Hypercore Maintenance in Dynamic Hypergraphs

问题：**However, the exponential number of hyperedges incurs unaffordable costs to recompute the hypercore number of vertices and hyperedges when updating a hypergraph.**
超边指数数，对重新计算顶点和超边的超核数，造成不可承受的代价

思路：新的超核维护方法，减少超核更新的时间
This motivates us to propose an effificient approach for exact hypercore maintenance with the intention of signifificantly reducing the hypercore updating time comparing with recomputation
approaches.

只需要遍历一个小子超图就可以精确地指出超核数需要更新的顶点和超边
最后是实验，在时间超图和现实世界有效

简单来说，我们所熟悉的图而言，它的**一条边**（edge）只能连接**两个顶点**（vertice）；而超图，人们定义它的**一条边**（hyperedge）可以和**任意个数的顶点**连接。图与超图的对比示意图如下：

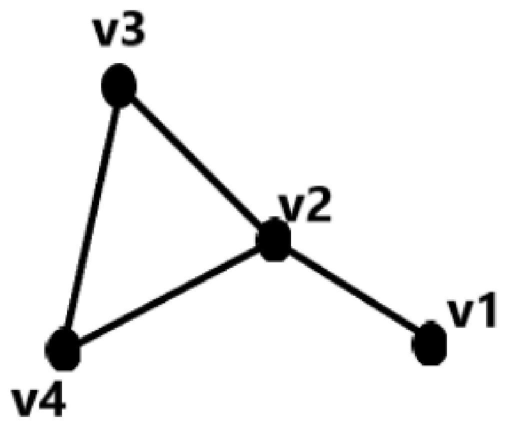

图1 常见的图

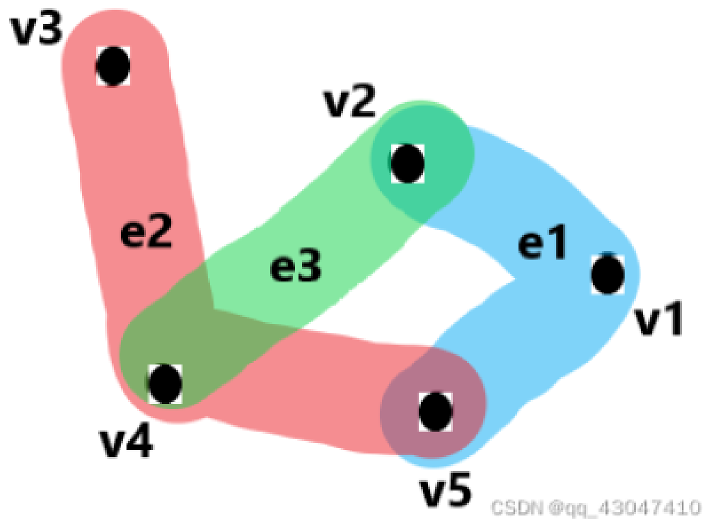

图2 超图

二、k-均匀超图（k-uniform hypergraph）

从超图中引申出来的一个概念就是**k-均匀超图（k-uniform hypergraph）**，它是指超图的每条边连接的顶点数相同，个数都为k。其中的2-均匀超图就是我们平常所见到的传统意义上的图。所以图是超图的一种特殊类型。

超图的应用：social networks, recommendation, knowledge graph, bioinformatics, VLSI(超大规模集成电路）, multimedia ecommerce, learning tasks based on hypergraphs in
cluding clustering [9], classifification [10], [11] and hyperedge prediction [12], [13].

Introduction

问题1：图中通常用点和边描绘成对关系，但忽略了多元关系的映射，导致在描述组和协同者时会缺少信息

问题2：在图的任务重最重要的是挖掘密集子图和许多相关的密集子图

定义名称	简单描述
average degree	密集子图的边数与节点数的比值来度量子图密度
k-core	密集子图*S*中任意一个节点都至少有*k*个邻居
kd-clique	密集子图*S*的直径小于等于*k*，即*S*中任意两节点的最短路径都小于等于*k*
Quasi-Clique (density-based)	密集子图*S*中应该包含至少r*S（S-1）/2 条边(r为定义的值)

Introduction

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

k-truss：束，G的最大子图，其中每条边至少包含在（k- 2）子图中的三角形。

k-clique.：是G的k个顶点的 集合，使得每对顶点都有一条边。

k-ECC. A k-ECC (k-edge connected component)：G的一个子图，在去掉任何k–1边后，它仍然是连通的。

Coreness of vertices: 顶点的核心 最大的k，使顶点可以在最小度为k的子图中

Core number of vertices：比顶点的度更有作用。
（1）Leng and Sun 提出了基于粗化阶段顶点超核属性的匹配策略？？ 利用核心的全局信息来发展其指导作用，改进了以往基于顶点局部信息的匹配方案
（2）Jiang 边数低于边缘密度阈值时，O（log log n）轮剥离就可以获得得到 empty k-core for r-uniform 超图，其中n是超图的顶点数
（3）Shun 提出了一种有效的超图并行核分解算法

Introduction

Question：　computing the hypercore number is still O(m) m is the number of the hyperedges

Now solution: Sun maintain approximate core values in hypergraphs considering edge information and deletion by adversary

Challenges

terpart in pairwise graphs. The challenges include (1) how to determine the hypercore number change after a graph change; (2) how to reduce the number of hyperedges and vertices traversed in the process of identifying those hyperedges and vertices that change the hypercore number; and (3) how to finally identify the vertices and hyperedges whose hypercore numbers will change.

1. 在经历图更新后确认超核数量
2. 减少图的遍历
3. 如何最终识别出超核数量改变的顶点和超边

Introduction

1. Prove that the hypercore number of every vertex and hyperedge can change at most by 1 after a hyperedge is inserted of deleted.
2. Pre-core and relationship of hypercore number between vertices and hyperedges
3. Support degree to identify the vertices and hyperedges which can change the hypercore number

Contributions

1) We propose the concept of hypercore number on hyperedges, and reveal the relationship of the hypercore number between vertices and hyperedges.
2) We present algorithms for exact hypercore maintenance in large-scale dynamic hypergraphs. Rigorous theoretical analysis ensures that our algorithm can update the hypercore numbers of vertices and hyperedges efficiently.

Preliminaries

$G = (V, E)$

E 为一组超边

e为一条超边，|e|为超边经过的顶点数

e（v）代表v所属的超边

E（v）代表v所属的一组超边集合

$\mathrm{N_G}(v) = \cup_{e \in E}\, e(v)$ 其邻接节点定义为同一超边经过的所有顶点

$\mathrm{d_G}(v)$ 顶点的度定义为其所属的超边数量

Cardinality of e 是其含的顶点数

Definition 1 (***k*-hypercore**): A k-hypercore is a connected maximal sub-hypergraph $H = (V', E')$ of $G = (V, E)$, such that $\forall v \in V'$, $d_H(v) \geq k$.

Preliminaries

coreV（v）： 顶点的超核数为，存在一个k使得有k-hypercore含有v但不存在k+1-hypercore含有v

coreE（e）： 边的超核数为，存在一个k使得e上所有顶点的超核数都不小于k

$$coreE(e) = \min\{coreV(v) : v \in e\}. \tag{1}$$

$$coreV(v) = arg\max_{K \geq 0}\{|\{e : e \in E(v), coreE(e) \geq K\}| \geq K\}. \tag{2}$$

Theoretical Basis

$pre - core\ number$: 新插入的超边的最初的超核数量

只有能够与更新的超边且超核数与更新的超边的pre-core number相等的顶点和超边的超核数才会改变

Support-degree on vertices: to determine the vertices and the hyperedges whose hypercore numbers need to be updated

Definition 4 (**Pre-core number**): After inserting a hyperedge e_0 into hypergraph G, the *pre-core number* of e, denoted by $\overline{coreE}(e)$, is defined as $\overline{coreE}(e) = \min\{coreV(v) : v \in e_0\}$.

Lemma 1: If a hyperedge e_0 is deleted from hypergraph $G = (V, E)$, then the hypercore number of every vertex v and every hyperedge e can decrease by at most 1.

Proof：基本采用了反证法
首先考虑顶点：若删除一条边后 coreV(v)=k-x 则其所属的k-hypercore 含有v
则其 $H' = H \backslash e_0$ 每个点的度最多减少1，则其coreV（v）=k-1 和假设相反

其次考虑超边的变化：
超边数量为边上顶点的最小超核数，由于删除某超边，顶点的超核最多减少1，
则超边最多减少1

Lemma 2: Let $G' = (V, E')$ denote the hypergraph obtained by inserting a hyperedge e_0 into the hypergraph $G = (V, E)$. Then the hypercore number of every vertex v and every hyperedge e in G can increase by at most 1.

Proof：基本采用了反证法
顶点可以通过Lemma1，将假设的增加的删边以后变化来反推

其次考虑超边的变化：和之前一样

Lemma 3: If a hyperedge e_0 is inserted into $G = (V, E)$, the pre-core number will increase by at most 1.

Lemma 4: If a hyperedge e_0 is inserted into $G = (V, E)$, for any vertex $v \in V$, v may increase its hypercore number only if $coreV(v) = \overline{coreE}(e_0)$.

Lemma 5: If a hyperedge e_0 is deleted from $G = (V, E)$, for any vertex $v \in V$, v may decrease its hypercore number only if $coreV(v) = coreE(e_0)$.

Lemma 6: If a hyperedge e_0 is inserted into $G = (V, E)$, for any hyperedge $e \in E$, e may increase its hypercore number only if $coreE(e) = \overline{coreE}(e_0)$.

Lemma 7: If a hyperedge e_0 is deleted from $G = (V, E)$, for any hyperedge $e \in E \setminus \{e_0\}$, e may decrease its hypercore number only if $coreE(e) = coreE(e_0)$.

这里做一个概念性的证明，由于pre-core为一条原来的超边上的所有顶点中，其顶点属于新插入的超边，且顶点超核最小

从插入来看，其一定只能改变coreV（v）=pre-core（e0），因为若其不相等，要么插入超边无影响；要么插入后周围仍有比其更小的顶点来影响其超核大小

超边也是一个道理

Theorem 1: If a hyperedge e_0 is inserted into hypergraph $G = (V, E)$, then only the vertices v and the hyperedges e, which satisfy $coreV(v) = \overline{coreE}(e_0)$, $coreE(e) = \overline{coreE}(e_0)$, and are reachable from e_0 via a path that consists of vertices and hyperedges with hypercore number equal to $\overline{coreE}(e_0)$, may increase the hypercore number.

Proof：证明比较简单，同样是反证法。
首先根据前面的引理很容易得到等式，对于其为相邻路径，则假设有两条路径不连通，插入后必有一个子超图中的顶点度没变，对应其超核数量不变

Theorem 2: If a hyperedge e_0 is deleted from hypergraph $G = (V, E)$, then only the vertices v and hyperedges e, which satisfy $coreV(v) = coreE(e_0)$, $coreE(e) = coreE(e_0)$, and are reachable from e_0 via a path that consists of vertices and hyperedges with hypercore number equal to $coreE(e_0)$, may decrease the hypercore number.

*Definition 5 (**Support Degree**):* The *support degree* of a vertex v, denoted as $sup(v)$, is defined as the number of hyperedges containing v and satisfying $coreE(e) \geq coreV(v)$. Each hyperedge e with $coreE(e) \geq coreV(v)$ is a called a *support hyperedge* of v.

Support Degree的定义是为了维护下面的两个引理，均使用反证法证明。
直观上理解，其Support Degree反应了顶点周围的边有几个可以用来支持其维持现有的coreV，若不满足条件，则需要对coreV进行增删，也就是说，维护Support Degree可以看到哪些需要删或加

Theorem 3:

1) After a hyperedge e_0 is inserted into $G = (V, E)$, where $v \in V$ and $sup(v) \leq coreV(v)$, then $coreV(v)$ will not increase.
2) After a hyperedge e_0 is deleted from $G = (V, E)$, where $v \in V$ and $sup(v) < coreV(v)$, then $coreV(v)$ will decrease by 1.

Algorithm 1: Incremental hypercore maintenance

Input : $G = (V, E)$, coreV, coreE, e_0
Output: $coreV, coreE$

$G \leftarrow G \cup \{e_0\}$;
$k \leftarrow \min\{\text{coreV}(v) : v \in e_0\}$; `// pre-core of` e_0
$\text{coreE}(e_0) \leftarrow k$;
exclude $\leftarrow \emptyset$;
sup $\leftarrow$ ComputeSupport$(G, coreE, coreV, e_0)$;
while $\exists sup(v) \leq k$ **do**
    exclude.$add(v)$;
    foreach $e \in E(v)$ ***and*** $coreE(e) = k$ **do**
        foreach $u \in e$ **do**
            if $sup(u) \neq null$ ***and*** $u \notin$ exclude **then**
                $sup(u) \leftarrow sup(u) - 1$;
foreach v *that* $sup(v) \neq null$ ***and*** $u \notin$ exclude **do**
    foreach $e \in E(v)$ ***and*** $coreE(e) = k$ **do**
        Update coreE(e) by Equation 1;
    $\text{coreV}(v) \leftarrow k + 1$;
return coreV, coreE ;

Algorithm 2: ComputeSupport$(G, coreV, coreE, e_0)$

Input : G, coreV, coreE, e_0
Output: sup

visit $\leftarrow \emptyset$;
sup $\leftarrow \emptyset$;
stack $\leftarrow \emptyset$;
$k \leftarrow \text{coreE}(e_0)$;
foreach $v \in V$ **do** visit$(v) \leftarrow false$;
foreach $v \in e_0$ ***and*** $\text{coreE}(v) = k$ **do**
    stack.$push(v)$;
    visit$(v) \leftarrow true$;
while stack $\neq \emptyset$ **do**
    $v \leftarrow$ stack.$pop()$;
    foreach $e \in E(v)$ **do**
        if $\text{coreE}(e) \geq \text{coreV}(v)$ **then**
            $\text{sup}(v) \leftarrow \text{sup}(v) == null ? 1 : \text{sup}(v) + 1$;
        if $\text{coreE}(e) = k$ **then**
            foreach $u \in e$ **do**
                if visit$(u) = false$ ***and*** $\text{coreV}(u) = k$ **then**
                    stack.$push(u)$;
                    visit$(u) \leftarrow true$;
return sup;

Algorithm 3: Hypercore Decomposition

Input : A hypergraph $G = (V, E)$

Output: Hypercore number of vertices and hyperedges

```
compute d(v) for v ∈ V;
k ← 1;
while G is not empty do
    while ∃v ∈ V such that d(v) ≤ k do
        foreach e ∈ E(v) do
            foreach u ∈ e do
                d(u) ← d(u) − 1;
            delete e from E ;
            coreE(e) ← k;
        delete v from V;
        coreV(v) ← k;
    k ← k + 1;
return coreE, coreV;
```

$G' = (V, E \cup \{e_0\})$ be the new hypergraph

C be the set of hypercore numbers of vertices in G.

V_k E_k 代表超核数为k的集合
d_{max}为顶点的最大度

$\hat{V} = \max_{k \in C}\{V_k\}$ and $\hat{E} = \max_{k \in C}\{E_k\}$.

Theorem 4: Algorithm 1 can correctly update the hypercore numbers of vertices and hyperedges in $O(|\hat{V}| \cdot d_{max} + |\hat{E}|s)$ time, after inserting a hyperedge e_0 into G.

For the running time, there are three processes in the algorithm. At first the hypercore number of each vertex in e_0 is computed, which takes $O(s)$ time. Then identifying the potential vertices using the DFS process takes $O(|\hat{V}| \cdot d_{max} + |\hat{E}|s)$ time, and distinguishing the vertices that will not increase the hypercore number takes $O(|\hat{V}| + |\hat{E}|s)$. Finally, it takes $O(|\hat{V}| + |\hat{E}|)$ time to update the hypercore numbers of vertices and hyperedges. Combining all together, the running time is $O(|\hat{V}| \cdot d_{max} + |\hat{E}|s)$. ■

Experiments

TABLE I
THE STATISTICS OF REAL-WORLD HYPERGRAPHS.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	c_{max}	**Ins.**(ns)	**Del.**(ns)
tags-math	1K	174K	5	0.374	0.313
DAWN	2K	143K	16	0.657	0.448
tags-ask-ubuntu	3K	151K	5	0.254	0.247
NDC-substances	5K	10K	25	0.567	0.121
threads-ask-ubuntu	125K	167K	14	0.163	0.037
threads-math	176K	595K	21	0.383	0.131
coauth-History	1.1M	895K	25	47.101	0.336
coauth-Geology	1.2M	1.2M	25	78.826	9.945

TABLE II
THE STATISTICS OF TEMPORAL HYPERGRAPHS.

Dataset	$\lvert V\rvert$	**#TS.**	**#Uniq.**	**Ins.**	**Del.**
tags-stack-overflow	50K	14.4M	5.6M	45	142
coauth-DBLP	1.9M	3.7M	2.6M	1.5K	5.6K
threads-stack-overflow	2.6M	11.3M	9.7M	1.9K	8.5K

[1]. All programs are implemented in Java and compiled with JDK 8. The evaluations are performed on a machine with Intel Core(TM) i7-7700 CPU and 24GB size memory.