Truss-based Community Search

丁霄汉 2017312365 吴超月 2017213865

1. 问题定义

基于论文 Truss-based Community Search: a Truss-equivalence Based Indexing Approach.（https://dl.acm.org/citation.cfm?id=3137640）提出的算法进行社区搜索。作为众所周知的社区检测问题的一个查询依赖变体，社区搜索就是在图中找到涉及给定查询顶点（或一组查询顶点）的内聚和密集连接的子图。

在本文中关注的网络图种类为无向无权无属性图，该社区搜索方法基于 truss。给定一个图 G，一个 k-truss $(k \geq 2)$ 就是图 G 中的的一个最大子图，并且这个子图中的每一条边都至少包括在 $(k-2)$ 个三角形中。由于 k-truss 得到的图可能不是连通图，所以在 k-truss 上加了 triangle-connectivity 约束来定义 k-truss Community。本文的社区搜索问题就是找到搜索节点的所有 k-truss Community。

2. 算法描述

算法的总体思路是在原始图 G 上建立一个索引 EquiTruss，然后 k-truss Community 的搜索可以只在索引上搜索，而不用访问原图 G。这涉及到两个部分：建立索引的过程和在索引上搜索的过程。

1. 建立索引。

首先介绍几个概念：

- Edge Trussness：包含这条边的子图具有的最大的 truss；
- k-triangle：三条边的 Edge Trussness 都大于等于 k 的三角形；
- k-triangle connectivity：两个三角形由多个 k-triangle 连接到一起；
- k-truss equivalence：两条边 Edge Trussness 均为 k，且两条边所在的三角形 k-triangle connectivity。

我们根据 k-truss equivalence 等价关系来建立索引，G 的每条边可以唯一的映射到 index 的一个等价类中。然后我们可以把索引定义为图 $\mathcal{G} = (\mathcal{V},\mathcal{E})$，其中节点 $\nu \in V$，ν 代表一个等价类，而边 $(\mu,\nu) \in \mathcal{E}$，代表两个等价类中各存在一条边，这两条边具有 triangle connectivity 的关系。

文中的算法一首先计算图 G 中每条边的 Edge Trussness，Edge Trussness 是利用 edge support（每条边被多少个三角形包含）来计算的，k 初始化为 2，根据 edge support 从小到大遍历每条边，将 k 赋给当前 edge support 最小的边，然后将与此条边构成三角形的所有边

的 edge support 减一，然后将边重新排序。计算 edge support 的时间复杂度为：$O(|E|^{1.5})$,而总的时间复杂度为：$O(|E|^{1.5})$，空间复杂度为：$O(|V|+|E|)$。算法具体如下：

Algorithm 1: Truss Decomposition

Input: A graph $G(V_G, E_G)$
Output: The edge trussness $\tau(e)$ for each $e \in E_G$

```
1  Computer sup(e) for each edge e ∈ E_G;
2  Sort all edges in the non-decreasing order of their support;
3  k ← 2;
4  while ∃e ∈ E_G, sup(e) ≤ (k − 2) do
5      e*(u, v) ← arg min_{e∈E_G} sup(e);
6      assume w.l.o.g. d(u) ≤ d(v);
7      foreach w ∈ N(u) and (v, w) ∈ E_G do
8          sup(u, w) ← sup(u, w) − 1;
9          sup(v, w) ← sup(v, w) − 1;
10         Reorder (u, w) and (v, w) w.r.t. new edge support;
11     τ(e*) ← k, remove e* from E_G;
12 if ∃e ∈ E_G then
13     k ← k + 1;
14     goto Step 4;
15 return {τ(e)|e ∈ E_G};
```

在计算了所有边的 Edge Trussness 之后，我们可以将边按照 Edge Trussness 的值分类，Edge Trussness 大小相同的边在同一个类里面。进一步的，我们可以计算 index，也就是得到图 $\mathcal{G}=(\mathcal{V},\mathcal{E})$。我们按照 k 从小到大遍历每个类里的每条边。我们利用一条边初始化节点$\nu \in V$并将其压入队列 Q，每次从 Q 中取一条边 e 加入ν，我们访问与边 e 构成三角形的所有边 e’，如果这条边的 Edge Trussness 为 k，我们将其加入队列，如果小于 k，不用处理，如果大于 k，将ν的 id 加入 e’的 list，用于建立$\mathcal{E}$。算法二的时间复杂度为：$O(|E|^{1.5})$,空间复杂度为：$O(|E|)$。具体算法如下：

Algorithm 2: Index Construction for EquiTruss

Input: A graph $G(V_G, E_G)$
Output: EquiTruss: $\mathcal{G}(\mathcal{V}, \mathcal{E})$

```
   /* Initialization */
1  Truss Decomposition (G);
2  foreach e ∈ E_G do
3      e.processed ← FALSE;
4      e.list ← ∅;
5      if τ(e) = k then
6          Φ_k ← Φ_k ∪ {e};
7  snID ← 0 ;          /* Super-node ID initialized to 0 */
   /* Index Construction */
8  for k ← 3 to k_max do
9      while ∃e ∈ Φ_k do
10         e.processed = TRUE;
11         Create a super-node ν with ν.snID ← ++snID;
12         V ← V ∪ {ν} ;      /* A new super-node for C_e */
13         Q.enqueue(e) ;
14         while Q ≠ ∅ do
15             e(u, v) ← Q.dequeue();
16             ν ← ν ∪ {e} ;      /* Add e to super-node ν */
17             foreach id ∈ e.list do
18                 Create a super-edge (ν, μ) where μ is an
                     existing super-node with μ.snID = id;
19                 E ← E ∪ {(ν, μ)} ;      /* Add super-edge */
20             foreach w ∈ N(u) ∩ N(v) do
21                 if τ(u, w) ≥ k and τ(v, w) ≥ k then
22                     ProcessEdge(u, w);
23                     ProcessEdge(v, w);
24             Φ_k ← Φ_k − {e}; E ← E − {e};
25 return G(V, E);
```

```
26 Procedure ProcessEdge(u, v)
27 if τ(u, v) = k then /* k-triangle connectivity */
28     if (u, v).processed = FALSE then
29         (u, v).processed = TRUE;
30         Q.enqueue(u, v);
31 else /* τ(u, v) > k */
32     if snID ∉ (u, v).list then
33         (u, v).list ← (u, v).list ∪ {snID};
```

2. 利用索引来搜索 k-truss community，算法 3 是社区搜索的过程，非常简单且直观，毕竟我们已经得到了等价类，作者直接选择用证明来说明算法的正确性。在算法 3 中，时间复杂度仅仅被搜索到的社区集合的大小决定，具体算法如下：

Algorithm 3: Community Search Based on EquiTruss

Input: $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the truss value $k \geq 3$, the query vertex q
Output: $\mathcal{A}$: all k-truss communities containing q

```
/* Initialization */
1  foreach ν ∈ V do
2      ν.processed ← FALSE;
3  l ← 0 ;
/* BFS traversal for community search */
4  foreach ν ∈ H(q) do
5      if τ(ν) ≥ k and ν.processed = FALSE then
6          ν.processed ← TRUE;
7          l ← l + 1;    A_l ← ∅;
8          Q ← ∅;    Q.enqueue(ν);
9          while Q ≠ ∅ do
10             ν ← Q.dequeue();
11             A_l ← A_l ∪ {e|e ∈ ν};
12             foreach (ν, μ) ∈ E do
13                 if τ(μ) ≥ k and μ.processed = FALSE then
14                     μ.processed ← TRUE;
15                     Q.enqueue(μ);
16 return {A_1, ..., A_l};
```

3. 实验结果与分析

我们用 Java 语言实现了上述算法，并在四个数据集上进行了实验。

实验环境：

处理器:	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz 3.60 GHz
已安装的内存(RAM):	16.0 GB (15.9 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器

```
C:\Users\Administrator>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

数据集：

四个来自于真实世界的数据集，数据集的基本情况见下表。其中 $d_{\max}$ 表示图中点的最大度数。本实验所利用的数据集均可从 http://snap.stanford.edu/data/index.html 获得。

Network	$\lvert V_G \rvert$	$\lvert E_G \rvert$	$d_{\max}$
Amazon	334863	925872	549

DBLP	317080	1049866	343
YouTube	1134890	2987624	28754
LiveJournal	399762	34681189	14815

3.1 索引构建

实验从对给定图建立索引开始，这个过程通常在进行社区搜索前完成。我们将建立好的索引保留在主存中，来作为后续进行大型图表社区搜索的有效工具。在实验中主要对索引构建两个方面的指标进行衡量：（1）索引构建的时间（2）构建的索引的大小。相关实验结果如下表所示：

Graph	Graph size (MB)	Index space (MB)	Construction time (Sec.)
Amazon	12	6.4	1.5
DBLP	13.2	6.92	2
YouTube	36.9	18.3	8.4
LiveJournal	478	312	366.3

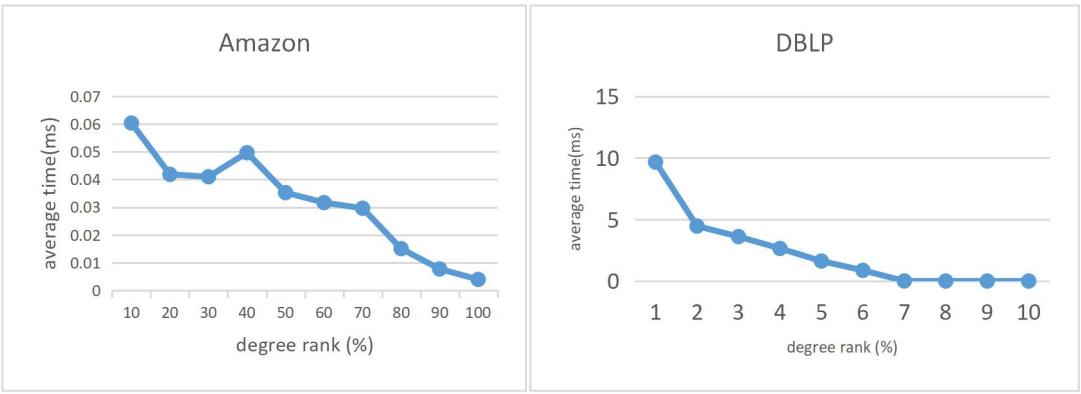
从表中可以看出，随着图大小的增加，构建的索引大小和所需时间也随之增加，但是索引的大小均比原图要小，这是因为在索引中没有存储冗余的信息。

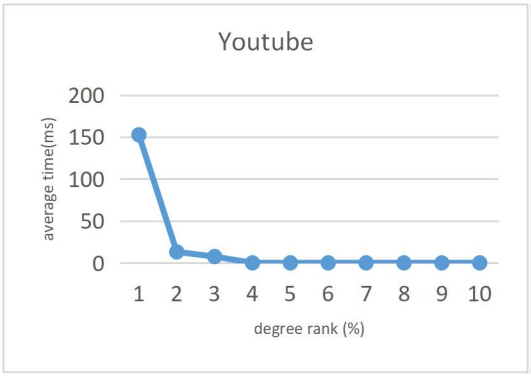
3.2 社区搜索

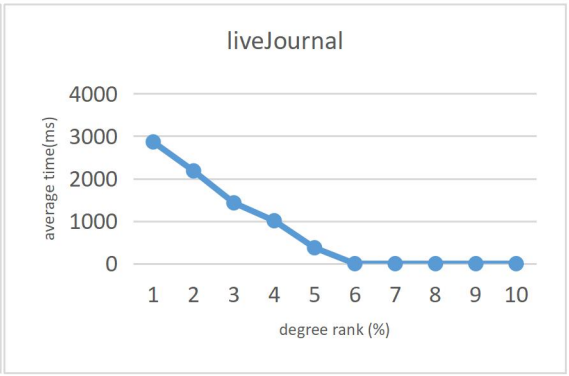
一旦建立好图对应的索引后，就可以利用这些索引进行社区搜索。在实验中考虑两种不同的设置：（1）查询具有不同的度数的点，即考察点的度数对社区搜索运行效率的影响情况（2）考察所设置的 truss k 的不同对社区搜索运行效率的影响情况。

3.2.1 变化点的度数

对每个数据集的点按照度数降序排序，并按照度数将这些点分成十份，使得第一份数据中的点拥有最高的度数。然后从十份数据中各随机选择 100 个点进行社区搜索，并计算每份数据中单个点的平均搜索时间。实验在 Amazon、DBLP、YouTube、LiveJournal 四个数据集上进行，其中 Amazon 数据集中设置的 truss k 为 4，DBLP 中的 truss k 为 5，YouTube 的 truss k 设置为 4，LiveJournal 的 truss k 为 6。实验结果如下所示。



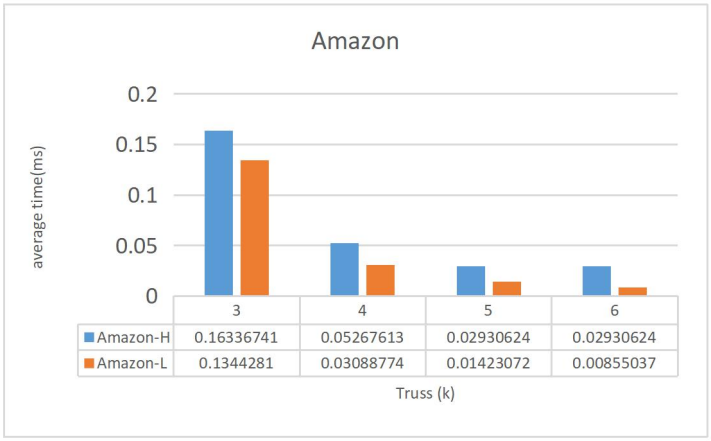


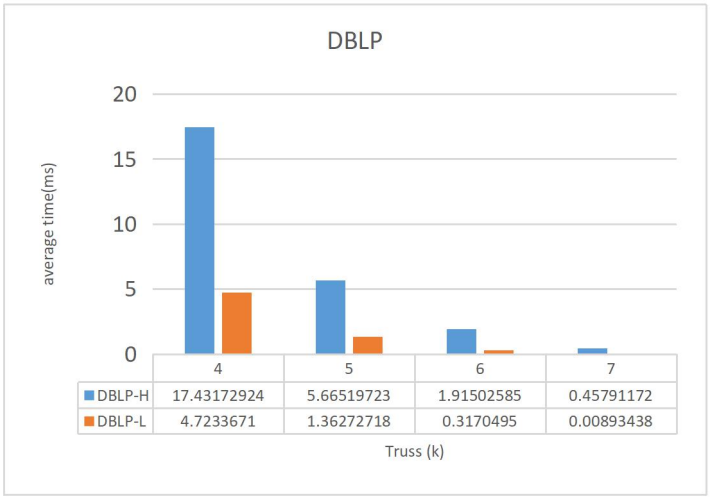


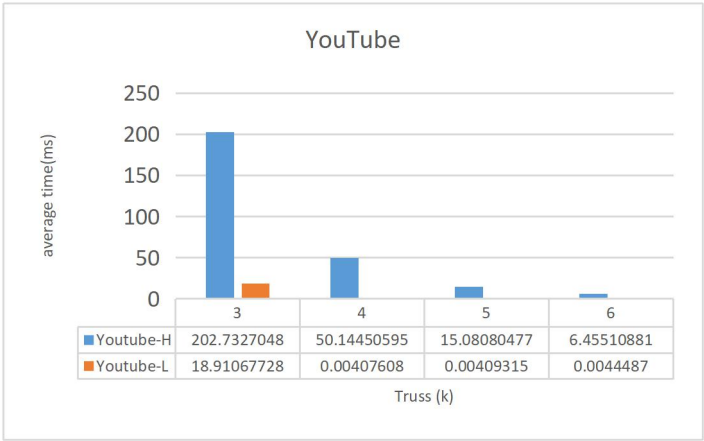
从折线图中可以看出，对拥有较高度数的点进社区搜索所需的时间要明显高于度数较低的点。特别的，在 YouTube 数据集中，随着点度数的降低，所需的时间迅速减少，这是因为对于那些度数少的点来说，基本上很少存在对应的 k-truss 社区。

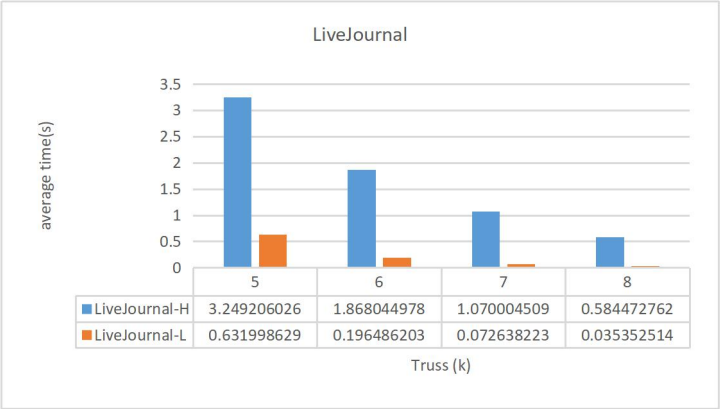
3.2.2 变化 truss k

对每个数据集的点按照度数降序排序，并按照度数将这些点分成十份，使得第一份数据中的点拥有最高的度数。然后从中选取两个待测数据集：（1）从度数在前 30%的点中任意选取 100 个点（2）从后 70%的点中任意选取 100 个点，对这些点进行社区搜索并计算每份数据中单个点的平均搜索时间。实验在 Amazon、DBLP、YouTube、LiveJournal 四个数据集上进行，实验结果如下所示。





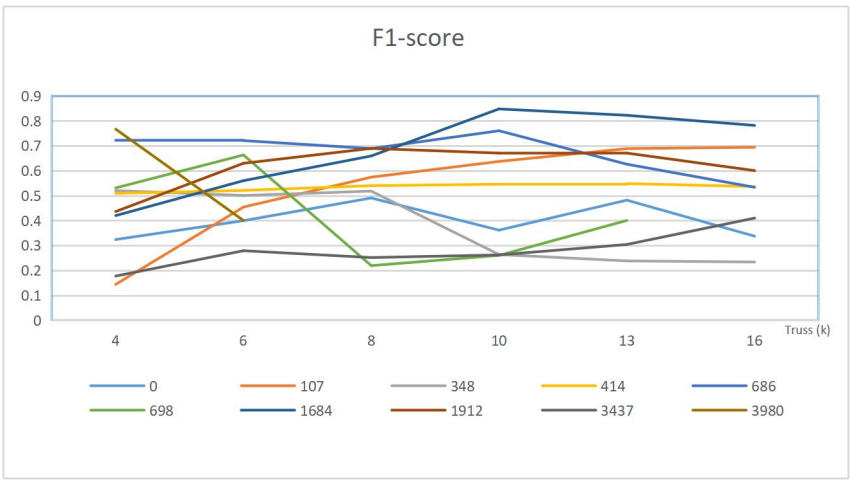


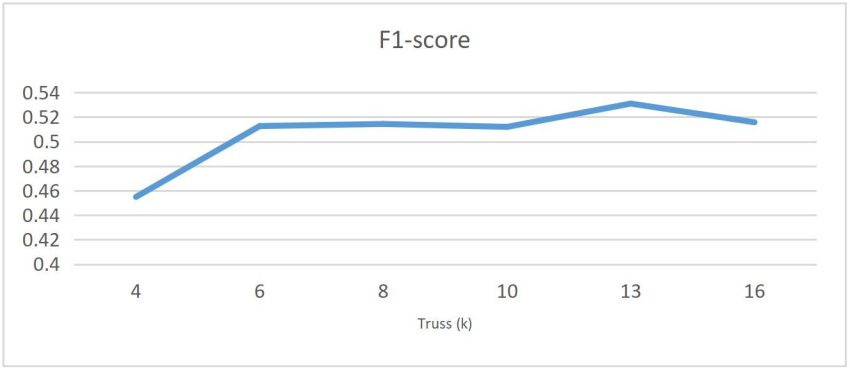


从上面的数据图表中可以看出，随着 truss k 的增加，所需要的搜索时间越来越少，这种情况在大型图中的搜索边的尤其显著。并且在每组实验中，拥有较高度数的点对应的社区搜索时间也越长，这与我们上一个实验的结论保持一致。

3.3 质量评估

在评价 k-truss community model 的有效性的时候，我们采用了 Facebook 数据集，因为这个数据集无向无环无属性，并且为单个点提供 ground truth，Facebook 数据集为 10 个点分别提供了它们所在的所有社区，我们可以直接利用我们得到的社区集合与点本身的社区集合计算 F1-Score，社区集合的 F1-Score 计算公式为：$\max_{f:C\to\hat{C}} \frac{1}{|f|}\sum_{C\in dom(f)} F_1(C,f(C))$。下面分别展示在 Facebook 数据集上进行社区搜索的 10 个点（用点的 index 表示）不同 k 下各自和平均的 F1-score。可以看出总体的 F1-score 在 0.45 以上，随着 k 的不同，F1-score 基本保持在一个稳定的数值，浮动较小。





4. 总结

该论文提出的基于 truss 的社区搜索是一个确定性的算法，不像有些解决方案属于近似算法。在建立索引后，节点的社区搜索时间复杂度非常低，这适合与图更新较少且搜索较多的情况。并且 k 值是可以设置的，这在宽泛意义上使得使用者对得到社区的紧密程度可调。此外，如果网络图存在边的增加和删除操作时，文章给出的方法较先前的工作可以进行有效的更新。但是，本篇工作查询节点的时候需要设置 k 值，虽然作者解释说 k 值的大小一定程度上反映着得到的社区的紧密程度，这的确在更宽泛的意义上是 make sense 的，但是当我们将这个算法用于真正的社区搜索的时候，我们发现 k 值会直接影响搜索的好坏程度，并且针对每个节点的好的 k 值都可以不一样，好的 k 值也没有肉眼可见的规律可调，这使得这个算法的真实的使用性降低了。