Truss Decomposition in Massive Networks

1. INTRODUCTION

1.1 Truss Decomposition Problem

对于所有的 k，找出 G 中的所有非空 k-truss 。

1.2 k-core

给定一个图 G，其 k-core 是 G 的最大子图，其中每个顶点在子图中的度至少为 k。

1.3 k-truss

给定一个图 G，其 k-truss 是 G 的最大子图，其中 **每条边都包含在至少 （$k-2$） 个三角形中。**

k-truss 是一种网络中的 **凝聚子群** cohesive subgraphs (or cohesive groups) 。

1.4 k-truss 和 k-core 的联系

k-truss 是一个 $(k-1)$-core，但反之不成立。

$(k-1)$-core 包含了 $k-$ truss 和团（clique）等其他凝聚子群，但是为了更加高校的网络分析，其依旧包含了很多可以被过滤掉的无用的信息。

k-truss 相比于 k-core 是更加严格的定义，其基于网络中基本的构建块，也就是三角形。三角形意味着两个点有一个公共的邻接点。

2. PROBLEM DEFINITION

2.1 图的表示方式

邻接表。每个点都有独立的编号（ID）。

2.2 概念及定义

2.2.1 基本概念

- $G = (V_G, E_G)$:

一个无向无权简单图 G，其中 V_G 表示点集，E_G 表示边集。

$n = |V_G|$ 即 G 的节点数量，$m = |E_G|$ 即 G 的边的数量。G 的大小为 $|G| = m + n$。

- $nb(v)$:

  表示节点 v 的所有邻接节点（邻居）构成的集合。即 $nb(v) = u : (u, v) \in E_G$。

- $deg(v)$:

  表示节点 v 的度数。即 $deg(v) = |nb(v)|$。

- $\triangle_{uvw}$:

  表示点 u, v, w 三点构成的一个三角形（三元环）。

  G 中的所有三角形构成的集合记为 $\triangle_G$。

2.2.2 定义 1 **(**support**)**

对于 G 中的一条边 $e = (u, v) \in E_G$，其 support 用 $sup(e, G)$ 表示。定义为 $| \triangle_{uvw} : \triangle_{uvw} \in \triangle_G |$，即 G 中包含边 $e = (u, v)$ 的三角形个数。

2.2.3 定义 2 **(**k-truss**)**

G 中的 k-truss ($k \geq 2$) 用 T_k 表示。定义为 G 的最大子图，满足 $\forall e \in E_{T_k}, sup(e, T_k) \geq (k-2)$，即对于 T_k 中的任意一条边 e，其在 T_k 中的 support 值 $sup(e, T_k)$ 都 **大于等于** $k - 2$。

显然根据定义， 2-truss 就是 G 本身。

$truss\ number$（$trussness$）：对于 G 中的一条边 e，其 $trussness$ 用 $\phi(e)$ 表示，定义为 $\max k : e \in E_{T_k}$，即 **最大的 使得边** e **包含于** T_k **中的** k **值**， 也就是说这个 k 满足存在 $e \in E_{T_k}$，且 $e \notin E_{T_{k+1}}$。

$k_{\max}$ 表示 G 中所有的边的 $truss\ number$ 的最大值。

2.2.4 定义 3 **(**k-class**)**

G 的 k-class 用 Φ_k 表示，定义为 $e : e \in E_G, \phi(e) = k$，即 G 中 $trussness$ 为 k 的边 e 构成的集合。

2.2.5 Truss Decomposition Problem

对于所有的 k，找出 G 中的所有非空 k-truss 。其中 $2 \leq k \leq k_{\max}$。

k-truss 包含了 $trussness$ **至少** 为 k 的所有边，即 $E_{T_k} = \bigcup_{j \geq k} \Phi_j$。

3. IN-MEMORY TRUSS DECOMPOSITION

3.1 Truss Decomposition 算法

3.1.1 算法流程

Algorithm 1 *Truss Decomposition*

Input: $G = (V_G, E_G)$
Output: the k-truss for $3 \leq k \leq k_{max}$

1. $k \leftarrow 3$;
2. **for** each $e = (u, v) \in E_G$ **do**
3. $sup(e) = |nb(u) \cap nb(v)|$;
4. **while**($\exists e = (u, v)$ such that $sup(e) < (k - 2)$)
5. $W \leftarrow (nb(u) \cap nb(v))$;
6.   **for** each $e' = (u, w)$ or $e' = (v, w)$, where $w \in W$, **do**
7. $sup(e') \leftarrow (sup(e') - 1)$;
8.   remove e from G;
9. output G as the k-truss;
10. **if**(*not* all edges in G are removed)
11. $k \leftarrow (k + 1)$;
12.   **goto** Step 4;

1. 计算 G 中每条边的 $support$。边 $e = (u, v)$ 的 $nb(u)$ 和 $nb(v)$ 的交集大小即为 $support$。
2. 初始 $k = 3$，每一轮去除当前所有 $support < k - 2$ 的 $e = (u, v)$。在去除了 e 之后， 包含 (u, v) 的三角形也不复存在，因此需要更新所有 (u, w) 和 (v, w) 的 $support$，其中 $w \in W = (nb(u) \cap nb(v))$。直到剩下的边都是 $support \geq k - 2$ 的，这些边就是 k-truss 所有的边。
3. 如果当前 G 还存在一些边没有被去除，进行 $k += 1$，继续重复上面的步骤，计算下一个 k-truss。

3.1.2 关键步骤实现

- **获取 $support < k - 2$ 的边**

  类似于 BFS 的做法。利用队列 $queue$ 来预先存储当前的 $support < k - 2$ 的边。通过当前队首的边 (u, v) 进行更新 (u, w) 和 (v, w) 后，判断一下更新的边是否也已经满足 $support < k - 2$ 并且还未存入队列中，如果满足此情况，则入队。直到最后队列为空，表明此时 G 中所有边都满足 $support \geq k - 2$。

- **记录被去除的边**

可以直接定义一个 $removed$ 数组，记录对应编号的边的状态。这样每次删边只要改变一下状态就行，每次删边就是 $O(1)$ 的时间复杂度。并没有实质上删除图中的边。

3.1.3 复杂度

- **空间复杂度**

  存储完整的图需要 $O(m+n)$ 的存储空间。

- **时间复杂度**

  主要的时间都消耗在计算 $e=(u,v)$ 的邻居的交集 W 上。此算法需要遍历所有边并计算其 W 来统计三角形以及更新 $support$，因此时间复杂度为 $O(\sum_{v\in V_G}(deg(u)+deg(v)))=O(\sum_{v\in V_G}(deg(v))^2)$。

  对于节点度数较高的图，时间复杂度较高。

3.2 Improved Truss Decomposition 算法

3.2.1 算法流程

Algorithm 2 *Improved Truss Decomposition*

Input: $G=(V_G,E_G)$
Output: the k-class, Φ_k, for $2\le k\le k_{max}$

1. $k\leftarrow 2,\ \Phi_k\leftarrow\emptyset$;
2. compute $sup(e)$ for each edge $e\in E_G$;
3. sort all the edges in ascending order of their support;
4. **while**($\exists e$ such that $sup(e)\le(k-2)$)
5.   let $e=(u,v)$ be the edge with the lowest support;
6.   assume, w.l.o.g., $deg(u)\le deg(v)$;
7.   **for** each $w\in nb(u)$ **do**
8.   **if**($(v,w)\in E_G$)
9. $sup((u,w))\leftarrow(sup((u,w))-1)$,
            $sup((v,w))\leftarrow(sup((v,w))-1)$;
10.   reorder (u,w) and (v,w) according to
            their new support;
11. $\Phi_k\leftarrow(\Phi_k\cup\{e\})$;
12.   remove e from G;
13. **if**(*not* all edges in G are removed)
14. $k\leftarrow(k+1)$;
15.   **goto** Step 4;
16. **return** Φ_j, for $2\le j\le k$;

1. 采用一种 $O(m^{1.5})$ 的算法统计三角形，从而得到每条边的 $support$ 值。Main-memory Triangle Computations for Very Large (Sparse (Power-Law)) Graphs

2. 采用 bin sort 根据 $support$ 对边进行排序，放入到一个有序数组。同时，需要记录每个 bin 的大小求得起始位置，以及每个边所在的位置。时间复杂度为 $O(m)$。An O(m) Algorithm for Cores

3. 从小到大遍历每一条边，每次删除 $support$ 最小的边，其 $support \leq k-2$，因此同时要加入到 $\Phi(k)$ 中，也就是 k-class 中。删除边 $e=(u,v)$ 后会对所属三角形造成影响，此时继续更新受影响边的 $support$ 值，并且修改边在有序数组中的位置。时间复杂度为常数级。

4. 删除边当前的 e，指针 p 遍历过这条边之后就相当于删去了这个最小 $support$ 的边。（在代码中在此基础上，还是实质上删除了边，采用 map 模拟的邻接表删除一条边时间复杂度为 $O(n)$）。

5. 直到最后所有边都被删去，输出所有的 k-class。

3.2.2 复杂度

- **时间复杂度**

  主要计算时间消耗在 triangle listing 上。在枚举三角形时，只需要遍历其中一个节点的邻居即可。设 $deg(u) \leq deg(v)$，我们选择枚举节点度数较小的 u 的邻居，因此最多 $deg(u)$ 次。

  设 $nb_{\geq}(u) = v : v \in nb(u), deg(v) \geq deg(u)$（在计算 $support$ 时，根据节点度数排序，并且将比 u 度数大的点放入到其邻居中）。对于每个 u，最多访问 $nb_{\geq}u$ 次。因此，这部分最多执行 $(deg(u) \cdot |nb_{\geq}(u)|)$ 次。

  $|nb_{\geq}(u)| \leq 2\sqrt{m}$。如果 $deg(u) \leq \sqrt{m}$，然而 $|nb_{\geq}(u)| \leq deg(u) \leq 2\sqrt{m}$；如果 $deg(u) > \sqrt{m}$，此时可以用反证法：假设 $|nb_{\geq}(u)| > 2\sqrt{m}$，那么 $\sum_{v \in nb_{\geq}(u)} deg(v) \geq (|nb_{\geq}(u)| \cdot deg(u)) > (2\sqrt{m}\sqrt{m}) = 2m$。然而这与 $\sum_{v \in V_G} deg(v) = 2m$ 是违背的。

  因此，这部分总的消耗时间为 $\sum_{v \in V_G}(deg(u) \cdot |nb_{ge}(u)|) \leq \sum_{u \in V_G}(deg(u) \cdot 2\sqrt{m}) = (m \cdot 2\sqrt{m}) = O(m^{1.5})$。

- **空间复杂度**

  存储整个图需要 $O(m+n)$ 的空间。bin sort 需要 $O(m)$ 的空间。