# Maximal D-truss Search in Dynamic Directed Graphs

Anxin Tian
The Hong Kong University of Science and Technology
atian@connect.ust.hk

Alexander Zhou
The Hong Kong University of Science and Technology
atzhou@cse.ust.hk

Yue Wang
Shenzhen Institute of Computing Sciences
yuewang@sics.ac.cn

Lei Chen
The Hong Kong University of Science and Technology
leichen@cse.ust.hk

## ABSTRACT

Community search (CS) aims at personalized subgraph discovery which is the key to understanding the organisation of many real-world networks. CS in undirected networks has attracted significant attention from researchers, including many solutions for various cohesive subgraph structures and for different levels of dynamism with edge insertions and deletions, while they are much less considered for directed graphs. In this paper, we propose incremental solutions of CS based on the D-truss in dynamic directed graphs, where the D-truss is a cohesive subgraph structure defined based on two types of triangles in directed graphs. We first analyze the theoretical boundedness of D-truss given edge insertions and deletions, then we present basic single-update algorithms. To improve the efficiency, we propose an order-based D-Index, associated batch-update algorithms and a fully-dynamic query algorithm. Our extensive experiments on real-world directed graphs show that our proposed solution can significantly exceed the SOTA solution by up to 6.89x, the scalability of proposed solutions over updates is also verified.

**PVLDB Reference Format:**
Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. Maximal D-truss Search in Dynamic Directed Graphs. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

## 1 INTRODUCTION

The graph is a concise structure consisting of vertices and edges and is widely used in multiple fields. Community search (CS) is a popular topic of graph processing, which intends to find a specifically connected subgraph for the given query vertices [22]. The community is a concept generated from social networks, while CS works in not only social analysis [1, 5], but also recommendation [3, 12], industry intelligence [30] and bio-medicine [20, 31, 33].

Many works have proposed various solutions based on different metrics [4, 25, 29] or cohesive subgraph structures [7, 9, 40] for solving CS problems. Though CS is well-studied in undirected graphs, it is less considered for directed graphs. For capturing asymmetric relationships between entities in directed graphs, the D-core (or $(k, l)$-core) [17] is proposed to find the community with minimum $k$ in-degree and $l$ out-degree requirements being fulfilled. However, compared with truss-like structures, core-like structures are sparse and unable to capture edge cohesiveness [9].

More generally, a truss-like structure can alleviate such limitations in directed graphs. The D-truss [28] is a cohesive subgraph structure which is defined based on two types of triangles in directed graphs. Different from undirected graphs, triangles can be classified into flow triangles and cycle triangles according to directions of edges among the three vertices [36]. Figure 1 (a) shows an example of directed triangles, where $v_2, v_3$ and $v_4$ form a cycle triangle, $v_2, v_5$ and $v_6$ form a flow triangle. It is worth to mention that the same three vertices can be a part of multiple triangles, e.g. $v_2, v_3$ and $v_4$ also form a flow triangle since $v_2$ and $v_4$ are bi-directionally connected. We can find a hub vertex in a flow triangle where the other two vertices always point to it, while we cannot distinguish such a vertex out of three vertices which are connected cyclically. In a D-truss (or $(k_c, k_f)$-truss) [28], each edge is involved in cycle (flow resp.) triangles with at least $k_c$ ($k_f$ resp.) other vertices. The $k$-truss [9] is proposed for undirected graphs, where each edge is contained in at least $k$ triangles, applying the $k$-truss in directed graphs by ignoring directions of edges cannot distinguish different triangle participations of each edge.

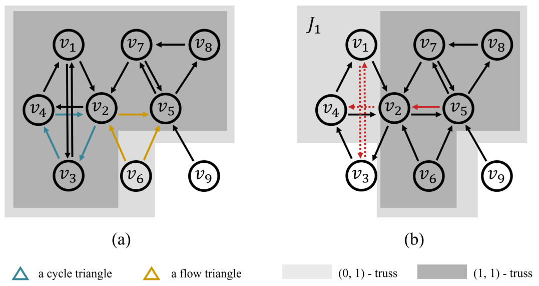


**Figure 1: The D-truss example in a toy directed graph**

The D-truss shows advantages in CS over directed graphs compared with the D-core and the $k$-truss. Formally, given a set of query vertices, D-truss community search (DCS) [28] is defined as finding the connected subgraph that: (1) contains the query vertices; (2) fulfils the requirement of D-truss number; (3) minimizes the diameter. Specifically, the existing solution to DCS [28] obtains the maximal satisfied subgraph first, and then peels off vertices

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

which are deemed as not tightly related to query vertices by minimizing the diameter. However, many existing CS works based on cohesive subgraph structures [17, 21, 38] which only require the first two conditions and then return the maximal satisfied subgraph directly as the query answer, because many applications need the maximality of the community such as network analysis [18, 23], recommendation [39] and event organisations [16]. We take the biological food web analysis [23] as an example, the maximal subgraph is necessary for discovering all taxa given the cohesiveness requirement. For proposing a general CS scheme based on the D-truss, we study the problem called maximal D-truss search (MDS), which removes the diameter constraint of DCS.

Considering directed graphs in the real-world, they are always changing with edges or vertices being continuously inserted into or removed from the original graph. Let the directed graph be $G$, the updates of it be $\Delta G$, and the result of the query given $G$ be $R(G)$, an incremental algorithm $\mathcal{A}_\Delta$ aims at obtaining the updated result and reusing $R(G)$ as much as possible, well-designed incremental algorithms are always more efficient and more scalable than from-scratch solutions [14, 15].

**Applications.** Finding the D-truss has many real-world applications, such as the analysis of social networks and the discovery of web networks [28]. It is inherent that many real-world directed graphs are updated frequently, thus having the D-truss in the latest graph maintained correctly and efficiently is extremely desired in many scenarios. We list some examples below.

• *Social network.* Figure 1 (a) shows an example of the social network, the direction of an edge represents the followed-following relationship between two users, i.e. the edge $\langle v_1, v_2 \rangle$ means the user $v_1$ follows the user $v_2$. Users always follow and unfollow others according to their interests, which will result in the update of the whole graph. In Figure 1 (a), we mark all D-trusses in different colors in the original graph $G$. In Figure 1 (b), let $\Delta G$ be $\{- \langle v_1, v_3 \rangle, - \langle v_3, v_1 \rangle, - \langle v_2, v_4 \rangle, + \langle v_5, v_2 \rangle\}$, where we denote the deletion by $-$ and the insertion by $+$. After applying $\Delta G$, the result of MDS becomes different from the result in Figure 1 (a).

• *Associative thesaurus.* The word associative network works as a useful tool for computational linguistics and natural language processing. The network of the Edinburgh Associative Thesaurus (EAT) [6] is built in this way: the word $w_i$ points to another word $w_j$ if $w_j$ appears as the response when $w_i$ is received as the stimulus. Given a query word, the D-truss will include all associative words of it and distinguish them via cycle triangles and flow triangles. In the case study of EAT [28], the $(3, 0)$-truss of the word "drink" consists of words that have an equal relationship with it, including "glass", "bottle", "wine" and "water". Instead the $(0, 7)$-truss of it shows the reminding hierarchy relationships around "drink". There exist words are in the upstream of the hierarchy relationship, e.g. the word "drunk" always reminds "drink" and other words in the subgraph, the word "drink" also reminds many words that in the downstream of the relationship, e.g. "rum" and "liquor". Once the thesaurus is updated with new edges added and out-of-time edges deleted, such relationships found by the D-truss may be changed, it is strongly motivated that maintaining these D-trusses instead of recomputing them from scratch.

**Challenges.** It is straightforward to check how the $k$-truss is addressed in dynamic undirected graphs. Though effective and efficient solutions for $k$-truss maintenance problem of both single-update [21] and batch-update [41] are developed, it is not trivial to extend those solutions to the D-truss. Instead, we must address the following challenges in order to solve the problem.

• *Challenge 1.* Compared with the $k$-truss model in undirected graphs, the D-truss takes the directions of edges within triangles into consideration. The direction of an edge makes it mean differently from an undirected edge, this fact implies two key points to notice: (1) the triangles in directed graphs can be classified into two types (i.e. cycle triangles and flow triangles) due to different directions within a triangle; (2) the $k$-truss ignore duplicate edges, while two edges with different directions are allowed between two vertices $u$ and $v$. Because of these two facts, the theoretical hardness of the D-truss maintenance is unknown compared with the $k$-truss maintenance.

• *Challenge 2.* According to the two different types of triangles, the D-truss decomposition [28] is naturally related to two types of supports. For single edge updates, existing work [21] proposes theorems that indicate the rules of $k$-truss maintenance depending on the only type of support. The fact that the same three vertices can be contained in multiple triangles in the D-truss makes such theorems cannot be extended correctly.

• *Challenge 3.* For more efficient batch-update solutions, the existing index [41] of the $k$-truss stores edges in the non-ascending order of $k$ values. However, there are a pair of values to consider (i.e. $(k_c, k_f)$) for the D-truss, the dominant property of $(k_c, k_f)$ [28] confirms that it is impossible to construct such a single order for the D-truss.

We denote that applying $\Delta G$ to $G$ by $G \oplus \Delta G$. Many existing solutions of CS have to recompute from scratch for $G \oplus \Delta G$, including those of the D-truss. The cost of recomputing given $\Delta G$ always depends on the size of the whole graph, the incremental solution would be more efficient if its cost is polynomial with a specific range of data accessed, which is always smaller than the size of the graph. Furthermore, the solution is more practical if it can handle the updates of different query vertices and $(k_c, k_f)$ requirements except for $\Delta G$, which is called fully-dynamic. In this paper, we aim at an efficient incremental solution for addressing the challenges above.

**Contributions.** For proposing a general incremental scheme for solving CS based on the D-truss, we first analyze the hardness of D-truss maintenance theoretically and then we propose solutions for both single-update and batch-update. More concretely, our contributions can be concluded below:

• We define the maximal D-truss search problem for generality and extend the existing solution to it.

• We conduct theoretical analysis about the boundedness and the unboundedness of the maximal D-truss maintenance.

• We present support update operators and basic single-update algorithms for edge insertion and deletion respectively for the maximal D-truss maintenance problem.

• We propose an order-based index structure, its associated batch-update algorithms and the fully-dynamic query algorithm.

**Outline.** In Section 2, we review the related work on this topic. Section 3 gives definitions of notations, concepts and problems. We show theoretical discussions in Section 4. Our basic methods are presented in Section 5 and index-based methods are introduced in Section 6. Section 7 gives batch-update algorithms and the query algorithm for the proposed index. We show experimental results in Section 8 and conclude this paper in Section 9.

## 2 RELATED WORK

In this section, we review existing works of cohesive subgraph models, their maintenance techniques in directed graphs and effectiveness measures of incremental algorithms.

**Cohesive subgraph structures over directed graphs.** We conclude popular cohesive subgraph structures into three types below.

• *Core-like models:* $k$-core [7, 8] is a cohesive subgraph structure of undirected graphs where every vertex has at least $k$ degrees. For directed graphs, D-core (or $(k, l)$-core) [17, 19] is proposed to capture the properties of considering in-degrees and out-degrees coordinately. Specifically, the D-core is limited by its inherent property that the fixed requirement of combinations of in-degree and out-degree. The degree-based features of some vertices can be identified explicitly, e.g. vertices with high in-degree and low out-degree imply that entities have influences over others like authorities, and vertices with low in-degree and high out-degree imply that entities tend to be followers [28]. Obviously, it is unable to include both two aforementioned types into a single community except for setting both in-degree and out-degree low, which makes the subgraph become too large and sparse.

• *Truss-like models:* In undirected graphs, $k$-truss [9] is defined based on the number of triangles that each edge is involved in. In this paper, we develop the work based on the D-truss (or $(k_c, k_f)$-truss) [28], which distinguishes two different types of triangles in semantics for directed graphs. More generally, the D-truss alleviates the limitation of the D-core mentioned above by varying different values of $k_c$ and $k_f$ [28].

**Cohesive subgraph maintenance over dynamic graphs.** In the real world, graphs are always changing over time, instead of recomputing subgraphs from scratch, many works focus on efficient incremental algorithms. For core maintenance, a suite of incremental $k$-core decomposition algorithms [34] are verified to be efficient, and a tree-like index [27] can be constructed for more efficient maintenance. A fully dynamic approximate solution for the $k$-core in hypergraphs [35] is also considered. In bipartite graphs, the problem of $(\alpha, \beta)$-core decomposition and maintenance is solved by an index-based solution [38]. For truss maintenance, different index structures are proposed based on the preserved information of triangle connectivity [21] and the edge decomposition order [41]. For clique maintenance, maintaining the $k$-clique in dynamic graphs is tackled by single-update [13] and batch-update methods [10].

## 3 PRELIMINARIES

Given a directed, simple and unweighted graph denoted by $G = (V(G), E(G))$. Here $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges that $E(G) = \{\langle u, v\rangle \,| u, v \in V(G)\}$. Note that double edges which point in opposite directions are allowed. The edge $\langle u, v\rangle$ is the directed edge that points from $u$ to $v$, where $u$ is an in-neighbor of $v$ and $v$ is an out-neighbor of $u$.

For a vertex $u \in V(G)$, we denote the set of in-neighbors of $u$ by $\mathsf{N}^{\mathsf{in}}(u, G) = \{v | \langle v, u\rangle \in E(G)\}$ and the set of out-neighbors of $v$ by $\mathsf{N}^{\mathsf{out}}(u, G) = \{v | \langle u, v\rangle \in E(G)\}$. $\mathsf{deg}^{\mathsf{in}}(u, G) = |\mathsf{N}^{\mathsf{in}}(u, G)|$ is used to denote the in-degree of $u$ and similarly $\mathsf{deg}^{\mathsf{out}}(u, G) = |\mathsf{N}^{\mathsf{out}}(u, G)|$ denotes the out-degree of $u$. The degree of $u$ is defined as the sum of its in-degree and out-degree. We use $n$ to denote the number of vertices in $G$ and $m$ to denote the number of edges in $G$. In this section, we give the definitions of the D-truss and existing solutions based on it.

### 3.1 Problem Definition

Before we give the formal problem statement, some basic concepts are defined as follows. We denote three vertices within a triangle by $u, v, w$ respectively. A cycle triangle is a triangle where each vertex within has exactly 1 in-degree and 1 out-degree, we denote it by $\triangle^c_{u,v,w}$. A flow triangle is a triangle where the out-degrees of vertices within are 0, 1, 2 and in-degrees of them are 2, 1, 0 correspondingly, we denote a flow triangle by $\triangle^f_{u,v,w}$. Figure 1 shows an example for directed triangles where $v_2, v_3$ and $v_4$ form a cycle triangle, $v_2, v_5$ and $v_6$ form a flow triangle. The D-truss [28] is proposed together with two types of support below.

**Table 1: Notations**

| Notation | Description |
|---|---|
| $G, \Delta G$ | A directed graph and updates to it |
| $G'$ or $G \oplus \Delta G$ | The graph after applying $\Delta G$ to $G$ |
| $V(G), E(G)$ | The set of vertices and edges respectively |
| $\mathsf{sup}^c_G(e), \mathsf{sup}^f_G(e)$ | The cycle support and flow support of $\forall e \in E(G)$ |
| $\mathsf{N}^c_G(e), \mathsf{N}^f_G(e)$ | The cycle set and flow set of $\forall e \in E(G)$ |
| $\mathsf{E}^c_G(e)$ | The set of edges that are spanned between $\mathsf{N}^c_G(e)$ and $\{u, v\}$, where $e = \langle u, v\rangle$. |
| $\mathsf{E}^f_G(e)$ | The set of edges that are spanned between $\mathsf{N}^f_G(e)$ and $\{u, v\}$, where $e = \langle u, v\rangle$. |
| $H_{k_c,k_f}$ | The D-truss ($(k_c, k_f)$-truss) in $G$ |
| $Q, \Delta Q$ | A set of query vertices in $V(G)$ and updates to this set |
| $Q'$ or $Q \oplus \Delta Q$ | The set of query vertices after applying $\Delta Q$ to $Q$ |
| $\mathcal{H}_{k_c,k_f,Q}(G)$ | The maximal $(k_c, k_f)$-truss community for $Q$ given $G$ |

**Definition 1.** ***Cycle Support*** [28]. Given a directed graph $G$ and an edge $e = \langle u, v\rangle \in E(G)$, the **cycle support** of $e$ in $G$ is the number of vertices that form cycle triangles with $e$ in $G$, denoted by $\mathsf{sup}^c_G(e) = |\{w \in V(G)| \triangle^c_{u,v,w} \text{ } in \text{ } G\}|$. We denote the **cycle set** of $e$ by $\mathsf{N}^c_G(e) = \{w \mid w \in \mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{out}}(v)\}$, which stores all vertices that form cycle triangles with $e$ in $G$. Let all edges that are spanned between $\mathsf{N}^c_G(e)$ and $\{u, v\}$ be $E^c_G(e)$.

**Definition 2.** ***Flow Support*** [28]. Given a directed graph $G$ and an edge $e = \langle u, v\rangle \in E(G)$, the **flow support** of $e$ in $G$ is defined similarly to the cycle support as $\mathsf{sup}^f_G(e) = |\{w \in V(G)| \triangle^f_{u,v,w} \text{ } in \text{ } G\}|$. We denote the **flow set** of $e$ by $\mathsf{N}^f_G(e) = \{w \mid w \in (\mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{in}}(v)) \cup (\mathsf{N}^{\mathsf{out}}(u) \cap \mathsf{N}^{\mathsf{in}}(v)) \cup (\mathsf{N}^{\mathsf{out}}(u) \cap \mathsf{N}^{\mathsf{out}}(v))\}$, which stores all vertices that form flow triangles with $e$ in $G$. Let all edges that are spanned between $\mathsf{N}^f_G(e)$ and $\{u, v\}$ be $E^f_G(e)$.

**Definition 3.** ***Maximal D-truss*** [28]. Given a directed graph $G$ and two non-negative integers $k_c$ and $k_f$, a subgraph $H_{k_c,k_f} \subseteq G$ is a D-truss ($(k_c, k_f)$-truss) if

- $\sup^c_G(e) \geq k_c$ and $\sup^f_G(e) \geq k_f$, $\forall e \in E(H_{c,f})$;
- $H_{k_c,k_f}$ is maximal (i.e. $H_{k_c,k_f}$ is not contained in any other $(k_c, k_f)$-trusses of $G$).

The pair $(k_c, k_f)$ is called a **trussness** of $e$, where $k_c$ is the **cycle truss number** of $e$ and $k_f$ is the **flow truss number** of $e$. Note that for a single edge $e \in E(G)$, it can be contained in different D-trusses, so there may exist multiple trussnesses for it. We denote the **trussness set** of $e$ as $\mathcal{T}(e) = \{(k_{c_i}, k_{f_i}) \mid e \in E(H_{c_i,f_i})\}$.

Based on the definitions above, we formally define a series of problems of maximal D-truss search below. First, we define the problem of maximal D-truss search (MDS) in Problem Statement 1.

**Problem Statement 1.** ***Maximal D-truss Search (MDS)*** [28]. Given a directed graph $G$, two non-negative integers $k_c$ and $k_f$, and a set of query vertices $Q \subseteq V(G)$, the maximal D-truss search (MDS) is to find a subgraph $\mathcal{H}_{k_c,k_f,Q}(G)$ such that:

- $\mathcal{H}_{k_c,k_f,Q}(G)$ is connected and $Q \subseteq V(\mathcal{H}_{k_c,k_f,Q}(G))$;
- $\mathcal{H}_{k_c,k_f,Q}(G)$ is a subgraph of the $(k_c, k_f)$-truss (i.e. $\mathcal{H}_{k_c,k_f,Q}(G) \subseteq H_{k_c,k_f}$);
- $\mathcal{H}_{k_c,k_f,Q}(G)$ is the maximal subgraph that satisfying two conditions above.

We extend the problem of MDS to two advanced problems considering the scenario of dynamic directed graphs below.

**Problem Statement 2.** ***Maximal D-truss Maintenance (MDM)*** Given a directed graph $G$ and a set of to-be-updated edges $\Delta G$, the problem of maximal D-truss maintenance (MDM) is to compute new trussness set of any edges $e \in E(G \oplus \Delta G)$, where $\oplus$ means that changes are applied. We use $G'$ for short of $G \oplus \Delta G$.

**Problem Statement 3.** ***Maximal D-truss Search Maintenance (MDSM).*** Given a directed graph $G$, the previously queried maximal D-truss $\mathcal{H}_{k_c,k_f,Q}$ in $G$, a set of to-be-updated edges $\Delta G$, new thresholds $k'_c$ and $k'_f$ and a set of to-be-updated query vertices $\Delta Q$, the maximal D-truss search maintenance (MDSM) is to search the new maximal D-truss $\mathcal{H}_{k'_c,k'_f,Q'}(G')$.

**Example 3.1.** Consider the given graph in Figure 1 (a), let $(k_c, k_f)$ be $(1, 1)$ and let $Q$ be $\{v_2\}$, in this way the result of MDS is the subgraph $H_3$. Then in Figure 1 (b), let $\Delta G$ be $\{\ominus\langle v_1, v_3\rangle, \ominus\langle v_3, v_1\rangle, \ominus\langle v_2, v_4\rangle, \oplus\langle v_5, v_2\rangle\}$, let $(k'_c, k'_f)$ be $(0, 1)$, let $\Delta Q_1$ be $\{\oplus v_5\}$ and let $\Delta Q_2$ be $\{\ominus v_2, \oplus v_3\}$. For $Q \oplus \Delta Q_1$, the result of MDSM is $J_1$, since both $v_2$ and $v_5$ are contained in the maximal $(0, 1)$-truss. For $Q \oplus \Delta Q_2$, the result of MDSM is null, it is obvious that $v_3$ is not contained in the maximal $(0, 1)$-truss $J_1$.

Note that the solution of MDS performs inefficiently when $\Delta G$ is given, because it have to recompute the result from scratch. The solution of MDM is expected to handle $\Delta G$ much more efficiently, which is the data-update of MDS. What's more, the solution of MDSM shall answer the new query by maintaining the old result, which is the query-update of MDS.

## 3.2 Background

The existing index-based solution for applying the D-truss in CS can be easily extended to obtain the maximal D-truss by sparing the procedures for minimizing the diameter [28]. Note that such a solution have to recompute the result when $\Delta G$ is given, which is inefficient compared with incremental solutions of MDM and MDSM. We introduce this easily-extended solution first before looking into the problem of MDM and MDSM.

**Skyline trussness.** Given a directed graph $G$ and consider an arbitrary edge $e \in E(G)$, $\mathcal{T}(e)$ contains trussnesses of all maximal D-trusses that $e$ is contained in. The property of dominance among trussnesses is revealed and the skyline trussness [28] is defined for the D-truss. Given a directed graph $G$, an $(k_c, k_f)$-truss and an $(k'_c, k'_f)$-truss that both contain $e \in E(G)$. The trussness $(k'_c, k'_f)$ *dominates* $(k_c, k_f)$ if $(k'_c > k_c, k'_f \geq k_f)$ or $(k'_c \geq k_c, k'_f > k_f)$. If we cannot find any other trussnesses $(k''_c, k''_f)$ containing $e$ which dominates $(k'_c, k'_f)$, then $(k'_c, k'_f)$ is a skyline trussness of $e$. For $\forall e \in E(G)$, there may exist several different skyline trussnesses, we define them as **skyline trussness set**, denoted by $\mathcal{ST}(e)$.

**D-truss decomposition.** With the computation of the skyline trussness set for all edges of the given graph, all possible queries of MDS can be efficiently answered from the constructed index [28]. For computing the skyline trussness sets $\mathcal{ST}(e)$ for $\forall e \in E(G)$ by D-truss decomposition, the algorithm [28] pre-computes the cycle support and the flow support for all edges first. Then it decomposes the graph according to all possible cycle truss numbers and stores them into subgraphs $D_{(k_c,0)}$. After that, it decomposes all subgraphs $D_{(k_c,0)}$ according to flow truss numbers, $\mathcal{ST}(e)$ for all edges $e$ can be effectively maintained during this process. Note that the skyline trussness sets work as an index used for answering queries of MDS.

**Index-based query algorithm.** A query algorithm [28] is given which can find the maximal D-truss that contains $Q$ by sparing the procedures for minimizing the diameter. Firstly, it selects a query vertex from $Q$ and pushes it into the empty queue used for breadth first search (BFS). Note that all unvisited edges that incident to vertices in the queue are checked based on the index that stores $\mathcal{ST}(e)$ for all edges. If there exists a skyline trussness of $e$ that dominates or equals the trussness threshold $(k_c, k_f)$, then it pushes this edge $e$ into the result community. During this process, unvisited incident vertices are also pushed into the queue until no more vertices are qualified for the queue.

## 4 THEORETICAL ANALYSIS OF MDM

For proposing the incremental solutions for the MDM problem, we need to analyze the hardness of MDM first in this section.

### 4.1 The Theoretical Model and Metrics

First, we introduce the class of locally persistent algorithms as the computation model for MDM.

**Locally persistent algorithms.** Many works [2, 14, 32, 41] on the boundedness of graph problems are based on the class of locally persistent algorithms. Following these works, we also require $\mathcal{A}_\Delta$ to be locally persistent. An algorithm is locally persistent [2, 32] if

- it may use a block of storage for each edge, where it saves pointers to (the blocks of storage for) its adjacent edges;
- no global auxiliary information is allowed, such as the blocks of each edge are not allowed to save pointers to non-adjacent edges;
- the block of edge $e$ may contain auxiliary status information status$(e)$ about $e$.

Given the updates $\Delta G$, a locally persistent algorithm $\mathcal{A}_\Delta$ starts from the pointers of edges in $E(\Delta G)$ and traverses the original graph $G$ via tracing pointers. The next-to-be-visited edge only depends on the status of all visited edges of $G$. Then we introduce the concept of boundedness and relative boundedness for an $\mathcal{A}_\Delta$.

**Boundedness.** For the problem of MDM, we denote the set of edges whose trussness set changes by CHANGED. An incremental algorithm $\mathcal{A}_\Delta$ is bounded [37] if its cost is a polynomial function of $\|\text{CHANGED}\|_c$, where $\|\text{CHANGED}\|_c$ denotes the size of $c$-hop neighbors of CHANGED for a positive integer $c$. The problem of MDM is bounded if there exists a bounded $\mathcal{A}_\Delta$, or is unbounded otherwise. In this way, a bounded $\mathcal{A}_\Delta$ is desired because its cost only depends on CHANGED, other than the size of the whole graph $G$. However, the boundedness is such a strong property that many graph problems cannot be solved boundedly [15]. Regarding this gap of hardness, the relative boundedness [14] is proposed for measuring the efficiency of incremental algorithms for graph problems.

**Relative boundedness.** The relative boundedness [14] is a proposed metric for batch algorithms. A batch algorithm $\mathcal{A}$ is given a specific query class and a graph, and it returns the answer of this query with the cost related to the size of the graph and the query set. Let the data accessed by $\mathcal{A}$ during the computing be $G_{\mathcal{A}}$, and the difference between $(G \oplus \Delta G)_{\mathcal{A}}$ and $G_{\mathcal{A}}$ be AFF. An incremental algorithm $\mathcal{A}_\Delta$ is bounded relative to $\mathcal{A}$ if its cost is a polynomial function of $|\text{AFF}|$. Existing works [14, 15, 41] show that an unbounded incremental algorithm is still practically efficient if it is of relative boundedness, which shows the impact of the relative boundedness.

## 4.2 The Boundedness Discussion of MDM

Existing works study the maintenance problems of other cohesive subgraph structures, including the truss maintenance [21, 41] and the core maintenance [27, 34] for undirected graphs. It is proved that both the truss maintenance and the core maintenance are asymmetric regarding the boundedness of edge insertions and deletions [41]. We show the theoretical conclusion in Theorem. 4.1 similarly and prove it in the rest of this section.

THEOREM 4.1. *Under the model of locally persistent algorithms, the problem of MDM is bounded for edge deletions, but unbounded for edge insertions.*

**The unboundedness of edge insertions.** The proof sketch is similar to the proof for the $k$-truss [41]. We construct a directed graph $G$ and two specific sets of edge updates $\Delta G_1$ and $\Delta G_2$. We list two key points of the proof sketch below.

(1) By applying $\Delta G_1$ and $\Delta G_2$ to the original graph $G$ respectively, the size of $|\text{CHANGED}|$ for each update is constant (i.e. $O(f(\|\text{CHANGED}\|_c)) = O(1)$). If there exists a bounded algorithm $\mathcal{A}_\Delta$, it can obtain the correct result with a constant cost for $\Delta G_1$ and $\Delta G_2$ respectively.

(2) For any locally persistent algorithms, the cost of applying $\Delta G_1$ and the cost of applying $\Delta G_2$ are $\Omega(l)$, where $l$ is not constant.

Note that the second point directly contradicts the existence of the bounded algorithm. Because if there exists a bounded algorithm, the cost of applying $\Delta G_1$ plus the cost of applying $\Delta G_2$ should also be constant.

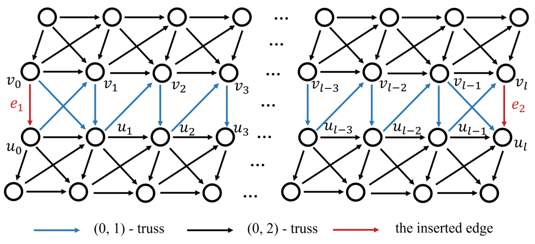


**Figure 2: The unboundedness of edge insertions for MDM**

According to the proof sketch introduced above, we construct the directed graph $G$ in Figure 2. Let the insertion of $e_1 = \langle v_0, u_0 \rangle$ be the update $\Delta G_1$ and the insertion of $e_2 = \langle v_l, u_l \rangle$ be the update $\Delta G_2$. Considering all edges in the original graph, we have the trussness set for each edge $e \in E(G)$ marked in Figure 2. We assume that there exists a bounded insertion algorithm $\mathcal{A}_\Delta$ for the MDM problem.

First, after applying $\Delta G_1$ singly, CHANGED $= \{e_1, \langle v_0, u_1 \rangle, \langle u_0, v_1 \rangle, \langle v_1, u_1 \rangle\}$, where these 4 edges have their trussness changed to $(0, 2)$. This implies that $\mathcal{A}_\Delta$ costs $O(f(\|\text{CHANGED}\|_c)) = O(1)$ time for the update $\Delta G_1$.

Similarly, CHANGED is also of constant size for the update $\Delta G_2$ singly, where CHANGED $= \{e_2, \langle v_{l-1}, u_l \rangle, \langle u_{l-1}, v_l \rangle, \langle v_{l-1}, u_{l-1} \rangle\}$. The time cost for the update $\Delta G_2$ is $O(f(\|\text{CHANGED}\|_c)) = O(1)$, too. Let $E_{\mathcal{A}}(G, \Delta G)$ be the sequence of edges that $\mathcal{A}_\Delta$ visits for applying $\Delta G$ to $G$. Both $|E_{\mathcal{A}}(G, \Delta G_1)|$ and $|E_{\mathcal{A}}(G, \Delta G_2)|$ are of constant size.

Then, we denote the subgraph that applies $\Delta G_1$ to $G$ by $H$, and then we compare the sequences visited by $\mathcal{A}_\Delta$ between applying $\Delta G_2$ to $G$ and $H$. According to the definition of the boundedness, $\mathcal{A}_\Delta$ chooses the next edge only depending on all visited edges. Thus the sequences are the same for $G \oplus \Delta G_2$ and $H \oplus \Delta G_2$ until an edge $e$ is visited whose status$(e)$ are different in these two sequences. Such $e$ must exist because CHANGED for these two updates in Figure 2 are different. Furthermore, this $e$ is definitely visited in $E_{\mathcal{A}}(G, \Delta G_1)$ since status$(e)$ are different in $H$ and $G$. In this way, $E_{\mathcal{A}}(G, \Delta G_1)$ overlaps $E_{\mathcal{A}}(G, \Delta G_2)$ with $e$. For any locally persistent algorithms, there exist a path from $e_1$ to $e$ in $E_{\mathcal{A}}(G, \Delta G_1)$ and a path from $e_2$ to $e$ in $E_{\mathcal{A}}(G, \Delta G_2)$ respectively, then we obtain a path between $e_1$ and $e_2$ with the length of $\Omega(l)$ which is obvious in Figure 2. This implies that $|E_{\mathcal{A}}(G, \Delta G_1)| + |E_{\mathcal{A}}(G, \Delta G_2)| = \Omega(l)$, which contradicts the fact that both $|E_{\mathcal{A}}(G, \Delta G_1)|$ and $|E_{\mathcal{A}}(G, \Delta G_2)|$ are of constant size. Above all, $\mathcal{A}_\Delta$ cannot be bounded for edge insertions of the MDM problem.

**The boundedness of edge deletions.** For the boundedness of edge deletions of the MDM problem, we provide the proof in Section 7.1, where we present a bounded batch-edge-deletion algorithm.

## 5 BASIC SINGLE-UPDATE ALGORITHMS

In this section, we consider the cases of single-edge deletion and insertion for the MDM problem, since the single update is the most basic scenario for dynamic directed graphs. Different from the $k$-truss maintenance problem [41], consider a D-truss, the fact that the same three vertices can be contained in multiple triangles makes the existing index [41] cannot have the cycle trussness and the flow trussness correctly updated. Before presenting the proposed algorithms, we introduce some typical operations of the support maintenance. Then we elaborate on algorithms and conduct the complexity analysis for them regarding Theorem 4.1. Without loss of generality, we assume all updates are valid, it must delete an existing edge or insert a new edge for each update.

## 5.1 Support Single-Update Operators

For the $k$-truss, the support of each edge $e = (u, v)$ is the size of the intersection of the neighbors of $u$ and $v$, which is equal to the number of triangles that $e$ is contained in. Existing works [21, 41] observe the fact that each single edge deletion (resp. insertion) results in the decrement (resp. the increment) of supports by 1 for a specific range of edges.

However, this fact does not hold for the D-truss. Because in directed graphs, (1) there are two different types of supports, (2) two edges are allowed between the same vertices (i.e. $\langle u, v\rangle$ and $\langle v, u\rangle$). We propose 4 operators for single-update scenarios below, applying these operators to maintain supports can directly update the trussness set of some edges in CHANGED to save the computation cost.

To distinguish different connections between two vertices $u$ and $v$ in directed graphs, we say $u$ and $v$ are *uni-linked* if there exists an edge either $\langle u, v\rangle$ or $\langle v, u\rangle$, they are *bi-linked* if there exist both $\langle u, v\rangle$ and $\langle v, u\rangle$, and they are *not linked* if there exists no edge between them.

Lemma 5.1. *Given a directed graph $G$, consider any two vertices $u, v \in V(G)$. For the insertion $e^+ = \langle u, v\rangle$ (resp. the deletion $e^- = \langle u, v\rangle$) if $u$ and $v$ are not uni-linked before $\oplus e^+$ (resp. $u$ and $v$ are not bi-linked before $\oplus e^-$), then for any affected edge, its cycle support, or its flow support, or both increase (resp. decrease) by at most 1.*

Proof. Considering the cases that are not related with *bi-linked* vertices before and after updating in Lemma 5.1. For existing edges, their directions are fixed, both insertion and deletion can only result in the support changing by at most 1. The type of the changed support depends on the directions of the other two edges in the triangle. □

The affected edges are composed of two parts, one is the changed edge $e^*$ (i.e. the inserted edge $e^+$ or the deleted edge $e^-$), and the other is all edges that are incident to all vertices in $\mathsf{N}^c_{G'}(e^*)$ and $\mathsf{N}^f_{G'}(e^*)$. For the changed edge $e^*$, it is inevitable to compute its supports from scratch, the cost of computing is different for the insertion and the deletion. It costs $O(\deg(u) + \deg(v))$ for the insertion because of computing the intersection between the neighbor sets of two endpoints $u$ and $v$, and it only costs $O(1)$ for the deletion by zeroing them. For the the other part, it costs $O(\sum_z(\deg(z) + \deg(\mathsf{N}(z))))$ for recomputing them from scratch, where $z$ denotes the shared neighbors of $u$ and $v$ and is bounded by $O(min(\deg(u), \deg(v)))$. We observe that it is not necessary to recompute supports of the latter part of affected edges but it can be maintained by two proposed operators $\mathsf{Ins}(e^+)$ and $\mathsf{Del}(e^-)$. Both operators can update supports efficiently for all edges that are incident to all vertices in $\mathsf{N}^c_{G'}(e^*)$ and $\mathsf{N}^f_{G'}(e^*)$ with the time cost of $O(\sum_z \deg(z))$ only. The correctness is guaranteed by Lemma 5.1.

**$\mathsf{Ins}(e^+)$.** We compute supports for the newly inserted edge $e^+$. According to Lemma 5.1, for any edge $e^c$, its cycle support increases by 1; for any edge $e^f$, its flow support increases by 1; note that there may exist the same edges in both $E^c_{G'}(e^+)$ and $E^f_{G'}(e^+)$.

$$\mathsf{Ins}(e^+)\begin{cases}\mathsf{sup}^c_{G'}(e^+) \leftarrow |\mathsf{N}^c_{G'}(e^+)|; \mathsf{sup}^c_{G'}(e^c) \leftarrow \mathsf{sup}^c_G(e^c) + 1, \forall e^c \in E^c_{G'}(e^+);\\ \mathsf{sup}^f_{G'}(e^+) \leftarrow |\mathsf{N}^f_{G'}(e^+)|; \mathsf{sup}^f_{G'}(e^f) \leftarrow \mathsf{sup}^f_G(e^f) + 1, \forall e^f \in E^f_{G'}(e^+).\end{cases}$$

**$\mathsf{Del}(e^-)$.** For the deleted edge, we zero its support. Similarly to $\mathsf{Ins}(e^+)$, we only need to update the cycle support of $e^c$ and the flow support of $e^f$.

$$\mathsf{Del}(e^-)\begin{cases}\mathsf{sup}^c_{G'}(e^-) \leftarrow 0; \mathsf{sup}^c_{G'}(e^c) \leftarrow \mathsf{sup}^c_G(e^c) - 1, \forall e^c \in E^c_{G'}(e^-);\\ \mathsf{sup}^f_{G'}(e^-) \leftarrow 0; \mathsf{sup}^f_{G'}(e^f) \leftarrow \mathsf{sup}^f_G(e^f) - 1, \forall e^f \in E^f_{G'}(e^-).\end{cases}$$

Now we consider the cases that $u$ and $v$ are *bi-linked* before and after applying the changed edge $e^* = \langle u, v\rangle$, we propose operators $\mathsf{Enga}(e^+)$ and $\mathsf{DisEnga}(e^-)$, and the correctness of them is shown in Lemma 5.2.

Lemma 5.2. *Given a directed graph $G$, consider any two vertices $u, v \in V(G)$. For the insertion $e^+ = \langle u, v\rangle$ (resp. the deletion $e^- = \langle u, v\rangle$), if $u$ and $v$ are uni-linked before $\oplus e^+$ (resp. $u$ and $v$ are bi-linked before $\oplus e^-$), then $\mathsf{Enga}(e^+)$ (resp. $\mathsf{DisEnga}(e^-)$) updates supports correctly for all affected edges.*

Proof. Consider the cases that are related to *bi-linked* before and after updating. Such an insertion (or deletion) is about the reversed edge of an existing edge, thus for the updated edge, we need to recompute the cycle support only, and then the flow support can be updated based on the supports of the existing edge. □

**$\mathsf{Enga}(e^+)$.** Consider a single edge insertion $e^+ = \langle u, v\rangle$, it is known that $u$ and $v$ are *uni-linked* before the insertion (i.e. there exists an edge $\widetilde{e^+} = \langle v, u\rangle \in E(G)$). For $e^+$, it costs $O(\deg^{\mathsf{in}}(u) + \deg^{\mathsf{out}}(v))$ for updating its cycle support and flow support. The cycle support increases by 1 for any edge $e^c$ and the flow support increases by 1 for any edge $e^f$ with the time cost of $O(\deg(u) + \deg(v))$.

$$\mathsf{Enga}(e^+)\begin{cases}\mathsf{sup}^c_{G'}(e^+) \leftarrow \left|\mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{out}}(v)\right|;\\ \mathsf{sup}^c_{G'}(e^c) \leftarrow \mathsf{sup}^c_G(e^c) + 1,\\ \forall e^c \in \{\langle v, w\rangle, \langle w, u\rangle \mid\\ \forall w \in \mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{out}}(v) \setminus \mathsf{N}^{\mathsf{in}}(v) \cap \mathsf{N}^{\mathsf{out}}(u)\};\\ \mathsf{sup}^f_{G'}(e^+) \leftarrow \mathsf{sup}^f_G(\widetilde{e^+}) - \mathsf{sup}^c_{G'}(e^+) + \mathsf{sup}^c_G(\widetilde{e^+});\\ \mathsf{sup}^f_{G'}(e^f) \leftarrow \mathsf{sup}^f_G(e^f) + 1,\\ \forall e^f \in \{\langle u, x\rangle, \langle x, v\rangle \mid\\ \forall x \in \mathsf{N}^{\mathsf{in}}(v) \cap \mathsf{N}^{\mathsf{out}}(u) \setminus (\mathsf{N}(u) \cap \mathsf{N}^{\mathsf{out}}(v)) \cup (\mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{in}}(v))\}.\end{cases}$$

**$\mathsf{DisEnga}(e^-)$.** Consider a single edge deletion $e^- = \langle u, v\rangle$, it is known that $u$ and $v$ are *bi-linked* before the deletion. For $e^-$, it costs $O(1)$ for zeroing supports. The cycle support decreases by 1 for any edge $e^c$ and the flow support decreases by 1 for any edge $e^f$ with the time cost of $O(\deg(u) + \deg(v))$.

$$\mathsf{DisEnga}(e^-)\begin{cases}\mathsf{sup}^c_{G'}(e^-) \leftarrow 0; \mathsf{sup}^c_{G'}(e^c) \leftarrow \mathsf{sup}^c_G(e^c) - 1,\\ \forall e^c \in \{\langle v, w\rangle, \langle w, u\rangle \mid\\ \forall w \in \mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{out}}(v) \setminus \mathsf{N}^{\mathsf{in}}(v) \cap \mathsf{N}^{\mathsf{out}}(u)\};\\ \mathsf{sup}^f_{G'}(e^-) \leftarrow 0; \mathsf{sup}^f_{G'}(e^f) \leftarrow \mathsf{sup}^f_G(e^f) - 1,\\ \forall e^f \in \{\langle u, x\rangle, \langle x, v\rangle \mid\\ \forall x \in \mathsf{N}^{\mathsf{in}}(v) \cap \mathsf{N}^{\mathsf{out}}(u) \setminus (\mathsf{N}(u) \cap \mathsf{N}^{\mathsf{out}}(v)) \cup (\mathsf{N}^{\mathsf{in}}(u) \cap \mathsf{N}^{\mathsf{in}}(v))\}.\end{cases}$$

## 5.2 Single Edge Deletion Algorithm

We denote the to-be-deleted edge by $e^- = \langle u, v\rangle$ and denote the updated skyline trussness sets by $\mathcal{ST}'(e)$ for $\forall e \in E(G\backslash e^-)$. First, we discuss the range of affected skyline trussness sets in Lemma 5.3.

Lemma 5.3. *Given a directed graph G, consider the deletion $e^- = \langle u, v\rangle \in E(G)$. Then for each edge $e \in \mathsf{E}^c_G(e^-) \cup \mathsf{E}^f_G(e^-)$ and its skyline trussness set $\mathcal{ST}(e) = \{(k_{c_i}, k_{f_i})| \text{ for all } i\}$, (1) if $(k_{c_i}, k_{f_i})$ dominates any skyline trussness $(k_c(e^-)+1, k_f(e^-)+1)$ in $\mathcal{ST}(e^-)$, then $(k_{c_i}, k_{f_i})$ will NOT be affected; (2) otherwise $(k_{c_i}, k_{f_i})$ may be affected.*

Proof. Based on Lemma 5.1 and Lemma 5.2, the upper bound of support decrement is exactly 1. It is straightforward that the upper bound of trussness decrement is 1 for both the cycle truss number and the flow truss number. Based on this fact, we can filter out those trussness sets that both exceed 1 in the cycle truss number and the flow truss number (i.e. $(k_c + 1, k_f + 1)$), remained trussness sets cannot be excluded from the checking. □

We present an algorithm that can update an edge deletion correctly for the MDM problem in Algorithm 1. In line 1, we initialize variables. In line 2, we update supports of corresponding edges with operators $\mathsf{DisEnga}(e^-)$ and $\mathsf{Del}(e^-)$. In lines 4-5, we start processing the queue that is used for detecting affected edges. We check each trussness of the current edge $\langle u_1, v_1\rangle$ in lines 6-12. In lines 7-9, we maintain trussness sets that can be done instantly after support maintenance. In line 10, we peel off edges whose cycle support is not qualified. We check the local cycle support and the local flow support of $\langle u_1, v_1\rangle$ in lines 11-12. At last, we mark $\langle u_1, v_1\rangle$ and continue to push edges into *Queue* in lines 13-14. When there is no edge in *Queue*, we have all skyline trussness sets correctly updated and return them as $\mathcal{ST}'(e), \forall e \in E(G')$.

**Algorithm 1:** Single Edge Deletion Algorithm

```
   Input: $G$, $e^- = \langle u,v\rangle$, $\mathcal{ST}(e), \forall e \in E(G)$
   Output: $\mathcal{ST}'(e), \forall e \in E(G\backslash e^-)$
1  $G' \leftarrow G\backslash e^-$; $Queue \leftarrow \emptyset$; $H \leftarrow \emptyset$; DelCheck2($e^-$);
2  Call DisEnga($e^-$) if $\langle v,u\rangle \in E(G)$, or call Del($e^-$) otherwise;
3  while !$Queue.empty()$ do
4      $H \leftarrow G'$; $\langle u_1, v_1\rangle \leftarrow Queue.top()$;
5      if $\langle u_1, v_1\rangle$ is NOT visited then
6          for each $k_{c_i}$ in associated trussnesses of $\langle u_1, v_1\rangle$ do
7              if $k_{c_i} == \mathsf{sup}^c_G(\langle u_1, v_1\rangle)$ and
                 $\mathsf{sup}^c_{G'}(\langle u_1, v_1\rangle) < \mathsf{sup}^c_G(\langle u_1, v_1\rangle)$ then
8                  $k_c \leftarrow k_c - 1$;
9              Repeat lines 7-8 similarly for the associated $k_f$;
10             $H \leftarrow H\backslash e'$, s.t. $k_c(e') < k_{c_i}$;
11             if LCS($\langle u_1, v_1\rangle, H$) $< k_{c_i}$ then $k_{c_i} \leftarrow k_{c_i} - 1$ ;
12             if LFS($\langle u_1, v_1\rangle, H$) $< k_{f_i}$ then $k_{f_i} \leftarrow k_{f_i} - 1$ ;
13         Mark $\langle u_1, v_1\rangle$ as visited;
14     if $\langle u_1, v_1\rangle$ is affected then DelCheck2($\langle u_1, v_1\rangle$);
15 return $\mathcal{ST}'(e), \forall e \in E(G')$;
   Procedure. DelCheck2($e$)
   Put all unvisited edges in $\mathsf{E}^c_G(e) \cup \mathsf{E}^f_G(e)$ together with their
     skyline trussness set belong to case (2) in Lemma 5.3 into $Queue$;
   Procedure. LCS($e, H$)
   Return the cycle support of the edge $e$ in the subgraph $H$;
   Procedure. LFS($e, H$)
   Return the flow support of the edge $e$ in the subgraph $H$;
```

## 5.3 Single Edge Insertion Algorithm

We denote the to-be-inserted edge by $e^+ = \langle u, v\rangle$ and denote the updated skyline trussness set by $\mathcal{ST}'(e)$ for $\forall e \in E(G \oplus e^+)$. First, we discuss the range of affected skyline trussness set in Lemma 5.4.

Lemma 5.4. *Given a directed graph G, consider the insertion $e^+ = \langle u, v\rangle$. Then for each edge $e \in \mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)$ and its skyline trussness set $\mathcal{ST}(e) = \{(k_{c_i}, k_{f_i})| \text{ for all } i\}$, if $(k_{c_i}, k_{f_i})$ dominates any skyline trussness $(k'_c(e^+)+1, k'_f(e^+)+1)$ in $\mathcal{ST}(e^+)$, then $(k_{c_i}, k_{f_i})$ will NOT be affected, otherwise $(k_{c_i}, k_{f_i})$ may be affected.*

Proof. Lemma 5.4 can be proved similarly to Lemma 5.3, thus we omit it here. □

We present an algorithm that can update an edge insertion correctly for the MDM problem in Algorithm 2. In lines 1-3, we initialize variables. We call operators to update supports in line 4. In lines 5-15, we maintain edges in $\mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)$ after applying operators. In line 14, note that the support checked within this loop has been correctly updated, in this way, $\mathsf{sup}_{G'}(e) > k$ can guarantee an increase in trussness. Because for those edges who forms triangles with $e^+$ and have their cycle truss or the flow truss of $k$, the increase in trussness is guaranteed with operators updating supports correctly. In line 16, we compute the skyline trussness set for $e^+$. We start processing the queue in lines 17-18 and check the change of cycle truss numbers and flow truss numbers successively in lines 19-35. In lines 20-22, we mark edges whose cycle truss number may increase and append the queue. In lines 23-25, we unmark edges whose cycle truss number remains the same. In lines 26-34, we conduct the checking for flow truss numbers similarly. We update trussness sets according to their marks in line 35 based on the correctly updated trussnesses from lines 7-15 and return the result in line 36.

## 5.4 Theoretical Analysis

Theorem 5.5. *For Algorithm 1, the time complexity is $O(\Delta k_c \cdot \sum_{e \in E_{DC2}} \deg(u_1) + \deg(v_1))$, where $e = \langle u_1, v_1\rangle$ and $|E_{DC2}|$ is the number of edges that are inspected by* DelCheck2 *in Algorithm 1 and is bounded by $\|\mathsf{CHANGED}\|_1$, and the space complexity is $O(m)$.*

Proof. Consider Algorithm 1, in lines 1-2, the cost of updating supports is linear with $|E_{DC2}|$ based on $\mathsf{Del}(\cdot)$ and $\mathsf{DisEnga}(\cdot)$ by Lemma 5.1 and Lemma 5.2. In line 3, the size of *Queue* is initialized by $\mathsf{DelCheck2}(\cdot)$ which is exactly the number of supports of $e^-$. The loop in line 6 is bounded by $\Delta k_c$, where $\Delta k_c = \max\{k_{cmax} - k_{c_0} \mid \{k_{cmax}, k_{c_0}\} \in \mathcal{ST}(\langle u_1, v_1\rangle)\}$. The cost of $\mathsf{LCS}(e, H)$ and $\mathsf{LFS}(e, H)$ is bounded by $O(\deg(u_1) + \deg(v_1))$, and the correctness is guaranteed by Lemma 5.3. The *Queue* is appended by cascading of edges that supports are changed in line 14. Note that only edges in $\|\mathsf{CHANGED}\|_1$ will be appended into this *Queue*. Above all, the time complexity of Algorithm 1 is bounded by $O(\Delta k_c \cdot \sum_{e \in E_{DC2}} \deg(u_1) + \deg(v_1))$, where $e = \langle u_1, v_1\rangle$. For the space complexity, the computation is bounded by the size of the original graph $G$ (i.e. $O(m)$). □

Theorem 5.6. *For Algorithm 2, the time complexity is $O(\Delta k_c \cdot \sum_{e \in E_{IC2}} \deg(u_1) + \deg(v_1))$, where $e = \langle u_1, v_1\rangle$ and $|E_{IC2}|$ is the*

**Algorithm 2:** Single Edge Insertion Algorithm

```
   Input: $G$, $e^+ = \langle u, v \rangle$, $\mathcal{ST}(e)$, $\forall e \in E(G)$
   Output: $\mathcal{ST}'(e)$, $\forall e \in E(G \oplus e^+)$
1  $G' \leftarrow G \oplus e^+$; $Queue \leftarrow \emptyset$; $H \leftarrow \emptyset$; InsCheck2($e^+$);
2  for each $e \in E(G')$ do
3      $\mathsf{c}_{\mathsf{inc}}(e) \leftarrow false$; $\mathsf{f}_{\mathsf{inc}}(e) \leftarrow false$; $\mathsf{D}_{\mathsf{unc}}(e) \leftarrow false$;
4  Call Enga($e^+$) if $\langle v, u \rangle \in E(G)$, or call Ins($e^+$) otherwise;
5  $k_{cmin} \leftarrow \min\{k_c \mid (k_c, k_f) \in \mathcal{ST}(e'), \forall e' \in \mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)\}$;
6  $k_{cmax} \leftarrow \max\{k_c \mid (k_c, k_f) \in \mathcal{ST}(e'), \forall e' \in \mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)\}$;
7  for $k_c \leftarrow k_{cmin}$ to $k_{cmax}$ do
8      if $\sup^c_{G'}(e^+) == k_c$ then
9          $k'_c(e^+) \leftarrow k_c$;
10         for each $e' \in \mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)$ do $\mathsf{D}_{\mathsf{unc}}(e') \leftarrow true$;
11     else if $\sup^c_{G'}(e^+) < k_c$ then
12         for each $e' \in \mathsf{E}^c_G(e^+) \cup \mathsf{E}^f_G(e^+)$ do $\mathsf{D}_{\mathsf{unc}}(e') \leftarrow true$;
13     else
14         $k'_c(e') \leftarrow k_c + 1$ s.t. $k_c(e') == k_c$;
15 Repeat lines 5-14 similarly for flow truss numbers;
16 Compute the skyline trussness set for $e^+$;
17 while !$Queue.empty()$ do
18     $H \leftarrow G'$; $\langle u_1, v_1 \rangle \leftarrow Queue.top()$;
19     for each $k_{c_i}$ in associated trussnesses $(k_{c_i}, k_{f_i})$ of $\langle u_1, v_1 \rangle$ do
20         if LCUB($\langle u_1, v_1 \rangle$, $k_{c_i}$) $\geq k_{c_i} + 1$ then
21             if !$\mathsf{c}_{\mathsf{inc}}(\langle u_1, v_1 \rangle)$ then
22                 $\mathsf{c}_{\mathsf{inc}}(\langle u_1, v_1 \rangle) \leftarrow true$; InsCheck2($\langle u_1, v_1 \rangle$);
23         else
24             if $\mathsf{c}_{\mathsf{inc}}(\langle u_1, v_1 \rangle)$ then
25                 $\mathsf{c}_{\mathsf{inc}}(\langle u_1, v_1 \rangle) \leftarrow false$;
26                 if LFUB($\langle u_1, v_1 \rangle$, $k_{f_i}$) $\geq k_{f_i} + 1$ then
27                     if !$\mathsf{f}_{\mathsf{inc}}(\langle u_1, v_1 \rangle)$ then
28                         $\mathsf{f}_{\mathsf{inc}}(\langle u_1, v_1 \rangle) \leftarrow true$;
29                         InsCheck2($\langle u_1, v_1 \rangle$);
30                 else
31                     if $\mathsf{f}_{\mathsf{inc}}(\langle u_1, v_1 \rangle)$ then
32                         $\mathsf{f}_{\mathsf{inc}}(\langle u_1, v_1 \rangle) \leftarrow false$;
33                         $\mathsf{D}_{\mathsf{unc}}(\langle u_1, v_1 \rangle) \leftarrow true$;
34                         InsCheck2($\langle u_1, v_1 \rangle$);
35         if $\mathsf{c}_{\mathsf{inc}}(e)$ or $\mathsf{f}_{\mathsf{inc}}(e)$ then Update $k_{c_i}$ or $k_{f_i}$ by 1;
36 return $\mathcal{ST}'(e)$, $\forall e \in E(G')$;
   Procedure. InsCheck2($e$)
   Put all unvisited edges in $\mathsf{E}^c_G(e) \cup \mathsf{E}^f_G(e)$ belong to case (2)
     together with their undominated skyline trussnesses into $Queue$;
   Procedure. LCUB($e$, $k_c$)
   Compute the local cycle support upper bound of $e$ by counting the
     number of cycle supports where the other two edges in the $\triangle^c$
     have their $k_{c_i} \geq k_c$ and $\mathsf{D}_{\mathsf{unc}}(\cdot)$ is $false$;
   Procedure. LFUB($e$, $k_f$)
   Compute the local flow support upper bound of $e$ by counting the
     number of flow supports where the other two edges in the $\triangle^f$
     have their $k_{f_i} \geq k_f$ and $\mathsf{D}_{\mathsf{unc}}(\cdot)$ is $false$;
```

*number of edges that are inspected by* InsCheck2 *in Algorithm 2, and the space complexity is $O(m')$.*

Proof. Consider Algorithm 2, in lines 1-4, the cost is linear with $|E_{DC2}|$ based on Ins$(\cdot)$ and Enga$(\cdot)$ by Lemma 5.1 and Lemma 5.2. In lines 5-6, the computation is bounded by $O(\Delta k_c \cdot (\deg(u) + \deg(v)))$, where $\langle u, v \rangle = e^+$. The loop in line 7 is executed $\Delta k_c$ times exactly and the cost is linear with $(\deg(u) + \deg(v))$ in lines 8-14. In this way, the cost of line 15 is $O(\Delta k_f \cdot (\deg(u) + \deg(v)))$. Then in lines 17-35, the cost is similar to Algorithm 1, bounded by $O(\Delta k_c \cdot \sum_{e \in E_{IC2}} \deg(u_1) + \deg(v_1)))$, where $e = \langle u_1, v_1 \rangle$. Overall, the time complexity of Algorithm 2 is $O(\Delta k_c \cdot \sum_{e \in E_{IC2}} \deg(u_1) + \deg(v_1))$. The space complexity is linear with the size of the updated graph $G'$ (i.e. $O(m')$). □

## 6 EFFICIENT ORDER-BASED INDEX

Though the MDM problem can be solved by single update algorithms proposed above, there are 2 weaknesses: (1) both algorithms are only able to handle to-be-updated edges one by one, (2) Algorithm 2 cannot be proved to show neither boundedness nor relative boundedness for edge insertions. In this section, we propose an order-based index called D-Index, which can perform batch updates. What's more, it can be proved that the batch edge deletion algorithm is bounded and the batch edge insertion algorithm is relatively bounded. We present the index design first and then illustrate its rationale for maintenance. Corresponding batch-update algorithms and query algorithms are further presented in Section 7.

### 6.1 D-Index Overview

The existing index [41] of the $k$-truss stores edges in the non-ascending order of $k$ values. However, there are a pair of values to consider (i.e. $(k_c, k_f)$) for the D-truss, the dominant property of $(k_c, k_f)$ [28] confirms that it is impossible to construct such a single order for the D-truss. This fact also increases the problem dimension for handling two types of trussness compared to $k$-truss, which makes the MDM problem harder. For addressing the difference of hardness, we design our D-Index and batch-update algorithms as follows.

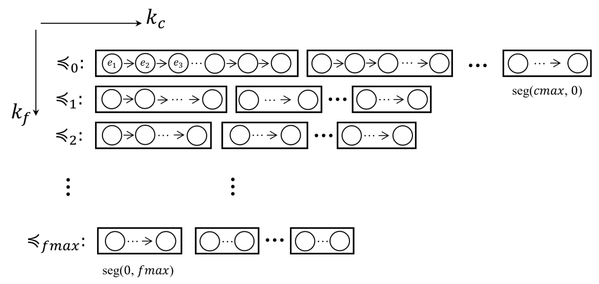


**Figure 3: The structure of D-Index**

**Definition 6.** ***Cycle Decomposition Order (CD Order).*** Given a directed graph $G$ and a flow truss number $k_{f_i}$, the cycle decomposition order (CD order) of $k_{f_i}$ is a sequence of edges in the subgraph $H_{0,k_{f_i}}$ that follows the order of the decomposition along cycle truss numbers, we denote it by $\preceq_{k_{f_i}} = \{e_1, e_2, \cdots, e_{|H|}\}$. For any $\preceq_{k_{f_i}}$ of $G$, it is divided into segments where edges within a segment share the same cycle truss number. We denote a segment by seg$(k_{c_j}, k_{f_i})$.

Then a CD Order $\preceq_{k_{f_i}}$ can be represented by segments $\preceq_{k_{f_i}} = \{\text{seg}(0, k_{f_i}), \text{seg}(1, k_{f_i}), \text{seg}(2, k_{f_i}), \cdots, \text{seg}(k_{c_j}, k_{f_i})\}$. Figure 3 illustrates the structure of the D-Index. Given a directed graph $G$, the D-Index of $G$ is the list of all $\preceq_{k_{f_i}}$ in the ascending order of $k_{f_i}$, we denote the D-Index of $G$ and $G'$ by $\mathbb{I}_D$ and $\mathbb{I}'_D$ respectively.

This index can be inherently constructed by the D-truss decomposition algorithm [28]. Note that there may exist many valid CD orders for a single $k_{f_i}$, since all edges within the same segment can be peeled off in any orders during the process of D-truss decomposition. We keep an arbitrary valid CD order for each $k_{f_i}$ in D-Index and maintain these CD orders one by one for the MDM problem.

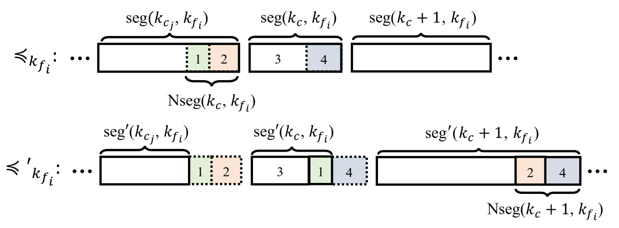
## 6.2 The Rationale of D-Index

**Maintenance in a CD order.** Given the updated edges for this CD order, the maintenance of it is illustrated below. Since we fix $k_{f_i}$ for each $\preceq_{k_{f_i}}$, we only need to decide new cycle truss numbers for all edges in $\preceq_{k_{f_i}}$ and the maintenance can be deemed as the movement of edges among segments. Figure 4 illustrates the pattern of such movement, which is extended from the order index for the $k$-truss [41].

We consider edge insertions first for the ease of illustration, the details for edge deletions are presented in Section 7.1. In Figure 4, we choose an arbitrary CD order $\preceq_{k_{f_i}}$ and itself after maintaining $\preceq'_{k_{f_i}}$. We use 3 segments to show the principle of maintaining this CD order (i.e. $\mathrm{seg}(k_{c_j}, k_{f_i})$, $\mathrm{seg}(k_c, k_{f_i})$ and $\mathrm{seg}(k_c+1, k_{f_i})$), and themselves after applying updates (i.e. $\mathrm{seg}'(k_{c_j}, k_{f_i})$, $\mathrm{seg}'(k_c, k_{f_i})$ and $\mathrm{seg}'(k_c+1, k_{f_i})$). Within $\mathrm{seg}(k_{c_j}, k_{f_i})$, we mark edges whose trussness will change in two boxes, where edges in box1 will move to $\mathrm{seg}'(k_c, k_{f_i})$, and edges in box2 will move to $\mathrm{seg}'(k_c+1, k_{f_i})$. Within $\mathrm{seg}(k_c, k_{f_i})$, we mark edges whose trussness remain same in box3, and mark edges that will move to $\mathrm{seg}'(k_c+1, k_{f_i})$ in box4. We denote the edges set where each edge exists in the $(k_c, k_f)'$-truss but not in the $(k_c, k_f)$-truss by $\mathrm{Nseg}(k_c, k_f)$. In this way, we have $\mathrm{Nseg}(k_c, k_{f_i}) = \{\mathrm{Box1} \cup \mathrm{Box2}\}$, $\mathrm{seg}(k_c, k_{f_i}) = \{\mathrm{Box3} \cup \mathrm{Box4}\}$, $\mathrm{Nseg}'(k_c, k_{f_i}) = \{\mathrm{Box2} \cup \mathrm{Box4}\}$ and $\mathrm{seg}'(k_c, k_{f_i}) = \{\mathrm{Box1} \cup \mathrm{Box3}\}$. We can regard all these unions as additions because each pair of boxes is disjoint, we naturally obtain that $\mathrm{Nseg}(k_c, k_{f_i}) \cup \mathrm{seg}(k_c, k_{f_i}) = \mathrm{Nseg}(k_c+1, k_{f_i}) \cup \mathrm{seg}'(k_c, k_{f_i})$. For each CD order and itself after maintaining, we have such Nseg − seg relationship hold for all flow truss numbers $k_{f_i}$.

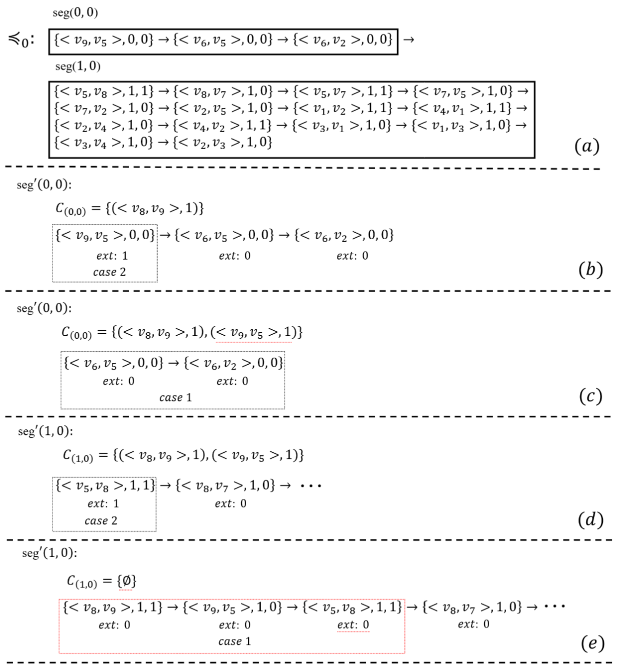
Let $e^*$ be the edge we currently visit and we initialize it as the first one in $\preceq_{k_{f_i}}$, two sets $C_{(k_c,k_f)}$ and $R_{(k_c,k_f)}$ are prepared for storing $\mathrm{Nseg}(k_c+1, k_{f_i})$ and $\mathrm{seg}'(k_c, k_{f_i})$. First, we initialize $C_{(k_c,k_{f_i})} = \mathrm{Nseg}(k_c, k_{f_i})$ and $R_{(k_c,k_{f_i})} = \emptyset$. For any edge in $\mathrm{seg}(k_c, k_{f_i})$ that is visited before $e^*$, if it exists in $\mathrm{Nseg}(k_c, k_{f_i})$, then it can be checked whether it exists in $\mathrm{seg}'(k_c, k_{f_i})$ or not. We move it into $R_{(k_c,k_{f_i})}$ if the answer is yes, and move it into $C_{(k_c,k_{f_i})}$ otherwise.

We construct and maintain 3 auxiliary variables for the correctness of maintenance in each CD order, including $rem_{\preceq}(\cdot)$, $s(\cdot)$ and $ext(\cdot)$. For $\forall e \in \preceq_{f_i}$, we denote the number of cycle support of $e$ which exist in the edges after $e$ in $\preceq_{f_i}$ by $rem_{\preceq}(e)$. For $\forall e \in C_{(k_c,k_{f_i})}$, we maintain $s(e)$ that is equal to the number of cycle support of $e$ in $(k_c, k_{f_i})'$-truss $\setminus R_{(k_c,k_{f_i})}$. For the current visiting $e^*$ and edges after $e^*$ in $\preceq_{f_i}$, $ext(e)$ is maintained such that $ext(e) + rem_{\preceq}(e)$ always equal the number of cycle support in the subgraph which consists of $C_{(k_c,k_{f_i})}$ and the edges after $e$ in $\preceq_{f_i}$. We design different procedures for processing all 3 cases of $e^*$, that is case 1 ($ext(e^*) = 0$), case 2 ($ext(e^*) > 0$ and $ext(e^*) + rem_{\preceq}(e^*) > k_c$) and case 3 ($ext(e^*) > 0$ and $ext(e^*) + rem_{\preceq}(e^*) < k_c$). For case 1, we can verify that $e^*$ exists in $\mathrm{seg}'(k_c, k_{f_i})$ straightforwardly, so we remove it from the CD order and move it into $R_{(k_c,k_{f_i})}$, all $s(\cdot)$ and $ext(\cdot)$ remain same. For case 2, $e^*$ may exist in the $(k_c+1, k_{f_i})$-truss of $G'$, we keep it in the CD order and add it into $C_{(k_c,k_{f_i})}$, all corresponding auxiliary variables need to be maintained according to their definitions. For case 3, $e^*$ cannot exist in the $(k_c+1, k_{f_i})$-truss of $G'$, we remove it from the CD order and move it into $R_{(k_c,k_{f_i})}$, meanwhile, we remove any edge $e \in C_{(k_c,k_{f_i})}$ recursively such that $s(e) \le k_c$, and update auxiliary variables.



$\mathrm{seg}(k_{c_j}, k_{f_i})$
$\mathrm{seg}(k_c, k_{f_i})$
$\mathrm{seg}(k_c+1, k_{f_i})$
$\preccurlyeq_{k_{f_i}}$: ⋯
1
2
3
4
⋯
$\mathrm{Nseg}(k_c, k_{f_i})$
$\mathrm{seg}'(k_{c_j}, k_{f_i})$
$\mathrm{seg}'(k_c, k_{f_i})$
$\mathrm{seg}'(k_c+1, k_{f_i})$
$\preccurlyeq'_{k_{f_i}}$: ⋯
1
2
3
1
4
2
4
⋯
$\mathrm{Nseg}(k_c+1, k_{f_i})$

**Figure 4: The principle of order-based maintenance idea**



seg(0, 0)
$\preccurlyeq_0$: $\{<v_9,v_5>,0,0\} \rightarrow \{<v_6,v_5>,0,0\} \rightarrow \{<v_6,v_2>,0,0\} \rightarrow$
seg(1, 0)
$\{<v_5,v_8>,1,1\} \rightarrow \{<v_8,v_7>,1,0\} \rightarrow \{<v_5,v_7>,1,1\} \rightarrow \{<v_7,v_5>,1,0\} \rightarrow$
$\{<v_7,v_2>,1,0\} \rightarrow \{<v_2,v_5>,1,0\} \rightarrow \{<v_1,v_2>,1,1\} \rightarrow \{<v_4,v_1>,1,1\} \rightarrow$
$\{<v_2,v_4>,1,0\} \rightarrow \{<v_4,v_2>,1,1\} \rightarrow \{<v_3,v_1>,1,0\} \rightarrow \{<v_1,v_3>,1,0\} \rightarrow$
$\{<v_3,v_4>,1,0\} \rightarrow \{<v_2,v_3>,1,0\}$
(a)
seg′(0, 0):
$C_{(0,0)} = \{(<v_8,v_9>,1)\}$
$\{<v_9,v_5>,0,0\} \rightarrow \{<v_6,v_5>,0,0\} \rightarrow \{<v_6,v_2>,0,0\}$
ext: 1
case 2
ext: 0
ext: 0
(b)
seg′(0, 0):
$C_{(0,0)} = \{(<v_8,v_9>,1),(<v_9,v_5>,1)\}$
$\{<v_6,v_5>,0,0\} \rightarrow \{<v_6,v_2>,0,0\}$
ext: 0
ext: 0
case 1
(c)
seg′(1, 0):
$C_{(1,0)} = \{(<v_8,v_9>,1),(<v_9,v_5>,1)\}$
$\{<v_5,v_8>,1,1\} \rightarrow \{<v_8,v_7>,1,0\} \rightarrow \cdots$
ext: 1
case 2
ext: 0
(d)
seg′(1, 0):
$C_{(1,0)} = \{\emptyset\}$
$\{<v_8,v_9>,1,1\} \rightarrow \{<v_9,v_5>,1,0\} \rightarrow \{<v_5,v_8>,1,1\} \rightarrow \{<v_8,v_7>,1,0\} \rightarrow \cdots$
ext: 0
ext: 0
ext: 0
case 1
ext: 0
(e)

**Figure 5: The illustration of Example 6.1**

**Example 6.1.** Consider the graph $G$ in Figure 1(a), assume that we insert a new edge $\langle v_8, v_9 \rangle$ to $G$, and the procedures that how $\preceq_0$ is maintained to $\preceq'_0$ are in Figure 5. We underline changes between consecutive subgraphs by red dashed lines. Figure 5 (a) shows a feasible CD order $\preceq_0$, which consists of $\mathrm{seg}(0,0)$ and $\mathrm{seg}(1,0)$. Note that each edge is stored in the format of $(e, k_c(e), rem_{\preceq}(e))$. After the insertion of $\langle v_8, v_9 \rangle$, we start to maintain $\mathrm{seg}(0,0)$ to $\mathrm{seg}'(0,0)$. First, in Figure 5 (b), we initialize $C_{(0,0)} = \{(\langle v_8, v_9 \rangle, 1)\}$ in the format of the pair of $(e, s(e))$, and process $C_{(0,0)}$ as in lines 5-15 in Algorithm 4, where $C_{(0,0)}$ remains unchanged. Then we start to process edges in $\mathrm{seg}(0,0)$, since $\langle v_9, v_5 \rangle$ belongs to case 2, so it is added to $C_{(0,0)}$. In Figure 5 (c), the remained two edges belong to case 1, by now $\mathrm{seg}'(0,0)$ is obtained and $C_{(0,0)} = \mathrm{Nseg}(1,0)$. In Figure 5 (d), we focus on $\text{seg}(1,0)$ and initialize $C_{(1,0)} = \{(\langle v_8, v_9\rangle, 1), (\langle v_9, v_5\rangle, 1)\}$, the checking turns out that they will be removed from $C_{(1,0)}$ and added at the head of $\text{seg}'(1,0)$. Figure 5 (e) shows $\langle v_8, v_9\rangle$ and $\langle v_9, v_5\rangle$ belong to case 1. Due to the change of $C_{(1,0)}$, $ext(\langle v_5, v_8\rangle)$ changes from 1 to 0, its case also changes from case 2 to case 1. By now all remained edges belong to case 1 thus are ommited. $C_{(1,0)}$ becomes empty, thus the maintenance to $\preceq'_0$ is finished.

**Maintenance among CD orders.** Instead of simply maintaining all orders independently, we utilize the dominant property of D-truss to prune unnecessary to-be-checked orders and accelerate the maintenance process as follows. It is obvious that all edges within $\preceq_{k_{f_i}}$ exactly form the subgraph $H_{0,k_{f_i}}$. For correctly maintain the index, each $\preceq'_{k_{f_i}}$ should form $H'_{0,k_{f_i}}$ after applying edge updates. We start maintaining D-Index from $\preceq_0$ given edge updates $E^+$, instead of maintaining CD orders one by one simply, the squeezing search based on the dominant property of D-truss can be conducted below.

For edge deletions, we first obtain $\preceq'_{k_{f_i}}$ by maintaining the CD order $\preceq_{k_{f_i}}$, then we need to maintain $\preceq_{k_{f_i}+1}$, based on the dominant property, we can pick a tighter upper bound for the cycle truss number of any edge $e$ in $\preceq_{k_{f_i}+1}$: $\{\min(k_c^1, k_c^2) \mid e \in \text{seg}(k_c^1, k_{f_i}+1), e \in \text{seg}'(k_c^2, k_{f_i})\}$. For edge insertions, the cycle truss of $e$ has a lower bound $k_c^1$ exist in $\preceq_{k_{f_i}+1}$, it also has an upper bound $k_c^2$ exist in $\preceq'_{k_{f_i}}$ for $\forall e \in \preceq_{k_{f_i}+1}$. Then the maintenance of $\preceq_{k_{f_i}+1}$ can be conducted more efficiently from both the head and the tail of the CD order. Note that we need to update $E^+$ by $E(H_{0,k_{f_i}+1}) \backslash E(H'_{0,k_{f_i}+1})$ for the next to-be-maintained CD order, and if $E^+$ becomes empty, which implies that no updated edges exist in the next CD order and all remained CD orders are unchanged, in this way, we can early-stop the maintenance correctly and efficiently.

## 7 INDEX-BASED ALGORITHMS

In this section, we propose the solutions for the problem of MDM and MDSM based on the D-Index, including the algorithms for batch edges deletion and insertion respectively for the MDM problem, the query algorithm for the MDSM problem, together with theoretical analysis for these algorithms.

### 7.1 Batch Edges Deletion Algorithm

We denote the batch of to-be-deleted edges by $E^-$. We present our batch edges deletion algorithm in Algorithm 3. We initialize variables in line 1, note that $E_{up}$ is initialized as $E^-$ and is updated for the next CD order in lines 2-19. In lines 3-7, we start maintaining trussness sets for edges in $E_{up}$ by initializing the queue. In lines 8-16, we maintain the order by auxiliary variables $rem_{\preceq}(\cdot)$ and $ts(\cdot)$, where $ts(e)$ is defined as the number of cycle triangles where the other two edges $e'$ and $e''$ both have their cycle truss numbers no smaller than $k_c$ of $e$. Note that all existing trussness sets are obtained from $\mathbb{I}_D$. In lines 17-19, we update the flow truss numbers of edges that exist in the next CD order by picking a tighter upper bound to squeeze the search space. We update $E_{up}$ and necessary variables in line 20 and return the result in line 21.

**Algorithm 3:** Batch Edges Deletion Algorithm

```
   Input: $G, E^-, \mathbb{I}_D$
   Output: $\mathbb{I}'_D$
1  $G' \leftarrow G \backslash E^-$; $E_{up} \leftarrow E^-$; $Queue \leftarrow \emptyset$; $k_c \leftarrow 0$ ;
2  while $!E_{up}.empty()$ do
3      for each edge $e^- \in E_{up}$ do
4          Remove $e^-$ from $\preceq_{k_f}$;
5          Update $rem_{\preceq}(\cdot)$ and $ts(\cdot)$ for edges that form a $\triangle$ with
             $e^-$;
6          for each $e$ with $ts(e) < k_c(e)$ in line 5 do
7              if $e \in Queue$ and $e \notin E^-$ then $Queue \leftarrow e$;
8      while $!Queue.empty()$ do
9          $e \leftarrow Queue.top()$; $k'_c(e) \leftarrow \max\{k_c \mid |\triangle_e^{k_c}| \geq k_c\}$;
10         for each $\triangle$ that contains $e$ in $G'$ do
11             Let $e'$ and $e''$ be the other two edges of $\triangle$;
12             Update $ts(e')$ by checking $k'_f(e), k_f(e'), k_f(e'')$;
13             if $ts(e') < k_f(e')$ and $e' \notin Queue$ then
                 $Queue \leftarrow e'$ ;
14             Repeat line 12-13 for $e''$;
15         Append $e$ into $\preceq'_{k_f}$;
16         Update $rem_{\preceq}(\cdot)$ and $ts(\cdot)$ for edges that form cycle
             triangles with $e$;
17     Conduct maintenance from $k_{cmax}$ similar to lines 8-16;
18     for each $e \in \preceq_{k_f+1}$ do
19         $k_c(e) \leftarrow \{\min(k_c^1, k_c^2) \mid e \in \text{seg}(k_c^1, k_{f_i}+1), e \in$
             $\text{seg}'(k_c^2, k_f)\}$
20     $E_{up} \leftarrow E(H_{0,k_f+1}) \backslash E(H'_{0,k_f+1})$; $Queue \leftarrow \emptyset$; $k_f \leftarrow k_f + 1$;
21 return $\mathbb{I}'_D$;
```

### 7.2 Batch Edges Insertion Algorithm

We denote the batch of to-be-inserted edges by $E^+$. Our batch edges insertion algorithm is presented in Algorithm 4. We initialize variables in line 1. As illustrated in Section 6.2, $E_{up}$ is initialized and maintained in lines 2-33. We initialize $C_{(k_c,k_f)}$ is initialized in line 5 and is processed in lines 6-13. In lines 14-15, we update $ext(\cdot)$ for edges after $e^*$ inclusively in $\preceq_{k_f}$. In lines 16-26, process the edge according to 3 cases of different $ext(\cdot)$ and $rem_{\preceq}(\cdot)$ mentioned in Section 6.2. We initialize $\text{Nseg}_{(k_{c+1},k_f)}$ with remained edges in $C_{(k_c,k_f)}$ in line 27 and update cycle truss numbers in $\text{Nseg}_{(k_c,k_f)} \backslash \text{Nseg}_{(k_{c+1},k_f)}$ in line 28. In lines 30-32, we squeeze the search space for the maintenance of each edge. We update $E_{up}$ and necessary variables in line 33. Note that OIns and ODel can be done in $O(1)$ based on the implementation of the order maintenance algorithm [11].

### 7.3 Fully-Dynamic Search Maintenance

Based on the batch edges update algorithms above, we present the fully-dynamic MDSM query algorithm in Algorithm 5. For the given batch edges update which consists of batch edges insertion and deletion, we remove the insertion and deletion of the same edge first, then move deletions in the first half of the updates, and leave insertions in the second half of updates. In this way, we maintain $\mathbb{I}_D$ by calling Algorithm 3 and Algorithm 4 in line 1. In lines 2-3, if the previous result $H$ is empty, then we need to compute this new query from scratch. In lines 5-8, we consider all edges in the given query result $\mathcal{H}_{k_c,k_f,Q}(G)$ before applying edge updates, find those whose trussnesses can dominate or equal $(k'_c, k'_f)$ and mark them as *kept*.

```
Algorithm 4: Batch Edges Insertion Algorithm
  Input: $G$, $E^+$, $\mathbb{I}_D$
  Output: $\mathbb{I}'_D$
1  $G' \leftarrow G \oplus E^+$; $E_{up} \leftarrow E^+$; $Queue \leftarrow \emptyset$; $k_f \leftarrow 0$;
     $\mathsf{Nseg}_{(k_c,k_f)} \leftarrow E^+$;
2  while $!E_{up}.empty()$ do
3      $k_c \leftarrow 0$;
4      while $!\mathsf{Nseg}_{(k_c,k_f)}.empty()$ do
5          $e^* \leftarrow \preceq_{k_f}.top()$; $C_{(k_c,k_f)} \leftarrow \mathsf{Nseg}_{(k_c,k_f)}$ with $s(e)$
             correctly initialized for $\forall e \in C_{(k_c,k_f)}$;
6          while $\exists e \in C_{(k_c,k_f)}$ with $s(e) \leq k_c$ do
7              $rem_{\preceq}(e) \leftarrow s(e)$;
8              for each $\triangle$ that contains $e$ do
9                  Let $e'$ and $e''$ be other two edges of $\triangle$;
10                 if $\triangle \notin G(C_{(k_c,k_f)} \cup E_{\preceq e^*})$ then continue;
11                 if $e' \in C_{(k_c,k_f)}$ then $s(e') \leftarrow s(e') - 1$;
12                 if $e'' \in C_{(k_c,k_f)}$ then $s(e'') \leftarrow s(e'') - 1$;
13             $C_{(k_c,k_f)} \leftarrow C_{(k_c,k_f)} \backslash e$; $\mathsf{OIns}(\preceq_{k_f}, e, e^*)$;
14         for each $e \in C_{(k_c,k_f)}$ do
15             Update $ext(\cdot)$ for edges after $e^*$ inclusively in $\preceq_{k_f}$;
16         while $e^* \neq nil$ do
17             $e^*_{next} \leftarrow$ the edge next to $e^*$ in $\preceq_{k_f}$;
18             if $ext(e^*) = 0$ then
19                 Find the first $e$ in $\preceq_{k_f}$ s.t. $ext(e) > 0$, else break;
20             else if $ext(e^*) + rem_{\preceq}(e^*) > k_c$ then
21                 Add $e^*$ to $C_{(k_c,k_f)}$ with $s(\cdot) = ext(\cdot) + rem_{\preceq}(\cdot)$;
22                 Update $ext(\cdot)$ for edges in $E_{\preceq e^*}$; $\mathsf{ODel}(\preceq, e^*)$;
23             else
24                 Update $rem_{\preceq}(e^*)$, $ext(e^*)$ and $s(\cdot)$ for $C_{(k_c,k_f)}$;
25                 Process all edges that are affected in $C_{(k_c,k_f)}$;
26             $e^* \leftarrow e^*_{next}$;
27         $\mathsf{Nseg}_{(k_c+1,k_f)} \leftarrow C_{(k_c,k_f)}$;
28         for $e \in \mathsf{Nseg}_{(k_c,k_f)} \backslash \mathsf{Nseg}_{(k_c+1,k_f)}$ do $k'_c \leftarrow k_c$;
29         $k_c \leftarrow k_c + 1$;
30     Conduct maintenance from $k_{cmax}$ similar to lines 4-29;
31     for each $e \in \preceq'_{k_f}$ do
32         $k_c(e, k_f + 1) \leftarrow \min\{k_c(e, k_f + 1), k'_c(e, k_f)\}$;
33     $E_{up} \leftarrow E(H'_{k_c+1,0}) \backslash E(H_{k_c+1,0})$; $Queue \leftarrow \emptyset$; $k_f \leftarrow k_f + 1$;
34 return $\mathbb{I}'_D$;
```

```
Algorithm 5: MDSM Query Algorithm
  Input: $G$, $Q$, $k_c$, $k_f$, $\mathcal{H}_{k_c,k_f,Q}(G)$, $\mathbb{I}_D$ and $\Delta G$, $\Delta Q$, $k'_c$, $k'_f$
  Output: $\mathcal{H}_{k'_c,k'_f,Q'}(G \oplus \Delta G)$
1  $H \leftarrow \mathcal{H}_{k_c,k_f,Q}$; Obtain $\mathbb{I}'_D$ by Algorithm 3 and Algorithm 4;
2  if $H$ is empty then
3      $H \leftarrow$ the result by the query algorithm [28] with
         $k'_c, k'_f, Q \oplus \Delta Q$ input;
4  else
5      for each $e \in E(H)$ do
6          for each $(k_{c_i}, k_{f_i}) \in \mathcal{ST}'(e)$ do
7              if $(k_{c_i}, k_{f_i}) \preceq (k'_c, k'_f)$ then Mark $e$ as kept; break;
8          if $e$ is NOT kept then $H \leftarrow H \backslash e$;
9      if $H$ is NOT connected then $H \leftarrow \emptyset$; return $H$;
10     $H \leftarrow$ the result by the query algorithm [28] that starts with
         $k'_c, k'_f, H$;
11     for each $u \in V(H)$ do Mark $u$ as visited;
12     for each $q' \in \Delta Q$ do
13         if $q'$ is NOT visited then $H \leftarrow \emptyset$; return $H$ ;
14 return $H$;
```

After inspecting trussnesses, we check the connectivity of $H$ in line 9. In line 10, we conduct searching based on $H$ by the index-based query algorithm [28]. In lines 11-13, we check the containment of $Q'$ of the result community $H$.

**The optimization of leveraging all previous results.** Algorithm 5 is only able to maintain the query result from the last one. When the results of two consecutive queries are not relevant, the extra checking procedures make the query maintenance algorithm worse than direct query algorithm. It is strongly motivated that making Algorithm 5 utilizes of all previous query results. The optimization can be extended based on Algorithm 5 as follows. With storing previous queries and their result subgraphs in advance, when we receive a new query $Q_{i+1}$, we can check the existence of query vertices in all previous result subgraphs $H_i, \forall i$ and obtain a candidate set of result subgraphs. Then we compute the difference of $\Delta G_{i+1}$ and each $\Delta G$ in the candidate set, where the difference is defined as the number of different deletions or insertions. Finally we select the previous query and its result subgraph with the smallest difference as the input of Algorithm 5.

## 7.4 Theoretical Analysis

**Boundedness of Algorithm 3.** The Algorithm 3 is bounded to the polynomial function of |CHANGED|, which is shown in Theorem 7.1. Our proposed index $\mathbb{I}_D$ can handle batch edges deletion boundedly by Algorithm 3, meanwhile, it can handle batch edges insertion relative-boundedly, which is shown in Theorem 7.2.

Theorem 7.1. *For Algorithm 3, the time complexity of is $O(\Delta k_f \cdot T_{\triangle}(\mathsf{CHANGED}) \cdot \max_{e \in \mathsf{CHANGED}} T_{\triangle}(e))$, where $T_{\triangle}(e)$ denotes the time of listing all triangles that contain $e$, and the space complexity is $O(m')$.*

Proof. Consider Algorithm 3, the loop in line 2 is bounded by the gap of CHANGED whose flow truss numbers change (i.e. $\Delta k_f$). In lines 3-7, the dominating cost is $O(T_{\triangle}(E^-))$. In the loop in lines 8-16, the cost is bounded by $O(T_{\triangle}(\mathsf{CHANGED}))$ since only edges in |CHANGED| can be pushed into $Queue$. Thus, the time complexity is $O(\Delta k_f \cdot T_{\triangle}(\mathsf{CHANGED}) \cdot \max_{e \in \mathsf{CHANGED}} T_{\triangle}(e))$. The space complexity of auxiliary variables and the size of $\mathbb{I}_D$ is bounded by the size of the updated graph $E(G')$ (i.e. $O(m')$). □

**Formulating** $\mathsf{AFF}_{\mathsf{k_f}}$**.** Given the batch edges update $\Delta G$, we consider two valid CD orders $\preceq_{k_f}$ and $\preceq^1_{k_f}$. We define the difference between them as $\mathsf{dif}(\preceq_{k_f}, \preceq^1_{k_f})$,

$$\mathsf{dif}(\preceq_{k_f}, \preceq^1_{k_f}) = \mathsf{CHANGED} \cup \{e_1 \in G \mid \exists e_2 \in G, s.t. e_1 \preceq_{k_f} e_2 \text{ and } e_2 \preceq^1_{k_f} e_1\}$$

, where $e_1 \preceq_{k_f} e_2$ means that $e_1$ appears before $e_2$ in the CD order $\preceq_{k_f}$, also implies that the cycle truss number of $e_1$ is no bigger than $e_2$'s. Then the $\mathsf{AFF}_{\mathsf{k_f}}$ of the CD order $\preceq_{k_f}$ is defined as,

$$\mathsf{AFF}_{\mathsf{k_f}} = \bigcap_{\preceq^1_{k_f}} \mathsf{dif}(\preceq_{k_f}, \preceq^1_{k_f}).$$

**Relative boundedness of Algorithm 4.** The MDM problem is unbounded for edge insertions, which is proved in Section 4. By formulating $\mathsf{AFF}_{\mathsf{k_f}}$ and proposing the index $\mathbb{I}_D$, we prove that the Algorithm 4 is relative bounded below.

Theorem 7.2. *The time complexity of Algorithm 4 is* $O(\Delta k_f \cdot |\mathsf{AFF}_{\mathsf{k_f}}| \, \|\mathsf{AFF}_{\mathsf{k_f}}\|_1^2 \cdot \log(|\mathsf{AFF}_{\mathsf{k_f}}| \, \|\mathsf{AFF}_{\mathsf{k_f}}\|_1^2))$, *and the space complexity is* $O(m')$.

Proof. For Algorithm 4, the outer loop in line 2 is bounded by $\Delta k_f$. In lines 4-15, the cost is bounded by $O(\sum_{k_f} T_\triangle(\mathsf{Nseg}_{(k_c,k_f)}))$. In lines 22 and 24, the cost is bounded by $O(T_\triangle(E_2^{\leq k_f} + E_3^{\leq k_f}))$, where $E_2^{\leq k_f}$ and $E_3^{\leq k_f}$ denote the edge sets which are visited in case 2 and case 3 in Section 6.2. Thus in lines 5-30, the total cost is $O(T_\triangle(E_2^{\leq k_f} + E_3^{\leq k_f}) + \sum_{k_f} T_\triangle(\mathsf{Nseg}_{(k_c,k_f)}))$. Furthermore, it can be obtained easily based on the proof [41] that $O(T_\triangle(E_2^{\leq k_f} + E_3^{\leq k_f}) + \sum_{k_f} T_\triangle(\mathsf{Nseg}_{(k_c,k_f)}))$ is bounded by $|\mathsf{AFF}_{\mathsf{k_f}}| \, \|\mathsf{AFF}_{\mathsf{k_f}}\|_1^2$. Overall, the time complexity is $O(\Delta k_f \cdot |\mathsf{AFF}_{\mathsf{k_f}}| \, \|\mathsf{AFF}_{\mathsf{k_f}}\|_1^2 \cdot \log(|\mathsf{AFF}_{\mathsf{k_f}}| \, \|\mathsf{AFF}_{\mathsf{k_f}}\|_1^2))$. Note that the logarithmic term generates from the heap-based implementation of the order structure. The space complexity of auxiliary variables and the size of $\mathbb{I}_D$ is bounded by $O(m')$. □

Theorem 7.3. *The time complexity of Algorithm 5 is* $O(k_{fmax} \cdot m)$, *the space complexity of Algorithm 5 is* $O(m')$.

Proof. Consider the worst case of the query maintenance in lines 2-3, the time cost of Algorithm 5 is equal to recompute from scratch by the query algorithm [28], which is bounded by $O(k_{fmax} \cdot m)$. The space complexity of the index $\mathbb{I}'_D$ is bounded by the size of the updated graph $E(G')$ (i.e. $O(m')$). □

## 8 EXPERIMENTS

In this section, we conduct all experiments on a Linux machine with an Intel Xeon Gold 6240R CPU @ 2.40GHz and 1007GB main memory.

**Datasets.** We use 7 real-world datasets which are publicly accessible, including Email (EM), Twitter (TW), BerkStan (BS), Wiki (WK), Pokec (PK) from SNAP [26], Edinburgh Associative Thesaurus (EAT) from Pajek [6], and Youtube Links (YL) and DBpedia Links (DL) from KONECT [24]. The summary of datasets is shown in Table 2. Note that DL cannot be decomposed within $6 \times 10^4$ seconds, thus we sample subgraphs of it at different rates in Exp-2 below.

**Algorithms.** We evaluate the following algorithms in this paper. For the MDS problem: (1) the D-truss decomposition algorithm Dec (the decomposition algorithm [28] with $G \oplus \Delta G$ input). For the MDM problem: (1) the single deletion algorithm SDel (Algorithm 1), (2) the single insertion algorithm SIns (Algorithm 2), (3) the batch deletion algorithm with unit edge processing UBDel and with batch edge processing BDel (Algorithm 3) and (4) the batch insertion algorithm with unit edge processing UBIns and with batch edge processing BIns (Algorithm 4). For the MDSM problem: (1) ReQry (the query algorithm [28] with $G \oplus \Delta G$ input), (2)CoQry (the query algorithm [28] after being maintained by Algorithm 3 and Algorithm 4), (3) MtQry (Algorithm 5) and (4) OpQry (Algorithm 5 with the optimization). All algorithms are implemented in C++. In all experiments, the $y$-axis denotes the running time in logarithmic scale, and we end algorithms which run longer than $2 \times 10^4$ seconds.

**Table 2: Summary of Datasets**

| Dataset | $\lvert V \rvert$ | $\lvert E \rvert$ | $\deg^{in}_{max}$ | $\deg^{out}_{max}$ | $k_{cmax}$ | $k_{fmax}$ |
|---|---|---|---|---|---|---|
| EM | 1.0K | 25.6K | 211 | 333 | 14 | 21 |
| EAT | 23.1K | 685K | 1073 | 78 | 3 | 8 |
| TW | 81.3K | 1.8M | 3, 383 | 1, 205 | 161 | 199 |
| YL | 1.1M | 4.9M | 25, 487 | 28, 564 | 19 | 38 |
| BS | 685K | 7.6M | 84, 208 | 249 | 41 | 80 |
| WK | 1.8M | 28.5M | 238, 040 | 3, 907 | 36 | 37 |
| PK | 1.6M | 30.6M | 13, 733 | 8, 763 | 18 | 27 |
| DL | 18.2M | 136.5M | 612, 308 | 8, 105 | −* | −* |

* The decomposition of the original dataset cannot be finished within $6 \times 10^4$ seconds.

### 8.1 Effectiveness Evaluation

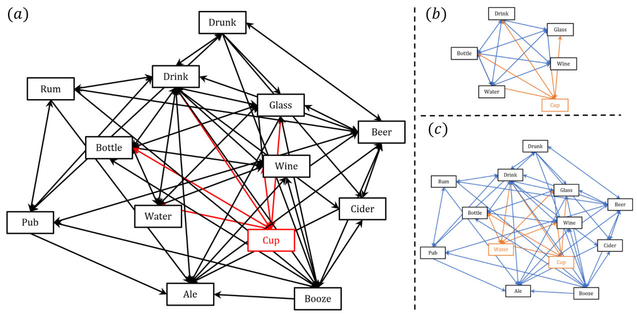


**Figure 6: Effectiveness evaluation**

EAT [6] is a word network which is built from the Edinburgh Associative Thesaurus (EAT). In the EAT network, the vertices denote the words, and the edge $\langle w_i, w_j \rangle$ from $w_i$ to $w_j$ means that the word $w_i$ points to another word $w_j$ if $w_j$ appears as the response when $w_i$ is received as the stimulus.

For clearly illustrating the use case, we sample the subgraph such that in Figure 6(a). Consider the following two queries for the word "drink", one is $Query1 : \{Q =$"drink"$, k_c = 3, k_f = 0\}$, the other is $Query2 : \{Q =$"drink"$, k_c = 2, k_f = 4\}$. The updated edges $E^+$ are marked red in Figure 6(a). Figure 6(b) and (c) show the query results before and after applying updated edges.

$Query1$ desires a $(3, 0)$-truss of the word "drink", before applying the updated edges, the result subgraph consists of "glass", "bottle", "wine" and "water". After maintaining by Algorithm 4, the new $(3, 0)$-truss of "drink" also contains "cup", which shares an equal relationship with all words in old result.

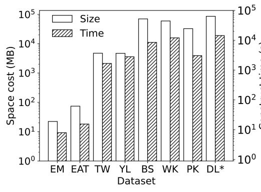
Considers $Query2$, a $(2, 4)$-truss of the word "drink" is returned, many words are contained in the old result subgraph, form a mixed structure around "drink". They are participated with the community of "drink" both in a relatively weak equal relationship and a relatively strong hierarchical relationship. Together with the result of $Query1$, we know that the words "glass", "bottle", "wine" forms a stronger equal relationship with "drink". The word "drunk" is in the upstream of "drink", which means that it always reminds people of "drink", while the words "booze", "cider", "beer", "rum", "ale" are in the downstream of "drink", which means that when people are given "drink" as the stimulus, these alcoholic words arise as a response. After maintaining by Algorithm 4, the new $(2, 4)$-truss of "drink" also contains "cup" and "water", which enlarge the community with a relatively strong equal relationship.

## 8.2 Efficiency Evaluation

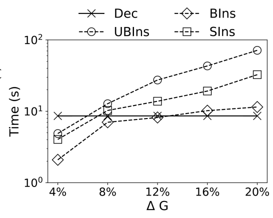
Due to page limitations, we only show experimental results on selected datasets, complete results of all datasets are available here.
**Exp-1: Index construction.** We show the index construction time and index size in Figure 7(a). For all the datasets, the D-Index can be built within $2 \times 10^4$ seconds, and the index size is within 85 GB. Such construction costs are acceptable for real world applications.

For evaluating the efficiency of solutions for the MDM problem, we design 2 experiments below by varying $|\Delta G|$ and $|G|$ respectively.
**Exp-2: Varying $|\Delta G|$.** We conduct the experiment of investigating the impact of $|\Delta G|$ over EM and TW. This testing includes all algorithms mentioned above. Note that given a directed graph $G$, its updates $\Delta G$ are sampled from $E(G)$ randomly, in this way, we set $G$ as the original graph for edge deletions and set $G \backslash \Delta G$ as the original graph for edge insertions. We vary $|\Delta G|$ from 4% to 20% of the original graph size. The running times of all algorithms on each graph are shown in Figure 7 (b), (c), (d), (e) respectively. The results show that, (1) BIns performs better than SIns and UBIns consistently, and BIns runs faster than Dec with $\Delta G$ of around 12% for EM, and of around 10% for TW; (2) BDel also performs better than SDel and UBDel consistently, and BDel is more efficient than Dec with $\Delta G$ of around 20% for both datasets; (3) SIns and SDel perform better than UBIns and UBDel coordinately, which implies that single-udpate algorithms are more efficient than the D-Index because of index-related processing are costly than index-free operators under the single-update setting; (4) BIns runs at most 4.12x and 1.63x faster than Dec for EM and TW respectively, and BDel runs at most 6.89x and 2.94x faster than Dec for two datasets. We observe that BDel is more efficient than BIns, because BDel is bounded to CHANGED, while BIns is unbounded. $|\Delta G|$ is usually very small relative to $|G|$ in real world [15], BDel and BIns are always more efficient than Dec with small $|\Delta G|$.
**Exp-3: Varying $|G|$.** The impact of $|G|$ is evaluated on 2 largest datasets PK and DL with 30.6M edges and 136.5M edges respectively. We also obtain 5 subgraphs for PK by randomly sampling from the original graph at rates of 20%, 40%, 60%, 80% and 100%. Note that $G_{20\%} \subseteq G_{40\%} \subseteq G_{60\%} \subseteq G_{80\%} \subseteq G_{100\%}$ is guaranteed. The results of PK are shown in Figure 7 (f) and (h). We sample DL similarly at rates of 20%, 25%, 30%, 35% and 40%, and results are shown in Figure 7 (g) and (i). In this experiment, we apply the same $\Delta G$ for these sampled subgraphs of each dataset, where $|\Delta G|$ is 1% of $|G|$. The usage of $|\Delta G|$ for insertions and deletions is exactly as same as Exp-2. We see that Dec runs super-linearly with the size of the graph for both PK and DL. Take Dec as the baseline method, (1) for edge insertions, BIns and SIns scale well and perform better than Dec as $|G|$ getting larger, but UBIns is not always scalable, especially for very large datasets such as DL, since too many low-level operations are executed in UBIns for each unit update and the time cost of these operations gets longer when $|G|$ gets larger; (2) for edge deletions, all algorithms perform well and BDel performs best in the experiment since all of them are bounded. Similar performance trends are observed from other datasets.

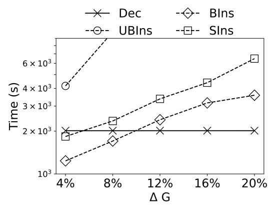
We set up 3 experiments for evaluating the efficiency of the MDSM problem, which vary |previous queries|, $|Q|$, $|\Delta Q|$ and $(k'_c, k'_f)$.
**Exp-4: Vary |previous queries|.** In this experiment, we generate 100 queries randomly, the queried trussness is guaranteed to be dominated by $(k_{cmax}, k_{fmax})$. Note that generated queries are not necessarily valid, since the solution should also be efficient to judge invalid queries. We construct $\Delta G$ in the following way: given the original graph $G$, we obtain $\Delta G$ by sampling 1% edges of $G$, and insert a half of $\Delta G$ back, note these edges as deletions and note the other half of edges as insertions. In this way, $\Delta G$ is composed of a half of edge deletions and the other half of edge insertions. We set $|Q| = 4$, $\Delta Q$ as null and keep $(k'_c, k'_f)$ same as $(k_c, k_f)$. We vary the number of previous queries of $19, 39, 59, 79, 99$. For ReQry, CoQry and MtQry, we report the average query time by each point; for CoQry, we report the exact query time at each point. In Figure 7 (k) and (l), we show the results on YL and WK respectively. When the number of previous queries gets larger, the query time changes slightly for ReQry, CoQry and MtQry. OpQry is faster than other methods on almost all points. For the point with 39 previous queries, the time cost of OpQry decreases compared to 19 previous queries, but keeps increasing as the number of previous queries gets larger. Because with more previous results, the checking procedures of them take longer time, which formulates a trade-off between the checking procedures and the maintenance itself. Similar performance trends are observed from other datasets.
**Exp-5: Varying $|Q|$ and $|\Delta Q|$.** In this experiment, the generation of queries and $\Delta G$ is similar to Exp-4. We give OpQry other 40 previous results and report average query time of remained 100 queries. In Figure 7 (m), we evaluate the impact of $|Q|$ on YL. We set $\Delta Q$ as null and keep $(k'_c, k'_f)$ same as $(k_c, k_f)$. We vary $|Q|$ of $1, 2, 4, 6$ and $8$, the query time becomes longer for all methods when $|Q|$ gets larger, because it costs longer for checking more query vertices. ReQry and CoQry cost more time than MtQry and OpQry when $|Q|$ gets larger, since the dominating cost of ReQry and CoQry is from the static property of Dec, which has to be executed from scratch when $\Delta G$ is given. MtQry and OpQry show better scalability of $|Q|$. In Figure 7 (n), we evaluate the impact of $|\Delta Q|$ on WK, where we set $|Q| = 4$ and keep $(k'_c, k'_f)$ same as $(k_c, k_f)$. $|\Delta Q|$ is varied from $-3$ to $3$, where $-3$ denotes deleting 3 query vertices out of $Q$ randomly and 3 denotes inserting 3 query vertices into $Q$ randomly. The impact of $|\Delta Q|$ is similar to that of $|Q|$, because when $|\Delta Q|$ gets larger, it needs to check more query vertices. In all tested cases, CoQry, MtQry and OpQry are much more efficient than ReQry. We note that CoQry takes longer time than MtQry in almost all cases, since the maintenance based on existing results spares the cost compared to re-quering. OpQry performs better than MtQry, since OpQry utilizes more previous results instead of only the latest one. Similar performance trends are observed from other datasets.
**Exp-6: Varying $(k'_c, k'_f)$.** In this experiment, the generation of queries and $\Delta G$ is similar to Exp-4. $Q$ and $\Delta G$ remain same as before varying $(k'_c, k'_f)$. We investigate the impact of $k'_c$ and $k'_f$



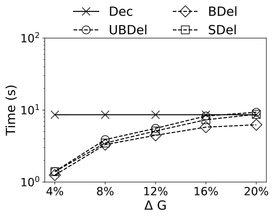
(a) Index construction



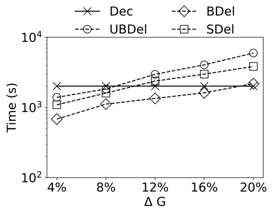
(b) EM, $\Delta G$ insertions



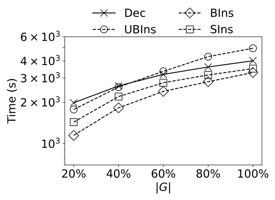
(c) TW, $\Delta G$ insertions



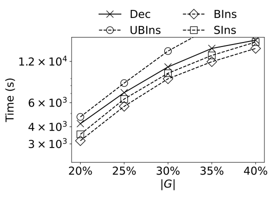
(d) EM, $\Delta G$ deletions



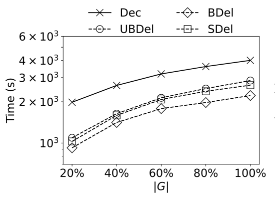
(e) TW, $\Delta G$ deletions



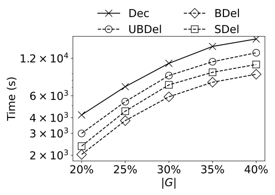
(f) PK, $\Delta G$ insertions



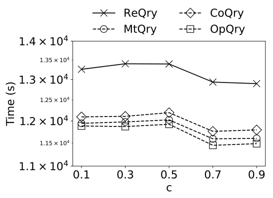
(g) DL, $\Delta G$ insertions



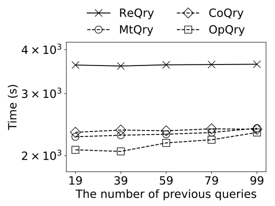
(h) PK, $\Delta G$ deletions



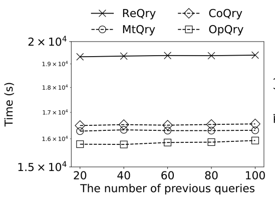
(i) DL, $\Delta G$ deletions



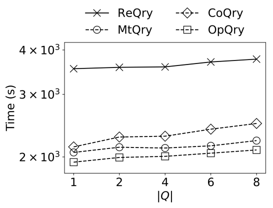
(j) BS, $k'_c = c \cdot k_{cmax}$, $k'_f = 0$



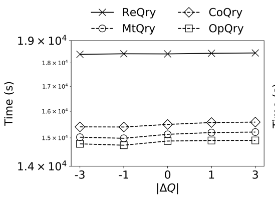
(k) YL, |previous queires|



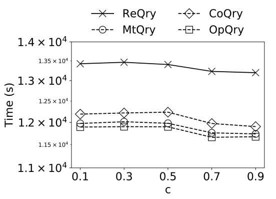
(l) WK, |previous queires|



(m) YL, $|Q|$



(n) WK, $\Delta Q$



(o) BS, $k'_c = 0$, $k'_f = c \cdot k_{fmax}$

**Figure 7: Efficiency Evaluation**

respectively in Figure 7 (j) and (o). Either $k'_c$ or $k'_f$ gets larger, all methods run faster because the size of the result gets smaller, OpQry is the most efficient one among all competitors with the same reason above. Similar performance trends are observed from other datasets.

## 9 CONCLUSIONS

In this paper, we propose a collection of incremental solutions for maximal D-truss search in dynamic graphs, including single-update algorithms and an order-based index that can handle batch updates efficiently. We conduct theoretical boundedness analysis of the maximal D-truss given edge insertions and deletions respectively. We further present a fully-dynamic query algorithm to help accelerate the query processing. Extensive experimental results on large real-world graphs validate the effectiveness and efficiency of our solutions.

## REFERENCES

[1] A. Acquisti and R. Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. *PETS*, pages 36–58, 2006.

[2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. *SODA*, pages 32–42, 1990.

[3] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group recommendation: Semantics and efficiency. *PVLDB*, 2(1):754–765, 2009.

[4] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

[5] J. Bang-Jensen and G. Z. Gutin. *Digraphs: theory, algorithms and applications*. SSBM, 2008.

[6] V. Batagelj and A. Mrvar. Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/, Jan. 2006.

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

[8] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. *ICDE*, pages 51–62, 2011.

[9] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 16(3.1), 2008.

[10] L. Dhulipala, Q. C. Liu, J. Shun, and S. Yu. Parallel batch-dynamic k-clique counting. *APOCS*, pages 129–143, 2021.

[11] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, pages 365–372, 1987.

[12] D. Ding, H. Li, Z. Huang, and N. Mamoulis. Efficient fault-tolerant group recommendation using alpha-beta-core. *CIKM*, pages 2047–2050, 2017.

[13] D. Duan, Y. Li, R. Li, and Z. Lu. Incremental k-clique clustering in dynamic social networks. *Artif. Intell. Rev.*, 38(2):129–147, 2012.

[14] W. Fan and C. Tian. Incremental graph computations: Doable and undoable. *TODS*, 47(2):1–44, 2022.

[15] W. Fan, C. Tian, R. Xu, Q. Yin, W. Yu, and J. Zhou. Incrementalizing graph algorithms. *SIGMOD*, pages 459–471, 2021.

[16] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *PVLDB*, 10(6):709–720, 2017.

[17] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *TKDE*, 31(11):2093–2107, 2018.

[18] Y. Fang, H. Zhang, Y. Ye, and X. Li. Detecting hot topics from twitter: A multiview approach. *J. Inf. Sci.*, 40(5):578–593, 2014.

[19] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowl. Inf. Syst.*, 35(2):311–343, 2013.

[20] R. Guimera and L. A. Nunes Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028):895–900, 2005.

[21] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. *SIGMOD*, pages 1311–1322, 2014.

[22] X. Huang, L. V. Lakshmanan, and J. Xu. Community search over big graphs: Models, algorithms, and opportunities. *ICDE*, pages 1451–1454, 2017.

[23] A. E. Krause, K. A. Frank, D. M. Mason, R. E. Ulanowicz, and W. W. Taylor. Compartments revealed in food-web structure. *Nature*, 426(6964):282–285, 2003.

[24] J. Kunegis. Konect: the koblenz network collection. *WWW*, pages 1343–1350, 2013.

[25] K. J. Lang and R. Andersen. Finding dense and isolated submarkets in a sponsored search spending graph. *CIKM*, pages 613–622, 2007.

[26] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[27] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian. Hierarchical core maintenance on large dynamic graphs. *PVLDB*, 14(5):757–770, 2021.

[28] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao. Truss-based community search over large directed graphs. *SIGMOD*, pages 2183–2197, 2020.

[29] F. Luo, J. Z. Wang, and E. Promislow. Exploring local community structures in large networks. *WIAS*, 6(4):387–400, 2008.

[30] J.-P. Onnela, A. Chakraborti, K. Kaski, J. Kertesz, and A. Kanto. Dynamics of market correlations: Taxonomy and portfolio analysis. *Phys. Rev. E*, 68(5):056110, 2003.

[31] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[32] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.

[33] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási. Hierarchical organization of modularity in metabolic networks. *Science*, 297(5586):1551–1555, 2002.

[34] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *VLDBJ*, 25(3):425–447, 2016.

[35] B. Sun, T.-H. H. Chan, and M. Sozio. Fully dynamic approximate k-core decomposition in hypergraphs. *TKDD*, 14(4):1–21, 2020.

[36] T. Takaguchi and Y. Yoshida. Cycle and flow trusses in directed networks. *R. Soc. Open Sci.*, 3(11):160270, 2016.

[37] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[38] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang. Efficient and effective community search on large-scale bipartite graphs. *ICDE*, pages 85–96, 2021.

[39] T. Yang, Y. Chi, S. Zhu, Y. Gong, and R. Jin. Directed network community detection: A popularity and productivity link model. *ICDM*, pages 742–753, 2010.

[40] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. *VLDBJ*, 25(2):171–196, 2016.

[41] Y. Zhang and J. X. Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. *SIGMOD*, pages 1024–1041, 2019.