2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00199

Hypercore Maintenance in Dynamic Hypergraphs

1st Qi Luo
Computer Science and Technology
Shandong University
Qingdao, P.R. China
luoqi2018@mail.sdu.edu.cn

2nd Dongxiao Yu
Computer Science and Technology
Shandong University
Qingdao, P.R. China
dxyu@sdu.edu.cn

3rd Zhipeng Cai
Department of Computing Science
Georgia State University
Atlanta, USA.
zcai@gsu.edu

4th Xuemin Lin
Computer Science and Engineering
University of New South Wales
Sydney, Australia
lxue@cse.unsw.edu.au

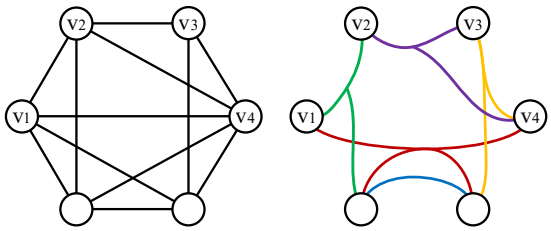
5th Xiuzhen Cheng
Computer Science and Technology
Shandong University
Qingdao, P.R. China
xzcheng@sdu.edu.cn

***Abstract*—In this paper, we study exact hypercore maintenance in large-scale dynamic hypergraphs. A hypergraph, whose hyperedges may contain a set of vertices rather than two vertices in pairwise graphs, can represent complex interactions in more sophisticated applications. However, the exponential number of hyperedges incurs unaffordable costs to recompute the hypercore number of vertices and hyperedges when updating a hypergraph. This motivates us to propose an efficient approach for exact hypercore maintenance with the intention of significantly reducing the hypercore updating time comparing with recomputation approaches. The proposed algorithms can pinpoint the vertices and hyperedges whose hypercore numbers have to be updated by only traversing a small sub-hypergraph. Extensive experiments on real-world and temporal hypergraphs demonstrate the superiority of our algorithms in terms of efficiency.**

***Index Terms*—Graph analytics, hypergraph, cohesive subgraph.**

I. Introduction

Graphs have been utilized as a powerful tool to model pairwise relationships between people or objects, which are extensively employed in various applications such as biology and social network analysis. Most of the applications adopt relational graphs in which only pairwise relationships between two vertices are described, but overlook the widespread usage of polyadic relationships. Polyadic relationships are ubiquitous in real-world scenarios. To name some, a paper may have three or more authors, an online group chat may involve tens of participants, and an email from an organization could be sent to many recipients. Conventional graphs only supporting pairwise relationships would result in information loss when representing groups or collaborators. Hypergraphs [1] are a natural extension of the conventional notion of graphs by allowing various sizes of edges. Formally, a hypergraph consists of a set of vertices and a set of hyperedges, where each hyperedge is a non-empty subset containing any number of vertices. They can represent complex interactions (i.e., interactions among any number of objects instead of only two objects in conventional graphs). To demonstrate this scenario, we take article categorization as an example. Given an article with authors' information, one may construct an undirected graph where the vertices are the authors, and two vertices are connected by an edge if they are co-authors of an article. However, in such a graph, it is hard to identify the authors of a particular article, while with a hypergraph, whose hyperedges represent the authors of a same article, can perfectly represent the collaboration relationships. Fig. 1 shows the differences between a hypergraph and a pairwise graph.



(a) pairwise graph (b) hypergraph

Fig. 1. Comparison of a hypergraph and a pairwise graph. The pairwise graph cannot tell us whether a same person is the author of three or more articles. The hypergraph can completely illustrate the complex relationships among authors and articles.

Taking into account polyadic interactions has been proven to be fruitful and indeed inevitable for challenging tasks. Hypergraphs have attracted much attention in various domains including social networks [2], recommendation [3], knowledge graphs [4], bioinformatics [5], VLSI [6], multimedia [7] e-commerce [8], and learning tasks based on hypergraphs including clustering [9], classification [10], [11] and hyperedge prediction [12], [13].

One of the most important analysis tasks in graphs is to mine cohesive subgraphs, and many related cohesive subgraphs, such as quasi-clique [14], k-core [15], k-truss [16], and corresponding indexes for reflexing cohesiveness, such as coreness and trussness, have been proposed and extensively studied

This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB2102600 and in part by NSFC under Grant 61971269, and Grant 61832012.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00199

[17]–[20]. Among these indexes, the coreness of vertices, which is defined as the maximum k such that the vertex can be in a subgraph whose minimum degree is k, has been the most commonly adopted one, due to its low computation cost [21] and effectiveness in studying the properties of networks, such as efficiently solving NP-hard problems [22] in real networks, and for large-scale network fingerprinting and visualization [23].

The core notation has also been introduced in hypergraph, which was firstly proposed in [24]. It is shown the core number of vertices can better reflect the importance of nodes in a hypergraph than the index of degree. To distinguish k-core in pairwise graphs, we call k-core in hypergraphs as k-hypercore. In [25], Leng and Sun proposed a matching strategy based on the hypercore property of vertices in the coarsening phase, which not only makes use of the global information of the core to develop its guidance role, but also improves previous matching schemes based on local information of vertices. Jiang *et al.* [26] showed that when the number of edges is below an edge density threshold, it only takes $O(\log \log n)$ rounds of peeling to obtain the empty k-core for r-uniform hypergraphs with a high probability, where n is the number of vertices in the hypergraph. Shun [27] presented an efficient parallel core decomposition algorithm in hypergraphs.

Even though the time complexity of computing the hypercore number is still $O(m)$ where m is the number of hyperedges [24], calculating hypercore becomes an exceptionally costly task, as the number of hyperedges can be exponential in the number of vertices. In real-world hypergraphs, there may be tens of millions even billions of vertices. Real-world hypergraphs evolve over time: new scientific publications are continuously added over time and new tweets are posted every second. All these make recomputing hypercore unaffordable. An efficient approach would be updating the hypercore numbers of vertices after every minor change of the hypergraph, referring to a hypercore maintenance algorithm. In [28], Sun *et al.* designed some algorithms to maintain approximate core values in hypergraphs considering edge insertion and deletion by an adversary. However, to the best of our knowledge, there are no existing results on exact hypercore maintenance. Furthermore, different from pairwise graphs, the core number of both vertices and hyperedges have practical significance. Vertices in a hyperedge are organized as a local fully connected community, and the cohesiveness of the hyperedge also needs to be deliberated in hypergraphs.

In this work, we focus on exact hypercore maintenance in large-scale dynamic hypergraphs. Note that due to the complex interactions of hypergraphs, exact hypercore maintenance is much more complex in hypergraphs comparing with its counterpart 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.

To tackle the challenges for hypercore maintenance, we focus on the scenario of single hyperedge insertion/deletion. The hypercore maintenance with multiple hyperedge insertions/deletions can be solved by iteratively invoking the single-hyperedge insertion/deletion algorithms. We prove that the hypercore number of every vertex and hyperedge can change at most by 1 after a hyperedge is inserted or deleted. To undertake the second challenge, we propose necessary conditions to determine the potential vertices and hyperedges that need to be traversed, based on a proposed concept of *pre-core* and a relationship of hypercore number between vertices and hyperedges. These necessary conditions greatly reduce the traversal range during the process of identifying potential vertices and hyperedges that may change the hypercore number. Furthermore, we propose the concept of *support degree* to finally identify the vertices and hyperedges which can change the hypercore number. Our contributions are summarized as follows.

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.

Roadmap. The remaining sections of this paper are outlined as follows: In Section II, the notations and definitions are given. In Section III, the theoretical basis of our algorithms is presented. We introduce our hypercore maintenance algorithms in Section IV. Empirical results on real-world hypergraph datasets and temporal datasets are presented in Section V. We discuss and conclude our work in Section VI.

II. Preliminaries

We formulate the problem in this section. We consider a simple and undirected hypergraph $G = (V, E)$ on a finite set of vertices V, where $E \subset 2^V$ is a set of hyperedges. Each hyperedge $e \in E$ represents a set of $|e|$ vertices that take interaction.

Let $e(v)$ denote a hyperedge e that v belongs to and $E(v)$ denote the set of hyperedges that v belongs to. The neighborhood of a vertex $v \in V$ in a hypergraph consists of all vertices in the hyperedges that contain v, denoted as $N_G(v)$, i.e., $N_G(v) = \bigcup_{e \in E} e(v)$. The hyper-degree of a vertex v, denoted as $d_G(v)$, is defined as the number of hyperedges containing v. When the context is clear, we simply call hyper-degree as degree. The cardinality of a hyperedge e is the number of vertices it contains. We next give the definition of k-hypercore as follows.

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

With the above definition, we can define the *hypercore number on vertices and hyperedges* as follows.

*Definition 2 (**Hypercore number of vertex**):* For a given vertex v in the hypergraph G, the hypercore number of vertex

v, denoted as $coreV(v)$, equals to k if there exists a k-hypercore containing vertex v, but there is not any $(k+1)$-hypercore containing v.

*Definition 3 (**Hypercore number of hyperedge**):* For a given hyperedge e in the hypergraph G, the hypercore number of hyperedge e, denoted as $coreE(e)$, equals to k if the hypercore number of each vertex in e is not smaller than k.

From the definitions of the hypercore number of vertices and hyperedges, it can be concluded that the hypercore number of a hyperedge e and a vertex v satisfy the following equations.

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

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

We consider the hypercore update problem in dynamic graphs, which is formally called the *hypercore maintenance* problem. Specifically, given a hypergraph $G = (V, E)$, the hypercore maintenance problem is to update the hypercore numbers of vertices and hyperedges after some graph changes. We here pay our attention to the scenario of single-hyperedge insertion/deletion from G, which are called *incremental* and *decremental* core maintenance respectively, as multiple changes of hyperedges or vertices can be solved by recursively invoking the single-hyperedge insertion/deletion algorithm.

III. Theoretical Basis

To handle the hypercore number update efficiently after an insertion/deletion of a hyperedge, the key is to identify the affected hyperedges and vertices in the hypergraph precisely and the changed values on hypercore numbers of hyperedges and vertices. Thus, we present theoretical analysis to quantify the hypercore number change and scope potential vertices and hyperedges whose hypercore numbers may change.

We discover that the hypercore numbers of vertices and hyperedges can change by at most 1 after a hyperedge is inserted/deleted. We propose a concept of *pre-core number* as the initial hypercore number of newly inserted hyperedge, and show that only the vertices and hyperedges that are reachable to the updated hyperedge and have the hypercore number equal to the pre-core number of the update hyperedge may change their hypercore numbers, such that the searching range of potential vertices and hyperedges can be greatly reduced. Finally, we define the *support degree* on vertices to determine the vertices and the hyperedges whose hypercore numbers need to be updated.

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: We first consider the vertices. Denote by $G' = G\setminus\{e_0\}$. Assume that there exists a vertex v whose hypercore number decreases by more than 1. Formally, assume that $coreV(v) = k$ and $coreV(v) = k-x$ after deleting hyperedge e_0, where $x \geq 2$. Denote by H the k-hypercore containing v in G. Let $H' = H \setminus e_0$. Notice that after deleting a hyperedge from G, the degree of every vertex in G decreases by at most 1. Hence, for each vertex $u \in H'$, $d'_H(v) \geq k-1$. This means that H' is a $k-1$-hypercore in G' and $coreV(v) = k-1$, which contradicts with the assumption.

For the hypercore change of hyperedges, because the hypercore of a hyperedge is the minimum hypercore number of vertices it contains, and as shown above, the hypercore number of every vertex can decrease by at most 1. Hence, the hypercore number of every hyperedge can also decrease by at most 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: We consider the hypercore number change of vertices. Assume that there exists a vertex v whose hypercore number changes by more than 1. Formally, assume that $coreV(v) = k$ in hypergraph G and $coreV(v) = k + x$ in hypergraph G', where $x \geq 2$. After deleting e_0 from G', the hypercore number of v can decrease by at most 1 by Lemma 1. This means in G, $coreV(v) \geq k+x-1 > k$, which contradicts with the hypercore number of v in G. So the hypercore number of every vertex can increase by at most 1.

The bound on the hypercore number change of edges can be proved similarly as the deletion case in Lemma 1, based on the above result on vertices and the definition of hypercore number. ■

We use an index of pre-core number defined as follows to determine whether a vertex or a hyperedge is potential to change the hypercore number.

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

Proof: The pre-core number of a hyperedge depends on the minimum hypercore number of vertices belonging to the hyperedge. We assume the hyperedge e's pre-core number increases by $x > 1$ after the insertion of e_0. According to the definition of pre-core number of hyperedge, the hypercore number of certain vertex in e must have increased by x. However, the hypercore number of every vertex can increase by at most 1 by Lemma 2, which is a contradiction. ■

With the definition of pre-core number, in the following Lemma 4 and Lemma 5, we give necessary conditions for vertices to change the hypercore.

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 $coreE(e) = coreE(e_0)$.

Based on the above Lemmas, we can finally get a result to depict necessary conditions for identifying potential vertices and hyperedges.

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: According Lemma 4 and Lemma 6, only if the hypercore number equals to the pre-core number of e_0, the hyperedge or vertex's hypercore number may increase. Assume that the vertices and hyperedges whose hypercore numbers increase do not form a connected sub-hypergraph. Then, there are at least 2-non-overlapping sub-hypergraphs of vertices and hyperedges whose hypercore numbers increase, H_1 and H_2. Since there is only one hyperedge insertion, only one of the two sub-hypergraphs can have vertices in the new hyperedge whose degrees can increase. Without loss of generality, we assume this sub-hypergraph is H_1. Clearly, H_2 does not have any change after inserting e_0. Hence, the hypercore number of vertices and hyperedges in H_2 cannot change accordingly. This contradiction completes the proof. ■

Using a similar argument as the insertion case, we can get the necessary condition for the deletion scenario as well.

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.

We finally propose sufficient conditions to determine whether a vertex or a hyperedge can change the hypercore number. At first, a concept of *support degree* is proposed to depict the possibility of hypercore change.

*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.

Based on the definitions of support degree and hypercore number, we can obtain the following sufficient condition for determining whether the hypercore number can change.

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.

Proof: For the case of hyperedge insertion. We prove the theorem by a contradiction. Assume $sup(v) = k$, $coreV(v) = k + 1$ after inserting a hyperedge. There are at least $k + 1$ hyperedges containing v and their hypercore numbers are at least $k+1$ according to Equation (2). However, there are only k hyperedges whose hypercore numbers are at least k according to the definition of support degree, and only these hyperedges can increase the hypercore number to at least $k + 1$. This contradiction completes the proof.

The deletion case can be similarly proved. ■

IV. Hypercore Maintenance

In this section, we present an algorithm for incremental core maintenance. We consider the scenario where a hyperedge e_0 is inserted into a hypergraph $G = (V, E)$. The incremental hypercore maintenance algorithm is given in Algorithm 1.

In the algorithm, the pre-core number of e_0 is first computed (Lines 2-3). All potential vertices that may change the hypercore number, i.e., those whose hypercore numbers are equal to $\overline{coreE}(e_0)$ and which are reachable from e_0 via a path that consists of vertices and hyperedges with hypercore number equal to $\overline{coreE}(e_0)$, are found using a DFS process (Line 5). Then the potential vertices with support degrees not larger than $\overline{coreE}(e_0)$ are identified (Lines 6-11). The hypercore number of these vertices will not increase by Theorem 3. Once a potential vertex is identified as being unable to increase the hypercore number, the support degrees of other vertices reachable from this vertex is updated. Finally, the vertices in sup that are not excluded will increase the hypercore number by 1, while the hyperedges containing these vertices update the hypercore number accordingly (Lines 12-15).

Algorithm 1: Incremental hypercore maintenance

```
   Input  : G = (V, E), coreV, coreE, e_0
   Output: coreV, coreE
1  G ← G ∪ {e_0} ;
2  k ← min{coreV(v) : v ∈ e_0} ;     // pre-core of e_0
3  coreE(e_0) ← k;
4  exclude ← ∅ ;
5  sup ← ComputeSupport(G, coreE, coreV, e_0) ;
6  while ∃sup(v) ≤ k do
7      exclude.add(v) ;
8      foreach e ∈ E(v) and coreE(e) = k do
9          foreach u ∈ e do
10             if sup(u) ≠ null and u ∉ exclude then
11                 sup(u) ← sup(u) − 1;
12 foreach v that sup(v) ≠ null and u ∉ exclude do
13     foreach e ∈ E(v) and coreE(e) = k do
14         Update coreE(e) by Equation 1;
15     coreV(v) ← k + 1 ;
16 return coreV, coreE ;
```

Analysis. To analyze the efficiency of our incremental algorithm, we first introduce some notations to measure the time complexity of our algorithm.

Let $G' = (V, E \cup \{e_0\})$ be the new hypergraph after inserting e_0. Denote by s the maximum cardinality of hyperedges. Let C be the set of hypercore numbers of vertices in G. For each $k \in C$, let V_k and E_k denote the set of vertices and

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

```
Input : G, coreV, coreE, e_0
Output: sup
visit ← ∅ ;
sup ← ∅ ;
stack ← ∅ ;
k ← coreE(e_0) ;
foreach v ∈ V do visit(v) ← false;
foreach v ∈ e_0 and coreE(v) = k do
    stack.push(v);
    visit(v) ← true ;
while stack ≠ ∅ do
    v ← stack.pop() ;
    foreach e ∈ E(v) do
        if coreE(e) ≥ coreV(v) then
            sup(v) ← sup(v) == null?1 : sup(v) + 1 ;
        if coreE(e) = k then
            foreach u ∈ e do
                if visit(u) = false and coreV(u) = k then
                    stack.push(u) ;
                    visit(u) ← true ;
return sup;
```

hyperedges whose hypercore numbers equal to k. Denote by $\hat{V} = \max_{k \in C}\{V_k\}$ and $\hat{E} = \max_{k \in C}\{E_k\}$. Let d_{max} denote the maximum degree of vertices.

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.

Proof: By Theorem 1, 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 $\overline{coreE}(e_0)$. Hence our algorithm can find all potential vertices and hyperedges that may change the hypercore value. Then by Theorem 3, a vertex cannot increase the hypercore number if its support degree is not larger than its hypercore number. By checking and updating the support degrees of vertices, our algorithm can correctly distinguish vertices that do not increase the hypercore number. After this process, all remaining potential vertices constitute a sub-hypergraph in which each vertex are connected to at least $k+1$ hyperedges. This means that all these vertices have a new hypercore number at least $k+1$, which ensures the correctness of our algorithm.

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 $O(|\hat{V}| \cdot d_{max} + |\hat{E}|s)$. ■

Discussion. In our algorithm, we take the hypercore numbers of vertices and hyperedges as input. There have been many recent work studying the hypercore decomposition problem, such as [24], [26]–[28]. However, none of them studies the hypercore number of hyperedges.

We here propose a core decomposition algorithm, as shown in Algorithm 3. In the algorithm, starting from $k = 1$, vertices whose degrees are less than k are repeatedly removed, together with hyperedges containing them. The hypercore numbers of vertices and hyperedges are set to k when they are removed. Different from previous algorithms, the algorithm computes the hypercore number of hyperedges in the decomposition process. The time complexity of the hypercore decomposition is $O(|E|)$.

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;
```

V. Experiments

In this section, we evaluate the proposed hypercore maintenance algorithms on real-world graphs and temporal graphs [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.

Table I shows the statistics of real-world datasets, where c_{max} is the maximum cardinality, **Ins.** and **Del.** are the average time of inserting or deleting one hyperedge. The hyperedge deletion process is faster that the insertion process. Next, we compare the performances of the hypergraph decomposition algorithm and the hypergraph maintenance algorithms by conducting on temporal hypergraphs. In each hypergraph, the timestamped hyperedges are seen as the inserted/deleted ones. In the experiments, the hypercore decomposition algorithms are only executed for once to update the hypercore numbers when a number of hyperedges are inserted/deleted. The experimental results are shown in Table. II. **#TS.** is the number of timestamped hyperedges and **#Uniq.** is the number of unique hyperedges. **Ins.** and **Del.** are the breakpoint of inserting and deleting hyperedges. It can be seen that the break point reaches around 1.5%-2.0% of the total number of hyperedges for all hypergraphs.

[1] All the datasets can be download in http://www.cs.cornell.edu/ arb/data/

TABLE I
THE STATISTICS OF REAL-WORLD HYPERGRAPHS.

Dataset	$\|V\|$	$\|E\|$	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	$\|V\|$	**#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

VI. CONCLUSION

We proposed the algorithms for exact hypercore maintenance in large-scale dynamic hypergraphs. Our algorithms provide an efficient approach to update hypercore numbers of vertices and hyperedges, where only a small sub-hypergraph needs to be traversed for identifying the vertices and hyperedges that have to change the hypercore numbers, avoiding the exceptionally expensive approach of recomputation after every graph change. Extensive experiments demonstrate our algorithms can significantly speed up the hypercore update process. Due to the ubiquitous applications of hypergraphs, studies on hypergraphs have attracted much attention recently. However, due to complex representations and lack of adequate tools, efficient solutions to some fundamental problems in hypergraphs are still not derived. Our work shows that it is possible to design efficient dense subgraph mining algorithms by deeply investigating the structural properties of hypergraphs. Hence, it deserves to making more efforts on mining tasks in hypergraphs.

REFERENCES

[1] P. S. Chodrow and A. Mellor, "Annotated hypergraphs: Models and applications," *Applied Network Science*, vol. 5, p. 9, 2020.

[2] D. Yang, B. Qu, J. Yang, and P. Cudré-Mauroux, "Revisiting user mobility and social relationships in lbsns: A hypergraph embedding approach," in *The World Wide Web Conference, WWW*. ACM, 2019, pp. 2147–2157.

[3] Y. Zhu, Z. Guan, S. Tan, H. Liu, D. Cai, and X. He, "Heterogeneous hypergraph embedding for document recommendation," *Neurocomputing*, vol. 216, pp. 150–162, 2016.

[4] B. Fatemi, P. Taslakian, D. Vázquez, and D. Poole, "Knowledge hypergraphs: Extending knowledge graphs beyond binary relations," *CoRR*, vol. abs/1906.00137, 2019.

[5] T. Hwang, Z. Tian, R. Kuang, and J. A. Kocher, "Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction," in *Proceedings of the 8th IEEE International Conference on Data Mining ICDM*, 2008, pp. 293–302.

[6] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[7] Q. Liu, Y. Huang, and D. N. Metaxas, "Hypergraph with sampling for image retrieval," *Pattern Recognition*, vol. 44, no. 10-11, pp. 2255–2262, 2011.

[8] J. Li, J. He, and Y. Zhu, "E-tail product return prediction via hypergraph-based local graph cut," in *Proceedings of the 24th ACM International Conference on Knowledge Discovery & Data Mining, KDD*, Y. Guo and F. Farooq, Eds. ACM, 2018, pp. 519–527.

[9] J. Huang, R. Zhang, and J. X. Yu, "Scalable hypergraph learning and processing," in *2015 IEEE International Conference on Data Mining, ICDM*. IEEE Computer Society, 2015, pp. 775–780.

[10] D. Zhou, J. Huang, and B. Schölkopf, "Learning with hypergraphs: Clustering, classification, and embedding," in *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems*. MIT Press, 2006, pp. 1601–1608.

[11] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *The Thirty-First Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 2019, pp. 3558–3565.

[12] D. Arya and M. Worring, "Exploiting relational information in social networks using geometric deep learning on hypergraphs," in *Proceedings of the International Conference on Multimedia Retrieval, ICMR*. ACM, 2018, pp. 117–125.

[13] D. Li, Z. Xu, S. Li, and X. Sun, "Link prediction in social networks based on hypergraph," in *22nd International World Wide Web Conference, WWW Companion Volume*. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 41–42.

[14] J. Abello, M. G. C. Resende, and S. Sudarsky, "Massive quasi-clique detection," in *5th Latin American Symposium of Theoretical Informatics Proceedings,LATIN*, ser. Lecture Notes in Computer Science, vol. 2286. Cancun, Mexico: Springer, 2002, pp. 598–612.

[15] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[16] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," in *National Security Agency Technical report*, vol. 16, 2008, pp. 3–29.

[17] Q. Luo, D. Yu, F. Li, Z. Dou, Z. Cai, J. Yu, and X. Cheng, "Distributed core decomposition in probabilistic graphs," in *8th International Conference Computational Data and Social Networks Proceedings*, ser. Lecture Notes in Computer Science, vol. 11917. Ho Chi Minh City, Vietnam: Springer, 2019, pp. 16–32.

[18] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1287–1300, 2020.

[19] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie, "Core maintenance in dynamic graphs: A parallel approach based on matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2416–2428, 2018.

[20] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS*, K. Lee and L. Liu, Eds. IEEE Computer Society, 2017, pp. 2366–2371.

[21] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[22] A. Das, M. Svendsen, and S. Tirthapura, "Incremental maintenance of maximal cliques in a dynamic graph," *VLDB J.*, vol. 28, no. 3, pp. 351–375, 2019.

[23] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Neural Information Processing Systems*, 2005, pp. 41–50.

[24] M. Leng, L. Sun, J. Bian, and Y. Ma, "An $o(m)$ algorithm for cores decomposition of undirected hypergraph," *Journal of Chinese Computer Systems*, vol. 34, no. 11, pp. 2568–2573, 2013.

[25] M. Leng and L. Sun, "Comparative experiment of the core property of weighted hyper-graph based on the ispd98 benchmark," *Journal of Information and Computational ence*, vol. 10, no. 8, pp. 2279–2290, 2013.

[26] J. Jiang, M. Mitzenmacher, and J. Thaler, "Parallel peeling algorithms," *ACM Trans. Parallel Comput.*, vol. 3, no. 1, pp. 7:1–7:27, 2016.

[27] J. Shun, "Practical parallel hypergraph algorithms," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2020, pp. 232–249.

[28] B. Sun, T. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 4, pp. 39:1–39:21, 2020.