KWIQ: Answering k-core Window Queries in Temporal Networks

Mahdihusain Momin
Indian Institute of Technology Delhi
New Delhi, India
momin@alumni.iitd.ac.in

Raj Kamal*
Indian Institute of Technology Delhi
New Delhi, India
csz188013@cse.iitd.ac.in

Shantwana Dixit
Indian Institute of Technology Delhi
New Delhi, India
xtshantwana78d@gmail.com

Sayan Ranu
Indian Institute of Technology Delhi
New Delhi, India
sayanranu@cse.iitd.ac.in

Amitabha Bagchi
Indian Institute of Technology Delhi
New Delhi, India
bagchi@cse.iitd.ac.in

ABSTRACT

Understanding the evolution of communities and the factors that contribute to their development, stability and disappearance over time is a fundamental problem in the study of temporal networks. The concept of k-core is one of the most popular metrics to detect communities. Since the k-core of a temporal network changes with time, an important question arises: *Are there nodes that always remain within the k-core?* In this paper, we explore this question by introducing the notion of *core-invariant nodes*. Given a temporal window Δ and a parameter $\mathcal{K}$, the core-invariant nodes are those that are part of the $\mathcal{K}$-core throughout Δ. Core-invariant nodes have been shown to dictate the stability of networks, while being also useful in detecting anomalous behavior. The complexity of finding core-invariant nodes is $O(|\Delta| \times |E|)$, which is exorbitantly high for million-scale networks. We overcome this computational bottleneck by designing an algorithm called KWIQ. KWIQ efficiently processes the cascading impact of network updates through a novel data structure called orientation graph. Through extensive experiments on real temporal networks containing millions of nodes, we establish that the proposed pruning strategies are more than 5 times faster than baseline strategies.

1 INTRODUCTION AND RELATED WORK

Several important real-world settings involve entities interacting at different points of time for different periods of time, e.g., online social networks [14], email collections, communication forums like Stack Exchange [28], call data records (CDR) [16], research collaboration networks among others. Temporal networks provide an abstraction to capture the time-varying nature of these interactions by storing interactions between entities (nodes) as time-stamped edges. The temporal network abstraction allows us to study time-varying activity in these settings, especially transient activity. Of particular interest in recent times is the formation of transient communities within temporal networks, a prominent example being Facebook's recent attempt to define and combat "coordinated inauthentic behaviour" [13], and the detection of similar issues on Twitter [27]. In this paper, we focus on the k-core [6] of a temporal network. For static networks the concept of k-core provides a useful abstraction of a community: A k-core is defined as a set of nodes that induce a largest subgraph of minimum degree k within the network. We extend this definition to temporal graphs and show how to efficiently track nodes that remain in the k-core over a period of time. We refer to such nodes as *core-invariant* nodes. More formally, *given a time window $[t_s, t_e]$ and a threshold $\mathcal{K}$, we want to identify all nodes with a core value of at least $\mathcal{K}$ throughout the given time window.* The parameter $\mathcal{K}$ allows us to focus only on communities that have core value above a given threshold. We refer to this threshold as the *cohesiveness* of this community.

1.1 Motivating Applications

Identification of core-invariant nodes is important for identifying coordinated inauthentic behavior in online social networks as Shao. et. al. [23] showed recently in a study on a Twitter data set collected during the 2016 US Presidential election. They observed that the instances of forwarding of inauthentic content went up sharply for subgraphs with high core values. However, their experiment flattens six months of data into a single network, which is not realistic since collusion over particular pieces of inauthentic content is expected to happen in the timescale of hours and days. Approximation to the k-core is defined and analyzed in [9] for MapReduce and streaming models. We refine the experiment of Shao. et. al. [23] by creating a temporal graph of their data set by removing interactions after 15 days and studying core-invariant nodes for different values of $\mathcal{K}$. Just like Shao et. al. [23], we find that inauthentic content was being shared at a *much* higher rate within core-invariant sets of nodes, with authentic content basically disappearing at a cohesiveness of 7, which is much lower than the core value of 20 that Shao et. al. [23] observe in the 6-month network. This experiment is presented in detail in §4.5, and we wish to make two points by presenting it: (1) the core-invariant definition is the correct definition for a temporal refinement of a real problem that researchers are already addressing and (2) the fact that our cohesiveness threshold for inauthentic behaviour is much lower than the core values of the inauthentic core discovered in the flattened graph shows that our definition of cohesiveness gives a better handle on the extent of the time-varying collusion taking place in inauthentic networks.

In a completely different application domain, a recent paper in Nature showed that core-invariant nodes dictate the stability of mutualistic ecosystems [20].

1.2 Existing Works and Baselines

Baseline approach: Currently, no algorithm exists to identify core-invariant nodes efficiently. The process of computing the core value of all nodes in a graph is called *core decomposition*[19].

*Raj Kamal is a Venkat Padmanabhan Doctoral Fellow.

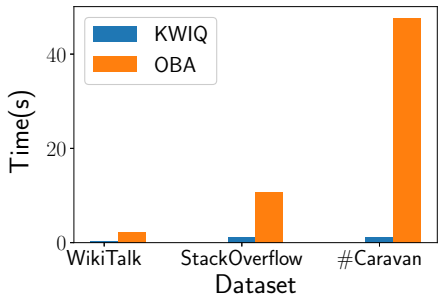


Figure 1: Efficiency comparison of KWIQ with OBA [29].

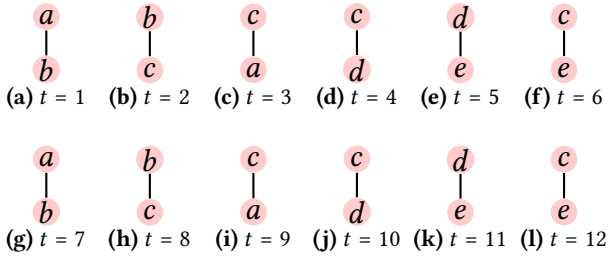


Figure 2: Edges added at different timestamps t.

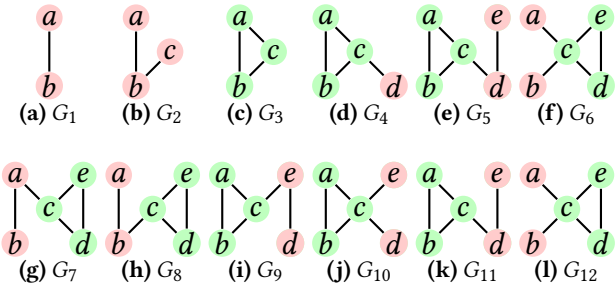


Figure 3: The structure of the temporal graph corresponding to the edge stream in Fig. 2. G_t denotes the graph at timestamp t. We assume an edge gets deleted after 5 timestamps since its arrival. Nodes in 2-core are in green.

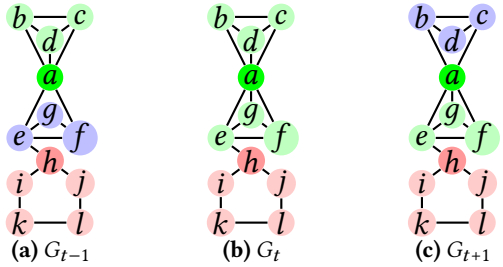
Core decomposition of a graph $G = (V, E)$ can be performed in time linear in the number of edges, i.e., $\Theta(|E|)$. The baseline approach is therefore to perform core decomposition following each event within the query time window $[t_s, t_e]$ and identify the core-invariant nodes that remain within the k-core that is recomputed at each point. Assuming the size of the graph remains stable, we consume $O((t_e - t_s) \times |E|)$ time. Both $(t_e - t_s)$ and $|E|$ can be extremely large. For example, Twitter contains billions of edges, and the granularity of timestamps is in seconds. Hence, the baseline algorithm is not scalable.

Core-Maintenance: A better alternative is to perform core maintenance by treating the events from t_s to t_e as a data stream [18, 21, 29]. Core maintenance algorithms track the core values of all nodes in the time window from t_s to t_e and then identify those that remain above the threshold $\mathcal{K}$ throughout $[t_s, t_e]$. However, even core-maintenance leads to inefficient querying since it is not specifically tailored for the task of querying core-invariant nodes. In this paper, we leverage key properties of core-invariant nodes to develop an efficient algorithm called *KWIQ: k-core Window Queries on temporal networks*. As evidence of the gain in efficiency, in Fig. 1, we show the running time of KWIQ against the fastest core-maintenance algorithm, OBA[29], for querying core-invariant nodes in three real datasets (See §4.1 for details). As visible, KWIQ is ≥ 5 times faster.

Persistent Cores: A recent work identifies *persistent cores* in a temporal network[17]. Given a cohesiveness threshold $\mathcal{K}$, a specified length τ of time duration and a positive integer θ. A (θ, τ)-persistent $\mathcal{K}$-core is the largest induced subgraph which is $\mathcal{K}$-core projected over a period of length θ, and total non-overlapped length of these θ length intervals is at least τ. The vertex set of the persistent core remains fixed. But the edge set of the persistent core may change as the θ length interval shifts, and this edge set consists of all edges being present at any time point of the θ length interval. For $\theta = 0$, the persistent core is a largest induced subgraph which remains $\mathcal{K}$-core for a minimum duration τ. Owing to this formulation, persistent cores will miss out on identifying those nodes that are always part of the $\mathcal{K}$-core but the $\mathcal{K}$ neighbors that make them part of the $\mathcal{K}$-core change with time. Thus, the persistent core is a subgraph search problem, whereas ours is a node search problem. Identifying the persistent core is NP-hard [17] and the associated techniques do not apply to identifying the core-invariant nodes. Fig. 3 illustrates this difference with an actual example. Fig. 2 shows a stream of edges and Fig. 3 shows the corresponding graph. We assume that an edge remains "active" for a duration of 5 timestamps after which it disappears. For example, the $a - b$ edge at timestamp $t = 1$ remains in the graph till $t = 5$ (i.e., G_5). To give a semantic context, two users of a collaboration network may be considered connected if they have collaborated at least once within a year.

In the temporal graph of Fig. 3, consider the time window $[3, 12]$. For cohesiveness threshold $\mathcal{K} = 2$, node c remains within the $\mathcal{K}$-core throughout $[3, 12]$. For $\theta = 0$, $\mathcal{K} = 2$, and $\tau = 9$, the (θ, τ)-persistent $\mathcal{K}$-core is empty because there is no 2-core for a duration of length 9. For $\theta = 0$, $\mathcal{K} = 2$, and $\tau = 6$, the (θ, τ)-persistent $\mathcal{K}$-core is the subgraph induced by $\{a, b, c\}$ because this set $\{a, b, c\}$ induces a largest subgraph which remains 2-core during the time windows $[3, 5]$ and $[9, 11]$. Thus, to summarize, persistent core tracks persistent communities for a given time period. In contrast, core-invariant nodes tracks nodes that are always part of *some* dense community through the time window.

Span Cores: Galimberti et al. [10, 11] proposed the problem of identifying *span-cores*, wherein, given a cohesiveness threshold $\mathcal{K}$ and time interval $\Delta = [t_s, t_e]$, the span core is the largest subgraph in which every node has at least $\mathcal{K}$ neighbors formed by the *intersection* of all graphs in the temporal interval $\Delta = [t_s, t_e]$. Due to the enforcement of intersection, only those edges matter that exist throughout Δ, which is not the case in our problem. In other words, span core problem finds an induced subgraph which remains k-core throughout the querying time. However, invariant nodes are nodes which remain part of k-core at each time step throughout the querying time. Kindly note that the adjacent nodes, which make an invariant node part of the k-core, may change over time. However, these adjacent nodes are fixed in case of span-core throughout the querying time. Therefore, invariant node set is a superset of span core for given value of $\mathcal{K}$ and for given querying time. As result, both the span core, and the techniques used to compute the span core, cannot be used to solve the proposed problem. For example, consider Fig. 4. We set $\mathcal{K} = 3$. Graph G_t is obtained from graph G_{t-1} by addition of edge (a, g). Similarly, graph G_{t+1} is obtained from graph G_t by deletion of edge (a, d). Node a is the only invariant node because



(a) G_{t-1} (b) G_t (c) G_{t+1}

Figure 4: We set $\mathcal{K} = 3$. Graph G_t is obtained from graph G_{t-1} by addition of edge (a, g). Similarly, graph G_{t+1} is obtained from graph G_t by deletion of edge (a, d). Then, SNIN consists of red and pink vertices. SNIN′ consists of pink vertices. Node a is the only invariant node because this is the only node whose core value remains 3 for the time window $[t-1,\ t+1]$, so set of non-invariant nodes (NIN) consists of all vertices except vertex a. PIN consists of green nodes. Blue nodes are sandwich nodes (*WN*).

this is the only node whose core value remains 3 for the given time points $t-1,\ t,\ t+1$. However, span 3-core for the querying time $[t-1, t+1]$ is empty because there is no induced subgraph which remains 3-core at all the given time points $t-1,\ t,\ t+1$. Several applications of span-cores are outlined in [8], which also extend to the proposed problem of invariant cores.

1.3 Contributions

The key contributions of our work are as follows:

- We formulate the problem of querying *core-invariant nodes* on temporal networks (§2).
- To address scalability challenges we devise an efficient algorithm called *KWIQ*. The novelty of KWIQ comes from the fact that it partitions the entire node set into three parts for a given value of $\mathcal{K}$: (1) nodes that are guaranteed not to be core-invariant, (2) nodes that are highly likely to be core-invariant and (3) a set of *sandwich* nodes that are hard to categorise easily into the first two classes. KWIQ handles the first two categories in $O(1)$ time and applies *orientation* algorithm on the sandwich nodes to compute the answer set (§3) using a novel data structure called the *orientation graph*.
- Through empirical evaluation on several real-world temporal networks, with millions of nodes, hundreds of millions of edges, and thousands of timestamps, we establish that KWIQ is 5 times faster than the baseline approaches (§4).

2 PROBLEM FORMULATION

Our problem intakes three inputs: a temporal graph, a time window, and a *cohesiveness threshold* $\mathcal{K}$.

DEFINITION 1 (TEMPORAL GRAPH). *A temporal graph $G = (V, T, \tau)$ contains a set of nodes V, a discrete time domain $T = [0, 1, \cdots, t_{max}] \subseteq N$, and a function $\tau : V \times V \times T \rightarrow \{0, 1\}$ defining for each pair of nodes $u, v \in V$ and each timestamp $t \in T$ whether edge (u, v) exists at t. Thus, $E_t = \{(u, v, t) \mid \tau(u, v, t) = 1, t \in T\}$ denotes the set of edges existing at time t and $E = \cup_{\forall t \in T} E_t$ is the set of all temporal edges in the network.*

We use $G_t = (V, E_t)$ to denote the static snapshot graph at time t. Furthermore, $deg(u, G_t)$ denotes the degree of node u in the static graph G_t. We assume that the granularity of the timestamps is small enough to impose a total ordering among all network events, i.e., all events happen at distinct times. To ease the notational burden, in our definition of the temporal graph, the node set V remains static but the edge set changes from time instant to time instant. However, our formulation can easily be generalized to dynamic node sets.

Edge deletion model: Every interaction (i.e., edge) has an associated timestamp indicating when it is inserted. However, the lifespan of an edge is often not explicitly mentioned. For example, in a collaboration network like DBLP, two users are connected by an edge if they jointly co-author a paper. How long should this edge exist? It would be unfair to delete the edge in the very next timestamp since conferences often happen once a year. At the same time, the collaboration network should not keep two users connected if they last collaborated ten years ago. Similarly, in Twitter, two users may be considered connected if they have interacted recently (through @mention, @retweet, etc.). However, how do you quantify "recency"? To capture this aspect, we assume an interaction is *alive* for a duration of X timestamps after which it gets deleted; X is a dataset specific hyper-parameter and we call this the *deletion window*. To give an example, let the deletion window DW be 10 minutes. Now, suppose an interaction I_1 happens between users u and v at time t. To capture this event, we add an edge between u and v with time t. Next, assume these two users interact again at $t + 3$ minutes, and thus the timestamp of the edge between u and v gets updated to $t + 3$; note that we do not add a new edge. If no interactions happen for the next $DW = 10$ minutes, then the edge gets deleted at time $t + 13$.

DEFINITION 2 (k-CORE). *Given a static graph $G_t = (V, E)$ and a subset of nodes $U \subseteq V$, the* induced subgraph, *$G_t[U] \subseteq G_t$, induced by U is a subgraph with node set U and edges $E[U] = \{(u, v) \in E : u, v \in U\}$. An induced subgraph $G_t[U]$ of G_t is called a* k-core, *for $k \geq 0$, if for all $u \in U$, the degree of u in $G_t[U]$ is at least k and $G_t[U]$ is the maximal induced subgraph of G_t with this property.*

The *core value* of a node u in a static graph G, denoted as $core(u, G_t)$, is the largest k for which $u \in V_k$, where V_k is the node set of the k-core of G_t. We next extend the notion of k-core to temporal networks through *core-invariant nodes*.

DEFINITION 3 (CORE-INVARIANT NODES). *Given a temporal interval $\Delta = [t_s, t_e]$ and a positive integer $\mathcal{K}$, the core-invariant nodes $I_{\mathcal{K},\Delta} \subseteq V$, of a temporal graph $G = (V, T, \tau)$ is the set $I_{\mathcal{K},\Delta} = \{u \in V \mid \forall t \in \Delta,\ core(u, G_t) \geq \mathcal{K}\}$.*

We call a node *non-invariant* if it is *not* core-invariant.

EXAMPLE 1. *Consider the temporal graph G with timestamps $t = [1, 12]$ in Fig 3. For $k = 2$ and $\Delta_1 = [3, 12]$, the set of nodes that remain in 2-core (highlighted in green color) throughout Δ_1 is $\{c\}$. Hence, invariant-core I_{2,Δ_1} is $\{c\}$. c remains in 2-core throughout Δ_1 because node set $\{a, b\}$ contributes towards 2-core in time interval $[3, 5]$, $[9, 11]$ and node set $\{e, d\}$ contributes towards 2-core in time interval $[6, 8]$, $[12]$. In other words, there may be nodes in the graph that are not part of the final answer set, but contributes towards the answer set.*

PROBLEM 1 (TEMPORAL CORE-INVARIANT QUERIES). *Given a temporal graph $G = (V, T, \tau)$, cohesiveness threshold $\mathcal{K} > 0$ and a temporal span $\Delta = [t_s, t_e]$, find the core-invariant nodes $I_{\mathcal{K},\Delta}$.*

Table 1 summarizes the notations used in our paper.

Table 1: Notations used in the paper.

$G_t = (V, E_t)$	$\triangleq$	Graph with vertex set V and edge set E_t at time t
$\overrightarrow{G_t} = (V, \overrightarrow{E_t})$	$\triangleq$	Directed Graph with edge set $\overrightarrow{E_t}$ at time t
(u, v)	$\triangleq$	Edge between vertex u and v
$\langle u, v \rangle$	$\triangleq$	Directed edge from vertex u to v
$\mathcal{K}$	$\triangleq$	Threshold for cohesiveness
Δ	$\triangleq$	Temporal window
DW	$\triangleq$	Deletion window
$[t_s, t_e]$	$\triangleq$	t_s is start time and t_e is end time of Δ
$deg(u, G_t)$	$\triangleq$	Degree of vertex u in graph G_t
$core(u, G_t)$	$\triangleq$	Core value of vertex u in graph G_t
$I_{\mathcal{K},\Delta}$	$\triangleq$	Core-invariant nodes on given $\mathcal{K}$ and Δ
NIN	$\triangleq$	Non-invariant nodes
$SNIN$	$\triangleq$	Strongly non-invariant nodes
PIN	$\triangleq$	Potentially invariant nodes
WN	$\triangleq$	Sandwich nodes
$SNIN'$	$\triangleq$	Subset of SNIN obtained by using degree bound
$N_{\mathcal{K},t}(u)$	$\triangleq$	Neighbours of u with core value above $\mathcal{K}$ at time t
V_c	$\triangleq$	Set of vertices whose core values need to be updated post edge update
V_{can}	$\triangleq$	Set of vertices whose core values may get updated post edge update
V_{can}^E	$\triangleq$	V_{can} set after Expansion stage
V_{can}^S	$\triangleq$	V_{can} set after Shrinking stage
π	$\triangleq$	Degeneracy order
$u \preceq v$	$\triangleq$	Vertex v is reachable from vertex u

3 KWIQ: $\mathcal{K}$-CORE WINDOW QUERIES

KWIQ derives its computational efficiency by identifying four sets of nodes within the search space: non-invariant nodes (NIN), *strongly* non-invariant nodes ($SNIN$), potential core-invariant node (PIN), and *sandwich* nodes (WN). A toy example for these four sets is given in Fig. 4. Each edge update within Δ is processed based on the specific sets of nodes that it affects.

- **Strongly non-invariant nodes (SNIN):** $SNIN$ contains those nodes whose core values remain below the cohesiveness threshold $\mathcal{K}$ for the entire duration Δ. We will argue in §3.1 that all updates that have endpoint in this set can be *completely ignored*. While it is clear that $SNIN$ is a subset of the non-invariant nodes NIN, there appears to be a circularity here since to identify $SNIN$ we need to know the core value, which is what we are trying to compute. To get around this we actually identify a subset of nodes whose *degree* remains below $\mathcal{K}$ since the degree of a node is a natural upper bound on its core value. This subset can be easily identified in an initial pass over the edge updates and turns out to be a reasonably large fraction of $SNIN$.
- **Potential Invariant nodes (PIN):** PIN contains nodes that are highly *likely* to be part of the core-invariant nodes. These are nodes with high core values whose neighbours also have high core values. If an edge addition or deletion contains both end points from PIN, we show in §3.2 that this edge can be processed in $O(1)$ time.
- **Sandwich Nodes (WN):** WN contains non-invariant nodes that are neither in the strongly non-invariant set nor in the potential invariant set. More specifically, $WN = NIN \setminus \{PIN \cup SNIN\}$. Updates involving WN are the hardest to process, and we track their core values following each edge update.

For example, consider Fig. 4. We set $\mathcal{K} = 3$. Graph G_t is obtained from graph G_{t-1} by addition of edge (a, g). Similarly, graph G_{t+1} is obtained from graph G_t by deletion of edge (a, d). Then, $SNIN = \{h, i, j, k, l\}$ because core value of these nodes remains 2 for the given time points $t-1$, t, $t+1$, $SNIN' = \{i, j, k, l\}$ because degree of these nodes remains 2 for the given time points $t-1$, t, $t+1$. Node a is the only invariant node because this is the only node whose core value remains 3 for the given time points $t-1$, t, $t+1$, so set of non-invariant nodes is $NIN = \{b, c, d, , e, f, g, h, i, j, k, l\}$. We also have $PIN_{t-1} = \{a, b, c, d\}$, $PIN_t = \{a, b, c, d, e, f, g\}$. $PIN_{t+1} = \{a, e, f, g\}$. Therefore, $WN_{t-1} = \{e, f, g\}$, $WN_t = \phi$, $WN_{t+1} = \{b, c, d\}$ because $WN = NIN \setminus \{PIN \cup SNIN\}$.

With the proposed partitioning scheme, if $\mathcal{K}$ is high, then most nodes fall within $SNIN$, which allows us to ignore majority of the edge updates, resulting in fast querying times. On the other hand, if $\mathcal{K}$ is small, majority of nodes go to PIN, which facilitates faster processing of edge updates, and therefore not compromise on efficiency.

3.1 Identifying a large subset of SNIN

From the definition of k-core, for a node to have core value k, it must have at least k neighbors with core value k or higher. Therefore, k-core of a static graph is independent of all nodes with core value below k. Since node u remains below $\mathcal{K}$ throughout Δ, it never affects the $\mathcal{K}$-core at any time instant $t \in \Delta$, which implies that $I_{\mathcal{K},\Delta}$ is independent of u. We state this as a fact.

FACT 1. *If $\forall t \in \Delta$, $core(u, G_t) < \mathcal{K}$ for node $u \in V$, then $I_{\mathcal{K},\Delta}$ of $G(V, \Delta, \tau)$ is same as $I_{\mathcal{K},\Delta}$ on $G(V \setminus \{u\}, \Delta, \tau)$. Note that removing u from V means ignoring any edge update with one endpoint on u.*

Fact. 1 defines a strongly non-invariant node. The set $SNIN$ is therefore $SNIN = \{\forall t \in \Delta, core(u, G_t) < \mathcal{K} | u \in V\}$. Prop. 1 further implies:

FACT 2. *$I_{\mathcal{K},\Delta}$ of $G(V, \Delta, \tau)$ is same as $I_{\mathcal{K},\Delta}$ on $G(V \setminus SNIN, \Delta, \tau)$.*

If we can identify $SNIN$ quickly, then we can avoid all edge updates involving these nodes. However, we face a *circular dependency*. Specifically, we want to avoid computing core values of all nodes by applying Fact 2. However, Fact 2 itself requires us to compute the core values of all nodes as Fact 2 is dependent on computing $SNIN$. To remove this circular dependency, we compute a subset of SNIN using what we call *degree bound*.

FACT 3 (DEGREE BOUND). *Since $deg(u, G_t) \geq core(u, G_t)$, given query duration $\Delta = [t_s, t_e]$ and threshold $\mathcal{K}$, if $\forall t \in \Delta$, $deg(u, G_t) < \mathcal{K}$ then $u \in SNIN$.*

Fact 3 allows us to compute a subset $SNIN' \subseteq SNIN$ based on just the degree of a node. More specifically, given $\Delta = [t_s, t_e]$, we first extract the static graph G_{t_s}, which takes $O(|E_{t_s}|)$ time. Next, we iterate over each time $t \in \Delta$ and update the degree of the affected node based on the edge addition or deletion at time t. Updating the degree of a node consumes $O(1)$ time. Thus, the total complexity of applying degree bound and computing $SNIN'$ is $O(|E_{t_s}| + t_e - t_s)$. This running time is drastically smaller than the brute force approach of computing k-core at each timestamp, which takes $O(|E_{t_s}| + \sum_{t=t_s}^{t_e} |E_t|) \approx O(|E_{t_s}|(t_e - t_s))$ time assuming the size of the graph remains similar throughout Δ.

3.2 Identifying and Processing PIN

FACT 4. *Let $e = (u, v)$ be an edge insertion between two nodes u and v in graph G_t such that both u and v are in the $\mathcal{K}$-core of G_t. In this scenario, the $\mathcal{K}$-core of G_{t+1}, after e is inserted, remains identical to the $\mathcal{K}$-core of G_t.*

PROOF. Since an edge addition can only increase the core value of a node, nodes in the $\mathcal{K}$-core of G_t would continue to remain in the $\mathcal{K}$-core of G_{t+1} as well. Now, we need to show that nodes that are not part of the $\mathcal{K}$-core of G_t, are not in the $\mathcal{K}$-core of G_{t+1} either. Let $\mathbb{N} = \{n \in V \mid core(n, G_t) < \mathcal{K}\}$ be the nodes not in $\mathcal{K}$-core of G_t. The core value of a node $n \in \mathbb{N}$ can change from $\mathcal{K} - 1$ in G_t to $\mathcal{K}$ in G_{t+1} in two ways:

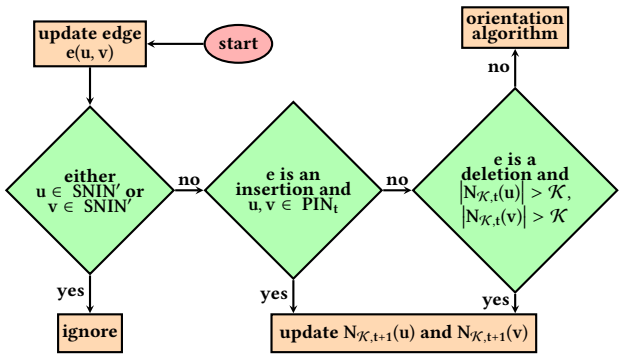


Figure 5: Flowchart to process edge updates in Kwiq where SNIN′ is the set of nodes whose degree remains below $\mathcal{K}$ throughout the query window, and PIN_t is the set of nodes which have at least $\mathcal{K}$ neighbors in $\mathcal{K}$-core at time t.

(1) **New neighbor:** A new edge connects n to a node with core value of at least $\mathcal{K}$
(2) **Cascading effect:** One of n's neighbors' core value changes from $\mathcal{K}-1$ to $\mathcal{K}$ and thus it has a cascading effect on n.

The first option is not possible since e connects two nodes with core values above $\mathcal{K}$. This, in turn, rules out the second possibility since a cascade can initiate only through a new neighbor. Thus, core value of all nodes not in $\mathcal{K}$-core of G_t remains the same in G_{t+1}. □

To leverage Fact 4, we introduce the notion of $\mathcal{K}$*-neighbors*, denoted as $N_{\mathcal{K},t}(u)$. Specifically, $N_{\mathcal{K},t}(u)$ includes all neighbors of u with core value of at least $\mathcal{K}$.

$$N_{\mathcal{K},t}(u) = \{v \mid core(v, G_t) \geq \mathcal{K},\ \tau(u, v, t) = 1\} \tag{1}$$

Based on $N_{\mathcal{K},t}(u)$, the set of potential core-invariant node at time t is defined as $PIN_t = \{u \in V \mid |N_{\mathcal{K},t}(u)| \geq \mathcal{K}\}$. All nodes in PIN_t are guaranteed to be in the $\mathcal{K}$-core of G_t. With this information, an edge update is processed as illustrated in Fig. 5.

- **Edge insertion:** Let $e = (u, v)$ be an edge that is inserted at time $t+1$. If both $u, v \in PIN_t$, then from Fact 4, the $\mathcal{K}$-core remains unchanged. Thus, we only update $N_{\mathcal{K},t+1}(u)$ and $N_{\mathcal{K},t+1}(v)$ by adding v and u to their respective neighborhoods in $O(1)$ time.
- **Edge deletion:** Let $e = (u, v)$ be an edge that is deleted at time $t+1$. If both $|N_{\mathcal{K},t}(u)| > \mathcal{K}$ and $|N_{\mathcal{K},t}(v)| > \mathcal{K}$, then we are guaranteed that the $\mathcal{K}$-core remains unchanged since both u and v continue to have at least $\mathcal{K}$ neighbors of core value higher than $\mathcal{K}$. Thus, we update $N_{\mathcal{K},t+1}(u)$ and $N_{\mathcal{K},t+1}(v)$ in $O(1)$ time.
- For updates violating the above criteria, and those specified by degree bound, we analyze their impact though the *orientation algorithm.*

3.3 Orientation Algorithm

The orientation algorithm is powered by the *orientation graph.* The orientation graph allows us to prune nodes that are guaranteed to not have been affected by an edge update.

3.3.1 The Orientation Graph. The orientation graph of an undirected graph $G_t = (V, E_t)$ is a directed graph $\overrightarrow{G_t} = (V, \overrightarrow{E_t})$ with the same vertex set. The direction of the orientation graph's edges stores information about the relative core values of the two endpoints as follows: each edge of $\overrightarrow{E_t}$ is an edge of E_t assigned a direction in such a way that if $\langle u, v\rangle$ is a directed edge oriented from u to v in $\overrightarrow{E_t}$ then the $core(u, G_t) \leq core(v, G_t)$.

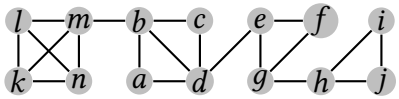


Figure 6: An example graph G_t.

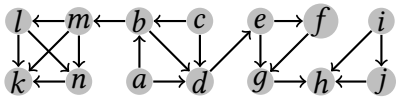


Figure 7: Orientation graph of the graph G_t given in Fig. 6.

The orientation graph is constructed from the output of the *core decomposition* algorithm (Alg. 1) [25]. This procedure iteratively computes core values of nodes by removing the one with the minimum degree in each iteration. Since there may be multiple nodes of minimum degree at every step of the core decomposition algorithm, the output order in which nodes are removed is not unique. The *degeneracy order* of a graph is one of the possible orders in which nodes may be removed.

Definition 4 (Degeneracy Order). *We call a permutation $\pi : V \to [n]$ of the node set a* degeneracy order *if it is a possible output order of the core decomposition algorithm, i.e., $\pi^{-1}(1)$ is the first node removed, $\pi^{-1}(2)$ is the second and so on. We use the notation $\pi(u)$, $u \in V$ to denote the order of node u and $\pi^{-1}(n)$, $n \in \mathbb{Z}$ to denote the n^{th} node in the permutation.*

We will use the convention that $\langle u, v\rangle$ denotes a directed edge oriented from u to v while (u, v) will be used to denote an undirected edge. Given the degeneracy order π of a static graph G_t, we can construct its *degeneracy orientation graph.*

Definition 5 (Degeneracy Orientation Graph). *Given an undirected graph $G_t = (V, E_t)$, a directed graph $\overrightarrow{G}_t = (V, \overrightarrow{E}_t)$ where $\overrightarrow{E_t}$ is obtained by assigning a direction to each edge of E_t is called an* orientation *of G_t. An orientation $\overrightarrow{G}_t$ is called a* degeneracy orientation graph (degeneracy orientation *for short) of G_t if there is a degeneracy order π such that for every directed edge $\langle u, v\rangle \in \overrightarrow{E_t}$ we have that $\pi(u) < \pi(v)$.*

Example 2. *Consider graph G in Fig. 6. The subgraph induced by {k,l,m,n} is the 3-core of G. There does not exist a 4-core in G. Thus, one degeneracy order π of G is (i,j,a,c,b,d,e,f,g,h,m,l,n,k). Fig. 7 shows the corresponding orientation graph.*

Proposition 1. *If $\overrightarrow{G}_t$ is a degeneracy orientation of G_t then*
(1) $\overrightarrow{G}_t$ is a DAG.
(2) If $u_1, u_2, \cdots, u_k$ is a directed path in $\overrightarrow{G}_t$ then $core(u_1, G_t) \leq core(u_2, G_t) \leq \cdots \leq core(u_k, G_t)$.

Proof. Since the orientation of the edges is defined by a total order π, $\overrightarrow{G_t} = (V, \overrightarrow{E})$ cannot have cycles, and hence it is a

Algorithm 1 core-decomposition

Require: undirected graph G_t
Ensure: Core value of all nodes set.
1: Set $i = 0$, $G_t^i = G_t$ and d = minimum degree of G.
2: **while** G_t^i is non-empty **do**
3: **while** there is a node v_i with degree $\leq d$ in G_t^i **do**
4: Set core value of v_i to be d.
5: Set G_t^{i+1} to be $G_t^i \setminus \{v_i\}$.
6: increment i.
7: increment d.

DAG. The second property can be deduced by observing that the core decomposition algorithm assigns core values to nodes in increasing order (lines 4 and 7 in Alg. 1). □

Hereon, we use $outDegree(u, \overrightarrow{G_t})$ to denote the *out degree* of node u in $\overrightarrow{G_t}$. Furthermore, we say node v is *reachable* from u if there exists a directed path from u to v in $\overrightarrow{G_t}$. We denote reachability using the notation $u \preceq v$.

The directionality of edges in the orientation graph of the original undirected graph contains information on how the cascading effect of an edge update spreads. In the next sections, we formally derive these properties.

3.3.2 Algorithm for Edge Insertions. Let (u, v) be the edge inserted to $G_{t-1} = (V, E_{t-1})$ to obtain $G_t = (V, E_t = E_{t-1} \cup \{(u, v)\})$. For simplicity of presentation we assume that $L = core(u, G_{t-1}) < core(v, G_{t-1})$. We discuss the small changes that are required if two nodes have the same core value later in the section.

The following two properties have already been established in the literature [21]:

- **Property 1:** Following an edge addition (resp. deletion), the core value of a node can increase (resp. decrease) by at most 1.
- **Property 2:** If an edge (u, v) is inserted (resp. deleted) in graph $G_t = (V, E)$, such that $core(u, G_t) < core(v, G_t)$, a node w's core value can increase (resp. decrease) only if there exists a path from u to w where all nodes have core value equal to $core(u, G_t)$.

We tighten Property 2 further by utilizing the orientation graph.

Theorem 1. *If an edge (u, v) is inserted in graph $G_t = (V, E)$, such that $core(u, G_t) < core(v, G_t)$, a node w's core value can increase only if there exists a directed path from u to w in the degeneracy orientation graph $\overrightarrow{G_t}$ where all nodes have core value equal to $core(u, G_t)$.*

Proof. Let w be a node whose core value increases when (u, v) is inserted into G_t with $core(u, G_t) < core(v, G_t)$. Clearly w must be reachable from u in G_t. Consider towards contradiction that w is not reachable from u in $\overrightarrow{G_t}$. This means that during core decomposition (Alg. 1), w was removed earlier than u. This, in turn, means the addition of edge (u, v) cannot change the core value assigned to w since if u's core value changes, then it can only affect the core values of nodes that are removed after u. So we get a contradiction.

We have argued that there must be a directed path from u to w in $\overrightarrow{G_t}$. Since core values can only increase along a directed path of $\overrightarrow{G_t}$ the core values of each node on this path is at least $core(u, G_t)$ (c.f Proposition 1). Now, let us assume towards contradiction that there is a node $z \neq w$ on this path whose core value is strictly greater than $core(u, G_t)$. Since, only nodes with core value equal to $core(u, G_t)$ can have their core value increased, core value of node z remains unchanged (Property 2 [21]). Furthermore, if core value of node z remains unchanged, then core value of w will not increase as all the neighbors have same core value as before insertion. Hence, we get a contradiction. □

Algorithm 2 Orientation-Insertion

Require: Degeneracy Orientation Graph $\overrightarrow{G_t}$, edge to be inserted (u, v)
Ensure: Core value of all nodes updated.
1: $L \leftarrow core(u, G_{t-1})$ ▷ WLOG assume $core(u, G_{t-1}) \leq core(v, G_{t-1})$
2: $\overrightarrow{E_t} \leftarrow \overrightarrow{E_t} \cup \langle u, v \rangle$;
3: $outDegree(u, \overrightarrow{G_t}) \leftarrow outDegree(u, \overrightarrow{G_t}) + 1$;
4: $sameL \leftarrow false$; ▷ True if core values of u and v are equal.
5: **if** sameL **then**
6: correct the direction $\langle u, v \rangle$; ▷ Update corresponding $outDegree$
7: $V_{can} \leftarrow$ ExpansionPhase($\overrightarrow{G_t}$);
8: $V_c \leftarrow$ ShrinkingPhase($\overrightarrow{G_t}, V_{can}$);
9: **for all** $w \in V_c$ **do**
10: $core(w, G_t) \leftarrow core(w, G_t) + 1$;

Algorithm 3 ExpansionPhase

Require: Degeneracy Orientation Graph $\overrightarrow{G_t}$, expansion root u
Ensure: V_{can}: potential candidate set;
1: $V_{can} \leftarrow$ an empty set;
2: Initialize an empty queue Q with node u
3: **while** $Q \neq \phi$ **do**
4: $w \leftarrow Q.dequeue()$;
5: **if** $outDegree(w, \overrightarrow{G_t}) + canInDegree(w, \overrightarrow{G_t}) > L$ **then**
6: $V_{can} \leftarrow V_{can} \cup \{w\}$;
7: **for all** $\langle w, w' \rangle$ **st** $core(w', G_t) = L$ **do**
8: $canInDegree(w', \overrightarrow{G_t}) \leftarrow canInDegree(w', \overrightarrow{G_t}) + 1$;
9: **if** $w' \notin Q \wedge w' \notin V_{can}$ **then**
10: $Q.enqueue(w')$;
11: **return** V_{can}

Since the orientation graph is a directed version of the original undirected graph, Thm. 1 is a provably tighter bound than Property 2. Hence, the search space is reduced. We further reduce the computation through the following result.

Theorem 2. *If $outDegree(u, \overrightarrow{G_t}) \leq L$, then $core(w, G_t) = core(w, G_{t-1})$ for all $w \in V$.*

Proof. From Thm. 1, we know only nodes reachable from u in $\overrightarrow{G_t}$ may have their core values changed. Thus, all nodes with a lower degeneracy order than u have their core values unchanged. For node u to move from L to $L + 1$ core, it must have at least $L + 1$ neighbors with a higher degeneracy order. However, this is not the case as $outDegree(n, \overrightarrow{G_t}) \leq L$. □

Thm. 2 establishes a stricter condition under which there is no effect of an edge update on the core values. Empowered with these bounds, we are now ready to design our algorithm to process edge updates. The orientation algorithm (Alg. 2) works in two phases:

(1) An *expansion* phase (Alg. 3), which does a *breadth-first search (BFS)* on $\overrightarrow{G_t}$ to discover a candidate set of nodes V_{can}, such that $V_{can} \supseteq V_c$, where V_c is the set of nodes whose core values will change due to insertion.
(2) A *shrinking and reorientation* phase (Alg. 4) where the false positive nodes, i.e., $V_{can} \setminus V_c$ are discarded and edges of $\overrightarrow{G_t}$ are reoriented to ensure that $\overrightarrow{G_t}$ is a orientation graph of G_t.

We now discuss these phases in more detail.

- **Initialization.** First, we set: $\overrightarrow{E_t} = \overrightarrow{E_{t-1}} \cup \{\langle u, v \rangle\}$.
- **Expansion Phase.** Alg. 3 lists the pseudocode of the expansion phase. Since only nodes reachable from u can be in V_c (c.f. Lem. 1), the expansion phase expands in *breadth-first manner (BFS)* from u in $\overrightarrow{G_t} = (V, \overrightarrow{E} \cup \langle u, v \rangle)$ and visits only nodes with core values equal to L. Since the BFS starts from u, we call u the *root* node.

In the first pass inside while loop (line 3), w will be u (line 4). So, if $outDegree(u, \overrightarrow{G_t}) \leq L$ then no vertices gets its core value updated (Thm. 2). Hence, the algorithm will terminate with empty V_{can}. If $outDegree\ (u, \overrightarrow{G_t}) > L$, then we initialize a *candidate set*, V_{can} to $\{u\}$ (line 6) and compute the initial *incoming candidate degree* of all nodes $w \in V$, defined as:

$$canInDegree(w, \overrightarrow{G_t}) = \left| \left\{ \langle w', w \rangle \in \overrightarrow{E_t} : w' \in V_{can} \right\} \right|.$$

Hereon, we start a (directed) BFS from u in $\overrightarrow{G_t}$ to grow the candidate set V_{can}. At every step, we retrieve the head node w from the BFS queue and check if $outDegree(w, \overrightarrow{G}_t) + canInDegree(w, \overrightarrow{G}_t) > L$. If the condition is satisfied then w is added to V_{can} and all of w's children (outgoing neighbors) with core value L, are added to the BFS queue. This condition is a result of the following lemma.

LEMMA 1. *If* $outDegree(w, \overrightarrow{G_t}) + canInDegree(w, \overrightarrow{G_t}) \leq L$ *then* $w \notin V_c$.

PROOF. For w to move from L-core in G_{t-2} to $L+1$-core in G_t, w must have at least $L+1$ neighbors in G_t with core value of $L+1$ or higher. $outDegree(w, \overrightarrow{G_t})$ counts all neighbors with higher degeneracy order and therefore potential outgoing neighbors with core value $L+1$. Among the incoming neighbors, $canInDegree(w, \overrightarrow{G_t})$ counts all incoming neighbors that could potentially move from L-core to $L+1$-core. Therefore, $outDegree(w, \overrightarrow{G_t}) + canInDegree(w, \overrightarrow{G_t})$ is an upper-bound on the number of neighbors with core value $L+1$ in G_t. Hence, if $outDegree(w, \overrightarrow{G_t}) + canInDegree(w, \overrightarrow{G_t}) \leq L$, w cannot move to the $L+1$-core and therefore $w \notin V_c$. □

Observation: It is important to note that at any intermediate stage of the BFS, the computation of V_{can} is partially complete. However, we are still able to apply the degree criterion, since $canInDegree(w, \overrightarrow{G_t})$ depends only on the incoming neighbors to w. BFS performs a level-wise traversal, and thus all incoming neighbors of w are evaluated before w is accessed for the *last* time. Consequently, any possible contributors to $canInDegree(w, \overrightarrow{G_t})$ has already been added to V_{can} by the time the BFS reaches w.

• **Shrinking and Reorientation Phase.** In this phase, our goal is to (1) eliminate the nodes in $V_{can} \setminus V_c$, and (2) ensure $\overrightarrow{G}_t$ is a orientation graph of G_t. Toward that, we adopt the following iterative procedure. Alg. 4 presents the pseudocode.

(1) For every node $w \in V_{can}$ identify: (line 1-2)

$$T_w = \{v : \langle w, v\rangle \in \overrightarrow{E_t}, core(v, G_{t-1}) = L, v \notin V_{can}\}.$$

(2) If $outDegree(w, \overrightarrow{G}_t) + canInDegree(w, \overrightarrow{G}_t) - |T_w| \leq L$, (line 3)
- Remove w from V_{can}. (line 4)
- For each $v \in T_w$, remove $\langle w, v\rangle$ from $\overrightarrow{E}_t$, and insert $\langle v, w\rangle$, i.e., flip each such edge. (line 5-6).
- Delete $\langle u, w\rangle$ from $\overrightarrow{E}_t$ for every $u \in V_{can}$, and insert $\langle w, u\rangle$.

(3) Since V_{can} may change with removal of nodes, the $canInDegree(w, \overrightarrow{G}_t)$ of some nodes $w \in V_{can}$ may change as well. Thus, iterate by re-starting the process from Step 1, till we reach a stage where no nodes are removed from V_{can}. (line 7)

Algorithm 4 ShrinkingPhase

Require: Degeneracy Orientation Graph $\overrightarrow{G_t}$, potential candidate set V_{can}
Ensure: V_c: set of nodes whose core values need to be changed
1: **for all** $w \in V_{can}$ **do**
2: $T_w \leftarrow \{v : \langle w, v\rangle \in \overrightarrow{E_t}, core(v, G_{t-1}) = K, v \notin V_{can}\}$
3: **if** $outDegree(w, \overrightarrow{G}_t) + canInDegree(w, \overrightarrow{G}_t) - |T_w| \leq L$ **then**
4: $V_{can} \leftarrow V_{can} \setminus \{w\}$
5: **for all** $v \in T_w$ **do**
6: Reverse $\langle w, v\rangle$ in $\overrightarrow{E_t}$ ▷ update corresponding *canInDegree,outDegree*
7: Reverse $\langle u, w\rangle$ in $\overrightarrow{E_t}$ for every $u \in V_{can}$ ▷ update corresponding *canInDegree,outDegree*
8: Repeat line 1-7 till no nodes can be further removed from V_{can}
9: **return** V_{can} ▷ $V_{can} = V_c$

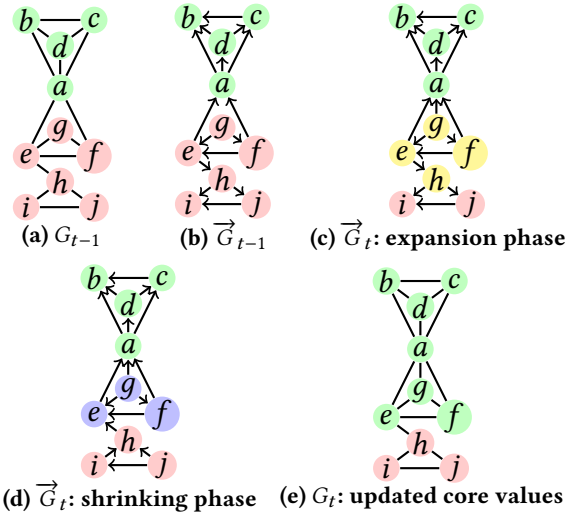


Figure 8: We set $\mathcal{K} = 3$. Graph G_t given in sub-figure (e) is obtained from the graph G_{t-1} given in sub-figure (a) by addition of edge (a, g). The graphs $\overrightarrow{G}_{t-1}$ and $\overrightarrow{G}_t$ are the orientation graphs of G_{t-1} and G_t respectively. The green vertices have core value 3, and the remaining vertices have core value less than 3. The candidate set V_{can} consists of yellow vertices after expansion phase. The set V_c consists of blue vertices whose core value is incremented after shrinking phase.

Core Value Updation. Following the completion of shrinking and reorientation phase, we increment the core value of each node in V_c by 1. This follows from Property 1 [18].

For example, consider Fig. 8. We set $\mathcal{K} = 3$. Graph G_t given in sub-figure 8e is obtained from the graph G_{t-1} given in sub-figure 8a by addition of edge (a, g). Therefore, $L = 2 = core(g, G_{t-1})$. The graphs $\overrightarrow{G}_{t-1}$ and $\overrightarrow{G}_t$ are the orientation graphs of G_{t-1} and G_t respectively. The green vertices have core value 3, and the remaining vertices have core value less than 3. The candidate set V_{can} consists of yellow vertices after expansion phase because sum of outDegree and canInDegree is greater than 2 for each of yellow nodes, and no other vertex satisfies this condition. The vertex h is discarded from V_{can} to obtain V_c during shrinking phase because $outDegree(h, \overrightarrow{G}_t) + canInDegree(h, \overrightarrow{G}_t) - |T_h| = 1 < L$. The direction of some edges of orientation graph is also reversed during this phase. The set V_c consists of blue vertices whose core value is incremented after shrinking phase.

THEOREM 3 (INSERTION CORRECTNESS). *Given a graph* $G_{t-1} = (V, E_{t-1})$ *and two vertices* $u, v \in V$ *such that* $(u, v) \notin E_{t-1}$ *and* $L = core(u, G_{t-1}) \leq core(v, G_{t-1})$, *the Insertion Algorithm has the following properties:*

(1) The vertices that remain in V_{can} *at the beginning of the Core Value Update Phase is precisely the set* V_c *of vertices whose core value is to be incremented, and*

(2) the directed graph $\overrightarrow{G}_t = (V, \overrightarrow{E}_t)$ *obtained at the end of the algorithm is a orientation graph of* G_t.

Insertion when $L = core(u, G_{t-1}) = core(v, G_{t-1})$

When the core values are the same, we do not know apriori which way to orient the edge (u, v). Since there should not be a cycle in $\overrightarrow{G}_t$, we make this decision by checking if u is reachable from v in $\overrightarrow{G}_{t-1}$ or vice versa, and orienting the edge accordingly

to ensure no cycle is created. From property 2 of Prop. 1, we are guaranteed that if at all the added edge induces a cycle in the orientation graph, this cycle must exist in the induced subgraph formed by nodes in V_{t-1}^{L} changed from v_c to this. Thus, we can limit the cycle checking exploration to a much smaller subgraph of the entire graph. This property indeed holds on real datasets, where we find that cycle-checking is performed only on 5%, 2.7%, and 4.2% of the edge updates in WikiTalk, StackOverflow, and #Caravan respectively.

Proof of Theorem 3

Proof. We establish this theorem through following lemmas.

Lemma 2 (Pruning correctness). *If we denote by V_{can}^{S} the set of vertices left in V_{can} after the Shrinking and Reorientation Phase, then $V_{can}^{S} = V_c$.*

Proof. Let $V_{t-1}^{\geq L+1}$ be the set of nodes in the $L+1$-core of G_{t-1}. Now, consider the subgraph induced by $V_{t-1}^{\geq L+1} \cup V_{can}$. Since $\forall w \in V_{can},\ T_w = \emptyset$, and $outDegree(w, \overrightarrow{G_t}) + canInDegree(w, \overrightarrow{G_t}) > L$, every node in V_{can} has at least $L+1$ neighbors from either V_{can} or $V_{t-1}^{\geq L+1}$. Thus, all nodes in V_{can} are part of $L+1core$. Therefore, $V_{can} = V_c$ since all nodes in V_{can} were in L-core in G_{t-1}. □

We next prove the second part of Theorem 3, i.e., orientation graph is correctly maintained.

Lemma 3 (Reorientation Correctness). *$\overrightarrow{G}_t = (V, \overrightarrow{E}_t)$ at the end of the Insertion algorithm is a orientation graph of G_t.*

Proof. Let us consider running Alg. 1 on G_t. Till the variable d reaches core value L, everything remains unchanged. Further, if u was removed in iteration i, everything remains the same till iteration $i-1$ as we have argued above. Now note that there are some nodes with core value L that were included in V_{can} in the Expansion Phase and removed in the Shrinking Phase. Their removal happened because their degree among candidates and higher core value nodes was not $L+1$. Clearly these nodes will be removed *before* any node of V_c is removed. But these nodes were discovered because there were edges oriented towards them from V_{can}. These edges, therefore, must be reversed otherwise $\overrightarrow{E}_t$ will not satisfy the property that it is produced by a core decomposition algorithm. Finally, once we dispense with these nodes the rest of the trace of Alg. 1 can follow the same order as it did for G_{t-1}, except that the value of d will be incremented to $L+1$ before we do so. □

This completes the proof of Thm 3. □

Time Complexity. The insertion algorithm involves a BFS in the forward direction and then a shrinking phase which goes backwards over the BFS tree constructed. To bound the running we note that the BFS stays within the set of nodes whose core value is L. Hence, the running time is $O\left(\sum_{v \in V_t^L} deg(v, G_t)\right)$, where V_t^L is the set of nodes with core value L in G_t.

3.3.3 Algorithm for Edge Deletions. Let (u, v) be the edge deleted to $G_{t-1} = (V, E_{t-1})$ to obtain $G_t = (V, E_t = E_{t-1} \setminus \{(u, v)\})$. Consistent with the previous discussion, we assume $L = core(u, G_{t-1}) \leq core(v, G_{t-1})$.

Deletion algorithm overview: Core maintenance, when an edge is deleted, is similar in spirit to the Shrinking Phase of our insertion algorithm as it involves: **(1)** identifying nodes that do not have the support to qualify for core value of L, and **(2)** ensuring that the orientation graph $\overrightarrow{G}_t$ that we get after deleting and changing the core values is a orientation graph of the updated graph G_t.

Algorithm 5 Orientation-Deletion

Require: undirected graph G_t, Degeneracy Orientation Graph $\overrightarrow{G_t}$, edge to be deleted (u, v)
Ensure: Core values of all nodes updated.
1: $L \leftarrow core(u, G_{t-1})$ ▷ WLOG assume $core(u, G_{t-1}) \leq core(v, G_{t-1})$
2: $V_c \leftarrow$ an empty set
3: Initialize an empty queue Q with node u
4: **if** $core(v, G_{t-1}) = L$ **then**
5: $Q.enqueue(v)$
6: **while** $Q \neq \phi$ **do**
7: $w \leftarrow Q.dequeue()$;
8: **if** no. of neighbors of w in $V_{t-1}^{\geq L} < L$ **then**
9: **for all** $\langle w', w \rangle \in \overrightarrow{G_t}$ **st** $w' \in V_{t-1}^{\geq L} \setminus V_c$ **do**
10: Reverse $\langle w', w \rangle$ in $\overrightarrow{E_t}$; ▷ update corresponding *outDegree*
11: $V_c \leftarrow V_c \cup \{w\}$
12: **for all** neighbors w' of w where $core(w', G_{t-1}) = L \wedge w' \notin V_c$ **do**
13: $Q.enqueue(w')$
14: **for all** $w \in V_c$ **do**
15: $core(w, G_t) \leftarrow core(w, G_{t-1}) - 1$

As in the case of insertion, we package both these activities, identifying V_c and reorienting edges together in a single phase. We now discuss the algorithm in greater detail, using the notation that $V_{t-1}^{\geq L}$ is the set of nodes with core value at least L in G_{t-1}. Alg. 5 outlines the pseudocode.

Initialization. First we set: $\overrightarrow{E_t} = \overrightarrow{E}_{t-1} \setminus \{\langle u, v \rangle\}$, and $E_t = E_{t-1} \setminus \{(u, v)\}$. Initialize $V_c = \emptyset$. (line 2)

Undirected Shrinking and Directed Reorientation. We iterate over each node $w \in V$ with $core(w, G_{t-1}) = L$ such that w has less than L (undirected) neighbours in $V_{t-1}^{\geq L} \setminus V_c$ (line 8). For each such node, we perform the following actions.

- Place w in V_c (line 11).
- If there is any edge of the form $\langle w', w \rangle \in \overrightarrow{E}_t$ s.t. $w' \in V_{t-1}^{\geq L} \setminus V_c$, flip it, i.e. remove it and insert $\langle w, w' \rangle$ in its place (line 9-10).
- Since V_c is changed with addition of w, some more nodes can be eligible for V_c. Hence, iterate by re-starting the process.

Note that above process can be easily converted to undirected BFS as outlined in Alg. 5.

Core Value Update Phase. We decrement the core value of all vertices in V_c by 1. (line 14-15)

Theorem 4 (Deletion Correctness). *Given a graph $G_{t-1} = (V, E_{t-1})$ and two vertices $u, v \in V$ such that $(u, v) \in E_{t-1}$ and $L = core(u, G_{t-1}) \leq core(v, G_{t-1})$, the Deletion Algorithm described above has the properties that*

(1) The vertices placed in V_c are precisely the set V_c of vertices whose core value is to be decremented by 1, and

(2) the directed graph $\overrightarrow{G}_t = (V, \overrightarrow{E}_t)$ obtained at the end of the algorithm is a orientation graph for G_t.

Proof. Property 1 of Thm. 4 has been established by Saríyüce et. al. in [21]. So we focus on the second property, i.e., reorientation correctness. Let us compare a trace of the core decomposition algorithm Alg. 1 on G_{t-1} to a trace on G_t. Till the value of $d = L-1$ the algorithm proceeds exactly the same. After these, nodes with core value L start being pulled out from G_{t-1}. However, in G_t, because of V_c being correctly selected, we begin to pull out vertices from V_c. Edges from these vertices should be oriented *towards* vertices in $V_t^{\geq L}$. Furthermore, since, at any stage in the algorithm $V_{t-1}^{\geq L} \setminus V_c \supseteq V_t^{\geq L}$ (by property 1), orienting edges from the vertices of V_c to the vertices of $V_{t-1}^{\geq L} \setminus V_c$ as $(V_{t-1}^{\geq L} \setminus V_c) \setminus V_t^{\geq L}$, i.e., the vertices with core value L in G_{t-1} that will enter V_c after w will also have the same core value as w. Thus, the edge is correctly oriented as long as we extract the vertices of V_c in the trace of

Table 2: Temporal graphs used in the experiments.

Dataset	|V|	|E|	Average Degree	Date Range	Granularity
WikiTalk	$1.14M$	$7.8M$	13.68	10/2001 – 1/2008	seconds
StackOverflow	$2.6M$	$63.5M$	48.84	8/2008 – 3/2016	seconds
Twitter	$15M$	$257M$	34.26	6/2009 – 12/2009	seconds
#Caravan	$2.4M$	$2.94M$	2.45	10/2018 – 11/2018	seconds

Table 3: Temporal graphs used for applications.

Dataset	|V|	|E|	Average Degree	Date Range	Granularity
Election	346, 573	$2.13M$	12.29	05/2016 – 11/2016	seconds
DBLP	217312	631283	5.81	1952 – 2012	years

Alg 1 on G_t in the same order as they were encountered in the trace of the algorithm on G_{t-1}. □

Time complexity: The algorithm takes time proportional to the number of edges incident on V_c. Thus, the time complexity is $O\left(\sum_{v \in V_c} deg(v, G_t)\right)$ which is at most $O\left(\sum_{v \in V_t^L} deg(v, G_t)\right)$, where V_t^L is the set of nodes with core value L in G_t.

3.3.4 Technical differences with OBA [29] . Following an edge update, let V_c be the set of vertices whose core values will change. To perform core maintenance, OBA [29] uses the following framework, which we also follow: (1) Use a *helper data structure* to efficiently compute the set V_c, (2) update core values by processing the identified set and (3) update the helper data structure efficiently. We differ from OBA in the choice of the helper data structure. OBA uses the *degeneracy order* of the graph, whereas we work with the *degeneracy orientation graph*. The key differentiating factor is that multiple degeneracy orders may map to the same degeneracy graph.

LEMMA 4. *Multiple degeneracy orders may map to the same degeneracy graph.*

PROOF. The degeneracy order π is a topological sort of $\overrightarrow{G_t}$, which implies that one orientation graph corresponds to multiple degeneracy orders. □

Lemma 4 indicates that the orientation graph is a more precise way of storing core value information, and it is this that helps us improve on the order-based algorithm. As a consequence, when an edge update happens, the degeneracy order may need to be changed but the degeneracy orientation graph may remain unaffected, which in turn, leads to faster performance. Our contribution, therefore, lies in **(1)** proposing the orientation graph data structure, **(2)** developing new algorithms on the platform provided by orientation graph to prune the search space, and **(3)** maintaining the orientation graph while handling a stream of edge updates.

4 EXPERIMENTS

In this section, we benchmark KWIQ and establish that:

- **Efficiency :** KWIQ is 5 times faster, on average, than performing core-maintenance [29] within the query window.
- **Scalability:** The proposed pruning algorithms are effective and allows KWIQ to scale to million-sized networks.

Our implementation can be downloaded from https://github.com/idea-iitd/KWIQ.

4.1 Datasets

Table 2 summarizes the temporal networks used for empirical evaluation. Table 3 summarizes the temporal networks used for applications. The semantics are as follows:

• **WikiTalk [5]:** Wikipedia is a free encyclopedia written collaboratively by volunteers around the world. Each registered user (node) has a *talk* page, that other users can edit. A directed edge from node u to node v represents that user u edited a talk page of user v.

• **StackOverflow [3]:** StackOverflow is a temporal network containing interactions on the stack exchange web site Stack Overflow. Each edge (u, v, t), denotes an interaction between user (node) u and v at time t.

• **Twitter [4]:** In this dataset, each node represents a twitter handle (entity), and an edge between (u, v, t) denotes an interaction between u and v where one of them has either "retweeted" or "mentioned" the other person in a tweet.

• **#Caravan [1]:** This is also a twitter dataset, but only contains tweets that contain the "#Caravan" hashtag related to the Central American migrant caravans [2]. We use this dataset to see if tweets pertaining to this viral hashtag has any different property than a more diverse set of tweets.

• **Election:** This dataset was collected by Shao et al. [23] during the 2016 US Presidential election from Twitter. In this network, each node is a user and each edge corresponds to a retweet that contains an URL. In addition, each edge is classified as either "authentic" or "inauthentic". This labeling is done by Shao et al. based on URL contained in the retweet. Specifically, if the URL is authentic, the retweet is classified as authentic and vice versa. We use this dataset to demonstrate an application of tracking core-invariant nodes.

• **DBLP:** This dataset is created from DBLP [26]. An edge (u, v, t) exists between papers u and v if paper u cites paper v at time t. We use this dataset to demonstrate that span core is empty while invariant node set is non-empty.

We set the deletion window (Recall our deletion model from Sec 2) to 15 days for all datasets except #Caravan (See Table 4). Since the Caravan hashtag became viral for a short duration and then slowly disappeared from social media after a month, the deletion window is set to 1 day.

4.2 Experimental Setup

All our implementations including baselines are in Java (OpenJDK version "11.0.14"). The experiments are performed on a machine equipped with Intel(R) Xeon(R) Platinum CPU 2.1GHz having 256 GB Ram running Ubuntu 18.04. All experiments are repeated 5 times for consistency and we plot the average across five runs.

4.2.1 Baseline Algorithms. We compare against performing core-maintenance within the query window $\Delta = [t_s, t_e]$ using the state-of-the-art *Order-based Algorithm (OBA)*[29]. While performing core-maintenance with OBA, we keep track of all nodes that have never fallen below the core-value threshold $\mathcal{K}$ till the current timestamp within Δ. If at any time, this set becomes empty, OBA terminates. Otherwise, OBA terminates at the ending timestamp and returns the core-invariant nodes.

4.2.2 Default Parameter Values. The default values of these parameters are set as follows.

Table 4: Default parameter values

Dataset	$\mathcal{K}$	DW (days)	Δ (days)
WikiTalk	4	15	30
StackOverflow	8	15	30
Twitter	11	15	30
#Caravan	3	1	2

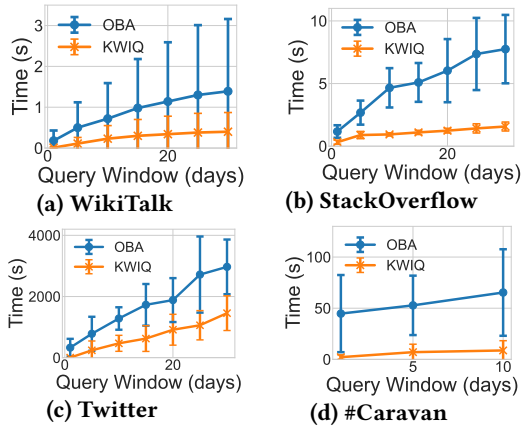


Figure 9: Impact of the size of the query window on the querying time.

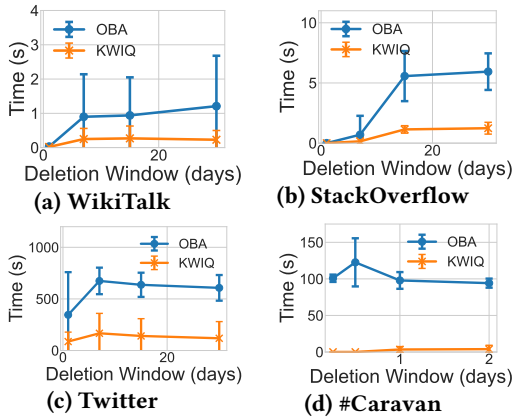


Figure 10: Impact of Deletion Window (DW) length on the querying time.

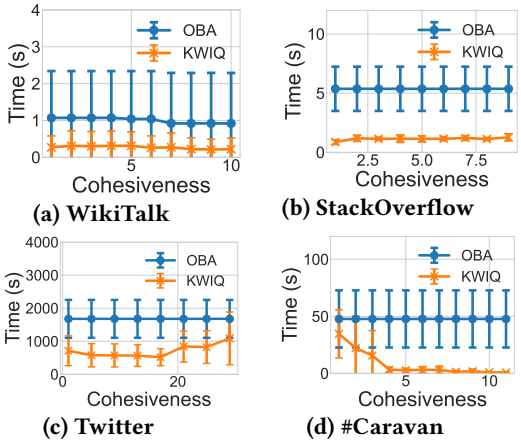


Figure 11: Impact of cohesiveness threshold $\mathcal{K}$ on querying time.

• **Cohesiveness threshold $\mathcal{K}$** : We set $\mathcal{K}$ in each dataset to the 95% percentile value, where 5% of the nodes in the network have a core-value at least as high as $\mathcal{K}$. The 95% percentile $\mathcal{K}$ values for each dataset is shown in Table 4.

• **Temporal Window** $\Delta = [t_s, t_e]$: Δ is set to 2 times the deletion window starting from a randomly chosen t_s.

Table 4 lists the default parameter values across all datasets. For those experiments where the cohesiveness threshold $\mathcal{K}$ is varied we have increased the value of $\mathcal{K}$ up to the point where the answer set is non-empty, i.e., for each data set beyond the largest value of $\mathcal{K}$ plotted on the x-axis the invariant core size is 0.

Table 5 shows the average temporal degree per node per time stamp. The dataset Twitter has highest temporal average degree. This is the reason why querying time is highest for Twitter as shown in Figures 9, 10, 11, and 13. The dataset StackOverflow has higher static average degree as compared to Twitter, whereas Twitter has higher value for temporal average degree in comparision with StackOverflow. Therefore, temporal average degree has better corelation with coreness.

4.3 Efficiency and Scalability

4.3.1 Impact of Query Window Size. First, we study the impact of the query window size, i.e., Δ on the querying time. Fig. 9 presents the querying times of KWIQ and OBA as the size of the query window is varied. As visible, across all datasets, KWIQ is significantly faster than OBA, which validates the efficacy of the proposed pruning strategies. On average, KWIQ is 6 times faster than OBA. The highest speed-up is observed in the #Caravan dataset where KWIQ is 10 times faster. As expected, with increase in the window size, we observe an increase in querying time since the number of edge updates we need to process is larger.

4.3.2 Impact of Deletion Window. In this experiment, we vary the length of the deletion window, while keeping the query window size and $\mathcal{K}$ fixed to their default values, and observe its impact on the querying time. Note that since popular social networks like Twitter measure active users in the scale ranging from one day to one month [15], we work with deletion window in this time scale. Figure 10 presents the results. Similar to previous experiments, KWIQ is significantly faster than OBA across all three datasets regardless of the deletion window length. We derive two key observations from this experiment. First, as deletion window increases, there is a mild increase in the running times of KWIQ as well as OBA. Increase in deletion window implies increase in density of the graph since each edge gets more time to remain active. When density is high, the cascading effect of an edge update reaches a larger number of nodes, and consequently, the average processing time of updates increases. The second important observation is that the increase in deletion window affects OBA more adversely. This trend is a manifestation of the fact that the edge density increases more intensely among the highly active nodes. These highly active nodes often fall within PIN, and all edge updates among nodes in PIN, regardless of the density, get processed in $O(1)$ time.

4.3.3 Impact of $\mathcal{K}$. Fig. 11 presents the querying times of KWIQ and OBA across all four datasets as $\mathcal{K}$ is varied. As can be seen, across all four datasets, KWIQ is significantly faster. The performance improvement of KWIQ is most pronounced in #Caravan, where KWIQ is more than 200 times faster. On average, KWIQ is 5 times faster than *OBA*. The range of cohesiveness

Table 5: Average Temporal Degree

Dataset	|V|	DW	Average Degree per node per time stamp
WikiTalk	1.14M	15 days	0.29
StackOverflow	2.6M	15 days	0.57
Twitter	15M	15 days	2.65
#Caravan	2.4M	1 day	0.80

(x-axis) for each dataset in Fig. 11 is chosen such that it gives non-empty core-invariant nodes.

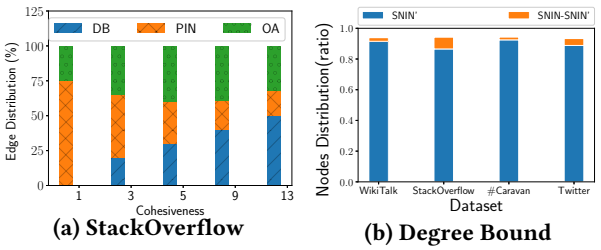
In KWIQ, as $\mathcal{K}$ increases, most node gets pruned out due to degree bound, which allows us to process a large percentage of edge updates in $O(1)$ time. OBA, in general, is independent of $\mathcal{K}$. The only exception is the situation where at a particular time point, *all* nodes become non-core invariant, and hence OBA terminates. This event happens with a higher frequency when $\mathcal{K}$ is large. In our experiment, this situation arises only in WikiTalk.

4.3.4 Comparison of KWIQ and OBA on dynamic synthetic datasets. Fig. 14 depicts the querying times of KWIQ and OBA on dynamic synthetic datasets. The dynamic graph generator which we use to create synthetic graphs is given in [22]. This benchmark generator generates graphs with overlapping communities in such a way that several crucial properties of the graph are maintained over time. For example, community sizes vary according to power law distribution. We create 400 dynamic datasets with different densities using the benchmark generator given in [22]. Each of these datasets consists of one hundred thousand nodes. For each dataset, we execute KWIQ and OBA 5 times. We plot the mean execution time versus average degree per node for KWIQ and OBA in fig. 14. Fig. 14b compares mean execution time of KWIQ and OBA on synthetic datasets for $\mathcal{K} = 10$. Similarly, fig. 14c compares mean execution time of KWIQ and OBA on synthetic datasets for $\mathcal{K} = 50$. We do smoothing for better demonstration by plotting the average of all values in a neighborhood of the point under consideration. We observe that KWIQ performs better than OBA on all generated synthetic datasets.

4.4 Efficacy of Pruning Strategies

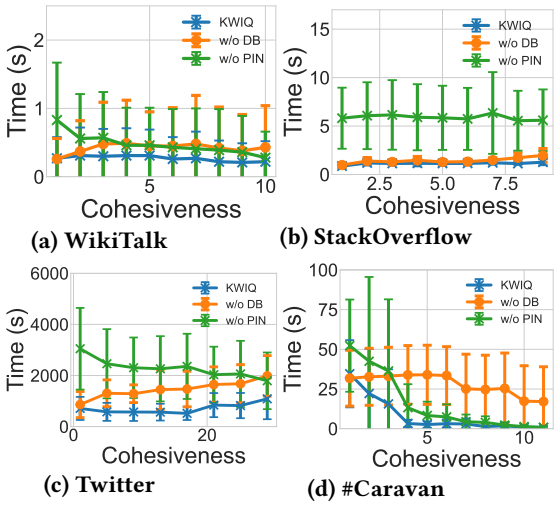
KWIQ relies on three pruning strategies: degree bound to approximate strong non-invariant nodes (DB), potential core-invariant node (PIN), and orientation algorithm (OA). We investigate the impact of each of these strategies.

In Figs. 13a-13d, we individually turn off each of the pruning strategies and see how they affect the querying time. In #Caravan, we observe that degree bound imparts a higher speed-up to the querying time when compared to PIN for most of cohesiveness (x-axis) range. On the other hand, in StackOverflow, PIN plays a more important role in keeping the querying time down for most of cohesiveness (x-axis) range. A closer inspection reveals that StackOverflow is mostly dominated by interactions among highly frequent users and thus PIN is crucial to efficiency. On the other hand, in #Caravan, there are a large number of dormant users, and therefore, degree bound plays a higher role in pruning out nodes from the search space.



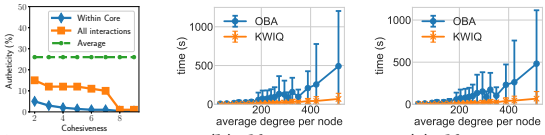
(a) StackOverflow (b) Degree Bound

Figure 12: (a) Distribution of edge updates based on who they are handled by among Degree Bound (DB), Potentially Invariant Nodes (PIN) and Orientation Algorithm (OA). Recall Fig. 4 and Fig. 5. (e) Accuracy of Degree Bound.



(a) WikiTalk (b) StackOverflow (c) Twitter (d) #Caravan

Figure 13: The efficacy of the various pruning strategies against cohesiveness threshold $\mathcal{K}$.



(a) Authenticity (b) $\mathcal{K} = 10$ (c) $\mathcal{K} = 50$

Figure 14: (a) Relationship of core-invariant nodes with authenticity. (b-c) Comparison of KWIQ and OBA on synthetic datasets.

As visible in Fig. 13c, Twitter is the only dataset where both DB and PIN are needed to reduce the running time; in all other datasets, as the cohesiveness threshold increases, either DB or PIN alone is enough to provide speed-up. To understand why this happens in Twitter, recall that DB is effective if most nodes have a degree less than the cohesiveness threshold. On the other hand, PIN is effective if most interactions are among nodes that are already in the k-core. Twitter has the highest temporal degree (See Table 5). Hence, DB is not as effective. On the other hand, we observe a large number of interactions in Twitter between nodes where one of them is not in PIN. In other words, the number of dormant users in Twitter is less compared to other social networks and hence, PIN is not as effective as in the other networks.

To further gain a deeper understanding behind the intricacies of the pruning strategies, in Fig. 12a, we study the distribution of edge updates based on how they are processed. We present the results in StackOverflow. We observe that regardless of the cohesiveness threshold, around 25% of the edge updates are processed through orientation algorithm. Among DB and PIN, cohesiveness has a big impact. As $\mathcal{K}$ increases, a larger portion of the updates get pruned out through degree bound (DB). On the other hand, for smaller $\mathcal{K}$, majority of the edges are pruned through PIN. Overall, this experiment clearly established that all three pruning strategies are important to obtain fast querying times.

Finally, we compute the portion of the strongly non-invariant nodes that is identified by degree bound. As shown in Fig. 12b, degree bound identifies more than 90% of the nodes in SNIN.

4.5 Applications

In this section, we showcase two concrete applications of tracking invariant cores in temporal networks.

4.5.1 Identification of Inauthentic Behavior. Several studies have found correlation of high core value nodes with inauthentic behavior [23, 24]. For example, Shao et. al. [23] had shown on the Election dataset that nodes with a high core value have a high affinity towards spreading fraudulent information. However, these studies do not take temporality into account. In this section, we bring in the temporal aspect and study the relationship between core-invariant nodes and the affinity towards spreading fraudulent information.

As described in §4.1, in the Election dataset, each edge is classified with a label of being either "authentic" or "inauthentic". To analyze the relationship between core-invariant nodes and inauthentic behavior, we vary the cohesiveness threshold $\mathcal{K}$ and measure the average authenticity of core-invariant nodes. The authenticity of a node is quantified using two different measures. The "All interactions" authenticity of a node u is defined as:

$$\frac{\|\{e = (u, v) \in E \mid e \text{ is authentic}\}\|}{\|\{e = (u, v) \in E\}\|} \times 100$$

where E is the set of all edges in the temporal network. The second authenticity measure, called "Within core" authenticity is defined as:

$$\frac{\|\{e = (u, v) \in E \mid e \text{ is authentic}, v \text{ is core-invariant}\}\|}{\|\{e = (u, v) \in E \mid \ v \text{ is core-invariant}\|} \times 100$$

While the "All interactions" authenticity measures the proportion of authentic tweets by a node, the "Within core" authenticity measures the authenticity only among interactions with other core-invariant nodes. In Fig. 14a, we plot these authenticity metrics against the cohesiveness threshold $\mathcal{K}$. The dashed line in Fig. 14a denotes the average proportion of authentic edges across all edges, which is $\approx 26\%$. Since the average is independent of $\mathcal{K}$, it is a straight line. If all nodes behave independently of whether they are core-invariant, we expect the authenticity to remain constant with cohesiveness threshold $\mathcal{K}$. However, we clearly see that with $\mathcal{K}$, the authenticity drastically reduces. In fact, authenticity goes to ≈ 0 as we move towards the innermost core-invariant nodes. Even more interestingly, the "All interactions" authenticity is always higher than the 'Within core" authenticity indicating that the more coordinated the interaction, the higher is the chance of inauthentic behavior. This experiment shows that our definition of core invariant nodes is useful in tracking coordinated inauthentic behavior that has been observed on Facebook [13] and Twitter [27] and it is widely considered to have very dangerous social and political consequences.

4.5.2 Mining Influential Papers. Finding hot topics and influential papers has been a topic of interest to the Scientometric community for a long time. Moving on from studies based on keywords, there have been attempts to use the network structure of the citation network to identify influential papers, e.g., by using metrics such as PageRank [12]. The concept of k-cores has also been applied to this effort [7]. Both these lines of work neglect temporality which is a key concept when it comes to the evolution of research. In this section, we briefly illustrate that invariant cores are a good mechanism to recognize the emergence of a hot topic while simultaneously identifying the key papers within the topic.

Table 6: Invariant core for $\mathcal{K}$ = 4, Δ = 1998-2004

SN	Paper
1.	Park, Jong Soo, Ming-Syan Chen, and Philip S. Yu. "An effective hash-based algorithm for mining association rules." Acm sigmod record 24, no. 2 (1995): 175-186.
2.	Agrawal, Rakesh, and Ramakrishnan Srikant. "Fast algorithms for mining association rules." In Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, pp. 487-499. 1994.
3.	Agrawal, Rakesh, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." In Proceedings of the 1993 ACM SIGMOD international conference on Management of data, pp. 207-216. 1993.
4.	Savasere, Ashok, Edward Robert Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. Georgia Institute of Technology, 1995.
5.	Han, Jiawei, and Yongjian Fu. "Discovery of multiple-level association rules from large databases." In VLDB, vol. 95, pp. 420-431. 1995.
6.	Toivonen, Hannu. "Sampling large databases for association rules. " In Vldb, vol. 96, pp. 134-145. 1996.
7.	Agrawal, Rakesh, and Ramakrishnan Srikant. "Mining sequential patterns." In Proceedings of the 11TH International Conference on Data Engineering, pp. 3-14. IEEE, 1995.

We find the invariant core for the DBLP dataset for $\mathcal{K} = 4$ and temporal span 1998-2004 (Table 6). Note that these 7 papers do not form a span-core [10, 11] during 1998-2004 since the most recent paper was published in 1996 and the deletion window is 2 years. Thus, the 7 papers induce a subgraph that becomes disconnected after 1998. Further, we compute top ten degree nodes during 1998-2004, and we found that only one out of these 7 papers is in the top ten most cited papers during 1998-2004. Therefore, invariant nodes reveal a pattern that neither a simple notion like number of citations nor a more sophisticated concept like span-core could reveal. The way this result can be interpreted is that these 7 papers form the most influential papers of a hot research topic related to association rule mining that was a hot topic in the time period we queried.

A fuller investigation is required to determine how well the invariant core definition is able to capture subjective notions of influence and "hotness." We postpone this to future work.

5 CONCLUSION

In this paper, we introduced the notion of *core-invariant nodes* on temporal networks. Invariant cores identify dense substructures that remain together for a significant duration of time. Invariant cores has been shown to be a key indicator of network stability. Querying core-invariant nodes on temporal networks is a computationally intensive task and therefore not scalable to large datasets. To overcome this computational bottleneck, we developed a technique called KWIQ, which strategically partitions the search space intro three disjoint sets such that edges belong to two of those sets can be processed in $O(1)$ time. To process edges from the third set, we propose *orientation algorithm* to efficiently compute the cascading effect of an edge update. To demonstrate the efficacy of the proposed pruning strategies, we performed extensive empirical analysis on real million-scale temporal networks and established that KWIQ is 5 times faster than the baseline strategy of performing core-maintenance using the state-of-the-art order-based algorithm.

REFERENCES

[1] 2018. Caravan. https://bit.ly/34VLWTo

[2] 2022. Caravan Wikipedia Page. https://en.wikipedia.org/wiki/Central_American_migrant_caravans

[3] 2022. StackOverflow. https://snap.stanford.edu/data/sx-stackoverflow.html

[4] 2022. Twitter. https://snap.stanford.edu/data/twitter7.html

[5] 2022. WikiTalk. https://snap.stanford.edu/data/wiki-Talk.html

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs.DS/0310049

[7] Xiuwen Chena, Jianming Chen, Dengsheng Wua, Yongjia Xiea, and Jing Lic. 2016. Mapping the research trends by co-word analysis based on keywords from funded project. *Procedia Computer Science* 91 (2016), 547–55.

[8] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. 2020. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific reports* 10, 1 (2020), 1–15.

[9] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and streaming algorithms for k-core decomposition. In *International Conference on Machine Learning*. PMLR, 1397–1406.

[10] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (Maximal) Span-cores from Temporal Networks. In *CIKM*.

[11] Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2020. Span-Core Decomposition for Temporal Networks: Algorithms and Applications. *ACM Trans. Knowl. Discov. Data* 15, 1, Article 2 (Dec. 2020), 44 pages.

[12] C. Gao, Z. Wang, X. Li, Z. Zhang, and W. Zeng. 2016. PR-Index: Using the h-Index and PageRank for Determining True Impact. *PLoS ONE* 11, 9 (2016), e0161755.

[13] N. Gleicher and O. Rodriguez. 2018. Removing Additional Inauthentic Activity from Facebook. https://newsroom.fb.com/news/2018/10/removing-inauthentic-activity/ Retrieved on 20th April 2019.

[14] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. Commit: A scalable approach to mining communication motifs from dynamic networks. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 475–489.

[15] J. Kastrenakes. 2019. Twitter keeps losing monthly users, so it's going to stop sharing how many. https://www.theverge.com/2019/2/7/18213567/twitter-to-stop-sharing-mau-as-users-decline-q4-2018-earnings. Retrieved on 11 October 2019.

[16] Vinay Kolar, Sayan Ranu, Anand Prabhu Subramainan, Yedendra Shrinivasan, Aditya Telang, Ravi Kokku, and Sriram Raghavan. 2014. People In Motion: Spatio-temporal Analytics on Call Detail Records. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*. IEEE, 1–4.

[17] R. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai. 2018. Persistent Community Search in Temporal Networks. In *ICDE*. 797–808.

[18] R. Li, J. X. Yu, and R. Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE TKDE* 26, 10 (Oct 2014), 2453–2465.

[19] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92.

[20] Flaviano Morone, Gino Del Ferraro, and Hernán A. Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics* 15, 1 (2019), 95–102. https://doi.org/10.1038/s41567-018-0304-8

[21] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for K-core Decomposition. *Proc. VLDB Endow.* 6, 6 (April 2013), 433–444. https://doi.org/10.14778/2536336.2536344

[22] Neha Sengupta, Michael Hamann, and Dorothea Wagner. 2017. Benchmark generator for dynamic overlapping communities in networks. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 415–424.

[23] C. Shao, P.-M. Hui, L. Wang, X. Jiang, A. Flammini, F. Menczer, and G. L. Ciampaglia. 2018. Anatomy of an online misinformation network. *Plos One* 13, 4 (April 2018).

[24] K. Shin, T. Eliassi-Rad, and C. Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis — Patterns, Anomalies and Algorithms. In *ICDM*. 469–478.

[25] G. Szekeres and Herbert S. Wilf. 1968. An inequality for the chromatic number of a graph. *Journal of Combinatorial Theory* 4, 1 (1968), 1 – 3. https://doi.org/10.1016/S0021-9800(68)80081-X

[26] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *Proc. 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'2008)*. 990–998.

[27] Zach Verdin, Brett Horvath, and Alicia Serrani. 2018. /voterfraud. https://www.iwr.ai/voterfraud/index.html. Retrieved on 11 July 2019.

[28] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1463–1480.

[29] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE*. 337–348.