Accelerating K-Core Computation in Temporal Graphs

Zhuo Ma
University of New South Wales
Sydney, Australia
zhuo.ma@unsw.edu.au

Dong Wen
University of New South Wales
Sydney, Australia
dong.wen@unsw.edu.au

Hanchen Wang
University of Technology Sydney
Sydney, Australia
hanchen.wang@uts.edu.au

Wentao Li
University of Leicester
Leicester, United Kingdom
wl226@leicester.ac.uk

Wenjie Zhang
University of New South Wales
Sydney, Australia
wenjie.zhang@unsw.edu.au

Lu Qin
University of Technology Sydney
Sydney, Australia
Lu.Qin@uts.edu.au

Abstract

We address the problem of enumerating all temporal k-cores given a query time range and a temporal graph, which suffers from poor efficiency and scalability in the state-of-the-art solution. Given a simple graph G and an integer k, a k-core of G is a maximal connected subgraph of G in which every vertex has at least k neighbors. The problem aims to output distinct k-cores in all sub-windows of the given time range. The main challenge lies in avoiding redundant results for a potentially quadratic number of sub-windows. Motivated by an existing concept called core time, we propose a novel algorithm to compute all temporal k-cores based on core times and prove that the algorithmic running time is bounded by the size of all resulting temporal k-cores, which is theoretically optimal. Meanwhile, we empirically show that the cost of computing core times is significantly smaller than the total enumeration cost, indicating that the overall runtime is dominated by the output size. We conduct extensive experiments to demonstrate the efficiency of our proposed method and the significant improvement over existing solutions.

Keywords

Temporal Graphs, Community Search, Algorithms

1 Introduction

The temporal graph, where each edge is associated with a timestamp, models various real-world interactions between entities, such as bank transactions and social network interactions over time. For example, in a money transaction network, vertices represent bank accounts and temporal edges represent transactions between accounts at specific times. The k-core model [20] is a fundamental concept to identify cohesive subgraphs, drawing significant research attention due to its wide range of applications, including community detection, network visualization, and system structure analysis [4, 6, 8, 9, 15, 23, 27]. Given a graph, a k-core is defined as a maximal induced subgraph where each vertex has a degree of at least k.

Identifying k-cores in temporal graphs is valuable for tasks such as detecting suspicious account networks during anti-money laundering efforts within specific time frames [5, 11, 22]. To capture the k-core structure in temporal graphs, Yang et al. [25] study the time-range k-core query, which enables detecting cohesive subgraphs in a flexible time range. Specifically, given a temporal graph, a query integer k, and a query time range, the problem aims to enumerate k-cores appearing in the snapshot over any time windows within the query time range. The snapshot over a time window is an unlabeled graph induced by all edges falling in the time window. They use the term temporal k-core for a time window to distinguish it from the k-core model in unlabeled graphs.

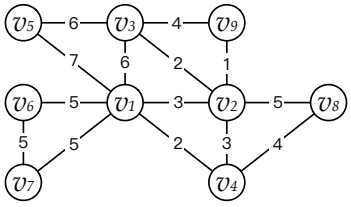


Figure 1: A temporal graph G.

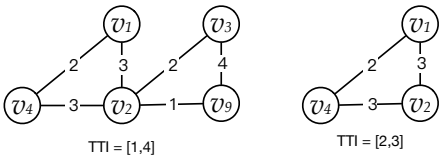


Figure 2: The temporal 2-cores of the sub-windows [1,4] and [2,3] of the graph in Figure 1 given the query time range $[1, 4]$.

Example 1. *Consider the temporal graph G in Figure 1. Given a query time range $[1, 4]$ and $k = 2$, we have two temporal 2-cores, one for the sub-window [1,4] and one for the sub-window [2,3], as shown in Figure 2. For the temporal 2-core of [2,3] containing $\{v_1, v_2, v_4\}$, it is the 2-core in the snapshots over the windows $[1, 3]$, $[2, 3]$, $[2, 4]$.*

Temporal k-core analysis is essential for identifying evolving dense structures in temporal graphs. While some applications focus on a fixed time range, many real-world scenarios require examining multiple sub-windows whose durations are not known in advance. For example, in social network analysis for misinformation detection [12, 14, 17], coordinated activities often occur in short and irregular bursts that may not align with any predefined window. Similarly, in disease outbreak monitoring and contact tracing [16, 21], transmission clusters can emerge and disappear rapidly over varying time scales. A real-world case study also has been conducted in [26] to use the temporal k-core model to mine real-world research communities from the collaboration network of DBLP. Enumerating temporal k-cores over all sub-windows enables the discovery of such transient yet meaningful structures that may be missed by single-window or static analyses.

Existing Studies. To enumerate all temporal k-cores in a time range, the general idea of [25] is to iterate all time windows

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.25

within the time range. Note that the k-core of a simple unlabeled graph can be computed by a peeling algorithm that continuously removes vertices with fewer than k neighbors. Motivated by this, they first process the widest window and iteratively narrow the window. Benefiting from this strategy, the temporal k-core of a window can be directly computed from the result of the previous window. Certain data structures and optimizations are also designed to speed up the algorithm. For instance, they skip a window if they already find the temporal k-core of the current window is the same as that of the previous window based on several properties. Despite a spectrum of optimizations, there is no clear better time complexity than $O(t_{max}^2 \cdot B)$ shown in [25], where t_{max} is the number of timestamps in the query time range and B is the average cost of computing the temporal k-core for one time window. The main bottleneck of their method lies in $O(t_{max}^2)$ iterations to scan windows. Note that t_{max} is up to the number of edges in the query range if the time label of each edge is unique. Another closely related work is the PHC-Index [26] which enables determining whether a vertex is in the k-core of an arbitrary window. The index relies on a key concept called core time, which is the earliest end time t_e for a vertex u, a start time t_s, and an integer k such that u is in the k-core of the snapshot of the time window $[t_s, t_e]$. For each possible integer k and each vertex u, the PHC-Index stores distinct core times of u for all start times. To construct this index, Yu et al. [26] propose an algorithm with time complexity $O(|PHC| \cdot deg_{avg})$, where $|PHC|$ denotes the size of the PHC index and deg_{avg} is the average vertex degree.

The Framework. In this paper, we propose an efficient framework for time-range k-core enumeration. Our approach initially computes core times in the query time range by the techniques of [26] and leverages core times to compute resulting subgraphs. Note that the original PHC-Index [26] computes core times for all integers k of k-core. We refer to the core times for a specific integer k as the Vertex Core Time index ($\mathcal{VCT}$) for clarity. Extending the $\mathcal{VCT}$ to enumerate time-range k-cores is challenging. Although the core time allows us to determine whether a vertex belongs to the temporal k-core for a time window, it does not clearly reveal the massive result redundancy across overlapping time windows. Specifically, the k-core subgraph in our problem may appear repeatedly across many sub-windows. Naively enumerating all such windows leads to substantial redundancy. Therefore, the central challenge lies in avoiding repeated outputs, which may incur unnecessary computing costs. Solving this problem requires identifying the minimal temporal window that a k-core exists and pruning dominated windows without explicitly enumerating them.

Based on $\mathcal{VCT}$, we derive minimal core windows for edges, defined as minimal time windows in which an edge belongs to the temporal k-core of a time window. Intuitively, the minimal core windows of edges compress the relationship between the edge and the k-cores of all possible time windows. Therefore, we expect to have an efficient algorithm to directly compute all temporal k-cores based on the minimal core windows, and the method immediately avoids computing k-cores of individual time windows by analyzing the relationship with minimal core windows and resulting subgraphs.

Efficient Enumeration. Our main technical contribution is an efficient method to enumerate the temporal k-cores for all time windows within the query time range using the minimal core windows of edges. Starting from a given start time t_s, we filter out minimal core windows that do not contribute to any k-core at t_s and sort the remaining windows by end times. We iteratively scan these windows, outputting the corresponding edges of all previous windows as a k-core when specific conditions are met, ensuring no duplicate results. To update the data structure from one start time to the next, we use a doubly linked list to maintain window order. By preprocessing all minimal core windows in linear time, we can update the order from $\mathcal{L}'$ to $\mathcal{L}$ in $O(|\mathcal{L} \setminus \mathcal{L}'|)$ time complexity, where $\mathcal{L} \setminus \mathcal{L}'$ is the difference between the set of windows in two orders. Consequently, the time complexity for enumerating k-cores for all start times is bounded by the result size, making it theoretically optimal in terms of time complexity. Overall, our solution has a time complexity of $O(|\mathcal{VCT}| \cdot deg_{avg} + |\mathcal{R}|)$, where $|\mathcal{VCT}| \cdot deg_{avg}$ accounts for computing minimal core windows and $|\mathcal{R}|$ represents the result size. We report the $|\mathcal{VCT}| \cdot deg_{avg}$ and $|\mathcal{R}|$ for several representative real datasets in Figure 4. The result size is much larger than $|\mathcal{VCT}| \cdot deg_{avg}$ in all datasets, which demonstrates that the overall running time of our algorithm is mainly related to the result size.

Contribution. We summarize our main contributions below.

- *A Novel Framework for Time-Range K-Core Queries.* We propose a new framework to precompute the minimal core windows of all edges instead of directly computing k-cores of each individual window in the state-of-the-art algorithm. Section 5.2 proposes a basic implementation to enumerate all results based on minimal core windows.
- *Enumeration in Theoretically Optimal Time Complexity.* Given minimal core windows of edges, we propose an algorithm to enumerate all temporal k-cores in a time complexity theoretically bounded by the result size.
- *Extensive Performance Studies.* In our experiments, our final algorithm is two orders of magnitude faster than the state-of-the-art algorithm for the same problem on most datasets.

2 Preliminary

We study an undirected temporal graph $G(V, E)$ where each edge $(u, v, t) \in E$ is associated with a timestamp t that indicates the interaction time between vertices u and v. We use E_t to denote all edges with the associated time t. Without loss of generality, we denote the timestamps of edges as a continuous set of integers starting from 1. We assume each pair of vertices has at most one edge for simplicity, and our solution can be easily extended for the existence of multiple edges between two vertices. We use $deg(u)$ to denote the degree of a vertex u. The projected graph of G over a time window $[t_s, t_e]$, denoted by $G_{[t_s,t_e]}$ is the temporal subgraph including all edges in $[t_s, t_e]$.

Definition 1 (K-Core [20]). *Given a simple graph G and an integer k, the k-core of G is the maximal connected subgraph of G in which every vertex has at least k neighbors.*

The concept of temporal k-core extends the k-core model for temporal graphs. We present it as follows.

Definition 2 (Temporal K-Core [25]). *Given a temporal graph G and a window $[t_s, t_e]$, the temporal k-core of $[t_s, t_e]$ is the maximal subgraph C of $G_{[t_s,t_e]}$ where every vertex has at least k neighbors.*

The temporal k-core of a time window may persist unchanged over a shorter time span. To capture this minimal stable duration, we introduce the notion of the tightest time interval (TTI), which will serve as a basic building block in later pruning and enumeration steps.

Definition 3 (Tightest Time Interval). *Given the temporal k-core C of a time window $[t_s, t_e]$, its tightest time interval (TTI),*

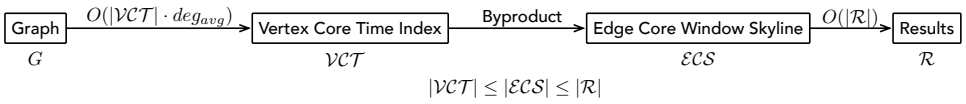


Figure 3: Overview of our framework.

denoted by $\mathcal{W}(C)$, is the minimal sub-window $[t'_s, t'_e] \subseteq [t_s, t_e]$ such that the temporal k-core of $[t'_s, t'_e]$ is identical to C, and no proper sub-window of $[t'_s, t'_e]$ has the same k-core.

The TTI thus captures the exact temporal span within which C remains structurally unchanged. Intuitively, the TTI of C corresponds to the minimal time window that contains all temporal edges contributing to C.

Definition 4 (Tightest Time Interval (old)). *The tightest time interval (TTI) of a temporal k-core C, denoted by $\mathcal{W}(C)$, is the minimal time window containing all edges in C.*

To study k-cores in a query time range of a temporal graph, we present the research problem studied in [25] as follows.

Problem Statement. Given a temporal graph G, integer k, and a time range $[T_s, T_e]$, we aim to compute the temporal k-cores of all time windows $[t_s, t_e] \subseteq [T_s, T_e]$.

We use the term *range* $[T_s, T_e]$ for the user-specified query range, *window* $[ts, te]$ for any specific time window examined inside the range. The results for the query window $[1, 4]$ and the query integer $k = 2$ on the graph G of Figure 1 is shown in Figure 2. Given a query time range $[T_s, T_e]$, we use $N(u)$ to denote the neighbors of u in $G_{[T_s,T_e]}$ for ease of presentation when the context is clear. We use t_{max} to denote the number of distinct time labels in the query range, i.e., $t_{max} = T_e - T_s + 1$. We use n and m to denote the number of vertices and the number of edges in the projected graph over the query range $[T_s, T_e]$. The temporal k-cores of two time windows are considered the same in [25] if they have the same set of edges. The same temporal k-core may exist in multiple time windows, and any solution for the problem should avoid repeated outputs. We omit proofs for some lemmas and theorems in the paper if they are straightforward.

3 Existing Studies

3.1 The State of the Art

To enumerate temporal k-cores, [25] proposed an algorithm called Optimized Temporal Core Decomposition (OTCD). The algorithm computes temporal k-cores decrementally by considering the projected graph from wide time windows to narrow time windows. Algorithm 1 presents the algorithm. Given a query time range $[T_s, T_e]$, OTCD first removes all edges not in $[T_s, T_e]$ and computes the k-core in the truncated graph by iteratively removing all vertices that do not have at least k neighbors. Next, OTCD enumerates each window $[t_s, t_e] \subseteq [T_s, T_e]$ in a specific order to compute the temporal k-core for the corresponding time windows decrementally from previously computed temporal k-cores. Specifically, OTCD initializes with $t_s = T_s$ and $t_e = T_e$. Then, it anchors the start time t_s and decreases the end time t_e from T_e to t_s. Once t_e reaches t_s, the algorithm increments t_s by one and repeats the process until t_s reaches T_e.

A key optimization in OTCD is the use of the tightest time interval (TTI), which identifies the minimal window over which a temporal k-core remains unchanged. Based on Definition 4, OTCD applies three rules: Pruning-on-the-Right (PoR), Pruning-on-the-Underside (PoU), and Pruning-on-the-Left (PoL) to avoid

Algorithm 1: OTCD

```
Input: G, k, [T_s, T_e]
Output: all distinct temporal k-cores in [T_s, T_e]
1  for t_s ← T_s to T_e do
2      t_e ← T_e;
3      if t_s = T_s then
4          obtain C_[t_s,t_e] by truncating G_[T_s,T_e] and
             performing core decomposition on it;
5      else
6          obtain C_[t_s,t_e] by truncating C_[t_s−1,t_e] and
             performing core decomposition on it;
7      output C_[t_s,t_e] if it is not marked as pruned;
8      for t_e ← T_e − 1 to t_s do
9          if [t_s, t_e] is not marked as pruned then
10             obtain C_[t_s,t_e] by truncating C_[t_s,t_e+1] and
                 performing core decomposition on it;
11             compute time windows that need to be
                 pruned based on TTI of the C_[t_s,t_e];
12             output C_[t_s,t_e];
```

computing temporal k-cores of unnecessary time windows. For a start time t_s and an end time t_e, a temporal k-core is derived with the TTI $[t'_s, t'_e]$. PoR prunes all the time windows starting from t_s ending from t'_e to t_e. In the case of $t'_s > t_s$, PoU prunes all the time windows starting not earlier than t_s and ending later than t'_s.To support these pruning rules efficiently, OTCD maintains a timestamp-ordered adjacency structure that allows checking, for any vertex, how its degree evolves as the time window shrinks. This enables constant-time tests for whether removing an end time violates the k-core condition, making the pruning operations efficient.

Challenges. Despite practical optimizations to prune certain time windows, the OTCD algorithm essentially checks every window in the query range and is time-consuming. The time complexity of OTCD is $O(|t_{max}| \cdot (m \cdot log(n) + m))$, where $O(m \cdot log(n) + m)$ is the running time to compute temporal k-cores for ranges from one start time and all possible end times. The time complexity can also be represented as $O(|t_{max}|^2 \cdot B)$, where B represents the average cost of computing the temporal k-core for one time window.

3.2 Other Related Works

Various k-core-related problems have been studied in temporal graphs, incorporating different temporal objectives and constraints beyond cohesiveness. Historical k-cores focus on snapshots at specific times [26], while maximal span-cores require edges to appear continuously throughout a time range [7]. The (θ, τ)-persistent k-core maintains a k-core across θ-length subwindows if its persistence exceeds τ [9]. Continual cohesive subgraph search finds subgraphs containing a queried vertex under structural and temporal constraints [10]. Dense subgraphs in

Table 1: The vertex core time index of the temporal graph G Figure 1 for $k = 2$.

v_1: [1, 3], [3, 5], [6, 7], [7, ∞]	v_6: [1, 5], [6, ∞]
v_2: [1, 3], [3, 5], [4, ∞]	v_7: [1, 5], [6, ∞]
v_3: [1, 4], [2, 6], [3, 7], [4, ∞]	v_8: [1, 5], [4, ∞]
v_4: [1, 3], [3, 5], [4, ∞]	v_9: [1, 4], [2, ∞]
v_5: [1, 7], [7, ∞]	

weighted temporal networks explore fixed vertex sets with varying edge weights [13]. The (k, h)-core ensures each vertex has at least k neighbors with h interactions [24], while density bursting subgraphs (DBS) identify subgraphs with the fastest-growing density over time [5]. Periodic community detection reveals recurring interaction patterns such as periodic k-cores [18, 19]. Finally, quasi-(k, h)-cores provide detailed measures for temporal graphs and focus on efficient maintenance [3]. These models enhance the understanding of cohesive substructures in dynamic networks.

4 Solution Overview

This section introduces the key concepts and high-level structure of our framework. To improve the efficiency of temporal k-core enumeration, we present an overview of our solution in Figure 3. Our idea is inspired by a study on computing vertex core times [26], which is formally defined as follows.

Definition 5 (Vertex Core Time). *Given a temporal graph G, an integer k, a start time t_s and a vertex u, the core time of u, denoted by $CT_{t_s}(u)_k$, is the earliest end time t_e such that u is in the temporal k-core of $G_{[t_s,t_e]}$.*

Example 2. *Refer to the temporal graph in Figure 1. For $k = 2$, we compute the core time of vertices at the start time $t_s = 1$ by expanding the time windows $[t_s, t_e]$ from $[1, 1]$ to $[1, 7]$. As we expand the time window to $[1, 3]$, v_1 joins the 2-core. This is the earliest end time such that v_1 joins the 2-core. Therefore, its core time for $t_s = 1$ is 3, i.e. $CT_1(v_1)_2 = 3$. Similarly, for $t_s = 3$, $CT_3(v_1)_2 = 5$.*

We omit the subscript k in Definition 5 when it is clear from the context. By replacing vertex with edge, we also have the concept of edge core time. The existing work [26] computes the core times of all vertices for all start times. Given that the core time of a vertex for a series of continuous start times may be the same, they only output distinct core times with the corresponding earliest start times. For simplicity, we call this structure Vertex Core Time Index, denoted by $\mathcal{VCT}$.

Example 3. *Table 1 shows the vertex core time index of the temporal graph G for $k = 2$ in Figure 1. For each vertex, the table lists all labels in its $\mathcal{VCT}$, where each label is a window $[t_s, CT_{t_s}]$ indicating that the core time of the vertex is CT_{t_s} for the start time t_s. The index adopts an incremental representation: a new label is stored only when the core time changes for a later start time. For the vertex v_1, $[1, 3]$ indicates that the core time of v_1 is 3, and the vertex is in the 2-core of all windows staring from 1 ending by a time not earlier than 3. The next label of $[1, 3]$ is $[3, 5]$. This implies the core time of v_1 for $t_s = 2$ is still 3, and its core time for $t_s = 3$ is 5.*

The vertex core time index can be computed in $O(|\mathcal{VCT}| \cdot deg_{avg})$ time complexity [26], where $|\mathcal{VCT}|$ is the size of the vertex core time index and deg_{avg} is the average vertex degree in the graph. We will discuss the basic idea of computing $\mathcal{VCT}$ in Section 5.1. The vertex core time index essentially compresses vertices in k-cores of all possible windows to some extent. This motivates us to develop algorithms to transfer it into all temporal k-cores. Given that each temporal k-core is distinguished by their edges [25], our idea is to first compute a structure to compress all possible windows for each edge e such that e is in the k-core of the window. Then, we utilize the windows of each edge and assemble them to produce the final temporal k-cores. To this end, we first derive the *edge core window skyline* of each edge, denoted as $\mathcal{ECS}$, as a by-product of computing vertex core times without increasing the time complexity. The core window skyline of each edge is the set of all minimal core windows defined as follows.

Table 2: The minimal core windows (edge core window skyline) of all edges in the temporal graph G of Figure 1 for $k = 2$

$(v_2, v_9, 1)$: [1, 4]	$(v_1, v_6, 5)$: [5, 5]
$(v_1, v_4, 2)$: [2, 3]	$(v_1, v_7, 5)$: [5, 5]
$(v_2, v_3, 2)$: [1, 4], [2, 6]	$(v_2, v_8, 5)$: [3, 5]
$(v_1, v_2, 3)$: [2, 3], [3, 5]	$(v_6, v_7, 5)$: [5, 5]
$(v_2, v_4, 3)$: [2, 3], [3, 5]	$(v_1, v_3, 6)$: [2, 6], [6, 7]
$(v_3, v_9, 4)$: [1, 4]	$(v_3, v_5, 6)$: [6, 7]
$(v_4, v_8, 4)$: [3, 5]	$(v_1, v_5, 7)$: [6, 7]

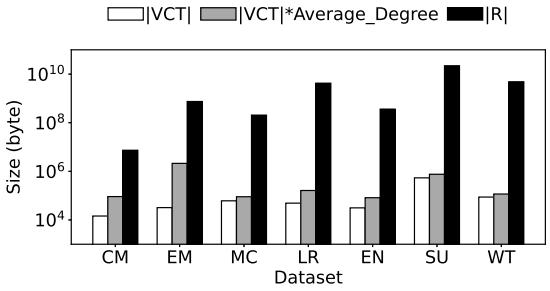


Figure 4: $|\mathcal{VCT}|$, $|\mathcal{VCT}| \cdot deg_{avg}$, and $|\mathcal{R}|$ for seven representative datasets given default parameters (k = 30% k_{max}, t = 10% t_{max}).

Definition 6 (Minimal Core Window). *Given a temporal graph G, an integer k, and an edge e, a time window $[t_s, t_e]$ is a minimal core window of e if (1) e is in the k-core of $G_{[t_s,t_e]}$; and (2) e is not in the k-core of $G_{[t'_s,t'_e]}$ for any $[t'_s, t'_e] \subset [t_s, t_e]$.*

Remark. The minimal core windows of an edge form a skyline because no minimal core window dominates another. A window $[t_s, t_e]$ dominates $[t'_s, t'_e]$ if $t_s \geq t'_s$ and $t_e \leq t'_e$. Suppose an edge appears in the k-core in both $[2, 3]$ and $[1, 4]$, then the time window $[2, 3]$ dominates $[1, 4]$, and $[1, 4]$ cannot be a minimal core window because it is dominated by another time window.

Example 4. *Table 2 presents the $\mathcal{ECS}$ for all edges in the graph of Figure 1 for $k = 2$. Each edge is associated with a set of labels stored incrementally: a new label $[t_s, t_e]$ is added only when the minimal core window of the edge changes for a new start time t_s, following Definition 6. Table 2 presents the $\mathcal{ECS}$ for all edges in the graph of Figure 1 for $k = 2$. The edge (v_2, v_9) has one minimal core window $[1, 4]$, which indicates that (v_2, v_9) is contained in a 2-core*

Algorithm 2: CoreTime

Input: $G(V, E), T_s, T_e, k$
Output: $\mathcal{ECS}$

```
1  compute the vertex core time index by [26];
   /* initialize edge core times                    */
2  foreach edge (u, v, t) in G_[T_s,T_e] do
3      CT(u, v) ← max(CT_{T_s}(u), CT_{T_s}(v), t);
4      ECS(u, v) ← ∅;
5  foreach T_s ≤ t < T_e do
6      foreach vertex u w.r.t. CT_{t+1}(u) ≠ CT_t(u) do
7          foreach ⟨v, t'⟩ ∈ N(u)_[t+1,T_e] do
               /* update edge core times            */
8              new_ct ← max(CT_{t+1}(u), CT_{t+1}(v), t');
9              if new_ct > CT(u, v) then
10                 ECS(u, v) ←
                     ECS(u, v) ∪ {[t, CT(u, v)]};
11                 CT(u, v) ← new_ct;
12 return ECS
```

in the time window $[1, 4]$. *For any sub-window of* $[1, 4]$, (v_2, v_9) *is not contained in any 2-cores in that sub-window.*

Framework Overview. Our algorithm consists of two phases: (i) preprocessing the temporal graph to construct the edge core window skyline, and (ii) enumerating all temporal k-cores over the query range. The $\mathcal{ECS}$ summarizes, for each edge, the maximal sub-windows in which both endpoints can simultaneously satisfy the k-core requirement. By compactly capturing all admissible windows, the $\mathcal{ECS}$ serves as a pruning structure that eliminates the vast majority of sub-windows that cannot lead to valid temporal k-cores. During enumeration, we iteratively expand each start time t_s and consult the $\mathcal{ECS}$ to determine the feasible end times t_e that may produce a valid temporal k-core. For each such candidate window, we extract the induced subgraph and compute its k-core, contributing to the final result set. In this way, the $\mathcal{ECS}$ provides a coarse but effective search boundary, while the enumeration phase extracts only the realizable dense structures. This design yields an overall complexity of $O(|\mathcal{VCT}| \cdot deg_{avg} + |\mathcal{R}|)$, where the first term corresponds to skyline construction and the second corresponds to output enumeration.

Remark. Figure 4 presents $|\mathcal{VCT}|$, $|\mathcal{VCT}| \cdot deg_{avg}$, and $|\mathcal{R}|$ for several representative datasets given the default query parameters in our experiments. All other datasets evaluated in our experiments present the similar trend. The sizes of the result sets are 2 to 4 orders of magnitude larger than $|\mathcal{VCT}| \cdot deg_{avg}$ for all datasets. This indicates that the time complexity of our final algorithm is mainly related to the result size $|\mathcal{R}|$ in practice.

5 Enumerating Temporal K-Cores

5.1 Computing Edge Core Window Skyline

We first review the existing technique [26] to compute the vertex core time index. Given the query window $[T_s, T - e]$, they first compute the core time of each vertex for the start time T_s. Then, they update the core time by increasing the start time from T_s to T_e and record all distinct core time values. To this end, certain values for each vertex are maintained when increasing the start time. Monitoring these values enables identifying whether the core time of any vertex changes. If so, the core time is recomputed for the current start time. They scan neighbors of a node to recompute the vertex core time for any start time, and no extra cost is required to monitor if the core time of any vertex needs change. As a result, they achieve $O(|\mathcal{VCT}| \cdot deg_{avg})$ time complexity.

Deriving Edge Core Window Skyline. To compute the edge core window skyline, we first demonstrate a correspondence between the vertex core time and the edge core time. In this way, we update the edge core time when any vertex core time changes.

Lemma 1. *Given a start time t_s and an integer k, the core time of an edge (u, v, t) is $\mathcal{CT}_{t_s}(u, v, t) = \max(\mathcal{CT}_{t_s}(u), \mathcal{CT}_{t_s}(v), t)$.*

Based on Lemma 1, we initialize the edge core time for the start time T_s in lines 3. $\mathcal{ECS}(u, v)$ in line 4 is used to collect all minimal core windows for (u, v). Then we update the core time for each edge in line 7 if the core time of any terminal of the edge updates. We present the following lemma to derive the edge minimal core window when the edge core time updates.

Lemma 2. *Given a start time t_s and an integer k, assume the core time of an edge $e = (u, v, t)$ for t_s is different from that for $t_s + 1$. We have $[t_s, \mathcal{CT}_{t_s}(e)]$ is a minimal core window of e for k.*

Proof. We prove this lemma by contradiction. Suppose we have $\mathcal{CT}_{t_s+1}(e) = \mathcal{CT}_{t-s}(e)$ and $[t_s, \mathcal{CT}_{t_s}(e)]$ is not a minimal core window of e, then e is within the k-core of the time window $[t_s+1, \mathcal{CT}_{t_s+1}(e)] \subset [t_s, \mathcal{CT}_{t_s+1}(e)]$, which contradicts the definition of minimal core window. □

Based on Lemma 2, we derive a minimal core window of the edge in line 10 when its core time updates.

Example 5. *Refer to the graph in Figure 1. We compute the $\mathcal{ECS}$ for $k = 2$ by Algorithm 2. Initially at $t_s = 1$, $\mathcal{CT}_1(v_2) = 3$ and $\mathcal{CT}_1(v_3) = 4$. Therefore by line 3, we have $\mathcal{CT}_1(v_2, v_3) = \max(3, 4, 2) = 4$. As we move onto $t_s = 2$, the core time of v_3 has changed to $\mathcal{CT}_2(v_3) = 6$, which is larger than the existing core time of the edge $\mathcal{CT}_1(v_2, v_3) = 4$. Therefore, we have $\mathcal{CT}_2(v_2, v_3) = 6$. Since the core time of (v_2, v_3) has changed at $t_s = 2$, we add $[1, 4]$ into $\mathcal{ECS}(v_2, v_3)$ by line 10. Then, we update the core time of (v_2, v_3) to 6 by line 11.*

Processing the edge core time (lines 2–4 and lines 9–11) does not incur additional time complexity. We achieve the same time complexity of computing vertex core time in [26], which is formally summarized as follows.

Theorem 1. *The time complexity of Algorithm 2 is $O(|\mathcal{VCT}| \cdot deg_{avg})$.*

5.2 A Straightforward Method

We first discuss a basic solution to enumerate all temporal k-cores given the minimal core windows of all edges. Similar to OTCD [25], our algorithm enumerates all time windows within the query time range. Given each time window, the following lemma demonstrates a way to derive its temporal k-core based on minimal core windows of edges.

Lemma 3. *Given a time window $[t_s, t_e]$ and its temporal k-core C, an edge e is in C if and only if there is a minimal core window $[t_1, t_2]$ of e contained in $[t_s, t_e]$, i.e., $[t_1, t_2] \subseteq [t_s, t_e]$.*

Based on Lemma 3, the union of all edges satisfying the condition is the temporal k-core of $[t_s, t_e]$. Algorithm 3 presents the basic solution. For each start time t_s from T_s to T_e, we initialize a

Algorithm 3: EnumBase

Input: a temporal graph G, an integer k, a time range $[T_s, T_e]$, the $\mathcal{ECS}$ of G for k
Output: all distinct temporal k-cores in $[T_s, T_e]$

```
1  R ← ∅;
2  foreach T_s ≤ t_s ≤ T_e do
3      foreach t_s ≤ t_e ≤ T_e do B[t_e] ← ∅;
4      foreach edge e ∈ ECS do
5          [t1, t2] ← the first window in ECS(e) such that
             t1 ≥ t_s;
6          B[t2] ← B[t2] ∪ {(e)}
7      C ← ∅;
8      foreach t_s ≤ t_e ≤ T_e do
9          if B[t_e] = ∅ then continue;
10         C ← C ∪ B[t_e];
11         if C ∈ R then continue;
12         R ← R ∪ {C};
13 return R
```

set of buckets (line 3). Then, we obtain the relevant window of all edges that satisfies the condition in Lemma 3 and insert the corresponding edge into the designated bucket based on the end time of the window (lines 4-6). After the bucket is constructed, for each end time t_e, we insert all vertices in the bucket to the current edge set (line 10). A minor optimization is applied in line 9, where $B[t_e] = \emptyset$ means the temporal k-core is the same as that for $[t_s, t_e - 1]$. Then, we check if the current temporal k-core is already in the result set in line 11. To this end, we maintain all computed temporal k-cores by a hash table in $\mathcal{R}$. We add the temporal k-core into the result set in line 12. By utilizing the edge core window skyline, we are able to find all temporal k-cores for each time window.

Drawbacks of Algorithm 3. The improvement opportunities of Algorithm 3 lie in two aspects. First, even with an optimization (line 9) applied to reduce the examination of unnecessary time windows, the algorithm still needs to scan $O(t_{max}^2)$ time windows in the worst case. Second, the same temporal k-core may be computed when processing multiple time windows. This implies significant costs for unnecessary computations.

5.3 Anchoring the Start Time

To respond to the drawbacks of the basic solution, we design an algorithm that (1) only scans time windows with an undiscovered temporal k-core and (2) visits each temporal k-core only once. Given the edge core window skyline in this section, we first discuss the data structure and the corresponding algorithm to enumerate all temporal k-cores for a certain start time, which is a subproblem of temporal k-core enumeration. We will discuss how to update the structure from one start time to the next start time in Section 5.4, which produces all final results.

Based on Definition 4, there is a one-to-one correspondence between a temporal k-core and its Tightest Time Interval (TTI). In other words, if we find TTIs of all temporal k-cores, we can output the temporal k-core of each TTI as a result. Recall that the temporal k-core of a time window (TTI) $[t_s, t_e]$ can be computed by the union of all edges with a minimal core window in $[t_s, t_e]$ (Lemma 3). Therefore, we mainly discuss the relation between TTIs and minimal core windows below, which produces a method to derive TTIs based on minimal core windows. For simplicity, we say the start time (resp. the end time) of a temporal k-core C to represent the start time (resp. the end time) of the TTI of C. Note that the temporal k-core C of a time window $[t_s, t_e]$ does not guarantee $[t_s, t_e]$ is a TTI of C (the TTI may be a subwindow of $[t_s, t_e]$). We start by showing that not all possible time labels can be the start time of a temporal k-core.

Lemma 4. *A temporal k-core starting from t_s exists if and only if there exists a minimal core window $[t_1, t_2]$ of an edge with $t_1 = t_s$.*

Proof. We first assume that a temporal k-core exists with TTI $[t_s, t_e]$. There must exist a minimal core window starting at t_s. Otherwise, we could increase the start time of the temporal k-core to the earliest value t_s' from which a minimal core window begins. By Lemma 3, the set of edges in the temporal k-core of $[t_s', t_e]$ is identical to that of $[t_s, t_e]$, contradicting the assumption that $[t_s, t_e]$ is the TTI. Next, let $[t_1, t_2]$ be a minimal core window of an edge e, and let C be the temporal k-core of $G_{[t_1,t_2]}$. It follows immediately that $[t_1, t_2]$ is the TTI of C, because e does not appear in the temporal k-core of any strict sub-window of $[t_1, t_2]$. □

Lemma 4 provides a necessary and sufficient condition for the start time of any temporal k-core. For simplicity, we call a time t_s a *valid start time* if there exists a minimal core window $[t_1, t_2]$ for an edge with $t_1 = t_s$. Next, given the valid start time t_s, we first discuss several properties of the end time t_e such that a temporal k-core exists with the TTI $[t_s, t_e]$. Then, we discuss the structure to efficiently enumerate edges of temporal k-core for different valid end times.

Valid End Times. We propose two necessary conditions for the end time of temporal k-cores as follows.

Lemma 5. *A temporal k-core exists with a TTI $[t_s, t_e]$ only if there exists a minimal core window $[t_1, t_2]$ for an edge e such that (1) $t_s \le t_1, t_2 = t_e$; and (2) there does not exist a minimal core window $[t_1', t_2']$ for e with $t_s \le t_1' < t_1$.*

Proof. We prove both conditions by contradiction. For the first condition, suppose we have a temporal k-core with a TTI $[t_s, t_e]$ and there does not exist a minimal core window $[t_1, t_2]$ such that $t_s \le t_1$ and $t_2 = t_e$. Based on Lemma 3, there must exist a minimal core window of an edge in the TTI. Therefore, we can decrease t_e until we find a window $[t_1, t_2]$ satisfying $t_s \le t_1$ and $t_2 = t_e$. As a result, we derive the same temporal k-core with a tighter TTI, which contradicts the initial assumption.

For the second condition, suppose we have a set of minimal core windows satisfying condition 1. For each of such windows, there exists another window $w' = [t_1', t_2']$ such that $t_s \le t_1' < t_1$. Based on the definition of $\mathcal{ECS}$, we have $t_2' < t_2$. Therefore, the same temporal k-core exists by decreasing the end time of TTI based on Lemma 3, which produces a contradiction. □

The minimal core window $[t_1, t_2]$ in Lemma 5 is clearly the earliest one among all minimal core windows of e starting not earlier than t_s. Lemma 5 implies that only one minimal core window for each edge is required to enumerate all temporal k-cores starting from t_s. Motivated by this, we define the active time of each minimal core window as follows to indicate whether the window should be considered for a specific start time.

Definition 7 (active time). *Given a set of minimal core windows of an edge e, let the windows be ordered by increasing t_1 (and consequently increasing t_2). For a specific minimal core window $w = [t_1, t_2]$, let $w' = [t_1', t_2']$ the immediately preceding window in*

this order, such that $t_1' < t_1$. If such a preceding window w' exists, the activation time of w, denoted as $w.active$, is $t_1' + 1$. Otherwise, $w.active = 1$.

Based on Definition 7, the condition (2) in Lemma 5 can be replaced by $[t_1, t_2].active \leq t_s$.

Example 6. *Consider the graph G in Figure 1 and the corresponding edge core window skyline in Table 2. For the minimal core window $[3, 5]$ of the edge $(v_1, v_2, 3)$, its active time is 3. That means we do not need to consider the window $[3, 5]$ for the temporal k-core with the TTI starting from 1 or 2.*

To derive the temporal k-core starting from t_s, Lemma 5 and Definition 7 motivate us to collect all minimal edge windows with active times not later than t_s and start times not earlier than t_s. Then, we only consider the end times of the windows as the potential end times of temporal k-cores. Next, we present the second property to further filter end times by minimal core time windows of edges.

Lemma 6. *A temporal k-core exists with a TTI $[t_s, t_e]$ only if there exists a minimal core window $[t_1, t_2]$ of an edge e such that $t_1 = t_s, t_2 \leq t_e$.*

Recall that there must exist at least one minimal core window starting from the same time of a temporal k-core based on Lemma 4. Lemma 6 indicates that the end time of the temporal k-core is not earlier than the earliest end time of those windows. Below, we show that the aforementioned two necessary conditions are sufficient for the end time of temporal k-cores.

Theorem 2. *A temporal k-core with a TTI $[t_s, t_e]$ exists if conditions in both Lemma 5 and Lemma 6 hold.*

Proof of Theorem 2. Based on either Lemma 5 or Lemma 6, there must exist a minimal core window contained in $[t_s, t_e]$. This proves that a temporal k-core C exists in $[t_s, t_e]$. Next, we prove that $[t_s, t_e]$ is the TTI of the temporal k-core C. To this end, let C' be the temporal k-core of $[t_s, t_e]$ satisfying Lemma 5 and Lemma 6. We show that narrowing the window will remove edges from C'. First, let e be the edge meeting the condition in Lemma 5. It is clear to see that e is in C', and decreasing the end time te of the window will exclude e from C'. Next, we show that increasing the start time t_s of the window will also exclude an edge from C' by the following lemma.

Lemma 7. *Let $[t_1, t_2]$ be a minimal core window such that there does not exist a minimal core window $[t_1', t_2']$ with $t_1 = t_1'$ and $t_2 > t_2'$. An edge $e = (u, v, t_1)$ exists and $[t_1, t_2]$ is a minimal core window of e.*

Proof. Assume the edge e does not exist. Removing all edges at t_s would not change the temporal k-core of $[t_s, t_e]$. All edges with the minimal core window $[t_1, t_2]$ are still in the temporal k-core of $[t_1 + 1, t_2]$, which contradicts that $[t_1, t_2]$ is a minimal core window. □

Given of all minimal core windows satisfying the condition in Lemma 6, let $[t_1', t_2']$ be one of them (i.e., $t_1' = t_s, t_2' \leq t_e$) with the earliest end time. Lemma 7 indicates there must exist an edge e at t_s with a minimal core window $[t_1', t_2']$. Therefore, e is in C', but increasing t_s will exclude e from the projected graph and the temporal k-core C'. We finish the proof of Theorem 2.

The Structure & Algorithm. We now have necessary and sufficient conditions for the end time given a valid start time t_s, which enables us to present the enumeration algorithm. To check

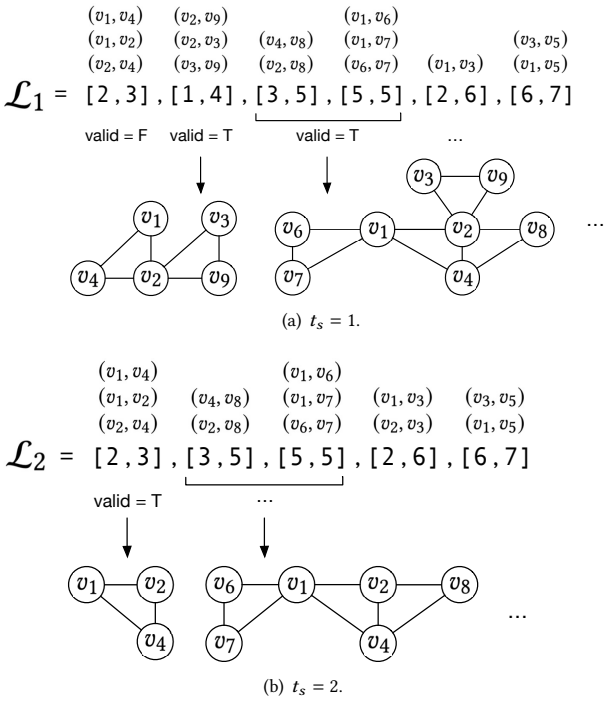


Figure 5: Illustration of enumerating $\mathcal{L}_{t_s}$ for $t_s = 1$ and $t_s = 2$.

the condition in Lemma 6 efficiently given the start time t_s, we maintain the following data structure for enumerating all temporal k-cores starting from t_s. The data structure, denoted by $\mathcal{L}_{t_s}$, includes all minimal core windows with active times (Definition 7) not later than t_s. To improve the enumeration efficiency, we arrange all windows in $\mathcal{L}_{t_s}$ by ascending order of end times.

Example 7. *Given the temporal graph G in Figure 1 and the minimal core windows of all edges for $k = 2$ in Table 2, Figure 5(a) shows the data structure $\mathcal{L}_{t_s}$ for $t_s = 1$ used in our algorithm. All minimal core windows with activate times not later than 1 are included in the structure and arranged in a ascending order of end times. Specifically, the minimal core windows $[2, 3]$ of (v_1, v_4), (v_1, v_2) and (v_2, v_4) are ranked as the first three windows in $\mathcal{L}_1$ since they have the earliest end time of 3 among all windows in $\mathcal{L}_1$. The minimal core window $[1, 4]$ of edges (v_2, v_9), (v_2, v_3) and (v_3, v_9) are ranked as the fourth to sixth windows in $\mathcal{L}_1$, as their end times are the second smallest. Note that each edge has at most one minimal core window in $\mathcal{L}_1$.*

Given the structure $\mathcal{L}_{t_s}$, we present the algorithm to enumerate all temporal k-cores starting from t_s in Algorithm 4. We will introduce how $\mathcal{L}_{t_s}$ is precomputed and maintained for different start times in Section 5.4. Given a minimal core window w, $w.start$ and $w.end$ denote the start time and the end time of w, respectively. $w.next$ denotes the next window of w in $\mathcal{L}_{t_s}$, and $w.edge$ denotes the corresponding edge of the window. We iteratively scan windows in $\mathcal{L}_{t_s}$ and add the corresponding edge into the edge set (line 5). We use a flag *valid* to check the condition in Lemma 6. Suppose window w^i is the first window we visit with star time $t_1 = t_s$. We can update the flag to True as the condition Lemma 6 holds. If the flag is False in line 7, we do not output the current result. The condition in line 10 means we have more windows with the same end time. We continue to the next one

Algorithm 4: AS-Output

Input: t_s, $\mathcal{L}_{t_s}$
Output: distinct temporal cores for a specific start time t_s

```
1  w ← the first window in L_{t_s};
2  R ← ∅;
3  valid ← False;
4  while w ≠ Null do
5      R ← R ∪ {w.edge};
6      if w.start = t_s then valid ← True;
7      if valid = False then
8          w ← w.next;
9          continue;
10     if w.next ≠ Null ∧ w.end = w.next.end then
11         w ← w.next;
12         continue;
13     output R;
14     w ← w.next;
```

(line 11) and only output the result for the last window with the same end time (line 13).

Example 8. *For the temporal graph in Figure 1, we enumerate all temporal 2-cores with $t_s = 1$ by scanning $\mathcal{L}_1$ (Figure 5(a)). The first three minimal core windows $[2, 3]$ have start times not equal to t_s, so their edges are added to R without output, and the flag valid remains False. The next three windows $[1, 4]$ have start times of 1; their edges are added to the edge-set, which is output as a result with $TTI = [1, 4]$, setting valid to* True *for the remainder of the iteration. Each subsequent matching window updates the edge-set, outputting distinct results. For $t_s = 2$ (Figure 5(b)), we scan $\mathcal{L}_2$. The first three windows meet the condition, and their edge-set is output, representing the subgraph skipped in the previous round ($t_s = 1$). The scan continues until the end of $\mathcal{L}_2$.*

Lemma 8. *Let $|\mathcal{R}_{t_s}|$ be the total size of all temporal k-cores starting from t_s. Given the data structure $\mathcal{L}_{t_s}$, the time complexity of Algorithm 4 is $O(|\mathcal{R}_{t_s}|)$.*

5.4 Enumeration for All Start Times

We now study how to enumerate temporal k-cores for all start times. This is achieved by maintaining the data structure $\mathcal{L}_{t_s}$ for different start times t_s. Recall that we need to maintain a set of minimal core windows in ascending order of end times in Section 5.3. Assume we increase the start time from t_s to $t_s + 1$. We expect to remove all minimal core windows with the start time t_s from the order and add all minimal core windows with the active time $t_s + 1$ to the order. To this end, we implement the order for each start time as a doubly linked list and present our final algorithm for temporal k-core enumeration in Algorithm 5.

Algorithm 5 takes the $\mathcal{ECS}$ computed by Algorithm 2 as input. We first compute the active time of each minimal core window in lines 1-4. Recall that the minimal core windows of each edge are computed by increasing the start time in Section 5.1. The windows returned by Algorithm 2 are naturally arranged chronologically. This enables us to sequentially scan minimal core windows of each edge and derive the active time for each window. The time complexity of this process is $O(|\mathcal{ECS}|)$.

After computing the active time, B_a (line 10) and B_s (line 11) are used to collect windows for each active time and start time,

Algorithm 5: Enum

Input: a temporal graph G, an integer k, a time range $[T_s, T_e]$, the $\mathcal{ECS}$ of G for k
Output: all distinct temporal k-cores in $[T_s, T_e]$

```
1  foreach edge e in ECS do
2      foreach 0 ≤ i < |ECS(e)| do
           /* compute active time                    */
3          if i = 0 then ECS(e)[0].active ← T_s;
4          else
               ECS(e)[i].active ← ECS(e)[i − 1].start + 1;
5  foreach T_s ≤ t ≤ T_e do
6      B_a[t] ← ∅;
7      B_s[t] ← ∅;
8  sort labels of ECS in ascending order of end times;
9  foreach w ∈ ECS do
10     B_a[w.active].push(w);
11     B_s[w.start].push(w);
12 L ← an empty doubly linked list;
13 foreach T_s ≤ t ≤ T_e do
14     if t > T_s then
15         foreach w ∈ B_s[t − 1] do
16             Delete(w);
17     h ← the dummy head node for L;
18     foreach w ∈ B_a[t] do
19         while h.next ≠ Null ∧ h.next < w do
20             h ← h.next;
21         Insert(w, h, h.next);
22         h ← w;
23     if B_s[t] = ∅ then continue;
24     AS-Output(L, t);
25 Procedure Delete(w)
26     w.pre.next ← w.next;
27     if w.next ≠ Null then w.next.pre ← w.pre;
28 Procedure Insert(w, a, b)
29     w.next ← b;
30     w.pre ← a;
31     a.next ← w;
32     if b ≠ Null then b.pre ← w;
```

respectively. Lines 8–11 arrange all windows in a certain order in B_a and B_s. The structure for each time key in B_a and B_s is an array, and the *push* function adds the item to the end of the array.

Given a window w, we use $w.next$ and $w.pre$ to denote the next item and the previous item in the doubly linked list, respectively. Delete and Insert operators are used to insert an item and delete an item in the doubly linked list, respectively. When increasing t, we remove all windows with the start time $t - 1$ from the doubly linked list in lines 14–16 because the windows cannot exist given the new start time $t + 1$. Removing those windows is safe because they are never considered in the temporal k-cores given the increase of the start time. We add all windows with the active time t to the order in lines 18–22. Given that all windows in $B_a[t]$ are in ascending order of end times, we search the doubly linked list in a single direction and start from the ending item h

for the w in the previous iteration of line 17. $B_s[t] = \emptyset$ in line 23 means no minimal core window exists with the start time t. No temporal k-core exists starting from t based on Lemma 4. We invoke AS-Output at the end to output all temporal k-cores with a start time of t_s.

Example 9. *For the temporal graph in Figure 1 and its $\mathcal{ECS}$ for $k = 2$ in Table 2, we enumerate all temporal 2-cores for the query time range $[1, 6]$. Starting with $t_s = 1$, we insert relevant minimal core windows into $\mathcal{L}_1$ (Figure 5(a)) and apply Algorithm 4 to enumerate temporal 2-cores. For $t_s = 2$, minimal core windows with start times less than 2, such as (v_2, v_9), (v_2, v_3) and (v_3, v_9), are removed from $\mathcal{L}_1$. This is done efficiently by linking windows in constant time. Next, windows with an activation time of 2, like $[2, 6]$ for (v_2, v_3), are inserted into $\mathcal{L}$ based on their position, forming $\mathcal{L}_2$. Algorithm 4 is then used to enumerate temporal 2-cores for $t_s = 2$. This process repeats for all t_s values until $t_s = T_e$.*

Theorem 3. *The time complexity of Algorithm 5 is $O(|\mathcal{R}|)$, where $|\mathcal{R}|$ represents the total size of all resulting temporal k-cores.*

Proof. The computation of active times (lines 1-4) and the initialization of the buckets (lines 5 - 11) takes $O(|\mathcal{ECS}|)$, which is bounded by $|\mathcal{R}|$. For each start time t_s in the main process, the deletion process (lines 14 - 16) takes $O(|B_s(t_s)|)$, which is bounded by $O(|\mathcal{L}_{t_s-1}|)$; the insertion process (lines 18 - 22) takes $O(|\mathcal{L}_{t_s}|)$; line 24 (AS-Output) takes $O(|\mathcal{L}_{t_s}|)$. Let $|\mathcal{R}_{t_s}|$ denote the total size of the set of temporal k-cores for a specific start time t_s. Since $O(|\mathcal{L}_{t_s}|)$ is bounded by $O(|\mathcal{R}_{t_s}|)$, the entire main process (lines 13-24) has a time complexity of $O(|\Sigma_{t_s \in [1, t_{max}]} \mathcal{R}_{t_s}|) = O(|\mathcal{R}|)$. Therefore, the time complexity of the algorithm is $O(|\mathcal{R}|)$. □

Based on Theorem 1 and Theorem 3, the total time complexity of Algorithm 2 and Algorithm 5 is $O(|\mathcal{VCT}| \cdot deg_{avg} + |\mathcal{R}|)$.

6 Experiments

In this section, we conduct experiments to evaluate the effectiveness and efficiency of our proposed algorithm. We run all experiments on a Linux machine with Intel i9-12900K CPU and 500GB RAM. All algorithms are implemented in C++ and compiled using g++ compiler at -O3 optimization level. Implementation of OTCD is provided by the author of [25].

Datasets. We use fourteen real-world temporal graphs of varying sizes for our experiments. Table 3 presents their basic statistics, where t_{max} is the number of distinct timestamps in the graph, and k_{max} is the maximum core number among all vertices. These datasets are sourced from SNAP [2] and the KONECT Project [1].

Parameters. We vary the size of the query time range to 5%, 10%, 20%, and 40% of t_{max}, with 10% as the default. For the integer k, we vary it from 10%, 20%, 30%, and 40% of k_{max}, using 30% as the default. We randomly select 100 query time ranges and record the average running time. Each query time range is guaranteed to contain at least one temporal k-core. We set the time limit to 6 hours, which is way beyond the maximum running time required for our final algorithm. For configurations that exceeded the timeout threshold (set to 6 hours), the corresponding points in Figures 7 and 8 are shown at the upper boundary of the plot (i.e., clipped at the ceiling of the y-axis). The time points used to construct the query ranges are selected directly from the raw timestamps in the dataset, without refining or resampling their granularity. These ranges may be overlapping or disjoint, as we do not impose any artificial constraints, in order to better reflect realistic and diverse query scenarios.

Table 3: Datasets.

Name	$\|V\|$	$\|E\|$	t_{max}	k_{max}
FB-Forum (FB)	899	33,786	33,482	19
BitcoinOtc (BO)	5,881	35,592	35,444	21
CollegeMsg (CM)	1,899	59,835	58,911	20
Email (EM)	986	332,334	207,880	34
Mooc (MC)	7,143	411,749	345,600	76
MathOverflow (MO)	24,818	506,550	505,784	78
AskUbuntu (AU)	159,316	964,437	960,866	48
Lkml-reply (LR)	63,399	1,096,440	881,701	91
Enron (ER)	87,273	1,148,072	220,364	53
SuperUser (SU)	194,085	1,443,339	1,437,199	61
WikiTalk (WT)	1,219,241	2,284,546	1,956,001	68
Wikipedia (WK)	91,340	2,435,731	4,518	117
ProsperLoans (PL)	89,269	3,394,979	1,259	111
Youtube (YT)	3,223,589	9,375,374	203	88

6.1 Efficiency

We compare the efficiency of our final algorithm with the baseline and the state-of-the-art OTCD algorithm, as shown in Figure 6. The results demonstrate significant performance differences across all datasets. OTCD fails to complete within the time limit on five datasets with many edges and distinct timestamps, while the baseline completes only on smaller datasets or those with fewer timestamps. In contrast, our final algorithm consistently outperforms both, completing all datasets with a 2-4 order of magnitude improvement. For example, on FB, BO, and CM, it surpasses OTCD by 2-3 orders and the baseline by 3-4 orders, with even greater gains of up to 4 orders on EM, MC, and MF. Interestingly, datasets like WK, PL, and YT, despite their large edge counts, have shorter execution times due to fewer distinct timestamps. For instance, YT, with only 20 distinct timestamps at 10% t_{max}, shows less drastic performance differences, but our algorithm still outperforms OTCD by an order of magnitude. This superior performance is due to our algorithm's efficient design, which eliminates redundant result verification across timestamps, streamlining computation. It is worth noting that the precomputation cost (computing core times, shown in blue in Figure 6) is identical for the baseline and the advanced algorithm. On datasets with many distinct timestamps, the precomputation cost is relatively small, contributing only 1-10% of the advanced algorithm's total runtime. However, for datasets with fewer timestamps, such as WK, PL, and YT, the precomputation step constitutes a larger fraction of the total runtime, reflecting the reduced time spent on enumeration in these cases.

Varying k. We evaluated the impact of varying k on four representative datasets: CollegeMsg and Email (small), WikiTalk (large with many timestamps), and Prosper (large with few timestamps). We varied k from 10% to 40% of $k_{\max}$, as shown in Figure 7. For CollegeMsg, Email, and WikiTalk, the running time consistently decreases as k increases, since larger k values yield fewer temporal k-cores. In particular, the running time at 40% of $k_{\max}$ is about one-tenth of that at 10% for CollegeMsg and WikiTalk, and nearly two orders of magnitude smaller for Email. In contrast, Prosper exhibits relatively stable running time across different k values, with only about a 30% reduction, due to its small number

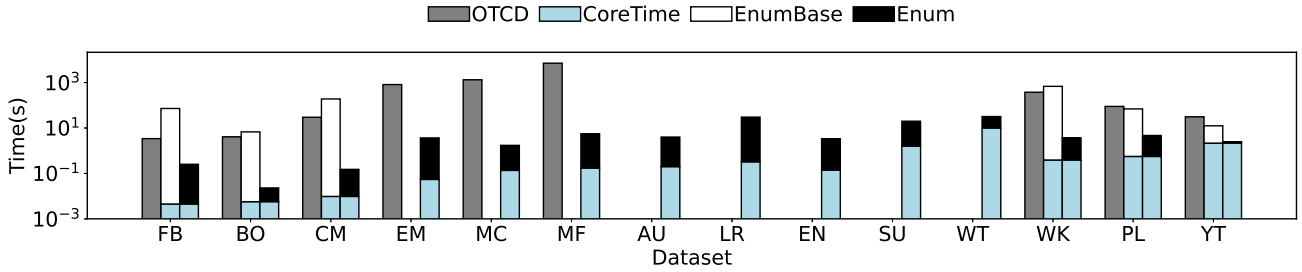
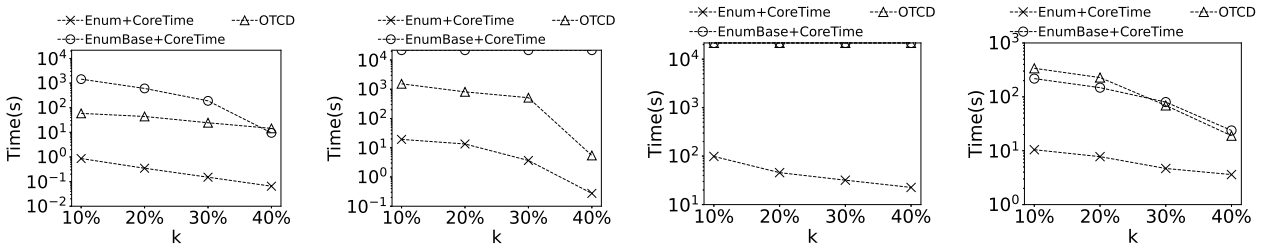
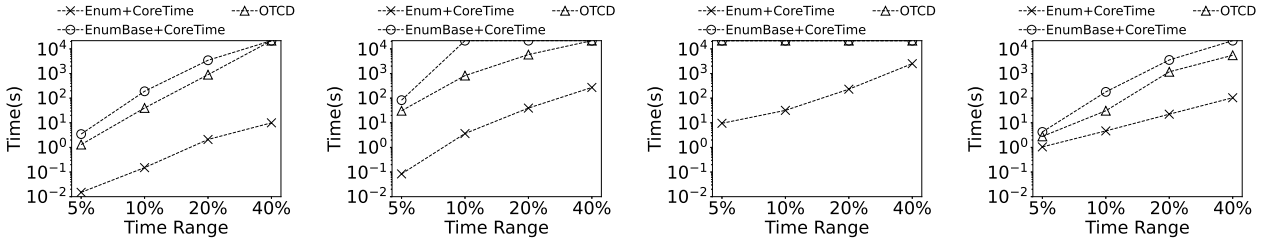


Figure 6: Average running time for 30% k_{max} on query time ranges of 10% t_{max}.



(a) CollegeMsg (b) Email (c) Wikitalk (d) Prosper

Figure 7: The average running time on varying k between 10%, 20%, 30% and 40% of k_{max}.



(a) CollegeMsg (b) Email (c) Wikitalk (d) Prosper

Figure 8: The average running time on varying query time range between 5%, 10%, 20% and 40% of t_{max}.

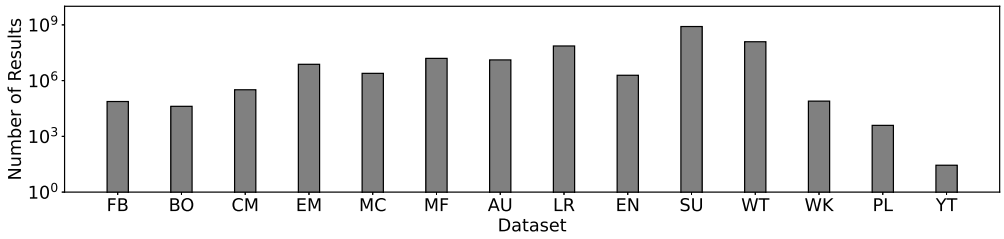
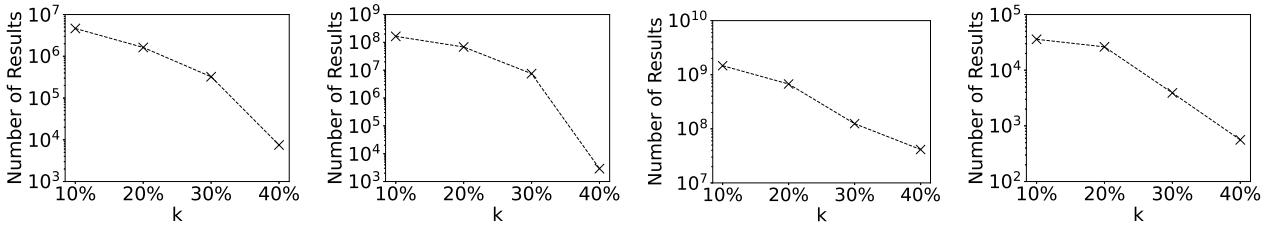


Figure 9: The average number of temporal k-cores for each dataset run with the default parameters.

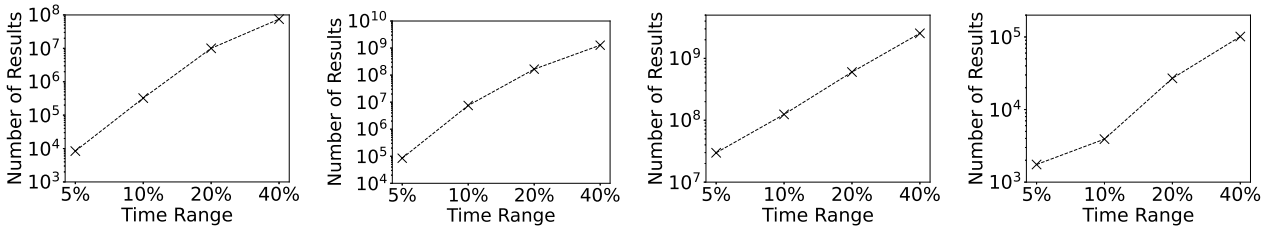
of timestamps and dense temporal k-cores. OTCD and the baseline show similar trends, while our final algorithm consistently benefits from increasing k.

Varying time range. We varied the query time range from 5% to 40% of $t_{\max}$ on four datasets, with results shown in Figure 8. As expected, the running time increases rapidly as the time range expands, due to the growth in the number and size of output results. Across all datasets, the running time increases by 2–3 orders of magnitude when the range grows from 5% to 40% of $t_{\max}$. In particular, doubling the time range leads to about a 5× increase for Prosper and nearly a 10× increase for WikiTalk, while Email shows an even sharper increase at small ranges. OTCD exhibits the same trend but fails to scale: it times out on CollegeMsg at 40%, on Email beyond 5%, and cannot run on WikiTalk for any range. In contrast, our final algorithm consistently handles larger time ranges, demonstrating superior scalability in large temporal graphs.



(a) CollegeMsg (b) Email (c) Wikitalk (d) Prosper

Figure 10: The average number of temporal k-cores on varying k between 10%, 20%, 30% and 40% of k_{max}.



(a) CollegeMsg (b) Email (c) Wikitalk (d) Prosper

Figure 11: The average number of temporal k-cores on varying time range between 5%, 10%, 20% and 40% of t_{max}.

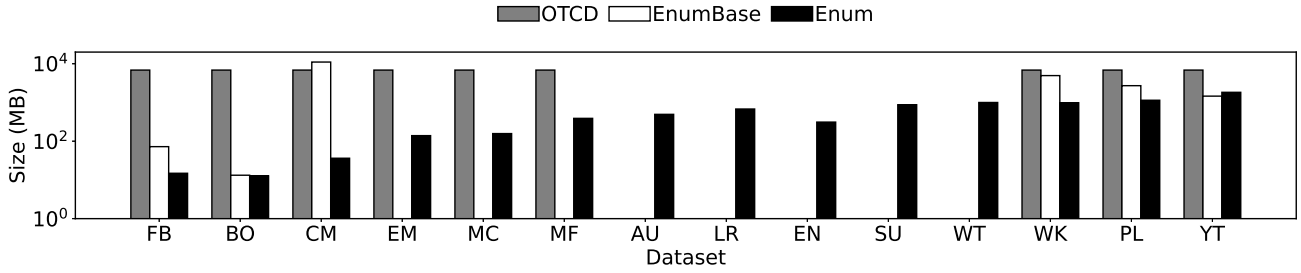


Figure 12: The maximum residential memory of each algorithm run on the default parameter.

6.2 Number of Temporal k-cores

Figure 9 shows the average number of temporal k-cores across datasets using default parameters. Datasets with more distinct timestamps, such as SU and WT, generate the most k-cores, while WK, PL, and YT, despite their large edge counts, produce fewer due to limited timestamps. Figure 10 illustrates the effect of varying k (10%-40% of k_{max}): the number of k-cores decreases as k increases. For CollegeMsg and Email, results drop by 3-4 orders of magnitude from 30% to 40%, while Wikitalk and Prosper show a 2-order reduction from 10% to 40%. Figure 11 highlights the impact of query time range (5%-40% of t_{max}): as the range grows, k-cores increase significantly, doubling by 2 orders of magnitude for CollegeMsg, Email, Wikitalk, and Prosper as the range expands.

6.3 Memory Overhead

Figure 12 compares the memory usage of each algorithm with default parameters. OTCD consistently incurs a memory cost of 7 GB on datasets it completes, while the baseline consumes even more due to storing previously generated temporal k-cores for comparisons. In contrast, our final algorithm uses significantly less memory across all datasets, maintaining a footprint of under 2 GB, except for YT, which has very few distinct timestamps. This efficiency is achieved by avoiding the storage of previously generated k-cores or maintaining them as subgraphs, unlike OTCD. Notably, datasets like WK, PL, and YT show the highest memory overhead despite having fewer distinct k-cores, due to their denser cores with many edges sharing the same timestamps.

7 Conclusion

We propose an efficient framework for enumerating temporal k-cores within a given time range of temporal graphs. By leveraging core times and introducing minimal core windows for edges, our approach achieves theoretical optimal enumeration time and significantly outperforms existing methods by up to two orders of magnitude across diverse datasets.

Artifacts

The source code and artifacts are available at: https://github.com/johnmzz/temporal_kcore. Datasets are publicly available at: http://konect.cc/ and https://snap.stanford.edu/.

Acknowledgments

Dong Wen is supported by ARC DP230101445 and DE240100668. Wenjie Zhang is supported by ARC DP230101445 and FT210100303.

References

[1] [n. d.]. The KONECT Project. http://konect.cc/
[2] [n. d.]. SNAP: Stanford Network Analysis Project. https://snap.stanford.edu/
[3] Wen Bai, Yadi Chen, and Di Wu. 2020. Efficient temporal core maintenance of massive graphs. *Information Sciences* 513 (3 2020), 324–340. doi:10.1016/J.INS.2019.11.003
[4] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Ozsu. 2011. Efficient core decomposition in massive networks. *Proceedings - International Conference on Data Engineering* (2011), 51–62. doi:10.1109/ICDE.2011.5767911
[5] Lingyang Chu, Yanyan Zhang, Yu Yang, Lanjun Wang, and Jian Pei. 2019. Online density bursting subgraph detection from temporal graphs. *Proceedings of the VLDB Endowment* 12 (9 2019), 2353–2365. Issue 13. doi:10.14778/3358701.3358704
[6] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2014), 991–1002. doi:10.1145/2588555.2612179
[7] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (maximal) Span-cores from Temporal Networks. *International Conference on Information and Knowledge Management, Proceedings* (10 2018), 107–116. doi:10.1145/3269206.3271767
[8] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9 (9 2015), 13–23. Issue 1. doi:10.14778/2850469.2850471
[9] Rong Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent community search in temporal networks. *Proceedings - IEEE 34th International Conference on Data Engineering, ICDE 2018* (10 2018), 797–808. doi:10.1109/ICDE.2018.00077
[10] Yuan Li, Jinsheng Liu, Huiqun Zhao, Jing Sun, Yuhai Zhao, and Guoren Wang. 2021. Efficient continual cohesive subgraph search in large temporal graphs. *World Wide Web* 24 (9 2021), 1483–1509. Issue 5. doi:10.1007/S11280-021-00917-Z/METRICS
[11] Longlong Lin, Pingpeng Yuan, Rong Hua Li, Chunxue Zhu, Hongchao Qin, Hai Jin, and Tao Jia. 2024. QTCS: Efficient Query-Centered Temporal Community Search. *Proceedings of the VLDB Endowment* 17 (2024), 1187–1199. Issue 6. doi:10.14778/3648160.3648163
[12] Tianrui Liu, Qi Cai, Changxin Xu, Bo Hong, Fanghao Ni, Yuxin Qiao, and Tsungwei Yang. 2024. Rumor Detection with a novel graph neural network approach. (3 2024). https://arxiv.org/abs/2403.16206v3
[13] Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, and Jinpeng Huai. 2020. An Efficient Approach to Finding Dense Temporal Subgraphs. *IEEE Transactions on Knowledge and Data Engineering* 32 (4 2020), 645–658. Issue 04. doi:10.1109/TKDE.2019.2891604
[14] Jonson Manurung, Poltak Sihombing, Mohammad Andri Budiman, and Sawaluddin. 2023. Dynamic Rumor Control in Social Networks Using Temporal Graph Neural Networks. *2023 IEEE International Conference of Computer Science and Information Technology: The Role of Artificial Intelligence Technology in Human and Computer Interactions in the Industrial Era 5.0, ICOSNIKOM 2023* (2023). doi:10.1109/ICOSNIKOM60230.2023.10364382
[15] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2011. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems* 24 (3 2011), 288–300. Issue 2. doi:10.1109/TPDS.2012.124
[16] Hung Nguyen, Thin Nguyen, and Duc Thanh Nguyen. 2021. A graph-based approach for population health analysis using Geo-tagged tweets. *Multimedia Tools and Applications* 80 (2 2021), 7187–7204. Issue 5. doi:10.1007/S11042-020-10034-0/TABLES/4
[17] Lutz Oettershagen, Athanasios L. Konstantinidis, and Giuseppe F. Italiano. 2023. Temporal Network Core Decomposition and Community Search. (9 2023). https://arxiv.org/abs/2309.11843v1
[18] Hongchao Qin, Rong Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. 2019. Mining periodic cliques in temporal networks. *Proceedings - International Conference on Data Engineering* 2019-April (4 2019), 1130–1141. doi:10.1109/ICDE.2019.00104
[19] Hongchao Qin, Rong Hua Li, Ye Yuan, Guoren Wang, Weihua Yang, and Lu Qin. 2022. Periodic Communities Mining in Temporal Networks: Concepts and Algorithms. *IEEE Transactions on Knowledge and Data Engineering* 34 (8 2022), 3927–3945. Issue 8. doi:10.1109/TKDE.2020.3028025
[20] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5 (9 1983), 269–287. Issue 3. doi:10.1016/0378-8733(83)90028-X
[21] Matteo Serafino, Higor S. Monteiro, Shaojun Luo, Saulo D.S. Reis, Carles Igual, Antonio S.Lima Neto, Matías Travizano, José S. Andrade, and Hernán A. Makse. 2021. Superspreading k-cores at the center of COVID-19 pandemic persistence. *PLoS Computational Biology* 18 (3 2021). Issue 4. doi:10.1371/journal.pcbi.1009865
[22] Michele Starnini, Charalampos E. Tsourakakis, Maryam Zamanipour, André Panisson, Walter Allasia, Marco Fornasiero, Laura Li Puma, Valeria Ricci, Silvia Ronchiadin, Angela Ugrinoska, Marco Varetto, and Dario Moncalvo. 2021. Smurf-Based Anti-money Laundering in Time-Evolving Transaction Networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12978 LNAI (2021), 171–186. doi:10.1007/978-3-030-86514-6_11
[23] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2015. I/O Efficient Core Graph Decomposition at Web Scale. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016* (11 2015), 133–144. doi:10.1109/ICDE.2016.7498235
[24] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. *2015 IEEE International Conference on Big Data (Big Data)* (10 2015), 649–658. doi:10.1109/BIGDATA.2015.7363809
[25] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2023. Scalable Time-Range k-Core Query on Temporal Graphs. *Proceedings of the VLDB Endowment* 16 (1 2023), 1168–1180. Issue 5. doi:10.14778/3579075.3579089
[26] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical K-Cores. *Proc. VLDB Endow.* 14 (2021), 2033–2045. Issue 11.
[27] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *Journal of Supercomputing* 53 (8 2010), 352–369. Issue 2. doi:10.1007/S11227-009-0299-0