Querying Structural Diversity in Streaming Graphs

Kaiyu Chen
The University of New South Wales
Australia
kaiyu.chen1@unsw.edu.au

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

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

Ying Zhang*
Zhejiang Gongshang University
China
ying.zhang@zjgsu.edu.cn

Xiaoyang Wang
The University of New South Wales
Australia
xiaoyang.wang1@unsw.edu.au

Xuemin Lin
Shanghai Jiao Tong University
China
xuemin.lin@sjtu.edu.cn

ABSTRACT

Structural diversity of a vertex refers to the diversity of connections within its neighborhood and has been applied in various fields such as viral marketing and user engagement. The paper studies querying the structural diversity of a vertex for any query time windows in streaming graphs. Existing studies are limited to static graphs which fail to capture vertices' structural diversities in snapshots evolving over time. We design an elegant index structure to significantly reduce the index size compared to the basic approach. We propose an optimized incremental algorithm to update the index for continuous edge arrivals. Extensive experiments on real-world streaming graphs demonstrate the effectiveness of our framework.

PVLDB Reference Format:
Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. Querying Structural Diversity in Streaming Graphs. PVLDB, 17(5): 1034 - 1046, 2024.
doi:10.14778/3641204.3641213

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/kevinChnn/structural-diversity-queries.

1 INTRODUCTION

In graph analysis, structural diversity [21] of a vertex is the number of connected components with sizes exceeding a predefined threshold in the induced subgraph of its neighbors. The connected components in a graph refer to subsets of vertices where each vertex is connected to other vertices in the subset, while vertices in different subsets are not connected. In social contagion, an individual's engagement with a phenomenon strongly correlates to the structural diversity of the individual's contact neighborhood. Individuals with diverse connections within their networks are more likely to engage with a particular idea or behavior. Previous studies have applied the concept of structural diversity to analyze the growth of Facebook users and predict the social contagion process [21].

*Ying Zhang is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641213

Applications. Structural diversity has been applied in various fields, such as user recruitment, user engagement, and viral marketing. We present several instances as follows.
- *User Engagement.* User engagement is a crucial metric for social networks. In a previous work [4], researchers utilized structural diversity to predict potential new paying customers, which can effectively advertise against potential customers and increase the conversion rate. In addition, structural diversity can be applied to recommendation systems [13, 18], as well as to analyze the strength of connections in social networks [19], and also to select trending topics [31]. By leveraging structural diversity, these applications enhance user engagement and promote active user participation.
- *Virality Prediction.* Virality in social networks is often characterized by a significant increase in spread. Previous research has shown that structural diversity can be used to effectively differentiate between viral and non-viral cascades, demonstrating its potential for understanding and predicting information propagation [6, 7]. In addition, structural diversity finds utility in predicting the popularity of content [1]. By leveraging structural diversity, these applications enable better prediction and management of viral phenomena.

Existing Studies. Existing techniques for structural diversity are most limited to static graphs. They study top-k structural diversity search, aiming to identify the k vertices with the highest structural diversity [2, 11, 12]. Building upon this line of research, [32] explores top-k edge structural diversity search, aiming to identify the k edges with the highest structural diversity. In addition to the top-k search, various structural diversity models have been proposed, such as the truss-based structural diversity model [8] to enhance the decomposability for analyzing large-scale networks and the parameter-free structural diversity model [9, 10] to address the model sensitivity problem associated with the size threshold.

Real graphs often exhibit dynamic behaviors and are presented as data streams that undergo continuous changes. Streaming graphs provide a way to represent and analyze the evolving nature of complex systems, making them well-suited for modeling and analyzing real-world phenomena, enabling the identification of interaction patterns, change monitoring, and real-time analysis [22, 23, 27]. Typical application scenarios of streaming graphs include co-authorship in collaboration networks, emails in communication networks, user messages in social networks, etc.

Our Problem. To the best of our knowledge, the problem of querying structural diversity in streaming graphs has not been studied. Given a streaming graph, we study the problem of computing the

structural diversity of a query vertex in the snapshot of an arbitrary query window, a.k.a. historical queries [25, 26, 30]. The snapshot is the induced graph of all edges arriving in the query time window. The application of querying structural diversity in streaming graphs is a natural extension of its application in static graphs. For instance, by analyzing streaming data in real-time, we can predict the popularity of content without delay and adjust marketing strategies accordingly. Furthermore, streaming graph processing techniques can be extended to accelerate offline analysis. Given the continuous edge insertions in streaming graphs, it is infeasible to create a new index from scratch each time to support historical queries. Therefore, we aim to incrementally maintain an index for efficient update and query processing. Note that our techniques can be also extended to handle the sliding window model [3, 5, 16, 17], which is widely studied in streaming graphs. The sliding window model only studies queries for the latest window with a fixed window size. Our experiments demonstrate that our approach for the sliding window model is more efficient than a reasonable baseline.

Straightforward Solutions and Challenges. Existing works observe that structural diversity computation is closely related to triangle enumeration [2, 11, 12]. A straightforward online method is to collect the neighbors of the query vertex in the snapshot and then compute the vertex structural diversity by listing and processing all its triangles. The computation of its triangles requires its two-hop neighbors and takes $O(d^2)$ time, where d is the average vertex degree. However, the vertex degree in real-world graphs can be very large, and users may investigate the structural diversity of numerous vertices in downstream applications. The inefficiency of the online solution motivates us to develop an index-based solution. A basic index is to incrementally maintain all vertex structural diversities for all possible query windows. To reduce the index space, an optimization is to assign a total order for all time windows and only record the changed structural diversity values compared with the previous ones in the order. To handle new incoming edges, we compute structural diversities for all new windows. Given the large volume of edges and possible time windows, the basic solution may take huge space and computational costs for handling new edges.

For the huge index space, the pruning effectiveness of the basic solution is limited because of *non-monotonicity* and *unbounded updates* of structural diversity. Considering adding a set of edges to a snapshot, the structural diversity of a vertex may increase or decrease. Therefore, the number of changed structural diversity of a vertex in an order of continuous time windows cannot be bounded. The high computational cost by new edges is for computing all triangles and the structural diversities for all new windows.

Our Approach. To tackle the above challenges, we present a new framework comprising an elegant index structure and an optimized incremental update algorithm. Our framework groups all time windows by their ending time. To reduce the index space, we observe that structural diversity can be effectively derived from an arithmetic combination of two monotonic properties of each vertex, called neighborhood cohesion and size-bounded neighborhood cohesion. The monotonicity means the value never decreases when arbitrary edges are inserted into the snapshot. Given a specific ending time, the monotonicity enables us to bound the number of different (size-bounded) neighborhood cohesion for windows of all start times by the vertex degree.

To handle new arriving edges, we update the index group for the previous end time to that for the new end time. Updating structural diversity is equivalent to updating all triangles. To improve the update efficiency, we losslessly compress triangles for all possible start times into a set of triangles with time labels on each triangle edge, called temporal triangles. To update the index, we directly update the temporal triangles by considering their active times, instead of updating triangles for windows of every start time. Leveraging the temporal triangles, we develop an optimized algorithm for incrementally updating the index when a set of new edges arrives.

Contributions. We summarize the main contributions as follows.

- *A framework for historical structural diversity queries.* We formulate the problem of querying structural diversity for an arbitrary window in streaming graphs. As far as we know, the problem has never been studied. The solution can also be extended for the sliding window model in streaming graphs.
- *An elegant index structure.* We introduce a new concept, named conditional neighborhood cohesion, from which structural diversity can be derived. By exploiting the monotonic properties of the concept, we design a novel index structure, called PNC-Index. The index size is bounded by $O(m^{1.5} + m \cdot \bar{t})$, where m is the number of edges and $\bar{t}$ is a small value in practice.
- *An optimized incremental update algorithm.* We propose a new concept called the temporal triangle to compress triangles for different time windows. By utilizing temporal triangles, we develop an optimized algorithm to incrementally update the index when new edges arrive. The algorithm running time is bounded by $O(\triangle^+ \cdot \log d)$, where $\triangle^+$ represents the state-of-the-art time complexity to incrementally enumerate triangles and d is the average vertex degree.
- *Extensive performance studies.* We conduct extensive experiments on 14 real-world streaming graphs. The results verify the effectiveness of our index and the efficiency of our update algorithm.

2 PRELIMINARIES

We study an undirected streaming graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. Each edge $e \in \mathcal{E}$ is a triplet (u, v, t) where u and v are two terminal vertices and t is the arrival time. We use n and m to denote $|\mathcal{V}|$ and $|\mathcal{E}|$, respectively. We assume that multiple edges may arrive at the same time and are labeled with the same timestamp consequently. We use t_{max} to denote the latest edge arrival time at a certain time point. The snapshot (or the projected graph) $G(V, E)$ over a time window $[t_s, t_e]$ is a simple graph induced by all edges arriving in $[t_s, t_e]$, i.e., $E = \{(u, v) | (u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}, V = \{u | \exists (u, v) \in E\}$.

We use $N(u)$ to denote the neighbors of u. In the context of streaming graphs, $N(u)$ is a set of pairs where each pair includes a neighbor ID and an arrival time of the edge. In the context of a snapshot (a simple graph), $N(u)$ is a set of vertices. We use $G[S]$ to denote the induced subgraph of a vertex set S in a snapshot G and use $V(E)$ to denote all vertices in an edge set E. The neighborhood induced subgraph of a vertex u in a snapshot G, denoted by $G[N(u)]$, is the subgraph of G induced by all neighbors of u.

Definition 1. ***(Structural Diversity [21])*** *Given a snapshot G and a size threshold $\tau \geq 1$, the structural diversity of a vertex u, denoted by $SD(u)$, is the number of connected components in $G[N(u)]$ whose size, measured by the number of vertices, is not smaller than τ.*

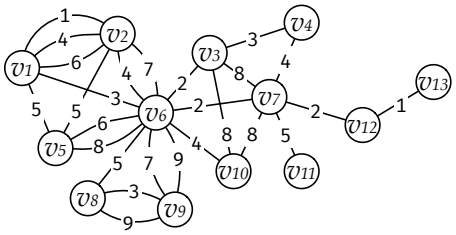


Figure 1: Representing an edge stream as a labeled graph.

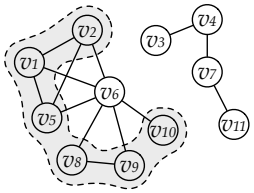


Figure 2: The snapshot of $\mathcal{G}$ over the time window $[3, 7]$ and the neighborhood induced subgraph of v_6.

Example 1. *Figure 1 shows a labeled graph view of an edge stream $\mathcal{G}$. Given the snapshot of $\mathcal{G}$ for $[3, 7]$ shown in Figure 2, The connected components in $G[N(v_6)]$ are $\{v_1, v_2, v_5\}$, $\{v_8, v_9\}$ and $\{v_{10}\}$. If the size threshold τ is set to 2, then $SD(v_6) = 2$.*

Problem Statement. We aim to incrementally maintain an index to efficiently query vertex structural diversity for an arbitrary window $[t_i, t_j]$ with $0 \le i \le j \le t_{max}$ in a streaming graph.

We will mainly study techniques for arbitrary window queries in this paper. Our techniques can be extended for the sliding window queries, which is commonly used in many streaming graphs studies.

Sliding Window Query. Given a predefined fixed time duration (window size) θ, the sliding window query aims to incrementally maintain an index to efficiently query vertex structural diversity in $[t_{max} - \theta, t_{max}]$. We will show how to extend our techniques for sliding window queries in Section 6. Our solution achieves higher efficiency compared with the reasonable baseline. For ease of presentation, we assume the timestamp starts from 1 and increases by 1 each time. As a result, we have t_{max} distinct timestamps when the latest edges arrive at t_{max}.

3 STRAIGHTFORWARD METHODS

3.1 The Online Algorithm

Existing methods observe that computing structural diversities is closely related to enumerating triangles [2, 11, 12]. Specifically, given a triangle $\langle u, v, w \rangle$, we identify that v and w are connected in the neighborhood induced subgraph of u. With all triangles of u, we can identify all edges and all connected components in $G[N(u)]$. To maintain the number of connected components, a disjoint-set data structure [20] is often adopted to merge connected neighbors [2, 11, 12]. Given a set of items, the disjoint-set data structure provides two operations, find() and union(). find() returns the identifier of the set containing the input item, and union() merges the sets of two input items. Each set is implemented as a tree structure for find() and union(). Based on the optimizations of *path compression* and *union by size/rank*, both of the operations can be completed in $O(\alpha(n))$ amortized time [20], where $\alpha()$ is the inverse Ackermann function and n is the number of items. $\alpha(n)$ is less than 5 in practice. In Algorithm 1, we present the pseudocode of computing the structural diversity of a query vertex u in a simple graph G, which is self-explanatory. It computes triangles of u and adopts the disjoint-set structure to incrementally compute the number of connected components for u.

Algorithm 1: SD-Online

```
Input: a graph G, a threshold τ, a vertex u
Output: structural diversity of u
1  if τ > 1 then SD(u) = 0;
2  else SD(u) = deg(u);
3  initialize a disjoint-set for N(u);
4  foreach v ∈ N(u) do
5      mark v as visited;
6  foreach v ∈ N(u) do
7      foreach w ∈ N(v) : v ≺ w do
8          if w is visited then UpdateSD(u, N(u), v, w);
9  Procedure UpdateSD(u, S, v, w)
10     rv ← S.find(v);
11     rw ← S.find(w);
12     if rv = rw then return;
13     if rv.size ≥ τ then SD(u) ← SD(u) − 1;
14     if rw.size ≥ τ then SD(u) ← SD(u) − 1;
15     r ← S.union(v, w);
16     if r.size ≥ τ then SD(u) ← SD(u) + 1;
```

Lemma 1. *The time complexity of Algorithm 1 is $O(d^2)$ where d is the average degree.*

Note that computing structural diversities of all vertices can be achieved in $O(m^{1.5})$, where $O(m^{1.5})$ is the time complexity to enumerate all triangles [14, 15, 28].

Solving Our Problem. To compute structural diversity for a query window, we can store neighbors and corresponding timestamps of each vertex in chronological order. New coming edges are naturally appended to the end of neighbor lists. We still use d to represent the average number of edges connecting to a vertex. It takes $O(\log d)$ time to locate the first neighbor of a vertex in the snapshot. Computing triangles of u requires two-hop neighbors of u in the snapshot. Therefore, the online algorithm to compute the structural diversity of a vertex takes $O(d \cdot \log d + d^2)$, which can be reorganized as $O(d^2)$. The degree of a vertex in real graphs can be very large, and users may investigate the structural diversity of multiple vertices. The efficiency of the online method is not satisfactory.

3.2 A Straightforward Index

A straightforward idea is to incrementally maintain structural diversities of all vertices for all time windows. Assume the next edges arrive at t_{new}. We update the index by computing vertex structural diversities of all additional windows, i.e., $[t_s, t_{new}]$ for

Algorithm 2: Base-Update

Input: t_{new} and E_{new}
Output: the updated structural diversities for all start times

1 **foreach** *possible start time* t_s *in decreasing order* **do**
2 **if** $t_s = t_{new}$ **then**
3 **foreach** $u \in \mathcal{V}$ **do**
4 initialize a disjoint-set for $N_{[t_{new},t_{new}]}(u)$;
5 $T \leftarrow$ compute all triangles in $\mathcal{G}[t_{new}, t_{new}]$;
6 **else**
7 **foreach** $(u, v) \in E_{t_s}$ **do**
8 $N_{[t_s,t_{new}]}(u) \leftarrow N_{[t_s+1,t_{new}]}(u) \cup \{v\}$;
9 $N_{[t_s,t_{new}]}(v) \leftarrow N_{[t_s+1,t_{new}]}(v) \cup \{u\}$;
10 $T \leftarrow$ compute new triangles by adding edges E_{t_s} to the snapshot $\mathcal{G}[t_s + 1, t_{new}]$;
11 **foreach** $\langle u, v, w\rangle \in T$ **do**
12 UpdateSD$(u, N_{[t_s,t_{new}]}(u), v, w)$;
13 UpdateSD$(v, N_{[t_s,t_{new}]}(v), u, w)$;
14 UpdateSD$(w, N_{[t_s,t_{new}]}(w), u, v)$;

all $1 \le t_s \le t_{new}$. Specifically, the approach computes triangles for all snapshots ending at t_{new} (i.e., considering all possible start times) and derives structural diversities accordingly. Instead of enumerating triangles for each window from scratch, we dynamically compute new triangles for each $[t_s, t_{max}]$ from $[t_s + 1, t_{max}]$. The pseudocode for the incremental index update procedure is presented in Algorithm 2. In line 8, $N_{[t_s,t_{new}]}(u)$ represents the neighbors of u in the snapshot of $[t_s, t_{new}]$, i.e., $N_{[t_s,t_{new}]}(u) = \{v|(u, v, t) \in \mathcal{E}, t_s \le t \le t_{new}\}$. In line 10, E_{t_s} denotes the edges arrived at t_s. For each new time t_{new}, we process each possible start time t_s in decreasing order from t_{new} (line 1). When $t_s = t_{new}$, we initialize a disjoint-set structure for each vertex in lines 3–4 and compute all triangles in the snapshot of $[t_{new}, t_{new}]$ (line 5). For subsequent start times, we update the neighbors of each vertex in lines 8–9 and compute new triangles for the snapshot (line 10). Lines 11–14 incrementally update the disjoint-set structure for each vertex, and the structural diversity value is also updated as a result.

Lemma 2. *Given a set of new/expired edges E_{new}, updating triangles can be done in $O(\sum_{\langle u,v\rangle \in E_{new}} \min(deg(u), deg(v)) + m_{inf})$ time, where m_{inf} is the number of edges in the influenced graph, i.e., $m_{inf} = |\sum_{u \in V(E_{new})} deg(u)|$ [29].*

Note that $deg(u)$ in the lemma refers to the degree of u in the new graph. The time complexity in Lemma 2 also holds for the case of computing all triangles in the induced subgraph of E_{new} (line 5). For simplicity, we use $\triangle^+$ to denote the above time complexity of updating triangles. More technical details on updating triangles will be covered in Section 5.

Lemma 3. *The time complexity of Algorithm 2 is $O(\triangle^+ \cdot t_{max})$, where t_{max} is the number of all possible start times.*

Given a streaming graph with t_{max} distinct timestamps, we perform Algorithm 2 for each arriving time for index construction. Therefore, the total index construction time is $O(\triangle^+ \cdot t_{max}^2)$.

Pruning the Index Space. Let t_{max} be the number of different timestamps at a certain time point. It is easy to see that the basic index space reaches $O(nt_{max}^2)$, which is extremely large. To reduce the index space, an optimization is to assign a total order for all time windows. For each vertex, the structural diversity for a window would not be indexed if it is the same as that for the previous window in the order. We follow the order of windows processed in Algorithm 2. Given the new time t_{new}, the algorithm first processes $[t_{new}, t_{new}]$ and decreases the starting time to 1. Given two windows $[t_s, t_e]$ and $[t'_s, t'_e]$ in the order, we have $[t_s, t_e] \prec [t'_s, t'_e]$ if 1) $t_e < t'_e$ or 2) $t_e = t'_e \wedge t_s > t'_s$. For each new window (at the end of each iteration of line 1), we check if the structural diversity of a vertex is the same as that in the previous window. If so, the value is pruned. This optimized index structure is referred to as Base-Index. We only record the changed structural diversity value and the corresponding window. Consequently, the index size is reduced to $O(n \cdot t_{base})$, where t_{base} is the average number of values stored for each vertex and $t_{base} < t_{max}^2$. Given a vertex and a window, it takes $O(\log t_{base})$ time to query the structural diversity of the vertex via binary search. The value of t_{base} depends on the update frequency of structural diversity values across consecutive windows. We report t_{base} for real-world datasets in our experiments.

4 OUR APPROACH

4.1 Drawbacks of the Baseline

Considering the index method in Section 3.2, one of the drawbacks lies in the extremely large space usage. The structural diversity of a vertex may frequently change when varying the window. That degrades the effectiveness of the pruning rule. The reasons for frequent structural diversity updates are twofold. The first one is the *non-monotonicity* of the structural diversity. Given a simple graph G and a set of new edges, the structural diversity of an arbitrary vertex u in G can either increase or decrease. The second reason is *unbounded updates* of the structural diversity. The structural diversity value of a vertex is bounded by its degree in a simple graph. However, the structural diversity of a vertex u is computed via two-hop neighbors, and the structural diversity may change even though u does not have any new neighbors. Because of these two reasons, the structural diversity of each vertex may change many times when varying the window. Given a streaming graph $\mathcal{G}$ and a vertex u, the number of different structural diversity values of u for all windows ending at t_{max} can reach $O(deg(u)^2)$ where $deg(u)$ is the number of edges connecting u in $\mathcal{G}$. Considering all vertices and all possible end times, the total space will be $O(n \cdot t_{max} \cdot d^2)$ where d is the average degree. Although the baseline method optimizes the index structure by only maintaining the changing structural diversity values and their corresponding timestamps, the worst-case space is still large according to the above analysis. The other drawback of the basic index is the inefficiency of incremental updates. The algorithm updates triangles for t_{max} times of edge insertions. It is costly when edges continuously arrive and t_{max} increases.

4.2 A Novel Index via Monotonic Properties

The New Framework. We propose a new framework to address the above challenges. Our idea breaks down the problem into two sub-problems. The first is to index structural diversities of each

v_6	NC	NC-Time	SNC	SNC-Time
	2	5	4	5
	3	3	5	3
	5	2	8	2

Figure 3: The NC and SNC hierarchies of v_6 for $t_e = 9$ ($\tau = 2$) in the streaming graph $\mathcal{G}$ of Figure 1.

vertex for all windows ending at a time t_{max}. We have observed several properties to reduce the index size to a reasonably small value with a theoretical guarantee. The second sub-problem is to design an algorithm to update the index when new edges arrive and t_{max} increases to $t_{max} + 1$. For each vertex, the index for the new ending time is pruned if it is the same as the last one. In the rest, we mainly discuss how to address the two sub-problems.

Index for One End Time. To index structural diversities effectively, we propose a novel index method based on the discovery of two monotonic vertex properties. Let $SD_{cond}(u)$ denote the number of connected components with sizes satisfying the condition *cond*. For instance, $SD_{\geq\tau}(u) = SD(u)$ and $SD_{<\tau}(u)$ represents the number of connected components with size smaller than τ in the neighborhood induced subgraph of u. We define a new metric called *conditional neighborhood cohesion* as follows.

Definition 2. ***(Conditional Neighborhood Cohesion)*** *Given a graph snapshot G, a condition cond and a vertex u, the conditional neighborhood cohesion of u, denoted by $NC_{cond}(u)$, is defined as $NC_{cond}(u) = deg(u) - SD_{cond}(u)$.*

Based on Definition 2, we pick two special conditions, $NC_{>0}(u)$ and $NC_{<\tau}(u)$. When it is clear from the context in the rest, we call them *neighborhood cohesion* (NC) and *size-bounded neighborhood cohesion* (SNC), respectively. We represent $NC_{>0}(u)$ and $NC_{<\tau}(u)$ as $NC(u)$ and $SNC(u)$ for simplicity, respectively. The structural diversity of a vertex can be represented by the neighborhood cohesion and size-bounded neighborhood cohesion as follows:

$$SD(u) = SNC(u) - NC(u) \tag{1}$$

Example 2. *Given the snapshot of $\mathcal{G}$ for $[3, 7]$ and $\tau = 2$, we have $NC(v_6) = deg(v_6) - SD_{>0}(v_6) = 3$ and $SNC(v_6) = deg(v_6) - SD_{<\tau}(v_6) = 5$. Thus, $SD(v_6) = SNC(v_6) - NC(v_6) = 2$.*

Representing structural diversity by NC and SNC is motivated by their monotonicity, which is shown as follows.

Lemma 4. *Given a graph snapshot G and a vertex u, $NC(u)$ never decreases when inserting new edges to G.*

Lemma 5. *Given a graph snapshot G and a vertex u, $SNC(u)$ never decreases when inserting new edges to G.*

Definition 3. ***(NC-Time and SNC-Time)*** *Given a vertex u in a streaming graph $\mathcal{G}$, an end time t_e and an integer k, the NC (resp. SNC) time of u for k is the largest timestamp t_s such that $NC(u) = k$ (resp. $SNC(u) = k$) in the snapshot of $[t_s, t_e]$.*

To handle all start times, our idea is to maintain the NC time and the SNC time for all possible NC values and SNC values of each vertex, respectively. This allows us to derive the NC and SNC values for an arbitrary start time based on Lemma 4 and Lemma 5. We take NC as an example. Given an end time t_e, when increasing the window size (i.e., decreasing the start time t_s from t_e), we keep track of each changed value of $NC(u)$ for each vertex u, and the corresponding timestamp t_s is the NC-time for NC (u). When t_s reaches the earliest timestamp in the streaming graph, we derive the NC-times for all possible NC values of u, and we call it a hierarchy structure of NC values for t_e. We apply the same strategy for SNC.

Example 3. *Figure 3 shows the NC and SNC hierarchies of v_6 for an end time $t_e = 9$. When $NC(v_6) = 2$, the corresponding NC time is 5. When $SNC(v_6) = 4$, the corresponding SNC time is also 5. Note that the NC time is omitted for the case $NC(v_6) = 0$, and similarly for the case $SNC(v_6) = 0$.*

Lemma 6. *Given a streaming graph $\mathcal{G}$ and an arbitrary vertex u, the number of NC values and corresponding NC time of u in the NC hierarchy is bounded by $deg(u)$.*

Lemma 7. *Given a streaming graph $\mathcal{G}$ and an arbitrary vertex u, the number of SNC values and corresponding SNC time of u in the SNC hierarchy is bounded by $deg(u)$.*

Supported by the above lemmas, given the NC and SNC hierarchies for an end time t_e, the structural diversity of a vertex for any arbitrary window ending at t_e can be computed by a total of two binary searches on the hierarchies, respectively.

Handling Different End Times. The index of NC and SNC hierarchies corresponds to a specific end time. When a set of new edges arrive, the number of affected vertices may be limited. As a result, when comparing the hierarchy for two consecutive end times, the neighborhood cohesion hierarchy of many vertices may remain consistent. Motivated by this observation, for each vertex u, we avoid indexing the NC or SNC hierarchies of u for an end time if it remains the same as that of the previous end time. This pruning technique enables us to create a condensed version of the complete index, which we refer to as PNC-Index (Pruned Neighborhood Cohesion). The pruned index minimizes storage usage by only maintaining the necessary information to effectively support historical queries. We use $\bar{t}$ to denote the average number of stored NC and SNC hierarchies for each vertex. The $\bar{t}$ values have been reported in Table 2 for all real datasets evaluated in experiments.

Theorem 1. *The query time complexity for a vertex u based on PNC-Index is $O(\log deg(u) + \log \bar{t})$.*

5 INCREMENTAL COMPUTATION

5.1 Maintaining Temporal Triangles

To efficiently update NC (and SNC) hierarchy for the new time t_{new}, our idea is to locate all affected vertices by the new triangles and update them accordingly, which is similar as Algorithm 2. For all affected vertices, we calculate the new NC (and SNC) hierarchy. Similar to computing the structural diversity, NC and SNC can be computed by considering all triangles of the vertex. A straightforward approach is to maintain a disjoint-set structure for each vertex and each window ending at t_{max}. However, the space cost is prohibitively expensive. To overcome this challenge, we maintain triangles for all possible start times in a compact structure and the disjoint-set data structure is built on the fly based on the updated

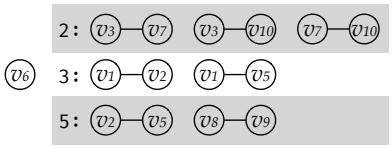


Figure 4: The temporal triangle structure of v_6.

triangles. Instead of maintaining triangles of a simple graph, we consider triangles formed by edges arriving at different times. For clearance, we call them temporal triangles.

Definition 4. *(Active Time) Given a triangle $\langle u, v, w\rangle$ formed by (u, v, t_1), (u, w, t_2), and (v, w, t_3), the active time of the triangle is defined as* $\min(t_1, t_2, t_3)$.

Lemma 8. *Given the latest time t_{max} in a streaming graph $\mathcal{G}$ and a triangle $\langle u, v, w\rangle$ with the active time t, $\langle u, v, w\rangle$ is a triangle in the snapshot of $[t_s, t_{max}]$ if $t_s \le t$.*

Based on Definition 4 and Lemma 8, we can identify all triangles in the snapshot of any starting time if we have all triangles formed by edges in the streaming graph and their corresponding active time. It is easy to observe that triangles belonging to the snapshot of $[t_s, t_{max}]$ must also belong to $[t'_s, t_{max}]$ if $t'_s \le t_s$. This observation motivates us to index all temporal triangles of each vertex in decreasing order of their active time. Given a vertex u and an arbitrary start time t_s, we can derive all triangles of u in the snapshot of $[t_s, t_{max}]$ by sequentially scanning the indexed temporal triangles and terminating once we find a temporal triangle with an active time smaller than t_s. In addition to NC and SNC, our new PNC-Index maintains temporal triangles for each vertex.

Example 4. *The temporal triangle structure of v_6 is presented in Figure 4. We record each temporal triangle using the opposite edge of v_6, and the temporal triangles are grouped by their active times.*

Handling Duplicate Edges. In the context of streaming graphs, it is common to encounter multiple edges with the same pair of terminals but at different times. This leads to the presence of duplicate edges in the graph. One particular issue arising from duplicate edges is the occurrence of duplicate temporal triangles that share the same active time. Fortunately, thanks to the disjoint-set data structure, the correctness of our approach is not affected by these duplicate triangles. To handle the problem of triangle duplication, our techniques can effectively avoid duplicate temporal triangles with the same active time, without incurring any additional theoretical complexity. During the process of scanning each neighbor, we naturally keep track of the arrival time of each neighbor. Leveraging this temporal information, we can check if the active time of newly formed triangles is the same as before by using the previous arrival time of each edge in the triangle. Furthermore, according to Lemma 8, we only need to store the latest active time for each temporal triangle, thereby simplifying the storage space.

Theorem 2. *The space complexity of PNC-Index is $O(m^{1.5} + m \cdot \bar{t})$, where m is the number of edges in the complete snapshot of the streaming graph and $\bar{t} \ll t_{max}$ in practice.*

Proof. Storing all temporal triangles requires $O(m^{1.5})$ space, while maintaining the NC and SNC hierarchies requires $O(m \cdot \bar{t})$ space, as discussed in Section 4.2. □

Algorithm 3: UpdateTriangles(t_{new}, E_{new})

```
   /* add new neighbors for each vertex                          */
 1 foreach ⟨u, v⟩ ∈ E_new do
 2     N[u].push(⟨v, t_new⟩);
 3     N[v].push(⟨u, t_new⟩);
 4 A ← ∅;
   /* initialization for computing new triangles                 */
 5 foreach u ∈ V(E_new) do
 6     New+ ← ∅;
 7     foreach ⟨v, t⟩ ∈ N[u] do
 8         if t = t_new then
 9             if u ≺ v then New+ ← New+ ∪ {v};
10             continue;
11         if u ≺ v then Old+[v] ← t;
12         else Old−[v] ← t;
       /* compute new triangles                                  */
13     foreach v ∈ New+ do
14         foreach ⟨w, t⟩ ∈ N[v] do
               /* Case 3                                         */
15             if t = t_new ∧ v ≺ w ∧ w ∈ New+ then
16                 AddTriangle(u, v, w, t, A);
               /* Case 2.2                                       */
17             else if t = t_new ∧ Old+[w] is defined then
18                 AddTriangle(u, v, w, Old+[w], A);
               /* Case 2.1                                       */
19             else if v ≺ w ∧ w ∈ New+ then
20                 AddTriangle(u, v, w, t, A);
               /* Case 1                                         */
21             else if Old+[w] is defined then
22                 AddTriangle(u, v, w, min(t, Old+[w]), A);
23             else if Old−[w] is defined then
24                 AddTriangle(u, v, w, min(t, Old−[w]), A);
25 return A
26 Procedure AddTriangle(u, v, w, t, A)
27     Tri[u][t].push(⟨v, w⟩);
28     Tri[v][t].push(⟨u, w⟩);
29     Tri[w][t].push(⟨u, v⟩);
30     A ← A ∪ {u, v, w};
```

5.2 Updating Temporal Triangles.

Instead of using the state-of-the-art triangle updating procedure as a black box in Algorithm 2, we extend the algorithm [29] to compute all new temporal triangles in streaming graphs and then update the temporal triangle order for each vertex. The algorithm categorizes new triangles into four cases as shown in Figure 5. The red line represents a pivot edge, the solid line represents an existing edge, and the dotted line represents a newly arrived edge.

- **Case 1:** If there is only a new edge in the triangle, the starting vertex of the new edge is chosen as the pivot vertex, and the new edge is chosen as the pivot edge. As shown in Case 1 of Figure 5, u is the pivot vertex and (u, v) is the pivot edge.

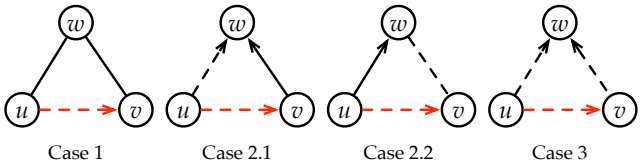


Figure 5: An illustration for dynamic triangle enumeration.

Algorithm 4: PNC-Update

Input: t_{new} and E_{new}
Output: the updated *PNC-Index*
/* compute and store all new triangles for each vertex */
1 $\mathcal{A} \leftarrow$ UpdateTriangles(t_{new}, E_{new});
2 **foreach** $u \in \mathcal{A}$ **do**
/* update NCH for t_{new} */
3 NCH-Update($u, Tri, NCH[t_{new}]$);
/* update $SNCH$ for t_{new} */
4 SNCH-Update($u, Tri, SNCH[t_{new}]$);

- **Case 2:** If there are two new edges, the vertex with two outgoing new edges or the vertex with one outgoing new edge and an outgoing original edge is chosen as the pivot vertex. This is illustrated in both Case 2.1 and Case 2.2 of Figure 5. In Case 2.1, the original edge (u, v) is chosen as the pivot edge. In Case 2.2, (u, v) is also chosen as the pivot edge because u is the pivot vertex and v has an outgoing original edge.
- **Case 3:** If all edges in the triangle newly arrive, the vertex with two outgoing new edges is chosen as the pivot vertex. This is shown in Case 3 of Figure 5, where u is the pivot vertex. As u is the pivot vertex and v has an incoming edge and an outgoing edge, (u, v) is chosen as the pivot edge.

Our algorithm is shown in Algorithm 3. Given a vertex u, $N(u)$ in Algorithm 3 represents all neighbors connecting to u in the streaming graph, with each neighbor associated with an arrival timestamp. Note that the same neighbor may exist at different times. Given the new edge set E_{new} and the new time t_{new}, we update the neighbors of each vertex in lines 1–3. $\mathcal{A}$ will be the set of all vertices contained in new triangles when the algorithm terminates. Lines 5–24 compute all new triangles. $V(E_{new})$ represents the set of all vertices appearing in the new edge set E_{new}. For each vertex u in line 5, New^+ maintains all new out neighbors of u, while Old^+ and Old^- are arrays that store the timestamp of the corresponding out neighbors and in neighbors before inserting E_{new}, respectively. The cases in lines 14–24 correspond to the cases in Figure 5. Each new triangle is indexed for each vertex using the procedure AddTriangle(). As mentioned earlier, the triangles of each vertex are arranged in non-increasing order of their active time. To achieve this, a binary search tree is used for each vertex u. The key of each item in the search tree is an active time t, and the value is a list of vertex pairs $\langle v, w \rangle$, indicating the existence of a triangle $\langle u, v, w \rangle$ exists with an active time t.

Algorithm 5: NCH-Update(u, Tri, NCH)

1 initialize a disjoint-set for all vertices in $N[u]$;
2 $nc \leftarrow 0$;
3 $NCH[u] \leftarrow$ an empty array;
4 **foreach** t *in decreasing order of* $Tri[u]$ **do**
5 $update \leftarrow$ false;
6 **foreach** $\langle v, w \rangle \in Tri[u][t]$ **do**
7 **if** find(v) = find(w) **then continue**;
8 $update \leftarrow$ true;
9 union(v, w);
10 $nc \leftarrow nc + 1$;
11 **if** $update$ **then** $NCH[u].push(\langle t, nc \rangle)$;

Algorithm 6: SNCH-Update($u, Tri, SNCH$)

1 initialize a disjoint-set for all vertices in $N[u]$;
2 $nc \leftarrow 0$;
3 $SNCH[u] \leftarrow$ an empty array;
4 **foreach** t *in decreasing order of* $Tri[u]$ **do**
5 $update \leftarrow$ false;
6 **foreach** $\langle v, w \rangle \in Tri[u][t]$ **do**
7 $rv \leftarrow$ find(v), $rw \leftarrow$ find(w);
8 **if** $rv = rw$ **then continue**;
9 $update \leftarrow$ true;
10 **if** $size(rv) < \tau$ **then** $nc \leftarrow nc + 1$;
11 **if** $size(rw) < \tau$ **then** $nc \leftarrow nc + 1$;
12 $r \leftarrow$ union(v, w);
13 **if** $size(r) < \tau$ **then** $nc \leftarrow nc - 1$;
14 **if** $update$ **then** $SNCH[u].push(\langle t, nc \rangle)$;

5.3 The algorithm

Our overall updating framework is in Algorithm 4. When a set of new edges arrives at t_{new}, we update the PNC-Index from the previous latest time t_{max} to the new latest time t_{new}. Given the affected vertices derived in line 1, we update the NC hierarchy and the SNC hierarchy for t_{new} in line 3 and line 4, respectively. We first discuss how to compute the NC hierarchy of affected vertices. The definition of NC implies the following lemma.

Lemma 9. *Given a vertex u in a simple graph and a set of new edges, the neighborhood cohesion of u changes only if there is a new triangle containing u.*

The pseudocode is presented in Algorithm 5. Based on Lemma 9, we only consider the active time of all triangles. Recall that a triangle of u with an active time t means the triangle does not exist in the snapshot of $[t+1, t_{max}]$ but exists in the snapshot of $[t, t_{max}]$ where t_{max} is the latest arrival time of edges. Once two connected components are merged (line 9), we increase the NC value of u by 1. At the end of each iteration (line 11), nc is $NC(u)$ for the snapshot of $[t, t_{new}]$. Computing SNCH is similar to computing NCH but additionally considers the size of connected components. The pseudocode is presented in Algorithm 6. When merging connected

Table 1: Comparison of different solutions for historical queries. n is the number of vertices, m is the number of edges, d is the average degree, t_{max} is the number of unique time labels, t_{base} is the average number of values stored for each vertex in Base-Index, $\bar{t}$ is a pruning factor in PNC-Index, $deg(u)$ is the degree of the query vertex u, $\triangle_u$ is the number of triangles containing u, and $\triangle^+$ is the state-of-the-art time complexity to incrementally enumerate triangles.

	Query Time	Index Space	Update Time
Online	$O(d^2)$	$\oslash$	$\oslash$
Baseline	$O(\log t_{base})$	$O(n \cdot t_{base})$	$O(\triangle^+ \cdot t_{max})$
NHCC	$O(deg(u))$	$O(m^{1.5})$	$\times$
HT	$O(\triangle_u)$	$O(m^{1.5} \cdot \bar{t})$	$O(\triangle^+)$
Ours	$O(\log deg(u) + \log \bar{t})$	$O(m^{1.5} + m \cdot \bar{t})$	$O(\triangle^+ \cdot \log d)$

components, nc is increased by one for each original small-size ($< \tau$) connected component (lines 10–11) and is decreased by one if the newly connected component is still small (line 13). Once the size-bounded neighborhood cohesion value changes, we add it to the hierarchy (line 14).

Theorem 3. *The time complexity of Algorithm 4 is $O(\triangle^+ \cdot \log d)$ where $\triangle^+$ is the state-of-the-art time complexity to incrementally enumerate triangles and d is the average degree.*

5.4 Comparing with Other Potential Solutions

We provide Table 1 to summarize the theoretical complexity of our approach compared with other solutions for historical structural diversity queries. We additionally discuss two methods below.

Historical Connected Components. An index-based solution has been studied for querying historical connected components [26]. Based on the definition of structural diversity, another baseline solution is to construct an index for historical connected components for the neighborhood induced subgraph of each vertex. For query processing of a vertex, we derive all connected components in its neighborhood induced subgraph and calculate the structural diversity. We refer to the method as NHCC (Neighborhood Historical Connected Components). An immediate drawback of NHCC lies in the lack of support for incremental updates. It is only for static temporal graphs where all edges with different arriving times are given. The index construction time and index space in [26] are $O(m \cdot t_{max})$ and $O(m)$, respectively. To index the connected components in the neighborhood induced subgraph of a vertex u, the number of edges in its subgraph is bounded by the number of triangles containing u. Therefore, the index construction time and index space of NHCC are $O(m^{1.5} \cdot t_{max})$ and $O(m^{1.5})$ respectively, where $O(m^{1.5})$ represents the number of all triangles in the graph. Even only considering a static temporal graph, we can run Algorithm 4 for all edges chronologically, and our time complexity is much smaller than that of NHCC. Querying all historical connected components in [26] is bounded by $O(n)$. Therefore, querying the structural diversity of a vertex u in NHCC is bounded by $O(deg(u))$, while our query time complexity is much smaller.

Historical Triangles. Based on the techniques of maintaining temporal triangles (Section 5.1), another potential solution, referred to as HT (Historical Triangles), is to just maintain temporal triangles for different end times. Specifically, given a vertex u and its temporal triangles for the end time t_e, we store the triangles in the index only if they are not the same as those for the end time $t_e - 1$. Therefore, the overall index space complexity of HT is $O(m^{1.5} \cdot \bar{t})$. To compute the structural diversity of a vertex u for a query window $[t_s, t_e]$ based on HT, we start by performing a binary search to find the temporal triangles for the end time t_e and then another binary search to locate all triangles between t_s and t_e. Both searches are bounded by $O(\log \triangle_u)$ time, where $\triangle_u$ is the number of triangles containing u. The structural diversity can then be derived by scanning the triangles (i.e., the edges in the neighborhood induced subgraph). Therefore, the overall query time complexity of HT is $O(\triangle_u)$. Similar to Algorithm 4, each update requires $O(\triangle^+)$ time, resulting in a total construction time of $O(\triangle^+ \cdot t_{max})$.

Algorithm 7: SW-Base-Update

Input: E_{new} and E_{old}
Output: the updated structural diversity

```
1  A ← ∅;
2  foreach ⟨u, v⟩ ∈ E_new do
3      update N(u) and N(v);
4      A ← A ∪ {u, v};
5      foreach w ∈ N(u) ∪ N(v) do A ← A ∪ {w};
6  repeat lines 2–5 by replacing E_new with E_old;
7  update triangles based on E_new and E_old;
8  foreach u ∈ A do
       /* recompute structural diversity          */
9      initialize a disjoint-set for N(u);
10     SD(u) ← 0;
11     foreach triangle ⟨u, v, w⟩ of u do
12         UpdateSD(u, N(u), v, w);
```

6 VARIATIONS

6.1 Sliding Window Queries

In this section, we study the sliding window query processing. To our best knowledge, there is no study on maintaining structural diversity in streaming graphs. Given a sliding window, a naive way is to store the structural diversity for each vertex in the snapshot, resulting in a space complexity of $O(n)$. The sliding window query can be answered in constant time, but the update process is costly. After computing new triangles for new edges, the structural diversity of each vertex in the new triangles may be updated, and we need to recompute the structural diversities for all affected vertices from scratch. The same process will be performed for expired edges.

The Baseline. To avoid the recomputation, a potential baseline is to store all triangles in the snapshot and dynamically update the triangles as edges change. Additionally, we update the structural diversity values of the affected vertices based on the updated triangles, as shown in Algorithm 7. Given the new edge set E_{new} and the old edge set E_{old}, we first update the neighbors of each vertex and record the affected vertices in lines 1–6. Subsequently, line 7 handles the updates of the triangles within the window. Finally, in

lines 8-12, we compute the structural diversity for each affected vertex using the updated triangles. The time complexity of Algorithm 7 is $O(\triangle^+)$ where $\triangle^+$ is the state-of-the-art time complexity to incrementally enumerate triangles. In addition to the space required for storing structural diversity values, an additional $O(m^{1.5})$ space is needed to store the triangles within the window, where m represents the number of edges in the snapshot of the window.

Our Approach. Our solution is to maintain the NC (and SNC) hierarchy of each vertex for the latest time. In other words, we maintain an index that covers all possible windows from an arbitrary start time to the latest time. An immediate benefit is to avoid the process of expired edges in the sliding window model. We can simply discard the indexed values for expired start times. For new edges, the process is the same as Algorithm 4. The theoretical update time is the same as the baseline. However, the practical performance of our method is much better, as verified in Section 7.

6.2 Handling Various Size Thresholds

Our approach can be extended to support various size thresholds for structural diversity. To this end, we can maintain $|\tau|$ SNC hierarchies, where $|\tau|$ represents the number of possible size thresholds. Each SNC hierarchy is associated with a specific size threshold, enabling us to adapt our approach accordingly. Note that the NC hierarchy is independent of the size threshold, it is sufficient to maintain only one instance for various size thresholds. Additionally, we retain the temporal triangles as they are. Indexes for different size thresholds can share and leverage the information contained in the temporal triangles to effectively update their respective indexes.

7 PERFORMANCE STUDIES

All algorithms in experiments are implemented in C++ and compiled with the g++ compiler at the -O3 optimization level. All the experiments are conducted on a Linux machine with dual Intel Xeon Gold 6342 2.8GHz CPUs and 512GB RAM. In the experiments, the size threshold τ is set to 2, which is commonly adopted in prior studies [2, 11, 12]. We evaluate the approaches on 14 publicly available real-world streaming graphs. Detailed statistics of the graphs are given in Table 2. Except for the CollegeMsg dataset, which is from SNAP[1], the other 13 datasets are from the KONECT[2] project.

7.1 Incremental Update

In this experiment, we evaluate the efficiency of different incremental update algorithms, including Base-Update, NHCC, HT, and our final solution PNC-Update. We evaluate the algorithms by performing incremental updates for each time label of each dataset until all updates are completed and record the cumulative running time. Note that NHCC does not support incremental updates, we record its running time by directly giving all edges with different time labels. Figure 6 reports the cumulative running time of the algorithms. The running time of Base-Update and NHCC for several datasets is not reported since they cannot be completed in 12 hours. For HT, it runs out of memory for several datasets. We can see that for all datasets, PNC-Update is several orders of magnitude faster than Base-Update and NHCC, thanks to our new concept of

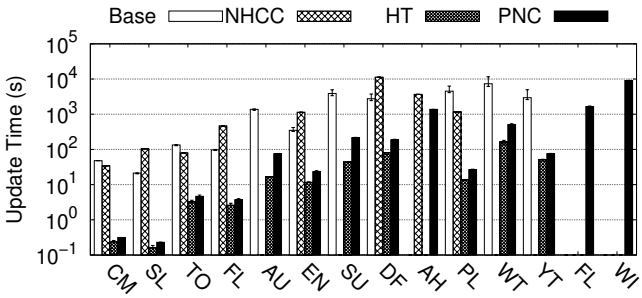


Figure 6: Cumulative incremental update time.

temporal triangles. For the smallest dataset CM, PNC-Update takes less than 1s, while Base-Update and NHCC take about 50s and 35s, respectively. For the large datasets WT and YT, PNC-Update completes in about 8min and 1min, respectively, while Base-Update takes about 100min and 45min to complete, respectively. We also notice that HT is slightly faster than PNC-Update because it saves time in updating the SNC/NC hierarchies. However, this comes at the cost of much slower query time and a much larger index size. Table 3 reports the percentage of running time on updating temporal triangles and SNC/NC hierarchies in PNC-Update. The majority of time in most datasets is on updating the temporal triangles.

7.2 Index Size

In this experiment, we compare the size of different index structures. We record the index size after all updates are completed. Figure 7 reports the size of the index structures. The size of multiple index structures for several datasets is not reported due to timeout or insufficient memory. We can see that our PNC-Index is about 1 order of magnitude smaller than the Base-Index and HT on average. For example, for the SU dataset, our PNC-Index takes about 2.5GB, while the Base-Index and HT require about 17.8GB and 22.2GB, respectively. Although NHCC is smaller in size than our PNC-Index, it comes at the cost of extremely slow query efficiency, extremely inefficient index construction, and no support for incremental updates.

Pruning Effect. The final PNC-Index includes two kinds of improvements in Section 4.2, i.e., the pruning rule for one end time and the pruning rule for different end times. We examine the effect of two pruning rules by reporting the NC-Index size in Figure 7. Compared with PNC-Index, the NC-Index only adopts the first pruning rule, which directly stores the indexes of all end times. We can see that the pruning rule for different end times is more effective. For example, for the SL dataset, NC-Index is about 3 times smaller than Base-Index, while PNC-Index is about 12 times smaller than NC-Index. Furthermore, we can observe that the overall pruning effect depends on the practical graph structure.

PNC-Index Breakdown. Table 4 reports the percentage of space used to store temporal triangles and SNC/NC hierarchies in our PNC-Index. We can see that the percentages vary considerably across different datasets.

7.3 Query Processing

In this experiment, we compare the efficiency of different query processing algorithms, including SD-Online, the straightforward

[1] https://snap.stanford.edu/
[2] http://konect.cc/

Table 2: Statistics of datasets. n is the number of vertices, m is the number of edges, m^* is the number of edges with unique terminals, d is the average degree, t_{max} is the number of unique time labels, t_{base} is the average number of values stored for each vertex in Base-Index, and $\bar{t}$ is a pruning factor in PNC-Index.

	n	m	m^*	d	Type	t_{max}	t_{base}	$\bar{t}$
CollegeMsg (CM)	1,899	59,835	13,838	63.02	Communication	3320	7914.36	13.23
Slashdot (SL)	51,083	140,778	116,573	5.51	Communication	384	429.92	0.52
Topology (TO)	34,761	171,403	107,720	9.86	Computer	556	800.37	3.49
FacebookWall (FW)	46,952	876,993	183,412	37.36	Communication	1473	1882.45	7.82
AskUbuntu (AU)	159,316	964,437	455,691	12.11	Online Contact	2059	2160.56	1.06
Enron (En)	87,273	1,148,072	297,456	26.31	Communication	1235	1391.92	5.96
SuperUser (SU)	194,085	1,443,339	714,570	14.87	Online Contact	2629	2808.22	1.96
DiggFriends (DF)	279,630	1,731,653	1,548,126	12.39	Online Social	1434	1636.07	5.57
arXivHepPh (AH)	22,908	4,596,803	3,148,447	401.33	Citation	2337	N/A	121.72
ProsperLoans (PL)	89,269	3,394,979	3,330,022	76.06	Interaction	1259	2361.19	6.15
WikiTalk (WT)	1,140,149	7,833,140	2,787,967	13.74	Communication	2166	2271.16	2.73
YouTube (YT)	3,223,589	9,375,374	9,375,374	5.82	Online Social	203	236.87	1.19
Flickr (FL)	2,302,925	33,140,017	22,838,276	28.78	Online Social	134	N/A	4.38
Wikipedia (WI)	1,870,709	39,953,145	36,532,531	42.71	Hyperlink	2198	N/A	27.86

Table 3: PNC-Update time breakdown.

PNC-Update	CM	SL	TO	FW	AU	EN	SU	DF	AH	PL	WT	YT	FL	WI
Temporal Triangle	2.87%	9.64%	17.81%	3.53%	1.72%	7.43%	1.59%	13.22%	82.29%	4.76%	3.30%	23.19%	83.44%	3.76%
SNC/NC Hierarchy	97.13%	90.36%	82.19%	96.47%	98.28%	92.57%	98.41%	86.78%	17.71%	95.24%	96.70%	76.81%	16.56%	96.24%

Table 4: PNC-Index size breakdown.

PNC-Index	CM	SL	TO	FW	AU	EN	SU	DF	AH	PL	WT	YT	FL	WI
Temporal Triangle	35.90%	52.21%	12.32%	40.47%	5.25%	18.23%	4.02%	10.72%	5.03%	37.49%	8.93%	42.44%	29.75%	7.40%
SNC/NC Hierarchy	64.10%	47.79%	87.68%	59.53%	94.75%	81.77%	95.98%	89.28%	94.97%	62.51%	91.07%	57.56%	70.25%	92.60%

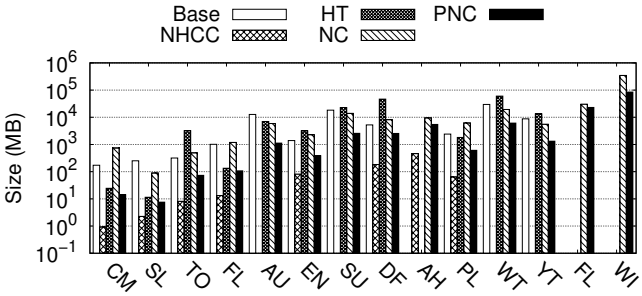


Figure 7: Index size of all datasets.

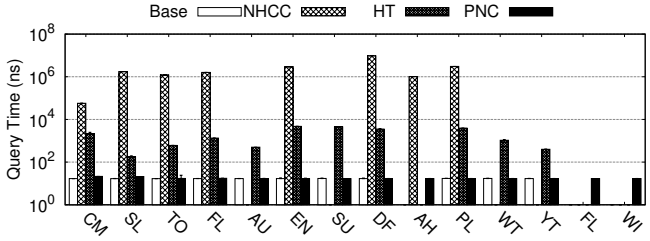
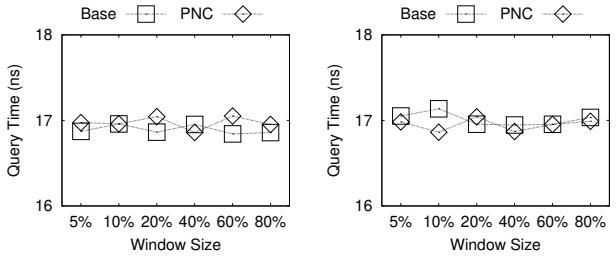


Figure 8: Average query time.

index-based Base-Query, NHCC, HT, and our final PNC-Query. For the index-based algorithms, we perform queries after the index structures are completely updated. Based on different datasets, we vary the query window size from 5% to 80% of t_{max} for each dataset, and the default is set to 60% of t_{max}. We perform 1,000 queries, each using a random vertex and a random query window with the given window size, and record the average running time of each query.

Overall Query Efficiency. We evaluate the overall query efficiency with the default window size (60% of t_{max}) for each dataset. SD-Online is 5–7 orders of magnitude slower than PNC-Query and is not reported for clearance. Figure 8 reports the average running time of the other query algorithms with the default query window size, except for the case of unsuccessful index construction. We can see that for all datasets, NHCC is the least efficient, being at least 3 orders of magnitude slower than PNC-Query, and HT is also significantly slower than PNC-Query. For example, for the DF dataset, PNC-Query takes an average of 17ns per query, while NHCC and HT take an average of $9.7 * 10^6$ns and $3.6 * 10^3$ns per query, respectively. This is mainly because PNC-Query only requires a few binary searches of the PNC-Index, whereas NHCC and TH require more complex processing. Also note that since both PNC-Query and Base-Query are index-based query algorithms based on binary searches, they have the same level of query efficiency.

Varying Query Window Size. We evaluate the query efficiency of Base-Query and PNC-Query by varying the query window size to 5%, 10%, 20%, 40%, 60%, and 80% of t_{max} for each dataset. Figure 9 reports the average querying time. We report the results of two large representative datasets, WT and YT, and the results of other



(a) WikiTalk (b) YouTube

Figure 9: Average query time by varying window size.

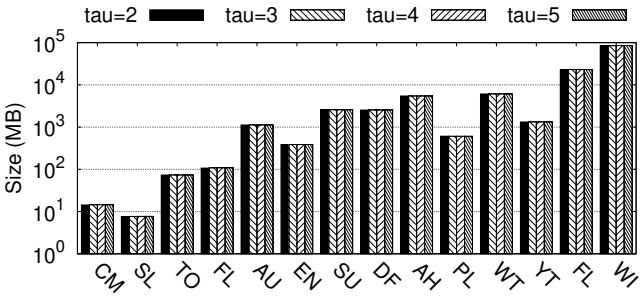


Figure 10: PNC-Index size of all datasets by varying τ.

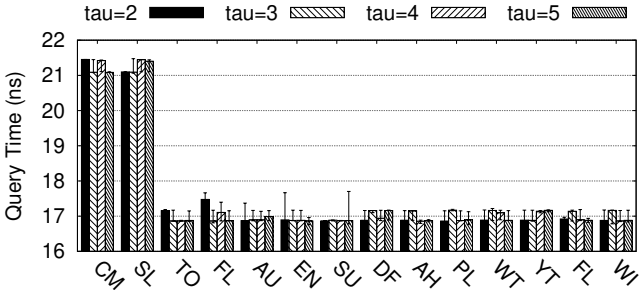


Figure 11: Average PNC-Query time by varying τ.

datasets show similar trends. As in the previous experiments, the efficiency of PNC-Query and Base-Query remains at the same level.

7.4 Varying τ

We study the impact of τ on index size and query efficiency of historical structural diversity queries. We vary the size threshold from $\tau = 2$ to $\tau = 5$.

Index Size. Figure 10 reports the size of PNC-Index under different size thresholds τ. We can see that the size threshold variation has no practical impact on the PNC-Index size.

Query Efficiency. Figure 11 reports the average time of PNC-Query by varying τ. We can see that the average PNC-Query time is stable and almost unaffected by size threshold changes.

7.5 Sliding Window Queries

In this section, we study the performance of the proposed methods for sliding window queries. Similarly, the default sliding window size is set to 60% of t_{max} for each dataset.

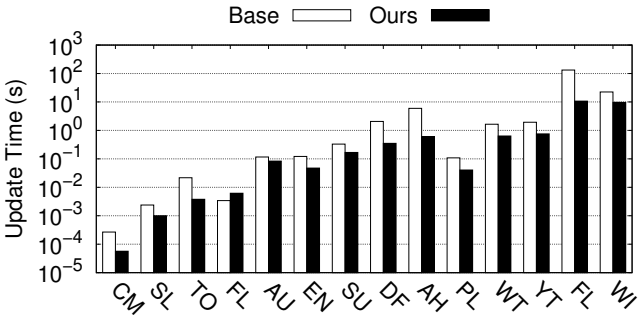


Figure 12: Index update time for sliding window queries.

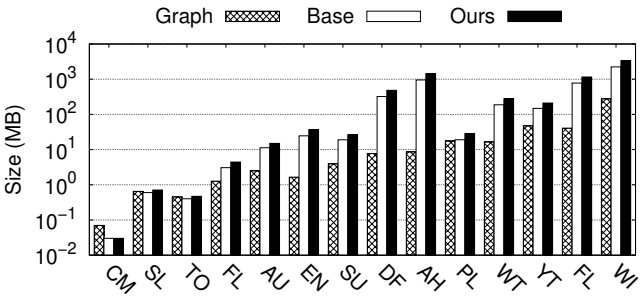
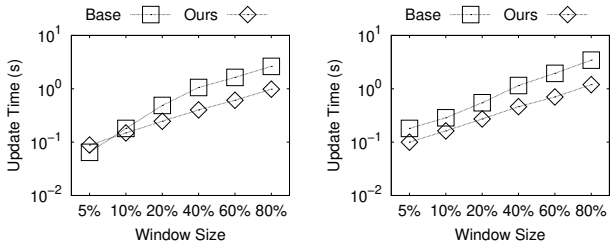


Figure 13: Index size for sliding window queries.

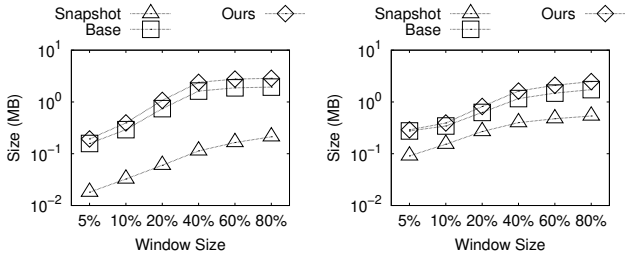
Incremental Update. In this experiment, we evaluate the efficiency of different incremental update algorithms, including the baseline method, referred to as Base, and our optimized method, denoted as Ours. We first initialize the two methods with edges within the first window. We then evaluate both methods by performing incremental updates for each time label of each dataset until all updates are completed and record the average running time of each update. Figure 12 reports the average incremental update time of our method in comparison to the baseline method. We can see that our method is several times faster than the baseline method for most datasets, thanks to the avoidance of processing expired edges. While for the FW dataset, our method is slightly slower than the baseline method because the dataset is right-skewed (i.e., the temporal distribution is uneven, with almost all edges arriving in the second half of t_{max}) and so has almost no expired edges when sliding, our method becomes slow due to the small overhead of maintaining temporal triangles.

Index Size. In this experiment, we compare the size of different index structures, including the baseline index, referred to as Base, and our index, denoted as Ours. We record the size of both index structures after all updates are completed (i.e., the last sliding window), and the snapshot size is the size of the corresponding snapshot in adjacency list format. Figure 13 reports the size of the two index structures with the snapshot size as a reference. We can verify that our index and the baseline index have the same level of space usage. The reason why our index is slightly larger than the baseline index is that our index stores additional temporal information to avoid processing expired edges. Furthermore, we can observe that the index size is close to the snapshot size for most datasets, as we only keep the information within the sliding window.



(a) WikiTalk **(b) YouTube**

Figure 14: Index update time for sliding window queries by varying window size.



(a) WikiTalk **(b) YouTube**

Figure 15: Index size for sliding window queries by varying window size.

Varying Sliding Window Size. To further evaluate the efficiency of the two incremental update algorithms and the size of the two index structures, we vary the sliding window size to 5%, 10%, 20%, 40%, 60%, and 80% of t_{max} for each dataset. Figure 14 reports the average incremental time of the two algorithms by varying sliding window sizes. Figure 15 shows the size of the two index structures with the snapshot size (the size of the corresponding snapshot in adjacency list format) as a reference. Similarly, we report the results of two large representative datasets, WT and YT, and the results of other datasets show similar trends. We can see that the average incremental update time, the index size, and the snapshot size all show a similar upward trend as the sliding window size increases. This is because as the sliding window size increases, there are more edges within the sliding window, which results in larger snapshot sizes and more triangles, which leads to an increase in the average incremental update time and index size.

Query Efficiency. As shown in Figure 8, the query efficiency of the online algorithm is unacceptably slow. For the index-based methods that directly store the structural diversity of all vertices, sliding window queries can be answered in $O(1)$ constant time, almost the same as shown in Figure 9, regardless of the sliding window size. We have omitted the experimental figures for sliding window queries due to the high consistency in query efficiency comparisons with the previous ones and due to space constraints.

8 RELATED WORKS

The concept of structural diversity was initially introduced by Ugander et al. [21], who examined its application in the context of social contagion processes. Subsequently, several variants of the structural diversity model have been proposed. For instance, Zhang et al. [32] proposed an edge-based structural diversity model that focuses on measuring the diversity exhibited by each edge rather than each vertex. Other variations include the k-truss-based model [8] and the k-core-based model [12, 24]. Additionally, to address the model sensitivity associated with the size parameter, Huang et al. [9, 10] proposed a parameter-free structural diversity model. However, these models are tailored to specific requirements, whereas our paper focuses on the classical structural diversity model proposed by Ugander et al. [21], which is more general. Prior research on structural diversity primarily revolves around the top-k structural diversity search, which aims to identify the k objects with the highest level of structural diversity. In [11, 12], a Union-Find-Isolate data structure was devised to maintain the known structural information of each vertex, and an effective upper bound was established for pruning purposes. However, a significant limitation of this approach is that it requires enumerating each triangle up to three times. Moreover, the computation time associated with each enumerated triangle is not amortized constant, rendering it unsuitable for scalability in larger graphs. To address these issues, Chang et al. [2] optimized the algorithm by ensuring that each triangle is enumerated at most once, and they developed efficient techniques to achieve amortized constant computation time per triangle. Nevertheless, both methods are specifically designed for static graphs and cannot be readily extended to streaming graphs.

9 CONCLUSION

In this paper, we study querying historical structural diversity in streaming graphs. We propose a new framework with an elegant index structure and an optimized incremental update algorithm. We also extend our techniques to support sliding window queries. Experimental results demonstrate the effectiveness of our framework. Several potential research directions remain open. For example, two-hop or multi-hop neighbors may be considered for structural diversity models instead of one-hop neighbors of each vertex. A possible approach is to compress and treat them as direct neighbors and then use our framework as usual. The challenge lies in how to update the compressed neighbors efficiently and needs to be addressed in future works. There are also several works studying structural diversity models based on k-core and k-brace instead of connected components. We could extend them for historical queries in future works. In addition, a parallel implementation of our algorithm may be studied to further improve the update efficiency.

ACKNOWLEDGMENTS

Dong Wen is supported by ARC DP230101445 and DE240100668. Wenjie Zhang is supported by ARC FT210100303 and DP230101445. Xiaoyang Wang is supported by ARC DP230101445 and DP240101322. Xuemin Lin is supported by NSFC U2241211, NSFC U20B2046, and GuangDong Basic and Applied Basic Research Foundation 2019B1515120048. This work is also supported by ZJNSF LY21F020012.

REFERENCES

[1] Peng Bao, Huawei Shen, Junming Huang, and Xueqi Cheng. 2013. Popularity prediction in microblogging network: a case study on sina weibo. In *WWW*. 177–178.

[2] Lijun Chang, Chen Zhang, Xuemin Lin, and Lu Qin. 2017. Scalable Top-K Structural Diversity Search. In *ICDE*. 95–98.

[3] Michael S. Crouch, Andrew McGregor, and Daniel M. Stubbs. 2013. Dynamic Graphs in the Sliding-Window Model. In *ESA (Lecture Notes in Computer Science)*, Vol. 8125. 337–348.

[4] Zhanpeng Fang, Xinyu Zhou, Jie Tang, Wei Shao, Alvis Cheuk M. Fong, Longjun Sun, Ying Ding, Ling Zhou, and Jarder Luo. 2014. Modeling Paying Behavior in Game Social Networks. In *CIKM*. 411–420.

[5] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD*. 645–657.

[6] Ruocheng Guo, Elham Shaabani, Abhinav Bhatnagar, and Paulo Shakarian. 2015. Toward Order-of-Magnitude Cascade Prediction. In *ASONAM*. 1610–1613.

[7] Ruocheng Guo, Elham Shaabani, Abhinav Bhatnagar, and Paulo Shakarian. 2016. Toward early and order-of-magnitude cascade prediction in social networks. *Soc. Netw. Anal. Min.* 6, 1 (2016), 64:1–64:18.

[8] Jinbin Huang, Xin Huang, and Jianliang Xu. 2022. Truss-Based Structural Diversity Search in Large Graphs. *IEEE Trans. Knowl. Data Eng.* 34, 8 (2022), 4037–4051.

[9] Jinbin Huang, Xin Huang, Yuanyuan Zhu, and Jianliang Xu. 2019. Parameter-Free Structural Diversity Search. In *WISE (Lecture Notes in Computer Science)*, Vol. 11881. 677–693.

[10] Jinbin Huang, Xin Huang, Yuanyuan Zhu, and Jianliang Xu. 2021. Parallel algorithms for parameter-free structural diversity search on graphs. *WWW* 24, 1 (2021), 397–417.

[11] Xin Huang, Hong Cheng, Rong-Hua Li, Lu Qin, and Jeffrey Xu Yu. 2013. Top-K Structural Diversity Search in Large Networks. *Proc. VLDB Endow.* 6, 13 (2013), 1618–1629.

[12] Xin Huang, Hong Cheng, Rong-Hua Li, Lu Qin, and Jeffrey Xu Yu. 2015. Top-K structural diversity search in large networks. *VLDB J.* 24, 3 (2015), 319–343.

[13] Xinyi (Lisa) Huang, Mitul Tiwari, and Sam Shah. 2013. Structural Diversity in Social Recommender Systems. In *Proceedings of the Fifth ACM RecSys Workshop on Recommender Systems and the Social Web (CEUR Workshop Proceedings)*, Vol. 1066.

[14] Alon Itai and Michael Rodeh. 1978. Finding a Minimum Circuit in a Graph. *SIAM J. Comput.* 7, 4 (1978), 413–423.

[15] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407, 1-3 (2008), 458–473.

[16] Andrew McGregor. 2014. Graph stream algorithms: a survey. *SIGMOD Rec.* 43, 1 (2014), 9–20.

[17] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *SIGMOD*. 1415–1430.

[18] Javier Sanz-Cruzado and Pablo Castells. 2018. Enhancing structural diversity in social networks by recommending weak ties. In *RecSys*. 233–241.

[19] Yotam Shmargad. 2018. Structural diversity and tie strength in the purchase of a social networking app. *J. Assoc. Inf. Sci. Technol.* 69, 5 (2018), 660–674.

[20] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281.

[21] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon M. Kleinberg. 2012. Structural diversity in social contagion. *Proc. Natl. Acad. Sci. USA* 109, 16 (2012), 5962–5966.

[22] Xubo Wang, Dong Wen, Wenjie Zhang, Ying Zhang, and Lu Qin. 2023. Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs. In *ICDE*. IEEE, 2538–2550.

[23] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient Structural Graph Clustering: An Index-Based Approach. *Proc. VLDB Endow.* 11, 3 (2017), 243–255.

[24] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2019. I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering. *IEEE Trans. Knowl. Data Eng.* 31, 1 (2019), 75–90.

[25] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. 2022. Span-reachability querying in large temporal graphs. *VLDB J.* 31, 4 (2022), 629–647.

[26] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 170:1–170:27.

[27] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully Dynamic Depth-First Search in Directed Graphs. *Proc. VLDB Endow.* 13, 2 (2019), 142–154.

[28] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2020. AOT: Pushing the Efficiency Boundary of Main-Memory Triangle Listing. In *DASFAA (Lecture Notes in Computer Science)*, Vol. 12113. 516–533.

[29] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. DPTL+: Efficient Parallel Triangle Listing on Batch-Dynamic Graphs. In *ICDE*. 1332–1343.

[30] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical K-Cores. *Proc. VLDB Endow.* 14, 11 (2021), 2033–2045.

[31] Aihua Zhang, Mingxing Zheng, and Bowen Pang. 2018. Structural diversity effect on hashtag adoption in Twitter. *Physica A: Statistical Mechanics and its Applications* 493 (2018), 267–275.

[32] Qi Zhang, Rong-Hua Li, Qixuan Yang, Guoren Wang, and Lu Qin. 2020. Efficient Top-k Edge Structural Diversity Search. In *ICDE*. 205–216.