Discovering and Maintaining the Best k in Core Decomposition

Deming Chu, Fan Zhang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Yinglong Xia, and Chenyi Zhang

Abstract—The mode of k-core and its hierarchical decomposition have been applied in many areas, such as sociology, the world wide web, and biology. Algorithms on related studies often need an input value of parameter k, while there is no existing solution other than manual selection. In this paper, given a graph and a scoring metric, we aim to find the best value of k such that the score of the k-core (or k-core set) is the highest. The problem is challenging because there are various community scoring metrics and the computation is costly on large datasets. With the well-designed vertex ordering, we propose time-and-space-optimal algorithms to compute the best k, which are applicable to most community metrics. As real-world networks are often fast-evolving, we also design a novel framework to maintain the best k-core (set) against graph dynamics. We prove the dynamic algorithms are bounded, i.e., the update cost is decided by the changes of input and output. The proposed algorithms can benefit the solutions to k-core-related problems and their dynamic counterparts. Extensive experiments are conducted on 10 real-world networks with size up to billion-scale, which validates the efficiency of our algorithms and the effectiveness of the resulting k-cores.

Index Terms—Cohesive subgraph, Core decomposition, k-core, Dynamic graph, Densest subgraph

1 Introduction

As a fundamental problem in network analysis, cohesive subgraph mining is to extract groups of densely connected vertices. The well-studied model of k-core is defined as a maximal *connected* subgraph where every vertex is connected to at least k other vertices in the same subgraph [32], [43]. We use k-core set to denote the subgraph formed by all the (connected) k-cores in the graph, for a fixed parameter k. The hierarchical decomposition by k-core (or core decomposition) computes the k-core set for every possible k. The hierarchy of core decomposition builds on the containment property of k-core sets, that is, the $(k+1)$-core set is always a subgraph of the k-core set.

The k-core and its hierarchy have a wide range of applications [22], such as finding communities in the web [19] and social networks [20], [47], [62], discovering molecular complexes in protein interaction networks [3], recognizing hub-nodes in brain functional networks [7], analyzing the structure and functional consequences of the Internet [9], understanding software systems [57], and predicting structural collapse in mutualistic ecosystems [35].

The k-core is desired in comparison with other community detection techniques because its computation only takes linear time and the constraint on vertex degree is fundamental in computing the communities. However, it is hard to specify the value of k because different applications have different focuses [22]. This motivates our study to find the best k and maintain the solutions.

Discovering the Best k. Despite the elegant structure of k-core hierarchy and the various applications, it is still a common open problem to find the best k-core set or the best single k-core. To our knowledge, there is no previous work that studies the best k value for k-core and for other cohesive subgraph models, e.g. k-truss, k-plex, k-vcc, and k-ecc. The existing solution can only use a user-defined k, or to decide k by trivial statistics, e.g., setting k to average vertex degree. The underlying problem is essentially to ask the quality of the k-cores, since different k values lead to different k-cores. In this paper, given a graph and a community scoring metric, we aim to compute a k value such that the k-core set has the highest score among all the k-core sets for every integer k. We also aim to find the best single k-core among all the k-cores for every integer k.

There is no uniform metric to evaluate the quality of a subgraph [54]. Different community scoring metrics are designed for optimizing different network properties [10], [54]. For example, the clustering coefficient is a goodness metric for clustering tendency [50], and modularity measures the quality of a graph partition [36], [37]. To apply our proposed algorithms to various community metrics and even new metrics, we extract representative primary values that can be utilized to calculate most community scores [10].

A basic solution for the problem is to conduct quality

- *D. Chu and W. Zhang are with the University of New South Wales, Australia. E-mail: {deming.chu, wenjie.zhang}@unsw.edu.au.*
- *F. Zhang is with the Cyberspace Institute of Advanced Technology, Guangzhou University, China. E-mail: zhangf@gzhu.edu.cn.*
- *X. Lin is with the Antai College of Economics and Management, Shanghai Jiao Tong University, China. E-mail: xuemin.lin@sjtu.edu.cn.*
- *Y. Zhang is with School of Computer Science & Information Technology and School of Statistics & Mathmatics, Zhejiang Gongshang University. E-mail: ying.zhang@zjgsu.edu.cn.*
- *Y. Xia is with Meta AI, USA. E-mail: yxia@meta.com.*
- *C. Zhang is with Huawei Technologies, China. E-mail: zhangchenyi2@huawei.com.*

Manuscript received; revised.
Fan Zhang is partially supported by NSFC (62002073) and Guangdong Basic and Applied Basic Research Foundation (2023A1515012603, 2024A1515011501). Deming Chu is supported by the scholarship of China Scholarship Council No. 202006140012. Wenjie Zhang is supported by ARC DP230101445 and ARC FT210100303. Xuemin Lin is supported by NSFC U2241211 and NSFC U20B2046.
(Corresponding author: Fan Zhang.)

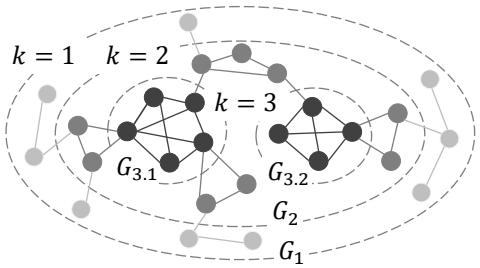


Fig. 1. Hierarchical Core Decomposition

computation on the k-core set for every possible input of k, which is costly. To efficiently handle billion-scale graphs, we design a light-weight vertex ordering technique to facilitate score computation. We compute the score of k-core sets from the center core of the graph to the margins, with optimal time and space cost.

Our paradigm can also find the best single k-core when the connectivity of different k-cores is considered. The score of every k-core can be retrieved by our algorithms. The k-core computation module with the best input k can support applications such as community search and improve algorithm efficiency. Particularly, our algorithm can help solve k-core related problems, e.g., finding the densest subgraph, maximum clique, and size-constrained k-core (Section 7.6).

Example 1. Figure 1 illustrates the hierarchy of a network decomposed by the k-core model, varying k from 1 to 3. When $k = 1$, the 1-core (set) G_1 is whole graph. When $k = 2$, we recursively delete every vertex with a degree less than 2, resulting in the 2-core (set) G_2. When $k = 3$, the 3-core set consists of two 3-cores: $G_{3.1}$ and $G_{3.2}$. Suppose the scoring metric is the average degree (twice the number of edges divided by the number of vertices), (i) the best k-core set is the 3-core set, as it has the highest score (average degree) among all the k-core sets, and the best k is 3; and (ii) the best single k-core is $G_{3.1}$ as it has the highest score among all the k-cores with different k.

Maintaining the Best k. Cohesive subgraphs are studied extensively on dynamic graphs where edges are inserted/removed constantly [24], [31], [59], as real-world graphs can be highly dynamic, e.g, orders are constantly created between E-commerce accounts. Thus, we also study the maintenance of the best k-core (set) against graph dynamics. The problem is challenging, as an edge insertion/removal can significantly change the k-cores and its hierarchy according to [31], [59], not to mention the scores of k-cores.

A baseline is to maintain the index of our static algorithm, and query the best k-core (set) with the static algorithms. However, this baseline is sub-optimal, because updating the index and recomputing the scores with the index still incurs a high time overhead.

To solve the above issues, we propose a novel framework for updating the best k-core (set), based on the equations that capture the effect of graph updates on the scores of k-cores. The update cost of our algorithms is bounded, meanwhile, the update and query costs are both largely reduced compared with the baseline. Our algorithms can also outperform the SOTA of the k-core related problems on dynamics graphs, e.g., dynamic densest subgraph [42].

Contributions. In the conference version of this paper [16], we proposed the time-and-space-optimal algorithms for discovering the best k (and k-core). In this paper, we substantially extend the study. In particular, we devise efficient dynamic algorithms for maintaining the best k (and k-core set) when the graph is updated frequently.

However, it is non-trivial to design algorithms with optimal time-and-space complexities and extend this idea to various metrics, to the case when connectivity is considered, and to the graphs with frequent updates. Our paper tackles all these challenges under a unified framework. In summary, our principal contributions are as follows.

- To our best knowledge, this is the first work to study the following problems: given a graph G and a community scoring metric Q, (i) find the best k-core set; and (ii) find the best single k-core, according to Q.
- We develop algorithms to solve our proposed problems in worst-case optimal time and space complexities. For many metrics, our algorithms can compute the score in $O(n)$ time where n is the number of vertices. The algorithms can be applied to various community metrics.
- For graphs with frequent updates, we propose a novel framework to maintain the answers to our proposed problems. Our dynamic algorithms are bounded, i.e., the update cost of our algorithms is decided by the changes of input and output.
- We conduct experiments on 10 real graphs with up to billions of edges, and show that the k-core (set) with the resulting k has a better quality than that with user-defined k or other trivial values. Towards efficiency, our static algorithms outperform the static baselines by 1-4 orders, while our dynamic algorithms outperform the dynamic baselines by up to 6 orders. For some k-core related problems and their dynamic counterparts, our algorithms can serve as good approximate solutions or benefit the algorithm design (Section 7.6).

Organization. Section 2 defines preliminary concepts. Sections 3 and 4 propose the algorithms for finding the best k and the best single k-core, respectively. Sections 5 and 6 devise the dynamic algorithms for maintaining the best k and the best single k-core, respectively. Section 7 evaluates our algorithms. Section 8 reviews the related works and Section 9 concludes the paper.

2 PRELIMINARIES

We consider an undirected and unweighted simple graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges (assume $m > n$). Given a vertex v in a subgraph S, $N(v, S)$ denotes the neighbor set of v in S, i.e., $N(v, S) = \{u \mid (u, v) \in E(S)\}$. The degree of v in subgraph S, i.e., $|N(v, S)|$, is denoted by $d(v, S)$. We omit the input graph G in the notations when the context is clear, e.g., we abbreviate $N(v, G)$ to $N(v)$. Table 1 summarizes the notations.

2.1 Core Decomposition

The model of k-core [32], [43] is defined as follows.

Definition 1 (k-Core). Given a graph G and an integer k, a subgraph S is a k-core of G, if (i) each vertex $v \in S$

TABLE 1
Summary of Notations

Notation	Definition
$G = (V, E)$	an undirected, unweighted simple graph
$n; m$	the number of vertices/edges in G ($m > n$)
S	a subgraph of G
$V(S)$; $E(S)$	the set of vertices/edges in S
$id(v)$	the unique identification of v in G
$N(v)$; $N(v, S)$	the set of neighbors of v in G / in S
$d(v)$; $d(v, S)$	the degree of v in G / in S
C_k	the k-core set of G
$c(v)$	the coreness of v in G, $\max\{k \mid v \in C_k\}$
k_{max}	the largest k s.t. C_k is not empty
H_k	the k-shell of G, $\{v \mid c(v) = k\}$
$rank(v)$	the vertex rank of vertex v
$n(S)$; $m(S)$	the number of vertices/edges in S
$b(S)$	the number of boundary edges in S
$\Delta(S)$; $t(S)$	the number of triangles/triplets in S
E'	an edge set to insert or remove
V^*	the set of vertices with coreness changed
$\|S\|_c$	the size of the c-hop neighbors of S

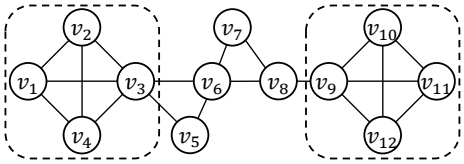


Fig. 2. A 2-core graph in which the 3-cores are circled

has at least k neighbors in S, i.e., $d(v, S) \geq k$; (ii) S is connected; and (iii) S is maximal, i.e., any supergraph of S is not a k-core except S itself.

Definition 2 (k-Core Set). Given a graph G and an integer k, the k-core set of G, denoted by C_k, is the subgraph formed by all the (connected) k-cores in G.

Given $k' \geq k$, the k'-core set is always a subgraph of the k-core set, i.e., $C_{k'} \subseteq C_k$. This implies that each vertex in a graph has a unique number of coreness [5].

Definition 3 (Coreness and k-Shell). Given a graph G, the coreness of a vertex $v \in G$ is $c(v) = \max\{k \mid v \in C_k\}$, i.e., the largest k such that the vertex v is in the k-core. The k-shell of G is $H_k = \{v \mid c(v) = k\}$, i.e., the set of vertices with coreness k.

Definition 4 (Core Decomposition). Given a graph G, core decomposition is to compute the coreness $c(v)$ for every vertex $v \in V(G)$.

The algorithm for core decomposition runs in $O(m)$ time complexity [5], and it recursively removes the vertex with the smallest degree in the graph. Besides, k_{max} (graph degeneracy) is decided by the input graph G, i.e., it is the largest k such that the k-core of G is not empty.

Example 2. Figure 2 shows a graph of 12 vertices. The graph itself is a 2-core. When $k = 3$, the k-core computation recursively deletes every vertex with a degree less than 3, i.e., v_5, v_7, v_6 and v_8. The 3-core set is the induced subgraph by the rest vertices. The coreness of v_5, v_7, v_6 and v_8 is 2, and the coreness of other vertices is 3. There are two connected components (3-cores) in the 3-core set, as circled in the figure.

2.2 Problem Definition

We are the first to propose and study the problems below.

Problem (Discovering the Best k). Given a graph G.

1) discover the k^*-core set C_{k^*} such that it has the highest score among all the k-core sets for every integer k with $0 \leq k \leq k_{max}$;
2) discover the single k^*-core S^* such that it has the highest score among all the k-cores for every integer k with $0 \leq k \leq k_{max}$.

Problem (Maintaining the Best k). Given a dynamic graph changed from G to G'.

1) update the best k-core set from $C_{k^*}(G)$ to $C_{k^*}(G')$;
2) update the best k-core from $S^*(G)$ to $S^*(G')$.

2.3 Community Scoring Metrics

Although there are various community scoring metrics considering different community properties, most of them are constructed on the following primary values [54].

Primary Values. Let S be a subgraph to evaluate, we study the following common primary values.

- $n(S)$: the number of vertices in S, i.e., $n(S) = |V(S)|$;
- $m(S)$: the number of edges in S, i.e., $m(S) = |E(S)|$;
- $b(S)$: the number of boundary edges in S, $b(S) = |\{(u, v) \mid (u, v) \in E, u \in S, v \notin S\}|$;
- $\Delta(S)$: the number of triangles in S;
- $t(S)$: the number of triplets in S, i.e., $t(S) = \sum_{v \in S} \binom{d(v,S)}{2}$, where a triplet is three vertices connected by two edges;

Metrics. Based on the above primary values, some community scoring metrics are defined as follows.

- Average Degree: $f(S) = \frac{2 \times m(S)}{n(S)}$ is the average degree of vertices in S;
- Internal Density: $f(S) = \frac{2 \times m(S)}{n(S) \times (n(S)-1)}$ is the internal density of vertices in S;
- Cut Ratio: $f(S) = 1 - \frac{b(S)}{n(S) \times (n - n(S))}$ is the difference of 1 and the fraction of existing boundary edges over all the possible ones;
- Conductance: $f(S) = 1 - \frac{b(S)}{2 \times m(S) + b(S)}$ is the difference of 1 and the proportion of degrees where the vertex points to the outside;
- Modularity: the modularity for a partition P on G is $f(P) = \sum_{i=1}^{k} \left(\frac{m(P_i)}{m} - \left(\frac{2 \times m(P_i) + b(P_i)}{2 \times m} \right)^2 \right)$, where P_i is the i^{th} community in the partition [37]. We regard different (connected) k-cores and the rest nodes as a partition, and compute the Modularity of this partition.
- Clustering Coefficient: $f(S) = \frac{3 \times \Delta(S)}{t(S)}$ measures the tendency of vertices to cluster together;

Given a scoring metric, for an integer k, the target subgraphs for computing the scores are all the k-cores with the given k, i.e., the subgraphs in C_k. Note that our algorithms are not limited to handling the above metrics, i.e., they can handle any metric formed by the above primary values.

3 Finding the Best K-Core Set

In this section, we propose the solution to find the best k for the k-core set, including the baseline in Section 3.1, and the improved algorithm in later sections.

3.1 Baseline Algorithm

Given a graph G and a community scoring metric Q, a baseline solution is to first conduct core decomposition, order the vertices by coreness to fast retrieve a k-core set, and then compute the score of every k-core set according to Q, for every integer k with $0 \leq k \leq k_{max}$.

We use a bin sort to order the vertices in G by coreness, and record every start position of the vertices with the same coreness, which takes $O(n)$ time. Then, retrieving the vertex set of a k-core set C_k takes $O(|V(C_k)|)$ time. Let $O(q_k)$ be the time cost to compute the score of C_k given $V(C_k)$, Q and G. The time complexity of the baseline algorithm is $O\left(\sum_{k=0}^{k_{max}} \left(q_k + |V(C_k)|\right)\right)$. Although the baseline runs in polynomial time, it is costly to handle large graphs.

3.2 Vertex Ordering for Optimal Neighbor Query

To compute the best k in optimal time and space cost, we first introduce the vertex ordering techniques.

We divide the vertices in G into $k_{max}+1$ arrays where each is associated with a unique coreness value. The arrays are ordered by the associated coreness to fast locate the vertices with different coreness.

For every vertex v in G, the neighbor set of v is ordered by increasing values of vertex rank that is defined as follows.

Definition 5 (Vertex Rank). Given vertices u and v, the rank of v is higher than u, denoted by $rank(v) > rank(u)$, if (i) $c(v) > c(u)$; OR (ii) $c(v) = c(u)$ and $id(v) > id(u)$, where $id(v)$ is the unique identification of v.

For every vertex v, to efficiently retrieve specific parts of its neighbor set, we also record some position tags where $same$ is the position of the first u in $N(v)$ such that $c(u) \geq c(v)$, $plus$ is the position of the first u in $N(v)$ such that $c(u) > c(v)$, and $high$ is the position of the first u in $N(v)$ such that $rank(u) > rank(v)$. Each of the above tags is set to $|N(v)|$ when there is no such $u \in N(v)$ satisfying the condition. Table 2 summarizes how to handle the neighbor queries based on the vertex ordering. Note that our proposed algorithms in the following sections (implicitly) use the vertex ordering whenever they call the neighbor queries in Table 2.

Algorithm 1 shows the pseudo-code for vertex ordering. Lines 1-4 use $k_{max}+1$ bins to sort vertices by rank, where each bin represents a unique value of coreness. Inspired by [26], we flatten $k_{max}+1$ bins to sort the edge set: (i) At Lines 5-8, for every vertex v with ascending vertex id, we push the pair (v, u) from every neighbor u of v into the bin represented by the coreness of v; (ii) Lines 9-11 retrieve the sorting from the $k_{max}+1$ bins by one scan from bin 0 to bin k_{max}, where $N'(u)$ is empty initially for every $u \in V$. For every element (v, u) in the bins, we push v into the new neighbor set of u s.t. the neighbors of u are ordered by vertex rank. Besides, Line 13 records the subscripts for each vertex according to Table 2, via one scan of the new edge set.

Complexity. Algorithm 1 costs $O(m)$ time and $O(m)$ space. After running Algorithm 1, we can answer any neighbor query $N(v, \cdot)$ listed in Table 2 using $O(|N(v, \cdot)|)$ time, and return any $|N(v, \cdot)|$ in $O(1)$ time, e.g., $|N(v, =)| = plus - same$. See our conference version paper [16] for the proofs.

Example 3. Figure 3 illustrates the vertex ordering for the graph in Figure 2. Each vertex in Figure 3 is colored according to different coreness values. On the top, all the vertices are ordered by coreness with ties broken by vertex id. The vertices in every neighbor set are also sorted by the above order. In the figure, we show the ordered neighbor sets of four vertices and their position tags. For vertex v_1 and v_9, the position $plus$ equals to the length of the neighbor set because the coreness of any neighbor is not larger than the vertex. By checking the ordering results and Table 2, all types of $|N(v, \cdot)|$ queries can be answered in $O(1)$, e.g., $|N(v_6, >)| = |N(v_6)| - plus = 1$.

TABLE 2
The Ordered Neighbor Set of v

Subscript of $u \in N(v)$	Constraint of u	Vertex Set
$[0, same-1]$	$c(u) < c(v)$	$N(v, <)$
$[same, plus-1]$	$c(u) = c(v)$	$N(v, =)$
$[plus, \|N(v)\|-1]$	$c(u) > c(v)$	$N(v, >)$
$[same, \|N(v)\|-1]$	$c(u) \geq c(v)$	$N(v, \geq)$
$[high, \|N(v)\|-1]$	$rank(u) > rank(v)$	$N(v, >_r)$

Every u in $N(v)$ is ordered by *ascending* order of rank;
$\underline{same}$: position of the first $u \in N(v)$ s.t. $c(u) \geq c(v)$;
$\underline{plus}$: position of the first $u \in N(v)$ s.t. $c(u) > c(v)$;
$\underline{high}$: position of the first $u \in N(v)$ s.t. $rank(u) > rank(v)$.

Algorithm 1: vertex ordering

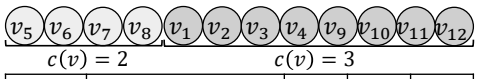
Input : a graph $G = (V, E)$, the coreness $c(v)$ of every vertex $v \in V$
Output : G with ordered V and E by vertex rank, and position tags

```
/*                Order vertex set V                */
1  bin1 ← a list of (k_max + 1) empty arrays;
2  for each v in V with ascending order of vertex id do
3      bin1[c(v)].push_back(v);
4  V ← concatenation of bin1[0], bin1[1], ..., bin1[k_max];
/*                Order edge set E                  */
5  bin2 ← a list of (k_max + 1) empty arrays;
6  for each v in V with ascending order of vertex id do
7      for each u in N(v) do
8          bin2[c(v)].push_back(pair(v, u));
9  for each i from 0 to k_max do
10     for each element (v, u) in bin2[i] from head to tail do
11         N'(u).push_back(v);
12 E ← each v ∈ V has neighbors N'(v);
13 same, plus, high ← scan N'(v) for every v ∈ V;
14 return G with ordered V and E, and position tags
```

3.3 Improved Algorithm for Best K-Core Set

An important property of hierarchical core decomposition is its containment nature: $C_{k+1} \subseteq C_k$ for every coreness k. This suggests that the primary values of the k-core set can be incrementally derived from the primary values of the $(k+1)$-core set, with minor modification according to their difference. Based on the above observation, we compute the scores of the k-core sets in a top-down manner, i.e., for k from k_{max} to 0. Note that it is costly to count some primary values in a bottom-up manner, that is, running the baseline in Section 3.1 for k from 0 to k_{max}.

Algorithm 2 shows the pseudo-code for computing the best k. The algorithm incrementally computes and updates the primary values of k-core sets. Note that we present the



Vertex	Ordered Neighbors	*same*	*plus*	*high*
v_1	v_2 v_3 v_4	0	3	0
v_6	v_5 v_7 v_8 v_3	0	3	1
v_8	v_6 v_7 v_9	0	2	2
v_9	v_8 v_{10} v_{11} v_{12}	1	4	1

Fig. 3. Example of Vertex Ordering

Algorithm 2: computing the best k

Input : a graph G, a community scoring metric Q
Output: the best k value for a k-core set

```
1  compute each k-core set C_k by core decomposition;
2  order G by Algorithm 1;
3  metric ← [0, ..., 0];
4  in ← 0, out ← 0, num ← 0;
5  for each integer k from k_max to 0 do
6      for v ∈ H_k do
7          in ← in + |N(v,>)| + (1/2)|N(v,=)|;
8          out ← out + |N(v,<)| − |N(v,>)|;
9          num ← num + 1;
10     metric[k] ← get_metric(Q, in, out, num);
11 return metric
```

computation for the clustering coefficient in Section 3.4. In Algorithm 2, array `metric` records the score of every k-core set. Let S be the subgraph induced by the visited vertices at Line 6. The variable `in` records the number of internal edges of S. The variable `out` records the number of boundary edges of S where each edge has exactly one endpoint in S. The variable `num` records the number of visited vertices.

Lines 3-4 initialize array `metric` and the primary values. Lines 5-10 incrementally compute the score of the k-core set from $k = k_{max}$ to $k = 0$. For each value of k, we only visit the vertices with coreness k at Line 6, and update the primary values at Line 7-9. Based on the primary values for the $(k+1)$-core set, we update the values by considering every neighbor u of the visited vertex v as follows.

- If $c(u) = c(v)$, then (u, v) should be counted in the value `in`. Line 6 will count (u, v) in `in` when we visit both u and v, so we add $\frac{1}{2}|N(v, =)|$ to `in` for visiting v.
- If $c(u) < c(v)$, then (u, v) should be counted in the value `out`. Line 6 will count (u, v) in `out` only when v is visited, so we add $|N(v, <)|$ to `out` for visiting v.
- If $c(u) > c(v)$, then $u \in C_{k+1}$ and $(u, v) \notin C_{k+1}$, i.e., (u, v) is a boundary edge of the $(k+1)$-core set. For the k-core, (u, v) becomes an internal edge. So we add $|N(v, >)|$ to `in` and subtract $|N(v, >)|$ from `out`.

Line 10 computes the score from the primary values, e.g., the average degree of a k-core set is $2 \times$ `in`$/$`num`, the cut ratio is $1 -$ `out`$/($`num`$\times (n -$ `num`$))$, etc. Line 11 returns the result.

Example 4. Given a community scoring metric such as average degree, Algorithm 2 runs on the graph in Figure 2 as follows. First, we compute the primary values of the 3-core set by visiting every vertex v in the 3-core set. Because there are 8 vertices in C_3, and every $v \in C_3$ has $|N(v, >)| = 0$ and $|N(v, =)| = 3$, the number of internal edges in the 3-core set is $\texttt{in} = 8 \times (0 + 3/2) = 12$. The average degree of the 3-core set is $2 \times 12/8 = 3$. Then we visit each vertex in 2-shell and incrementally count the internal edges in the 2-core set. When v_5, v_6, v_7 and v_8 are sequentially visited, we have $\texttt{in} = 12 + (1 + 1/2) + (1 + 3/2) + (0 + 2/2) + (1 + 2/2) = 19$. The average degree of the 2-core set is $2 \times 19/12 \approx 3.17$. Overall, the 2-core set is preferred due to the highest average degree.

Algorithm 3: computing k by triangles and triplets

Input : a graph G, a community scoring metric Q
Output: the best k value for a k-core set

```
1  metric ← [0, ..., 0];
2  triangle ← 0, triplet ← 0;
3  f_>[v] ← f_≥[v] ← 0 for each v ∈ V;
4  compute each k-core set C_k by core decomposition;
5  order G by Algorithm 1;
6  for each k from k_max to 0 do
7      for v ∈ H_k do
8          for u ∈ N(v, >_r) do
9              x ← u; y ← v;
10             swap(x, y) if deg(x) > deg(y) ;
11             for w ∈ N(x, >_r) do
12                 triangle ++ if w ∈ N(y, >_r);
13         triplet ← triplet + C(|N(v,≥)|, 2);
14     kshell_nbr ← ⋃_{u∈H_k} N(u, >);
15     for v ∈ kshell_nbr do
16         f_>[v] ← f_≥[v];
17     for v ∈ H_k do
18         for u ∈ N(v) do
19             f_≥[u] ++;
20     for v ∈ kshell_nbr do
21         gt_k ← f_>[v]; eq_k ← f_≥[v] − f_>[v];
22         triplet ← triplet + C(eq_k, 2) + gt_k × eq_k;
23     metric[k] ←
       get_metric(Q, triangle, triplet);
24 return metric
```

Complexity and Optimality. Algorithm 2 takes $O(m)$ time and $O(m)$ space. The complexity of Algorithm 2 is optimal. See our conference version paper [16] for the proofs.

3.4 Computation of Triangles and Triplets

Algorithm 3 shows the variant of Algorithm 2 for computing the best k when the community metric is based on triangles and triplets, e.g., clustering coefficient. Similar to Algorithm 2, we incrementally compute the primary values in a top-down manner: from $k = k_{max}$ to $k = 0$. For every k-core set, array `metric` records its score, `triangle` records the number of triangles, and `triplet` records the number of triplets. They are initialized in Line 1-2.

Triangle Counting. Lines 7-12 incrementally update the number of triangles in the k-core set based on that in the $(k+1)$-core set. Recall that the rank and the neighbor query $N(v, \cdot)$ are defined in Section 3.2. For each edge (u, v) with $rank(u) > rank(v)$ (Lines 7-8), we count the triangles containing (u, v) by visiting the neighbors of x where x is the one in $\{u, v\}$ with smaller degree (Lines 9-12). To

count triangles, we enumerate each vertex w in $N(x, >_r)$ and check if w is also in $N(y, >_r)$. By visiting $N(\cdot, >_r)$ of each vertex, we ensure $rank(w) > rank(u) > rank(v)$ for every counted triangle, i.e., every triangle is counted exactly once since we count the three vertices by increasing order of vertex rank.

Triplet Counting. Recall that a triplet $\langle u, v, w\rangle$ is formed by three vertices connected by two edges (v, u) and (v, w), centered on v. (i) For each center vertex v in H_k, any pair of the neighbors of v in the k-core can form a triplet centered on v. Line 13 adds the number of such pairs to `triplet`. (ii) Lines 14-22 count the new triplets centered on the vertices of the $(k+1)$-core set.

We use $f_>[v]$ and $f_\geq[v]$ to count the number of $u \in N(v)$ such that $c(u) > k$ and $c(u) \geq k$, respectively. Here we do not use $N(v, \cdot)$, because a neighbor query in Section 3.2 only works for vertices in the k-shell, i.e., $f_>[v] = |N(v, >)|$ and $f_\geq[v] = |N(v, \geq)|$ iff $k = c(v)$. Line 14 collects the vertices in $N(u, >)$ of every u with coreness k, because they are the center vertices in the $(k+1)$-core set. Line 15-19 computes $f_>[v]$ and $f_\geq[v]$ for every vertex v in the $(k+1)$-core set, by updating the values from the previous iteration.

Line 20 iterates over each center vertex in the $(k+1)$-core set, and Line 21 assigns to gt_k and eq_k the number of v's neighbor u such that $c(u) > k$ and $c(u) = k$, respectively. To form a triplet (uncounted) with a center vertex in the $(k+1)$-core set, (a) the other two vertices are from H_k; or (b) one vertex is from C_{k+1} and the other is from H_k. In Line 22, $\binom{eq\_k}{2}$ is the number of the triplets in case (a), and $gt\_k \times eq\_k$ is the number of the triplets in case (b).

Line 23 calculates community score using the primary values, e.g., the clustering coefficient of a k-core is $3 \times$ `triangle/triplet`. Line 24 returns the result.

Example 5. Given a community scoring metric such as clustering coefficient, the execution of Algorithm 3 on the graph in Figure 2 is as follows. We first compute the number of triangles and triplets in 3-core set by Line 7-13: `triangle` $= 8$ and `triplet` $= 24$. The clustering coefficient of 3-core is $3 \times 8/24 = 1$. Then, we incrementally compute the values for the 2-core set. Line 7-12 detect two new triangles (v_5, v_6, v_3) and (v_6, v_7, v_8), so `triangle` $= 8 + 2 = 10$. Line 13 detects $\binom{2}{2}$ new triplets for v_5, $\binom{4}{2}$ for v_6, $\binom{2}{2}$ for v_7, and $\binom{3}{2}$ for v_8. Line 22 iterates over `kshell_nbr`, that is $\{v_3, v_9\}$, and counts $\binom{2}{2} + 3 \times 2$ new triplets for v_3 and 3×1 new triplets for v_9. To sum up, `triplet` $= 24 + 21 = 45$. The clustering coefficient of 2-core is $3 \times 10/45 \approx 0.67$. So, the 3-core set is preferred when the metric is clustering coefficient.

Complexity and Optimality. Algorithm 3 takes $O(m^{1.5})$ time and $O(m)$ space. The complexity of Algorithm 3 is optimal. See our conference version paper [16] for the proofs.

4 Finding the Best K-Core

We first review the forest hierarchy of k-cores in Section 4.1. To find the best single k-core, we propose a baseline in Section 4.2 and the improved algorithm in Section 4.3.

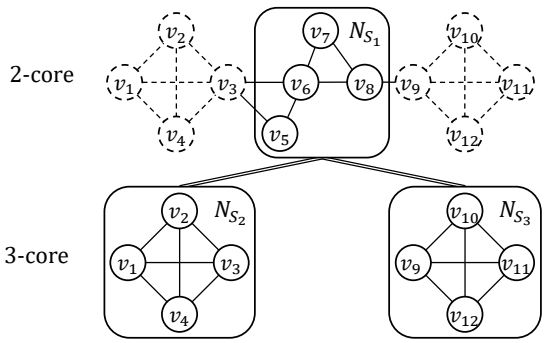


Fig. 4. A Core Forest (Tree)

4.1 Hierarchy of k-Cores

According to the definition of k-core, the following properties hold for every integer k:

- (Disjointness) every k-core is disjoint from each other in the same k-core set.
- (Containment) a k-core is contained by exactly one $(k-1)$-core.

The hierarchy of k-cores can be represented by a set of trees where each tree node is associated with a k-core, and each tree edge is the parent-child relationship between two tree nodes. For any k-core with k from 0 to k_{max}, we define the k-core tree node as follows.

Definition 6 (k-Core Tree Node). Given any k-core S, we build a k-core tree node N_S that contains (i) the vertices in S with coreness k (i.e., $S \cap H_k$) when $S \cap H_k$ is not empty, and (ii) the pointer to its parent tree node.

The vertices in a k-core tree node are not certainly connected, because the k-shell vertices in a k-core may be separated by other vertices with a higher coreness.

Definition 7 (Parent Tree Node). Given a graph G, a k_1-core S_1 associated with N_{S_1}, and a k_2-core S_2 associated with N_{S_2}, k_1-core tree node N_{S_1} is the parent of k_2-core tree node N_{S_2}, if (i) $k_1 < k_2$; (ii) $S_2 \subset S_1$; and (iii) any k'-core tree node with $k_1 < k' < k_2$ is not the parent of N_{S_2}.

Given a k-core S associated with N_S, the tree node contains the k-shell vertices while leaving the rest in its descendants. Assume N_S has c children $N_{C_1}, N_{C_2} \cdots N_{C_c}$, the original S can be reconstructed recursively by the vertices in N_S and $\bigcup_{i=1}^{c} C_i$, if every C_i has been reconstructed before.

Definition 8 (Core Forest). Given a graph G, the core forest is constituted by all k-core tree nodes for every integer k from 0 to k_{max}, where every tree node is connected to its parent if it exists.

The core forest organizes all k-cores into a hierarchy that can be stored in $O(n)$ space, because each vertex exists in one tree node and each tree node has one parent.

Example 6. Figure 4 shows the core forest of Figure 2 which has only one tree. There are three tree nodes N_{S_1}, N_{S_2}, and N_{S_3}, associated with S_1, S_2, and S_3, respectively. N_{S_1} is associated with the whole graph (a 2-core) while only the vertices in the 2-shell are contained in N_{S_1}. Although N_{S_1} is also a 1-core, no 1-core tree node would be built according to our definition. We can reconstruct S_1 from the vertices in N_{S_1}, S_2, and

Algorithm 4: computing the best single k-core

Input : a graph G, a community scoring metric Q
Output: the best single k-core according to Q

```
1  metric ← [0, ..., 0];   pri_val ← [0, ..., 0];
2  compute the coreness by core decomposition;
3  order G by Algorithm 1;
4  construct the forest T by LCPS [32];
5  for each i from 0 to T.size() − 1 do
6      for each tv ∈ T[i].child do
7          pri_val[i] ← pri_val[i] + pri_val[tv];
8      for each v ∈ T[i].delta do
9          pri_val[i] ← pri_val[i] + impact of v;
10     metric[i] ←get_metric(Q, pri_val[i]);
11 return the k-core with highest score in metric
```

S_3. Also, we have $|S_1| = |N_{S_1}| + |S_2| + |S_3|$, and $m(S_1) = m(N_{S_1}) + m(S_2) + m(S_3) + 3$ (boundary edges).

Forest Construction. The SOTA for constructing the forest of k-cores is LCPS [32] (Level Component Priority Search). The time complexity of LCPS is $O(m)$, if a bucket data structure is applied [41]. The pseudo-code of LCPS is provided in [16].

To adapt LCPS for our needs, three steps are required: (i) run LCPS [32]; (ii) compress empty tree nodes; and (iii) sort all remaining tree nodes in descending order of the coreness of elements and store them into an array $\mathcal{T}$.

4.2 Baseline Algorithm

Given a graph G and a community scoring metric Q, a baseline solution is to first conduct core decomposition, and forest construction for fast retrieval of a k-core. Then, it computes the score of every k-core according to Q, for every integer k from 0 to k_{max}.

By visiting the forest of k-cores, it takes $O(|V(S_i)|)$ time to retrieve the vertex set of every k-core S_i. Let $O(q_i)$ be the time cost to compute the score of S_i given $V(S_i)$, Q and G. The time complexity of the baseline algorithm is $O\left(\sum_{k=0}^{k_{max}} \sum_{S_i \in C_k} \left(q_i + |V(S_i)|\right)\right)$. Although the baseline is polynomial-time, it is still costly to handle large graphs.

4.3 Improved Algorithm for Best Single K-Core

We apply the vertex ordering in Section 3.2 and the forest structure in Section 4.1 to design the improved algorithm. The pseudo-code is shown in Algorithm 4. Array `metric` stores the score of each k-core. Array `pri_val` stores the primary values of each k-core. Here we use one variable `pri_val`$[i]$ to illustrate the computation of all the primary values at tree node i. Array $\mathcal{T}$ stores the tree nodes. $\mathcal{T}[i]$ is the i^{th} element (a tree node) in $\mathcal{T}$, where i is the id of $\mathcal{T}[i]$.

At Line 5, the algorithm processes each tree node in $\mathcal{T}$ sequentially, as the nodes in $\mathcal{T}$ represent all the k-cores for every integer k from k_{max} to 0. At Line 6-7, for each k-core, its primary values are incrementally computed, based on the values from the child nodes (Line 8-9) and the additional values caused by current tree node (Line 8-10). Line 10 computes the score according to the metric Q and the primary values. Line 11 returns the best single k-core.

Primary Values. Line 7 updates the primary values `in`, `out`, and `num` in three arrays respectively, like `pri_val`. To compute these primary values, Line 9 can be replaced by Line 7-9 of Algorithm 2 where `in`, `out`, `num` are replaced by `in`$[i]$, `out`$[i]$, and `num`$[i]$, respectively. To compute the primary values `triangle` and `triplet`, we replace Line 9 by Line 7-22 of Algorithm 3 where `triangle`, `triplet` become `triangle`$[i]$ and `triplet`$[i]$, respectively.

Complexity and Optimality. The space complexity is $O(m)$. Algorithm 4 takes $O(n)$ time for primary values `in`, `out`, and `num`, and $O(m^{1.5})$ time for primary values `triangle` and `triplet`. The complexity of Algorithm 4 is optimal for any metric in Section 2.3. See our conference version paper [16] for the proofs.

5 Maintaining the Best K-Core Set

In this section, we maintain the best k-core set on dynamic networks regarding different community metrics. We first introduce a baseline that maintains the proposed vertex ordering, and then propose an improved solution with a new framework.

Type-A and Type-B Metrics. We divide the metrics in Section 2.3 into two categories according to computing complexity. The type-A metric is defined on $n(S)$, $m(S)$ and $b(S)$, while the type-B metric is defined on graph motifs [6] including $\Delta(S)$ and $t(S)$.

5.1 A Baseline with Vertex Ordering Maintenance

Given the index (vertex ordering and coreness), our proposed static score computation is quite efficient, e.g., it costs $O(n)$ time on the type-A metric. Thus, a baseline solution is to update the index (vertex ordering and the coreness of each vertex) and query the best k-core set by the static score computation (Algorithm 2 or 3) with the index. We use core maintenance [59] to efficiently update the coreness. The vertex ordering is updated as follows.

Maintenance Analysis. Vertex ordering maintenance needs to update (i) the neighbor sets ordered by vertex rank, (ii) the position tags in the neighbor sets (Table 2), and (iii) the vertices ordered by vertex rank. Case (iii) can be instantly updated after core maintenance because the vertex rank of a vertex is decided by its coreness and vertex id.

In the following, we update the vertex ordering for inserting (resp. removing) an edge set E'.

1. Update by Edge Change: for every $(u, v) \in E'$, we (i) use insertion sort to insert v into $N(u)$ (resp. remove v from $N(u)$) such that $N(u)$ remains ordered by vertex rank and (ii) increase (resp. decrease) $same(u)$ by 1 if $c(v) < c(u)$, $plus(u)$ by 1 if $c(v) \leq c(u)$, and $high(u)$ by 1 if $r(v) < r(u)$.

2. Update by Coreness Change: let V^* denote the set of vertices with coreness changed by E'. For every $v \in V^*$ whose coreness changes from $c(v)$ to $c'(v)$, we (i) move v backward (resp. forward) in any adjacency list containing v until the list is ordered by vertex rank again and (ii) update position tags for v and every neighbor $u \in N(v)$ with $c(v) \leq c(u) \leq c'(v)$ (resp. $c'(v) \leq c(u) \leq c(v)$).

Complexity. Let $O(\mathcal{CM})$ be the cost of core maintenance [59] and $O(\mathcal{VOM}) = O(\|E'\|_1 + \|V^*\|_2)$ be the cost of vertex ordering maintenance, where $\|V^*\|_c$ is the size of the

Algorithm 5: $\texttt{ins}^*/\texttt{rem}^*_{\texttt{A}}(G, c, E')$

```
1  for each (v,u) ∈ E' do upd_e_A(v,u);
2  V* ← vertices with coreness changed to c'(v);
3  for each v ∈ V* do upd_cn_A(v);
4  def upd_e_A(v,u):
5      lo, hi ← min and max of c(v) and c(u);
6      I_lo ++; O_hi ++; O_lo −−; (opposite sign if rem*);
7      G ← G with (v,u) inserted or removed;
8  def upd_cn_A(v):
9      N_{c(v)} −−; N_{c'(v)} ++;
10     for each u ∈ N(v) do
11         upd_e_A(v,u) for removing (v,u);
12         upd_e_A(v,u) for inserting (v,u) back, where the
           coreness of v is replaced by c'(v);
13     c(v) ← c'(v);
14 def query_A( ):
15     for each integer k from k_max to 0 do
16         num ← Σ_{i=k}^{k_max} N_i;  in ← Σ_{i=k}^{k_max} I_i;  out ← Σ_{i=k}^{k_max} O_i;
17         Q(C_k) ← compute_metric(in, out, num);
18     return k* with the highest Q(C_{k*});
```

c-hop neighborhood of V^* and $\|E'\|_c$ is the size of the c-hop neighborhood of all endpoints in E'. The baseline can (i) update index in $O(\mathcal{CM} + \mathcal{VOM})$ time and (ii) query the best k in $O(n)$ or $O(m^{1.5})$ time by Algorithm 2 or 3.

5.2 Dynamic Algorithm for Best K-Core Set (Type-A)

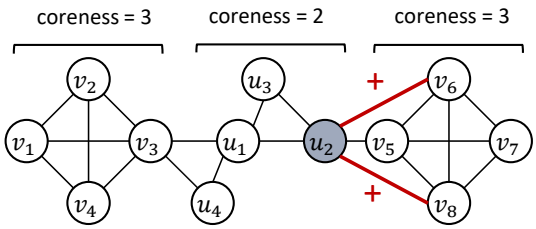
Although the baseline can maintain the best k, its update of the index is costly and its query has to recompute the score of each k-core set, which should be further optimized. To maintain the scores of the k-core sets efficiently, we propose a new framework to incrementally maintain Equation 1 that formalizes the increment of primary values when Algorithm 2 visits every k-shell H_k. The scores and the best k can then be quickly computed from the equations.

$$\begin{aligned} N_k &= \sum_{v \in H_k} 1 = |H_k| \\ I_k &= \sum_{v \in H_k} |N(v,>)| + \frac{1}{2}|N(v,=)| \\ O_k &= \sum_{v \in H_k} |N(v,<)| - |N(v,>)| \end{aligned} \tag{1}$$

In Equation 1, $N(v,>)$ (resp. $N(v,=)$ or $N(v,<)$) records the subset of v's neighbors with coreness larger than (resp. equal to or less than) v (defined in Table 2).

For every integer k from k_{max} to 0, N_k (resp. I_k or O_k) is the *increment* of the number of vertices (resp. edges or boundary edges), i.e., `num` (resp. `in` or `out`), when Algorithm 2 processes the k-shell H_k. The score of every k-core set and the best k can be computed from N, I, O, e.g., the average degree of C_k equals $2 \times \sum_{i=k}^{k_{max}} I_i / \sum_{i=k}^{k_{max}} N_i$.

Algorithm 5 maintains the best k-core set for a type-A metric. N, I, O are either maintained in the graph or initialized by Equation 1. Then, we can (i) *update*: Lines 1-3 update N, I, O by edge and coreness change, in two steps, such that Equation 1 holds after inserting/removing E'. OR (ii) *query*: Lines 15-18 ($\texttt{query}_\texttt{A}$) query the best k-core set.



	G	Step 1	Step 2
I_2	7	9	6
I_3	12	12	15
O_2	-3	-5	-4
O_3	3	5	4

Step 1. Update by Edge Change
$+ (u_2, v_6), (u_2, v_8)$

Step 2. Update by Coreness Change
$c(u_2) = 2 \rightarrow 3$

Fig. 5. An Example of $\texttt{ins}^*/\texttt{rem}^*_{\texttt{A}}$ (Algorithm 5)

Step 1. Update by Edge Change (Line 1). The edge change will affect $N(v,>)$, $N(v,=)$ and $N(v,<)$ in Equation 1, where v is an endpoint of the inserted/removed edge. $\texttt{upd\_e}_\texttt{B}$ updates I, O for every edge in E' without considering the coreness change. Let (u, v) be an edge with $c(u) \leq c(v)$. If $c(u) < c(v)$, then v is inserted into $N(u,>)$ and u is inserted into $N(v,<)$. Lines 5-6 update their size in I, O accordingly. Lines 5-6 also hold for $c(v) = c(u)$. We use the opposite sign for edge removal at Line 6.

Step 2. Update by Coreness Change (Line 3). The coreness of a vertex v may change many times in the update, starting from $c(v)$ and ending at $c'(v)$. Our algorithm only needs one update that moves v from $H_{c(v)}$ to $H_{c'(v)}$ in Equation 1.

$\texttt{upd\_cn}_\texttt{A}$ can update every vertex $v \in V^*$ whose coreness is changed from $c(v)$ to $c'(v)$ in N, I, O. Line 9 updates $N_{c(v)}$ and $N_{c'(v)}$ according to the coreness change. For every edge (v, u) incident to v, for conciseness, we update I and O by first removing (v, u) (Line 11) and then inserting it back (Line 12) with the updated coreness $c'(v)$.

Query from N, I, O (Lines 15-18). We use $\texttt{query}_\texttt{A}$ to accumulate the scores of k-core sets by suffix sum and find the best k. The query cost is $O(k_{max})$ time and is usually minor, e.g., $k_{max} \leq 2208$ in our datasets that are up to billion-scale.

Example 7. Figure 5 inserts (u_2, v_6) and (u_2, v_8) into G. Initially, there are $I_3 = 12$ edges in the 3-core set C_3 and 19 edges in C_2 ($I_2 = 7$ more edges than C_3). Step 1 inserts (u_2, v_6) and (u_2, v_8). Then, I_2 += 2, because $lo = 2$ for both edges. Step 2 updates the coreness of u_2 from 2 to 3. Line 11 first removes all five edges incident to u_2 and each of them has $lo = 2$, i.e., I_2 −= 5. After updating the coreness to 3, Line 12 inserts back the edges, where $lo = 2$ for (u_2, v_1) and (u_2, v_3), and $lo = 3$ for the other three edges, i.e., we have I_2 += 2 and I_3 += 3. Finally, we have $I_3 = 12 + 3 = 15$ and $I_2 = 7 + 2 - 5 + 2 = 6$, i.e., there are 15 internal edges in the 3-core set and $I_2 + I_3 = 21$ edges in the 2-core set.

For O_k, there are initially $O_3 = 3$ boundary edges in C_3, and no boundary edges ($O_2 + O_3 = 0$) in C_2. Step 1 inserts two new edges with $lo = 2$ and $hi = 3$. Thus, O_2 −= 2 and O_3 += 2. Step 2 first removes five edges incident to u_2 where three edges have $lo = 2, hi = 3$,

i.e., O_2 += 3, O_3 −= 3. All edges are then inserted back with coreness $c'(u_2) = 3$, and there are two edges with $lo = 2, hi = 3$, i.e., O_2 −= 2, O_3 += 2. Finally, $O_3 = 3+2-3+2 = 4$ and $O_2 = -3-2+3-2 = -4$, i.e., there are four boundary edges in C_3 and no boundary edge ($O_2 + O_3 = 0$) in C_2.

Correctness. For every k, the correctness of updating N_k is immediate. We prove $\texttt{upd\_e}_\text{A}$ and $\texttt{upd\_cn}_\text{A}$ can correctly maintain I_k and O_k. Then, the best k can be computed from Equation 1 as proved in Section 3.
($\texttt{upd\_e}_\text{A}$): Let (v, u) be an edge to insert where $lo = c(v) \leq c(u) = hi$, WLOG. When $c(v) < c(u)$, $|N(v, >)|$ and $|N(u, <)|$ are increased by 1, we update correctly by I_{lo}++, O_{lo}++, and O_{hi}−−, according to Equation 1. When $c(v) = c(u)$, both $|N(v, =)|$ and $|N(u, =)|$ are increased by 1, so we increase I_{lo} by $\frac{1}{2} \times 2 = 1$ and $O_{lo=hi}$ is unchanged. The proof is similar for edge removal with the opposite sign.
($\texttt{upd\_cn}_\text{A}$): Given a vertex $v \in V^*$ whose coreness changes from $c(v)$ to $c'(v)$. Line 9 correctly maintains N. Line 11-12 can correctly renew I_k and O_k by calling $\texttt{upd_e}_\text{A}$to remove every incident edge and insert it back with the updated corenesses of the endpoints.

Complexity. The initialization of coreness and N, I, O take $O(m)$ time before the update. The update time of Algorithm 5 is $O(\mathcal{CM} + |E'| + \|V^*\|_1)$ and the query time is $O(k_{max})$, where $\|S\|_c$ is the size of the c-hop neighborhood of S. This largely outperforms the baseline that has $O(\mathcal{CM} + \|E'\|_1 + \|V^*\|_2)$ update time and $O(n)$ query time ($n \gg k_{max}$, e.g., $k_{max} \leq 2208$ in our datasets that are up to billion-scale.).

Core maintenance (Line 2) costs $O(\mathcal{CM})$ time [59], Step 1 ($\texttt{upd\_e}_\text{A}$) visits each edge in E', Step 2 ($\texttt{upd\_cn}_\text{A}$) visits the 1-hop neighbors of each $v \in V^*$.

Boundness. The boundness is a measure criterion for the effectiveness of dynamic algorithms [39], [58]. A dynamic algorithm is *bounded* if its cost is of $O(f(\|\mathsf{CHANGED}\|_c))$ for some polynomial function f and some positve integer c, where CHANGED comprises the changes to both the graph and the result and $\|\mathsf{CHANGED}\|_c$ is the size of the c-hop neighborhood of CHANGED [58].

In our problem, CHANGED includes the changed edges E' and the set of vertices with coreness changed V^*. After core maintenance, the update cost of Algorithm 5 is $O(|E'| + \|V^*\|_1)$ time which is bounded in $|E'|$ and the 1-hop neighborhood of V^*.

5.3 Dynamic Algorithm for Best K-Core Set (Type-B)

The baseline is more costly for type-B metrics, as the score computation may cost $O(m^{1.5})$ time. Besides, the best k-core set for a type-B metric is harder to maintain as we need to update on triangles and triplets. We maintain the best k-core set for a type-B metric by updating Equation 2 which locates a triangle/triplet by its smallest vertex coreness.

$$TA_k = |\{\text{triangle } (u, v, w) \mid \min\{c(u), c(v), c(w)\} = k\}|$$
$$TP_k = |\{\text{triplet } (u, v, w) \mid \min\{c(u), c(v), c(w)\} = k\}| \quad (2)$$

For every k from k_{max} to 0, TA_k (resp. TP_k) equals the *increment* of `triangle` (resp. `triplet`) when Algorithm 3 processes the k-shell H_k. Then, the clustering coefficient of C_k equals $3 \times \sum_{i=k}^{k_{max}} TA_i / \sum_{i=k}^{k_{max}} TP_i$, and we can update the best k from the scores.

Algorithm 6: $\texttt{ins}^*/\texttt{rem}^*{}_\text{B}(G, c, E')$

```
1  for each (v,u) ∈ E' do upd_e_B(v,u);
2  V* ← vertices with coreness changed to c'(v);
3  for each v ∈ V* do upd_cn_B(v);
4  def upd_e_B(v,u) :        // opposite sign if rem*
5      for each w ∈ N(v) do
6          lo ← min{c(v), c(u), c(w)};
7          TP_lo ++;
8          if (v,u,w) is triangle then TA_lo ++ ;
9      for each w ∈ N(u) do run Lines 6-7;
10     G ← G with (v,u) inserted or removed;
11 def upd_cn_B(v) :
12     L ← min{c(v), c'(v)};  U ← max{c(v), c'(v)};
       /*     prune if c(u) ≤ L or c(w) ≤ L        */
13     for each u ∈ N(v) do
14         for each w ∈ N(u) \ {v} do
15             lo ← min{c(v), c(u), c(w)};
16             lo' ← min{c'(v), c(u), c(w)};
17             TP_lo −−;  TP_lo' ++;
18             if (v,u,w) is a triangle not visited then
19                 TA_lo −−;  TA_lo' ++;
20     gt_k ← Σ_{k≥U} |{u | u ∈ N(v) ∧ c(u) = k}|;
21     TP_c(v) −= C(gt_k, 2);  TP_c'(v) += C(gt_k, 2);
22     for each k from (U − 1) down to (L + 1) do // prune
23         eq_k ← |{u | u ∈ N(v) ∧ c(u) = k}|;
24         add ← eq_k × gt_k + C(nc_k, 2);
25         lo ← min{k, c(v)};  lo' ← min{k, c'(v)};
26         TP_lo −= add;  TP_lo' += add;
27         gt_k += eq_k;
28     c(v) ← c'(v);
```

Algorithm 6 shows the pseudo-code to maintain the best k-core set for a type-B metric. We can (i) *update* by edge change and coreness change, in two steps, which is facilitated by a strategy for uniquely updating triangles/triplets and an effective pruning rule to eliminate trivial triangles/triplets. OR (ii) *query* the best k-core set from TA, TP.

Step 1. Update by Edge Change (Line 1): We use $\texttt{upd\_e}_\text{B}$ to update TA_k, TP_k for inserting/removing (u, v) without considering the coreness change. Let (u, v) be an edge in E' for insertion/removal and w be a neighbor of v or u. For edge insertion, TA_{lo} and TP_{lo} will be increased, where lo is the minimum coreness of u, v and w. The edge (u, v) can form a triplet centered at v with a neighbor of v (Line 7), or a triplet centered at u with a neighbor of u (Line 9), or a triangle with a common neighbor of v and u (Line 8). Note that the opposite sign is used for edge removal.

Step 2. Update by Coreness Change (Line 3): Let v be a vertex whose coreness changes from $c(v)$ to $c'(v)$. Given a triangle (v, u, w), we will first remove the triangle with coreness $c(v)$ from TA_{lo}, and then insert the triangle back with coreness $c'(v)$ into $TA_{lo'}$, where $lo = \min\{c(v), c(u), c(w)\}$ and $lo' = \min\{c'(v), c(u), c(w)\}$. The above operation can update every triangle/triplet containing v accordingly.
Pruning by L (Lines 13-14, 22): Let L be $\min\{c(v), c'(v)\}$. Given a triangle/triplet (v, u, w), if $c(u) \leq L$ or $c(w) \leq L$, then the coreness change of v do not change TA_k and TP_k

in Equation 2, where $k = \min\{c(v), c(u), c(w)\}$. Thus, Lines 13-14 can safely skip any u or w whose coreness is not larger than L, and Line 22 can safely skip any iteration with $k \leq L$.

Update Triangles (Lines 15-19): We visit a neighbor $u \in N(v)$ and a common neighbor $w \in N(v) \cap N(u)$, which forms a triangle (v, u, w), s.t. we visit every triangle containing v. The intuition is to remove a triangle with the old corenesses and then insert it back with the new corenesses, i.e., move the triangle according to lo and lo'. Line 18 updates on each triangle which is not visited before by labeling and checking the corresponding (u, w) pair.

Update Triplets (Lines 17, 20-27): Line 17 updates any triplet (v, u, w) centered at $u \in N(v)$ while updating the triangles, located by lo and lo'. Lines 20-27 update any triplet (u, v, w) centered at v based on different $k = \min\{c(u), c(w))\}$, i.e., the minimum coreness of u and w. (i) If $k \geq U$, it implies the minimum coreness of the triplet changes from $c(v)$ to $c'(v)$. Let gt_k be the number of v's neighbors whose coreness is no less than U. There are $\binom{gt\_k}{2}$ such triplets as we can choose any pair of vertices out of the gt_k vertices. Line 21 removes $\binom{gt\_k}{2}$ triplets from $TP_{c(v)}$ and inserts them back into $TP_{c'(v)}$. (ii) If $L < k < U$, Line 24 either chooses a neighbor with coreness k and another neighbor with coreness more than k by $eq\_k \times gt\_k$; OR chooses two neighbors with coreness k by $\binom{eq\_k}{2}$. Then, Line 26 removes $eq\_k \times gt\_k + \binom{eq\_k}{2}$ such triplets from TP_{lo}, and then inserts them into $TP_{lo'}$. Note that gt_k is increased by eq_k after every round, and we can retrieve eq_k and gt_k in all iterations by one scan of the neighbor set of v. (iii) If $k \leq L$, pruned as discussed above.

Query from TA, TP. The query function $\texttt{query}_\texttt{B}$ has the same idea as $\texttt{query}_\texttt{A}$ in the last section, except it uses TA, TP rather than N, I, O. The query time is also $O(k_{max})$.

Correctness. $\texttt{upd\_e}_\texttt{B}$ is correct on updating the edge change. $\texttt{upd\_cn}_\texttt{B}$ correctly enumerates all the triangles and triplets for update (centered at v or its neighbor u). The pruning by L is correct by the definition of TA_k and TP_k. Overall, Algorithm 5 is correct.

Complexity. Before running the algorithm, we initialize coreness in $O(m)$ time and TA, TP in $O(m^{1.5})$ time. The total update time of Algorithm 6 is $O(\mathcal{CM} + \|E'\|_1 + \|V^*\|_2)$ time and the query cost is $O(k_{max})$ time. Our algorithm largely reduces the query time compared with the baseline, because $m^{1.5} \gg k_{max}$.

Step 1 ($\texttt{upd\_e}_\texttt{B}$) visit the neighbors for every edge $(u, v) \in E'$. Given a vertex $v \in V^*$, Step 2 ($\texttt{upd\_cn}_\texttt{B}$) visit the 2-hop neighbors for Line 13-19 and needs $O(d(v))$ time for Line 20-27, because $U \leq c(v) \leq d(v)$.

Boundness. After core maintenance, the update cost of Algorithm 6 is $O(\|E'\|_1 + \|V^*\|_2)$ time and is bounded in the 1-hop neighbors of E' and the 2-hop neighbors of V^*.

Algorithm 7: $\texttt{ins}^*/\texttt{rem}^*_\texttt{HA}(G, c, \mathcal{T}, E')$

```
1  for each (v,u) ∈ E' do upd_e_HA(v,u);
2  V* ← vertices with coreness changed to c'(v);
3  for each v ∈ V* do upd_cn_HA(v);
4  T ← run hierarchical core maintenance [31] and update
     the arrays accordingly;
5  def upd_e_HA(v,u) :        // opposite sign if rem*
6      assume c(v) ≤ c(u);
7      if c(v) ≠ c(u) then
8          I_v ++;   O_u ++;   O_v --;
9          I_tid(v) ++;  O_tid(u) ++;  O_tid(v) --;
10     else
11         I_v += 1/2;  I_u += 1/2;  I_tid(v) += 1/2;  I_tid(u) += 1/2;
12     G ← G with (v,u) inserted or removed;
13 def upd_cn_HA(v) :
14     for each u ∈ N(v) do
15         upd_e_HA(v,u) for removing (v,u);
16         upd_e_HA(v,u) for inserting (v,u) back where the
             coreness of v is replaced by c'(v);
17     c(v) ← c'(v);
18 def query_HA( ) :
19     num ← N; in ← I; out ← O;
20     for each tree node T[i] from bottom to top do
21         num[i] += Σ_{j∈T[i].child} num[j];
22         in[i] += Σ_{j∈T[i].child} in[j];
23         out[i] += Σ_{j∈T[i].child} out[j];
24         Q(S_i) ← compute_metric(num[i], in[i], out[i]);
25     return S* with the highest Q(S*);
```

6 Maintaining the Best K-Core

6.1 A Baseline with Vertex Ordering Maintenance

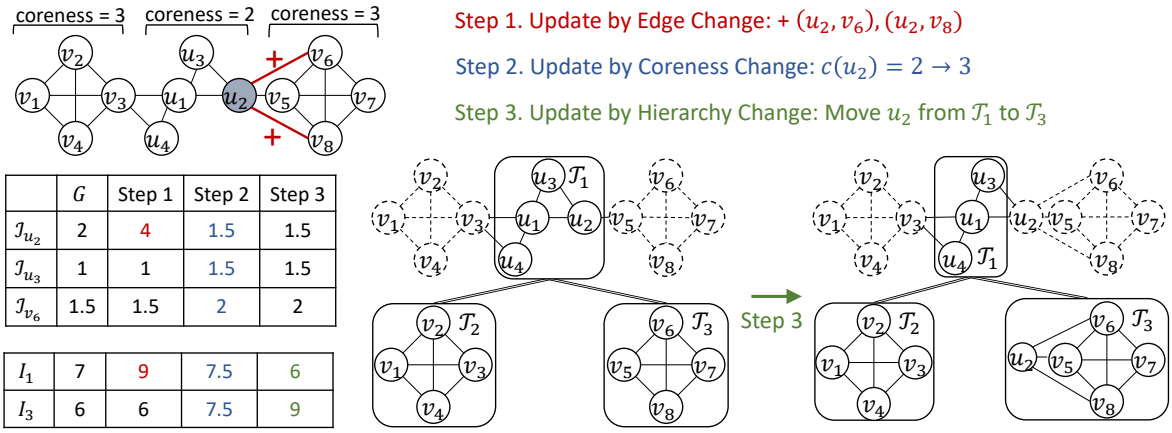
Given the index (vertex ordering and the core forest), our static score computation to find the best k-core is efficient, e.g., it costs $O(n)$ time on a type-A metric. Thus, our baseline is to efficiently maintain the index and then update the best k-core, similar to the intuition of Section 5.1. Specifically, we maintain the coreness and forest structure using hierarchical core maintenance [31], maintain vertex ordering using Section 5.1, and compute the best k-core by Algorithm 4 in Section 4 with the updated index.

Let $\mathcal{HCM}$ and $\mathcal{VOM}$ be the cost of hierarchical core maintenance [31] and vertex ordering maintenance in Section 5.1, respectively. Given an edge set E', the solution needs $O(\mathcal{HCM} + \mathcal{VOM})$ time to update the index. Then, we can use Algorithm 4 to query the best k-core from the index in $O(n)$ or $O(m^{1.5})$ time respectively.

6.2 Dynamic Algorithm for Best Single K-Core (Type-A)

The update of the best k-core is more challenging than that of the best k-core set, as the connectivity among k-cores is complex and the hierarchy of k-cores (i.e, core forest) may change significantly for a slight graph update [31]. To avoid recomputing the score of every k-core, we use Equation 3 to formalize both the primary value increment PV_i (including N_i, I_i and O_i) of a tree node $\mathcal{T}[i]$ in the hierarchy and the increment $\mathcal{PV}_v$ (including $\mathcal{I}_v$ and $\mathcal{O}_v$) contributed by a vertex $v \in \mathcal{T}[i]$. Hence, we can quickly move a vertex in the hierarchy with $\mathcal{PV}_v$ and update the best k-core by PV_i.

$$
\begin{aligned}
I_i &= \sum_{v \in \mathcal{T}[i].\mathrm{V}} \mathcal{I}_v \text{ , where } \mathcal{I}_v = |N(v,>)| + \frac{1}{2}|N(v,=)| \\
O_i &= \sum_{v \in \mathcal{T}[i].\mathrm{V}} \mathcal{O}_v \text{ , where } \mathcal{O}_v = |N(v,>)| - |N(v,<)| \\
N_i &= |\mathcal{T}[i].\mathrm{V}| \text{ is maintained in } \mathcal{T}
\end{aligned}
\tag{3}
$$



	G	Step 1	Step 2	Step 3
$\mathcal{I}_{u_2}$	2	4	1.5	1.5
$\mathcal{I}_{u_3}$	1	1	1.5	1.5
$\mathcal{I}_{v_6}$	1.5	1.5	2	2

I_1	7	9	7.5	6
I_3	6	6	7.5	9

Fig. 6. An Example of ins*/rem*$_{\text{HA}}$ (Algorithm 7)

Let $\mathcal{T}[i]$ be a tree node in the hierarchy $\mathcal{T}$ of k-cores (defined in Section 4.1), $\mathcal{T}[i].\texttt{V}$ be the set of the vertices contained in $\mathcal{T}[i]$, and $\mathcal{T}[i].\texttt{child}$ be the children tree node. For every tree node $\mathcal{T}[i]$ from bottom to top of $\mathcal{T}$, N_i (resp. I_i, O_i) in Equation 3 equals the *increment* of the number of vertices (resp. edges and boundary edges), i.e., $num[i]$ (resp. $in[i], out[i]$), when Algorithm 4 processes the tree node $\mathcal{T}[i]$. Given a vertex $v \in \mathcal{T}[i]$, $\mathcal{I}_v$ (resp. $\mathcal{O}_v$) stores the contribution of v in I_i (resp. O_i). We can get the primary value of every k-core by a bottom-up summation on the hierarchy, and compute the score of every k-core from the primary values.

Algorithm 7 maintains the best k-core based on Equation 3. $N, I, O, \mathcal{I}, \mathcal{O}$ are either maintained in the graph or initialized by Equation 3. Then, we can (i) *update*: Lines 1-4 update by edge, coreness, and hierarchy change, in three steps. OR (ii) *query*: Lines 19-25 ($\texttt{query}_{\text{HA}}$) query the best k-core.

Step 1. Update by Edge Change (Line 1): The update follows Algorithm 5 except for two differences. First, Line 11 must explicitly maintain the vertex contribution $\frac{1}{2}$ when $c(v) = c(u)$, as the granularity of counting is on vertex level unlike that on k-core sets (Algorithm 5). Second, we update $\mathcal{I}_v, \mathcal{O}_v$ and $I_{tid(v)}, O_{tid(v)}$ simultaneously for each vertex v, where $tid(v)$ is the tree node containing v. We use the opposite sign for edge removal.

Step 2. Update by Coreness Change (Line 3): For a vertex $v \in V^*$, we remove every incident edge with $c(v)$ and then insert back it with $c'(v)$. We do not process N because it is already maintained in $\mathcal{T}$.

Step 3. Update by Hierarchy Change (Line 4): The hierarchy structure of k-cores is updated by hierarchical core maintenance (HCM) [31]. Meanwhile, we can update all the changes in HCM by PV_i (including N_i, I_i and O_i) and $\mathcal{PV}_v$ (including $\mathcal{I}_v$ and $\mathcal{O}_v$), where PV_i and $\mathcal{PV}_v$ contain the *increments* of the primary values of the tree node $\mathcal{T}[i]$ and the vertex $v \in \mathcal{T}[i]$, respectively. Our update only needs constant time for every hierarchy change (move of a vertex).

- create a tree node $\mathcal{T}[i]$: $PV_i \leftarrow 0$.
- parent change of $\mathcal{T}[i]$: PV_i stays the same.
- move a vertex v from $\mathcal{T}[i]$ to $\mathcal{T}[j]$: run PV_i $-=$ $\mathcal{PV}_v$ and PV_j $+=$ $\mathcal{PV}_v$ to move vertex contribution.
- merge tree nodes: move the corresponding vertices.
- split a tree node: create tree node(s) and move vertices.

Query from N, I, O (Lines 19-25). In $\texttt{query}_{\text{HA}}$, we sum up the primary values from bottom to top of $\mathcal{T}$, update the score of every k-core, and find the best k-core. The time cost is $O(|\mathcal{T}|)$ time which is also usually minor, e.g., $|\mathcal{T}| \leq 4087$ in our datasets that are up to billion-scale.

Example 8. Figure 6 shows the change of primary values and core forest after edge insertion, where $\mathcal{T}_2$ is neglected because it is not changed. Initially, u_2 has two neighbors u_1, u_3 with the same coreness and one neighbor v_5 with a larger coreness, i.e., $\mathcal{I}_{u_2} = \frac{1}{2} \times 2 + 1 = 2$. Step 1 inserts edges $(u_2, v_6), (u_2, v_8)$, and we have $\mathcal{I}_{u_2}$ $+=$ 2 because u_2 has a lower coreness in both edges. We also update by I_1 $+=$ 2 as $u_2 \in \mathcal{T}[1]$. Step 2 updates the coreness of u_2 from 2 to 3. Line 15 remove all incident edges of u_2, where u_2 has a lower coreness than v_5, v_6, v_8 and the same coreness to u_1, u_3, i.e., $\mathcal{I}_{u_2}$ $-=$ 4 $(3 + 2 \times \frac{1}{2})$. Line 16 inserts back the edges with $c'(u_2) = 3$, where u_2 has two neighbors u_1, u_3 with a lower coreness and three neighbors v_5, v_6, v_8 with the same coreness, i.e., $\mathcal{I}_{u_2}$ $+=$ 1.5 $(3 \times \frac{1}{2})$. Step 2 also updates I_1 (resp. I_3) when updating a vertex contained in $\mathcal{T}[1]$ (resp. $\mathcal{T}[3]$). Finally, Step 3 updates the hierarchy, by moving u_2 from $\mathcal{T}_1$ to $\mathcal{T}_3$, i.e., I_1 $-=$ $\mathcal{I}_{u_2}$ and I_3 $+=$ $\mathcal{I}_{u_2}$.

Correctness. The update in $\texttt{upd\_e}_{\text{HA}}$ and $\texttt{upd\_cn}_{\text{HA}}$ is correct as analyzed in Algorithm 5. The correctness of the update on hierarchy change is immediate as we follow the definition of core forest in Section 4.1.

Complexity. Algorithm 7 can update an edge set E' in $O(\mathcal{HCM} + |E'| + \|V^*\|_1)$ time, and each query takes $O(|\mathcal{T}|)$ time. The baseline has $O(\mathcal{HCM} + \|E'\|_1 + \|V^*\|_2)$ update time and $O(n)$ query time. Our algorithm largely reduces the update cost and the query time ($n \gg |\mathcal{T}|$, e.g., $|\mathcal{T}| \leq 4087$ in our datasets that are up to billion-scale). Step 1 needs $O(|E'|)$ time, Step 2 iterates over the neighbors of V^* in $O(\|V^*\|_1)$ time, Step 3 needs $O(\mathcal{HCM})$ time.

Boundness. After HCM, the update cost is $O(|E'| + \|V^*\|_1)$ time, which is bounded in the size of E' and the 1-hop neighbors of V^*. Step 3 can update any hierarchy change during HCM [31] in constant time and can be ignored.

Algorithm 8: $\texttt{ins}^*/\texttt{rem}^*_{\text{HB}}(G, c, \mathcal{T}, E')$

```
1  for each (v,u) ∈ E' do upd_e_HB(v,u);
2  V* ← vertices with coreness changed to c'(v);
3  for each v ∈ V* do upd_cn_HB(v);
4  T ← run hierarchical core maintenance [31] and update
     the arrays accordingly;
5  def get_lov(v,u,w) :
6      return the endpoint with the lowest vertex rank;
7  def upd_e_HB(v,u) :        // opposite sign if rem*
8      for each w ∈ N(v) do
9          lov ← get_lov(v,u,w);
10         TP_{tid(lov)} ++; TA_{tid(lov)} ++;
11         if (v,u,w) is triangle then
12             TA_{lov} ++; TA_{tid(lov)} ++;
13     for each w ∈ N(u) do run Lines 9-10;
14     G ← G with (v,u) inserted or removed;
15 def upd_cn_HB(v) :
16     L ← min{c(v), c'(v)};
       /*      prune if c(u) < L or c(u) < L      */
17     for each u ∈ N(v) do
18         for each w ∈ N(u) \ {v} do
19             lov ← get_lov(v,u,w);
20             lov' ← get_lov(v,u,w) where the coreness
                 of v is replaced by c'(v);
21             TP_{lov} --; TP_{tid(lov)} --;
22             TP_{lov'} ++; TP_{tid(lov')} ++;
23             if (v,u,w) is a triangle not visited then
24                 TA_{lov} --; TA_{tid(lov)} --;
25                 TA_{lov'} ++; TA_{tid(lov')} ++;
26     i ← 0;
27     for each u ∈ N(v) in descending vertex rank do
28         lov, lov' ← run Line 19-20 with w omitted;
29         TP_{lov} -= i; TP_{tid(lov)} -= i;
30         TP_{lov'} += i; TP_{tid(lov')} += i;
31         i ++;
32     c(v) ← c'(v);
```

6.3 Dynamic Algorithm for Best Single K-Core (Type-B)

For type-B metrics, we should compute the vertex contribution regarding triangles and triplets s.t. the best k-core can be efficiently maintained. Thus, we propose Equation 4 to assign a triangle/triplet to its endpoint with the lowest vertex rank (defined in Section 3.2), i.e., the vertex with the lowest id among the endpoints having the smallest coreness.

$$\begin{aligned} \mathcal{TA}_v &= |\{\text{triangle } (u,v,w) \mid v \text{ has the lowest rank}\}| \\ \mathcal{TP}_v &= |\{\text{triplet } (u,v,w) \mid v \text{ has the lowest rank}\}| \\ TA_i &= \sum_{v \in \mathcal{T}[i].\text{V}} \mathcal{TA}_v \ , \ TP_i = \sum_{v \in \mathcal{T}[i].\text{V}} \mathcal{TP}_v \end{aligned} \tag{4}$$

For every tree node $\mathcal{T}[i]$ from bottom to up, TA_i (resp. TP_i) represents the *increment* of the number triangles (resp. triplets) by $\mathcal{T}[i]$ and $\mathcal{TA}_v$ (resp. $\mathcal{TP}_v$) represents the increment by a vertex $v \in \mathcal{T}[i]$.

Algorithm 8 maintains the best k-core based on Equation 4. $TA, TP, \mathcal{TA}, \mathcal{TP}$ are either maintained in the graph or initialized by Equation 4. Then, we can (i) *update* by edge, coreness, and hierarchy change, in three steps. OR (ii) *query* the best k-core.

Step 1. Update by Edge Change (Line 1): The idea is similar to Algorithm 6, and the differences are (i) vertex rank is used and (ii) we update $TA, \mathcal{TA}$ (resp. $TP, \mathcal{TP}$) simultaneously, where lov is the endpoint with the lowest vertex rank. The opposite sign is used for removals.

Step 2. Update by Coreness Change (Lines 2-3): Let $v \in V^*$ be a vertex whose coreness changes from $c(v)$ to $c'(v)$. Let lov (resp. lov') be the endpoint with the lowest vertex rank, when the coreness of v is $c(v)$ (resp. $c'(v)$). Given a triangle/triplet, we first remove it from $\mathcal{TA}_{lov}$ and $TA_{tid(lov)}$, then insert it back into $\mathcal{TA}_{lov'}$ and $TA_{tid(lov')}$. The above operation can update every triangle/triplet containing v.

Pruning by L (Lines 17-18): The logic is like the pruning in Algorithm 6, except the lower limit L is open as two vertices with the same coreness may have different vertex ranks.

Update Triangles (Lines 23-25): Lines 23-25 visit a neighbor $u \in N(v)$ and a common neighbor $w \in N(v) \cap N(u)$, to update the numbers of triangles containing v by lov, lov'.

Update Triplets (Lines 21-22, 26-31): Lines 21-22 update the triplets centered at $u \in N(v)$ during the update of triangles. Lines 26-31 update every triplet (u,v,w) centered at v. Line 27 visits every neighbor of v in descending order of vertex rank, where we use bucket sort for ordering. In each round, u has i neighbors of v with larger vertex rank (i.e., w), and we update all such triplets (u,v,w) using lov, lov'.

Step 3. Update by Hierarchy Change (Line 4): The update is same to Step 3 of Section 6.2, except that we replace PV_i and $\mathcal{PV}_v$ by TA_i, TP_i and $\mathcal{TA}_v, \mathcal{TP}_v$, respectively.

Query from TA, TP. The idea of $\texttt{query}_{\text{HB}}$ is similar to $\texttt{query}_{\text{HA}}$ in Section 6.2. We do not maintain the scores due to the same reason in Section 6.2.

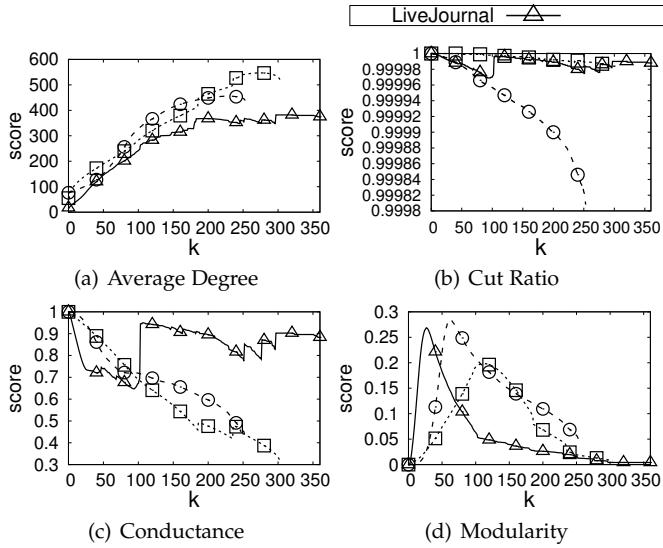
Correctness. $\texttt{upd\_e}_{\text{HB}}$ is correct on updating the edge change. For a vertex $v \in V^*$ whose coreness is changed, $\texttt{upd\_cn}_{\text{HB}}$ correctly updates every triangle and triplet (centered at v or its neighbor u) containing v, according to our analysis above. The update by hierarchy change is correct as we follow Step 3 of Algorithm 5. Overall, Algorithm 8 can correctly update the best k-core for a type-B metric.

Complexity. Before the update, we initialize the coreness, the core forest $\mathcal{T}$, and arrays in $O(m^{1.5})$ time. Algorithm 8 can update an edge set E' in $O(\mathcal{HCM} + \|E'\|_1 + \|V^*\|_2)$ time, and each query takes $O(|\mathcal{T}|)$ time. Lines 26-31 run in $O(d(v))$ time by bin sort. The baseline has $O(\mathcal{HCM} + \|E'\|_1 + \|V^*\|_2)$ update time and $O(m^{1.5})$ query time. Our algorithm largely reduces the query time ($m^{1.5} \gg |\mathcal{T}|$).

Boundness. After HCM, the update cost of Algorithm 8 is $O(\|E'\|_1 + \|V^*\|_2)$ time, which bounded in the 1-hop neighborhood of E' and the 2-hop neighborhood of V^*. Step 3 updates any change in HCM instantly and can be ignored.

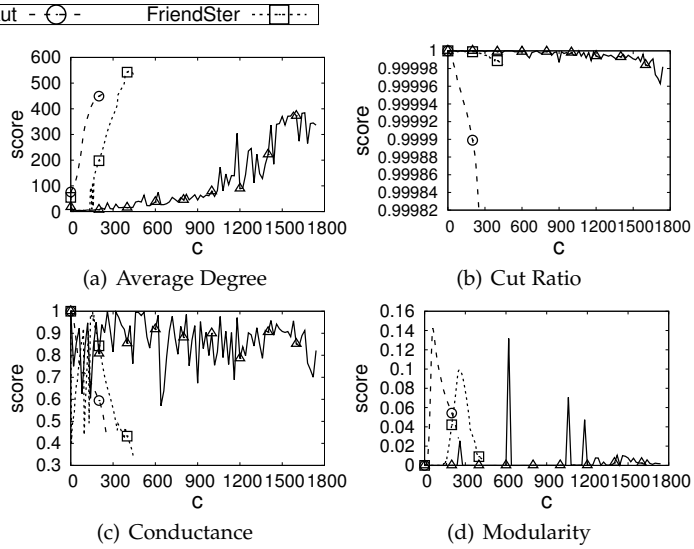
7 Experimental Evaluation

Datasets. We use 10 public real-world networks from different areas including collaboration networks, Internet topology, brain networks, and social networks. Hollywood and Human-Jung are from http://networkrepository.com, and the others are from http://snap.stanford.edu. The details of the data are shown in Table 3, ordered by the number of edges, where d_{avg} is the average degree, k_{max} is the largest vertex coreness, and $|\mathcal{T}|$ is the size of the core forest.



(a) Average Degree (b) Cut Ratio (c) Conductance (d) Modularity

Fig. 7. Scores of Every k-Core Set



(a) Average Degree (b) Cut Ratio (c) Conductance (d) Modularity

Fig. 8. Score of Every k-Core

TABLE 3
Statistics of Datasets

Dataset	n	m	d_{avg}	k_{max}	$\|\mathcal{T}\|$
Astro-**P**h	18,772	198,110	21.1	56	376
Gowalla	196,591	950,327	9.7	51	74
DBLP	317,080	1,049,866	6.6	113	766
Youtube	1,134,890	2,987,624	5.3	51	139
As-**S**kitter	1,696,415	11,095,298	13.1	111	902
Live**J**ournal	3,997,962	34,681,189	17.4	360	1755
Hollywood	1,069,126	56,306,653	105.3	2208	678
Orkut	3,072,441	117,185,083	76.3	253	253
Human-**J**ung	784,262	267,844,669	683.1	1200	4087
Friend**S**ter	65,608,366	1,806,067,135	55.1	304	450

TABLE 4
Summary of Algorithms

Notation	Description
`static`	our static baseline (see Sections 3.1 and 4.2)
`static*`	our static algorithm (see Algorithms 2-4)
`ins/rem`	our dynamic baseline (see Sections 5.1 and 6.1)
`ins*/rem*`	our dynamic algorithm (see Algorithms 5-8)

Metrics. We use the six representative community scoring metrics in Section 2.3, i.e., average degree, density, cut ratio, conductance, modularity, and clustering coefficient.

Algorithms. Table 4 presents the algorithms we evaluate, where the static baseline is state-of-the-art in finding the best k-core set (and the best k-core) before proposing our improved algorithms. Given a metric Q mentioned above, each static (resp. dynamic) algorithm can compute (resp. update) the best k-core set and the best k-core regarding Q. For clarity, we rename the static algorithms in the present work, i.e., the naming of them is different from [16].

Environment. We perform experiments on a CentOS Linux server (Release 7.5.1804) with a Quad-Core Intel Xeon CPU (E5-2630 v4 @ 2.20GHz) and 128G memory. All algorithms are implemented in C++ and are compiled with g++ 7.3.0 under O3 optimization.

7.1 Community Quality by Different k

Scores of k-Core Sets. Figure 7 shows the scores of the k-core set for different k, regarding different community metrics. According to the trends of score on different k, trivial values for k will often find low-quality k-core sets. A user-defined k value (e.g., 100) is not promising. Besides, a value based on dataset statistics (e.g., average degree or k_{max}) does not necessarily produce a high-quality k-core set. Very small k values are often preferred for the metric of cut ratio or conductance, as a small k can minimize the inter-connections among different k-cores. Thus, using these metrics solely may lead to trivial k values, and it is better to combine them with other metrics to decide a k value. Our paper aims to consider as many metrics as possible, but these metrics are not equally useful in practice.

Scores of Single k-Cores. Figure 8 shows the scores of all the k-cores, where c in the x axis is the sequence id of a k-core in the order that ranks all k-cores by ascending values of k with ties broken by ascending scores of the k-cores. The y-axis is the community score of a k-core. The figure presents a finer granularity of score distribution in core decomposition on different metrics. To clearly show the result, the trends are depicted by the average score of every 20 (resp. 5) consecutive k-cores in LiveJournal (resp. Orkut and Friendster). The trends of score imply that many high-score k-cores come from relatively low-score k-core sets. Similar to finding the best k-core set, considering only the cross-connections of k-cores (e.g., cut ratio and conductance) may not be effective since extremely small k values are preferred by these metrics. We may consider using a combination of different metrics to find the high-quality k-cores.

Best k Values. Table 5 shows the best k identified by finding the best k-core set and the best single k-core. The name of the result is formed by two parts. The first part uses `KCS-` (resp. `KC-`) to represent the result of the best k-core set (resp. the best k-core). The second part describes the used community scoring metric (Section 2.3), i.e., average degree, density, cut ratio, conductance, modularity, and clustering coefficient. For conciseness, we abbreviate them to `ad`, `den`, `cr`, `con`, `mod`, and `cc`, respectively. For example, `KCS-ad` is the best k-core set by average degree, and `KC-cc` is the best k-core by clustering coefficient. When multiple k values provide the highest score, we record the largest k value.

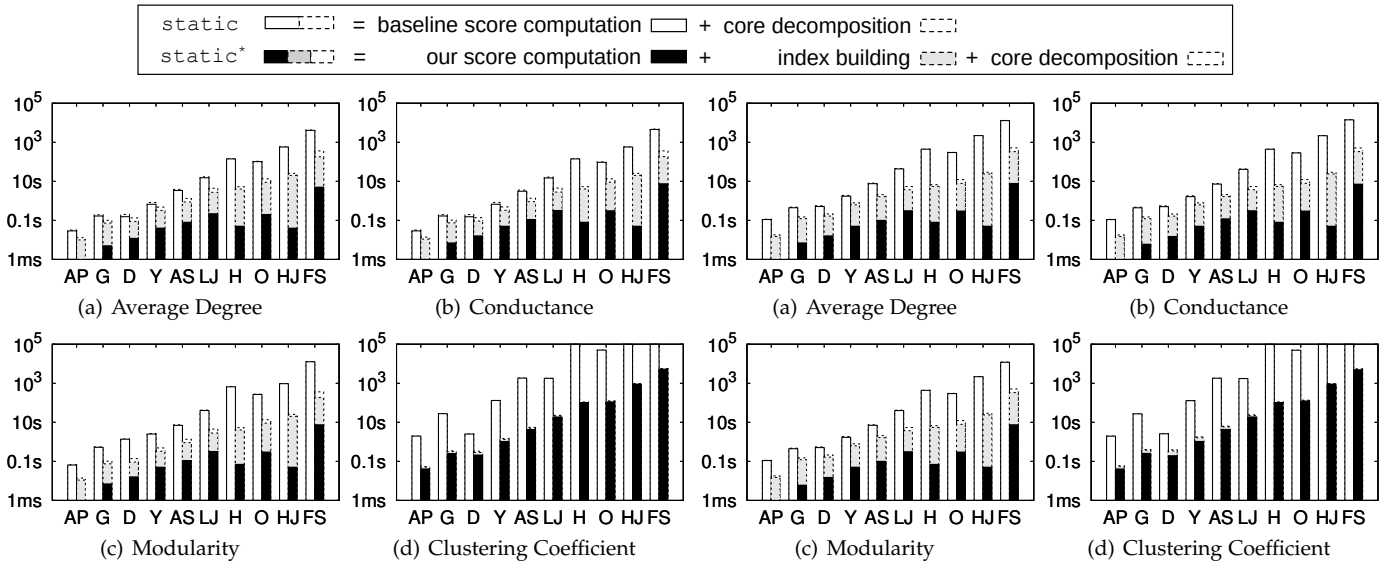


Fig. 9. Time of Finding the Best k-Core Set (Static)

Fig. 10. Time of Finding the Best Single k-Core (Static)

TABLE 5
Best k for the k-Core (Set)

Result	AP	G	D	Y	AS	LJ	H	O	HJ	FS
KCS-ad	36	45	113	47	97	320	2208	229	1059	274
KCS-den	56	51	113	51	111	360	2208	253	1200	304
KCS-cr	1	1	1	1	1	3	1	3	1	44
KCS-con	1	1	1	1	1	1	1	1	1	1
KCS-mod	26	13	7	6	14	27	303	62	675	112
KCS-cc	56	51	113	51	111	360	2208	253	1200	304
KC-ad	36	45	64	47	90	194	2208	229	1058	274
KC-den	56	8	113	5	3	89	2208	253	11	9
KC-cr	17	1	1	1	2	1	1	1	5	1
KC-con	17	1	1	1	2	1	1	1	5	61
KC-mod	26	13	7	6	14	27	303	62	675	66
KC-cc	56	8	113	5	3	89	2208	253	11	9

TABLE 6
Community A ($k = 17$)

Vijay Gadepally	William Arcand	Andrew Prout
Charles Yee	Michael Jones	Albert Reuther
Anna Klein	Matthew Hubbell	Jeremy Kepner
Bill Bergeron	David Bestor	Julie Mullen
Antonio Rosa	Chansup Byun	Peter Michaleas
Lauren Milechin	Siddharth Samsi	Michael Houle

TABLE 7
New Members of Community A (2019-2024)

Guillermo Morales	Chad R. Meiners	Doug Stetson
Hayden Jananthan	Timothy Davis	Sandeep Pisharody

TABLE 8
Community B ($k = 9$)

Danyang Zhao	Xianyi Wang	Congliang Liu
Cheng Liu	Yueqiang Sun	Qifei Du
Dongwei Wang	Yuerong Cai	Chunjun Wu
Weihua Bai	Junming Xia	Xiangguang Meng

Different community metrics have different preferences on the best value of k. For k-core set, `ad`, `den` and `cc` prefer large values of k for high cohesiveness, while `cr` and `con` give high scores when k is small for sparse cross-connection. Some metrics choose an extreme value of k, which implies that a combination of these metrics should be used. The `mod` metric chooses moderate values of k considering both the inner and outer connection of the k-cores.

7.2 Case Study on High Score k-Cores

Using different community scoring metrics, our algorithms can find different high-quality k-cores in the DBLP dataset.

Table 6 shows community A (a 17-core) in which the members are mainly from Lincoln Laboratory Supercomputing Center, MIT. This community is identified as the best by the score of average degree, internal density, and clustering coefficient, respectively. Community A are highly collaborative in research: they have 6 co-authored papers by all members, 15 co-authored papers by at least 17 members, and 20 co-authored papers by at least 16 members.

Table 8 shows community B (a 9-core) in which the members are from the National Space Science Center, Chinese Academy of Sciences. Community B is relatively isolated from other authors, according to its highest cut ratio and conductance. The members in Community B are also tightly connected: there are 8 co-authored papers by all members and 13 co-authored papers by at least 11 members.

7.3 Case Study on Dynamics of High Score k-Cores

Our proposed algorithms can track the dynamics of high-quality k-cores in the DBLP network. For example, community A (Table 6 in Section 7.2) in the DBLP network is found in our conference paper [16], while DBLP has undergone significant evolution during these years. Recall that community A was formed by scholars from MIT Lincoln Laboratory Super-computing Center in 2019. In this case study, Table 7 presents 6 new members of this community that our method detects in 2024, according to the same metric in Section 7.2. Most new members are either new staff at MIT Lincoln Laboratory, or existing staff who moved from other groups into the Cyber Operations and Analysis Technology Group. Dr. Timothy Davis is a professor at Texas A&M who collaborates with MIT Lincoln Laboratory and is the only member outside MIT. Our algorithm takes no more than 1ms per edge to update community A.

7.4 Runtime of Static Algorithms

Finding the Best k Value. In Figure 9, `static` is the baseline algorithm proposed in Section 3.1 which contains

score computation and core decomposition [5]. `static*` is the improved algorithm (Algorithm 2 or 3) which contains score computation, core decomposition, and index building, i.e., the vertex ordering proposed in Section 3.2.

Figure 9 shows that `static*` is significantly faster than `static` on all the datasets. For each dataset, index building and core decomposition only need to be executed one time, while score computation can be run for many times given the different community metrics. The margins become larger (1-4 orders of magnitude) if we do not count index building time in `static*`. On Hollywood, Human-Jung, and FriendSter, the baseline cannot finish within 10^5 seconds for clustering coefficient. As analyzed in Section 3.3, the time complexity of score computation is determined by the number of vertices for 5 metrics, which exactly matches the trends in Figure 9. The runtime on density and cut ratio is almost the same as that on average degree.

Finding the Best Single k-Core. In Figure 10, `static` is the baseline algorithm proposed in Section 4.2, `static*` is the improved algorithm (Algorithm 4.3) where the index contains core decomposition, the vertex ordering in Section 3.2, and the forest construction in Section 4.1.

Figure 9 shows that `static*` still largely outperforms `static`. The trends of runtime are very similar to Figure 9, except the runtime of the algorithms is larger due to the computation of k-core connectivity. On Hollywood, Human-Jung, and FriendSter, the baseline cannot finish in 10^5 seconds for clustering coefficient. The runtime on density and cut ratio is almost the same as that on average degree. Overall, `static*` outperforms `static` by 1-4 orders of magnitude in runtime on different datasets.

7.5 Runtime of Dynamic Algorithms

When testing dynamic algorithms, we randomly remove x edges from the graph and then insert them back. We repeat each experiment three times and report the average result.

Maintaining the Best k Value (1 Edge). Figure 11 reports the runtime of maintaining the best k-core set. The runtime includes update and query time. Our improved solution significantly outperforms the baseline, achieving up to 4 orders of magnitude faster runtime for type-A metrics and 6 orders for type-B metrics.

Maintaining the Best Single k-Core (1 Edge). Figure 12 shows the runtime (update+query) of maintaining the best k-core. For type-A metric, our solution is up to 4 orders faster than the baseline. However, the total runtime of both algorithms is relatively close, as the dominant cost is from HCM. For the type-B metric, our solution is 3 orders faster compared with the baseline. The edge removal of our algorithm has a greater time cost than the insertion, as HCM incurs a higher cost of removal.

Scalability for Maintenance. Figure 13-14 report the running time of our solution, varying the number of edges to insert/remove x from 1 to 1024. Our improved algorithms scale almost linearly on all the datasets.

Figure 13 reports the scalability of maintaining the best k-core set. The runtime of our solution is less than 100ms for the type-A metric and 15s for the type-B metric, respectively. The time cost on the Holly dataset increases faster than the others, because Holly contains a 2209-clique that is likely to create a large V^* with coreness changed, leading to a larger cost. The cost of edge insertion is close to edge removal.

Figure 14 reports the scalability of maintaining the best k-core. The runtime has a close trend to the cost of HCM as it is dominant in total cost. The time cost is higher than that in Figure 13, as HCM is more costly than core maintenance. The removal is slower than the insertion because HCM has a higher cost on edge removal.

TABLE 9
Performance of `static*-D` on Densest Subgraph & Maximum Clique

Dataset	`CoreApp`		`static*-D`		`static*-D` (output S^*)	
	d_{avg}	time (s)	d_{avg}	time (s)	$MC \subseteq S^*$	$\lvert S^*\rvert/n$
AP	56.035	0.11	**58.923**	**0.02**	✓	7.87%
G	76	0.195	**87.593**	**0.093**		0.28%
D	113	0.233	**113.13**	**0.138**	✓	0.04%
Y	86.066	0.594	**91.1**	**0.498**	✓	0.18%
AS	150.018	**1.145**	**178.801**	1.374		0.03%
LJ	374.71	4.943	**387.027**	**4.832**	✓	0.01%
H	**2208**	**3.002**	**2208**	3.635	✓	0.21%
O	438.64	20.14	**455.732**	**11.72**		0.85%
HJ	2013.879	15.272	**2114.915**	**14.457**	✓	1.15%
FS	513.852	1041.528	**547.035**	**836.279**		0.08%

S^* is the output of `static*-D`. d_{avg} is the average degree of output. $MC \subseteq S^*$ means the maximum clique is contained in S^*. $\lvert S^*\rvert/n$ is the vertex proportion of S^* in the whole graph.

TABLE 10
Performance of `static*-SC` on Size-constrained k-core (DBLP)

$c(v)$	$k=10$	$k=15$	$k=20$	$k=30$	$k=40$
30	96.45%	88.31%	76.21%	20.97%	/
43	97.46%	91.41%	82.10%	37.87%	6.69%
51	99.12%	96.75%	92.64%	49.81%	30.77%
64	98.61%	95.15%	89.62%	61.88%	57.86%
113	98.61%	95.15%	89.62%	61.88%	57.86%

Given a random node v with coreness $c(v)$, the percentage that `static*-SC` returns a k-core with at most 5% size deviation to h.

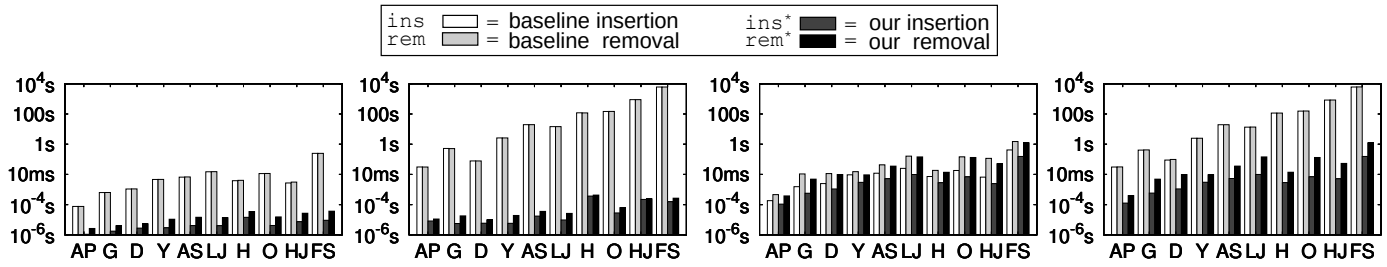
7.6 Applications on Other Problems

For k-core-related problems, our algorithms can serve as better approximate solutions compared with the state-of-the-art, or produce useful intermediate results.

Densest Subgraph. The densest subgraph (DS) problem is to find the subgraph with the largest average vertex degree [23]. Let `static*-D` be our static algorithm that returns the best k-core regarding average degree (Algorithm 4). Our `static*-D` produces a $\frac{1}{2}$-approximate solution for DS problem, because the k_{max}-core is one of our candidate results, which is a $\frac{1}{2}$-approximate solution [23]. We compare our algorithm with the recent approximate solution named `CoreApp` [23] which is the fastest algorithm for DS problem. As shown in Table 9, `static*-D` outperforms `CoreApp` in both output quality (average degree) and runtime on most datasets. The better values in Table 9 are marked in bold.

Maximum Clique. The maximum clique (MC) problem is to find the largest subset of vertices such that every pair of vertices in the subset is adjacent [11]. As shown in Table 9, it is likely that S^* contains the maximum clique (6 out of 10 datasets), although S^* is not large (the proportion of S^* in the whole vertex set is often within 1%). This finding can benefit the algorithm design for MC problem.

Size-Constrained k-Core. Given an integer k, an integer h, and a vertex v, the problem of size-constrained k-core (SCK) is to find a k-core of size h which contains v. We adapt

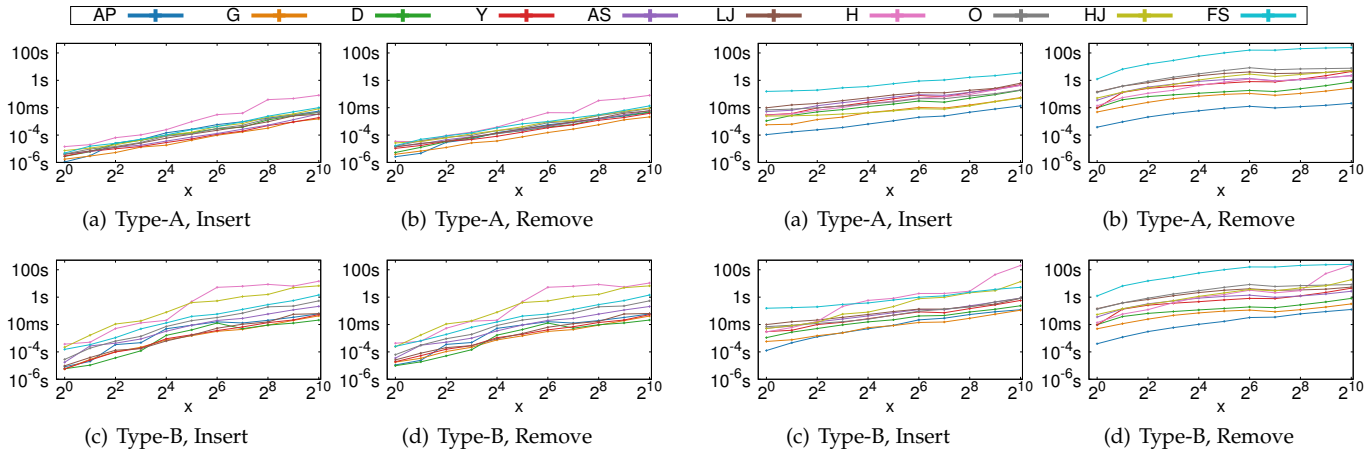


(a) Type-A, 1 edge (b) Type-B, 1 edge

Fig. 11. Time of Maintaining the Best k-Core Set (Dynamic)

(a) Type-A, 1 edge (b) Type-B, 1 edge

Fig. 12. Time of Maintaining the Best k-Core (Dynamic)



(a) Type-A, Insert (b) Type-A, Remove

(c) Type-B, Insert (d) Type-B, Remove

Fig. 13. Time of Inserting/Removing x Edges (k-Core Set)

(a) Type-A, Insert (b) Type-A, Remove

(c) Type-B, Insert (d) Type-B, Remove

Fig. 14. Time of Inserting/Removing x Edges (Single k-Core)

TABLE 11
Performance of ins*-D/rem*-D on Dynamic Densest Subgraph

Dataset	Sawlani and Wang [42] ($\epsilon = 0.5$)			Our ins*-D/rem*-D				
	0-5% graph		full graph	0-5% graph		full graph		
	ins 1k	rem 1k	space	ins 1k	rem 1k	space	ins 1k	rem 1k
AP	18.8	18.9	124GB	0.03	0.05	8.5MB	0.01	0.02
G	41.1	41.3	OOM	0.67	0.26	43.9MB	0.09	0.35
D	51.0	51.1	OOM	1.64	0.18	64.7MB	0.29	0.93
Y	92.6	93.8	OOM	5.28	1.26	224MB	0.77	5.78
AS	107.8	108.3	OOM	1.14	3.86	374MB	0.66	2.87
LJ	148.7	150.5	OOM	1.63	4.34	946MB	0.94	5.93
H	82.6	83.2	OOM	0.55	1.52	620MB	1.63	3.12
O	132.9	134.6	OOM	0.58	3.78	1.42GB	0.26	8.17
HJ	77.2	79.2	OOM	0.15	0.85	2.23GB	0.06	6.73
FS	195.7	196.5	OOM	7.95	95.4	25.2GB	4.90	354.09

'OOM' is Out Of Memory. 'ins 1k' (resp. 'rem 1k') denotes the running time of inserting (resp. removing) 1000 edges in seconds.

our static algorithm to the SCK problem with the following method called static*-SC. It first selects the k'-core S' with the highest average degree by Algorithm 4, where (i) $k' \geq k$; (ii) S' contains v; and (iii) $|V(S')| \geq h$. Then, static*-SC peels S' until $|V(S')| \leq h$ to find more results (k-cores): in each step, it removes the vertex with the lowest degree (skip v) and any vertex with degree less than k in S'.

Table 10 shows the performance of static*-SC on DBLP. We say static*-SC hits a query if the returned k-core contains v and has at most 5% size deviation to h. static*-SC hits the query with high probability when $c(v)$ is larger than k. Note that some coreness values may have no vertex with such coreness. As static*-SC runs in linear time, it can benefit the algorithm design for SCK problem.

Dynamic Densest Subgraph. The dynamic densest subgraph (DDS) problem is to update the subgraph with the largest average vertex degree on dynamic graphs [42]. Let ins*-D/rem*-D be our dynamic algorithm that maintains the best k-core regarding average degree (Algorithm 7). Our ins*-D/rem*-D also maintains a $\frac{1}{2}$-approximate DS, as it has the same output as the static algorithm. We compare our algorithm to the Sawlani and Wang's algorithm [42], the SOTA for dynamic $(1-\epsilon)$-approximate DS (we set $\epsilon = 0.5$).

Table 11 summarizes the performance for the dynamic densest subgraph. The algorithms are run on either a small subset of the graph (5% or less if out-of-memory, i.e., OOM) or the full graph. When inserting/removing 1000 edges, our proposed solution outperforms the SW algorithm by up to orders and is up to 3 orders faster than re-running static*-D (Table 9). Importantly, the SW algorithm faces an OOM error when running on the full graph for all datasets, except the smallest one. The reason is that the SW algorithm must insert each edge individually into an empty graph, resulting in a prohibitive number of edge copies (e.g., $59,118$ for Astro-Ph and even more for other datasets). In contrast, our algorithm can initialize the full graph either by the static algorithm or by inserting all edges in a batch, thereby avoiding OOM issues.

Our dynamic algorithms can also help update the maximum clique or size-constrained k-core.

8 Related Work

Cohesive Subgraph Search. Diverse models of cohesive subgraph are proposed in network analysis [49], [53], e.g., clique [15], quasi-clique [38], k-core [5], [26], [43], k-truss [18], [24], k-plex [48], and k-ecc [12], [61]. Some cohesive subgraph models can decompose a graph into a hierar-

chy, e.g., core decomposition [34], [51], truss decomposition [44], [46], [60], and ecc decomposition [12], [55].

k-Core Decomposition. An $O(m)$ time in-memory algorithm for core decomposition is proposed in [5]. Core maintenance [59] can update a k-core decomposition for dynamic graphs. The core decomposition under distributed configuration is introduced in [34]. An I/O efficient algorithm for core decomposition is proposed in [51]. Various variants of k-core are explored, including (k, r)-core [56], diversified coherent k-core [63], and skyline k-core [29].

Core Forest. Core forest is also known as hierarchical core decomposition (HCD). `LCPS` [32] can build the HCD of a graph in $O(m)$ time. The algorithm of HCD maintenance on large dynamic networks is proposed in [31]. The parallel algorithms for HCD and the subgraph search on the HCD are proposed in [17]. Core forest and its equivalences can facilitate community search query, e.g., `ShellStruct` [4] can handle the local query of k-cores, `CL-Tree` [21] can search attributed communities, and `ICP-Index` [30] can answer influential communities time-optimally.

Community Scoring Metrics. The metrics for community evaluation are surveyed in [10], [54] such as modularity [36] and clustering coefficient [25]. Community scoring metrics can formalize the notion of network communities and compare the communities produced by different algorithms [28]. Different communities can be found by the optimizations on different community metrics, e.g., [1], [14], [52]. Miyauchi et al. [33] introduce a reasonable community metric that outperforms the modularity. `GREEDY++` [8] is an $(1 - \epsilon)$-approximate solution for the densest subgraph problem [13]. Andersen et al. [2] propose an approximation algorithm for conductance using the Personalized PageRank. Tsourakakis et al. [45] study the k-clique densest subgraph problem, and the method cannot adapt to our problem. Different from these works, our paper aims to find the best k with respect to different metrics under a unified framework.

9 Conclusion

We study the problems of computing the best k-core set and the best single k-core with respect to a community scoring metric. We propose time and space optimal algorithms for the problems on static graphs, and time-bounded algorithms for the problems on dynamic graphs. Extensive experiments on 10 real-world graphs show that our algorithms significantly outperform the baselines in runtime and provide profound insights for the related problems.

References

[1] D. Aloise, G. Caporossi, P. Hansen, L. Liberti, S. Perron, and M. Ruiz. Modularity maximization in networks by variable neighborhood search. In *Graph Partitioning and Graph Clustering*, pages 113–128, 2012.

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

[3] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.

[4] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *DMKD*, 29(5):1406–1433, 2015.

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

[6] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[7] M. Bola and B. A. Sabel. Dynamic reorganization of brain functional networks during cognition. *Neuroimage*, 114:398–413, 2015.

[8] D. Boob, Y. Gao, R. Peng, S. Sawlani, C. Tsourakakis, D. Wang, and J. Wang. Flowless: Extracting densest subgraphs without flow computations. In *WWW*, pages 573–583, 2020.

[9] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using k-shell decomposition. *PNAS*, 104(27):11150–11154, 2007.

[10] T. Chakraborty, A. Dalmia, A. Mukherjee, and N. Ganguly. Metrics for community analysis: A survey. *CSUR*, 50(4):54:1–54:37, 2017.

[11] L. Chang. Efficient maximum clique computation over large sparse graphs. In *KDD*, pages 529–538, 2019.

[12] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, pages 205–216, 2013.

[13] C. Chekuri, K. Quanrud, and M. R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *SODA*, pages 1531–1555. SIAM, 2022.

[14] C. Chen, R. Peng, L. Ying, and H. Tong. Network connectivity optimization: Fundamental limits and effective algorithms. In *KDD*, pages 1167–1176, 2018.

[15] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *TODS*, 36(4):21:1–21:34, 2011.

[16] D. Chu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Y. Xia, and C. Zhang. Finding the best k in core decomposition: A time and space optimal solution. In *ICDE*, pages 685–696. IEEE, 2020.

[17] D. Chu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang. Hierarchical core decomposition in parallel: From construction to subgraph search. In *ICDE*, pages 1138–1151. IEEE, 2022.

[18] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16:3–1, 2008.

[19] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense implicit communities in the web graph. *TWEB*, 3(2):7:1–7:36, 2009.

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

[21] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[22] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *VLDB J.*, 29(1):353–392, 2020.

[23] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin. Efficient algorithms for densest subgraph discovery. *PVLDB*, 12(11):1719–1732, 2019.

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

[25] L. Katzir and S. J. Hardiman. Estimating clustering coefficients and size of social networks via random walk. *TWEB*, 9(4):19:1–19:20, 2015.

[26] W. Khaouid, M. Barsky, S. Venkatesh, and A. Thomo. K-core decomposition of large networks on a single PC. *PVLDB*, 9(1):13–23, 2015.

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

[28] J. Leskovec, K. J. Lang, and M. W. Mahoney. Empirical comparison of algorithms for network community detection. In *WWW*, pages 631–640, 2010.

[29] R. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, and Z. Zheng. Skyline community search in multi-valued networks. In *SIGMOD*, pages 457–472, 2018.

[30] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *PVLDB*, 8(5):509–520, 2015.

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

[32] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.

[33] A. Miyauchi and N. Kakimura. Finding a dense subgraph with sparse cut. In *CIKM*, pages 547–556, 2018.

[34] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE TPDS*, 24(2):288–300, 2013.

[35] F. Morone, G. Del Ferraro, and H. A. Makse. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics*, 15(1):95, 2019.

[36] M. E. Newman. Modularity and community structure in networks. *PNAS*, 103(23):8577–8582, 2006.
[37] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.
[38] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *KDD*, pages 228–238, 2005.
[39] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2):233–277, 1996.
[40] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[41] A. E. Sariyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *PVLDB*, 10(3):97–108, 2016.
[42] S. Sawlani and J. Wang. Near-optimal fully dynamic densest subgraph. In *STOC*, pages 181–193, 2020.
[43] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
[44] Y. Shao, L. Chen, and B. Cui. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*, pages 613–624, 2014.
[45] C. Tsourakakis. The k-clique densest subgraph problem. In *WWW*, pages 1122–1132, 2015.
[46] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.
[47] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin. Efficient computing of radius-bounded k-cores. In *ICDE*, pages 233–244, 2018.
[48] Y. Wang, X. Jian, Z. Yang, and J. Li. Query optimal k-plex based community in graphs. *DSE*, 2(4):257–273, 2017.
[49] Y. Wang, L. Yuan, Z. Chen, W. Zhang, X. Lin, and Q. Liu. Towards efficient shortest path counting on billion-scale graphs. In *ICDE*, pages 2579–2592. IEEE, 2023.
[50] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440, 1998.
[51] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. In *ICDE*, pages 133–144, 2016.
[52] Y. Wu, R. Jin, J. Li, and X. Zhang. Robust local community detection: On free rider effect and its elimination. *PVLDB*, 8(7):798–809, 2015.
[53] J. Xie, Z. Chen, D. Chu, F. Zhang, X. Lin, and Z. Tian. Influence maximization via vertex countering. *VLDB*, 17(6):1297–1309, 2024.
[54] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *KAIS*, 42(1):181–213, 2015.
[55] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.*, 26(2):275–300, 2017.
[56] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. When engagement meets similarity: Efficient (k, r)-core computation on social networks. *PVLDB*, 10(10):998–1009, 2017.
[57] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *Journal of Supercomputing*, 53(2):352–369, 2010.
[58] Y. Zhang and J. X. Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In *SIGMOD*, pages 1024–1041, 2019.
[59] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *ICDE*, pages 337–348, 2017.
[60] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.
[61] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*, pages 480–491, 2012.
[62] Z. Zhou, W. Zhang, F. Zhang, D. Chu, and B. Li. Vek: a vertex-oriented approach for edge k-core problem. *World Wide Web Journal*, pages 1–18, 2021.
[63] R. Zhu, Z. Zou, and J. Li. Diversified coherent core search on multi-layer graphs. In *ICDE*, pages 701–712, 2018.

Deming Chu is currently a Ph.D. student at the School of Computer Science and Engineering, University of New South Wales. He received the B.Eng. degree from the Software Engneering Institute, East China Normal University in 2017 and the M.Eng. degree from the same school in 2020. His research interests include graph data management and social networks.

Fan Zhang is a Professor at Guangzhou University. His research interests focus on graph management and analysis, including cohesive subgraphs, graph decomposition, network stability, influence spread, graph summarization, etc. Since 2017, he has published more than 20 papers as the first author or corresponding author in top venues, e.g., SIGMOD, SIGKDD, PVLDB, WWW, ICDE, TKDE, and VLDB Journal.

Wenjie Zhang received the PhD degree in computer science and engineering from the University of New South Wales, in 2010. She is currently a professor and ARC DECRA (Australian Research Council Discovery Early Career Researcher Award) fellow in the School of Computer Science and Engineering, the University of New South Wales, Australia. Her research interests lie in big data management and processing.

Xuemin Lin is a Chair Professor at Antai College of Economics and Management, Shanghai Jiao Tong University. He is Fellow of the IEEE, Member of Academia Europaea, and Fellow of Asia-Pacific Artificial Intelligence Association. He received his BSc degree in applied math from Fudan University in 1984 and his PhD degree in computer science from the University of Queensland in 1992. His principal research areas are databases and graph visualization.

Ying Zhang received his BSc and MSc degrees in Computer Science from Peking University, and PhD in Computer Science from the University of New South Wales. His research interests include query processing and analytics on large-scale data with focus on graphs and high dimensional data.

Chenyi Zhang received the PhD degree in computer science from both Zhejiang University and Simon Fraser University, in 2015. He is currently a technical expert at Huawei Cloud. His research interests include graph databases, graph processing systems and data mining.

Yinglong Xia leads the graph learning and analytics systems in Meta AI. Prior to that, he was a chief architect in Futurewei Technologies on AI and graph platform services, and a technical lead at IBM T.J. Watson Research Center, on graph database and reasoning frameworks. He serves as an associate editor of TKDE, and IEEE trans. on Big Data (TBD); he was a general co-chair of IEEE HiPC'19, a vice co-chair of IEEE BigData'19, a TPC member of KDD'19, CIKM'19, VLDB'20, and ICDE'19, a SPC member of KDD'22 and WSDM'23.