Theoretical Computer Science 1044 (2025) 115278

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Maximum core spanning tree maintenance for large dynamic graphs [☆]

Xiaowei Lv, Yongcai Wang, Deying Li [*]

School of Information, Renmin University of China, Beijing 100872, China

A R T I C L E   I N F O

Section Editor: Paul G. Spirakis

Handling Editor: Ding-Zhu Du

Keywords:
Dense subgraph mining
k-core
Maximum core spanning tree
Dynamic maintenance

A B S T R A C T

With the increase in network scale and online applications, the maintenance problem of cohesive structures in large graphs has attracted great attention. The Maximum Core Spanning Tree (MCST) is a representative cohesive structure generated based on k-core, which is the maximum edge weight spanning tree indicating the "staired coreness hierarchy" in each connected component. The edge weight here is defined as $w_{uv} = \min\{core(u), core(v)\}$, and $core(x)$ is the corness of vertex x. Unlike the maintenance problem of Maximum Spanning Tree (MST) which has known efficient algorithms, MCST maintenance raises special challenges, which is mainly due to the cascaded vertex coreness changes after single-edge insertion or deletion. In this paper, we show a series properties of MCST and MCST maintenance problems and propose an OrderPassed method and a LoopFree method to maintain the MCST efficiently. In particular, the time complexity for MCST maintenance for edge insertion and deletion is bounded by $\mathcal{O}(|E^*|+|V|)$ and $\mathcal{O}(|E^*|+\sum_{i=1}^{K}|O_i|)$ respectively, where E^* is the edge set whose edge weight changes after insertion/deletion and $|O_i|$ denotes the number of edges whose edge weight is i. Through extensive evaluations, we show the proposed MCST maintenance algorithms have good efficiency, scalability and stability on real-world datasets.

1. Introduction

Dense subgraph mining of complex networks is widely used in various graph structures, including social networks [2,3], biology [4], and advertisement in e-commerce, etc. Among the dense subgraph models, one of the most widely used is k-core [5]. It extracts connected subgraphs in which vertices have degree at least k, and serves as a foundation for other cohesive graph models, such as k-clique [4], k-ECC [6], k-trusses [7], and so on.

The Maximum Core Spanning Tree (MCST) [8] based on k-core further reveals the inner structure of the graph. In particular, a MCST is the maximum edge weight spanning tree, where the edge weight is defined as $w_{uv} = min\{core(u), core(v)\}$. Note that $core(u)$ represents the coreness of vertex u. We show a MCST has a stair structure, where the tree edges connecting the vertices with the same coreness can be considered on the same stair. In MCST generation process, MCST will greedily connect all the vertices in a higher

☆ This is an enhanced and extended version of a paper [1] presented in The 18th International Conference on Algorithmic Aspects in Information and Management (AAIM 2024). Dr. Li is supported in part by the National Natural Science Foundation of China Grant No. 12071478. Dr. Wang is supported in part by the National Natural Science Foundation of China Grant No. 61972404, Public Computing Cloud, Renmin University of China, and the Blockchain Lab. School of Information, Renmin University of China.

* Corresponding author.
E-mail addresses: lvxiaowei@ruc.edu.cn (X. Lv), ycw@ruc.edu.cn (Y. Wang), deyingli@ruc.edu.cn (D. Li).

https://doi.org/10.1016/j.tcs.2025.115278
Received 18 December 2024; Received in revised form 17 January 2025; Accepted 24 April 2025

Available online 29 April 2025

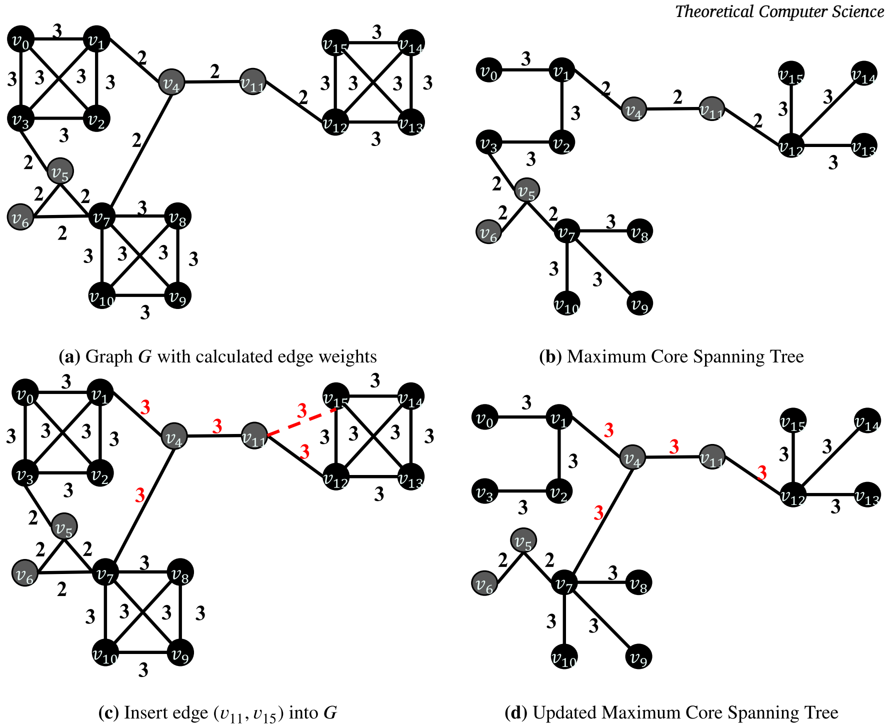

(a) Graph G with calculated edge weights
(b) Maximum Core Spanning Tree
(c) Insert edge (v_{11}, v_{15}) into G
(d) Updated Maximum Core Spanning Tree

Fig. 1. An Example for Maximum Core Spanning Tree Maintenance.

stair before connecting to the low stair vertices. So a MCST indicates a "staired coreness hierarchy" in each connected component, which can be considered as a different hierarchical representation of the k-cores.

In large scale networks, when the vertices and connections are dynamic, how to maintain the extracted cohesive structures efficiently is a crucial problem [9–14]. For tracking the inner structure of k-cores, the maintenance of MCST is important as well, which is however not well investigated in existing studies. In comparison, the maintenance of traditional Maximum Spanning Tree (MST) has attracted a lot of attentions and a series of efficient algorithms are proposed [15–18]. But existing MST maintenance methods are usually designed for weight change of a single edge. Whereas the maintenance of MCST raises special challenges due to the special property of k-core. A single-edge insertion or deletion in the graph may trigger a significant number of cascaded vertex coreness changes [12], so the MCST maintenance must consider simultaneous weight changes of a set of edges after single edge deletion or insertion.

We show the main challenges to maintain MCST are:

(1) How to determine which set of edges will change their weights?

(2) How to update the MCST according to the edge-weight changes?

In response to above challenges, we investigate a series of properties of MCST and for MCST maintenance, including (1) the stair-like structure of MCST; (2) Unique determination of MCST by a given Kruskal-order; and (3) Various principles to narrow down the maintenance scope. Using these properties, we design complexity-bounded maintenance algorithms. In particular, an OrderPassed (OP) algorithm is designed for edge deletion, which reduces the search scope for the changed tree edges by maintaining the Kruskal-order. And a LoopFree (LF) algorithm is designed for edge insertion, which maintains MCST by deleting the smallest weight edge in the loop. Extensive evaluations show the proposed MCST maintenance algorithms have good efficiency, scalability, and stability on real-world datasets.

Example 1. Fig. 1 shows an example of MCST maintenance. Fig. 1(a) shows the weighted graph where the edge weights are determined by the vertex coreness. Fig. 1(b) is the initial MCST. In Fig. 1(c), if we add edges (v_{15}, v_{11}), the coreness values of v_4 and v_{11} will change from 2 to 3, bringing the weight updating of adjacent edges $\{(v_1, v_4), (v_4, v_7), (v_4, v_{11}), (v_{11}, v_{12}), (v_{11}, v_{15})\}$. Now (v_4, v_7) will replace the original edge (v_3, v_5) in the MCST as shown in Fig. 1(d).

2. Related work

Dense subgraph mining. In dense subgraph mining, the standard model structures include the k-clique [19], quasi-clique [20], nucleus [21], k-core [22], [23], [24], [25,26], k-truss [26–29], k-VCC [30], k-plex [31], k-ecc [32,33], LhCDS [34], etc. Among them, the k-core model is the most widely used because of its simple structure and easy maintenance. The core decomposition [35–37,14] is also widely used in various practical scenarios, including community discovery [38–43], network analysis [44], etc. Based on k-core,

Table 1
Summary of Notations.

Notation	Definition
$G = (V, E)$	undirected simple graph
$n = \|V\|$; $m = \|E\|$	number of vertices; number of edges
$N_G(v)$	the set of neighbors of v
$deg_G(v)$	the degree of v
$core(v, G)$	the coreness of v in graph G
k_{max}	the maximum coreness value of all vertices
$w_{uv}(G)$ or $w_e(G)$	$min\{core(v), core(u)\}$
$G_w = (V, E, W)$	the weighted graph of G according to w_{uv}
$G_c = (V, E \pm E', W_c)$	the weighted graph with inserted/removed edges
$T(G)$	MCST of graph G
$mcst(G)$	edge set in MCST of graph G
$cw(G)$	the total weight of the MCST
E^*	$\{e \in E(G) \| w_e(G) \neq w_e(G^*)\}$
$O_w(u, v)/O_c(u, v)$	Kruskal order of (u, v) before/after the update

problems such as k-core Maximization [45] and Coreness Maximization [36] as well as related quantitative node importance [46] are also extended, which further illustrates the importance of k-core.

Dynamic graph structure analysis. With the characteristics of real-time processing of different graph structures, the research on dynamic graph maintenance is becoming more and more extensive, including but not limited to k-truss [47,10], k-clique [48–50], densest subgraph [11,51], etc. And the maintenance of the k-core [12–14] has also been conducted by many researches. Among them, [13] is the state-of-the-art to maintain the coreness, which can quickly find the set of vertices whose coreness changed by maintaining k-order.

Maximum Spanning tree problem. The update method of the maximum spanning tree (MST) is the same as that of the minimum spanning tree. The method of the last century [16–18] is aimed at MST maintenance under single-edge deletion and insertion. [15] presented an experimental study on several algorithms for the MST problem, which also proposed an efficient implementation of the algorithm of [52].

3. Preliminaries

We define an undirected simple graph as $G = (V, E)$, where $n = |V|$ represents the number of vertices, and $m = |E|$ represents the number of edges. The neighbors of v in G is defined as $N_G(v)$, and the degree of vertex v is denoted as $deg_G(v) = |N_G(v)|$. When the context is clear, we simply use $N(v)$ and $deg(v)$ for clarity. All notations are summarized in Table 1. The specific definitions of k-core and coreness are as follows.

Definition 1 *(k-Core)*. Given an integer $k (k \geq 0)$, the k-core of G, denoted by C^k, is the maximal subgraph of G, such that $\forall v \in C^k, deg_{C^k}(v) \geq k$.

Definition 2 *(Coreness)*. Given a graph G, the coreness value of a vertex $v \in V(G)$, denoted by $core(v)$, is the largest k such that v is in the k-core, i.e., $core(v) = max_{v \in C^k}\{k\}$, where C^k is set as k-core.

3.1. Maximum core spanning tree

Let $G_w = (V, E, W)$ be a weighted graph, which has the same vertices and edges as G, and the edge weight is defined based on the coreness values of the vertices. For $(u, v) \in E$, $w_{uv} \in W$ is defined as $w_{uv} = min\{core(u), core(v)\}$. Then we can get the MCST on G_w.

Definition 3 *(Maximum Core Spanning Tree)*. Given the weighted graph G_w generated from G, the maximum core spanning tree (MCST) T of G, denoted by $T(G)$ is a maximum spanning tree in G_w, and the maximum means the spanning tree of the largest sum weight.

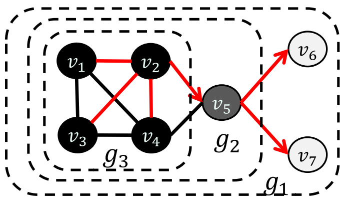
In order to quickly get the MCST, we can use the degeneracy order [53] of the vertices to assign a partial order relationship to the vertices, so that we can generate the initial MCST directly on G using the disjoint set based Kruskal algorithm.

Definition 4 *(Degeneracy Ordering [53])*. Given a graph G, a permutation $(v_1, v_2, ...v_n)$ of all vertices of G is a degeneracy ordering of G if every vertex v_i has the minimum degree in the subgraph of G induced by $\{v_i, ...v_n\}$.

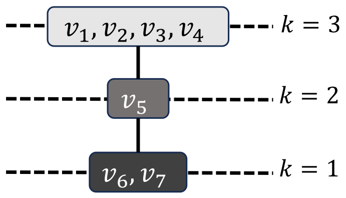
We can use the method in [8] to generate the MCST efficiently, which implements the Kruskal algorithm by disjoint-set data structure as shown in Algorithm 1, where *seq* represents the degeneracy order corresponding to the vertex. The time complexity can

Algorithm 1: Maximum Core Spanning Tree.

```
Input  : A graph $G = (V, E)$, a degeneracy order seq of vertices, and core numbers core(.) of vertices
Output: A maximum core spanning tree $T(G)$ of $G$
1 $T(G) \leftarrow (V, \emptyset)$;
2 Initialize a disjoint-set data structure $\mathscr{F}$ for $V$;
3 foreach vertex $u$ in seq in reverse order do
4     foreach neighbor $v$ of $u$ in $G$ that appear after $u$ in seq do
5         if $v$ and $u$ are not in the same set in $\mathscr{F}$ then
6             Union $u$ and $v$ in $\mathscr{F}$;
7             Add $(u, v)$ with weight $core(u)$ into $T(G)$;
8 return T(G);
```



(a) Maximum Core Spanning Tree



(b) Stair-like Structure

Fig. 2. Example to show the stair-like structure. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

be accelerated to $O(m)$ by using the sorted nature of the degeneracy order [53]. After generating the MCST $T(G)$, the set of edges selected in MCST is denoted as $mcst(G_w)$. To handle graph dynamics, we consider the maintenance of MCST.

Definition 5 *(Maximum Core Spanning Tree Maintenance).* Given a graph G, let E' be the edge set we want to insert to (or remove from) G. Let G_w denote the original weighted graph obtained by G. Let G_c denote the changed weighted graph, i.e., $G_c = (V, E + E', W_c)$ (resp. $G_c = (V, E - E', W_c)$), where W_c is the updated edge weights. Maximum Core Spanning Tree maintenance is to update $T(G_w)$ to $T(G_c)$, and updates the MCST edge set $mcst(G_w)$ to $mcst(G_c)$ and the total weight of MCST from $cw(G_w)$ to $cw(G_c)$.

Since the insertion and deletion of vertices can be regarded as the insertion and deletion of multiple edges, and the response to multiple edge insertion or deletion can be accomplished by calling single edge maintenance multiple times, so we only consider the insertion and deletion of single edge here. Besides, $T(G)$ and $cw(G)$ can be easily obtained from $mcst(G)$, so we mainly maintain $mcst(G)$.

4. Properties

We firstly investigate the properties of MCST and the properties for MCST maintenance. Let G_w be the weighted graph of G. Let G_c represent the weighted graph updated from G_w after edge deletion/insertion from/into G, where $G_c = (V, E \pm (x_1, x_2), W_c)$.

4.1. Properties of MCST

As MCST is generated by defining edge weights through coreness, it has the following properties.

1. MCST has a stair-like structure. If we consider the vertices with the same coreness are on the same stair, then the MCST forms a natural stair structure during its construction. By the greedy construction algorithm of MST, the higher stair (weight) tree edges will be added into MCST before the lower stair (weight) tree edges. We give an example in Fig. 2 to show stair-like structure.

In Fig. 2a, g_3, g_2, and g_1 are 3-core, 2-core, and 1-core, respectively, where the red edges represent the edges in MCST. For a tree edge (u, v), if $core(u) > core(v)$, we convert (u, v) to be a directed edge from u to v, then the vertices connected by the undirected tree edges have the same coreness value. Each directed tree edge is going down from a higher stair to a lower stair. We can further construct the stair structure as shown in Fig. 2b. It can be considered as a different representation of the peel hierarchies [8] of k-core.

2. A MCST is uniquely determined by a given Kruskal-order. Given a Kruskal-order defined below, the result MSCT is unique. The same graph may produce different MCST and Kruskal-order due to the traversal order of vertices, but the Kruskal-order determines the unique traversal order of edges, so it can uniquely determine the MCST.

Definition 6 *(Kruskal Order).* Given a graph G, the order in which edges are traversed for the first time in Algorithm 1 is the Kruskal-order. That is to say, $e_i \succeq e_j$ if (i) $w_{e_i} > w_{e_j}$ or (ii) $w_{e_i} = w_{e_j}$ when e_i is earlier visited than e_j in Algorithm 1.

3. Edge weight changes at most one after single edge insertion/deletion. The edge whose edge weight changes can be determined by the vertexes whose coreness changes. Single-edge insertion or deletion will only change the edge weight by at most one, the coreness of each vertex changes at most one. Based on this property we can use the state-of-the-art method [13] to update the vertex coreness efficiently. We define V^* as the set of vertices whose coreness values have changed after deleting/inserting the edge (x_1, x_2), where V^* may be empty. By the property, we know that for $v \in V^*$, there is $core(v, G_w) = K$ and $core(v, G_c) = K \pm 1$ (insertion or deletion).

Then, we use V^* to find the edge set E^* whose edge weight changes by visiting the neighbor $N_{G_c}(v)$ of each vertex v in V^*. If $min(core(u), core(v))$ changes, then edge (u, v) will be added to E^*.

4.2. Properties of MCST in responding to edge deletion

Inheriting the properties of MST, not all changes will affect MCST when a single-edge (x_1, x_2) in G_w is deleted. In this case, $G_c = (V, E - (x_1, x_2), W_c)$.

Theorem 1 *(Deletion of non-tree edges will not impact MCST). If $(x_1, x_2) \notin mcst(G_w)$, and all $(u, v) \in E^*$ are not in $mcst(G_w)$, then $mcst(G_c) = mcst(G_w)$.*

Proof. Assuming that the weight changed edge is (u, v), we set the Kruskal-order of this edge in G_w to O_1, and the Kruskal-order in G_c after deleting (x_1, x_2) to O_2. Regardless of other edges influence, since $w_{uv}(G_c) = w_{uv}(G_w) - 1$, then $O_1 \geq O_2$. According to Kruskal's algorithm definition, it will firstly visit the edge with a larger Kruskal order. Then, since (u, v) has not been added to the edge set of MCST at position O_1, it cannot be added at position O_2. The deleted edge (x_1, x_2) is similar to the above. Therefore, neither the deleted edge (x_1, x_2) nor the weight reduced edges will change the MCST, and the theorem is proved. □

Theorem 1 guarantees that when deleting a single edge, only the deletion or the weight reduction of the tree edge will affect MCST. So, we do not need to consider the changes in non-tree edges. Theorem 1 can be easily proved by the properties of MCST.

Lemma 1. *In the maintenance problem of MST, if a tree edge (x, y) whose w_{xy} has not changed after deleting an edge, (x, y) will still be in the MST.*

Proof. Let $Tree(x)$ be the connected part of the MST where x is located. After deleting a certain edge $e_i = (u, v)$, the original MST becomes two connected parts $Tree(u)$ and $Tree(v)$.

We prove this by contradiction. Assuming that a tree edge $e_j = (x, y)$ in $Tree(v)$ is removed, and there is an edge e_k that can connect the two parts of them to replace e_j and make the weight of the spanning tree larger. First, the spanning tree will become three parts, $Tree(x)$, $Tree(y)$, and $Tree(u)$. Then we set $O_i > O_k$ and $O_k > O_j$, where O_i is the Krsukal order of vertice i, because according to the traversal order of Kruskal's algorithm, other cases can easily prove that the assumption is wrong.

Then we discuss whether e_k can replace e_j. (i) It connects the two parts of $Tree(x)$ and $Tree(y)$. In this case, e_k can replace e_j in the original graph, so e_j will not become a tree edge in the original graph, and the assumption is wrong. (ii) It connects the two parts of $Tree(x)$ (or $Tree(y)$) and $Tree(u)$. In this case, e_j can still be chosen as a tree edge to connect $Tree(x)$ and $Tree(y)$, and e_k is just a replacement edge to e_i. Therefore, there is no e_k as an alternative edge to e_j to make the weight of the MST larger, which contradicts the assumption, so it is proved. □

Theorem 2 *(Weight unchanged edges remain in MCST). If edge $w_{uv}(G_c) = w_{uv}(G_w)$, $(u, v) \in mcst(G_w)$, then after an edge deletion in G, there is $(u, v) \in mcst(G_c)$.*

Proof. According to Lemma 1, deleting edges will not affect the part of the MCST whose edge weight remains unchanged. The impact of edge weight reduction is similar to that of deletion. The theorem is proved by the above analysis. □

Even if MCST changes due to the deletion or weight reduction of tree edges, most of edges in MCST remain unchanged. Theorem 2 ensures that weight unchanged tree edges will remain in the tree.
Kruskal-order can further narrow down the search scope for changed tree edges under single edge deletion. We will introduce this property in detail in Section 5.2.

4.3. Insertion properties of MCST

Similar to deletion, not all weight changes will affect MCST when a single edge (x_1, x_2) is inserted. In this case, $G_c = (V, E + (x_1, x_2), W_c)$.

Theorem 3 *(Pure weight increase of tree edges will not change the edge set of MCST). If $E^* \subseteq mcst(G_w)$, then $mcst(G_c) = mcst(G_w)$, $T(G_c) = T(G_w)$ and $cw(G_c) = cw(G_w) + |E^*|$, where $|E^*|$ denotes the number of edges in E^*.*

Proof. We denote the updated edge as (u, v). Its Kruskal order before insertion is O_1, and after insertion is O_2, where $O_2 > O_1$. Ignoring the changes of other edges, according to the enumeration order of the Kruskal algorithm, the order of edge enumeration before O_1 and after O_2 remains unchanged. Since $O_2 > O_1$, edge (u, v) must be enumerated first in the new graph. Then it will still become a tree edge and connect the original parts of the MCST where u and v are located, which is the same as its contribution in O_1 order. Thus, the edge set in the MCST remains unchanged. However, due to the change in edge weight from K to $K+1$, $cw(G_c)$ increases by one accordingly. As a result, a total of $|E^*|$ edges will make $cw(G_c) = cw(G_w) + |E^*|$, so the above problem is proved. □

Theorem 3 makes sure that when inserting a single edge, if only some tree edges have increased their weights, they will not change the edge set in the MCST and only increase the total weight of MCST. Therefore, we mainly need to deal with the inserted edge and the weight-increased non-tree edges.
MCST can be maintained by removing the loops under single edge insertion. We will introduce this property in detail in Section 6.2.

5. Maintenance for edge deletion

5.1. Deletion overview

Our main idea is to achieve fast maintenance of MCST under single-edge deletion by maintaining Kruskal-order to reduce the search scope for the replacement edges. We first look for edge sets whose edge weight change according to the coreness change mentioned in Section 4.1. Secondly, an efficient deletion algorithm is proposed according to whether the changed edge is a tree edge or a non-tree edge. In the maintenance process, for the deleted tree edge, it is necessary to find a replacement edge in the remained edges. We use Kruskal-order to reduce the search scope for seeking the replacement edge.

5.2. The impact of the deleted edge and edge weight changes

For the deleted edge and weight-changed edges, their impacts to MCST have following two cases.
Case-1: According to Theorem 1, if the deleted edge and the weight reduced edges are all non-tree edges, the MCST will not change. If **Case-1** happens, MCST does not need to be updated.
Case-2: The deleted edge and the weight reduced edges contain tree edges and non-tree edges. There are two sub-cases here, namely, **Case-2a**: the deleted edge does not belong to the tree edge, and **Case-2b**: the deleted edge belongs to the tree edge.

We have the same strategy for **Case-2a** and **Case-2b**. If any of the deleted edge or weight-reduced edge is a tree edge, a replacement edge needs to be found in $E \setminus mcst(G_w)$. And we can get that the tree edges without edge-weight change will still be in MCST according to the Theorem 2. It follows that the efficient way to maintain the MCST is to remove the deleted edge and the weight-changed tree edges from the MCST and then find the edges with the largest weight from the original graph to connect the MCST without destroying the tree structure. Therefore, our optimization aims at using the smallest traversal to obtain the replacement edges for the deleted edge and weight-reduced tree edges from the original graph. We limit the enumeration range according to Kruskal-order with the following Theorem.

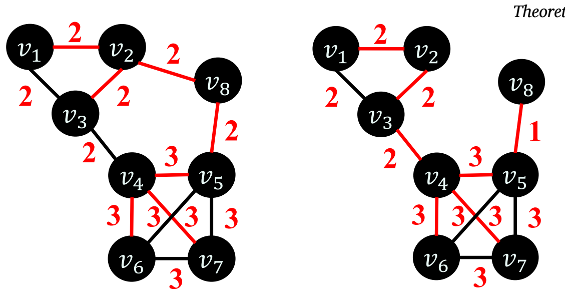
Theorem 4 *(Narrow down the search scope for the replacement edges by Kruskal-order). Consider (u,v) is a weight-reduced tree edge, if (u,v)'s Kruskal-order in G_w is $O_w(u,v)$, and the Kruskal-order in G_c is $O_c(u,v)$, then the edges whose Kruskal-order are before $O_w(u,v)$ or after $O_c(u,v)$ in G_c will not become tree edges.*

Proof. For edge e_1, assuming its Kruskal order is before O_w. According to the MCST definition, the traversal order of e_1 in both G_w and G_c is before (u,v). Therefore, it remains unaffected by the Kruskal order change of (u,v). Similarly, for edge e_2, assuming its Kruskal order is after O_c, and its traversal order in both G_w and G_c is after (u,v), it will not be affected by the Kruskal order changes of (u,v). Based on these arguments, the theorem is proven. □

For **Case-2a**, non-tree edge deletion may decrease the weight of tree edges. Consider (u,v) is a weight-reduced tree edge, we search following the Kruskal order to find whether there is an higher weight non-tree edge that can replace (u,v). If no higher weight non-tree edge can replace (u,v), (u,v) will remain to be a tree edge. So our search range is between (u,v)'s Kruskal-order $O_w(u,v)$ before the update and the updated Kruskal-order $O_c(u,v)$ according to Theorem 4. The weight reduction of (u,v) will only move its Kruskal-order at most from O_K to O_{K-1}, where O_K represents the starting position of the Kruskal order with the edge weight K, so the search range is at most $|O_K|$, where $|O_K|$ represents the number of edges whose weight is K.

For **Case-2b**, we need to find a replacement edge for the deleted edge (x_1, x_2). Similar to **Case-2a**, we search following the Kruskal order. However, in the worst case, we may search to the lowest stair (resp. the end of the Kruskal-order) to find a replacement edge, so the search range is at most $\sum_{i=1}^{K} |O_i|$. Since there are multiple edges with changed edge weight (including the deleted edge) need to find replacement edges under single-edge deletion, we declare O_{start} as the smallest Kruskal-order in G_w and O_{end} as the largest Kruskal-order in G_c among all the edge-weight reduced edges (including the deleted edge) and then find replacement edges by traversing from O_{start} to O_{end}. We give an example to show the specific operation.

Example 2. An example is given in Fig. 3. In Fig. 3a, the red edges represent edges in MCST, and the numbers on the edges represent weights. Due to the deletion of (v_2, v_8) and edge-weight reduction of tree edge (v_5, v_8), we need to find replacement edges. We declare



(**a**) Graph (**b**) Delete (v_2, v_8)

	(v_4,v_5)	(v_4,v_6)	(v_4,v_7)	(v_5,v_6)	(v_5,v_7)	(v_6,v_7)	(v_2,v_8)	(v_2,v_1)	(v_2,v_3)	(v_5,v_8)	(v_1,v_3)	(v_3,v_4)
Kruskal-order	0	1	2	3	4	5	6	7	8	9	10	11
Edge weight	3	3	3	3	3	3	2	2	2	2	2	2

(**c**) Kruskal-order of edges before the update

Fig. 3. Examples for Dynamic MCST maintenance under Edge Deletion.

Algorithm 2: OrderPassed Algorithm.

```
Input  : A graph $G_w$, edge set of MCST $mcst(G_w)$, deleted edge $(x_1, x_2)$
Output: Updated MCST edge set $mcst(G_c)$
1  $V^* \leftarrow$ vertices with coreness changed by deleting $(x_1, x_2)$ to $G$, $E^* \leftarrow \emptyset$;
2  Delete $(x_1, x_2)$ from Kruskal order;
3  $old\_tree\_num \leftarrow |mcst(G_w)|$, $cnt\_mcst \leftarrow 0$;
4  $O_{start} \leftarrow |V|$, $O_{end} \leftarrow 0$;
   /* calculate $E^*$                                                    */
5  foreach vertex $v \in V^*$ do
6      foreach vertex $u \in N_{G_w}(v)$ do
7          if $(u, v) \in E^*$ then continue ;
8          if $min(core(u), core(v))$ changes then
9              $E^* \leftarrow E^* \cup (u, v)$, update the Kruskal order of $(u, v)$;
10             if $(u, v) \in mcst(G_w)$ then
11                 Delete $(u, v)$ from $mcst(G_w)$;
12                 $O_{start} \leftarrow min\{O_{start}, O_w(u, v)\}$, $O_{end} \leftarrow max\{O_{end}, O_c(u, v)\}$;
13 if $(x_1, x_2) \in mcst(G_w)$ then
14     $O_{end} \leftarrow$ the end of the Kruskal-order; // Case-2b
15 if no update of $O_{start}$ and $O_{end}$ then
16     return $mcst(G_w)$;// Case-1
   /* Case-2                                                             */
17 Initialize a disjoint-set data structure $\mathscr{F}$ for $V$;
18 $mcst(G_c) \leftarrow mcst(G_w)$, correspondingly update $\mathscr{F}$, cnt_mcst = $|mcst(G_c)|$;
19 for $O_i$ from $O_{start}$ to $O_{end}$ do
20     $(u, v) \leftarrow$ the edge in Kruskal-order position $O_i$;
21     if $(u, v)$ can be a replacement edge then
22         Union $u$ and $v$ in $\mathscr{F}$, $cnt\_mcst \leftarrow cnt\_mcst + 1$;
23         Add edge $(u, v)$ with $w_{uv}$ into $mcst(G_c)$;
24     if $cnt\_mcst = old\_tree\_num$ then break;
25 return $mcst(G_c)$;
```

O_{start} and O_{end} to record the start and end Kruskal-order of the traversal. As shown in Fig. 3c, $O_{start} = min(O_w(v_2, v_8), O_w(v_5, v_8)) = 6$ and $O_{end} = max(O_c(v_2, v_8), O_c(v_5, v_8)) = 11$. Therefore, we traverse from O_{start} to O_{end} to find replacement edges for (v_2, v_8) and (v_5, v_8) and find (v_3, v_4) and (v_5, v_8) itself as replacement edges. The updated MCST is shown in Fig. 3b. We reduce the traversal of the edges before $O_w(v_2, v_8)$ in this process.

5.3. OrderPassed algorithm

Based on the above analysis, we propose an OrderPassed algorithm to implement the dynamic update of the MCST after deleting (x_1, x_2), as shown in Algorithm 2. Lines 1-4 are the initialization part, where O_{start} and O_{end} are used to record the starting and ending positions of Kruskal-order traversal. Lines 5-12 look for E^* and update the corresponding Kruskal order. Lines 13-16 deal with **Case-2b** and **Case-1**. Lines 17-24 deal with **Case-2**, where line 21 determines whether (u, v) can be a replacement edge by judging whether they belong to different sets in $\mathscr{F}$. Since MCST has the property of MST, the number of edges of MCST will not increase after deleting edge, so we use cnt_mcst in Algorithm 2 to record the number of edges and terminate the algorithm early.

Theorem 5. *The time complexity of Algorithm 2 is $\mathcal{O}(|E^*| + \sum_{i=1}^{K} |O_i|)$, where $\mathcal{O}(|E^*|) = \mathcal{O}(deg(v) \times |V^*|)$.*

Proof. In terms of time complexity, there are three parts. The first part comes from the update of coreness [13], which is $\mathcal{O}(\sum_{w \in V^*} deg(w) + log|Q_K|. \sum_{w \in V^*} deg_K(w) + |V^*|.log|Q_{K-1}|)$, where Q_K represents the number of vertices whose coreness is K, and $deg_K(w)$ is the number of neighbors w' of w that $core(w') = K$. The time complexity of the first part can be omitted, since it is smaller than the latter two parts. The second part comes from the calculation of E^*, whose complexity is $\mathcal{O}(|E^*|) = \mathcal{O}(deg(v) \times |V^*|)$. The third part comes from the traversal of the Kruskal order. For **Case-2a** and **Case-2b**, the worst time complexity is $\mathcal{O}(|O_K|)$ and $\mathcal{O}(\sum_{i=1}^{K} |O_i|)$ respectively, where $|O_i|$ is the number of edges whose edge weight is i. The final algorithm time complexity is $\mathcal{O}(|E^*| + \sum_{i=1}^{K} |O_i|)$ in the worst case, where we ignore the first part as it is smaller. $\square$

6. Maintenance of edge insertion

6.1. Insertion overview

Our main idea is to rapidly maintain MCST under single-edge insertion to delete the smallest edge forming loop with the tree edges to maintain MCST. We first look for edge sets whose edge weight changes according to the coreness change mentioned in Section 3.1. Secondly, an efficient insertion maintenance algorithm is proposed for whether the changed edge is a tree or non-tree edge. We add the inserted edge and non-tree edges into the new MCST, and then we update the MCST by removing the corresponding edges forming loop with tree edges. It is also necessary to maintain the Kruskal order after the insertion operation.

6.2. The impact of the inserted edge and edge weight changes

We analyze the following two cases for the weight-increased edges.
Case 1: As stated in Theorem 3, if all weight-increased edges are tree edges, the set of edges in the MCST remains unchanged. Therefore, in **Case 1**, we only need to update the total weight of the MCST. However, it is still necessary to process the newly inserted edge (x_1, x_2), which is a non-tree edge.
Case-2: The weight-increased edges include both tree edges and non-tree edges. From Theorem 3, it follows that the weight-increased tree edges only contribute to an increase in the overall weight of the MCST. Therefore, the focus in **Case 2** shifts to dealing with the weight-increased non-tree edges.

In **Case-2**, the edges in $E_{nonTree}$ will be added to the new MCST candidate edge set. The candidate edge set in the MCST consists of three parts. The first part is the inserted edge (x_1, x_2) (**Case-1** is the same) because its insertion may become a candidate edge of the $mcst(G_c)$. The second part is the non-tree edges with increased weight. Due to the increase in edge weight of non-tree edge (u, v), $O_c(u, v)$ correspondingly moves forward to make it become a candidate edge of the $mcst(G_c)$. The third part is the edges of the $mcst(G_w)$. Because they were selected as tree edges of MCST before the update, we must analyze whether they will be removed after inserting (x_1, x_2). We record all the above three parts' edges as E_{cand}. E_{cand} may form extra loops, and our next step is to remove the loops.

Theorem 6 *(MCST can be maintained by removing loops). If $E' \subseteq E_{cand}$ constitute a loop, then $(u, v) \notin mcst(G_c)$ if w_{uv} is the smallest among all the edges in E'.*

Proof. According to the definition of the MCST, it cannot contain loops. Therefore, edges must be removed until they become free of loops. According to the definition of the MST, the edge with the smallest weight in the loop should be deleted. By considering all loops and examining all candidate sets, the selected edge set is deemed optimal. $\square$

In accordance with Theorem 6, the MCST is maintained by eliminating loops in E_{cand}, specifically by removing the edges with the smallest weights. In practice, this can be efficiently implemented using the union-find algorithm. By traversing the candidate edges in descending order of weight, we can determine whether the two vertices of each edge belong to the same union-find set. If the vertices are in different sets, the edge is added to the resulting MCST; otherwise, the edge is skipped.

According to Theorem 6, the MCST is maintained by removing loops in E_{cand}, specifically by removing the edges with the smallest weights. In practice, this can be efficiently implemented using the union-find. By traversing the candidate edges in descending order of weight, we can determine whether the two vertices of each edge belong to the same union-find set. If the vertices are in different sets, the edge is added to the resulting MCST; otherwise, the edge is skipped.

6.3. LoopFree algorithm

Based on the above analysis, we propose a LoopFree algorithm to implement the dynamic update of the MCST after inserting (x_1, x_2), as shown in Algorithm 3. In Algorithm 3, lines 1-3 are the initialization part, where *cnt_mcst* is used to count the number

Algorithm 3: LoopFree Algorithm.

```
Input  : A graph $G_w$, edge set of MCST $mcst(G_w)$, inserted edge $(x_1, x_2)$
Output: Updated MCST edge set $mcst(G_c)$
1  $E^* \leftarrow \emptyset$, $cnt\_mcst \leftarrow 0$, $old\_tree\_num \leftarrow |mcst(G_w)|$;
2  $V^* \leftarrow$ vertices with coreness changed by inserting $(x_1, x_2)$ to $G$;
3  Insert $(x_1, x_2)$ into Kruskal order;
   /* calculate $E^*$                                                    */
4  foreach vertex $v \in V^*$ do
5      foreach vertex $u \in N_{G_w}(v)$ do
6          if $(u, v) \in E^*$ then continue;
7          if $min(core(u), core(v))$ changes then
8              $E^* \leftarrow E^* \cup (u, v)$ ;
9              update the Kruskal order of $(u, v)$;
               /* Case-1                                                   */
10             if $(u, v) \in mcst(G_w)$ then $E^* \leftarrow E^* - (u, v)$;
   /* Case-2                                                               */
11 $E_{cand} \leftarrow$ an empty queue;
12 Mergesort all edges in $E^*$ and $mcst(G_w)$ according to the edge weight and output the result to $E_{cand}$;
13 Initialize a disjoint-set data structure $\mathscr{F}$ for $V$;
14 while $E_{cand} \neq \emptyset \wedge cnt\_mcst \leq old\_tree\_num + 1$ do
15     $(u, v) \leftarrow E_{cand}$.dequeue();
16     if $(u, v)$ will not form a loop with $mcst(G_c)$ then
17         Union $u$ and $v$ in $\mathscr{F}$;
18         Add edge $(u, v)$ with $w_{uv}$ into $mcst(G_c)$;
19         $cnt\_mcst \leftarrow cnt\_mcst + 1$;
20 return $mcst(G_c)$;
```

of edges in the MCST. Lines 4-10 find the corresponding E^* set according to V^*. Among them, Line 10 processes the tree edges in **Case-1**. Lines 11-19 deal with **Case-2**, where line 16 determines whether u, v belongs to different sets in $\mathscr{F}$ to determine whether it forms a loop with $mcst(G_c)$. Since the edges are traversed in decreasing order, if (u, v) forms a loop with $mcst(G_c)$, (u, v) must be the edge with the smallest weight. Similar to deletion, early stop can still be performed by counting the number of edges during MCST insertion maintenance.

Example 3. In Fig. 1, after inserting edge (v_{11}, v_{15}), the updated edge set E^* becomes $\{(v_1, v_4), (v_4, v_7), (v_4, v_{11}), (v_{11}, v_{12}), (v_{11}, v_{15})\}$. Consequently, all edges in E^*, along with the edges of the original MCST, i.e., $mcst(G)$, should be considered as part of the candidate edge set E_{cand}.

Then, there are two loops. The first cycle involves the vertices $\{v_{11}, v_{12}, v_{15}\}$, and the second comprises $\{v_1, v_2, v_3, v_4, v_5, v_7\}$. To maintain the MCST, it is necessary to eliminate the edges with the smallest weights from these loops. Accordingly, the edges (v_3, v_5) and (v_{11}, v_{15}) are removed. The remaining edges constitute the edge set of the final MCST, i.e., $mcst(G_c)$.

Theorem 7. *The time complexity of Algorithm 3 is $\mathcal{O}(|E^*| + |V|)$, where $\mathcal{O}(|E^*|) = \mathcal{O}(deg(v) \times |V^*|)$.*

Proof. The time complexity has also three parts. The first part comes from the update of coreness [13], which needs $\mathcal{O}(deg(v) \times max\{|Q_K|, |Q_{K+1}|\})$, where $|Q_K|$ represents the number of vertices whose coreness is K. The first part can be omitted due to it is small. The second part is to calcualte E^*, where $\mathcal{O}(|E^*|) = \mathcal{O}(|V^*| \times deg(v))$, and the time complexity of mergesort is $\mathcal{O}(|E^*| + |V| - 1)$. The time complexity of the third part for selecting the edge set in $MCST$ is also $\mathcal{O}(|E^*| + |V| - 1)$. The process of updating Kruskal order is $\mathcal{O}(1)$, and the insertion cost is $\mathcal{O}(k_{max})$, where k_{max} is the maximum value of the coreness value of the G_w. The insertion is only called once at most. Adding up the three parts, the final time complexity is $\mathcal{O}(|E^*| + |V|)$. □

7. Experiments

In this section, experiments are carried out on 13 large real-world graphs. We conducted comparative experiments in Section 7.1 to prove the efficiency of our algorithm. In addition, we verified the stability and scalability of our algorithm through experiments.

Settings. All algorithms are implemented in C++ and compiled by g++ compiler at -O3 optimization level, and they are conducted on a machine with Inter(R) Xeon(R) CPU E5-2667 v4@3.20 GHz processor and 128 GB memory, with Ubuntu installed. We call the Snap [54] library to implement the graph structure.

Datasets. The 13 publicly available datasets are listed in Table 2. These datasets contain different types of actual application scene graphs, among which Facebook [55] is an online social network, which can be downloaded on Konect. The following datasets scc_reality, twitter_copen, infect_dublin, human_gene2, BlogCatalog are all from [56]. The first three are temporal graphs, human_gene2 is a biological network, and BlogCatalog is a social network. The rest of the datasets can be downloaded from the SNAP [57] official website, including social networks (DBLP, pokec, LiveJournal), network graphs (BerkStan and Google), location-based social network (Gowalla) and network with ground-truth communities (orkut).

Table 2
Datasets.

Dataset	$n = \|V\|$	$m = \|E\|$	avg.deg	max k
scc_reality	6,809	4,714,485	1,385	1,235
twitter_copen	8,580	473,614	110	582
infect_dublin	10,972	175,573	32	83
human_gene2	14,340	9,027,024	1,259	1,902
Facebook	63,731	817,035	25.64	52
BlogCatalog	88,784	2,093,195	47	222
Gowalla	196,591	950,327	9.67	51
DBLP	317,080	1,049,866	6.62	112
BerkStan	685,230	6,649,470	19.41	201
Google	875,713	4,322,051	9.87	44
pokec	1,632,803	30,622,564	37.51	47
orkut	3,072,441	117,185,083	76.28	253
LiveJournal	4,847,571	68,993,773	28.46	372

Table 3
Performance on All the Datasets.

Datasets	Delete (s)			Insert (s)		
	OP	CSR	SR	LF	CSR	SR
scc_reality	**0.510**	13.985	24.962	**0.441**	14.272	24.911
twitter_copen	**0.134**	1.977	3.310	**0.132**	1.976	3.308
infect_dublin	**0.264**	1.278	2.304	**0.258**	1.278	2.247
human_gene2	**0.872**	42.365	84.134	**0.865**	42.212	83.532
Facebook	**1.365**	7.446	15.026	**1.355**	7.224	15.158
BlogCatalog	**1.889**	12.938	27.610	**1.905**	12.690	27.010
Gowalla	**3.995**	18.151	36.317	**4.015**	18.319	36.53
DBLP	**7.063**	29.625	55.051	**6.743**	30.116	54.971
BerkStan	**15.671**	77.821	161.076	**14.818**	77.153	161.669
Google	**21.664**	104.144	239.988	**20.531**	105.378	242.581
pokec	**44.527**	390.246	1127.912	**45.479**	400.832	1126.494
orkut	**108.889**	1546.453	6749.278	**101.631**	1569.060	6501.904
LiveJournal	**180.736**	1051.513	3108.603	**178.170**	1073.798	3207.107

7.1. Comparative experiment

We compare the proposed OrderPassed (OP) Algorithm (Algorithm 2) and LoopFree (LF) Algorithm (Algorithm 3) with the static recalculation (SR) (Algorithm 1) and coreness update in [13] with static recalculation MCST (CSR). The SR algorithm recalculates the degeneracy order after each insertion/deletion and calls Algorithm 1 to obtain the MCST. Compared with SR, CSR saves the time of calculating the degeneracy order by dynamically maintain coreness.

We repeatedly delete or insert 500 edges to test the algorithm's performance. For the temporal graphs (infect_dublin, twitter_dublin, and scc_reality), we take the 500 edges with the maximum timestamps and randomly select 500 edges from all edges for other datasets. For each graph, we measure the sum time cost for the MCST maintenance under 500 edges insertion/deletion. The results are shown in Table 3. We compare our proposed algorithms (OP and LF) with CSR and SR. The MCST maintenance time of OP and LF significantly outperforms the CSR and SR algorithms in all tested datasets. For human_gene2, the speed up even achieves up to 48 times and 96 times compared to CSR and SR in both the cases of insertion and deletion.

7.2. Stability testing

We conducted stability tests on Gowalla, BlogCatalog, and DBLP datasets. At first, we randomly sample 50,000 edges and randomly divide them into 100 groups for insertion and deletion. Secondly, we insert/delete these edges to/from the graph group by group. For each group, we measure the accumulated maintenance time of MCST used by Algorithm 2 and 3. Fig. 4 shows the experimental results. The performance of the LF and OP algorithms is well bounded. On the one hand, the performance of OP and LF is primarily constrained by the graph size, specifically the number of vertices, as indicated in the time complexity analysis. On the other hand, the edge set E^* is influenced by the specific inserted edge and the underlying graph structure, leading to observable fluctuations across the three datasets. For example, occasional irregularities, such as bumps in the BlogCatalog dataset, can be noted. However, these fluctuations have a negligible impact on overall runtime, demonstrating the stability of our algorithm.

7.3. Scalability testing

We conducted scalability tests on Facebook, Blogcatlog, and Gowalla datasets, where we randomly sampled vertices and edges separately, increasing from 20% to 100%. The experiment results are shown in Fig. 5, where the subtitle of the figure indicates

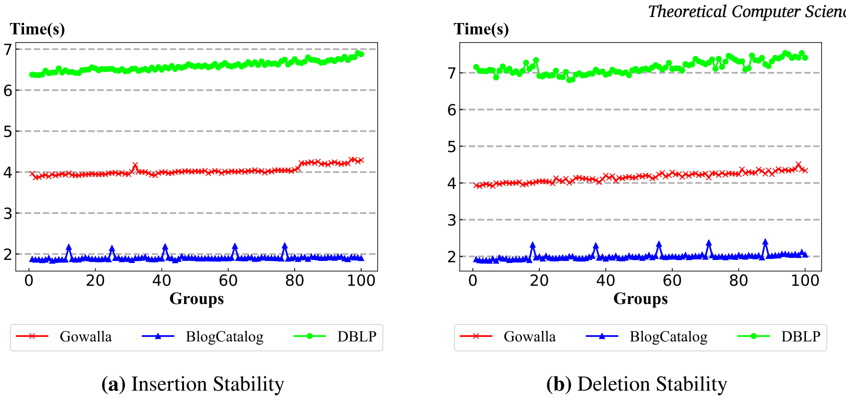


Fig. 4. The Stability of the Dynamic Maintenance Algorithm.

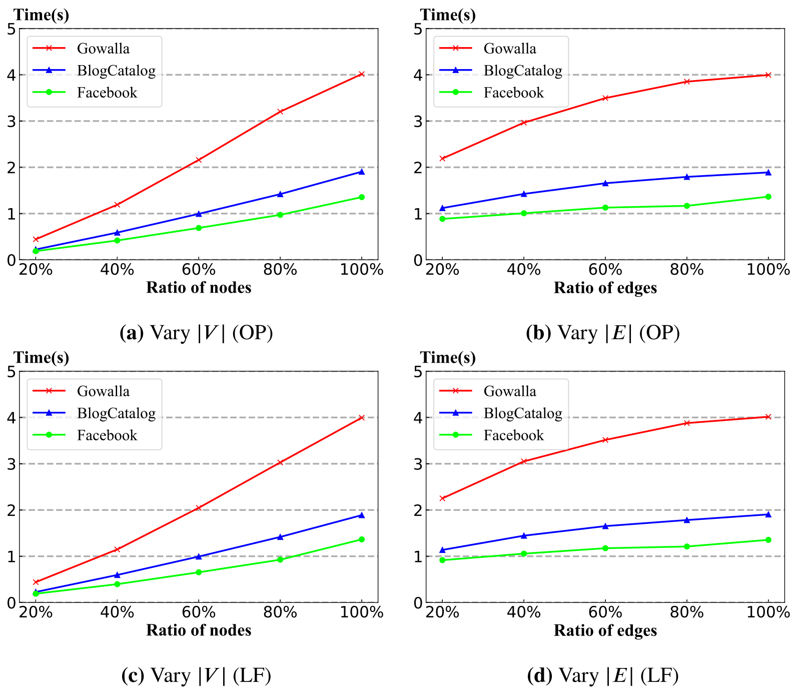


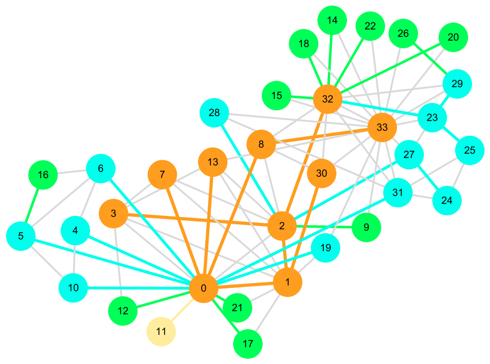
Fig. 5. The Scalability of the Dynamic Maintenance Algorithm.

whether we are sampling vertices or edges, and the brackets indicate the algorithm we use to maintain MCST. From each subgraph sampled, we further sample 500 edges for testing the sum time cost for the MCST maintenance under 500 edges insertion/deletion. It can be concluded from Fig. 5 that as the number of sampled vertices and edges increases rapidly, the running time of the algorithm increases smoothly, which means that our algorithm has good scalability.

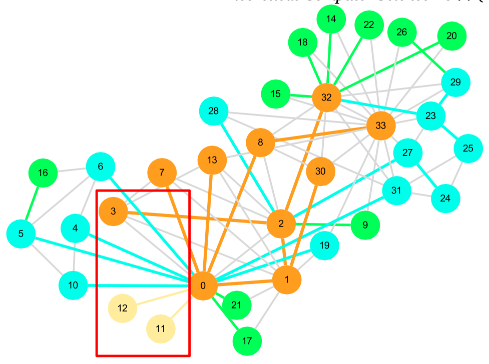
By comparing Fig. 5a and Fig. 5b, as well as Fig. 5c and Fig. 5d, it becomes evident that changes in the vertex ratio have a greater impact on the time complexity of the algorithm. Furthermore, the growth rate observed in the Gowalla dataset is significantly higher compared to the other two datasets. This is due to Gowalla's lower density compared to the other two graphs, which causes a significant increase in the number of vertices and edges as the vertex and edge ratios rise, thereby leading to a rapid escalation in time complexity.

7.4. Visualization

To more intuitively represent the form of the MCST, we selected a small social network dataset soc-karate (https://networkrepository.com) for visualization. The different colors of the vertices represent their different coreness values, and the colors of the edges correspond to their edge weights. The edges that are not selected into the MCST are all gray. As shown in Fig. 6a, the MCST can optimize the graph by eliminating many useless edges. As shown in Fig. 6b, after deleting the edge (v_3, v_{12}), the MCST structure of the graph will also change accordingly, so reasonable maintenance is required.



(a) Graph



(b) Removing edge (v_3, v_{12})

Fig. 6. Visualization on the soc-karate dataset.

8. Conclusions

In this work, we study the problem of MCST maintenance regarding dynamic edge insertion and deletion. We investigate a series of properties, such as the stair-like structure; the unique determination of MCST by a given Kruskal-order; and various principles to narrow down the maintenance cost. Using these properties, we propose the OP and LF algorithms for MCST maintenance under single-edge deletion and insertion. The key is to maintain the Kruskal order of the graph, which can reduce the search scope for replacement edges. We prove the effectiveness, scalability, and stability of our algorithm on 13 real-world datasets. Our OP and LF algorithms outperform the CSR and SR up to 48 times and 96 times in both the cases of insertion and deletion.

CRediT authorship contribution statement

Xiaowei Lv: Writing – original draft, Software, Investigation. **Yongcai Wang:** Writing – review & editing, Writing – original draft. **Deying Li:** Writing – review & editing, Writing – original draft.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Deying Li reports financial support was provided by National Natural Science Foundation of China Grant No. 12071478. Yongcai Wang reports financial support was provided by National Natural Science Foundation of China Grant No. 61972404; Public Computing Cloud, Renmin University of China; Blockchain Lab. School of Information, Renmin University of China. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] X. Lv, Y. Wang, D. Li, H. Ping, Maximum core spanning tree insertion maintenance for large dynamic graphs, in: International Conference on Algorithmic Aspects in Information and Management, Springer, 2024, pp. 3–15.
[2] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, Graph structure in the web, Comput. Netw. 33 (2000) 309–320.
[3] Y. Fang, R. Cheng, X. Li, S. Luo, J. Hu, Effective community search over large spatial graphs, Proc. VLDB Endow. 10 (2017) 709–720.
[4] G. Palla, I. Derényi, I. Farkas, T. Vicsek, Uncovering the overlapping community structure of complex networks in nature and society, Nature 435 (2005) 814–818.
[5] S.B. Seidman, Network structure and minimum degree, Soc. Netw. 5 (1983) 269–287.
[6] A. Gibbons, Algorithmic Graph Theory, Cambridge University Press, 1985.
[7] J. Wang, J. Cheng, Truss decomposition in massive networks, arXiv preprint, arXiv:1205.6693, 2012.
[8] L. Chang, L. Qin, Cohesive Subgraph Computation over Large Sparse Graphs: Algorithms, Data Structures, and Programming Techniques, Springer, 2018.
[9] Z. Lin, F. Zhang, X. Lin, W. Zhang, Z. Tian, Hierarchical core maintenance on large dynamic graphs, Proc. VLDB Endow. 14 (2021) 757–770.
[10] Q. Luo, D. Yu, X. Cheng, H. Sheng, W. Lyu, Exploring truss maintenance in fully dynamic graphs: a mixed structure-based approach, IEEE Trans. Comput. 72 (2022) 707–718.
[11] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, S. Tirthapura, Dense subgraph maintenance under streaming edge weight updates for real-time story identification, VLDB J. 23 (2014) 175–199.
[12] R.-H. Li, J.X. Yu, R. Mao, Efficient core maintenance in large dynamic graphs, IEEE Trans. Knowl. Data Eng. 26 (2013) 2453–2465.
[13] Y. Zhang, J.X. Yu, Y. Zhang, L. Qin, A fast order-based approach for core maintenance, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), IEEE, 2017, pp. 337–348.
[14] D. Wen, L. Qin, Y. Zhang, X. Lin, J.X. Yu, I/o efficient core graph decomposition at web scale, in: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), IEEE, 2016, pp. 133–144.
[15] G. Cattaneo, P. Faruolo, U.F. Petrillo, G.F. Italiano, Maintaining dynamic minimum spanning trees: an experimental study, in: Algorithm Engineering and Experiments: 4th International Workshop, ALENEX 2002 San Francisco, CA, USA, January 4–5, 2002, Springer, 2002, pp. 111–125, Revised Papers.
[16] M.R. Henzinger, V. King, Maintaining minimum spanning trees in dynamic graphs, in: Automata, Languages and Programming: 24th International Colloquium, ICALP'97 Bologna, Italy, July 7–11, 1997, Springer, 1997, pp. 594–604, Proceedings 24.

[17] G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, in: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 252–257.
[18] G. Amato, G. Cattaneo, G.F. Italiano, Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms, SODA, vol. 97, Citeseer, 1997, pp. 314–323.
[19] J. Cheng, Y. Ke, A.W.-C. Fu, J.X. Yu, L. Zhu, Finding maximal cliques in massive networks, ACM Trans. Database Syst. 36 (2011) 1–34.
[20] J. Pei, D. Jiang, A. Zhang, On mining cross-graph quasi-cliques, in: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, 2005, pp. 228–238.
[21] A.E. Sariyuce, A. Pinar, Fast hierarchy construction for dense subgraphs, arXiv preprint, arXiv:1610.01961, 2016.
[22] D. Chu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Y. Xia, C. Zhang, Finding the best k in core decomposition: a time and space optimal solution, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 685–696.
[23] W. Khaouid, M. Barsky, V. Srinivasan, A. Thomo, K-core decomposition of large networks on a single pc, Proc. VLDB Endow. 9 (2015) 13–23.
[24] Y. Shao, L. Chen, B. Cui, Efficient cohesive subgraphs detection in parallel, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 613–624.
[25] D.W. Matula, L.L. Beck, Smallest-last ordering and clustering and graph coloring algorithms, J. ACM 30 (1983) 417–427.
[26] N. Chiba, T. Nishizeki, Arboricity and subgraph listing algorithms, SIAM J. Comput. 14 (1985) 210–223.
[27] M. Ortmann, U. Brandes, Triangle listing algorithms: back from the diversion, in: 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2014, pp. 1–8.
[28] J. Cohen, Trusses: cohesive subgraphs for social network analysis, Nat. Secur. Agency Tech. Rep. 16 (2008).
[29] X. Huang, H. Cheng, L. Qin, W. Tian, J.X. Yu, Querying k-truss community in large and dynamic graphs, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 1311–1322.
[30] H. Liu, Y. Wang, X. Xu, D. Li, Bottom-up k-vertex connected component enumeration by multiple expansion, in: 2024 IEEE 40th International Conference on Data Engineering (ICDE), IEEE, 2024, pp. 3000–3012.
[31] Y. Wang, X. Jian, Z. Yang, J. Li, Query optimal k-plex based community in graphs, Data Sci. Eng. 2 (2017) 257–273.
[32] L. Chang, J.X. Yu, L. Qin, X. Lin, C. Liu, W. Liang, Efficiently computing k-edge connected components via graph decomposition, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 205–216.
[33] R. Zhou, C. Liu, J.X. Yu, W. Liang, B. Chen, J. Li, Finding maximal k-edge-connected subgraphs from a large graph, in: Proceedings of the 15th International Conference on Extending Database Technology, 2012, pp. 480–491.
[34] X. Xu, H. Liu, X. Lv, Y. Wang, D. Li, An efficient and exact algorithm for locally h-clique densest subgraph discovery, Proc. ACM Manag. Data 2 (2024) 1–26.
[35] B. Liu, F. Zhang, C. Zhang, W. Zhang, X. Lin, Corecube: core decomposition in multilayer graphs, in: Web Information Systems Engineering–WISE 2019: 20th International Conference, Proceedings 20, Hong Kong, China, January 19–22, 2020, Springer, 2019, pp. 694–710.
[36] Q. Linghu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Global reinforcement of social networks: the anchored coreness problem, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 2211–2226.
[37] A. Montresor, F. De Pellegrini, D. Miorandi, Distributed k-core decomposition, in: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2011, pp. 207–208.
[38] L. Chen, C. Liu, K. Liao, J. Li, R. Zhou, Contextual community search over large social networks, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 88–99.
[39] L. Chen, C. Liu, R. Zhou, J. Li, X. Yang, B. Wang, Maximum co-located community search in large scale social networks, Proc. VLDB Endow. 11 (2018) 1233–1246.
[40] C. Giatsidis, F. Malliaros, D. Thilikos, M. Vazirgiannis, Corecluster: a degeneracy based graph clustering framework, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28, 2014.
[41] C. Li, F. Zhang, Y. Zhang, L. Qin, W. Zhang, X. Lin, Efficient progressive minimum k-core search, Proc. VLDB Endow. (2020).
[42] R.-H. Li, L. Qin, F. Ye, J.X. Yu, X. Xiao, N. Xiao, Z. Zheng, Skyline community search in multi-valued networks, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 457–472.
[43] R.-H. Li, J. Su, L. Qin, J.X. Yu, Q. Dai, Persistent community search in temporal networks, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 797–808.
[44] J.I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, A. Vespignani, K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases, arXiv preprint, arXiv:cs/0511007, 2005.
[45] X. Sun, X. Huang, D. Jin, Fast algorithms for core maximization on large graphs, Proc. VLDB Endow. 15 (2022) 1350–1362.
[46] F. Zhang, Q. Linghu, J. Xie, K. Wang, X. Lin, W. Zhang, Quantifying node importance over network structural stability, in: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023, pp. 3217–3228.
[47] E. Akbas, P. Zhao, Truss-based community search: a truss-equivalence based indexing approach, Proc. VLDB Endow. 10 (2017) 1298–1309.
[48] A. Das, M. Svendsen, S. Tirthapura, Incremental maintenance of maximal cliques in a dynamic graph, VLDB J. 28 (2019) 351–375.
[49] T.J. Ottosen, J. Vomlel, Honour thy neighbour: clique maintenance in dynamic graphs, Probab. Graph. Models (2010) 201.
[50] A.-S. Himmel, H. Molter, R. Niedermeier, M. Sorge, Adapting the bron–kerbosch algorithm for enumerating maximal cliques in temporal graphs, Soc. Netw. Anal. Min. 7 (2017) 1–16.
[51] A. Epasto, S. Lattanzi, M. Sozio, Efficient densest subgraph computation in evolving graphs, in: Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 300–310.
[52] J. Holm, K. De Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, J. ACM 48 (2001) 723–760.
[53] D.R. Lick, A.T. White, k-degenerate graphs, Can. J. Math. 22 (1970) 1082–1096.
[54] J. Leskovec, R. Sosič, Snap: a general-purpose network analysis and graph-mining library, ACM Trans. Intell. Syst. Technol. 8 (2016) 1.
[55] B. Viswanath, A. Mislove, M. Cha, K.P. Gummadi, On the evolution of user interaction in Facebook, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09), 2009.
[56] R.A. Rossi, N.K. Ahmed, The network data repository with interactive graph analytics and visualization, in: AAAI, 2015, https://networkrepository.com.
[57] J. Leskovec, A. Krevl, SNAP datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.