山东大学（威海）
SHANDONG UNIVERSITY , WEIHAI

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

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$O(n^k)$形式的算法，其中n为该问题的实例规模，k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$P \subset NP$，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$G = (V, E)$和s, t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s, t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$O(mn^2)$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$G = (V, E)$，删去某些边E'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E'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 s,t 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 s,t 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 s,t 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$O(|V||E|+|V|^2\log|V|)$，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 s,t 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$|V|-1$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 $G=(V,E)$中一条边 (u,v)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 $n^2 \ln n$ 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 c，那么图中每个点的度数至少为 c。如果图有 n 个节点，那么至少有$\mathrm{n}*\frac{\mathrm{c}}{2}$条边。我们不能选特定的 c 条边，否则就不是特定的割了。不选这 c 条边的概率是$\left(1-\frac{\mathrm{c}}{\text{边数}}\right) \geq \left(1-\frac{2}{\mathrm{n}}\right)$。每合并一次，点的数目减一。那么在整个过程中都不选那 c 条边的概率 $\geq \left(1-\frac{2}{n}\right)*\left(1-\frac{2}{n-1}\right)*\ldots\ldots*\left(1-\frac{2}{3}\right)=\frac{2}{\mathrm{n}*(\mathrm{n}-1)} \geq \frac{1}{n^2}$

将算法运行多次。在每次运行中，没得到特定最小割的概率$\leq 1-\frac{1}{\mathrm{n2}}$。那么运行 m 次没得到的概率$\leq \left(1-\frac{1}{n^2}\right)^m$。当 $m=n$ 时，失败的概率不到 $1/e$，当$\mathrm{m}=n^2 \ln(\mathrm{n})$时，失败的概率更是不到 $1/n$，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：$G=(V,E)$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$e=(u,v) \in E$，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$O(n^2\ log^3 n)$。

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 $n^2 \ln n$ 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：

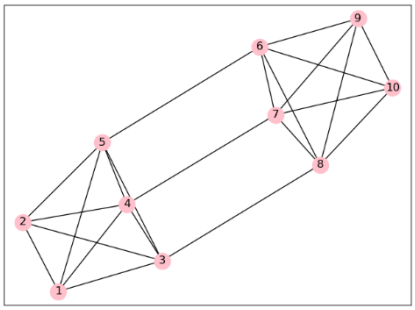

图 1.1 Karger 初始图

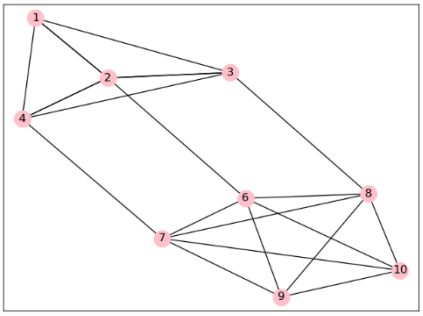

图 1.2 Karger 一步合并图

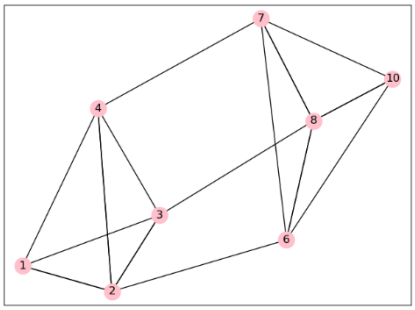

图 1.3 Karger 二步合并图

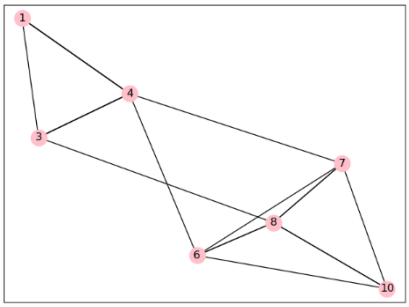

图 1.4 Karger 三步合并图

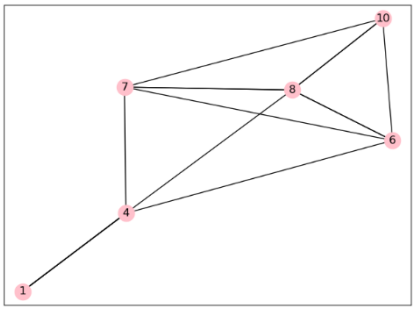

图 1.5 Karger 四步合并图

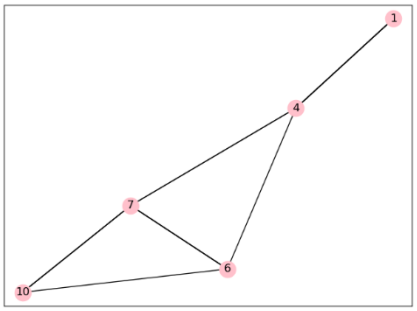

图 1.6 Karger 五步合并图

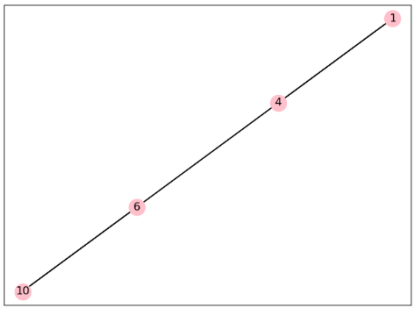

图 1.7 Karger 六步合并图

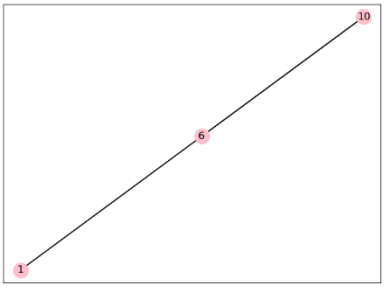

图 1.8 Karger 八步合并图

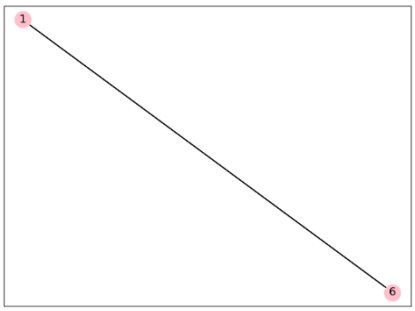

图 1.9 Karger 最终合并图

2.2 Stoer-Wagner 算法

算法原理

给定一个正权无向图 $G=(V,E)$，对于图中任意两点 s,t，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 s,t 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 C，由全局最小割定义可知，对图 G 任一割 S，权重之和 $C \leq S$。若 s,t 两点在 C 的异侧，则设 s-t 最小割为 D，权重之和 $C \leq D$。又因为 s,t 两点在 C 的异侧，可知 C 也为 s-t 的一个割，故有权重之和 $D \leq C$。由此可得权重之和 $C=D$，由全局最小割定义可知 D 为全局最小割。

由上述证明可推知，对于任意 $s,t \in V$，G 的全局最小割 C 必然等于原图的 s-t 最小割 D 和将 s,t 两点进行合并后的图 G'的全局最小割 C'二者中较小的一个。即 $C=\min\{D,C'\}$。

Stoer-Wagner 算法即是每次计算当前图中某两个 s,t 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 s',t' 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 $G=(V,E)$ 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 A 为一点集，初始时为空集。设 $wage$ 为一长度为 $|V|$ 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 $u \in V$ 加入点集 A。
3. 令 $w(A,x)=\sum weight(v[i],x),\ v[i] \in A$。

4. 选择 $V-A$ 中 $w(A,x)$ 值最大的点 x 加入集合 A, 并对于 $V-A$ 中所有与 x 相连的点 p , 更新 $wage[p]+=weight(x,p)$。

5. 若 $A=V$，停止；否则，进入第 4 步。

6. 令 s 为倒数第二个加入 A 的点， t 为最后一个加入 A 的点， $wage[t]$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 $GlobaMinCut=+\infty$。
2. 在 $G=(V,E)$ 中求出 s-t 最小割 D。
3. 迭代更新 $GlobaMinCut=\min\{D,GlobalMinCut\}$
4. 合并点 s,t 并得到新图 $G'=(V',E')$。
5. 若 $|V'|>1$ 则令 $G=G'$，转入第 2 步；否则，输出 $GlobalMinCut$。

第 4 步中，合并点 s,t 为新点 c 时，对于任意 $v\in V$, 令 $weight(v,c)=weight(a,v)+weight(b,v)=weight(c,v)$，若两点不相连，则设该边的权重值为 0，从而得到新图 $G'=(V-s-t+c=V',E')$。

Stoer-Wagner 算法伪代码如下：

```
def MinimumCutPhase(G, w, a):
    A ← 任意一点{u}
    wage=[0,0,0,…]  列表长度为 G 中点的个数
    while A ≠ V:
        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
    cut-of-the-phase ← wage[t]   t 为最后加入 A 的点
    合并最后两个加入到 A 的顶点 s、t 并形成新图 G=G'
    return cut-of-the-phase

def SW-GlobalMinCut(G, w, a):
    GlobalMinCut=+∞
    while |V| > 1
        cut-of-the-phase=MinimumCutPhase(G, w, a)
        GlobalMinCut=min(cut-of-the-phase,GlobalMinCut)
    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：

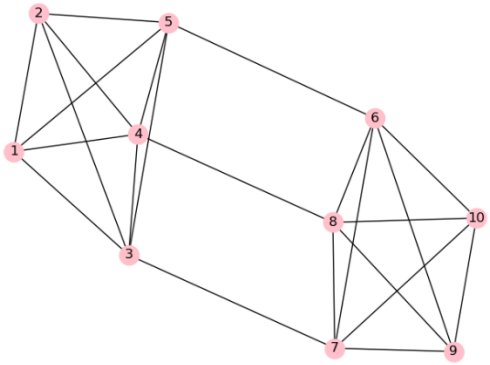

图 2.1 S-W 初始图

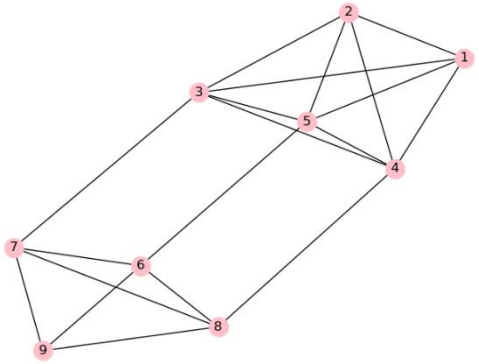

图 2.2 S-W 一步收缩图

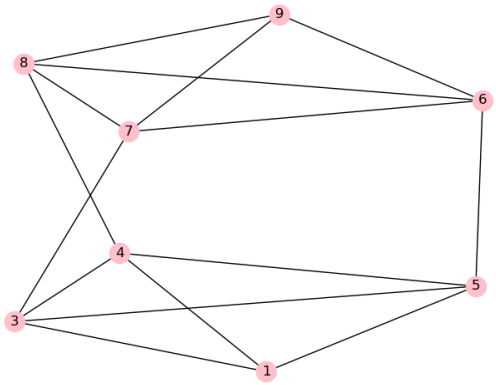

图 2.3 S-W 两步收缩图

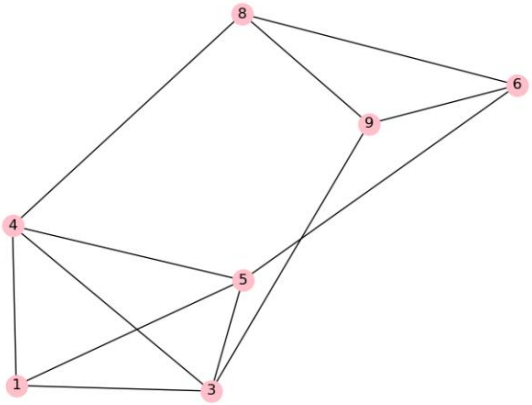

图 2.4 S-W 三步收缩图

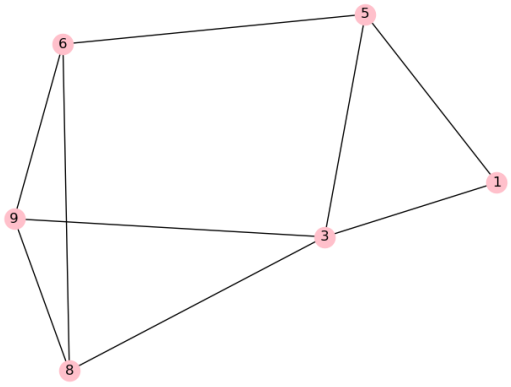

图 2.5 S-W 四步收缩图

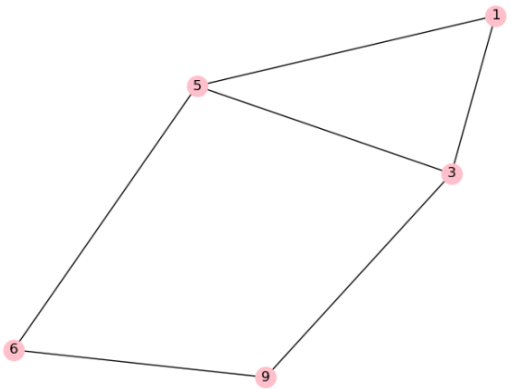

图 2.6 S-W 五步收缩图

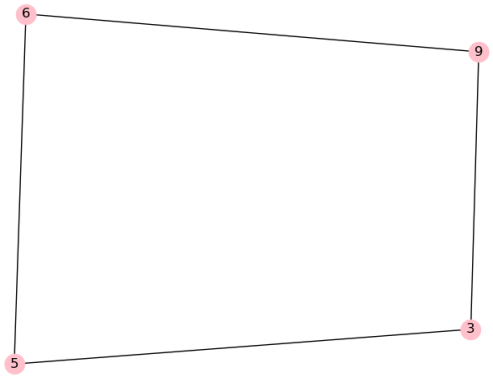

图 2.7 S-W 六步收缩图

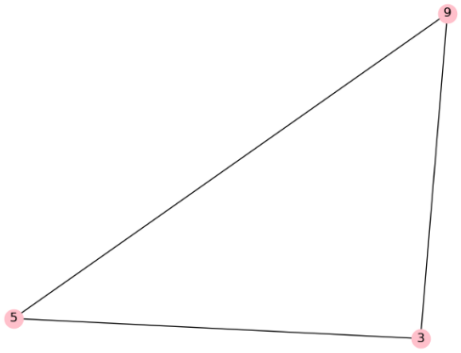

图 2.8 S-W 七步收缩图

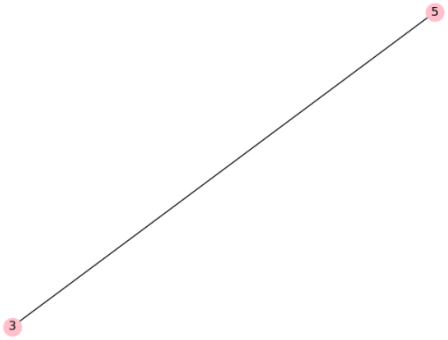

图 2.9 S-W 八步收缩图

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 $G=(V,E)$, Stoer-Wagner 算法可得其全局最小割值、全局最小割所分图 G 成的两个独立连通分量各自的点集 $T, V-T$，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 T 为包含全局最小割 t 点的连通分量点集，则 $V-T$ 即是包含 s 点的连通分量点集。将每次合并的两点 a,b 以元组 (a,b)的格式记录在列表变量 $shousuo$ 中，并将全局最小割出现在第几次收缩存储为变量 $globalMinCutN$。假设$globalMinCutN=i$，表明在第 i 次合并以后所得的新图的割都不会变小，而第 i 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 $i-1$次收缩均为有效收缩，收缩的点都在全局最小割同侧，第 i 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 s,t。故 t 点为 $shousuo[i-1][1]$。进而通过列表 $shousuo[0:i-1]$ 绘制新的图，包含前$i-1$条所有有效收缩的边，其中与 t 连通的所有点即组成连通分量 T。

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$n/\sqrt{2}$，获得图G_1与G_2。

2. 分别对G_1与G_2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$O(n^2 ln^{O(1)} n)$。

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$n/\sqrt{2}$，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$G = (V, E)$，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）$i = k = 0$，$S = U = \{s\}$，$T = V - \{s\}$，$N = \{t\}$，$P = \{s\}$。

（2）如果存在节点$u \in T - N_i$，且u和S相邻，那么$S \cup \{u\}$是 MCV，则转到步骤三，否则跳到步骤五。

（3）如果$G\big(T - \{u\}，E(T - \{u\})\big)$是连通网络，那么$S \cup \{u\}$是 MCV，转到步骤四执行，否则，$S \cup \{u\}$不是 MCV。

（4）令$i = i + 1; k = k + 1; S = S \cup \{u\}; U_k = S; P = P \cup \{U_k\}; T = T - \{u\}; N_i = N_i - 1$，转到步骤二。

（5）如果$i = 1$，结束。否则，删除S中最后加入的节点v，$i = i - 1; N_i = N_i\{v\}$，$T = T\{v\}$。 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 $O(|V|)$。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 A，从任选的一个起点 u 开始不断向集合 A 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$O(n^2)$，计算 $n-1$次最小割后算法复杂度为$O(n^3)$，添加堆优化后复杂度降为$O(n^2 \log n)$。

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$G=(V,E)$，删去某些边 E'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 E'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 $O(n^3)$降为 $O(n^2 \log n)$，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

[6] https://www.cnblogs.com/zhang-qc/p/6516432.html

7 附录

（一）Karger 算法代码：karger.py

```
import random
import copy
import time
from math import *
import networkx as nx
import matplotlib.pyplot as plt

def contract(ver, e):
    while len(ver) > 2:
        G = nx.MultiGraph()
        G.add_nodes_from(ver)
        G.add_edges_from(e)
        nx.draw_networkx(G,node_color='pink')
        plt.show()

        ind = random.randrange(0, len(e))
        [u,v] = e.pop(ind)  # pick a edge randomly
        ver.remove(v)  # remove v from vertices
        newEdge = []
        for i in range(len(e)):
            if e[i][0] == v:
                e[i][0] = u
            elif e[i][1] == v:
                e[i][1] = u
            if e[i][0] != e[i][1]: newEdge.append(e[i])  # remove self-loops
        e = newEdge
    # G = nx.MultiGraph()
    # G.add_nodes_from(ver)
    # G.add_edges_from(e)
    # nx.draw_networkx(G,node_color='pink')
    # plt.show()
    return (len(e),e)  # return the number of the remained edges and the remained edges


if __name__ == '__main__':
    # f = open('BenchmarkNetwork.txt')
    # f = open('Corruption_Gcc.txt')
```

```
    # f = open('Crime_Gcc.txt')
    f = open('PPI_gcc.txt')
    # f = open('RodeEU_gcc.txt')
    # f = open('4data.txt')

    _f = list(f)
    edges = []
    vertices = []
    for i in range(len(_f)):
        s = _f[i].split()
        vertices.append(int(s[0]))
        vertices.append(int(s[1]))
        vertices = list(set(vertices))
        edges.append([int(s[0]), int(s[1])])
    n=len(vertices)
    n=int((n^2)*log(n))  # 循环次数
    print(n)

    result1 = []
    result2 = []
    start = time.time()
    for i in range(n):
        v = copy.deepcopy(vertices)
        e = copy.deepcopy(edges)
        rl,re = contract(v, e)
        result1.append(rl)
        result2.append(re)
    r=result1.index(min(result1))
    end = time.time()
    print("全局最小割值：",min(result1))
    print("s,t:",result2[r])
    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)

```
import networkx as nx
```

```
import matplotlib.pyplot as plt
import numpy as np
import time
np.random.seed(100) #保持每次随机数相同
start_time=time.time()
G = nx.Graph()
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)

cut_value=-1
originNodes=set(G)  #记录初始点集
originN=len(G)   #记录初始点的数量
n=len(G)
shousuo=[] #每一轮都收缩最后两个点 记下来每次收缩的两个点
globalMinCutN=-10
while n>1:
    nodes=set(G)
    # nx.draw(G,with_labels=True,node_color='pink')
    # plt.savefig('第'+str(10-n)+'次')  #画出每次收缩两个点之后新图的图像
    # plt.show()
    for item in nodes:
        G.nodes[item]['visit'] = 0
    wage = {}
    t = np.random.choice(G) #随机选一个起点
    G.nodes[t]['visit']=1
    for i in range(1,n): #i 表示添加点的次数
        p=-1
        # print('i=',i,'t=',t,'wage=',wage)
        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
            if G.nodes[v]['visit'] != 1:
                wage[v] = wage.get(v,0)+e["weight"]
                if p == -1 or wage[v]>wage.get(p,0):
                    p=v #找到当前 wage 最大的点 v
        #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]
        if p==-1:
            newWage={}
            for item in nodes:
                if G.nodes[item]['visit']!=1 and wage.get(item,0)>0:
                    newWage[item] =wage[item]
            # print('newWage',newWage)
            maxV=max(newWage,key=newWage.get)
            p=maxV
```

```
44.
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
57.
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
65.
66.         t=p
67.
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')
80.
81.
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')
```

```
print('所有两两合并的点的列表,前者为 S，后者为 T： ',shousuo)
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)

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import time
np.random.seed(100) #保持每次随机数相同
#读取文件 构建图模型
G = nx.Graph()
edges=[]
with open('BenchmarkNetwork.txt','r') as f:
# with open('Corruption_Gcc.txt', 'r') as f:
# with open('Crime_Gcc.txt','r') as f:
# with open('PPI_gcc.txt','r') as f:
# with open('RodeEU_gcc.txt','r') as f:
    for line in f.readlines():
        edge = tuple(line.split())
        edges.append(edge)
# print(edges)
print('边总数为',len(edges))
G.add_edges_from(edges,weight=1)

#定义 SW 函数
def Stoer_W(G):
    cut_value = -1
    originNodes = set(G)
    originN = len(G)
    n = len(G)
    A = set()
    shousuo = []  # 每一轮都收缩最后两个点 记下来每次收缩的两个点
    globalMinCutN = -10

    while n>1:
        nodes=set(G)
        # nx.draw(G,with_labels=True)
        # plt.show()
        for item in nodes:
            G.nodes[item]['visit'] = 0
        wage = {}
        t = np.random.choice(G) #随机选一个起点
        G.nodes[t]['visit']=1
```

```
        A.add(t)
        for i in range(1,n): #i 表示添加点的次数
            p=-1
            # print('i=',i,'t=',t,'wage=',wage,'n=',n)
            # print('当前 t 相接的',G[t].items())
            # print('A 集合',A)

            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
                # print('v=',v,'e=',e,'visit',G.nodes[v]['visit'])
                if G.nodes[v]['visit'] != 1:
                    # print('当前 t=',t,'相接的没访问过的',v)
                    wage[v] = wage.get(v,0)+e["weight"]
                    if p == -1 or wage[v]>wage.get(p,0):
                        p=v #找到当前 wage 最大的点 v
                #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]
            if p==-1:
                newWage={}
                for item in nodes:
                    if G.nodes[item]['visit']!=1 and wage.get(item,0)>0:
                        newWage[item] =wage[item]
                        # print('当前未找过的点是',item)
                # print('newWage',newWage)
                maxV=max(newWage,key=newWage.get)
                p=maxV

            # print('i=',i,'p=',p,type(p))
            G.nodes[p]['visit']=1   #点 p 是当前找到的最大 wage 点加到 A 里
            A.add(p)
            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
                for w, e in G[p].items():
                    if w != t:
                        if w not in G[t]:
                            G.add_edge(t, w, weight=e["weight"])
                        else:
                            G[t][w]["weight"] += e["weight"]
                G.remove_node(p)
                shousuo.append((t,p))
                n=n-1

                if cut_value==-1:
                    cut_value=wage[p]
                    globalMinCutN=originN-1-n
                else:
```

```
                    if cut_value>wage[p]:
                        cut_value=min(cut_value,wage[p])
                        globalMinCutN = originN - 1 - n

            t=p
    return cut_value,originNodes,originN,shousuo,globalMinCutN,G

#开始计时并测试数据
start_time=time.time()
cut_value,originNodes,originN,shousuo,globalMinCutN,G_changed=Stoer_W(G)

Terminal=shousuo[globalMinCutN][1] #全局最小割的 T 点
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
if globalMinCutN==0:
    F.add_node(Terminal)
else:
    F.add_edges_from(shousuo[0:globalMinCutN])
    F.add_node(Terminal)

# print(shousuo)

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
partition = (list(Tset), list(originNodes - Tset)) #V-T 即为包含 S 点的连通分量
print('全局最小割所分成的连通分量点集 T 为',partition[0])
end_time=time.time()
time_used=end_time-start_time
print('另一个连通分量点集为 V-T')
print('算法用时',time_used,'秒')
print('全局最小割值为',cut_value)
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')
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)

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import time
np.random.seed(100) #保持每次随机数相同
# 问题 4-(1)中十个点的图为 G1

```

```
G1 = nx.Graph()
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)
time_start1 = time.time()
ans1=nx.stoer_wagner(G1)
time_end1 = time.time()
time_used1 = time_end1 - time_start1
print('问题 4-(1)最小割数',ans1[0],'连通分量点集 T',ans1[1][0])
print('G1 用时',time_used1,'秒')

# 问题 4-(2)附件 data 中 BenchmarkNetwork 的图为 G2
G2 = nx.Graph()
edges2=[]
with open('BenchmarkNetwork.txt','r') as f:
    for line in f.readlines():
        edge2 = tuple(line.split())
        edges2.append(edge2)
# print(edges2)
print('G2 边数',len(edges2))
G2.add_edges_from(edges2,weight=1)

time_start2 = time.time()
ans2=nx.stoer_wagner(G2)
time_end2 = time.time()
time_used2 = time_end2 - time_start2
print('BenchmarkNetwork 最小割数',ans2[0],'连通分量点集 T',ans2[1][0])
print('G2 用时',time_used2,'秒')

# nx.draw(G2,with_labels=True)
# plt.title('BenchmarkNetwork')
# plt.show()

# 问题 4-(2)附件 data 中 Corruption_Gcc 的图为 G3

G3 = nx.Graph()
edges3=[]
with open('Corruption_Gcc.txt','r') as f:
    for line in f.readlines():
        edge3 = tuple(line.split())
        edges3.append(edge3)
# print(edges3)
print('G3 边数',len(edges3))
G3.add_edges_from(edges3,weight=1)
```

```
50.
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')
57.
58. # nx.draw(G3,with_labels=True)
59. # plt.title('Corruption_Gcc')
60. # plt.show()
61. # print('G3-19 邻居',G3['19'])
62.
63. # 问题 4-(2)附件 data 中 Crime_Gcc 的图为 G4
64.
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)
74.
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')
81.
82. # nx.draw(G4,with_labels=True)
83. # plt.title('Crime_Gcc')
84. # plt.show()
85. # print('G4-606 邻点',G4['606'])
86.
87. # 问题 4-(2)附件 data 中 PPI_gcc 的图为 G5
88.
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())
```

```
        edges5.append(edge5)
# print(edges5)
print('G5 边数',len(edges5))
G5.add_edges_from(edges5,weight=1)

time_start5 = time.time()
ans5=nx.stoer_wagner(G5)
time_end5 = time.time()
time_used5 = time_end5 - time_start5
print('PPI_gcc 最小割数',ans5[0],'连通分量点集 T',ans5[1][0])
print('G5 用时',time_used5,'秒')

# nx.draw(G5,with_labels=True)
# plt.title('PPI_gcc')
# plt.show()
# print('G5-1760 邻居',G5['1760'])

# 问题 4-(2)附件 data 中 RodeEU_gcc 的图为 G6

G6 = nx.Graph()
edges6=[]
with open('RodeEU_gcc.txt','r') as f:
    for line in f.readlines():
        edge6 = tuple(line.split())
        edges6.append(edge6)
# print(edges6)
print('G6 边数',len(edges6))
G6.add_edges_from(edges6,weight=1)


time_start6 = time.time()
ans6=nx.stoer_wagner(G6)
time_end6 = time.time()
time_used6 = time_end6 - time_start6
print('RodeEU_gcc 最小割数',ans6[0],'连通分量点集 T',ans6[1][0])
print('G6 用时',time_used6,'秒')

# nx.draw(G6,with_labels=True)
# plt.title('RodeEU_gcc')
# plt.show()
# print('G6-589 邻居',G6['589'])
```