基于 Karger 算法和 Stoer-Wagner 算法的最小割问题研究

摘要

最小割问题是网络问题中的研究热点，对于不同条件、不同定义下的最小割，前人已经有了丰富的成果。按照研究思路的不同，我们将学术界现有的最小割算法分为两类：基于最大流-最小割定理和基于合并点定理，并以后者作为全文的研究中心。本文从割的定义出发，对不同条件下的最小割进行了系统的分类，并给出相应的数学定义；在此基础上，讨论并证明了不同定义下最小割的 P/NP 问题，为下文研究最小割的多项式算法奠定了基础。

本文的研究对象是最小割问题中的最小 2 割问题，重点研究了其中的 Karger 算法和 Stoer-Wagner 算法，二者均基于合并点定理，且分别代表合并点的两类主流算法：确定性算法和随机算法。我们的主要工作是：首先使用 python 对上述算法进行了复现，进而对后继出现的各种改进算法进行对比与总结，进一步地分析了日后可能的改进方向，最后分别对 Karger 算法和 Stoer-Wagner 算法提出一种改进方案。

关键词： 最小割　Karger 算法　Stoer-Wagner 算法

Abstract

The minimum cut problem is a research hotspot in network problems, and there have been fruitful results for minimum cuts under different conditions and definitions in the literature. Based on the current study, we divide the existing minimal cut algorithms into two categories: one is based on the maximum flow-minimal cut theorem and the other is based on edge contraction. We focus on the latter one in this paper. From the beginning, the definition of cut is introduced. We then systematically categorize the minimal cut according to different conditions and provide the corresponding mathematical definition, based on which we discuss and prove the P/NP problem of minimal cut under various definitions. All of these lay the foundation for the following study of polynomial algorithms of minimal cut.

The focus of this paper is the minimum 2-cut problem, and we pay our attention to the Karger algorithm and the Stoer-Wagner algorithm, both of which are based on the edge contraction and represent two types of algorithms: deterministic algorithms and stochastic algorithms, respectively. First, we reproduce the above algorithms with python. Then we compare and summarize some subsequently improved algorithms, after which we analyze the possible improvement directions. Finally, we propose an improvement scheme for the Karger algorithm and Stoer-Wagner algorithm respectively.

Keywords: munimum cut　Karger's algorithm　Stoer-Wagner's algorithm

1 研究综述

图的连通性是图论中的经典问题之一。其中，最小割问题在电路设计、通信网络、交通运输等方面具有众多实际应用。

寻找无向图的最小割是一个基本的算法问题，即：如何找到图的一个割，将该图划分为两部分，且割的权重取得最小。目前，学术界已有多种算法解决此问题，而这些算法按照解决思路的不同又可以分为**基于最大流最小割定理**和**基于合并点**两类。

Ford 和 Fulkerson 提出的最大流最小割定理 [1]，证明了对于任意的源点 s 和汇点 t，$s-t$ 最大流和 $s-t$ 最小割问题是对偶问题。第一类算法便基于上述定理，将最小割问题与最大流问题密切联系起来。对于给定的源点 s，Gomory 和 Hu 提出了一种算法，用于计算 $|V|-1$ 个 $s-t$ 最大流，得到 s 的最大流后用对偶定理，得到以 s 为源点的最小割，遍历所有的 s，从而找到全局最小割。在这之后，Hao 和 Orlin [2] 提出使用 Goldberg 和 Tarjan [3] 的最大流算法来解决最小割问题，并将时间复杂度降至 $\mathcal{O}(|V||E|log(|V|^2/|E|)$。

第二类算法基于合并点的思想建立。这类算法最早由 Nagamochi 和 Ibaraki [4] 提出，他们构建了一种生成树，将图的边集 E 分解为互不相交的子集 $E_1,\ldots,E_\lambda$，并对较高权重的边的端点进行合并，最终在具有非负边权的无向图中将时间复杂度降至 $O(|V||E|+|V|^2log|V|)$。Stoer-Wagner 算法 [5] 是 Stoer 和 Wagner 对 Nagamochi-Ibaraki 算法的改进，通过构造点的最大邻接顺序（$MA\text{-}order$）代替原算法中构造的生成树。在所有合并点的算法中，Karger 算法 [6] 是一类比较独特的算法，由 Karger 和 Stein 于 1993 年提出。与其他确定性算法不同，该算法使用随机搜索的方法，它并不保证找到全局最小割，而是以较高的概率找到全局最小割，其时间复杂度为 $\mathcal{O}(|V|^2log^3V)$

本文采取的研究思路属于第二种：基于点的合并。首先从割的基本定义出发，证明了最小割问题与 P/NP 问题之间的关系；进而对“合并点”算法中的 Karger 算法和 Stoer-Wagner 算法进行了研究和复现，并使用真实数据对上述两个算法进行实际验证和对比；最后在前人改进的基础上提出了我们对于算法的改进。

2 符号与定义

2.1 割、割集与 s-t 割

通常所讲的割是 2 割，用 C 表示（如图1），其定义如下：

定义 2.1. 割 $C=(S,T)$ 是图 $G=(V,E)$ 中点 V 的一种分割，C 将 G 的点集 V 分成两个集合 S 和 T。

定义 2.2. *s-t* 割 如果 s 和 t 是图 G 的两个指定的顶点，那么 ***s-t* 割**表示 $s\in S$ 且 $t\in T$ 的 *2* 割。

s-t 割的一个例子如图2。

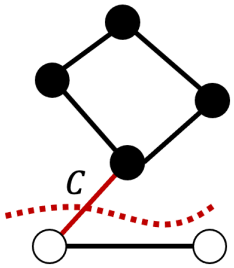

图 1: 割

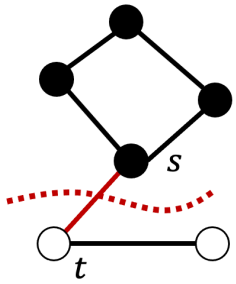

图 2: s-t 割

将 2 割进行推广便得到 k 割，其定义如下：

定义 2.3. ***k* 割** $C = (S_1, S_2, ...S_k)$ 是图 $G = (V, E)$ 中点 V 的一种分割，C 将 G 分割成至少 k 个不连通的分量。

定义 2.4. 割集 对于 *2* 割而言，某个割 $C = (S, T)$ 的**割集**是边的集合：$\{(u, v) \in E \mid u \in S, v \in T\}$，其中一个端点 u 在 S 中，另一个端点 v 在 T 中。对于 k 割而言，某个割 $C = (S_1, S_2, ...S_k)$ 的**割集**是边的集合 $\{(s_i, s_j) \in E \mid s_i \in S_i, s_j \in S_j, i \neq j\}$

2.2 割的权重

定义 2.5. 在有权图 G 中，设边 e_i 的权重为 w_i，则

2 **割的权重**定义为

$$c := \sum_{e_i \in C} w_i \tag{1}$$

k **割的权重**定义为：

$$c := \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{\substack{v_1 \in C_i \\ v_2 \in C_j}} w\left(\{v_1, v_2\}\right) \tag{2}$$

若 G 为无权图，则令 $w_i = 1$ 可同样定义割的权重。

显然，在无权无向图中，割的权重是与割相交的边的数量；在加权图中，权重是与割相交的边的权重之和。

我们指出，带边权的最小割问题允许有负权边，可通过对所有边权取相反数转化为最大流问题进行求解。

2.3 最小割与全局最小割

按照割的数量来划分，最小割分为最小 2 割与最小 k 割；其中最小 2 割按照有无指定的源点和终点又可以分为两类。只有当源点和终点没有指定时，讨论全局最小割才是有意义的，否则最小割的结果唯一。

定义 2.6. 给定图 $G = (V, E)$，对于 $\forall S, T \subset V$，使得式 *(1)* 最小的 $C(S, T)$ 称为**最小 *2* 割**；

定义 2.7. 给定图 $G = (V, E)$ 以及源点 $s \in S$ 和汇点 $t \in T$，使得式 *(1)* 最小的 $C(S, T)$ 称为**最小 *s*-*t* 割**；

定义 2.8. 给定图 $G=(V,E)$ 以及整数 k，使得式 *(2)* 最小的 $C=(S_1,S_2,...S_k)$ 称为**最小 k 割**；

定义 2.9. 图 $G=(V,E)$ 的所有 *2* 割 $C(S,T)$ 中割的权重最小者称为**全局最小 *2* 割**；图 $G=(V,E)$ 的所有 k 割 $C=(S_1,S_2,...S_k)$ 中割的权重最小者称为**全局最小 k 割**。统称为**全局最小割**。

为叙述上的方便，下文所指的全局最小割均为全局最小 2 割。一个全局最小 2 割的例子如图3。

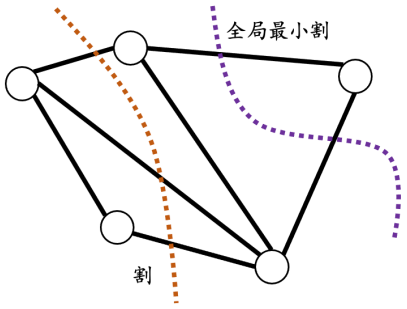

图 3: 割与全局最小割

2.4 图的收缩

定义 2.10. 图 $G/u\sim v$ 由图 G 将点 u 和 v 合并形成，即点 u 和点 v 由一个新的点 $[u]=[v]$ 取代，且 $w(x,[u])=w(x,u)+w(x,v)$，即任意一点到该点的权重由原来的权重叠加而成。

2.5 最大邻接顺序

定义 2.11. 设 $v_1,v_2,...v_n$ 为图 G 中的点。若对 $\forall k\geq i$，都有：

$$w\left(v_1,\ldots,v_{i-1};v_i\right):=\sum_{j=1}^{i-1}w\left(v_i,v_j\right)\geq\sum_{j=1}^{i-1}w\left(v_k,v_j\right)=:w\left(v_1,\ldots,v_{i-1};v_k\right)\tag{3}$$

则称 $v_1,v_2,...v_n$ 为**最大邻接顺序**（*maximum adjacency order*）或 ***MA-order***。

上述定义的 MA-order 是包括 Stoer-Wagner 算法在内的合并点算法的基础，其含义是：对 MA-order 中的每一个点 v_i，它到先前点 $v_1,v_2,...v_{i-1}$ 的权重之和是所有 v_i 之后的点中最大的。

3 P/NP 问题

3.1 定义

定义 3.1. 若问题 X 可以由一个确定型图灵机在多项式表达的时间内解决，则称 X 为 ***P*** 问题；

定义 3.2. 若对于问题 X，任意给定输入 s，可以在多项式时间验证 s 是否为问题 X 的解，则称 X 为 ***NP*** 问题；

定义 3.3. 对于 $X \in NP$, 若对任意 $Y \in NP$, $Y \leq_p X$, 则称 X 为 ***NP*-完全**(*NP-Completeness*, 简称 *NPC*) 问题。

即 NPC 是一个 NP 问题，且所有的 NP 问题在多项式时间内都能约化到 NPC 问题，如果可以解决这个 NPC 问题，那么所有 NP 问题也都可以得到解决。

定义 3.4. 对于问题 X，若对任意 $Y \in NP$, $Y \leq_p X$，则称 X 为 ***NP* 难** (*NP-Hard*) 问题。

从定义可以看出，NP 难问题不一定是 NP 问题。

3.2 四者之间的关系

按照大多数研究者认为 $P \neq NP$ 的假设，可作出关系图4:

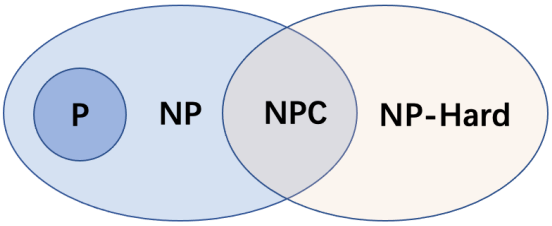

图 4: P/NP 问题之间的关系

即：NPC 必为 NP 问题，NP 难可能是 NP 问题，也可能不是 NP 问题。

3.3 最小割的 P/NP 问题

对于最小 k 割的问题:

(1) **k=2** 时，无向图的最小割问题可以转化为 s-t 最大流问题，而最大流问题是有多项式算法（如 Ford–Fulkerson 算法、Push–relabel 最大流算法、Compact networks 算法等）可解的，因此是 **P 问题**，如下给出了两个算法的算法步骤。**进一步**，考虑在有向图中添加析取约束研究上述最大流问题：当图中添加负析取约束时，问题是 **NP 难**的；当图中添加正析取约束时，在整数条件约束下问题是 **NP 难**的，否则是 **P 问题**，可以在多项式时间内解决，已经由 Ahuja RK, Magnanti TL 给出了多项式算法 [7]。

① Push–relabel 最大流算法

算法步骤：给定一张边的容量为 c 的图 $G = (V, E)$，源点 s 以及汇点 t，求从 s 到 t 的最大流 f。

- 初始化：将与源点 s 相连的管道流量 $f(0, i)$ 设为该管道的容量，即 $f(0, i) = c(0, i)$；将源点 s 的高度 $h(0)$ 设为图的顶点个数，所有其他节点的标签均为零。
- 搜索是否有节点的点余量 $e(u) > 0$，如果存在，则进行重标记或者压入流的操作:
  - 检查与该点相邻的所有点 v，若 $h(u) > h(v)$，将该点的余量以最大方式压入，然后对节点余量 e、边 (u, v) 的容量进行相应的进行减加操作；

– 如果找不到高度较低的相邻节点, 则对该节点的高度增加 1。以上操作循环进行, 直至该点的余量为 0。

- 重复第 2 步，至找不到余量大于 0 的节点，停止算法。最后输出的终点的余量 $e(t)$, 即为所求的最大流 (最小割)。

算法复杂度 $\mathcal{O}(n^2m)$

② Compact networks 算法

算法步骤：给定一张图 $G=(V,E)$，令 $n=|V|$，$m=|E|$。

- 定义 $\Delta := r(S,T)$;
- 令 c 表示 Δ 的节点数:
  - 如果 $c > m^{9/16}$，那么在剩余网络 $G(r)$ 上寻找一个最优流 $\Delta/(4m)$;
  - 如果 $m^{1/3} \le c < m^{9/16}$ 那么:
    * 令 G' 表示 Δ-*compact network*;
    * 在 G' 上寻找一个最优流 $x'(\Delta/(8m))$;
    * 将流 x' 转化为 $G[r]$ 上的最优流 $x^*(\Delta/(4m))$。
  - 如果 $c < m^{1/3}$，那么:
    * 选择一个最小值 Γ, 满足:在 $(\Delta,\Gamma^{'})$-*compact network* 中的节点数少于 $2m^{1/3}$;
    * 令 G' 表示 $(\Delta,\Gamma^{'})$-*compact network*;
    * 在 G' 上寻找最优流 x';
    * 将流 x' 转化为 $G[r]$ 上的 *Gamma*-最优流。

算法复杂度：$\mathcal{O}(mn)$

下面证明**在有向图中添加正约束时，整数条件下问题是 NP 难的**。

证明. 在下面的证明过程中，我们将添加正析取约束的图叫做**强制图**（*forcing graph*），其最大流问题称为 **MFFG**（*maximum flow problem with forcing graph*）。

记 $G=(N,A)$ 是由 n 个顶点和 m 条边组成的有向连通图, 假设每个顶点 $v \in N\backslash\{s,t\}$ 的入度和出度都大于 0，源点为 s，终点为 t。目标是使得从 s 发送到 t 的总流量最大，不超过边上的容量，且在每个顶点保持流量平衡。记最大流量值为 f_{MF}:

$$\begin{aligned} \max \quad & f \\ s.t. \quad & \textstyle\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0 \quad ,\forall i \in N\backslash\{s,t\} \\ & \textstyle\sum_{j:(s,j)\in A} x_{sj} - \sum_{j:(j,s)\in A} x_{js} = f \\ & \textstyle\sum_{j:(t,j)\in A} x_{tj} - \sum_{j:(j,t)\in A} x_{jt} = -f \\ & 0 \le x_{ij} \le u_{ij} \end{aligned} \tag{4}$$

其中，$MFFG$ 向 MF 添加由强制图 $H=(A,E)$ 产生的整数约束:

$$(MFGG) \qquad (a,\bar{a}) \in E \to (x_a + x_{\bar{a}} \ge 1) \tag{5}$$

为了便于证明，引入以下定义:

定义 3.5. ***2-ladder* 图**是一个路径长度为 *1* 的无向图，即连接成对顶点构成的孤立边。

为了证明该问题是 NP 难问题，首先**顶点覆盖问题**（已知的 NPC 问题）可以规约到该问题。

给定一个无向图 $\Gamma(V,E)$，顶点覆盖问题要求：找到子集 $V'\subseteq V$，满足图 Γ 中的每一条边都可以映射到 V' 中的至少一个顶点，V' 的势最多为 K，子集 V' 称为顶点覆盖（*vertex cover*，简称 VC）。

令 $N(j)\subseteq V$ 表示 Γ 中顶点 j 的邻域，即所有在 $V\backslash\{j\}$ 中的顶点均与 j 相邻。按以下方式构造图 G_{MFFG}（见图5）：引入三个特殊的点 s,s_h,t，记连接 s_h 和 t 的边为 e_{sh}。对每一个点 $j\in V$，都有一条长度为 $|N(j)|$ 的有向（从点 s_h 到 t）路径 P_j。显然，Γ 中的每一条边 (i,j) 都表明了图 G_{MFFG} 中的边 e_{ji} 在 P_j 中出现以及边 e_{ij} 在 P_i 中出现。

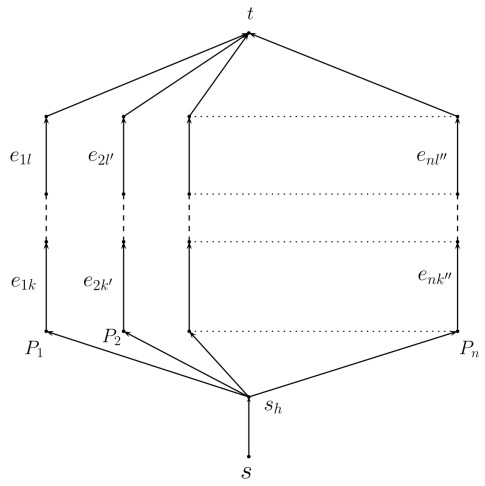

图 5: G_{MFFG}

现在定义一个 $2-ladder$ 强制图 G_{DIS}，其孤立边恰为由边 $(i,j)\in\Gamma$ 构成的 G_{MFFG} 的边 e_{ij} 和 e_{ji}。所有这些边（除 e_{sh} 以外）的容量均设为 1，e_{sh} 的容量取值于集合 $\{1,\ldots,n\}$。

令 I 表示由 G_{MFFG} 和 G_{DIS} 定义的 $MFFG$ 图。如果有向图 $G(V,E)$ 的源点 s 和终点 t 之间的路径不相交，那么在每条路径上发送尽可能多的流量就可以解决一般的最大流问题，并且满足所有正的不相交约束条件。从而得到以下定理：

定理 3.1. *I 的 $MFFG$ 问题是 NP 难的。*

证明. 只要证明以下等价成立：

$$\exists\ I \text{ 中一个流量为 } K \text{ 的流} \iff \exists\ VC,\ \text{图 } \Gamma \text{ 中有 } V, \text{势为 } K$$

充分性：将 e_{sh} 的容量设定为 K。对每一个顶点 $j\in V'$，在路径 P_j 上发送一个单位流量；对所有其他顶点，在 P_j 上不发送额外流量，在 e_{sh} 上发送 K 单位的流量。这样就给出了一条值为 $|V'|=K$ 的可行流。由于受到 e_{sh} 容量为 K 的限制，这条可行流是最大流。根据 G_{DIS} 的构造，Γ 中的每条边至少覆盖 V' 的一个顶点，故满足正约束条件。

必要性：首先设置 e_{sh} 的容量 K 等于 1，下面讨论 I 是否满足条件。如果不满足条件，将容量 K' 加 1 并反复求解 I 直到找到可行的解 f。记 K 为 f 的解，那么 K 对应 Γ 中顶点覆盖的大小。这是因为：如果 f 包括经过路径 P_j 的流，则将顶点 j 加入到 Γ 的顶点集

V' 中。由于所有的边容量均为 1，那么 K 条路径必然对流 f 做出了贡献，因此构建的顶点集具有势 K。因为 f 满足强制图 G_{DIS} 的条件，对于每条边，Γ 至少有一个入射顶点被添加到 V' 中。因此，得到的顶点集 V' 是一个顶点覆盖。 □

根据如上规约，得到下面的推论：

推论 3.1. *MFFG 不存在任何多项式时间逼近算法。*

根据定理3.1证明的第二部分，具有最小值 K 的可行流会产生最小的顶点覆盖。因此正整数约束下该问题是 NP 难的。

□

(2) $\mathbf{k \geq 3}$ 时：若 k 是固定的数值，则可以在多项式时间内用 Goldschmidt 和 Hochbaum 在 1988 年提出的算法解决，故为 **P 问题**；但是当指定的 k 个点必须分别在 k 个不连通分量中时，最小 k 割问题为 **NPC 问题**；同时，当 k 也作为问题的输入时，该问题也为 **NPC 问题**，下面对 $k = 3$ 给出证明 [8]：

首先说明该问题为 PT($Partition\ into\ Triangles$) 问题：

图 $G = (V, E)$，$w(v) \in Z^+$，$|V| = 3q$（q 为整数）。问：图 G 能否被分成 q 组不相交的点 $V_1, V_2, \ldots, V_n$，其中每个点集里面有 3 条边。

证明. 首先，在非确定算法中，只需要猜测图 G 点的 q 个不相交的三元组，并检查多项式时间，使每个顶点在 G 中可以产生一个三角形（即 3 割）。显然，这是 NP 问题；

下面证明任何 NP 问题都可以归约为 PT 问题。证明思路 [9] 是将三维匹配问题（3DM，已知的 NP-完全问题）向 PT 问题进行归约，从而表明 PT 问题是 NP-完全问题。

三维匹配问题 (3DM)：三个集合 X, Y, Z，找到一个集合 M，使得 $M \subseteq X \times Y \times Z$ 且 M 中的每一个三元组 $m = (x_a, y_b, w_c)$，在 X, Y, Z 中都有元素分别与它对应。

已知 3DM 是 NP 完全的 [10]。

构造如下：

令 $X = \{x_1, x_2, \ldots, x_q\}$,$Y = \{y_1, y_2, \ldots, y_q\}$,W= $\{w_1, w_2, \ldots, w_q\}$ 和 $M \subseteq X \times Y \times Z$ 是 3DM 的输入。构造顶点的初始集合 V（成为“公共顶点”），作为 X, Y, W 的并集。对于 M 中的每一个三元组 $m = (x_a, y_b, w_c)$，构造图 $G_m = (V_m, E_m)$, 如图6。

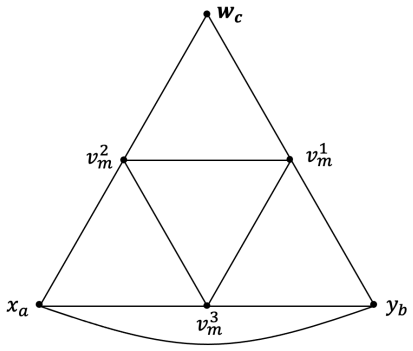

图 6: 3DM $\propto$ PT

这里，V_m 包含三个公共顶点 x_a, y_b, w_c 和三个 “私有顶点” v_m^1, v_m^2, v_m^3。E_m 的 10 条边是 $v_m^1v_m^2$,$v_m^1v_m^3$,$v_m^2v_m^3$,$x_av_m^3$,$x_av_m^2$,$y_bv_m^3$,$y_bv_m^1$, $w_cv_m^1$,$w_cv_m^2$,x_ay_b。整个图 G_M 从所有 G_m 中的并集取得：$G_M = (V_M, E_M) = (\bigcup_m V_m, \bigcup_m E_m)$。

性质 3.1. *根据构造方式可知，E_M 没有在 $X \bigcup W$ 或 $Y \bigcup W$ 的两端都有节点的边。*

性质 3.2. *图 G_M 是可分为 3 部分的。*

这是因为：首先考虑 $\{x_a, v_m^1\}, \{y_b, v_m^2\}, \{w_c, v_m^3\}$，易知，$G_m$ 是可分为 3 部分的。同样的，考虑三个集合 $X \bigcup \{v_m^1 : m \in M\}, Y \bigcup \{v_m^2 : m \in M\}$ 和 $W \bigcup \{v_m^3 : m \in M\}$，可以看到整个图 G_M 也是可以分为 3 部分的。

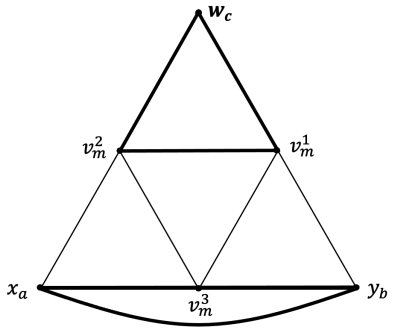

图 7: 完全态

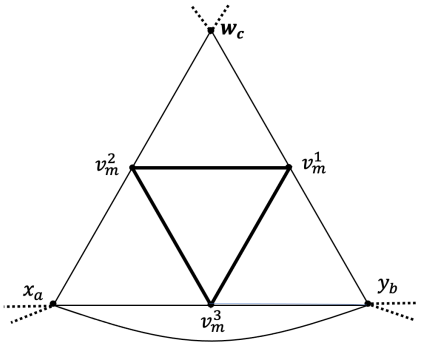

图 8: 共享态

每个 G_m 可以通过顶点 $x_ay_bv_m^3$ 和 $w_cv_m^1v_m^2$ 分成两个三角形，如图7，将其称为完全态。另外，也可以通过使三角形 $v_m^3v_m^1v_m^2$ 覆盖私有节点并使 G_M 三角形覆盖公共节点，来覆盖所有的 G_m 节点，将其称为共享态（如图8）。其中：

推论 3.2. *令 m 为 M 中的任何三元组，G_m 中可以覆盖 G_M 的且含有一个私有顶点的三角形是 $x_ay_bv_m^3$。*

证明. 只含有一个私有顶点意味着需要连接两个公共顶点，由性质3.1可知，x_ay_b 是 G_m 中唯一一条连接两个公共顶点的边。 □

引理 3.1. *如果图 G_M 划分为三角形，则每个 G_m 必须处于完全状态或共享状态。*

证明. 这是因为，考虑将 G_M 的分区 P 分为上述两种状态的小三角形和 G_m。假设 G_m 不是共享态的，即三个私有顶点不在 P 中的同一个三角形中，那么唯一的 G_m 恰好占据一个私有顶点的小三角形是 $x_ay_bv_m^3$。但是，由于其余的私有顶点只有被三角形 $w_cv_m^1v_m^2$ 覆盖才有可能性，所以若 $x_ay_bv_m^3$ 处于 P 中，则 G_m 必须处于完全态。 □

这样，可以使用完全态三角形和共享态三角形对 G_M 进行覆盖，且所有私有顶点和公共顶点都被获取一次，从而覆盖 X, Y, Z 中的所有元素。同时，因为 P 中三角形的顶点是不相交的，这说明不同小三角形之间不会共享顶点，因此 M 三元组互不相交。

综上所述：归约只需要在多项式时间内即可完成，具体时间将取决的 3DM 输入的大小。$k \geq 3$ 时为 NP 完全问题。 □

3.4 全局最小割的 P/NP 问题

对于带有边权的无向图，全局最小割问题可使用 Stoer-Wagner 算法在多项式时间内进行求解；在无权无向图中，只需将权重设为 1，或直接使用 Karger 算法；在带有边权的有向图中，可用最小割树算法在多项式时间内求解出答案。

综上所述，对于全局最小割问题，目前已有多项式算法（如 Stoer-Wagner 算法、最小割树法、Karger 算法等）可以进行求解，因此无论图是否有权重、是否有向，图的全局最小割问题都是 P 问题。但是当图的权重限制在整数上或具有负析取约束时，由于最小割本身即为 NP 难问题，故全局最小割也为 NP 难问题。

Stoer-Wagner 算法和 Karger 算法将会在下文中进行详细介绍，下面简要介绍**最小割树算法**。

最小割树：对于树上的所有边 (s,t) ，树上去掉 (s,t) 后产生的两个集合恰好是原图上 (s,t) 的最小割把原图分成的两个集合，且边 (u,v) 的权值等于原图上 (u,v) 的最小割。

算法步骤：通过递归实现。

- 在当前点集中随意选取两个点 u,v，计算出两点之间的最小割；
- 在树上连接出一条从 u 到 v, 权值为 $\lambda(u,v)\lambda(u,v)$ 的边；
- 找出 u,v 分属的两个点集，对这两个点集进行递归；
- 当点集中只剩一个点的时候停止递归。

算法复杂度：$\mathcal{O}(mn^3)$

4 Karger 算法

Karger 算法是一类基于无向无权图的合并点的算法，由 David Karger 在 1993 年首次提出。它并不保证能够找到全局最小割，而是以极大的概率能够找到全局最小割。

4.1 基本思想

从整体上看，Karger 算法基于这样的思想：图的全局最小割包含的边一般只占原图的很小一部分，因此任意选取一条边恰好是全局最小割的边的概率很小。所以即使我们选取了一条边并合并了边的两个端点，很大概率上这两个点在割的同一侧，这样全局最小割是不受影响的。因此可以不断地迭代上述过程，直到该图只剩下两个点。这两点之间的权重有一定的概率是全局最小割，因此迭代一定次数后，能够以较大的概率找到全局最小割。

将上述思想具体到定理与算法只需要两方面的工作：首先是给出合并点的方法，下文4.2给出了合并点的算法；其次是确定上述过程的迭代次数，我们在4.3中对于成功概率和时间复杂度给出了定理支撑，进而可以确定迭代次数。

4.2 算法

首先讨论无权图，进而将结果推广到有权图。

Karger 算法的基本操作是“点的合并”。在图 G 中，设待合并的边为 $\{v_1,v_2\}$，v_1,v_2 为该边的节点，则合并为新节点 v，v 的邻接边即为 v_1,v_2 邻接边的并集。若 v_1,v_2 同时与

某个顶点 u 相邻，则在 u, v 之间形成多条边，且去掉因此产生的所有闭环，如图9所示。合并的最终结果为产生一个只含有两个节点 $\{x, y\}$ 的新图 G_{new}。

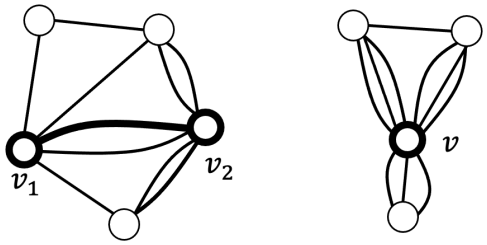

图 9: Karger 算法

合并点的算法伪代码如下：

Algorithm 1 Karger Algorithm

Input: Karger($G = (V, E)$)

Output: A cut of G

```
while |V| > 2 do
    choose e ∈ E uniformly at random
    G ← G/e
end while
```

将上述算法推广到有权图：若连接 u, v 两点的边具有权重 w，则只需将该边用 w 条无权边替代即可，只要这 w 条边中有一条被选中，则合并 u, v 两点。这样便做到了选边的概率与边的权重成正比。

当上述算法终止时，原图 G 中的任意一点必定属于合并后的两个点 u, v 之一，这相当于 u, v 将原图的点划分成两个集合 (V_1, V_2)，(u, v) 之间的连边数就是形成的割的边数。第4.3部分将证明该算法找到最小割的概率至少是 $\binom{2}{n}^{-1}$。

4.3 准确性及时间复杂度

设图 $G = (V, E)$ 中的全局最小割的边的数目为 c，每一次合并完成后的顶点数为 r，边数为 m。

为叙述上的方便，将本节中图 G 的全局最小割简称为 **“全局最小割”**，将算法通过合并点得到的全局最小割简称为 **“最小割”**。

引理 4.1. *在合并点点过程中，图 G 中每个点的度数始终大于等于 c。*

证明. 反证法。

如果某个点的度数小于 c，可以把这个点和其他点分开，形成的最小割边数小于 c，与假设矛盾。

又因为合并得到的点的度数等于它代表的集合与外界的连接数，所以结论在合并点过程中仍然成立。 □

定理 4.1. *通过合并点算法1获得的图 G 的全局最小割，正确率至少为* $\binom{2}{n}^{-1}$。

证明. 算法每进行一次收缩，图 G 的顶点数都会减少 1。当合并到某一步图中有 r 个点时，根据引理4.1，每个顶点的度至少为 c，所以图 G 至少有 $rc/2$ 条边。又因为最终得到的全局最小割只有 c 条边，因此在接下来的一次收缩过程中能收缩到最小割边的概率不会超过 $\frac{2}{r}$。将上述过程执行 $m-2$ 次，由乘法公式可知一次收缩完毕后，始终没有收缩到全局最小割边的概率至少是：

$$(1-\frac{2}{n})(1-\frac{2}{n-1})\dots(1-\frac{2}{3})=\binom{2}{n}^{-1} \tag{6}$$

该定理为我们证明如下定理，并进一步确定迭代次数奠定了基础。 □

定理 4.2. *当合并点的算法执行 $n^2ln(n)$ 之后，找到全局最小割的错误率不高于 $\frac{1}{n}$。*

证明. 由定理4.1可知，执行一次合并点的算法即可获得全局最小割的概率不低于 $\binom{2}{n}^{-1}$，即得不到全局最小割的概率不高于 $(1-\frac{1}{n^2})^T$。因此当迭代 T 次之后，仍然得不到全局最小割的概率

$$P_{false}\le(1-\frac{1}{n^2})^T \tag{7}$$

当 $T=n^2ln(n)$ 时，错误率为：

$$P<\frac{1}{n} \tag{8}$$

这说明可以以低于的 $\frac{1}{n}$ 的失败概率获得最小割，依据此算法得到全局最小割的正确率很高。

□

在图 G 中重复运行 T 次，总运行时间为：

$$\mathcal{O}(m)=\mathcal{O}(n^2mln(n)) \tag{9}$$

5 Stoer-Wagner 算法

Stoer-Wagner 算法是一种递归算法，可以用来解决具有非负权重的无向加权图中的最小割问题，由 Mechthild Stoer 和 Frank Wagner 于 1995 年提出。

5.1 基本思想

首先我们指出，Stoer-Wagner 算法基于如下定理：

定理 5.1. *若 $s,t\in V$，则图 G 的全局最小割是上述 $s-t$ 割和 $G/s\sim t$ 的全局最小割中较小者。*

该定理之所以成立，是因为要么有一个 G 的最小割将 s 和 t 分开，那么 G 的最小 $s-t$ 割就是 G 的最小割；若没有，合并 s,t，求 $G/s\sim t$ 的最小割即可。基于该定理，我们只需要递归地求解图的 $s-t$ 割即可最终得到全局最小割。下面的引理给出了求图的一个 s-t 最小割的方法：

引理 5.1. 对无向有权图 $G=(V,E)$ 的每一个 $MA\text{-}order\, v_1, v_2, ...v_n$，割 $(\{v_1, v_2, ...v_{n-1}\}, \{v_n\})$ 是一个最小 v_n-v_{n-1} 割。

引理的证明将在5.3小节中给出。实际上，上述 $v_n - v_{n-1}$ 的割值就是 v_n 的度，同时也可以看出，我们并不关心找到的 $s-t$ 割中的 s,t 点是哪两个点，只关心基于这两个点得到的 $s-t$ 割。至此，便有了求得图 G 的全局最小割的手段：

首先排序出图 G 的任意一个 $MA-order$ $v_1, v_2, ...v_n$，记录下 $(\{v_1, v_2, ...v_{n-1}\}, \{v_n\})$ 的割值；

继而用同样的方式求出图 $G/v_{n-1} \sim v_n$ 的最小割，若比先前计算的割值小，则更新最小割。如此递归地进行，直到图 G 中只有 2 个点，最终得到的最小割即为全局最小割。

5.2 算法

记当前求出的 $v_n - v_{n-1}$ 割为 CP(cut-of-the-phase)。将上述讨论规范化，便得到 Stoer-Wagner 算法，分为如下两部分（伪代码见算法2）：

实际上，$MinimumCutPhase$ 的目的就是求出一个 $MA\text{-}order$，并计算 $(\{v_1, v_2, ...v_{n-1}\}, \{v_n\})$ 的割值；$GlobalMinCut$ 只是在递归地调用 $MinimumCutPhase$，不断记录每次求出的最小割及其割值而已。

Algorithm 2 Stoer-Wagner Algorithm

Input: Graph $G=(V,E)$

Output: a cut of G

```
function MinimumCutPhase((G, w, a))
    A ← {a}
    while A ≠ V do
        add to A the most tightly connected vertex
    end while
    store the CP and shrink G by merging the two vertices added last
end function

function GlobalMinCut((G, w, a))
    while |V| > 1 do
        MinimunCutPhase(G, w, a)
        if the CP is lighter than the current minumun cut then
            store the CP as the current minimum cut
        end if
    end while
end function
```

5.3 准确性和时间复杂度

为了证明 Stoer-Wagner 算法的正确性，首先证明5.1。

证明. 集合 A 的添加顺序：集合 A 从只含有一个点（记该点为 a）开始，到最后将 v_{n-1}、v_n 两点添加进去。易知，当前图的 $s-t$ 最小割（记为 C）的权重至少与 $v_{n-1} - v_n$ 割（记

为 CP）的权重相同。

当点 $v(v \neq a)$ 和在 v 之前添加的顶点位于 C 中的两个不同部分时，称顶点 v 是活动顶点（相对于 C）。

令 w_c 为 C 的权重，A_v 表示在点 v 被添加进集合 A 之前的所有顶点的集合，$C_v := A_v \bigcup \{v\}$。

对于每一个活动顶点 v，有：

$$w(A_v, v) \leq w(C_v) \tag{10}$$

下面使用**数学归纳法**进行证明：

当有一个活动顶点时，不等式显然成立；

假设当活动顶点为 v 时不等式成立，现在加入一个新的活动顶点 u，那么有：

$$w(A_u, u) = w(A_v, u) + w(A_u \backslash A_v, u) =: \alpha \tag{11}$$

选择 v 作为与 A_v 关系最紧密的连接点，其中 v 满足 $w(A_v, u) \leq w(A_v, v)$。通过对 $w(A_v, v) \leq w(C_v)$ 进行归纳，所有在 $A_u \backslash A_v$ 和 u 之间的边都连接 C 的不同部分。因此有：

$$\alpha \leq w(C_v) + w(A_u \backslash A_v, u) \leq w(C_u) \tag{12}$$

由于 t 始终是相对于 C 的活动顶点，因此得到：$v_{n-1} - v_n$ 最小割的权重不会超过 $s-t$ 最小割的权重。定理证毕。 □

将上述引理与5.1相结合，可知 CP 要么是全局最小割，要么就在全局最小割的同一侧，因此将所有的 CP 记录下来，其中最小的便是全局最小割。

在顶点和边的数量减少的图上调用算法 $GlobalMinCut$，该算法的运行时间是 $|V|-1$ 个 $MinimumCutPhase$ 算法运行时间的加和；

对于算法 $MinimumCutPhase$，最多只需要运行 $\mathcal{O}(|E| + |V|log|V|)$；

总运行时间为两阶段复杂度的乘积，即 $\mathcal{O}(|V||E| + |V|^2 log|V|)$。

减少算法运行时间的关键在于当选择要添加到集合 A 中的下一个顶点时，确保加入此顶点后可以令算法的实现更加简单。在算法执行期间，每当将顶点 v 添加进集合 A 中时，都必须对执行队列进行更新，同时 v 必须在队列中删除，而且需要修改连接到顶点 v 的所有不在集合 A 中的权重 vw。因此，执行次数为 $|V|EXTRACTMAX + |E|INCREASEKEY$.

如果使用**斐波那契堆** [11] 进行更新，就可以在 $\mathcal{O}(log|V|)$ 的时间内执行 $EXTRACTMAX$ 操作，在 $\mathcal{O}(1)$ 的时间内执行 $INCREASEKEY$ 操作。此时，时间复杂度下降为 $\mathcal{O}(|E| + |V|log|V|$。

6 算法测试

6.1 数据集分析

题设共给出 5 个数据集，每个数据集均表示一个无向无权图，因此可以使用上述算法进行求解。对 Karger 算法和 Stoer-Wagner 算法进行复现（代码见附录A），并将算法应用在给定的数据集中，可以求解出每个图的全局最小割。

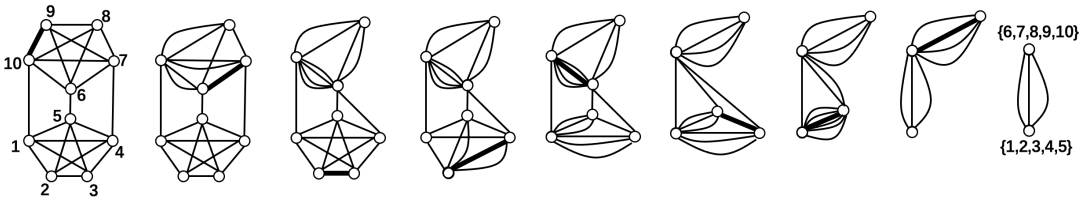

图 10: Karger 算法在 4（1）问中的运行过程

6.2 使用数据集对 Karger 算法进行测试

使用 Karger 算法求解给定数据集中的全局最小割，得到全局最小割值以及最小割集（见表1），部分数据见附录D.1.1。

表 1: Karger 算法在给定数据集中的运行结果

数据集	最小割值	最小割集	备注
4（1）问	3	{3,4,2,5,1}	见图10
		{7,10,6,8,9}	
Benchmark	2	见附录D.1.1	如图11
		见附录D.1.1	
Corruption	1	{19}	如图12
		$V\backslash\{19\}$	
Crime	1	{728}	如图13
		$V\backslash\{728\}$	
PPI	1	{2154}	如图14
		$V\backslash\{2154\}$	
RodeEU	1	{1013}	如图15
		$V\backslash\{1013\}$	

[1] 图中有多种可划分的顶点的集合，这里只展示了其中一种。

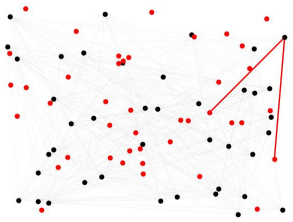

图 11: *Benchmark*

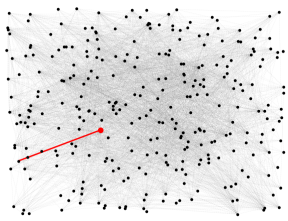

图 12: *Corruption_Gcc*

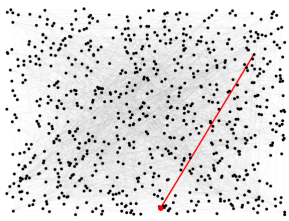

图 13: *Crime_Gcc*

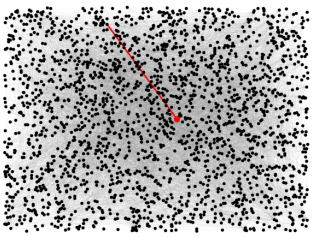

图 14: PPI_gcc

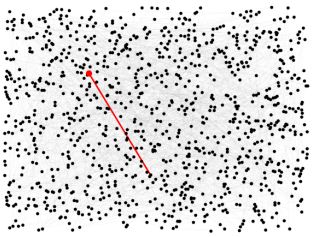

图 15: $RodeEU\_gcc$

6.3 使用数据集对 Stoer-Wagner 算法进行测试

使用 Stoer-Wagner 算法求解给定数据集中的全局最小割，得到全局最小割值以及最小割集（见表2），部分数据见附录D.1.2。

表 2: Stoer-Wagner 算法在给定数据集中的运行结果

数据集	最小割值	最小割集
4（1）问	3	{1,2,3,4,5}
		{6,7,8,9,10}
Benchmark	2	见附录D.1.2
		见附录D.1.2
Corruption	1	{19}
		$V\backslash\{19\}$
Crime	1	{606}
		$V\backslash\{606\}$
PPI	1	{1760}
		$V\backslash\{1760\}$
RodeEU	1	{589}
		$V\backslash\{589\}$

[1] 图中有多种可划分的顶点的集合，这里只展示了其中一种。

6.4 算法运行说明

- 在这六个数据集中，Karger 算法和 Stoer-Wagner 算法得到的全局最小割值是相同的；
- 两个算法得到的割的不同是因为图中的全局最小割不止有一条，karger 为随机算法，每次运行之后可能会返回不同的全局最小割。

7 算法评价与推广

7.1 Karger 算法

7.1.1 前人的推广与改进

(1)Karger-stein 算法

上文已经证明 Karger 算法的正确率（见7），然而当图的边数较少时，通过 Karger 算法找到的全局最小割的错误率会变大。David Karger 和 Clifford Stein 在原有算法基础上实现了一个数量级的改进（见算法3），称之为**递归压缩算法**。

该算法的基本思想是执行合并点的过程直到图达到 $[1+n/\sqrt{2}]$ 个顶点。其中，第 k 次迭代成功的概率为:

$$P[x_0 \cap ... \cap x_{k-1}] \geqslant \Pi_{i=0}^{k-1}(1-\frac{2}{n-i}) = \Pi_{i=0}^{k-1}\frac{n-i-2}{n-i} = \frac{(n-l)(n-l-1)}{n(n-1)} \tag{13}$$

Algorithm 3 Karger-Stein Algorithm

Input: Graph $G=(V,E)$

Output: a cut of G

```
if V ≤ 6 then return Karger Algorithm
end if
t ← ⌈n/sqrt2⌋
for i ← 1 to 2 do
    G_i ← RandContract(G, t)
    (S_i, T_i) ← FastRandMinCut(G_i)
end for
if w(S_1, T_1) ≤ w(S_2, T_2) then return (S_1, T_1)
else    return (S_2, T_2)
end if
```

引理 7.1. *合并点算法 $Contract(G, n/\sqrt{2})$ 没有收缩最小割的概率至少为 1/2。*

引理 7.2. *karger-stein 算法的运行时间为 $O(n^2logn)$，其中 $n=|V(G)|$。*

定理 7.1. *使用 karger-stein 算法得到最小割的值概率大于 $2log2/logn$，可记为 $\Omega(1/logn)$*

根据上述定理，运行 Karger-Stein 算法 $clog^2n$ 次，可以保证算法输出的最小割概率大于 $1-\frac{1}{n^2}$，c 是足够大的常数。

(2) 改进的 Karger-stein 算法

在 Karger-Stein 算法基础上，Karger 提出了一种更新的改进方法，引入了无向图的**随机稀疏化**和 **α-近似算法**的概念：通过随机稀疏化产生一个接近原始图最小割结构的稀疏无权图；通过求得稀疏无权图的最小割可以近似得到原图的最小割。

由于稀疏无权图的最小割可以通过快速算法 Gabow's 算法解决，因此整个算法可以在 $O(m+n(log^3n)/\epsilon^4)$ 的时间复杂度下找到 $(1+\epsilon)$-最小割，该算法比 Karger-Stein 算法更快。

首先我们提出 α-近似算法的概念。

定义 7.1. **α 最小割** *若某个割的权值与最小割的权值只相差一个因子 α，则称该割为 α 最小割；*

α-近似算法指的就是在原图上找到 α-最小割的算法。

进一步引入稀疏子图（sparse certificate）：

定义 7.2. *图 G 的 k 连通稀疏子图 (k-connectivity certificate) 是 G 的一个子图，它最多包含 kn 条边，且使得原图中所有权值不超过 k 的割在子图中具有相同的权值。*

Nagamochi 和 Ibaraki 给出了寻找稀疏 k 连通性 certificates 的算法，该算法在无权图中的时间复杂度是 $\mathcal{O}(m)$，在有权图中的时间复杂度是 $\mathcal{O}(m + nlogn)$。

该算法的核心是将原图进行随机的稀疏化。如果一个稀疏图与原图有完全相同的点，则称该稀疏图为一个原图的一个骨架。

构建 G 的一个骨架 $G(p)$ 的方法是：复制原图中所有的点，对原图中对每一条边，以概率 p 将其复制到骨架中。按照这种方式，一个在 G 中有 k 条边的割有大约 pk 条边包含在 $G(p)$ 中。因此，若该割在 G 中的权重为 k，则它在 $G(p)$ 中的权重的数学期望应该为 pk。若恰好取得该数学期望，那么 $G(p)$ 中的最小割值将是 pc, 并且 G 中相应的割将是最小割。

但由于骨架 $G(p)$ 只是一个随机图，因此这个比例一般是不成立的，会有一些割极大地偏离这个数学期望，但是根据如下引理，α 最小割的数量有上限的，因此可以得到一个模拟采样的算法。

引理 7.3. *在无向图中，α 最小割的数量不超过 $n^{2\alpha}$。*

基于该定理，作者给出了构造 $G(p)$ 的算法:

- 使用 $Matula$ 线性时间算法估计最小割 c 到一个因子 3。估计值 c' 满足 $c < c' < 3c$;
- 使用 Nagamochi 和 Ibaraki 的算法 [4] 在线性时间内构造一个稀疏的 $2c'$-连通稀疏子图;
- 在子图中随机选取 $6nk$ 条边。

使用上述算法构建骨架 $G(p)$ 后，问题转化成如何在稀疏图上找一个好的近似算法，作者改进了 Matula 的 $(2+\epsilon)$-近似算法：

- 给定图 G，计算最小度 d;
- 计算一个 $\left(\frac{d}{2+\epsilon}\right)$-连通稀疏子图;
- 收缩所有非子图的边并进行递归，在合并后的图中找到一个近似的最小割;
- 返回较小的 d 和上步的结果。

借助于 $\alpha-$ 近似算法和稀疏子图，从而可以得到 Karger-Stein 算法的改进如下：

引理 7.4. *如果一个有 n 个点的图经过合并后有 h 个顶点，那么以概率 $\Omega((k/n)^{2/\alpha})$，下一步合并将要么保持图中的最小割，要么显示出一个 α-最小割。*

算法4采用前面提到的稀疏子图算法。该算法在 $O(\alpha c)$ 时间内运行，并找到一个包含 $O(\alpha cn)$ 条边的 (αc)-连通稀疏子图连接子图。任何在 G 中值超过 αc 的切割对应于子图中值超过 αc 的割。

Algorithm 4 Improved Karger-Stein Algorithm

Input: A graph G of size n

Output: a cut of G

if $n = 2$ **then**

examine the implied cut of the original graph

else

Find an (αc)-connectivity certificate G' of G

end if

repeat twice

$G^{''} \leftarrow CA(G', n/2^{\alpha/2})$

$RCA(G^{''}, n/2^{\alpha/2})$

对于一个 α-最小割和将一个 n 顶点图合并为 $n/2^{a/2}$ 个顶点，有 50% 的机会找到一个 α-最小割或保留原图的最小割。因此，对稀疏子图算法和 RCA 的调用只需要 cn 处理器。此时，处理器成本有以下递归式：

$$T(n) = 2(cn + T(n/2^{\alpha/2})) \tag{14}$$

7.1.2 我们的推广与改进

从定理4.1的证明过程中可以看出：当图中点的数量较多时，最小割边被收缩的概率较小；而当图中点的数量逐渐变少时，最小割边被收缩的概率会逐渐增大，所以整个算法才需要足够多的循环次数才能保证以较小的错误率找到全局最小割。事实上，在前几次合并时，最小割边被收缩的概率很小，对这个过程进行重复的循环是没有必要的，只需给出一个错误率的下界，将图中的点合并到错误率允许的范围内，对这个规模较小的图进行更多次数的合并即可。因此我们提出**分层合并策略**，将整个合并过程分为两层，具体步骤如下：

首先给定一个我们可以接受的算法错误率下界 α，我们的改进算法无法返回全局最小割的概率不会超过 α。在第一层合并点的过程中，将原图的 n 个点合并至 k 个点，这一层点的合并影响到全局最小割的概率记为 α_1，则有：

$$\alpha_1 \leq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \dots (1 - \frac{2}{n-k}) \tag{15}$$

此时图中剩下 $n-k-1$ 个点，在该子图上执行第二层合并。根据4.2可知，对该子图执行合并点的算法 $(n-k)^2 ln(n-l)$ 次之后，无法返回全局最小割的概率

$$\alpha_2 \leq \frac{1}{n-k-1} \tag{16}$$

令：$\alpha_1 + \alpha_2 \leq \alpha$，解出 k 便可确定第一层的循环次数。

改进算法的伪代码如下：

Algorithm 5 Improved Karger Algorithm

Input: Graph G, a given error rate α

Output: the global min cut of G

$\alpha_1 \leftarrow (1 - \frac{2}{n})(1 - \frac{2}{n-2}) \dots (1 - \frac{2}{n-k})$

$\alpha_2 \leftarrow \frac{1}{n-k-1}$

$G' \leftarrow contract(G, n-k-1)$

$Karger(G')$

7.2 Store-Wagner 算法

Stoer-Wagner 算法在每一次迭代中，计算了无向图中点的最大邻接顺序，然后根据这个顺序合并排在末尾的两个点。

7.2.1 前人的推广与改进

一种新的改进算法由 Michael Brinkmeier 在 2007 年提出 [12]，为了叙述上的方便，下文将改进后的算法称为 Michael Brinkmeier 算法。该算法对 Stoer-Wagner 算法做了两处改进：首先，Michael Brinkmeier 算法每次合并点时不再是只合并两个点，而是尽可能地合并多个点，这就将最坏情形复杂度降低到了 $\mathcal{O}(max\{logn, min\{\frac{m}{n}, \frac{\delta_G}{\epsilon}\}\}n^2)$，其中 ϵ 为最小的边权；其次，Michael Brinkmeier 算法将 Stoer-Wagner 算法中的最大邻接顺序放宽了限制，定义了一种“松弛”邻接顺序，降低了寻找最大邻接点的时间复杂度。

首先，Michael Brinkmeier 通过如下方式实现了对图 G 同时合并多个点：

对 $\tau \geq 0$，以及 MA-order$v_1, v_2, ...v_n$，如下定义 G_i^τ：

$$G_1^\tau := G, G_{i+1}^\tau := G_i^\tau \quad if \quad w(v_1, v_2, ...v_i; v_{i+1}) < \tau \quad else \quad G_{i+1}^\tau := G_i^\tau / [v_i] \sim v_{i+1} \tag{17}$$

其中 $[v_i]$ 是在图 G 中已经合并到 v_i 中的所有点。经过上述合并，图 G 中所有邻接度大于 τ 的点都已经合并到前一个点当中。

进一步地，为了降低寻找最大邻接点的时间复杂度，基于定义2.5，定义如下松弛邻接顺序：

定义 7.3. *$G = (V, E)$ 是无向、有权图，$\tau \geq 0$. 若 $v_1, v_2, ...v_n$ 对 $\forall k \geq i$，满足 $min\{\tau, w(v_1, v_2, ...v_{i-1}; v_i)\} \geq min\{\tau, w(v_1, v_2, ...v_{i-1}; v_k\}$，则称 $v_1, v_2, ...v_n$ 为松弛邻接顺序或关于 τ 的 LA-order。*

通过该定义可以看出，*LA-order* 在排序时，并不要求每一个点的邻接度是后继所有点中最大的，而只要该点的邻接度高于阈值 τ 即可，这样在构建 *LA-order* 时，便不需要如 *MA-order* 一样遍历每一个点。接下来的关键就是 Michael Brinkmeier 证明了该 LA-order 构造出的图 G^τ 与 *MA-order* 合并后的图是等价的。

因此便有了如下算法6：

若边权是非负实数，上述算法的时间复杂度是 $\mathcal{O}(max\{logn, min\{\frac{m}{n}, \frac{\delta_G}{\epsilon}\}\}n^2)$，其中 ϵ 为最小的边权。

Algorithm 6 Improved Stoer-Wagner Algorithm

Input: An undirected graph $G = (V, E)$ with weights w

Output: A set outof vertices of G forming a minimal cut of weight τ

```
cut← ∅, τ ← ∞
while |V| ≥ 2 do
    for all v ∈ V do
        Q.insert(v, 0)
        if deg(v) < τ then
            τ ← deg(v), cut ← [v]
        end if
    end for
    while Q is not empty do
        adj ← Q.maxKey()
        v ← Q.extraMax()
        for all u ∈ V with (v, u) ∈ E do
            if u ∈ Q then
                Q.increaseKey(u, Q.key(u) + w(v, u))
            end if
        end for
        if τ ≤ adj then
            contract v into u
            [u] ← [u] ⋃ [v]
        end if
        u ← v
    end while
end while
```

7.2.2 我们的推广与改进

在 Stoer-Wagner 算法中，构建 $MA\text{-}order$ 是整个算法当中时间复杂度最高的部分，该算法在每次合并 $MA\text{-}order$ 中的最后两个点之后，又从初始点 v_1 开始重新构建了图 $G/v_{n-1} \sim v_n$ 的 $MA\text{-}order$。实际上，对于中等规模及大规模数据，这极大地浪费了先前构建好的 $MA\text{-}order$。若对于某个 k，最终合并的两个点 v_{n-1}, v_n 与 $v_1, v_2, ...v_k$ 没有邻边，则新构建的 $MA\text{-}order$ 与上一轮构建的 $MA\text{-}order$ 前 k 个点是完全相同的，这就省去了 $k|V|$ 次遍历，只需从第 $k+1$ 个点开始构建 $MA\text{-}order$ 即可。

因此在 Stoer-Wagner 算法的基础上改进为我们的算法7:

Algorithm 7 Improved Stoer-Wagner Algorithm

Input: Graph $G = (V, E)$

Output: a cut of G

```
function MinimumCutPhase((G, w, a))
    if length of A ≥2 then
        i ← the first vertex conneted to the last two vertices
        A ← A(0 : i)
    else
        A ← {a}
    end if
    while A ≠ V do
        add to A the most tightly connected vertex
    end while
    store the CP and shrink G by merging the two vertices added last
end function

function GlobalMinCut((G, w, a))
    while |V| > 1 do
        MinimunCutPhase(G, w, a)
        if the CP is lighter than the current minumun cut then
            store the CP as the current minimum cut
        end if
    end while
end function
```

7.3 未来可能的改进方向

从问题的本质上考虑，最小割实际就是对图的连通性的一种刻画，它在一定程度上包含了图的某些信息。无论是 karger 算法还是 Stoer-Wagner 算法，都设计一系列的策略来尽可能地反映这些信息，如 karger 算法中合并点的算法本质上就是将点进行划分，对原图进行简化；Stoer-Wagner 算法中定义的 MA-order 是对点邻接程度的一种刻画。因此改进的原则是仍然尽量不破坏原图的信息。

karger 算法作为一种随机算法，目前文献中出现的改进主要集中在两大方向：

（1）控制算法的错误率，同时降低算法的时间复杂度；

（2）控制算法的时间复杂度，同时降低算法的错误率。

因此未来的工作可能更多地集中于创造新的合并点的策略，使得合并之后得到的子图仍然包含原图的一些关键信息，进而对子图进行一系列的操作。这些策略能够在一定程度上降低错误率或降低时间复杂度，我们在上文中提出的改进便基于后者。

Stoer-Wagner 算法保证能够返回全局最小割，因此其改进方向便只局限于降低时间复杂度或研究其并行的可能性。与 Karger 算法相比，目前学术界对它的改进相对较少，且集中在对 MA-order 的处理上。未来的方向可能在于能否更好地利用事先计算出的 MA-order，或者定义其他的能够刻画点的邻接性的数据结构，构建这种数据结构所需的时间比 MA-order 更少，我们在上文中提出的改进便基于前者。

参考文献

[1] FORD L R, FULKERSON D R. Maximal flow through a network[J]. Math, 1956(8): 399 – 404.

[2] HAO J, ORLIN J B. A faster algorithm for finding the minimum cut in a graph[C] // The 3rd ACM-SIAM Symposium on Discrete Algorithms. 1992: 165 – 174.

[3] GOLDBERG A V, TARJAN R E. A new approach to the maximum-flow problem-[J]. ACM, 1988(4): 1 – 7.

[4] NAGAMOCHI H, ONO T, IBARAKI T. Implementing an efficient minimum capacity cut algorithm[J]. Math, 1994, 67: 325 – 341.

[5] STOER M, WAGNER F. A Simple Min-Cut Algorithm[J]. Proceedings of the 2nd Annual European Symposium on Algorithms, 1994, 855: 141 – 147.

[6] R.KARGER D. A New Approach to the Minimun Cut Problem[J]. Laboratory for Computer Science Massachusetts Instutute of Technology, 1996.

[7] PFERSCHY U, SCHAUER J. The maximum folw problem with disjunctive constraints[J]. Complexity of Computer Computations, 2011: 109 – 119.

[8] GAREY M R, JOHNSON D S. COMPUTERS AND INTRACTABILITY:A Guide to the Theory of NP-Completeness[M]. : W.H. FREEMAN AND COMPANY New York, 1979: 209.

[9] MORANDINI M. NP–COMPLETE PROBLEM: PARTITION INTO TRIANGLES[J]. Corso di Complessita‘ Prof. Romeo Rizzi, 2003-2004.

[10] R.M. K. Reducibility among combinatorial problems[J]. Complexity of Computer Computations, 1972: 85 – 103.

[11] FREDMAN M L, TARJAN R E. Fibonacci heaps and their uses in improved network optimization algorithms[J]. ACM, 1987, 34(3): 596 – 615.

[12] STOER M, WAGNER F. A simple min-cut algorithm[J]. Clarivate Analytics Web of Science, 1997: 585 – 591.

附录

A Karger 算法复现代码

```
from random import randint
from math import log
import numpy as np
import math
import copy
class Karger():
    ##数据处理工作，将图初始化成字典的形式
    def __init__(self,graph_file):
        self.graph={}
        self.vertex_count=0
        self.edges=0
        self.over_poind = {}
        my_data = np.loadtxt(graph_file, dtype="int")
        lie_1 = my_data[:, 0]
        lie_2 = my_data[:, 1]
        unique_data = np.unique(my_data)
        for i in range(1, len(unique_data) + 1):
            self.graph[i] = []
            for num in range(0, len(lie_1)):
                if lie_1[num] == i:
                    self.graph[i].append(lie_2[num])
            for num2 in range(0, len(lie_2)):
                if lie_2[num2] == i:
                    if lie_1[num2] not in self.graph[i]:
                        self.graph[i].append(lie_1[num2])
        for i in range(1,len(self.graph)+1):
            self.vertex_count+=1
            self.edges+=len(self.graph[i])
            self.over_poind[i] = [i]###supervertices用来存储最后合并的结果
    #这里开始寻找最小割的函数
    def search_min_cut(self):
        minimumcut = 0##让他初始化等于0
        while len(self.graph)>2:
            #print("这一次循环的变化情况")
            #print(self.supervertices)
            # 先选择一条随机的边，同时得到随机的点
            vertice1, vertice2 = self.random_select()
            self.edges -= len(self.graph[vertice1])###边数应该减掉和这两个点相通的所有点长度
                                                    的和，即在边集合中删除这俩
            self.edges -= len(self.graph[vertice2])
            # 然后合并这个图
            self.graph[vertice1].extend(self.graph[vertice2])###图的数据集中把vertice2相关的
                                                              点加到vertice1里面去
            # Update every references from v2 to point to v1
            for vertex in self.graph[vertice2]:
                self.graph[vertex].remove(vertice2)
                self.graph[vertex].append(vertice1)
            # 删除自环！！！！！！
            self.graph[vertice1] = [x for x in self.graph[vertice1] if x != vertice1]
            # 更新图的全部的边
            self.edges += len(self.graph[vertice1])###再加上合并之后的值
            self.graph.pop(vertice2)###图的集合中彻底删除vertice2的数据
```

```
            # 更新图的割集
            self.over_poind[vertice1].extend(self.over_poind.pop(vertice2))
            ###循环之后图中就剩两个点，这两个点中间的连线就是最小割
        #然后现在可以来计算最小割了
        for edges in self.graph.values():
            minimumcut = len(edges)
        #  最后返回最小割和形成的两个集合
        return minimumcut,self.over_poind
    # 选取随机边的函数
    def random_select(self):
        rand_edge = randint(0, self.edges-1)###生成一个随机的值，从0到所有边数中
        for key, value in self.graph.items():
            if len(value) < rand_edge:###从第一行开始逐渐寻找那条边，第一行不是就减掉第一行
                                                                的长度，知道该行长度超过了边数
                rand_edge -= len(value)
            else:
                vertex_origin = key
                vertex_last = value[rand_edge-1]
                return vertex_origin, vertex_last    ##返回索引点和索引点集合里的点


if __name__ == '__main__':
    minimumcut=9999###只是一个足够大的值用来比较最小割
    graph_feature = Karger('data/RodeEU_gcc.txt')
    iterations = len(graph_feature.graph) * len(graph_feature.graph) * int(math.log(len(
                                                graph_feature.graph)))  #运算次数
    print("一共需要运行{}次".format(iterations))
    for i in range(10):
        graph_copy =Karger('data/RodeEU_gcc.txt')
        result = graph_copy.search_min_cut()
        #if result[0]==1:###任何一次找到最小割为1都可以停止循环，因为最小为1.
           # minimumcut = result[0]
            #over_point = result[1]
            ##break
        #else:
        if result[0] < minimumcut:
            minimumcut = result[0]
            over_point = result[1]
    print("最小割的值是:{}".format(minimumcut))
    for key in over_point:
        print("源点是{}，集合内的元素为{}".format(key, over_point[key]))
```

B Store-Wagner 算法复现代码

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

def MyStoerWagner(Graph):
    # 输入：用networkx表示的图Graph
    # 输出：全局最小割及其值
    vertex_num=len(Graph)            # 点的数量
    cut_value = float('inf')         # 最小割值初始化为无穷大
    nodes = set(Graph)               # 点集
    contractions = []                # 记录每次合并的点
    InitialPNT=Graph.nodes()[np.random.randint(0,vertex_num)]  # 随机选一个初始点加入到A中
    print(InitialPNT)
    for i in range(vertex_num-1):                # 直到除了A之外只有一个点
        A = {InitialPNT}
        h = []  # h用来记录A中点的邻接点
        for vertex, attribute in Graph[InitialPNT].items():
            heapq.heappush(h,[-attribute['weight'],vertex])   # vertex是和u相邻的点，weight
                                                                是权重，因此h存的是初始点的所有邻
                                                                接点
        for j in range(vertex_num-i-2):          # 每次合并一个点，因此往A中加的点就少一个
            print('h=',h)
            most_tighted_v = heapq.heappop(h)[1]     # most_tighted_v是离初始点权重最大的点
            print("most tighted is ",most_tighted_v)
            A.add(most_tighted_v)
            print("after adding, now A is:",A)
            for v, e, in Graph[most_tighted_v].items():      # u是刚加入A的点，如果它的邻接
                                                               点v不在A中，就加到h中去
                if v not in A:
                    index=get_heap_element(h,v)
                    if index!=-1:
                        h[index][0]-=e['weight']             # 如果v已经在h中了，就把权重累加
                        heapq.heapify(h)
                    else:
                        heapq.heappush(h, [-e['weight'],v])      # 如果v不在h中，就新建一个
        print("测试last的时候h中是不是只有一个点，h=",h)
        last, w = h[0][1],h[0][0]  # 上面的for循环结束之后只剩下一个点v不在A中了，它是最后一
                                     个点也是h中唯一的点
        w=-w
        print("phase=",i,'min cut value=',w)
        if w < cut_value:
            cut_value = w
            best_phase = i
        # 收缩最后两个点
        print("contractions=",contractions)
        for w, e in Graph[last].items():  # 遍历last的所有邻接点
            if w != most_tighted_v:
                if w not in Graph[most_tighted_v]:  # 如果v的邻接点不和u相邻，就让他们相邻
                    Graph.add_edge(most_tighted_v, w, weight=e['weight'])
                else:
                    Graph[most_tighted_v][w]['weight'] += e['weight']  # 如果v的邻接点和u相
                                                                         邻，权重相加
        Graph.remove_node(last)  # 把v删掉
        # print("removed:",last)
    partition=[]
    for i in range(best_phase):
        for j in contractions[i]:
```

```
            if j not in partition:
                partition.append(j)
    partition=(set(partition),nodes-set(partition))
    return cut_value,partition

def get_heap_element(HEAP,vertex):
    vertex_name_list=[]
    for i in HEAP:
        vertex_name_list.append(i[1])
    if vertex in vertex_name_list:
        index=vertex_name_list.index(vertex)
        return index
    else:
        return -1



import networkx as nx
import numpy as np
from MyStoerWagner import MyStoerWagner
BenchmarkNetwork=np.loadtxt('data/BenchmarkNetwork.txt',dtype=int)
Corruption_Gcc=np.loadtxt('data/Corruption_Gcc.txt',dtype=int)
Crime_Gcc=np.loadtxt('data/Crime_Gcc.txt',dtype=int)
PPI_gcc=np.loadtxt('data/PPI_gcc.txt',dtype=int)
RodeEU_gcc=np.loadtxt('data/RodeEU_gcc.txt',dtype=int)
BenchmarkNetwork_Graph=nx.Graph()
for line in BenchmarkNetwork:
    BenchmarkNetwork_Graph.add_edge(int(line[0]),int(line[1]),{'weight':1})
cut_value, partition = MyStoerWagner(BenchmarkNetwork_Graph)
print("BenchmarkNetwork的最小割是: ",cut_value,partition)

RodeEU_gcc_Graph=nx.Graph()
for line in RodeEU_gcc:
    RodeEU_gcc_Graph.add_edge(int(line[0]),int(line[1]),{'weight':1})
cut_value, partition = MyStoerWagner(RodeEU_gcc_Graph)
print("RodeEU_gcc的最小割是: ",cut_value,partition)

Corruption_Gcc_Graph=nx.Graph()
for line in Corruption_Gcc:
    Corruption_Gcc_Graph.add_edge(int(line[0]),int(line[1]),{'weight':1})
cut_value, partition = MyStoerWagner(Corruption_Gcc_Graph)
print("Corruption_Gcc的最小割是: ",cut_value,partition)

Crime_Gcc_Graph=nx.Graph()
for line in Crime_Gcc:
    Crime_Gcc_Graph.add_edge(int(line[0]),int(line[1]),{'weight':1})
cut_value, partition = MyStoerWagner(Crime_Gcc_Graph)
print("Crime_Gcc的最小割是: ",cut_value,partition)

PPI_gcc_Graph=nx.Graph()
for line in PPI_gcc:
    PPI_gcc_Graph.add_edge(int(line[0]),int(line[1]),{'weight':1})
cut_value, partition = MyStoerWagner(PPI_gcc_Graph)
print("PPI_gcc的最小割是: ",cut_value,partition)
```

C Karger-Stein 算法复现

```
import math
import copy
import numpy as np
####karger算法，用于将图缩放到某几个点时停止
def karger(graph,k):
    while len(graph)>k:##与普通的karger算法不同，缩放到t个点的时候算法就可以返回值了
        vertex_1 =np.random.choice(list(graph.keys()))##选取开始的点
        vertex_2 = np.random.choice(list(graph[vertex_1]))###选取结束的点
    #把选中的另一个点相关的边添加到另一个点的集合中
        for edges in graph[vertex_2]:
            if edges!= vertex_1: # 防止产生自环
                graph[vertex_1].append(edges)
#删除掉和删除点相关的边
        for edge1 in graph[vertex_2]:
            graph[edge1].remove(vertex_2)
            if edge1 != vertex_1:###防止再把相同的加进去
                graph[edge1].append(vertex_1)
        del graph[vertex_2]
#这里开始可以计算最小割了，并进行存储。
    #mincut = len(graph[list(graph.keys())[0]])
    #cuts.append(mincut)
    return graph
###karger-stein算法，是我们这个文档的主算法，用的是递归的函数
def karger_stein(graph):
    if len(graph) < 6:####当图较小时，可以直接运行karger算法，或者当图缩放到一定程度时，运行
                                                    karger算法
        return karger(graph,2)
    else:
        k = 1 + int(len(graph)/math.sqrt(2))###计算需要把图变成多大的
        #print(t)
        graph_1 = karger(graph, k)##这个缩放结果跑两次，因为每一次的概率都大于1/2.
        graph_2 = karger(graph, k)
        ###找到较小的作为返回值
        if len(graph_1) > len(graph_2):###选取两个中较小的那个继续递归
            return karger_stein(graph_2)
        else:
            return karger_stein(graph_1)
###首先处理数据
my_data = np.loadtxt('./data/BenchmarkNetwork.txt',dtype="int")###读取数据
#print(my_data)
lie_1=my_data[:,0]
lie_2=my_data[:,1]
graph={}
cuts = []
unique_data = np.unique(my_data)
for i in range(1,len(unique_data)+1):
    graph[i] = []
    for num in range(0,len(lie_1)):
        if lie_1[num]==i:
            graph[i].append(lie_2[num])
    for num2 in range(0, len(lie_2)):
        if lie_2[num2] == i:
           if lie_1[num2] not in graph[i]:
                graph[i].append(lie_1[num2])
count = int(math.log(len(graph)))*int(math.log(len(graph)))# running times
print("count value is "+str(count))
###先跑一次作为比较值的出现
```

```
graph1 = copy.deepcopy(graph)
compared = karger_stein(graph1)
#print(len(list(g.values())[0]))
mincut=len(list(compared.values())[0])
for i in range(0, 100):##这里的次数应该是count，但是因为数据集比较快，用100次就可以了
    graph1 = copy.deepcopy(graph)
    g = karger_stein(graph1)
    #print(len(list(g.values())[0]))
    if len(list(g.values())[0])<mincut:
        mincut=len(list(g.values())[0])
print("最小割的值是"+str(mincut))
```

D 算法的测试结果

D.1 BenchmarkNetwork.txt

D.1.1 Karger 算法

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

{41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60} 和 {61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80}

D.1.2 Stoer-Wagner 算法

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

D.2 Corruption_Gcc.txt

D.2.1 Karger 算法

{309} 和 $V\backslash\{309\}$

D.2.2 Stoer-Wagner 算法

{19} 和 $V\backslash\{19\}$

D.3 Crime_Gcc.txt

由于数据点较多，如下几个数据集仅列出的是最小割的两个点集中具有较少点的集合。

D.3.1 Karger 算法

{614}, {609}, {332}, {235}, {729}, {683}, {728}, {513}, {180, 269}, {413}, {620}, {750}, {16, 375, 422, 624}, {745}, {500}, {658}, {629}, {753}, {686}, {530}, {524}, {135,

330, 331, 698}, {718}, {1, 90, 629, 686}, {120, 441, 520, 744}, {543}, {694}, {627}, {207}, {561}, {640}, {156, 343, 563, 605}, {50}, {390}, {700}, {559}, {384}, {698}, {371}, {194}, {232}, {230, 278, 309, 385, 454, 480, 533, 571, 578, 631, 637, 638, 666, 691}, {458}, {277}, {535}, {349}, {76, 134, 522, 652, 696}, {174, 223, 252, 328, 367, 417, 670, 671, 743}, {207, 471}, {628}, {103, 104, 164, 248, 532, 643, 740}, {189}, {688}, {113}, {339}, {643}, {451}, {641, 642, 657}, {223}, {635}, {376}, {263}, {293}, {82}, {321}, {214}, {194, 195}, {571}, {184}, {115}, {176}, {512}, {280}, {132}, {131}, {405}, {147}, {669}, {316}, {520}, {170, 538, 745}, {344}, {112, 126}, {187}, {315}, {591}, {61}, {175}, {261}, {66}, {440}, {650}, {121}, {495}, {239, 681, 747}, {489}, {606}, {278, 385, 454, 533, 571, 578}, {306}, {84}, {133}, {504}, {126}, {31, 34, 250, 572, 690}, {159}, {385}, {95}, {317, 626}, {30, 325, 373, 458, 472, 474}, {261, 310}, {452}, {626}, {90}, {19, 20, 23, 37, 63, 69, 82, 105, 106, 119, 125, 141, 233, 281, 358, 359, 360, 361, 362, 363, 371, 393, 424, 428, 460, 495, 499, 548, 606, 639, 656, 658, 669, 727}, {170}, {3}, {80}, {22}, {468}, {33}, {201}, {214, 463}, {460}, {63}, {269}, {92}, {197}

D.3.2 Stoer-Wagner 算法

{606}

D.4 PPI_gcc.txt

D.4.1 Karger 算法

{1948}, {2205}, {1141}, {2034}, {1514}, {1499}, {1558}, {2223}, {2197}, {2180}, {980}, {2196}, {1403}, {2199}, {2137}, {2144}, {2087}, {1593}, {2195}, {2210}, {1315}, {2096}, {1944}, {667}, {1718}, {2162}, {2171}, {2147}, {1996}, {1220}, {2112, 2192, 2220, 2221}, {2192, 2220, 2221}, {2154}, {1992}, {2071}, {2220}, {1977}, {1895}, {2224}, {1883}, {1473}, {2208}, {1616}, {2163}, {2187}, {2164}, {2076}, {1736}, {1570}, {2214}, {2221}, {2082}, {2036}, {259}, {938}, {1633}, {2044}, {893}, {1809}, {2176}, {1783}, {2218}, {2092}, {2222}, {1949}, {1940}, {2009}, {2059}, {1128}, {2021}, {1878}, {1740}, {2201}, {1389}, {2126}, {2113}, {1761}, {2168}, {2128}, {1772}, {2217}, {1124}, {1414}, {859}

D.4.2 Stoer-Wagner 算法

{1760}

D.5 RodeEU_gcc.txt

D.5.1 Karger 算法

{713, 714, 715, 716, 717}, {985, 987}, {1011, 1012}, {983}, {1032, 1033}, {1029, 1030}, {610}, {114}, {885, 886, 887, 888}, {1039}, {589, 590}, {982}, {897}, {909, 910}, {990}, {848, 849}, {885, 886, 887}, {965}, {845, 846}, {76, 77, 78, 79, 80, 598, 599, 600}, {1002}, {1012}, {879}, {1029}, {938}, {1028}, {561, 562}, {580}, {589, 590, 591, 592, 593, 594, 595, 596}, {1022}, {996, 997, 998, 999}, {1038, 1039}, {659}, {930}, {787}, {987}, {534}, {755}, {323}, {953, 954, 955, 956}, {875}, {1034}, {1001}, {880, 881, 884}, {834, 835, 836,

837, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1016, 1017, 1018}, {713, 714, 715, 716, 717, 718, 719, 857}, {585, 586}, {893, 894}, {999}, {76, 77, 598, 599, 600}, {829, 830}, {1003, 1004}, {192}, {972, 973}, {589, 590, 591, 592}, {713, 714, 715, 716, 717, 718, 719, 856, 857}, {69, 70, 71, 72, 73, 74, 75, 107, 108, 109, 110, 111, 112, 113, 114, 143, 144, 337, 338, 339, 340, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 610, 720, 721, 722, 723, 724, 725, 726, 727, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 762, 885, 886, 887, 888, 889, 890, 891}, {1034, 1035, 1036, 1037}, {713, 714, 715, 716, 717, 718}, {966}, {1003}, {962}, {105, 106, 1022}, {225}, {1034, 1035}, {113, 114}, {1009, 1010, 1011, 1012}, {1005, 1006, 1016}, {589, 590, 591, 592, 593, 594}, {1006}, {904, 905}, {931}, {704, 705, 706}, {754, 755}, {971}, {988}, {706}, {903, 904, 905, 906, 907, 908, 909, 910}, {1031, 1032, 1033}, {145, 146}, {705, 706}, {585, 586, 587, 588, 739, 740, 741, 742, 743, 744}, {802}, {1016}, {906}, {947}, {554}, {995, 996, 997, 998, 999}, {831, 832}, {842}, {861}, {972}, {267, 268, 269}, {831}, {935}, {991}, {555}, {589, 590, 591, 592, 593, 594, 595}, {543}, {972, 973, 974, 975, 976, 977, 978, 991}, {713, 714, 715, 716, 717, 718, 719}, {910}, {859}, {998, 999}, {840, 841, 842}, {1033}, {884}, {69, 70}, {1010, 1011, 1012}, {580, 581, 582, 583, 584, 890}, {269}, {580, 581, 582, 583, 584}, {953, 954}, {860}, {598}, {186, 187, 188}, {730, 731, 732}, {256, 257, 506, 507, 508, 509}, {880, 881, 882, 884}, {953, 954, 955}, {1019}, {580, 581}, {76}, {972, 973, 974, 975, 976}, {1008, 1009, 1010, 1011, 1012, 1017, 1018}, {981, 982}, {578, 579}, {969}, {870, 871, 872}, {585, 586, 587, 588, 739, 740, 741, 742, 743, 744, 745}, {589, 590, 591, 592, 593, 594, 595, 596, 597, 746, 747, 748, 749}, {566}, {585, 586, 587, 588, 739}, {902}, {1017}, {972, 973, 974, 975, 976, 977, 978, 979, 991}, {844}, {829}, {569}, {585, 586, 587, 588}, {279, 280, 281, 282, 283, 850, 851, 852}, {589}, {871, 872}, {972, 973, 974, 975, 976, 977}, {885, 886}, {953}, {912}, {611}, {989, 990}, {944}, {787, 788}, {932}, {585, 586, 587, 588, 739, 740, 741, 742, 743}, {585, 586, 587, 588, 739, 740}, {292}, {983, 984}, {905}, {598, 599, 600}, {1013, 1014}, {942}, {112, 113, 114, 610}, {574, 575, 576, 577, 578, 579}, {911, 912}, {110, 111, 112, 113, 114, 610}, {972, 973, 974, 975}, {839}, {1007, 1008, 1009, 1010, 1011, 1012, 1017, 1018}, {1013}, {589, 590, 591, 592, 593, 594, 595, 596, 597, 746, 747}, {952}, {730, 731}, {553, 554}, {951}, {868, 869}, {658, 659}, {763}, {824, 825}, {1017, 1018}, {835, 836, 837, 1007, 1008, 1009, 1010, 1011, 1012, 1017, 1018}, {957}, {850, 851, 852}, {580, 581, 582, 583, 584, 890, 891}, {713, 714, 715, 716, 717, 718, 719, 855, 856, 857}, {730}, {918}, {1034, 1035, 1036}, {701, 702, 703, 704, 705, 706}, {895}, {328}, {895, 896}, {851, 852}, {730, 731, 732, 733, 734}, {703, 704, 705, 706}, {872}, {874, 875}, {611, 612}, {713, 714, 715}, {533, 534}, {885}, {145}, {577, 578, 579}, {282, 283}, {329}, {837}, {588}, {713, 714, 715, 716, 717, 718, 719, 853, 854, 855, 856, 857}, {674}, {907, 908, 909, 910}, {972, 973, 974}, {713, 714}, {849}, {257}, {270}, {894}, {256, 257, 465, 506, 507, 508, 509}, {137, 138, 139, 140, 141, 142, 189, 190, 191, 845, 846, 1038, 1039}, {997, 998, 999}, {908, 909, 910}, {576, 577, 578, 579}, {756}, {937}, {883}, {402}, {585, 586, 587, 588, 739, 740, 741}, {111, 112, 113, 114, 610}, {76, 77, 78, 598, 599, 600}, {960}, {831, 832, 833}, {589, 590, 591}, {885, 886, 887, 888, 889}, {648}, {869}, {291, 292, 713, 714, 715, 716, 717, 718, 719, 853, 854, 855, 856, 857}, {713}, {926},

{561, 562, 563, 1015}, {838, 839}, {192, 193}, {571}, {598, 599}, {401, 402}, {522}, {852}, {675}, {464, 981, 982}, {532, 533, 534}, {589, 590, 591, 592, 593, 594, 595, 596, 597, 746, 747, 748, 749, 750, 751, 885, 886, 887, 888, 889}, {570}, {256, 257}, {579}, {787, 788, 789}, {585, 586, 587, 588, 739, 740, 741, 742}

D.5.2 Stoer-Wagner 算法

{589}