



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 2018 级数据科学班

学生姓名 学号

王 紫 201800810253

叶家辉 201800830004

日 期: 2020 年 12 月 29 日

得 分:

全局最小割的问题分析及算法研究

摘要：全局最小割问题是图论中的重要问题，在工程实践中有着广泛的应用场景和光明的应用前景。在深入研究问题前，先对问题有一个全局性的认识是至关重要的。本文首先定性地探讨了什么是最小割和全局最小割问题，并进一步分析了它们的时间复杂度，对它们是否属于P问题或NP问题做出了讨论。针对求解全局最小割的问题，历史上已有许多研究者提出了一系列行之有效的巧妙方法，其中包括通过引入随机方法降低复杂度的Karger算法和通过循环查找任意s-t最小割来找到全局最小割的Stoer-Wagner算法。本文从理论与实践两方面深入研究了这两种算法，并借助Python程序设计语言进行了算法实现。然后，在六个给定的网络上对上述基于Python实现的两种算法进行了测试，用以检验算法的可靠性。最后，结合搜集到的文献资料与自己的实验研究，对两种算法的改进和推广提出可能的方案。

关键词：全局最小割、P/NP问题、Karger算法、Stoer-Wagner算法

1 最小割与全局最小割问题的概述

最小割问题是图论中的典型问题之一，指的是将图的顶点划分为两个集合，使得端点分别位于这两个集合的一组边总权重最小。它在现实生活中有非常多的实际应用，如网络可靠性分析等。

最小割问题有许多不同的定义，大体上可分为s-t最小割和无s-t最小割两类。当图中给定了源节点(source)和宿节点(sink)，要求源节点和宿节点必须分别位于划分后的两个集合，该类问题称为s-t最小割问题；若不指定源节点与宿节点，该类问题称为无s-t最小割问题。

而网络本身可从是否具有权重和方向的角度分为加权图、无权图和有向图、无向图，由于无权图可视为权重相等且均为1，因此上述两类最小割问题可细分为有向加权和无向加权两种情形。特别的，在无s-t和无向的情形下，最小割问题也称全局最小割问题。

最小割问题	s-t 最小割	有向加权
		无向加权
	无 s-t 最小割	有向加权
		无向加权/全局最小割

目前，已有许多算法可以解决最小割问题。我们在第一小节中对表格前三种类型的最小割进行了细致探讨，在第二小节中论述了全局最小割问题，第三小节对最小割的变体——最小 k 割进行了简单介绍。

1.1 最小割问题的探讨

1.1.1 s - t 有向加权情形

s - t 最小割问题指定了源节点与宿节点，在有向加权情形下，由最大流最小割定理易知，此时要求解的 s - t 最小割问题，可以转换为其对偶问题—— s 为源点， t 为汇点的最大流问题。目前，存在许多多项式时间的算法可以解决最大流问题。下面以寻找增广路的 Edmond-Karp 算法为例，对 s - t 最小割的问题进行求解。

Edmond-Karp 算法的步骤如下：

1. 初始化网络；
2. 利用 BFS（广度优先搜索）在残差网络中找到一条 s 至 t 的最短路径，即增广路；
3. 令这条增广路上允许通过的最大流量为 Min ，增广路所有正向边的剩余流量减去 Min ，反向边加上 Min ；
4. 重复步骤 2、3，直到没有增广路的存在。

接下来分析该算法的时间复杂度。从上述算法步骤可以看出，算法总时间 = BFS 运行时间 \times BFS 运行次数。在实际分析中，我们发现关键边的个数与 BFS 运行次数是等价的。关键边，指剩余容量恰为该增广路允许通过的最大流量的边。我们有如下定理：

定理： *Edmond-Karp 算法中关键边的总数为 $O(VE)$ 。*

证明： 首先，对于关键边 (u,v) ，由于 (u,v) 位于最短路径上，因此有 $d[v] = d[u] + 1$ 。增广后， (u,v) 重新成为关键边的条件是 (v,u) 出现在新的增广路上，此时 $d'[u] = d'[v] + 1$ 。因为每次增广都会使 s 到所有顶点 $v \in V - \{s,t\}$ 的最短距离 $d[v]$ 增大， $d'[v] \geq d[v]$ 。于是有 $d'[u] \geq d[v] + 1 = d[u] + 2$ ，即边 (u,v) 从成为关键边到下一次成为关键边，源结点 s 到 u 的距离至少增加 2 个单位。同时，源结点 s 到 u 的中间结点不可能包括 s 、 u 或 t ，所以 $d[u]$ 距离最长为 $|V| - 2$ 。

因此，每条边最多做关键边 $\frac{|V|}{2} - 1$ 次。图中共有 E 条边，所以关键边的总数为 $O(VE)$ 。证毕。

而每次 BFS 运行时间为 $O(E)$ ，故 Edmond-Karp 算法的复杂度为 $O(VE) \times O(E) = O(VE^2)$ 。由此可得出结论，存在多项式时间算法 Edmond-Karp 找到最大流(s-t 最小割)。同时可以断定，有向加权的 s-t 最小割问题属于 P 类问题。

由于 P 类问题是 NP 类问题的子集，因此有向加权的 s-t 最小割既属于 P 类问题，也属于 NP 类问题。

1.1.2 s-t 无向加权情形

对于无向加权情形，我们可以将其当做有向情形来处理——为每一条无向边建立两条反向的有向边，其中，反向边的容量和正向边相同。而利用 EK 算法进行求解的过程中，这两条边都会生成对应的反向边。因此，一条无向边实际上对应 4 条边。

由上可知，无向加权的 s-t 最小割既属于 P 类问题，也属于 NP 类问题。

1.1.3 无 s-t 有向加权情形

在无 s-t 的情形下，我们可以在图中固定一点作为 s ，穷举所有可能的 t ，分别求解 s-t 最小割，其中最小值即为求解的最小割。

在 1.1.1 中，我们得出结论，有向加权情形下，存在 Edmond-Karp 算法使得找到某一指定 s-t 最小割的时间复杂度为 $O(VE^2)$ 。因此，通过上述的穷举方式得到全局最小割的时间复杂度为 $O(VE^2) \times O(V) = O(V^2E^2)$ 。

由上可知，有向加权的无 s-t 最小割既属于 P 类问题，也属于 NP 类问题。

1.2 全局最小割问题的探讨

由定义知，全局最小割是无向加权的无 s-t 最小割问题。类似的，可以通过 1.1.3 中的穷举方式求解。

在 1.1.2 中，我们得出结论，无向加权的 s-t 最小割问题可以在多项式时间内求解。因此，通过穷举方式仍然可以在多项式时间内求解全局最小割。

由上可知，全局最小割既属于 P 类问题，也属于 NP 类问题。

从算法复杂度来看，虽然我们能够在多项式时间内求解全局最小割问题，

但在实际生活中，需要处理的往往是大型网络，运行效率仍然很低。因此，出现了许多随机化算法寻找全局最小割。后文我们将深入探讨 Karger 算法和 Stoer-Wagner 算法，它们的出现极大地提升了寻找全局最小割的效率。

1.3 最小 k 割的探讨

最小 k 割问题是最小割问题的延伸。它指的是移除掉该组割后，形成了 k 个连通分量。通过查阅文献[1]，我们了解到，对于固定的 k，存在多项式时间算法解决最小 k 割问题；若 k 不确定，是程序的额外输入，则是一个 NPC 问题。

因此，我们得出结论：对于固定的 k，最小 k 割是 P 问题，也是 NP 问题；若 k 不固定，它是一个 NPC 问题，自然也是 NP 问题。但目前不存在多项式时间解决算法，因此无法确定是否为 P 问题。

2 全局最小割求解算法

2.1 Karger 算法

2.1.1 算法概述

Karger 算法是非常著名的基于随机化思想的全局最小割算法，它的描述十分简单：随机选择图中一条边，把边的两个端点合二为一。将原来与这两个点相邻的边连到合并后的节点去，同时删除所有由于合并而形成自环的边。当合并至图中只剩下两个点时，一次搜索结束，得到一组割，并保存两点之间的边数。

可以看到，由于随机选取，上述的合并过程找到的割并不一定是全局最小割。但若将此过程重复执行，我们能够以低于某个极小值的失败概率获得全局最小割。这就是 Karger 算法的基本思想。

2.1.2 理论分析

这一节将从数学理论的角度分析算法的正确性。

以 n 个点的无向图为例进行说明。假设实际的全局最小割的边数为 c，由反证法易知，图中每个点的度数至少为 c（否则若某个点的度数小于 c，可以将这个点和其余点分开，形成的割边数小于 c，与假设矛盾），那么整张图至少有 $\frac{nc}{2}$ 条边。

由于图的全局最小割可能不止一种，在这里仅考虑一组特定割的情况。假设这一组割边数为 c 。要想获得正确全局最小割，在选择边进行合并时则不能选到这特定的 c 条边。即：

$$P(\text{未选择最小割集中的边}) = 1 - \frac{c}{\text{总边数}} \geq 1 - \frac{2}{n} \quad (1)$$

注意到，上述概率不等式 (1) 在整个合并过程中均成立。由此可推出，一次合并过程找到的割恰好为全局最小割的概率至少为：

$$\left(1 - \frac{2}{n}\right) \times \left(1 - \frac{2}{n-1}\right) \cdots \times \left(1 - \frac{2}{3}\right) = \frac{n-2}{n} \times \frac{n-3}{n-1} \cdots \times \frac{1}{3} = \frac{2}{n(n-1)} \geq \frac{1}{n^2} \quad (2)$$

因此，执行一次 Karger 算法找到全局最小割的概率小于或等于 $1 - \frac{1}{n^2}$ ，执行 m 次未找到全局最小割的概率则在 $\left(1 - \frac{1}{n^2}\right)^m$ 以下。当取 $m = n^2 \ln n$ 时，易知：

$$P(\text{执行 } n^2 \ln n \text{ 次未找到全局最小割}) \leq \left(1 - \frac{1}{n^2}\right)^{n^2 \ln n} \leq e^{-\ln n} = \frac{1}{n} \quad (3)$$

当 $n \rightarrow \infty$ ，即网络中的点足够多的时候，失败的概率极小，可忽略不计。

2.1.3 python 实现

2.1.3.1 库函数的导入与操作

网络的输入一般是一系列包含两个结点序号的数对，其中每个数对表示一条边。直接处理这样的数据是抽象的、困难的，因此第一步需要将输入的网络转化成邻接矩阵的表示形式。这一过程可以借助 NetworkX 工具包来实现。

而对于矩阵的操作就需要借助 NumPy 工具包，这可以大大降低操作难度，提升操作效率。

库函数的导入具体代码如下所示：

```
import numpy as np
import networkx as nx
```

在程序中，我们从外部读入.txt 文件（以 Benchmark Network 为例），用这些点与边的数据为初始化的图添加点和边，再借助 NetworkX 获得这张图的邻接矩阵 G_mat 。当然，也可以选择把这张图打印出来。

网络操作的具体代码如下所示：


```

E = np.loadtxt("data/BenchmarkNetwork.txt")
G = nx.Graph()
G.add_edges_from(E)
A = nx.adjacency_matrix(G)    # 得到图的邻接矩阵
G_mat = A.todense()
# print('邻接矩阵:\n',G_mat)  # 输出这个邻接矩阵
# 画出这张图
# nx.draw(G)

```

在本文中，图网络结构的导入均借助 NetworkX 工具包来实现，后续将不再进行格外说明。

2.1.3.2 算法整体框架

karger 算法整体包含以下步骤：

1. 初始化最小割 mincut 为一个比较大的数；
2. 在图 G 中任选一条边，对该边进行 contract 操作；
3. 重复执行步骤 2，直至图中仅剩两个节点，得到一组割。记两点间的连边数为 mc，并令 $\text{mincut} = \min(\text{mincut}, \text{mc})$ ；
4. 执行步骤 2、3 多次，此时的 mincut 即为全局最小割。

步骤 2 中任选一条边的操作，我们通过依次选取两个点 u, v 来实现；contract 操作则通过更新邻接矩阵等变量来实现，具体方式将在下文中细致说明。

2.1.3.3 实现细节

以下是利用 Python 实现上述步骤的一些具体细节。

1. 用于记录点度数的向量 D

在程序中，我们利用 networkx 得到邻接矩阵之后，接着生成向量 D 以记录各点的度数。它的作用有如下几点：

- (1) 便于每次选择点 u ；

我们每次在 D 中选择一个度数非零的节点 u ，接着利用邻接矩阵任意选取一个与其有邻边的顶点 v 。将 u, v 合并，形成新的 u 点。合并后的 u 点拥有的度数为原来两点度数之和，再减去 u, v 之间重复计算的邻边。而点 v 实际上已不存在，于是将 v 的度数置为 0。因此，每次从 D 中任选一个度数非零的节点作为点 u 即可。

- (2) 便于停止一次搜索和记录割边数；

当合并至图中仅剩两个节点时，一次搜索停止。在 D 中则表现为仅有两点

的度数非零；同时，这两点的度数（必然相同）则为该组割的边数。

2. contract 具体操作

此过程分为两步。首先删去自环的影响，即将邻接矩阵[u, v]和[v, u]处的值置为 0；其次，对图中的非 u、v 点进行更新。这些点与 u 相连的边数相当于原来的连边数加上与 v 的连边数，而与 v 相连的边均置为 0。

代码中的具体更新步骤如下：

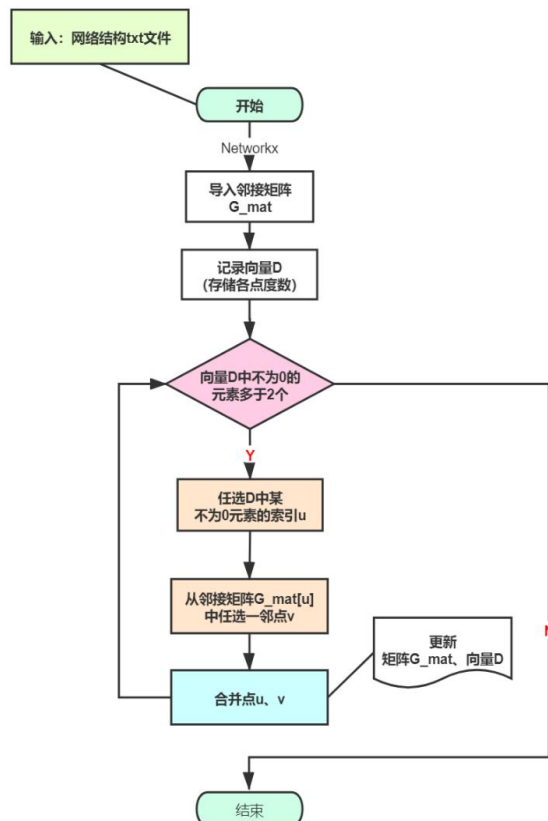
(1)更新向量 D

```
D(u) = D(u) + D(v) - 2G_mat(u, v)
D(v) = 0
```

(2)更新邻接矩阵 G_mat

```
G_mat[u, v] = 0, G_mat[v, u] = 0
v_ = np.where(graph[v] > 0)[1] # 与 v 相连的顶点
# 更新与 v 有边的点的列表
for vertex in v_:
    if vertex != u and vertex != v:
        graph[vertex, u] = graph[vertex, u] + graph[vertex, v]
        graph[u, vertex] = graph[u, vertex] + graph[v, vertex]
        graph[vertex, v] = 0
        graph[v, vertex] = 0
```

下图为 Python 实现 karger 算法的整体流程图。完整代码请见附录 A。



2.1.3.4 输出与作图

在算法的最后，我们采用“递归加点”的方法找到了具体的割。“递归加点”，就是从最后图中剩下的两点出发，依次找到它们合并过的点，最后返回两个点集。

下面是找到合并过的点的核心函数，`u_list` 和 `v_list` 是程序中两两合并过的顶点，彼此一一对应：

```
# 规约找点：输入索引，返回索引
def find_nodes(array, u_list, v_list, nodes):
    v_ = []
    u_ = []
    for each in array:
        v_.append(v_list[each])
    nodes.extend(v_)
    for every in v_:
        u_.extend([i for i, x in enumerate(u_list) if x == every])
    if len(u_) == 0:      # 空数组,即已找到所有合并过的点
        return 0
    return u_
```

接下来是规约操作，为图中剩下的点依次找到其合并过的所有点。在这里，我们定义了 `find_set` 函数，该函数为我们返回合并过的点集，其中，`node_set` 存有最后剩下的节点索引：

```
def find_set(set, u_list, v_list, nodes):
    result1 = find_nodes(set, u_list, v_list, nodes)
    while result1 != 0:
        result1 = find_nodes(result1, u_list, v_list, nodes)
    nodes = [f(x) for x in nodes]
    return nodes

# 输出点集
set1 = ([i for i, x in enumerate(u_list) if x == node_set[0]])
setA = find_set(set1, u_list, v_list, A)
print("点集 1: ", setA)

set2 = ([i for i, x in enumerate(u_list) if x == node_set[1]])
setB = find_set(set2, u_list, v_list, B)
print("点集 2: ", setB)
```

注意到，由于在程序中我们直接对邻接矩阵进行操作，即默认将点标号映射至索引 0、1、2……，因此最后输出点时需要将其还原至原来的编号，这由函数 f 实现：

```
pre = list(np.loadtxt(data, dtype=np.int).flatten())
new_ = list(set(pre))
new_.sort(key=pre.index)    # 点的排序

# 点标号的映射
def f(x):
    value = new_[x]
    return value
```

最后，我们作出合并后的网络图。由于 networkx 自带的绘制函数无法展示平行边，我们通过读取邻接矩阵手动绘制，并将其封装成 plot 函数：

```
# 作图
def plot(matrix):
    mat = copy.deepcopy(matrix)
    arr = []
    for i in range(len(mat)):
        for j in range(len(mat)):
            if mat[i, j] > 0:
                mat[j, i] = 0
                number = mat[i, j]
                for m in range(number):
                    arr.append((f(i), f(j)))

    # print(array)
    G = nx.MultiGraph(arr)
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, node_size= 550, node_color='r', alpha=1)
    ax = plt.gca()
    for e in G.edges:
        ax.annotate("",
                    xy=pos[e[0]], xycoords='data',
                    xytext=pos[e[1]], textcoords='data',
                    arrowprops=dict(arrowstyle="-", color="0.5",
                                    shrinkA=5, shrinkB=5,
                                    patchA=None, patchB=None,

connectionstyle="arc3,rad=rrr".replace('rrr', str(0.05 * e[2])), ), )
    nx.draw_networkx_labels(G, pos,
                            labels=None, font_size=20,
                            font_color='k', font_family='sans-serif',
                            font_weight='normal', alpha=1.0, bbox=None, ax=None)
```

```
# 保存为透明图像
plt.savefig("图片.png", transparent=True)
plt.show()
```

2.2 Stoer-Wagner 算法

2.2.1 算法概述

Stoer-Wagner 算法可以用来求解无向图 $G=(V,E)$ 的全局最小割。该算法基于这样的基本定理，即对于图中的任意两个结点 s 和 t ，图 G 的全局最小割或者等于图中的 s - t 最小割，如若不然的话就等于将 s 和 t 进行合并(merge)后得到收缩(shrink)的图 G' 的全局最小割[2]。合并的过程具体来说就是删除被合并的两个点并添加一个新的点，新的点与其它点边的权值等于被合并的两个点与其权值的和（没有边记权值为 0）。如此一来我们就可以对图进行循环操作，直至图收缩到只剩下一个结点，然后找到在这过程中最小的 s - t 最小割，即为全局最小割。

可以看出，Stoer-Wagner 算法与一般最小割算法的不同之处在于，它的 s - t 点是在求解过程中产生的，而非预先给定的。在每一次寻找 s - t 最小割的时候，Stoer-Wagner 算法则借助 prim 算法扩展出最大生成树，并记录下最后扩展到的点和最后扩展到的边。

2.2.2 理论分析

定义 $w(A, x) = \sum w(v[i], x), v[i] \in A$ 。定义 A_x 为在 x 前加入 A 的所有点的集合（不包括 x ）。

Stoer-Wagner 算法中需要求解任意 s - t 最小割。具体的流程为：

1. 初始化 A 为空集，从 V 中任取一个点加入 A ；
2. 每次选取集合 $V \setminus A$ 中使得 $w(A, x)$ 最大的点 x 加入集合 A ；
3. 当 $|A|=|V|$ 时结束，否则重复第 2 步。

则最后一个加入 A 中的点为 t ，倒数第二个加入 A 中的点为 s 。 s - t 最小割为 $w(A_t, t)$ 。该方法与 Dijkstra 和 Prim 的算法类似。

我们来证明一下上述寻找任意 s - t 最小割方法的正确性[3]。

令 C 为任意一个 s - t 割。定义一个点 v 是活跃的，当且仅当 $(u, v) \in C$ ，其中 u 为在 v 前一个加入 A 中的点。定义 $C_v = \{(a, b) \mid a, b \in (A_v \cup v) \text{ 且 } (a, b) \in C\}$ 。

下面需要证明 $w(A_t, t) \leq w(C_t) = w(C)$ 。

由于 t 显然为活跃点，下面应用数学归纳法证明对于任意活跃点 x 有 $w(A_x, x) \leq w(C_x)$ 。

首先证明第一个活跃点。

设第一个活跃点为 $v[0]$ ，有 $w(A_{v[0]}, v[0]) \leq w(C_{v[0]})$ 。

由于 $v[0]$ 为第一个活跃点，所以 $A_{v[0]}$ 中的点到 $v[0]$ 的边是仅有的跨越 $C_{v[0]}$ 的边，故上式等号成立。

下面由数学归纳法证明，假设有 $w(A_{v[i-1]}, v[i-1]) \leq w(C_{v[i-1]})$ 成立，则有 $w(A_{v[i]}, v[i]) \leq w(C_{v[i]})$ 成立。

令 $v[i-1]$ 为任意活跃点， $v[i]$ 为 $v[i-1]$ 之后第一个活跃点，设 $w(A_{v[i-1]}, v[i-1]) \leq w(C_{v[i-1]})$ 成立，由 $w(A, x)$ 定义可得： $w(A_{v[i]}, v[i]) = w(A_{v[i-1]}, v[i]) + w(A_{v[i]} - A_{v[i-1]}, v[i])$ 。

由于 $v[i-1]$ 在 $v[i]$ 前加入 A ， $w(A_{v[i-1]}, v[i]) \leq w(A_{v[i-1]}, v[i-1])$ 。

由归纳假设 $w(A_{v[i-1]}, v[i-1]) \leq w(C_{v[i-1]})$ ，故 $w(A_{v[i]}, v[i]) \leq w(C_{v[i-1]}) + w(A_{v[i]} - A_{v[i-1]}, v[i])$ 。

$w(A_{v[i]} - A_{v[i-1]}, v[i])$ 对 $w(C_{v[i]})$ 有贡献而对 $w(C_{v[i-1]})$ 没有。故 $w(C_{v[i-1]}) + w(A_{v[i]} - A_{v[i-1]}, v[i]) \leq w(C_{v[i]})$ 。

所以 $w(A_{v[i]}, v[i]) \leq w(C_{v[i]})$ ，故 $w(A_x, x) \leq w(C_x)$ 对于任意活跃点 x 都成立。

所以 $w(A_t, t) \leq w(C_t) = w(C)$ 成立，即算法所求得的 s - t 割为 s - t 的最小割。

2.2.3 Python 实现

2.2.3.1 库函数的导入与操作

与 2.1.3.1 节“库函数的导入与操作”完全相同，不再赘述。

2.2.3.2 算法整体框架

Stoer-Wagner 算法整体包含三个步骤：

1. 初始化最小割 mincut 为一个比较大的数；
2. 在图 G 中求出任意 s - t 最小割 mc ，并令 $mincut = \min(mincut, mc)$ ；

3. 对图 G 将 s 和 t 结点进行合并操作 (contract 操作), 得到新的图 G' , 当 G' 中结点个数大于 1 的时候令 $G=G'$, 并回到第 2 步, 如若不然则此时的 mincut 即为原图的全局最小割。

其中, 合并 s 和 t 结点的 contract 操作可以描述为: 删除点 s 和 t , 以及边 (s,t) , 加入新结点 u , 对于任意 $v \in V$, $w(v, u) = w(u, v) = w(s, v) + w(t, v)$ 。

2.2.3.3 实现细节

由于寻找任意 s - t 最小割的过程本身比较慢, 所以我们可以得到最小割为 1 的运行结果只有就跳出 while 循环, 在不改变运行效果的前提下尽可能降低运算成本。这个部分会在后续 4.1.2 和 4.2.1 展开, 并给出优化后的代码。

从实践的层面上来说, 结点合并的过程其实就是更新邻接矩阵 G_mat 的过程。本质上, Stoer-Wagner 算法中的 contract 操作与 Karger 算法中的 contract 操作是一样的。但是更新邻接矩阵的实现方法有很多, 这里选择一种与 Karger 中不同的方法进行说明。具体来说分为三个步骤:

1. 假设当前 G_mat 是一个 $n \times n$ 的矩阵, 则初始化一个 $(n+1) \times (n+1)$ 的 new_mat , 并使 new_mat 的前 n 行 n 列等于 G_mat 矩阵;

```
new_mat = np.zeros((len(G_mat)+1, len(G_mat)+1))
new_mat[:len(G_mat), :len(G_mat)] = G_mat
```

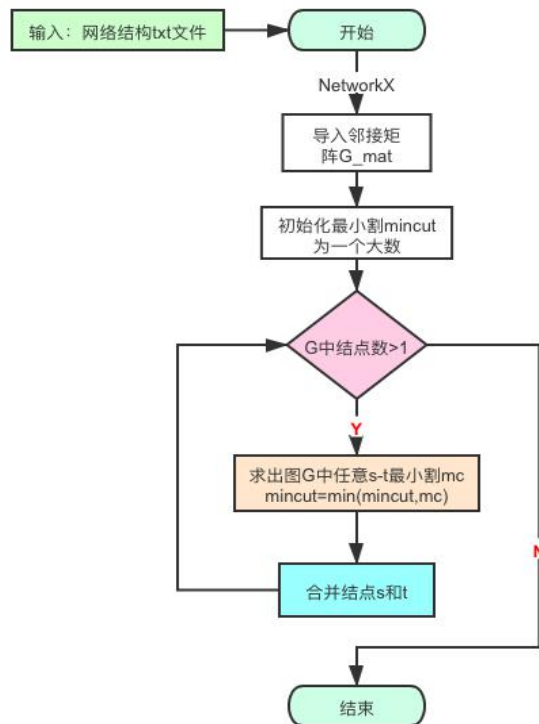
2. 对当前集合 V 中除了 s 和 t 的每一个结点 i , 将它们与 s 和 t 的边权重的和添加到最后一行、最后一列作为新的结点;

```
for i in range(len(G_mat)):
    if i != s and i != t:
        new_mat[i, -1] = new_mat[i, s] + new_mat[i, t]
        new_mat[-1, i] = new_mat[s, i] + new_mat[t, i]
```

3. 删去 new_mat 中的第 (s,t) 行和第 (s,t) 列。

```
new_mat = np.delete(new_mat, (s,t), axis = 0)
new_mat = np.delete(new_mat, (s,t), axis = 1)
```

下图为 Python 实现 Stoer-wagner 算法的整体流程图。完整代码请见附录 B。



2.2.3.4 输出与作图

最后的输出应该包含合并到最后的两个超结点所包含的结点序号，为此我们需要进行一些操作。

为了得到最终的最小割属于哪两个点，我们需要找出到最后有哪两个点还没有被合并。因此可以创建一个数组 D 用来存储每个结点的信息，尚在 D 中序号所代表的结点表示还未被合并的点。当 D 最后剩下两个元素的时候，这两个结点就是全局最小割所在的两个端点。

具体的更新过程为：删除索引为 t 的点在原图中的序号，并把索引为 s 的点在原图中的序号换到最后。为了保证可以正确地根据索引进行删点、换点操作，需要对 $s < t$ 和 $s > t$ 分两种情况讨论，以防止删点之后对换点产生影响，反之亦然。具体代码呈现如下：

```

# 删除 t 点编号，把 s 点编号换到最后面
if s < t:
    D = np.delete(D,t)
    zjl = D[-1]
    D[-1] = D[s]
    D[s] = zjl
else:
    zjl = D[-1]
    D[-1] = D[s]
    
```

```
D[s] = zjl
D = np.delete(D,[t])
```

关于如何得到两个超结点所含结点序号，这里选用一种与上述 2.1.3.4 中不同的方法。首先创建一个列表 E_，然后把合并过的(s,t)数对作为元素添加到列表中。借助 Python 的 NetworkX 模块以 E_为边的集合生成一张图，然后寻找这张图的连通子图（理论上应为两个连通子图），这两个连通子图所包含的结点就为最后的两个超结点所包含的结点：

```
# 通过 E_寻找两个超结点所含结点
G = nx.Graph()
G.add_edges_from(E_)
Set1 = set()
Set2 = set()
components=list(nx.connected_components(G))

for i in components[0]:
    Set1.add(f(i))

if len(components) > 1:
    for i in components[1]:
        Set2.add(f(i))

print(Set1,Set2)
```

注意到，由于在程序中我们直接对邻接矩阵进行操作，即默认将点标号映射至索引 0、1、2……，因此最后输出点时需要将其还原至原来的编号，这由函数 f 实现：

```
L = list(np.loadtxt("data/"+filename+".txt", dtype=np.int).flatten())
Array = list(set(L))
Array.sort(key=L.index)
# 点标号的映射
def f(x):
    value = Array[x]
    return value
```

最后，我们作出合并后的网络图。由于 networkx 自带的绘制函数无法展示平行边，我们通过读取邻接矩阵手动绘制：

```
# 作图展示测试结果
G=nx.MultiGraph(array)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color = 'r', alpha = 1)
ax = plt.gca()
```



```

for e in G.edges:
    ax.annotate("",
                xy=pos[e[0]], xycoords='data',
                xytext=pos[e[1]], textcoords='data',
                arrowprops=dict(arrowstyle="-", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,

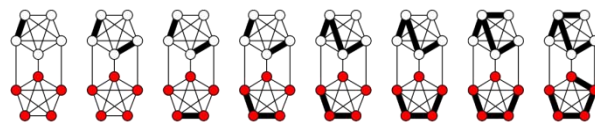
connectionstyle="arc3,rad=rrr".replace('rrr',str(0.3*e[2])
                                ),
                                ),
                )
nx.draw_networkx_labels(G, pos, labels=None, font_size=12, font_color='k',
font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None)
# 保存为透明图像
plt.savefig(filename, transparent=True)
plt.show()

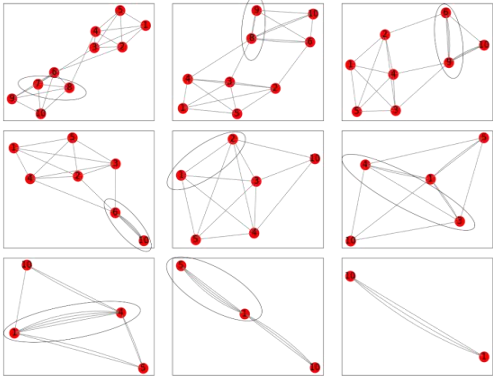
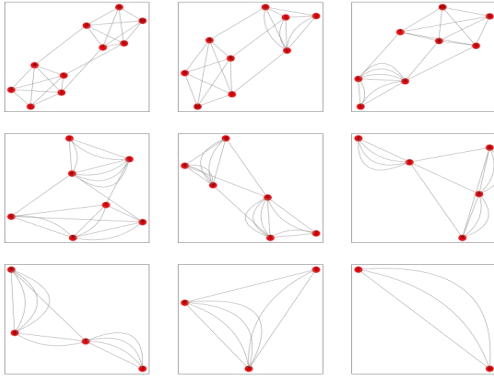
```

3 全局最小割求解算法的数据测试

注：以下所有运行结果中的 SW 算法均未添加 4.2.1 中提及的优化方法。

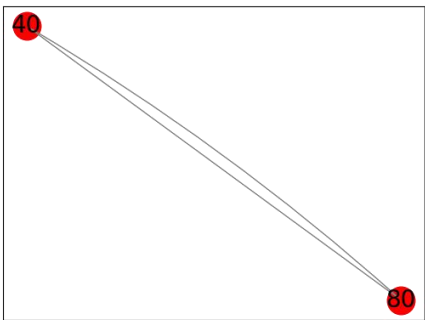
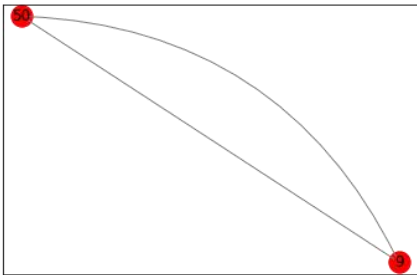
3.1 示例网络的测试



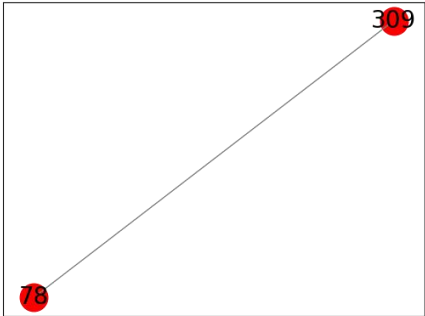
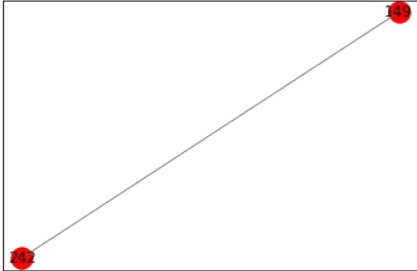
Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
{1,2,3,4,5}	{6,7,8,9,10}	{1,2,3,4,5}	{6,7,8,9,10}
			
全局最小割	3	全局最小割	3
合并过程	(8,7)	合并过程	(1,3)

	(9,8)		(4,5)
	(6,9)		(10,9)
	(10,6)		(7,9)
	(1,2)		(4,1)
	(4,3)		(9,8)
	(1,4)		(6,9)
	(1,5)		(6,8)

3.2 Benchmark Network 的测试

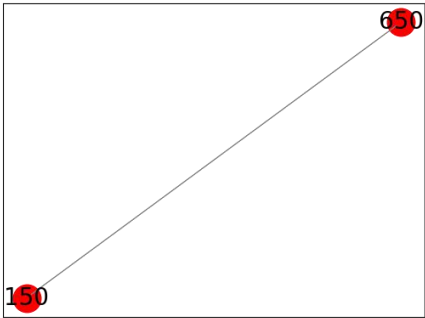
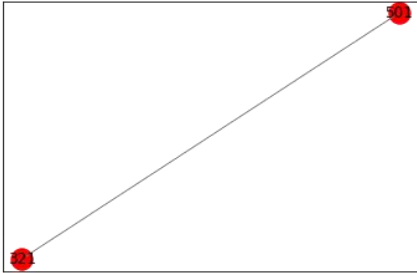
Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
{40, 29, 27, 35, 3, 37, 25, 30, 15, 28, 24, 39, 18, 6, 11, 23, 21, 26, 14, 12, 13, 20, 8, 1, 9, 33, 19, 10, 16, 7, 17, 4, 2, 5, 32, 36, 22, 38, 34, 81, 31}	{80, 70, 77, 53, 48, 51, 78, 52, 55, 57, 74, 79, 65, 83, 49, 66, 72, 69, 45, 54, 68, 71, 61, 63, 42, 47, 75, 76, 50, 64, 60, 44, 58, 82, 46, 56, 41, 67, 62, 43, 59, 73}	{3, 4, 5, 11, 12, 15, 16, 17, 18, 19, 20, 21, 23, 25, 26, 27, 28, 29, 30, 33, 34, 35, 36, 38, 39, 40, 41, 44, 47, 48, 49, 50, 51, 52, 57, 58, 59, 62, 63, 64, 65, 66, 67, 68, 69, 71, 72, 73, 74, 76, 77, 78, 81, 82, 83}	{1, 2, 6, 7, 8, 9, 10, 13, 14, 22, 24, 31, 32, 37, 42, 43, 45, 46, 53, 54, 55, 56, 60, 61, 70, 75, 79, 80}
			
全局最小割	2	全局最小割	2

3.3 Corruption Gcc 的测试

Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
V-B	{309}	V-B	{7, 8, 11, 13, 18, 20, 21, 24, 26, 28, 33, 45, 48, 50, 52, 53, 55, 56, 60, 62, 68, 81, 82, 89, 90, 100, 101, 102, 108, 109, 111, 118, 119, 120, 122, 123, 126, 133, 135, 140, 147, 148, 158, 161, 172, 175, 176, 178, 181, 183, 186, 190, 202, 206, 208, 209, 215, 221, 222, 225, 227, 229, 232, 235, 242, 243, 244, 247, 259, 260, 264, 265, 268, 269, 272, 275, 276, 278, 281, 285, 294, 301, 305}
			
全局最小割	1	全局最小割	1

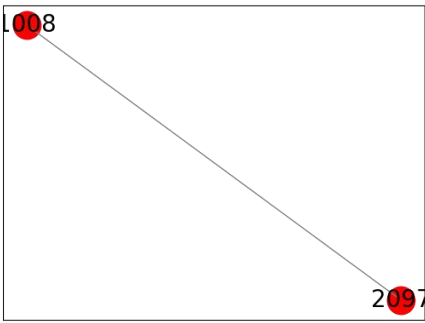
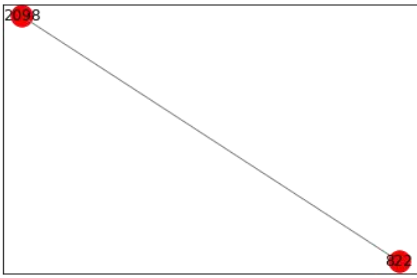
3.4 Crime Gcc 的测试

Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
V-B	{650}	V-B	{1, 3, 518, 521, 522, 15, 21, 31, 34, 46, 50, 563, 564, 565, 58, 574, 575, 63, 73, 590, 80, 81,

			86, 87, 88, 90, 95, 618, 625, 117, 630, 119, 633, 635, 128, 640, 652, 657, 662, 668, 669, 164, 171, 684, 174, 691, 181, 696, 698, 187, 188, 701, 198, 712, 715, 218, 734, 737, 226, 747, 235, 749, 237, 752, 252, 257, 276, 281, 283, 284, 297, 299, 306, 308, 309, 321, 330, 334, 338, 341, 349, 350, 353, 370, 384, 389, 405, 407, 415, 468, 476, 480, 481, 484, 496}
			
全局最小割	1	全局最小割	1

3.5 PPI Gcc 的测试

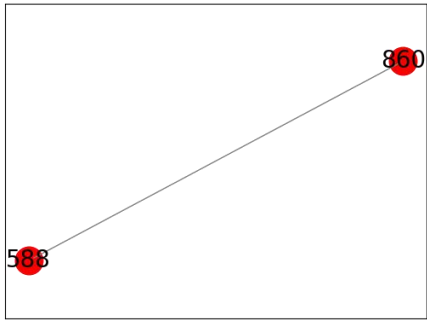
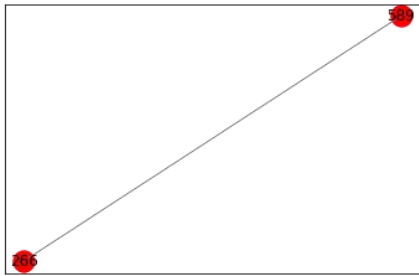
Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
V-B	{2097}	V-B	{384, 2067, 2070, 2071, 2078, 2079, 2080, 2082, 2083, 2088, 1066, 2090, 2092, 2093, 2094, 689, 2098, 2102, 695, 2104, 1502}

			
全局最小割	1	全局最小割	1

3.6 RodeEU Gcc 的测试

Karger		Stoer-Wagner	
点集 A	点集 B	点集 A	点集 B
V-B	{860}	V-B	{512, 513, 514, 515, 3, 517, 518, 519, 8, 516, 1, 11, 15, 16, 17, 531, 20, 21, 532, 535, 533, 534, 27, 30, 31, 34, 546, 547, 549, 38, 548, 552, 553, 42, 43, 556, 47, 560, 51, 564, 53, 54, 566, 568, 567, 570, 571, 569, 572, 573, 574, 576, 65, 577, 580, 585, 589, 590, 591, 592, 596, 597, 85, 599, 600, 601, 90, 91, 92, 86, 603, 95, 606, 607, 614, 104, 616, 617, 115, 117, 118, 122, 634, 124, 125, 637, 131, 645, 135, 648, 649, 140, 141, 144, 658, 659, 149, 661, 662, 154, 155, 156, 669, 670, 158, 160, 161, 159, 681, 690, 691, 178, 179, 180, 696, 702, 703, 192, 709, 199, 711, 204,

			<p>717, 716, 719, 208, 721, 722, 214, 217, 218, 219, 220, 221, 222, 223, 730, 225, 226, 227, 228, 741, 230, 742, 232, 233, 743, 238, 239, 751, 243, 246, 247, 760, 249, 761, 762, 251, 252, 248, 766, 255, 256, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 780, 781, 257, 782, 278, 279, 791, 281, 282, 280, 283, 797, 284, 287, 289, 290, 291, 803, 292, 804, 293, 296, 809, 297, 294, 811, 812, 305, 313, 315, 317, 318, 831, 832, 833, 333, 334, 847, 336, 848, 850, 335, 341, 343, 344, 346, 350, 351, 352, 874, 875, 876, 368, 369, 370, 371, 885, 376, 377, 378, 379, 897, 898, 900, 389, 390, 901, 391, 388, 902, 392, 903, 393, 398, 904, 404, 917, 406, 918, 923, 411, 925, 926, 927, 413, 417, 418, 415, 932, 416, 422, 934, 432, 440, 954, 454, 457, 460, 461, 462, 463, 978, 472, 473, 474, 475, 483, 484, 485, 487, 493, 496, 508, 509, 510, 511}</p>
--	--	--	--

			
全局最小割	1	全局最小割	1

4 全局最小割求解算法的改进与推广

4.1 历史文献的改进与推广

4.1.1 Karger-Stein 改进算法

4.1.1.1 算法概述

由上述2.1.2中不等式 (1) 知，每次在选择点进行合并时，选择正确的概率至少为 $1 - \frac{2}{n}$ (n 为当前图中顶点数)。每合并一次，图中点数减少1，由此可推断，前期合并效果较好，而随着合并步骤的进行，选择正确边数的概率在减小。基于此，David Karger和Clifford Stein提出了Karger-Stein改进算法。

Karger-Stein改进算法的思路主要有以下三点：

1. 共享效果好的合并部分以节省运行时间

前期由于节点数很大，合并效果较好。若每次都从原始图网络出发进行合并，需耗费大量时间。因此，可以共享前期的合并节点，而总体效果并不会变差。

2. 在后期容易错选节点的步骤上多次实验

随着后期节点数的减少，错选边的概率会越来越大。因此在后期合并时多次实验。

3. 利用规约思想得到全局最小割

设定规约操作的下限节点6。每次合并至网络中有 $\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil$ 个节点，并在此基础上多次迭代。当图中节点大于6时，反复进行规约操作；否则直接利用karger中的操作得到一组割集。由于多次迭代，最终能得到大量割集，返回这些割的最小值，即为全局最小割。

4.1.1.2 算法实现

可以看出，该算法的基本步骤与karger算法无异，差别在于利用了规约思想降低算法的时间复杂度：合并节点直到图中节点数为 $\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil \approx 70\% \times n$ ，再在此图基础上继续合并约70%的节点，多次操作，返回割的最小值。具体实现过程如下：

1. 选择节点合并，直到网络中仅有 $\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil$ 个节点。此步骤执行两次，得到两个图 G_1 、 G_2 。

2. 当网络中节点数大于6时，分别对 G_1 、 G_2 重复操作步骤（1）；否则利用karger算法中的合并操作，直接得到一组割。

3. 取得到的全部割集的最小值，即为全局最小割。

4.1.1.3 时间复杂度

从算法流程中可知，可得到时间复杂度的递归方程：

$$T_n = 2T\left(\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil\right) + O(n^2)$$

解得， $T_n = O(n^2 \log n)$ 。运行 $\log^2 n$ 次，可知，本算法总体的时间复杂度为 $O(n^2 \log^3 n)$ 。

4.1.2 并行化方法

4.1.2.1 算法概述

David Karger和Clifford Stein在Karger-Stein改进算法的同一篇论文中提出了并行化方法，以进一步降低时间复杂度。在正式介绍该方法之前，两人对前文中的选择节点、合并等操作进行了重定义。

Karger算法寻找割的关键在于任意挑选边，将边的两个点进行合并，直到最后图中只剩两个节点。因此从逆向角度看，可以用如下方式寻找一组割：将原始网络的边与点剥离开，并对边进行任意排序，每次往图中添加一条边，并将边的两个点合并；直到图中只剩下两个连通分量，剩下的边中顶点分别位于这两个连通分量的边，即为一组割。

若逐次添加一条边，效率太低。因此在并行改进算法中，两人采用了二分法进行搜索，大大降低了时间复杂度。

4.1.2.2 算法实现

符号说明

符号	含义
L	随机打乱的边集
V	图中所有的顶点
$H(V, L')$	由 L 的子集 L' 与 V 构成的图网络
L_1/L_2	将 L_2 与点集 V 进行合并后, 仍在 L_1 中的边 (即非自环边)

算法整体流程如下[4], 它将此思想推广至最小 k -割搜索 (k 为任意正整数):

<p>COMPACT(G, L, k)</p> <p><u>input:</u> A graph G, list of edges L, and parameter k</p> <p>if G has k vertices or $L = \emptyset$</p> <p>then</p> <p style="padding-left: 20px;">return G</p> <p>else</p> <p style="padding-left: 20px;">Let L_1 and L_2 be the first and second halves of L</p> <p style="padding-left: 20px;">Find the connected components in graph $H = (V, L_1)$</p> <p style="padding-left: 20px;">if H has fewer than k components</p> <p style="padding-left: 20px;">then</p> <p style="padding-left: 40px;">return COMPACT(G, L_1, k)</p> <p style="padding-left: 20px;">else</p> <p style="padding-left: 40px;">return COMPACT($G/L_1, L_2/L_1, k$).</p>
--

下面以 $k=2$ 为例进行解释说明。具体步骤如下:

1. 将边集 L 一分为二, 得到前后各一半边集 L_1 、 L_2
2. 先选 L_1 与 V 进行合并。若形成了一个连通分量, 则表明选的边多了, 对 L_1 接着进行二分, 重复步骤1、2; 若形成多于1个连通分量, 则对 L_2/L_1 进行二分, 重复步骤1、2; 若刚好形成两个连通分量, 则可以直接得到一组割。

4.1.2.3 算法时间复杂度

该算法的核心在于利用二分法进行搜索, 即每次选一半的边进行合并, 同时检验当前图中连通分量的个数。

假设边的条数为 m , 下表列出了各步骤的算法复杂度。

打乱边序	$O(m)$
二分法搜索	$O(\log m)$

连通分量数的确定	$O(m)$
合并操作	$O(m)$

由此看出，算法复杂度为 $O(m) \times O(\log m) = O(m \log m)$ ，并不够好。因此在论文中，作者提出采用多个处理器并行运算的方式，进一步提高了效率。

具体的并行方式在此不赘述，通过运算可得，当采用 $m = O(n^2)$ 个处理器时，上述1、3、4步骤的复杂度可降为 $O(1)$ 、 $O(\log n)$ 、 $O(1)$ 。从而整体的时间复杂度为：

$$O(\log m) \times (O(1) + O(\log n)) = O(\log^2 n)$$

4.1.2 Stoer-Wagner 的改进算法

前面提到过，全局最小割或者等于图中任意 $s-t$ 最小割，或者等于将 s 和 t 点合并后形成的新图的全局最小割[2]，所以全局最小割在数值上一定不超过当前找到的 $s-t$ 最小割。由此，可以在算法找到大小为 1 的割的时候就退出循环，即代表这张图的全局最小割就是 1。可以说，这是一种基于全局最小割特性的优化方案。

另一方面，来考察一下每次选取边进行合并的过程。SW 算法中求解 $s-t$ 最小割的步骤，最优先考虑的应当是 Dijkstra 和 Prim 的算法[5]。因此我们可以马上知道，使用 Fibonacci 堆来实现求解一次 $s-t$ 最小割的时间复杂度为 $O(m + n \log n)$ 。也就是说，SW 算法一共要求解 n 轮这样的 $s-t$ 最小割然后合并它们，而每一轮就要耗费 $O(m + n \log n)$ 这么多的时间，总的时间复杂度就为 $O(mn + n^2 \log n)$ 。有没有什么更快速的算法呢？

我们发现，耗时较多的是每次寻找 $s-t$ 最小割的过程。其实可以想到，我们也许不必每次都规规矩矩找到一组 $s-t$ 最小的割，而是引入一定的随机性，以此来降低运算量。基于这样的思想，Karger 和 Stein 提出了一种提高运算效率的方法：每次随机选择一条边进行 contract 操作[5]。

为简单起见，我们在此假设输入图 G 为无权图。收缩边缘可能会产生自环和平行边。丢弃自环，但保留平行边，因为这是必不可少的，这对应于上一部

分中的权重运算。算法以相同的概率选择每个边。边的收缩会创建超结点 (super-vertices)，这些超结点对应于原始图的顶点子集。边缘将收缩，直到只剩下两个超结点。该算法返回这两个剩余的超结点。

这样的思路发展下去就锻造了 Karger 算法。因此从某种意义上说，Karger 算法是 SW 算法的一种优化，是通过引入随机选边来降低时间复杂度的一种做法。

结合 2.1.2 中的理论分析，由 (2) 式可以得到，对于含有 n 个结点的图，通过随机寻找合并的结点，一轮合并过程找到的割恰好为全局最小割的概率 $\geq \frac{1}{n^2}$ ，即执行一次算法找到全局最小割的概率 $\leq 1 - \frac{1}{n^2}$ ，当取尝试次数 $m = n^2 \ln n$ 时，就得到 (3) 式： $P(\text{找不到全局最小割}) \leq \left(1 - \frac{1}{n^2}\right)^{n^2 \ln n} \leq e^{-\ln n} = \frac{1}{n}$ 。因此我们可以看到，图中所含结点数量越多，Karger 算法出错的概率就越小。这在直观上是好理解的。图中所含边数的最小值与点的个数直接相关，而因为取到每条边的概率是一样的，所以边的总数越多，随机选择合并边的时候选到最小割的边的概率就越小。

根据上述讨论，对于结点数较多的复杂图，在合并初期，用 Karger 算法是比较不容易出错的。同时，由于当前图中结点数较多，选用时间复杂度较高的 SW 算法来找出 s - t 最小割是比较低效的。但是随着合并的不断进行，边数的减少会增大最小割被合并的风险，这时候又需要通过一种有效的方法合理地选择合并的边，此时还是选用 SW 算法比较保险。由此，一种综合了 Karger 和 SW 两种算法各自优势的改进方案就呼之欲出了。

4.2 本文的改进与推广

4.2.1 基于全局最小割特性的优化方案

由于全局最小割一定是所有存在的最小割中最小的，全局最小割一定小于等于 SW 算法当前找到的 s - t 最小割，所以一旦找到权和为 1 的最小割，程序就可以直接停止，此时全局最小割就为 1。实践表明，对于比较稀疏的图，这一做法可以极大减少运算量，提高程序运行速度。

```
while len(G_mat) > 2:
    s,t,mc = MinCut(G_mat)

    if mc < mincut:
```

```

mincut = mc
# 当发现最小割为 1 时直接退出
if mincut == 1:
    E_.append([D[t],D[t]]) # 相当于此时不要把点添加进去, 同时保证 E_ 中至少有一条边
    D = [D[s],D[t]]
    break

```

基于全局最小割特性的 SW 算法的优化方案完整代码请见附录 C。

4.2.2 结合两种算法的综合优化方案

Wolpert 和 Macready 于 1997 年提出了著名的 “No Free Lunch” 理论[6], 该理论对如何比较两种算法哪种更好做出了阐释。这个理论可以给我们这样的启示, 即其中一种算法对于一个特定的问题在某方面的性能优势, 必然以其在另一些问题或该问题另一些方面上的性能下降作为代价。不同的算法适用于不同的问题, 或者同一问题的不同情况。

具体来说, 在寻找全局最小割的问题上, Karger 算法完成一次结点合并操作的时间复杂度远低于 SW 算法, 但是存在一定的出错的可能, 所以如果要靠 Karger 算法找到真正的全局最小割就需要重复执行多次 Karger 算法, 然后取整个过程中找到的最小的割。而 SW 算法每次都能可靠地找到最小的 s-t 割, 整个执行过程可以确保最终找到的 s-t 割是全局最小的, 但是每一轮找 s-t 割进行结点合并的时间复杂度就比较大。

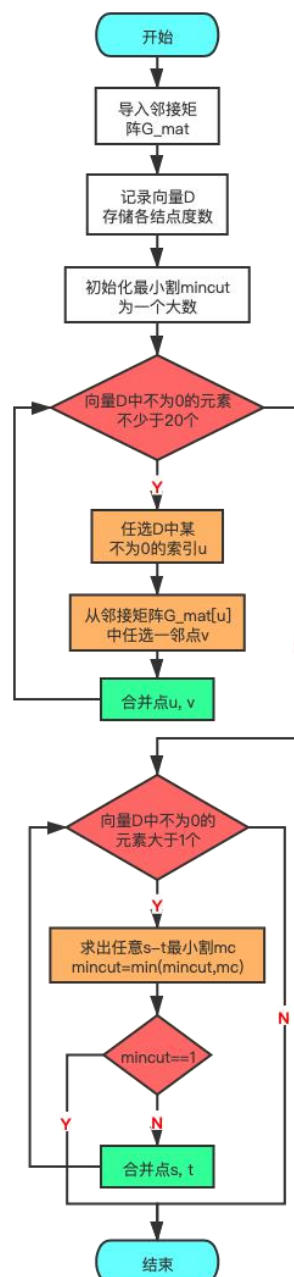
通过 4.1.2 中的讨论, 可以得到这样一种改进算法: 首先进行结点数判定, 如果结点数小于某个阈值 M, 则直接选择 SW 算法; 否则先通过 Karger 算法随机选边进行 contract 操作, 直至合并到结点数小于 M, 再用 SW 算法进行后续的合并。

下一步, 就需要确定阈值 M, 即使用 Karger 算法直至何种境地, 改用 SW 算法比较合适。统计学上认为, 发生概率小于等于 5% 的事件为 “小概率事件”, 即这样的事件理论上可以发生但是发生概率很小, 在一次试验中发生的可能性则几乎为零。在单次实验中认为不可能发生。这其实是基于统计学上的小概率原理, 即小概率事件在一次试验中发生的概率很小, 如果真的发生了, 统计学则怀疑其真实性[7]。因此, 当 Karger 算法的出错率低于 5% 的时候, 可以认为它是可靠的而选择使用它, 如若不然就选择 SW 算法。根据 2.1.2 中的 (3) 式, 当取尝试次数 $m = n^2 \ln n$ 时, 失败的概率小于等于 $\frac{1}{n}$, 即当结点数 $n \geq 20$ 时,

失败概率小于等于 5%。由此我们可以很容易得到，当当前结点数大于等于 20 时选择 Karger 算法，当结点数降至 20 以下（不含）时改用 SW 算法。如此一来可以保证在较低的出错率下最大限度提升算法性能。

```
def karger_Min_Cut(graph, D):
    . . . . .
    while np.sum(D > 0) > 20:      # 设定 Karger 的阈值
        . . . . .
    . . . . .
```

下图为 Python 实现结合 Karger 和 Stoer-Wagner 两种算法并基于全局最小割特性进行改进的整体流程图。完整代码请见附录 D。



4.2.3 改进方案实现细节

值得注意的是，Karger 算法毕竟具有一定的随机性，所以在综合改进方案中我们可以考虑以下两点：

1. 生成多张子图。一次 Karger 运行过程中难免会选到全局最小割进而造成出错，而通过生成多张子图，然后选择这些子图使用 SW 求解全局最小割结果最小的一张图，则可以降低整体出错的概率。当然，这也会增加一定的运算时间，因此需要进行一定的权衡 (trade-off)。正如前文所说，当尝试次数达到 $m = n^2 \ln n$ 时，可以让“出错”变成一件“小概率事件”，但是在实践中不一定需要生成这么多子图（因为引入了 SW 以保证后续操作的正确性）。这里给出参考值 $m = 5$ ，在实际操作中应使得参考值尽量比 5 大，而比 $n^2 \ln n$ 小。

2. 适当增大阈值。在点越来越多的时候 Karger 就越不容易破坏掉全局最小割，因此我们可以及早停止随机合并，以降低出错的概率。上文给出的参考值为 $M = 20$ ，在实际操作中应使得阈值尽量不比 20 小。

参考文献

- [1] Olivier Goldschmidt, Dorit S. Hochbaum. A Polynomial Algorithm for the k-cut Problem for Fixed k[J]. Mathematics of Operations Research, 1994, 19(1).
- [2] Stoer, Mechthild, and Frank Wagner. "A simple min-cut algorithm." Journal of the ACM (JACM) 44.4 (1997): 585-591.
- [3] Etrnls, "最小割 Stoer-Wagner 算法 2007-4-15.
- [4] David R. Karger, Clifford Stein. A new approach to the minimum cut problem[J]. Journal of the ACM (JACM), 1996, 43(4).
- [5] Uri Zwick. "Lecture notes for Analysis of Algorithms: Global minimum cuts". Spring 2008.
- [6] David H. Wolpert. "What Does Dinner Cost?". NASA Ames Research Center <http://ti.arc.nasa.gov/people/dhw/>. NASA-ARC-05-097.
- [7] 朱继民. "如何理解统计学中的小概率事件?". 百度文库.

附录 A

```

import networkx as nx
import numpy as np
import copy
from random import choice
import matplotlib.pyplot as plt

# Corruption_Gcc    Crime_Gcc    PPI_gcc    RodeEU_gcc    BenchmarkNetwork
data = "BenchmarkNetwork.txt"
E = np.loadtxt(data)
G = nx.MultiGraph()
G.add_edges_from(E)
# 得到图的邻接矩阵
A = nx.adjacency_matrix(G)
G_mat = A.todense()

# 得到每个点的度数
D = []
for i in range(0, len(G_mat)):
    each = G_mat[i][0]
    each[each > 0] = 1
    degree = each.sum()
    D.append(degree)
D = np.array(D)

# 点的排序
pre = list(np.loadtxt(data, dtype=np.int).flatten())
new_ = list(set(pre))
new_.sort(key=pre.index)

# 点标号的映射
def f(x):
    value = new_[x]
    return value

def karger_Min_Cut(graph, D):
    pair = []
    while np.sum(D > 0) > 2:
        # 随机选一个顶点
        u_beixuan = np.array(np.where(D > 0))[0]
        u = choice(u_beixuan)
        u_no = np.where(graph[u] > 0)[1]
        v = choice(u_no)    # 选出一条边

```

```

        pair.append((u,v))
        contract(graph, u, v, D)
    return D, pair

def contract(graph, u, v, D):
    # 更新点的度数
    D[u] = D[u] + D[v] - 2*graph[u, v]
    D[v] = 0
    # 删除 uv 相连的边 (自环)
    graph[u, v] = 0
    graph[v, u] = 0
    v_ = np.where(graph[v] > 0)[1] # 与 v 相连的顶点
    # 更新与 v 有边的点的列表
    for vertex in v_:
        if vertex != u and vertex != v:
            graph[vertex, u] = graph[vertex, u] + graph[vertex, v]
            graph[u, vertex] = graph[u, vertex] + graph[v, vertex]
            graph[vertex, v] = 0
            graph[v, vertex] = 0

# 规约找点: 输入索引, 返回索引
def find_nodes(array, u_list, v_list, nodes):
    C = []
    D = []
    for each in array:
        C.append(v_list[each])
    nodes.extend(C)
    for every in C:
        D.extend([i for i, x in enumerate(u_list) if x == every])
    if len(D) == 0: # 空数组
        return 0
    return D

def find_set(set, u_list, v_list, nodes):
    result1 = find_nodes(set, u_list, v_list, nodes)
    while result1 != 0:
        result1 = find_nodes(result1, u_list, v_list, nodes)
    nodes = [f(x) for x in nodes]
    return nodes

```

```

def plot(matrix):
    mat = copy.deepcopy(matrix)
    arr = []
    for i in range(len(mat)):
        for j in range(len(mat)):
            if mat[i, j] > 0:
                mat[j, i] = 0
                number = mat[i, j]
                for m in range(number):
                    arr.append((f(i), f(j))) # 默认 0 索引, 现在加上

    # print(array)
    G = nx.MultiGraph(arr)
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, node_size= 550, node_color='r', alpha=1)
    ax = plt.gca()
    for e in G.edges:
        ax.annotate("",
                    xy=pos[e[0]], xycoords='data',
                    xytext=pos[e[1]], textcoords='data',
                    arrowprops=dict(arrowstyle="-", color="0.5",
                                    shrinkA=5, shrinkB=5,
                                    patchA=None, patchB=None,
                                    connectionstyle="arc3,rad=rrr".replace('rrr',
str(0.05 * e[2])), ), )
        nx.draw_networkx_labels(G, pos, labels=None, font_size=20, font_color='k',
font_family='sans-serif',
                                font_weight='normal', alpha=1.0, bbox=None, ax=None)

    # 保存为透明图像
    plt.savefig("图片.png", transparent=True)
    plt.show()

Matrix = []
pair_ = []
D_ = []
countMin = float('inf')
for i in range(100):
    if (i % 30 == 0):
        print('第', str(i), '局')
        MatCopy = copy.deepcopy(G_mat) # 副本
        DCopy = copy.deepcopy(D) # 副本
        D_no, pair_array = karger_Min_Cut(MatCopy, DCopy) # x 返回向量 D 和合并过的节点对
        count = np.max(D_no)

```

```

        if count < countMin:
            countMin = count
            Matrix.append(MatCopy)
            pair_.append(pair_array)
            D_.append(D_no)
print("karger_min_cut is " + str(countMin))

last = Matrix[-1]
degree = D_[-1]
node_set = ([i for i, x in enumerate(degree) if x != 0])
print(node_set)

last_pair = pair_[-1]
A = []
B = []
A.append(node_set[0])    # 点集 1
B.append(node_set[1])    # 点集 2
u_list = []
v_list = []
for m in range(len(last_pair)):
    each = last_pair[m]
    u = each[0]
    v = each[1]
    u_list.append(u)
    v_list.append(v)

# 输出点集
set1 = ([i for i, x in enumerate(u_list) if x == node_set[0]])
setA = find_set(set1, u_list, v_list, A)
print("点集 1: ", setA)

set2 = ([i for i, x in enumerate(u_list) if x == node_set[1]])
setB = find_set(set2, u_list, v_list, B)
print("点集 2: ", setB)

# 作图
plot(last)

```

附录 B

```

import numpy as np
import networkx as nx

```

```

import matplotlib.pyplot as plt

# Stoer-Wagner 算法
def GlobalMinCut(G_mat):
    # 记录每个结点
    D = np.arange(len(G_mat))
    # 初始化 mincut 为一个大数
    mincut = 10000
    E_ = []

    while len(G_mat) > 2:
        s,t,mc = MinCut(G_mat)
        E_.append([D[s],D[t]])

        if mc < mincut:
            mincut = mc

        # 合并 s,t
        G_mat = contract(G_mat,s,t)

        print(D[s],D[t],mincut)

        # 删除 t 点编号, 把 s 点编号换到最后面
        if s<t:
            D = np.delete(D,t)
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
        else:
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
            D = np.delete(D,[t])

        # 输出剩余的点的标号
        # print(D)

    return mincut,D,E_

# 求解任意 st 最小割的函数
def MinCut(G_mat):
    A = []

```

```

a = np.random.randint(0,len(G_mat)-1)
A.append(a)

# 构造点集 V-A 记为 V_no_
V_no_ = list(np.arange(len(G_mat)))

while(len(A)<len(G_mat)):

    V_no_.remove(A[-1])
    # print(V_no_)
    # print(A)

    A.append(V_no_[np.argmax(np.sum(G_mat[A][:,V_no_],axis=0))])

s = A[-2]
t = A[-1]
mincut = np.sum(G_mat[A,t])-G_mat[A[-1],t]

return s,t,mincut

def contract(G_mat,s,t):
    new_mat = np.zeros((len(G_mat)+1,len(G_mat)+1))
    new_mat[:len(G_mat),:len(G_mat)] = G_mat
    for i in range(len(G_mat)):
        if i != s and i != t:
            new_mat[i,-1] = new_mat[i,s] + new_mat[i,t]
            new_mat[-1,i] = new_mat[s,i] + new_mat[t,i]
    new_mat = np.delete(new_mat, (s,t), axis = 0)
    new_mat = np.delete(new_mat, (s,t), axis = 1)
    return new_mat

filename = "BenchmarkNetwork"
# filename = "Corruption_Gcc"
# filename = "Crime_Gcc"
filename = "PPI_gcc"
# filename = "RodeEU_gcc"

E = np.loadtxt("data/"+filename+".txt")
G = nx.Graph()
G.add_edges_from(E)
# 得到图的邻接矩阵
A = nx.adjacency_matrix(G)

```

```

G_mat = A.todense()
# 输出这个邻接矩阵
# print('邻接矩阵:\n',G_mat)
# 画出这张图
# nx.draw(G)

print("The number of nodes is:",len(G_mat))
mincut,D,E_ = GlobalMinCut(G_mat)

L = list(np.loadtxt("data/"+filename+".txt", dtype=np.int).flatten())
Array = list(set(L))
Array.sort(key=L.index)
# 点标号的映射
def f(x):
    value = Array[x]
    return value

# 映射到原文件中的点
D[0] = f(D[0])
D[1] = f(D[1])
array = []
for i in range(int(mincut)):
    array.append(D)

print("\nThe global minimum cut is:",D,mincut,"(在原文件中的标号)")

# 作图展示测试结果
G=nx.MultiGraph(array)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color = 'r', alpha = 1)
ax = plt.gca()
for e in G.edges:
    ax.annotate("",
                xy=pos[e[0]], xycoords='data',
                xytext=pos[e[1]], textcoords='data',
                arrowprops=dict(arrowstyle="-", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,
                                connectionstyle="arc3,rad=rrr".replace('rrr',str(0.3*e[2]))
                                ),

```



```

    ),
    )
nx.draw_networkx_labels(G, pos, labels=None, font_size=12, font_color='k',
font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None)
# 保存为透明图像
plt.savefig("Advanced"+filename, transparent=True)
plt.show()

# 通过 E_寻找两个超结点所含结点
G = nx.Graph()
G.add_edges_from(E_)
Set1 = set()
Set2 = set()
components=list(nx.connected_components(G))
for i in components[0]:
    Set1.add(f(i))

if len(components) > 1:
    for i in components[1]:
        Set2.add(f(i))

print(Set1,"\n",Set2)

```

附录 C

```

import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

# Stoer-Wagner 算法
def GlobalMinCut(G_mat):
    # 记录每个结点
    D = np.arange(len(G_mat))
    # 初始化 mincut 为一个大数
    mincut = 10000
    E_ = []

    while len(G_mat) > 2:
        s,t,mc = MinCut(G_mat)

        if mc < mincut:

```

```

        mincut = mc

        # 当发现最小割为 1 时直接退出
        if mincut == 1:
            E_.append([D[t],D[t]]) # 相当于此时不要把点添加进去，同时保证 E_ 中至少
            有一条边
            D = [D[s],D[t]]
            break

        E_.append([D[s],D[t]])
        # 合并 s,t
        G_mat = contract(G_mat,s,t)

        print(D[s],D[t],mincut)

        # 删除 t 点编号，把 s 点编号换到最后面
        if s<t:
            D = np.delete(D,t)
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
        else:
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
            D = np.delete(D,[t])

        # 输出剩余的点的标号
        print(D)

    return mincut,D,E_

# 求解任意 st 最小割的函数
def MinCut(G_mat):
    A = []
    a = np.random.randint(0,len(G_mat)-1)
    A.append(a)

    # 构造点集 V-A 记为 V_no_
    V_no_ = list(np.arange(len(G_mat)))

    while(len(A)<len(G_mat)):

```

```

        V_no_.remove(A[-1])
        # print(V_no_)
        # print(A)

        A.append(V_no_[np.argmax(np.sum(G_mat[A][:,V_no_],axis=0))])

    s = A[-2]
    t = A[-1]
    mincut = np.sum(G_mat[A,t])-G_mat[A[-1],t]

    return s,t,mincut

def contract(G_mat,s,t):
    new_mat = np.zeros((len(G_mat)+1,len(G_mat)+1))
    new_mat[:len(G_mat),:len(G_mat)] = G_mat
    for i in range(len(G_mat)):
        if i != s and i != t:
            new_mat[i,-1] = new_mat[i,s] + new_mat[i,t]
            new_mat[-1,i] = new_mat[s,i] + new_mat[t,i]
    new_mat = np.delete(new_mat, (s,t), axis = 0)
    new_mat = np.delete(new_mat, (s,t), axis = 1)
    return new_mat

filename = "BenchmarkNetwork"
filename = "Corruption_Gcc"
# filename = "Crime_Gcc"
# filename = "PPI_gcc"
# filename = "RodeEU_gcc"

E = np.loadtxt("data/"+filename+".txt")
G = nx.Graph()
G.add_edges_from(E)
# 得到图的邻接矩阵
A = nx.adjacency_matrix(G)
G_mat = A.todense()
# 输出这个邻接矩阵
# print('邻接矩阵:\n',G_mat)
# 画出这张图
# nx.draw(G)

print("The number of nodes is:",len(G_mat))

```

```

mincut,D,E_ = GlobalMinCut(G_mat)

L = list(np.loadtxt("data/"+filename+".txt", dtype=np.int).flatten())
Array = list(set(L))
Array.sort(key=L.index)
# 点标号的映射
def f(x):
    value = Array[x]
    return value

# 映射到原文件中的点
D[0] = f(D[0])
D[1] = f(D[1])
array = []
for i in range(int(mincut)):
    array.append(D)

print("\nThe global minimum cut is: ",D,mincut,"(在原文件中的标号)")

# 作图展示测试结果
G=nx.MultiGraph(array)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color = 'r', alpha = 1)
ax = plt.gca()
for e in G.edges:
    ax.annotate("",
                xy=pos[e[0]], xycoords='data',
                xytext=pos[e[1]], textcoords='data',
                arrowprops=dict(arrowstyle="-", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,

connectionstyle="arc3,rad=rrr".replace('rrr',str(0.3*e[2])
                                ),
                                ),
                                )

nx.draw_networkx_labels(G, pos, labels=None, font_size=12, font_color='k',
font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None)
# 保存为透明图像
plt.savefig("Advanced"+filename, transparent=True)
plt.show()

# 通过 E_寻找两个超结点所含结点

```

```
G = nx.Graph()
G.add_edges_from(E_)
Set1 = set()
Set2 = set()
components=list(nx.connected_components(G))
for i in components[0]:
    Set1.add(f(i))

if len(components) > 1:
    for i in components[1]:
        Set2.add(f(i))

print(Set1,Set2)
```

附录 D

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from random import choice
import copy

filename = "BenchmarkNetwork"
# filename = "Corruption_Gcc"
# filename = "Crime_Gcc"
# filename = "PPI_gcc"
# filename = "RodeEU_gcc"

E = np.loadtxt("data/"+filename+".txt")
G = nx.Graph()
G.add_edges_from(E)
# 得到图的邻接矩阵
A = nx.adjacency_matrix(G)
G_mat = A.todense()
# 输出这个邻接矩阵
# print('邻接矩阵:\n',G_mat)
# 画出这张图
# nx.draw(G)

# 得到每个点的度数
D = []
```

```

for i in range(0, len(G_mat)):
    each = G_mat[i][0]
    each[each > 0] = 1
    degree = each.sum()
    D.append(degree)
D = np.array(D)

L = list(np.loadtxt("data/"+filename+".txt", dtype=np.int).flatten())
Array = list(set(L))
Array.sort(key=L.index)
# 点标号的映射
def f(x):
    value = Array[x]
    return value

def karger_Min_Cut(graph, D):
    pair = []
    while np.sum(D > 0) > 20:      # 设定 Karger 的阈值
        # 随机选一个顶点
        u_beixuan = np.array(np.where(D > 0))[0]
        u = choice(u_beixuan)
        u_no = np.where(graph[u] > 0)[1]
        v = choice(u_no)      # 选出一条边
        pair.append((u,v))
        contract_K(graph, u, v, D)
    return D, pair

def contract_K(graph, u, v, D):
    # 更新点的度数
    D[u] = D[u] + D[v] - 2*graph[u, v]
    D[v] = 0
    # 删除 uv 相连的边 (自环)
    graph[u, v] = 0
    graph[v, u] = 0
    v_ = np.where(graph[v] > 0)[1]  # 与 v 相连的顶点
    # 更新与 v 有边的点的列表
    for vertex in v_:
        if vertex != u and vertex != v:
            graph[vertex, u] = graph[vertex, u] + graph[vertex, v]
            graph[u, vertex] = graph[u, vertex] + graph[v, vertex]
            graph[vertex, v] = 0
            graph[v, vertex] = 0

```

```

# 规约找点: 输入索引, 返回索引
def find_nodes(array, u_list, v_list, nodes):
    C = []
    D = []
    for each in array:
        C.append(v_list[each])
    nodes.extend(C)
    for every in C:
        D.extend([i for i, x in enumerate(u_list) if x == every])
    if len(D) == 0:      # 空数组
        return 0
    return D

# Stoer-Wagner 算法
def GlobalMinCut(G_mat):
    # 记录每个结点
    D = np.arange(len(G_mat))
    # 初始化 mincut 为一个大数
    mincut = 10000
    E_ = []

    while len(G_mat) > 2:
        s, t, mc = MinCut(G_mat)

        if mc < mincut:
            mincut = mc

        # 当发现最小割为 1 时直接退出
        if mincut == 1:
            E_.append([D[t], D[t]]) # 相当于此时不要把点添加进去, 同时保证 E_ 中至少
            有一条边
            D = [D[s], D[t]]
            break

        E_.append([D[s], D[t]])
        # 合并 s, t
        G_mat = contract_SW(G_mat, s, t)

    print(D[s], D[t], mincut)

    # 删除 t 点编号, 把 s 点编号换到最后面

```

```

        if s<t:
            D = np.delete(D,t)
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
        else:
            zjl = D[-1]
            D[-1] = D[s]
            D[s] = zjl
            D = np.delete(D,[t])

        # 输出剩余的点的标号
        print(D)

    return mincut,D,E_

# 求解任意 st 最小割的函数
def MinCut(G_mat):
    A = []
    a = np.random.randint(0,len(G_mat)-1)
    A.append(a)

    # 构造点集 V-A 记为 V_no_
    V_no_ = list(np.arange(len(G_mat)))

    while(len(A)<len(G_mat)):

        V_no_.remove(A[-1])
        # print(V_no_)
        # print(A)

        A.append(V_no_[np.argmax(np.sum(G_mat[A][:,V_no_],axis=0))])

    s = A[-2]
    t = A[-1]
    mincut = np.sum(G_mat[A,t])-G_mat[A[-1],t]

    return s,t,mincut

def contract_SW(G_mat,s,t):
    new_mat = np.zeros((len(G_mat)+1,len(G_mat)+1))
    new_mat[:len(G_mat),:len(G_mat)] = G_mat

```



```

    for i in range(len(G_mat)):
        if i != s and i != t:
            new_mat[i,-1] = new_mat[i,s] + new_mat[i,t]
            new_mat[-1,i] = new_mat[s,i] + new_mat[t,i]
    new_mat = np.delete(new_mat, (s,t), axis = 0)
    new_mat = np.delete(new_mat, (s,t), axis = 1)
    return new_mat

Matrix = []
pair_ = []
D_ = []
# print(karger_Min_Cut(G_mat))
countMin = float('inf')
for i in range(100):
    if (i % 30 == 0):
        print('第', str(i), '局')
    MatCopy = copy.deepcopy(G_mat)    # 副本
    DCopy = copy.deepcopy(D)          # 副本
    count, pair_array = karger_Min_Cut(MatCopy, DCopy)    # 节点对
    count_number = np.max(count)
    if count_number < countMin:
        countMin = count_number
        Matrix.append(MatCopy)
        pair_.append(pair_array)
        D_.append(count)
print("karger_min_cut is " + str(countMin))

last = Matrix[-1]
degree = D_[-1]
node_set = ([i for i, x in enumerate(degree) if x != 0])
print(node_set)

last_pair = pair_[-1]
A = []
B = []
A.append(node_set[0])    # 点集 1
B.append(node_set[1])    # 点集 2
u_list = []
v_list = []
for m in range(len(last_pair)):
    each = last_pair[m]
    u = each[0]
    v = each[1]

```

```

u_list.append(u)
v_list.append(v)

# 点集 1
set1 = ([i for i, x in enumerate(u_list) if x == node_set[0]])
result1 = find_nodes(set1, u_list, v_list, A)
while result1 != 0:
    result1 = find_nodes(result1, u_list, v_list, A)
A = [f(x) for x in A]    # 还原标号
print("点集 1: ", A)

# 点集 2
set2 = ([i for i, x in enumerate(u_list) if x == node_set[1]])
result2 = find_nodes(set2, u_list, v_list, B)
while result2 != 0:
    result2 = find_nodes(result2, u_list, v_list, B)
B = [f(x) for x in B]    # 还原标号
print("点集 2: ", B)

# 作图
last = Matrix[-1]
arr = []
for i in range(len(last)):
    for j in range(len(last)):
        if last[i, j] > 0:
            last[j, i] = 0
            number = last[i, j]
            for m in range(number):
                arr.append([f(i), f(j)])
# print(arr)

G=nx.MultiGraph(arr)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color = 'r', alpha = 1)
ax = plt.gca()
for e in G.edges:
    ax.annotate("",
                xy=pos[e[0]], xycoords='data',
                xytext=pos[e[1]], textcoords='data',
                arrowprops=dict(arrowstyle="-", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,

```

```

connectionstyle="arc3,rad=rrr".replace('rrr',str(0.3*e[2])),),)
nx.draw_networkx_labels(G, pos, labels=None, font_size=12, font_color='k',
font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None)
plt.show()

# 结点数降至阈值以下改用 SW 算法
G = nx.MultiGraph()
G.add_edges_from(arr)
# 得到图的邻接矩阵
A = nx.adjacency_matrix(G)
G_mat = A.todense()
mincut,D,E_ = GlobalMinCut(G_mat)

L = list(np.array(arr).flatten())
Array = list(set(L))
Array.sort(key=L.index)
# 点标号的映射
def F(x):
    value = Array[x]
    return value

# 映射到在原文件中的编号
D[0] = F(D[0])
D[1] = F(D[1])

array = []
for i in range(int(mincut)):
    array.append(D)

print("\nThe global minimum cut is:",D,mincut,"(在原文件中的标号)")

# 作图展示测试结果
G=nx.MultiGraph(array)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color = 'r', alpha = 1)
ax = plt.gca()
for e in G.edges:
    ax.annotate("",
                xy=pos[e[0]], xycoords='data',
                xytext=pos[e[1]], textcoords='data',
                arrowprops=dict(arrowstyle="-", color="0.5",
                                shrinkA=5, shrinkB=5,

```

```

                                patchA=None, patchB=None,

connectionstyle="arc3,rad=rrr".replace('rrr',str(0.3*e[2])
                                ),
                                ),
                                )
nx.draw_networkx_labels(G, pos, labels=None, font_size=12, font_color='k',
font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None)
# 保存为透明图像
plt.savefig("Advanced"+filename, transparent=True)
plt.show()

# 通过 E_寻找两个超结点所含结点
G = nx.Graph()
G.add_edges_from(E_)
Set1 = set()
Set2 = set()
components=list(nx.connected_components(G))
for i in components[0]:
    Set1.add(F(i))

if len(components) > 1:
    for i in components[1]:
        Set2.add(F(i))

print(Set1,"\n",Set2)

```