:



(a) Karger　　　　(b) Stoer-Wagner

图 **7:** *BenchmarkNetwork*

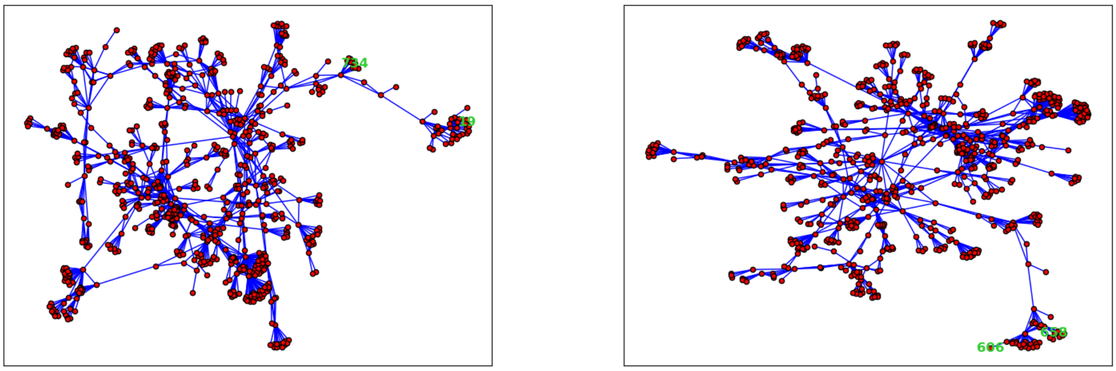
```
# 最小割 最小割点
#Karger 算法
2 ['60', '80']
#Stoer-Wagner 算法
2 (42, 79)
```



(a) Karger　　　　(b) Stoer-Wagner

图 **8:** *Corruption_Gcc*

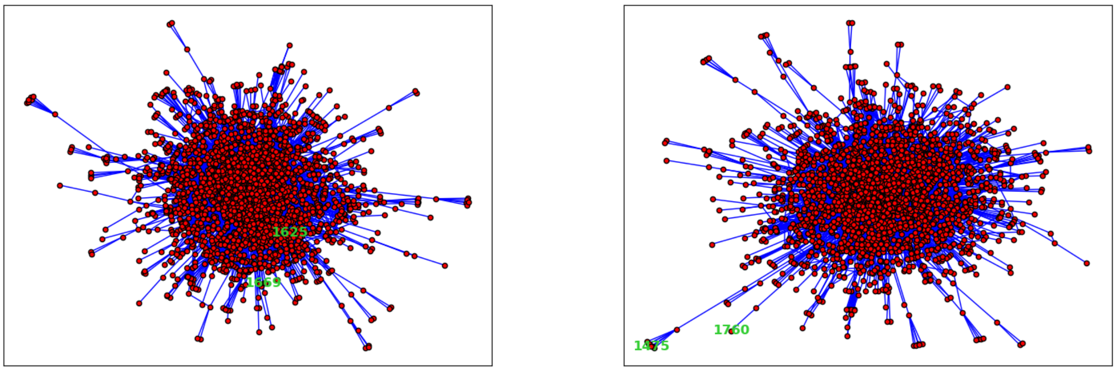
```
# 最小割 最小割点
1 ['304', '309']
1 (19, 290)
```



(a) Karger (b) Stoer-Wagner

图 9: *Crime_Gcc*

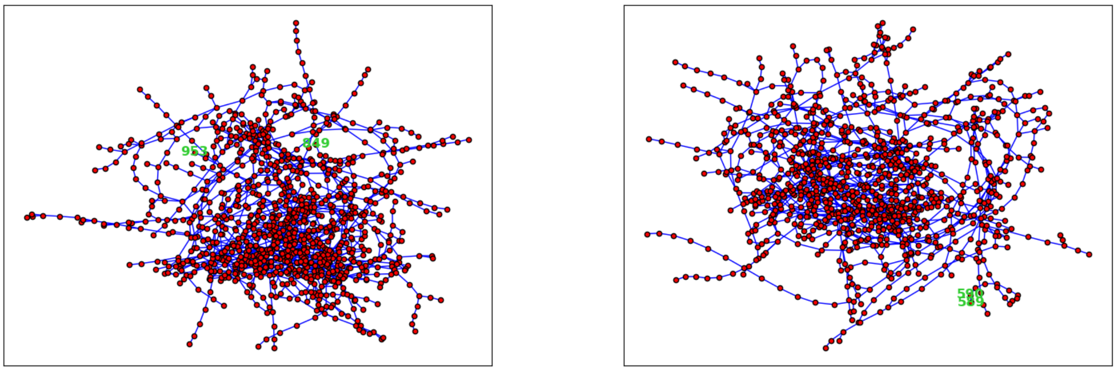
```
# 最小割 最小割点
1 ['19', '734']
1 (606, 658)
```



(a) Karger (b) Stoer-Wagner

图 10: *PPI_gcc*

```
# 最小割 最小割点
1['1625','1669']
1 (1760, 1475)
```



(a) Karger　　(b) Stoer-Wagner

图 11: *RodeEU_gcc*

```
# 最小割 最小割点
1 ['849','953']
1 (589,590)
```

5 算法改进

5.1 Karger 算法

5.1.1 算法描述

Algorithm 2: The Karger-Stein algorithm: KargerStein(G)

input : Graph G with n vertices.

1 **if** $n \geq 6$ **then**

2 　Do 2 runs and **output** the best of the 2:

3 　Use Karger's mincut algorithm to contract edges until $\frac{n}{\sqrt{2}}+1$ vertices are left, denoted as G';

4 　Recurse KargerStein(G').

图 12

因 $\frac{n}{\sqrt{2}}+1 \geq 2$，故 $n \geq 6$。相比于之前的算法，该改进使时间复杂度变为 $O(n^2log^3n)$。

5.1.2 算法分析

Karger 算法时间复杂度为 $O(n^2mlnn)$, 可以将失败概率降至 $1/n$。

Karger Stein 做 $O(log^n)$ 次算法，时间复杂度为 $O(n^2log^3n)$, 失败概率 $O(1/n)$

5.2 Stoer-Wagner 算法

3.2.2 步骤 2 中找 s,t 最小割的过程中，由最后三点合并至最后两点时得到的 s,t 最小割默认为未合并的最后两点之间的最小割，如下图：

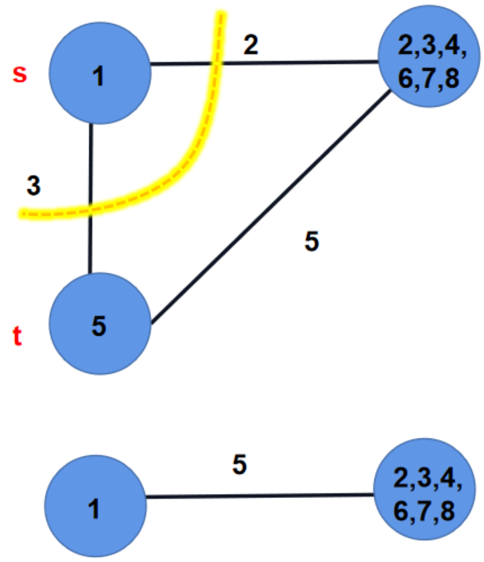


图 13

事实上，我们有如下情况，此种划分得到最小割不是 s,t 的最小割但小于 s,t 最小割，如下图：

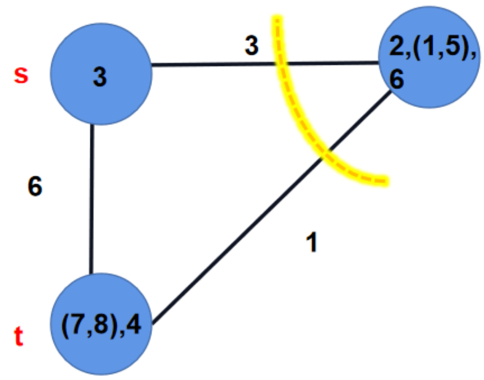


图 14

基于以上情况出发，我们做如下改进：

当合并至三点时，我们不再默认最后未合并两点为 s,t，而是在三点中求得最割法。例如上图中，2 与 s 点（最小割为 4，小于 s,t 之间的最小割 7。于是我们包留最小割 4，在原图中合并 2 与 s 点。此时我们已知 s,t 之间的最小割不可能成为全局最小割，故同时将将 s,t 点合并，最终 2,s,t 三点合并为一点。

相比上文中的算法，以上改进由每次迭代合并两点改进至合并三点，同时保证全局最小割不变，有效提升算法运行速度。

5.3 参考文献

[1] E.Dahlhaus,D.S.Johnson,C.H.Papadimitriou,P.D.Seymour,M.Yannakakis.The Complexity of Multiterminal Cuts[J].SIAM J.Comput, 1994(23).

[2]Goldschmidt, O.; Hochbaum, D. S. (1988), Proc. 29th Ann. IEEE Symp. on Foundations of Comput. Sci., IEEE Computer Society, pp. 444–451.

[3]Garey, M. R.; Johnson, D. S. (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, ISBN 978-0-7167-1044-8.

[4] Eric Vigoda.Karger’s min-cut algorithm and the Karger-Stein algorithm[R], Spring 2019.

附录：源代码

Karger 算法　karger.py

```python
from random import choice
from copy import deepcopy
import networkx as nx
import matplotlib.pyplot as plt
import time

def init_graph():
  G = nx.Graph()
  edges = []
  with open('data/Example.txt', 'r') as graphInput:
    for line in graphInput:
      edges.append(line.split())
      G.add_edges_from(edges,weight=1)
  nx.draw(G,with_labels=True)
  plt.show()
  return G

def karger(G):
  while(len(G.nodes())>2):
    u = choice(list(G.nodes()))
    v = list(G[u])[0]
    neighbours_v = dict(G[v])
    G.remove_node(v)
    del neighbours_v[u]
    for i in neighbours_v:
      if(G.has_edge(u,i)):
        G[u][i]['weight'] += neighbours_v[i]['weight']
      else:
        G.add_edge(u,i,weight=neighbours_v[i]['weight'])
  return G[list(G.nodes())[0]][list(G.nodes())[1]]['weight'],G

if __name__ == '__main__':
  G = init_graph()
```

```
  cut = [karger(deepcopy(G)) for i in range(100)]
  cut.sort(key=lambda x:x[0])
  nx.draw(cut[0][1], with_labels=True)
  plt.show()
  print("global min cut:",cut[0][0],"\nnodes:",cut[0][1].nodes())
```

Stoer-Wagner 算法　stoer_wagner.py

```
from random import choice
from copy import deepcopy
import networkx as nx
import matplotlib.pyplot as plt
import sys
sys.setrecursionlimit(5000)

def init_graph():
  G = nx.Graph()
  edges = []
  with open('data/Example.txt', 'r') as graphInput:
    for line in graphInput:
      ints = [int(x) for x in line.split()]
      edges.append(ints)
    G.add_edges_from(edges,weight=1)
  nx.draw(G,with_labels=True)
  plt.show()
  return G

def merge(G,s,t):
  neighbours = dict(G[t])
  G.remove_node(t)
  for i in neighbours:
    if(s==i):
      pass
    elif(G.has_edge(s,i)):
      G[s][i]['weight'] += neighbours[i]['weight']
    else:
```

```
            G.add_edge(s,i,weight=neighbours[i]['weight'])
    return G

def min_cut(G,s,clo):
    if(len(G)>2):
        clo = max(G[s].items(),key=lambda x:x[1]['weight'])[0]
        merge(G,s,clo)
        return min_cut(G,s,clo)
    else:
        return list(dict(G[s]).keys())[0],clo,list(dict(G[s]).values())[0]['weight']

def stoer_wagner(G,global_cut,u_v,s):
    #print("number of points:",len(G))
    if(len(G)>2):
        clo = 0
        u,v,w = min_cut(deepcopy(G),s,clo)
        merge(G,u,v)
        if(w<global_cut):
            global_cut = w
            u_v = (u,v)
        return stoer_wagner(G,global_cut,u_v,s)
    else:
        last_cut = list(dict(G[s]).values())[0]['weight']
        if(last_cut<global_cut):
            global_cut = last_cut
            u_v = (s,list(G[s])[0])
        return global_cut,u_v

if __name__ == '__main__':
    G = init_graph()
    s = choice(list(G.nodes()))
    global_cut = 99999
    u_v = ('0', '0')
    global_cut, u_v = stoer_wagner(G, global_cut, u_v, s)
    print("global min cut",global_cut,"\nnodes:", u_v)
```