Maintaining Biconnected Components in Streaming Graphs

ZHAO LU, University of New South Wales, Australia
DONG WEN*, University of New South Wales, Australia
WENTAO LI, University of Leicester, UK
XUEMIN LIN, Shanghai Jiao Tong University, China
WENJIE ZHANG, University of New South Wales, Australia

Biconnected components (BCCs) are fundamental structures in graph analysis, with applications spanning various domains. To support these applications over continuously evolving data, we study the problem of maintaining BCCs in streaming graphs under the widely used sliding-window model. Existing methods suffer from a severe bottleneck in edge deletion, which dominates their practical running time. To overcome this limitation, we design an index that eliminates the need to maintain BCCs under edge deletions. This index compresses all BCCs across sub-windows ending at the current timestamp and achieves optimal space complexity. When an edge expires and is deleted, the portion of the index corresponding to the sub-window containing this edge can be directly discarded in constant time. For edge insertions, we develop a two-step maintenance framework that progressively transforms the outdated index into the updated version through a sequence of tree-edge rotations. Extensive experiments on real-world datasets demonstrate that our approach considerably outperforms state-of-the-art methods.

CCS Concepts: • **Information systems** → **Data structures**.

Additional Key Words and Phrases: Biconnected Component, Sliding Window, Streaming Graph

ACM Reference Format:
Zhao Lu, Dong Wen, Wentao Li, Xuemin Lin, and Wenjie Zhang. 2026. Maintaining Biconnected Components in Streaming Graphs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 207 (June 2026), 25 pages. https://doi.org/10.1145/3802084

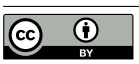
1 Introduction

Computing biconnected components (BCCs) is a fundamental problem in graph analysis [12, 18, 28, 66, 69]. An undirected graph is *biconnected* if it remains connected after the removal of any vertex. For example, the graph in Figure 1 is biconnected, since removing any vertex would not disconnect it. A *biconnected component* is a maximal biconnected subgraph. Biconnected components have found applications across a wide range of domains, such as robustness analysis [45, 47], centrality computation [35, 48], graph visualization [9], urban infrastructure design [42], and glacier simulations [2].

In this paper, we study the maintenance of biconnected components (BCCs) in streaming graphs under the sliding-window model. We assume that edges arrive in non-decreasing timestamp order, following prior work on streaming graphs [5, 70, 71]. The sliding-window model is widely

*Dong Wen is the corresponding author.

Authors' Contact Information: Zhao Lu, zhao.lu@unsw.edu.au, University of New South Wales, Sydney, Australia; Dong Wen, dong.wen@unsw.edu.au, University of New South Wales, Sydney, Australia; Wentao Li, livent@126.com, University of Leicester, Leicester, UK; Xuemin Lin, xuemin.lin@gmail.com, Shanghai Jiao Tong University, Shanghai, China; Wenjie Zhang, wenjie.zhang@unsw.edu.au, University of New South Wales, Sydney, Australia.



ACM 2836-6573/2026/6-ART207
https://doi.org/10.1145/3802084

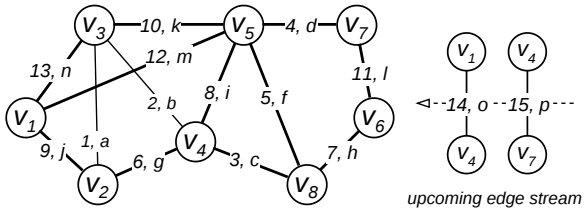


Fig. 1. A Streaming Graph. Each edge is labeled with its arrival timestamp and identifier. The thick edges form a graph within the sliding window $[3, 13]$, while the thin edges have expired.

adopted in streaming data analysis, as only a small portion of the most recent data is of interest in many scenarios [21, 46]. This fixed-size window is a standard model used in streaming graph problems [5, 10, 17, 70, 71]. When an edge arrives and the window reaches its capacity, the oldest edge expires, changing the graph and triggering BCC maintenance.

Applications. We present three scenarios where maintaining BCCs in streaming graphs under the sliding-window model is beneficial.

- *Power Grid Resilience.* Modern power grids continuously stream signals from substations and transmission lines, enabling online analysis through central control systems [49]. Maintaining BCCs within a predefined window provides a safeguard against islanding—a hazardous condition in which part of the grid becomes isolated from the main network. Rather than detecting such events after they occur using existing techniques [6, 40], an islanding alert can be issued to the control center when a critical region becomes non-biconnected.
- *Critical Connection Monitoring.* Mission-critical infrastructure networks, such as telecommunication backbones and financial transaction systems, naturally form streaming graphs where links appear and disappear as messages, packets, or heartbeats are exchanged. In these networks, at least one redundant communication path is required by specification [36]. Maintaining BCCs over the most recent interactions enables continuous validation of redundancy requirements, as any two vertices within a BCC are connected by two vertex-disjoint paths.
- *Real-Time Network Analysis.* Various network analysis tasks use BCCs as a fundamental building block, including centrality computation [35] and planarity testing [33]. For example, the dynamic betweenness centrality algorithm in [53] relies on vertices that belong to multiple BCCs, and our techniques can facilitate the extension of this algorithm to streaming graphs.

Existing Studies. Among all methods that support dynamic BCC maintenance in a sliding window [13, 26, 29, 50, 51, 58], the line of HDT algorithms [28, 29, 58] represents the state of the art [23]. The HDT algorithm amortizes the cost of edge deletion maintenance to improve overall efficiency. Edge deletion is the primary bottleneck in BCC maintenance, as the algorithm must scan the graph to determine whether the BCCs remain biconnected. The HDT algorithm organizes the graph into $\lceil \log n \rceil$ hierarchical layers to avoid re-scanning. When a level-i edge is deleted, the algorithm scans the graph and promotes the BCCs that remain biconnected to layer $i + 1$ if they contain at most $\lceil n/2^{i+1} \rceil$ vertices. These promoted BCCs would remain biconnected after any edge deletion at layer i. Thus, the HDT algorithm avoids rescanning higher-layer BCCs in future deletions. The HDT algorithm also addresses the intransitivity of biconnectivity across vertices, historically a main challenge noted in [25, 28]. The HDT algorithm maintains partitions of biconnected neighbors for each vertex at every level. When an edge arrives, the affected partitions are merged. When an edge is removed, the affected partitions must be split and then restored via a two-pass scanning process (see Section 3.2 for details), which forms the main bottleneck. All subsequent theoretical follow-ups [29, 58] build directly upon this framework [28], inheriting the same bottleneck. There

also exist works that maintain BCCs in different settings or compute BCCs in static graphs; see Section 3.3 for details.

Our Framework. Although the cost of the HDT algorithms is amortized over deletions, deleting an edge may still require scanning the whole graph in the worst case. Our approach eliminates this bottleneck by building an index that represents BCCs across all sub-windows ending at the current latest timestamp. In this way, when an edge has expired, there exists only one sub-window containing the expired edge, and we can directly discard this sub-window instead of maintaining the BCCs within it. Given that expanding a sub-window never increases the number of BCCs, we compress the BCCs of such sub-windows at all start times using a tree-structured index. Then, the index supports BCC queries in linear time and edge deletion in constant time. Our index can be extended to query the BCCs containing specific query vertices.

The main challenge of index maintenance lies in edge insertion, which may affect multiple sub-windows. To address this, we first precisely analyze the update of BCCs. Based on the analysis, we add a new tree node and connect some relevant leaf nodes to a carefully-picked existing node in the tree. This step transforms the tree into a graph (which could contain cycles after inserting the node) that is equivalent to the expected tree index for the updated window. We then formulate two structural transformation operators that reduce cycle lengths in the graph while guaranteeing this equivalence property. By continuously applying these operators, we adjust the graph structure, eliminate the cycles, remove the unnecessary nodes, and derive the up-to-date tree structure.

Contributions. This paper makes the following key contributions:

- *A space-optimal index.* We design a novel index structure whose size and query time are both bounded by the total number of vertices across all BCCs within the window.
- *Efficient maintenance algorithms.* We develop an $O(1)$ time edge deletion algorithm and an $O(h \cdot h_t)$ time edge insertion algorithm for index maintenance when the window slides, where h is the height of our tree index and h_t is the height of the maximum spanning tree. They are both loosely bounded by the number of vertices, but are practically small.
- *Extensive experimental evaluation.* We evaluate our approach on 17 graphs, demonstrating significant improvements in update speed, query efficiency, and memory usage over the baselines.

2 Preliminary

An *edge stream* is an infinite sequence of undirected edges, denoted by $\mathcal{E}_s = \langle e_1, e_2, \dots \rangle$, where edges continuously arrive in non-decreasing timestamp order. Without loss of generality, we assume that one edge arrives at each timestamp $t \in \mathbb{N}$. If an edge e arrives at time t, we denote its arrival time as $\tau(e) = t$. Let $t_{\max}$ denote the current timestamp.

A *sliding window* consists of the edges that arrived within the most recent θ timestamps, denoted by $\mathcal{W} = \{e \in \mathcal{E}_s \mid t_{\max} - \theta < \tau(e) \leq t_{\max}\}$. We use $G(V, E)$ to denote the graph in the current window, where $E = \mathcal{W}$. We use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges in G, respectively. A graph $H(V', E')$ is a subgraph of $G(V, E)$ if $V' \subseteq V$ and $E' \subseteq E$, denoted by $H \subseteq G$. We say $H \subset G$ if $H \subseteq G$ and $H \neq G$. Given a subgraph H, we denote its vertex set and edge set by $V(H)$ and $E(H)$, respectively.

A graph G is *connected* if there exists a path between every pair of vertices in G. A *connected component* is a maximal connected subgraph of G, and a connected component query returns the vertex sets of all components [63]. An edge e is a *bridge* if removing it disconnects G (i.e., increases the number of connected components). Similarly, a vertex v is an *articulation point* if its removal disconnects G. A graph G is *biconnected* if it contains no articulation points [28], i.e., removing any single vertex would not disconnect G.

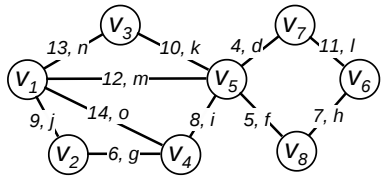
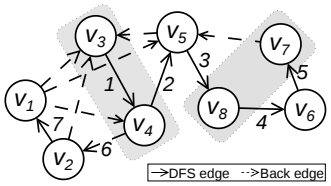


Fig. 2. An Updated Window. After the edge o arrives at time 14, the oldest edge (v_4, v_8) expires.



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
first	7	6	0	1	2	4	5	3
low	0	0	0	0	2	2	2	2

Fig. 3. Illustration of baseline methods.

Definition 2.1 (Biconnected Component). *A subgraph $H \subseteq G$ is a biconnected component if H is a maximal biconnected subgraph, i.e., there exists no biconnected subgraph $H' \subseteq G$ such that $H \subset H'$.*

Example 2.1. *Figure 1 shows a streaming graph with continuously arriving edges and its corresponding graph within the sliding window $[3, 13]$. When the next edge (v_1, v_4) arrives, the window slides to $[4, 14]$ and the graph G in the updated window is shown in Figure 2. The graph in Figure 1 consists of a single BCC, whereas the updated G contains two BCCs: one induced by $\{v_1, \ldots, v_5\}$ and another by $\{v_5, \ldots, v_8\}$.*

Problem Statement. We study the problem of maintaining the biconnected components (BCCs) within a sliding window of size θ. We aim to design a compact index that represents these BCCs, supports BCC queries, and can be maintained efficiently as the window slides.

3 Baseline Solutions

3.1 Online Computation

We can employ the classic *Hopcroft–Tarjan algorithm* (HT algorithm) [32] to compute and update the BCCs as the window slides.

The HT algorithm performs a depth-first search (DFS) and maintains $first(v)$ and $low(v)$ values for each vertex v. The value $first(v)$ is the discovery time of v in the preorder traversal of the DFS tree. An edge (u, v) is called a *back edge* if $first(u) > first(v)$. The value $low(v)$ represents the smallest discovery time reachable from v by traversing some non-back edges and following at most one back edge. If a vertex u is the root of the DFS tree, it is an articulation point if it has more than one child. Otherwise, u is an articulation point if there exists a child v of u in the DFS tree such that $first(u) \leq low(v)$. We can obtain the BCCs by partitioning the graph at articulation points. The algorithm runs in $O(m + n)$ time and space.

Example 3.1. *Figure 3 illustrates the execution of the HT algorithm on the graph in Figure 2. In Figure 3, solid arrows indicate tree edges labeled by traversal order, and dashed arrows show back edges. The DFS starts at v_3. The table on the right lists the $first$ value and low value for every vertex. Vertex v_8 has $low(v_8) = 2$ since it can reach v_5 via two tree edges and a back edge, and $first(v_5) = 2$. Since $first(v_5) \leq low(v_8)$ and v_8 is a child of v_5, vertex v_5 is an articulation point that partitions the graph into two BCCs.*

3.2 The HDT Algorithm

The HDT algorithm [28, 29, 58] is the state-of-the-art for maintaining biconnectivity under edge insertions and deletions [23], which amortizes the cost of update operations.

The HDT algorithm maintains its spanning forest F as the framework. To identify BCCs, it is necessary to traverse non-tree edges. To avoid repeated traversal of non-tree edges, it assigns a level value (with maximum value $\lceil \log_2 n \rceil$) to each non-tree edge. When an edge e of level i is deleted, we only traverse the non-tree edges of the same level. Let e' be a non-tree edge of level i encountered during the maintenance. At level-i, a BCC B containing e' is promoted to level $i+1$, if B contains no more than $\lceil \frac{n}{2^{i+1}} \rceil$ vertices. To promote B, every non-tree edge in B updates its level value to $i+1$. The future deletion maintenance of a non-tree edge at level i does not require re-examining non-tree edges in higher levels. Thus, the deletion maintenance cost is amortized across deletions.

Besides maintaining a spanning forest, the HDT algorithm also maintains neighbor partitions to address vertex intransitivity; that is, given three vertices v, u, and w such that u is biconnected to both v and w, v and w are not necessarily biconnected. To handle this, the HDT algorithm partitions the neighbors of each vertex u, grouping its biconnected neighbors into the same partition. Since the HDT maintains BCCs across multiple levels, these partitions are also maintained for every vertex at each level.

Edge insertion falls into two cases. Let (v, w) be the edge being inserted. If v and w are not connected, (v, w) would become a bridge, and the algorithm adds it to F. Otherwise, the algorithm *covers* the path p_{vw} connecting v and w in F: for each edge pair (u_1, u_2) and (u_2, u_3) on the cycle formed by p_{vw} and (v, w), it merges the neighbor partitions of u_2 containing u_1 and u_3 on level 0.

Example 3.2. *After inserting all edges in Figure 2, let the solid-line tree (without edge direction) in Figure 3 represent the spanning forest maintained by the HDT algorithm. All non-tree edges are initially at level* 0*, and the level-*0 *biconnected neighbor partitions of* v_5 *are* $\{v_3, v_4\}$ *and* $\{v_7, v_8\}$*, as shown by the shaded rectangles. Let* (v_1, v_6) *be the edge being inserted. This edge becomes a level-*0 *non-tree edge, and the two neighbor partitions of* v_5 *are merged into the same partition.*

Edge deletion falls into three cases. If the deleted edge is a bridge, it can be directly removed. To delete a non-bridge tree edge, the algorithm first swaps it with a non-tree edge and then deletes it as a non-tree edge. To delete a non-tree edge (v, w), the algorithm undoes the merges of neighbor partitions covered by (v, w). It first splits the neighbor partitions of each vertex u on the path p_{vw} connecting v and w in F at levels 0 through i. Since these split partitions may still be covered by other non-tree edges, the algorithm performs two scans along p_{vw} to recover them. The first scan starts from $u = v$ and moves toward w. It searches for a non-tree edge e that belongs to a level-i BCC B containing the edge (u, u'), where u' is a neighbor of u on the path p_{vw}. If $|V(B)| \leq \lceil n/2^{i+1} \rceil$, e is inserted into level $i+1$ using the insertion procedure; otherwise, e is reinserted into level i, and the forward pass terminates. The second scan traverses p_{vw} from w back to v, restoring neighbor partitions in the same manner. With top-tree structures [1, 31], the HDT algorithm achieves an amortized update complexity of $O(\log^5 n)$ while using $O(m + n \log^2 n)$ space [28].

Example 3.3. *Continuing Example 3.2. Suppose the edge* (v_1, v_6) *is deleted. The HDT algorithm first undoes the merges previously performed by* (v_1, v_6)*. This operation splits the neighbor partitions of* v_5 *at level* 0 *into four parts:* $\{v_3\}$*,* $\{v_4\}$*,* $\{v_7\}$*, and* $\{v_8\}$*. It then uses non-tree edges* (v_1, v_3)*,* (v_5, v_7) *and* (v_5, v_3) *to recover the neighbor partitions along the path* $p_{v_1 v_6}$*. Only* (v_5, v_7) *is promoted to level* 1*, as the BCC containing the other non-tree edges exceeds the size limit. Next, if* (v_3, v_5) *is deleted, the non-tree edge* (v_5, v_7) *is skipped during recovery because it resides at a higher level.*

3.3 Other Related Work

BCC Computation in Static Graphs. The classical DFS-based algorithms compute all BCCs in linear time and space [32, 56]. Since DFS is inherently sequential [52], recent research focuses on parallel approaches that replace DFS with either a breadth-first search (BFS) tree or an arbitrary spanning tree (AST). Cheriyan and Thurimella [8] proved that a BFS tree can serve as a sparse certificate for biconnectivity, inspiring a series of parallel algorithms based on this idea [4, 11, 19, 54, 60]. Tsin and Chin [59] and Tarjan and Vishkin [57] proposed two early AST-based parallel algorithms that extended the DFS-based method via auxiliary graphs. GBBS [11] implemented [57]. More recently, Dong et al. [12] presented the FAST-BCC algorithm, which eliminates the need for auxiliary graphs by directly identifying the fence edges that separate BCCs on an AST. Both GBBS [11] and FAST-BCC [12] have $O(m)$ total work.

BCC Maintenance in Dynamic Graphs. Henzinger [25, 50] developed the first sublinear dynamic biconnectivity algorithm that supports edge insertions, deletions, and biconnectivity queries by augmenting Frederickson's data structure [20]. Holm et al. [27, 28] improved upon this line of work by partitioning non-tree edges into $\lceil \log_2 n \rceil$ layers and maintaining a neighborhood partition for each vertex at each layer based on biconnectivity, extending [26] into the first deterministic polylogarithmic dynamic biconnectivity algorithm. Building on the HDT framework, Thorup claimed an improved amortized update bound of $O((\log n)^4 \log\log n)$ [58][1].

For insertion-only scenarios, Westbrook and Tarjan [62] designed a union-find–like structure that supports nearly constant amortized-time insertions. The deletion-only case is more challenging: Holm and Rotenberg recently proposed a decremental algorithm [30], while La Poutré and Westbrook [41] studied deletion maintenance performed in the reverse order of insertions. Since the theoretical follow-ups [29, 58] of the original HDT algorithm inherit the same deletion bottleneck in practice, we regard the HDT algorithm as our state-of-the-art baseline. Notably, the HDT algorithm is also the only BCC maintenance algorithm mentioned in the recent survey [23].

Maintenance of Other Structures in Streaming Graphs. Zhang et al. [70] proposed an SOTA approach for maintaining connected components (CCs) in sliding windows using a bidirectional disjoint-set structure. The maintenance of CCs in streaming graphs under more general settings has been studied in recent work [55, 64] and implemented in graph data systems [7, 17]. Efficient maintenance algorithms for edge-connected components [10, 43], strongly connected components [14, 37], depth-first search trees [15, 67], density-based reachability [38, 39], triangle counts [22, 44], minimum spanning trees [65], and structure diversity [5] have also been studied in streaming graph settings. Some recent work supports user-defined query windows in streaming graphs, maintaining connected components [55, 63] and k-cores [61, 68]. However, these approaches maintain graph structures with the vertex-transitive property. Thus, they cannot be directly extended to maintain biconnected components, since biconnectivity is not vertex-transitive.

4 Our Approaches

The main bottleneck of BCC maintenance lies in edge deletion. As shown in Example 3.3, it may require scanning the entire graph to reconstruct the neighbor partitions.

Our main idea is to maintain a data structure that eliminates the need to maintain BCCs under edge deletions. This structure captures all BCCs of every sub-window that ends at the current timestamp $t_{\max}$. The BCCs of the sub-window $[t, t_{\max}]$ for each $t_{\max} - \theta < t \leq t_{\max}$ can be retrieved directly from this structure, where θ is the window size. In this way, when the window slides and $t_{\max}$ increases by 1, we discard the part in our data structure that represents the BCCs in the

[1]The $O((\log n)^3 \log\log n)$ complexity in [58] has been corrected in [28].

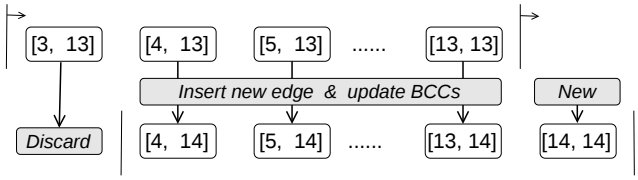


Fig. 4. Illustration of the native approach in Example 4.1.

expired window $[t_{\max} - \theta, t_{\max} - 1]$, and maintain the BCCs in each remaining sub-window under the insertion of the edge that arrives at $t_{\max}$. No BCC maintenance under edge deletions is needed.

A naive approach is to store the BCCs of each sub-window $[t, t_{\max}]$ with $t_{\max} - \theta < t \leq t_{\max}$, where $t_{\max}$ is the current timestamp. When a new edge arrives and the window slides, we update this straightforward index as follows. We first discard the BCCs of the sub-window $[t_{\max} - \theta, t_{\max} - 1]$ that contains the expired edge, and then update the BCCs in the remaining sub-windows $[t, t_{\max} - 1]$ with $t_{\max} - \theta < t < t_{\max}$ under the edge insertion. Any path connecting the endpoints of the inserted edge must pass through all articulation points. We can merge the BCCs that contain at least one articulation point and one edge on this path. Finally, we initialize the new window $[t_{\max}, t_{\max}]$ by directly adding the new edge.

Example 4.1. *Figure 4 shows our naive approach. Suppose the current window is* $[3, 13]$ *in Figure 1. We store the BCCs in each sub-window ending at* $t_{\max}$*, i.e.,* $[3, 13], [4, 13], \ldots, [13, 13]$*. Suppose a new edge* (v_2, v_8) *arrives at time* 14*. We discard the BCCs in* $[3, 13]$*, insert the edge into the empty sub-window* $[14, 14]$ *and the sub-windows* $[4, 13], \ldots, [13, 13]$*, and update their BCCs under the edge insertion. For example,* $[4, 13]$ *contains two BCCs induced by* $\{v_1, \ldots, v_5\}$ *and* $\{v_5, \ldots, v_8\}$*, whereas the updated sub-window* $[4, 14]$ *contains only one BCC formed by merging the previous two.*

4.1 Index Foundations

This naive approach treats each sub-window ending at the current timestamp $t_{\max}$ independently, leading to redundant storage and update computations. Given the overlap between BCCs of different sub-windows, we explore the relationships between BCCs across sub-windows ending at $t_{\max}$ and design a compact representation.

Monotonicity of BCCs. BCCs exhibit *monotonicity* across sub-windows ending at $t_{\max}$. Specifically, when adding edges in decreasing order of arrival times, BCCs are never split and may only merge into larger ones. The following lemma formalizes this property.

Lemma 4.1. *Let* B_1 *and* B_2 *be two BCCs in sub-windows* $[t_1, t_{\max}]$ *and* $[t_2, t_{\max}]$*, respectively, with* $t_1 \geq t_2$*. Either* B_1 *is a subgraph of* B_2*, or they share no edges; that is,* $B_1 \subseteq B_2$ *or* $E(B_1) \cap E(B_2) = \emptyset$*.*

Example 4.2. *In Figure 1, there exist three BCCs* B_4 *induced by* $\{v_5, \ldots, v_8\}$*,* B_6 *induced by* $\{v_1, \ldots, v_5\}$*, and* B_{10} *induced by* $\{v_1, v_3, v_5\}$ *in the sub-windows* $[4, 13]$*,* $[6, 13]$*, and* $[10, 13]$*, respectively. Since* $E(B_{10})$ *intersects* $E(B_6)$*, we have* $B_{10} \subseteq B_6$*. Since* $B_{10} \nsubseteq B_4$*,* $E(B_{10})$ *does not intersect* $E(B_4)$*.*

We define the coverage relation among BCCs across all sub-windows ending at $t_{\max}$ to capture the monotonicity of BCCs. Based on this relation, we then discuss the lossless compression of BCCs.

Definition 4.1 (BCC Coverage). *Let* B_1 *and* B_2 *be two BCCs in sub-windows* $[t_1, t_{\max}]$ *and* $[t_2, t_{\max}]$ *with* $t_1 > t_2$*, respectively, we say that* B_2 *covers* B_1*, denoted* $B_1 \sqsubset B_2$*, if* $B_1 \subset B_2$ *and there exists no* $B_3 \subset B_2$ *such that* B_3 *is a BCC that exists in a sub-window* $[t_3, t_{\max}]$ *with* $t_1 > t_3 > t_2$ *and* $B_1 \subset B_3 \subset B_2$*.*

Example 4.3. *Continuing Example 4.2, all edges in the current window* $[3, 13]$ *form a BCC* B_3*, with* $B_{10} \sqsubset B_6 \sqsubset B_3$*. Let* B_8 *be the BCC in sub-window* $[8, 13]$ *induced by* $\{v_1, v_2\}$*. Then* $B_8 \not\sqsubset B_3$ *since there exists a BCC* B_6 *in* $[6, 13]$ *such that* $B_5 \subset B_6 \subset B_3$*.*

We can compress BCCs across sub-windows ending at $t_{\max}$ according to their coverage relation. Let $B_1 \sqsubset B_2$ denote two BCCs in two sub-windows $[t_1, t_{\max}]$ and $[t_2, t_{\max}]$, respectively. To represent B_2, it suffices to store the difference between B_1 and B_2 rather than the entire B_2 in the sub-window $[t_2, t_{\max}]$. Thus, we only need to store these incremental differences.

Edge-centric representation. Next, we discuss the structure to represent BCCs for one sub-window and extend the structure for multiple sub-windows based on the monotonicity. We adopt an edge-centric method to compress BCCs. By edge-centric, we mean that we view each BCC as a set of edges. Compared with the vertex-centric method used in previous approaches, our edge-centric framework simplifies the index design and has a well-bounded index size. Specifically, the vertex-centric methods [26, 27, 50, 51, 62] require auxiliary structures on vertices, such as neighbor partitions [26, 27], to maintain intransitive biconnectivity among adjacent vertices. By contrast, biconnectivity is transitive on edges: if e_1 and e_2 are biconnected, and e_2 and e_3 are biconnected, e_1 and e_3 are also biconnected. Thus, such auxiliary structures are not necessary for edge-centric methods. Straightforwardly, BCCs can be stored as disjoint edge sets in $O(m)$ space, since each edge belongs to exactly one BCC. Lemma 4.2 shows that it suffices to use only the edges of a spanning tree[2] rather than all graph edges.

Lemma 4.2. *Let* T *be an arbitrary spanning tree of* G*, and let* B *be a BCC in* G*. There exists an edge set* $S \subseteq E(T)$*, such that* $V(S) = V(B)$*.*

Proof. Let $S = E(T) \cap E(B)$. Then, $V(S) \subseteq V(B)$. Assume for contradiction that $V(B) \not\subseteq V(S)$. Let w be a vertex in $V(B) \setminus V(S)$. For any $v \in V(S)$, let $p = p_{vw} \setminus E(B)$, where p_{vw} is the path connecting v and w in T. If $|p| = 0$, $w \in V(S)$. If $|p| > 0$, $E(B) \cup p$ forms a biconnected graph larger than B, since p starts and ends at two vertices of B. □

Example 4.4. *The DFS tree in Figure 3 contains two sets of tree edges, each corresponding to one of the two BCCs in Figure 2, which are separated by an articulation point.*

We now describe how to compress the BCCs across all sub-windows ending at $t_{\max}$ in an edge-centric manner. Since the BCCs of each sub-window can be represented by its spanning tree edges, it suffices to compress the spanning trees of all such sub-windows. Instead of maintaining a separate spanning tree for each sub-window, we utilize the maximum spanning tree (MST) to compactly represent these spanning trees, based on the following lemma.

Lemma 4.3. *Let* T *be the maximum spanning tree of the graph* G *in the current window, weighted by edge timestamps. For any sub-window* $[t, t_{\max}]$ *with* $t_{\max} - \theta < t \leq t_{\max}$*, there exists an edge set* $S \subseteq E(T)$ *such that* S *forms a spanning tree in* $[t, t_{\max}]$*.*

Proof. This follows directly from Lemma 4.12 in [55]. □

Lemma 4.3 implies that the spanning tree T' of any sub-window $[t, t_{\max}]$ with $t_{\max} - \theta < t \leq t_{\max}$ can be derived from the MST T of G. Specifically, the edges in T with timestamps no smaller than t constitute T', i.e., $E(T') = \{ e \in E(T) \mid \tau(e) \geq t \}$. Therefore, it suffices to use T alone to compress the BCCs across all sub-windows ending at $t_{\max}$.

Example 4.5. *Continuing Example 4.3, the MST* T *is shown in Figure 5. The edge sets* $E(B_{10}) \cap E(T)$ *and* $E(B_6) \cap E(T)$ *cover the vertices of* B_{10} *and* B_6*, respectively.*

[2]We use the terms tree and forest interchangeably when the context is clear.

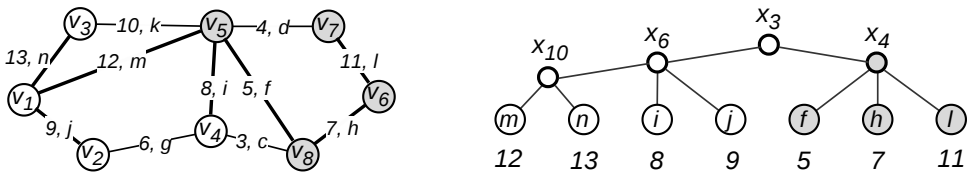


Fig. 5. The MST (thick lines) of the graph in $[3, 13]$ (left) and the BBF index (right). The shaded vertices $\{v_5, \ldots, v_8\}$ induce a BCC in $[4, 13]$, which is represented by the shaded subtree rooted at x_4 in the BBF index.

4.2 Our Index Structure

We now build a hierarchy of BCC coverage relations (Definition 4.1) using only the edges of the MST of G, defined as follows:

Definition 4.2 (Backward BCC Forest (BBF)). *Given a graph G in the current window and the MST T of G weighted by edge timestamps, the Backward BCC Forest is a forest $\mathcal{T} = (\mathcal{V}, \mathcal{E})$:*

- *For each node x, we associate it with a timestamp, denoted by $x.\tau$, such that x represents a BCC B_x that exists in the sub-window $[x.\tau, t_{\max}]$ but not in $[x.\tau + 1, t_{\max}]$.*
- *If x is a leaf node, we associate it with an edge in T, denoted by $x.\gamma$. The leaf nodes represent all edges in T, that is, for every $e \in E(T)$, there exists a leaf node x with $x.\gamma = e$.*
- *A link $(x, y) \in \mathcal{E}$ exists if and only if $B_x \sqsubset B_y$, where B_x and B_y are the BCCs represented by node x and y, respectively.*

Example 4.6. *Figure 5 illustrates a Backward BCC Forest (BBF) constructed on the MST on the left, where each internal node x_i has timestamp i. A leaf j with $j.\tau = 9$ corresponds to the MST edge j, which is a bridge in $[9, 13]$. The internal node x_4 represents B_4, the BCC induced by $\{v_5, \ldots, v_8\}$, which exists in $[4, 13]$ but not in $[5, 13]$. The leaves of the subtree rooted at x_4 are $\{f, h, l\}$, whose corresponding MST edges cover $V(B_4)$. Similarly, the internal node x_6 represents B_6, which exists in $[6, 13]$ but not in $[7, 13]$. A link $(x_{10}, x_6) \in \mathcal{E}$ exists because $B_{10} \sqsubset B_6$, where B_{10} is represented by x_{10}.*

Lemma 4.4. *The Backward BCC Forest (BBF) requires $O(n)$ space, where n is the number of vertices in the current sliding window.*

We now formalize the *equivalence* between the connected components in the BBF and the BCCs in the graph G. Observe that in Figure 5, the shaded connected component on the right corresponds exactly to the shaded BCC over $[4, 13]$ on the left.

Definition 4.3 (BCC Equivalence). *A vertex-weighted graph G' is BCC-equivalent to an edge-weighted graph G if, for any weight threshold t, the connected components of the subgraph of G' induced by vertices with weights $\geq t$ correspond bijectively to the biconnected components of the subgraph of G induced by edges with weights $\geq t$.*

In the BBF, each connected component induced by nodes with timestamps $\geq t$ is a subtree rooted at a node x with $x.\tau \geq t$, whose parent either does not exist or has a timestamp smaller than t. Hence, this connected component represents a BCC induced by edges with timestamps $\geq t$. For any BCC B in $[t, t_{\max}]$, there also exists a corresponding subtree in the BBF.

Lemma 4.5. *The BBF, viewed as a graph weighted by node timestamps, is BCC-equivalent to the graph within the current window, viewed as a graph weighted by edge arrival timestamps.*

Algorithm 1 presents the index-based BCC query algorithm that retrieves all BCCs within the current window. It first selects an unvisited node x (line 2) and traverses the BBF to obtain its connected component CC_x (line 3), marking all visited nodes (line 4). The vertex set of each BCC,

Algorithm 1: QueryBCCs

Input : The BBF $\mathcal{T}$ and the MST T
Output: All BCCs in the current window

1 $\mathcal{B} \leftarrow \emptyset$;
2 **while** *there exists an unvisited* $x \in \mathcal{V}$ **do**
3 $CC_x \leftarrow$ the connected component of x in $\mathcal{T}$;
4 Mark every node in CC_x as visited;
5 $B_x \leftarrow \{u \mid \exists y \in CC_x \wedge y.\gamma = (u, v)\}$;
6 $\mathcal{B} \leftarrow \mathcal{B} \cup \{B_x\}$;
7 **return** $\mathcal{B}$;

Algorithm 2: DeleteEdge

Input : The BBF $\mathcal{T}$ and MST T for $[t_{\max} - \theta, t_{\max} - 1]$
Output: The BBF and MST for $[t_{\max} - \theta + 1, t_{\max} - 1]$

1 $e \leftarrow$ the edge in the MST T with smallest timestamp;
2 **if** $\tau(e) \leq t_{\max} - \theta$ **then**
3 Remove e from T;
4 $x \leftarrow$ the node $x \in \mathcal{V}$ with smallest timestamp;
5 **if** $x.\tau \leq t_{\max} - \theta$ **then**
6 Remove x from $\mathcal{V}$;

denoted by B_x, is obtained by collecting all vertices covered by MST edges contained in CC_x (line 5) and added to the result set $\mathcal{B}$ (lines 6–7). Compared with $O(m)$ BCC recomputation methods [12, 32], our index-based query algorithm runs in $O(n)$ time, where n is the number of vertices and m is the number of edges in the current window.

Theorem 4.1. *The time complexity of Algorithm 1 is $O(n)$, where n is the number of vertices in the current sliding window.*

5 Index Maintenance

As the window slides, we preserve the BCC-equivalent property of our BBF index, which encodes all BCCs across sub-windows ending at the current $t_{\max}$. We perform two update operations: (i) *edge deletion*, which removes the edge with timestamp smaller than the new $t_{\max} - \theta + 1$, and (ii) *edge insertion*, which adds the edge arriving at $t_{\max}$.

5.1 Edge Deletion

After edge deletion, we discard the BCCs captured in the expired window $[t', t_{\max} - 1]$, where $t' = t_{\max} - \theta$ and $t_{\max}$ is the updated timestamp. We ensure that (1) no connected component in the updated BBF contains a node with timestamp $\leq t'$, and (2) all other connected components induced by nodes with timestamps $> t'$ remain unchanged. We achieve this by removing the node x with $x.\tau = t'$. If such a node x exists, it must be a root in the BBF. Hence, the removal of x guarantees both (1) and (2). We also need to update the MST on which the BBF was built. We can directly remove the edge with timestamp $\leq t'$, since the MST induced by the edges of timestamp $> t'$ is an MST in $[t' + 1, t_{\max}]$ (Lemma 4.3).

Algorithm 2 presents the edge deletion update procedure. With our index structure, it is unnecessary to store the edges within the current window. Hence, Algorithm 2 first examines the oldest tree edge e from the MST T (line 1). If e has expired, it removes e from the MST T (lines 2–3). Next,

it retrieves the minimum-timestamp node x from the BBF (line 4). If x has expired, it removes x from the node set $\mathcal{V}$ (lines 5–6). For the links incident to x, we adopt lazy deletion. These links are not explicitly removed in Algorithm 2, as they can be ignored during queries and are finally eliminated when the children of the expired node x expire.

Example 5.1. *Figure 5 illustrates the BBF of the graph in Figure 1 within the window* $[3, 13]$*, which is constructed upon the MST* T *shown on the left. The minimum timestamp in* T *is* 5*, indicating that no MST update is required. We then delete the node* x_3 *with* $x_3.\tau = 3$ *from the BBF, splitting the structure in Figure 5 into two trees.*

Theorem 5.1. *Algorithm 2 runs in* $O(1)$ *time.*

5.2 Edge Insertion

We preserve the BCC equivalence (Definition 4.3) of the BBF index after inserting a new edge into the graph. A straightforward approach is to check whether the BCCs change in each sub-window $[t, t_{\max}]$ and adjust the BBF accordingly. This approach is inefficient, as it repeatedly checks biconnectivity and updates the structure.

We now present an example that motivates our algorithm framework. Let (v_1, v_4) arrive at timestamp 14 in Figure 5. A new BCC B formed by $\{v_1, v_3, v_4, v_5\}$ emerges in $[8, 14]$. Thus, to encode B, we need to *update* the BCC equivalence. We form a subtree by linking some leaf nodes to a new internal node. This structure contains cycles, and we need to *transform* it into a BBF.

We design a two-step insertion framework. First, we update the BBF into a structure that is BCC-equivalent to the graph in the updated window. Specifically, we add a new tree node and connect it to certain leaf nodes to capture a critical cycle introduced by the new edge. Then, we iteratively refine this structure until it becomes a BBF. In this step, we remove the cycles added in the first step and eliminate nodes that are replaced by the newly created node.

Equivalence Update. Starting from the BBF for $[t_{\max} - \theta + 1, t_{\max} - 1]$ (after edge deletion update), we incorporate the edge e' arriving at $t_{\max}$ to obtain a structure that is BCC-equivalent to the graph in the current window ($[t_{\max} - \theta + 1, t_{\max}]$). We first show that inserting e' introduces at most one new internal node x in the index (Lemma 5.1), then identify the BCC B represented by x (Lemma 5.2), and finally describe how to add the necessary links to x based on B (Lemma 5.4).

Lemma 5.1. *Let* e' *be the edge arriving at* $t_{\max}$*,* $\mathcal{T}$ *be the BBF for the window* $[t_{\max} - \theta + 1, t_{\max} - 1]$*, and* $\mathcal{T}'$ *be the updated BBF for the current window* $[t_{\max} - \theta + 1, t_{\max}]$*. The updated BBF* $\mathcal{T}'$ *contains at most one internal node whose timestamp differs from the timestamps of the internal nodes in* $\mathcal{T}$*.*

Proof. After inserting e', assume for contradiction that there exist two non-bridge BCCs B and B', where B exists in $[t, t_{\max}]$ but not in $[t+1, t_{\max}]$, and B' exists in $[t', t_{\max}]$ but not in $[t'+1, t_{\max}]$, with $t_{\max} > t' > t \geq t_{\max} - \theta$, and no internal node x in $\mathcal{T}$ such that $x.\tau = t$ or $x.\tau = t'$. Let $e_{\min}$ and $e'_{\min}$ be the edges with timestamps t and t', respectively, and let T be the MST of the graph in the current window. Since $e_{\min}$ is a non-tree edge in T, it forms a cycle C together with the path in T connecting its endpoints, and C must contain e'. Similarly, there exists another cycle C' that also contains both e' and $e'_{\min}$. Hence, without e', $e_{\min}$ and $e'_{\min}$ remain biconnected in the sub-window $[t, t_{\max}]$ but not in $[t+1, t_{\max}]$, implying the contradiction that an internal node with timestamp t exists in $\mathcal{T}$. Thus, there exists at most one non-bridge BCC that exists in $[t, t_{\max}]$ but not in $[t+1, t_{\max}]$, such that no internal node in $\mathcal{T}$ has timestamp t. By definition, there would be at most one new internal node. □

We identify the timestamp of this new internal node by Lemma 5.2. We then show in Lemma 5.3 that every updated BCC must intersect a critical cycle. This result establishes the basis for Lemma 5.4,

where we identify the neighbors of the new internal node. Given a cycle C, we use $\tau(C)$ to denote the minimum timestamp among the edges in C, i.e., $\tau(C) = \min\{\tau(e) \mid e \in C\}$.

Lemma 5.2. *Let e' be the edge arriving at $t_{\max}$, T be the MST of the graph in $[t_{\max} - \theta + 1, t_{\max} - 1]$, and C be the cycle formed by e' and the path in T that connects the endpoints of e'. A new internal node x exists with $x.\tau = \tau(C)$ in the updated BBF.*

Proof. Let $t = \tau(C)$. There is no internal node in the BBF for $[t_{\max}-\theta+1, t_{\max}-1]$ with timestamp t, since t is the timestamp of a leaf node. Because the cycle C does not exist in $[t + 1, t_{\max}]$, an internal node x with $x.\tau = t$ in the updated BBF must exist. □

Lemma 5.3. *Let e' be the edge arriving at $t_{\max}$, and C be the cycle formed by e' and the path in the MST T in $[t_{\max} - \theta + 1, t_{\max} - 1]$. A BCC B in $[t, t_{\max} - 1]$ with $t \le \tau(C)$ becomes larger in $[t, t_{\max}]$, i.e., the corresponding BCC B' in $[t, t_{\max}]$ satisfies $|V(B)'| > |V(B)|$, only if B intersects C at some edge other than e', i.e., $E(B) \cap (C \setminus \{e'\}) \neq \emptyset$.*

Proof. Assume for contradiction that $E(B) \cap (C \setminus \{e'\}) = \emptyset$. Let e_1 and e_2 be an edge in B and an edge in the expanded part of B' (i.e., $e_2 \in E(B') \setminus E(B)$), respectively. In T, there exists a path p connecting e_1 and e_2. Some articulation points must exist on this path p in $[t, t_{\max} - 1]$; otherwise, e_1 and e_2 would already be biconnected. In B', there exists another path p' connecting e_1 and e_2 such that p and p' are vertex-disjoint and $e' \in p'$ in $[t, t_{\max}]$. Then, the cycle C must contain an articulation point in p; otherwise, a cycle $C'' \subseteq \{e_1, e_2\} \cup p \cup p' \setminus \{e'\}$ would exist and make $e_2 \in E(B)$. However, if C intersects p at one of these articulation points, B is not maximal in $[t, t_{\max} - 1]$, since there exists a larger biconnected component in $C \cup p \cup p' \cup B$. □

Lemma 5.4. *Let e' be the edge arriving at $t_{\max}$. Let T and $\mathcal{T}$ be the MST and BBF for $[t_{\max} - \theta + 1, t_{\max} - 1]$, respectively. Let C be the cycle formed by e' and the path connecting its endpoints in T, and let x be the node with timestamp $\tau(C)$ in $\mathcal{T}$. The following operations result in a BCC-equivalent graph $\mathcal{G}$ for $[t_{\max} - \theta + 1, t_{\max}]$:*

- *add a new leaf node with $\tau = t_{\max}$ and $\gamma = e'$;*
- *remove the link (x, p) if x has a parent p;*
- *add a link (y, x) for every node y such that $y \neq x$ and $y.\gamma \in C$.*

Proof. Let z be a node in $\mathcal{T}$ and CC be the connected component induced by z and all nodes with timestamps no less than $z.\tau$ in $\mathcal{G}$. If $z.\tau > \tau(C)$ or the BCC B_z represented by z in $[z.\tau, t_{\max} - 1]$ satisfies $B_z \cap C = \emptyset$, then inserting e' does not affect B_z, by Lemma 5.3. And CC remains unchanged and still represents B_z. Hence, we only consider z with $z.\tau \le \tau(C)$ and $B_z \cap C \neq \emptyset$. Let B'_z denote the updated BCC of z in $[z.\tau, t_{\max}]$. Clearly, $B_z \subset B'_z$. Let $e \in E(B'_z) \setminus E(B_z)$. Then, e belongs to a BCC $B \not\subset B_z$ in $[z.\tau, t_{\max} - 1]$. By Lemma 5.3, $E(B) \cap C \neq \emptyset$. Let y be the leaf node representing an edge in $E(B) \cap C$. By linking (y, x), CC now covers all vertices in B. □

Based on Lemma 5.4, Algorithm 3 updates the BBF to restore BCC-equivalence. It first creates a new leaf node x for the edge (v, w) arriving at timestamp $t_{\max}$ (line 1). If v and w are in different connected components of the MST T for $[t_{\max} - \theta + 1, t_{\max} - 1]$, (v, w) would be a bridge, and we complete the BCC-equivalence update. Otherwise, it converts the leaf node z that represents the edge e'' with the minimum timestamp in the cycle C (lines 3-6) into an internal node (lines 7-12), where C is formed by the path p_{vw} connecting v and w in T together with (v, w). It then links the leaves representing the edges in C except e'' to z (lines 10-12).

Example 5.2. *Suppose an edge $e'(v_1, v_4)$ arrives at timestamp 14 in Figure 5 and we maintain the BBF for $[3, 13]$. By Lemma 5.1, a new internal node x appears. As shown in Figure 5, the cycle C introduced by e' in the MST is $\{e', m, i\}$. By Lemma 5.2, $x.\tau = \tau(C) = 8$. Figure 6 illustrates the update*

Algorithm 3: UpdateEquivalence

Input : A new edge $e'(v, w)$ arriving at $t_{\max}$, and the BBF $\mathcal{T}$ and MST T for $[t_{\max} - \theta + 1, t_{\max} - 1]$

Output : A BCC-equivalent graph for $[t_{\max} - \theta + 1, t_{\max}]$

```
1  Add a new tree node x with x.γ = e' and x.τ = t_max;
2  if v and w are connected in T then
3      p_vw ← the path connecting v and w in T;
4      C ← p_vw ∪ {e'} ;
5      e'' ← the edge with minimum timestamp in C;
6      z ← the leaf node with z.γ = e'';
7      p ← the parent of z;
8      Cut(z, p);
9      z.γ ← ∅ ;                                   // z becomes an internal node
10     foreach e ∈ C \ {e''} do
11         y ← the leaf node with y.γ = e;
12         Link(y, z);
```

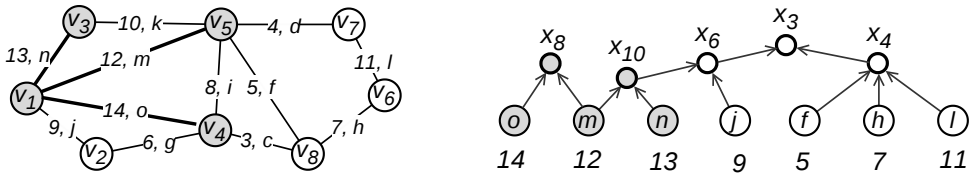


Fig. 6. The updated graph G in $[3, 14]$ (left) and its BCC-equivalent graph $\mathcal{G}$ (right). Links in $\mathcal{G}$ are directed from the node with a larger timestamp to the smaller one.

process of Algorithm 3, which turns the leaf node i (with timestamp 8) in Figure 5 into the internal node x_8, and connects the leaves representing m and o to x_8, where $o = e'$.

We now introduce *edge directions* to $\mathcal{G}$ to distinguish the neighbors of a node. In Figure 6, instead of verbosely saying that x_6 is a neighbor of x_{10} with a smaller timestamp, we say that x_6 is an *out-neighbor* of x_{10}. The *out-neighbors* of a node x, denoted by $\mathcal{N}^-(x)$, are the set of its neighbors with smaller timestamps, and the *in-neighbors* $\mathcal{N}^+(x)$ are its neighbors with larger timestamps.

Equivalent Transformation. We now proceed to the second step, which transforms the equivalent graph $\mathcal{G}$ into a valid BBF, where $\mathcal{G}$ is the output of Algorithm 3. We first establish the goal of this transformation—a sufficient condition for $\mathcal{G}$ to qualify as a BBF (Lemma 5.6).

Compared with the equivalent graph in Figure 6, the transformed BBF in Figure 7 is clearly an *in-tree* [24], where any node can traverse to the root by following the edge directions. To transform an equivalent graph into an in-tree, we introduce Lemma 5.5.

Lemma 5.5. *A directed graph is an in-tree if and only if exactly one node has out-degree zero and all others have out-degree one* [24].

After the BCC equivalence update, a node may have more than one out-neighbor. Thus, we first refine the graph into an in-forest by reducing the out-degrees of some nodes, and then refine the in-forest into a compact BBF by removing unnecessary nodes. To achieve this, Lemma 5.6 summarizes the structural properties of a BBF.

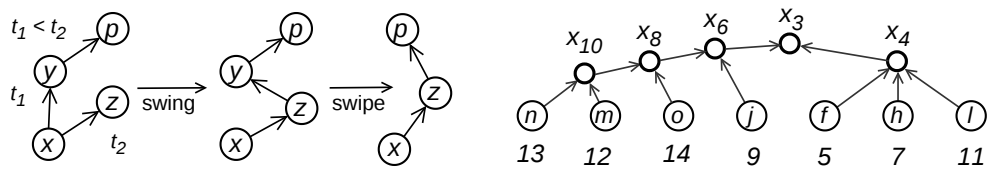


Fig. 7. The Swing and Swipe operators (left) transform the BCC-equivalent graph $\mathcal{G}$ into a BBF index (right).

Lemma 5.6. *A BCC-equivalent graph $\mathcal{G}$ is a BBF if every node x satisfies: (1) x has at most one out-neighbor and (2) has either no in-neighbor or more than one in-neighbor.*

Proof. By Lemma 5.5 and condition (1), the $\mathcal{G}$ forms an in-forest. By BCC equivalence, every node x represents a BCC in $[x.\tau, t_{\max}]$ by the subtree rooted at it. It remains to prove that every link (x, y) represents that $B_x \sqsubset B_y$ (Definition 4.1), where x is an in-neighbor of y and B_x and B_y are the BCCs represented by x and y, respectively. We have $B_x \neq B_y$ and $B_x \subset B_y$, as y has at least two in-neighbors. Assume for contradiction that $B_x \not\sqsubset B_y$. Then, there exists a BCC B in $[t, t_{\max}]$ with $t > x.\tau$ such that $B_x \subset B \subset B_y$. Thus x would have another out-neighbor, contradicting (1). □

We now present two operators that refine the equivalent graph to satisfy these conditions while preserving the BCC equivalence property. The Swing operator reduces the out-degree of a node with more than one out-neighbor to satisfy condition (1). The Swipe operator removes a node with only one in-neighbor to satisfy condition (2).

Definition 5.1 (Swing Operator). *Given a BCC-equivalent graph $\mathcal{G}$, a node x, and two of its out-neighbors y and z such that $z.\tau > y.\tau$, the Swing operator replaces the link (x, y) with (z, y).*

Example 5.3. *Figure 7 illustrates a subgraph of a BCC-equivalent graph on the left, where $x.\tau > z.\tau > y.\tau$. Since x has two out-neighbors, the Swing operator removes (x, y) and adds (z, y). The connected components induced by x and the nodes with timestamps $\geq x.\tau$ remain unchanged. Similarly, the connected components containing y and z, respectively, remain unchanged.*

Lemma 5.7. *After applying a Swing operator to a node x in a BCC-equivalent graph, the resulting graph remains BCC-equivalent.*

The Swing operator decreases the out-degree of a node x. However, it can increase the out-degree of an out-neighbor of x. Thus, it may need to be applied recursively to the out-neighbor with increased out-degree. The Swing operator also removes an (undirected) cycle if there already exists an edge between the two out-neighbors of x. When this happens, the in-degree of the out-neighbor with the smaller timestamp decreases, which may trigger the Swipe operator.

Definition 5.2 (Swipe Operator). *Given a BCC-equivalent graph $\mathcal{G}$ and a node x with exactly one out-neighbor y and one in-neighbor z, the Swipe operator removes the node x and adds a link (z, y). As a special case, if x has no out-neighbor, this operator only removes x.*

Example 5.4. *In Figure 7, suppose that y has only one in-neighbor z after applying the Swing operator. The Swipe operator removes y and links z to p, where p is the only out-neighbor of y.*

Lemma 5.8. *After applying a Swipe operator to a node x in a BCC-equivalent graph, the resulting graph remains BCC-equivalent.*

The Swipe operation requires that the node x has at most one out-neighbor. Lemma 5.9 shows that during the structure refinement, any internal node with one in-neighbor has at most one out-neighbor. Thus, the Swipe operator can be applied to every internal node x with in-degree

Algorithm 4: InsertEdge

Input : A new edge $e'(v, w)$ arriving at $t_{\max}$, and the BBF $\mathcal{T}$ and MST T for $[t_{\max} - \theta + 1, t_{\max} - 1]$
Output : The BBF and the MST for $[t_{\max} - \theta + 1, t_{\max}]$

1 **if** *exists an edge* $e \in E(T)$ *such that* $e = e'$ **then**
2 $x \leftarrow$ the leaf node with $x.\gamma = e$;
3 $x.\tau \leftarrow t_{\max}$;
4 **return**;
5 `UpdateEquivalence(`e'`,`$\mathcal{T}$`,` T`)` ; `// Algorithm 3`
6 $p_{vw} \leftarrow$ path connecting v and w in T;
7 $X \leftarrow \{y \in \mathcal{V} | y.\gamma \in p_{vw}\}$;
8 **while** $X \neq \emptyset$ **do**
9 $x \leftarrow$ an element in X;
10 **if** $|\mathcal{N}^-(x)| \leq 1$ **then**
11 $X \leftarrow X \setminus \{x\}$;
12 **continue**;
13 $y, z \leftarrow$ two nodes in $\mathcal{N}^-(x)$ with $y.\tau < z.\tau$;
14 `Swing(`x`)`;
15 **if** $|\mathcal{N}^+(y)| \leq 1$ **then** `Swipe(`y`)`;
16 **if** $|\mathcal{N}^-(z)| > 1$ **then** $X \leftarrow X \cup \{z\}$;
17 Insert e' into T;
18 **if** p_{vw} *exists* **then**
19 $e'' \leftarrow$ the edge with minimum timestamp in p_{vw};
20 Delete e'' from T ;

1. By repeatedly applying these two equivalence-preserving operators, the equivalent graph is eventually transformed into a BBF structure that satisfies Lemma 5.6.

Lemma 5.9. *Given a BCC-equivalent graph after a series of Swing operators, every internal node y with at most one in-neighbor has at most one out-neighbor.*

Proof. Assume that an internal node y has more than one out-neighbor and it has only one in-neighbor x. Then y must have appeared as the larger-timestamp out-neighbor in some Swing operator. Suppose this Swing operator is applied to a node $x' \neq x$. Then x' remains an in-neighbor of y, which contradicts the assumption. If $x' = x$ and this Swing operator is the only operator where y appears as a larger-timestamp out-neighbor, y would have only one in-neighbor in the BBF of the last window (i.e., $[t_{\max} - \theta, t_{\max} - 1]$), contradicting the definition of BBF. □

The Algorithm. We now present the edge insertion update procedure given in Algorithm 4. If the new edge e' already exists in T, it just updates the timestamp of the leaf node representing e' and terminates (lines 1-4). Otherwise, it updates the BCC equivalence using Algorithm 3 (line 5). It then iteratively applies the Swing and Swipe operators (lines 14-15) to refine the equivalent graph into a BBF (lines 7–16). The set X contains the nodes with more than one out-neighbor (line 7). Whenever a node in X no longer violates the condition, it is removed from X (lines 10-12). If an out-neighbor z of a node in X increases its out-degree beyond one, z is inserted into X (line 16). Each Swing operation decreases the number of out-neighbors of a given node and shrinks existing cycles. Hence, X will eventually become empty, and Algorithm 4 successfully updates the BBF.

Finally, it updates the MST by swapping the edge e'' with the smallest timestamp in the cycle introduced by e' with the new edge e' (lines 17–20).

Example 5.5. *Continuing Example 5.2. Figure 7 shows the process of Algorithm 4. After constructing the equivalent graph in Figure 6, we have $X = \{o, m\}$. We then apply the Swing operator on m: remove (m, x_8) and add (x_{10}, x_8). Since x_{10} now has two out-neighbors, we apply Swipe on x_{10}: remove (x_{10}, x_6) and add (x_8, x_6).*

Equivalence update (Algorithm 3) takes $O(|p_{uv}|)$ time where p_{uv} is the path connecting the endpoints of the new edge in the MST T. Each Swing operator triggers an additional Swing unless the node z receiving an out-neighbor y had no out-neighbor or the link (z, y) already exists. Therefore, each node in X triggers at most $O(h)$ Swing and Swipe operations, where h is the height of the BBF and $|X| = O(|p_{uv}|)$.

Theorem 5.2. *Algorithm 4 runs in $O(h \cdot h_t)$ time, where h is the height of the BBF index and h_t is the height of its MST.*

The Optimization. We now develop two techniques to improve the efficiency of the operators. Let y and z be the two out-neighbors of x with $y.\tau < z.\tau$. First, instead of using y and z as the operands of the Swing operator on x, we use y and p, where p is the highest ancestor of z (following the edge directions) whose timestamp is still larger than $y.\tau$. This is equivalent to applying the Swing operator to y and all nodes from z to p. Second, if two nodes y and z have already been processed as the operands of a previous Swing operator (with $y.\tau < z.\tau$), then y is already an ancestor of z, and we can remove the link (x, y).

Algorithm 5 presents the advanced Swing-and-Swipe (SS) process. We update z with $y.\tau < z.\tau < x.\tau$ to its highest ancestor p with a timestamp larger than y (lines 3–11), and then we process x, y and z as in a normal SS operation (lines 12–15). If an ancestor of z and y has already been the two out-neighbors in a previous SS operation, we cut (x, y), remove y if necessary, and finish (lines 4–8); otherwise, we mark this ancestor and y as processed (lines 9–11), and continue the iteration to find the highest ancestor. To detect previously processed pairs and maintain the compactness of the BBF index, we only add two attributes to each node p: the last timestamp when p participated in a Swing operation, and the other out-neighbor operand y involved in that operation. If the recorded timestamp equals the current timestamp and the recorded operand equals the current operand, we identify the pair as previously processed. After the Swing operation, if z violates the out-degree constraint (lines 16–17), we return z (to Algorithm 4) for further refinement. That is, the returned z should be added to the set of nodes with out-degree greater than 1 for further processing (line 16 in Algorithm 4). Otherwise, we return $\emptyset$, indicating that the SS operation on x terminates and no further refinement is required (lines 8 and 18).

Example 5.6. *Continuing Example 5.5, let (v_4, v_7) be the next edge arriving at timestamp 15. Figure 8 shows the process of optimized refinement. In the MST of the graph over $[3, 14]$, the path $p_{v_4v_7}$ is $\{o, m, f, h, l\}$. The updated equivalent graph is shown in the middle of Figure 8. We first apply the advanced SS operation at l: link x_5 to x_4 and cut (l, x_4). Next, we process h. Since x_5 and x_4 have already been processed, we directly remove (h, x_4). Node x_4 now has only one in-neighbor x_5, so we delete x_4 and attach x_5 to x_3. Next, we move to m. We identify x_6 as the highest ancestor of x_5 whose timestamp is greater than the timestamp of x_5. We then link x_6 to x_5, remove (m, x_5), and mark all nodes from x_{10} to x_6 as processed with x_5. For the next node o, we directly cut (o, x_5), since the pair (x_8, x_5) has already been processed at $t_{\max}$. Finally, we apply SS to x_6, and x_3 is then swiped.*

As shown in Example 5.6, the advanced SS operation accelerates the process by initiating the Swing operation at the highest ancestor and skipping already processed out-neighbor pairs. This

Algorithm 5: ADV-SS

Input : A node x with $|\mathcal{N}^-(x)| > 1$
Output: The ancestor z of x with increased $|\mathcal{N}^-(z)|$ or $\emptyset$

$y, z \leftarrow$ two nodes in $\mathcal{N}^-(x)$ with $y.\tau < z.\tau$;
$p \leftarrow z$;
while $p \neq \emptyset$ ***and*** $p.\tau > y.\tau$ **do**
  if (p, y) *already processed* **then**
    Cut(x, y);
    if $|\mathcal{N}^+(y)| \leq 1$ **then**
      `Swipe(`y`)`;
    return $\emptyset$;
  Mark (p, y) as processed;
  $z \leftarrow p$;
  $p \leftarrow$ the node in $\mathcal{N}^-(p)$; // $|\mathcal{N}^-(p)| \leq 1$
Link(z, y);
Cut(x, y);
if $|\mathcal{N}^+(y)| \leq 1$ **then**
  `Swipe(`y`)`;
if $|\mathcal{N}^-(z)| > 1$ **then**
  return z;
return $\emptyset$

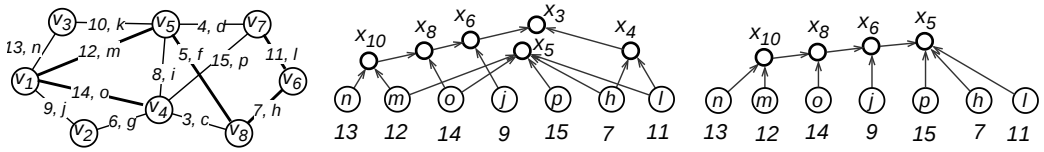


Fig. 8. The updated graph in $[3, 15]$ (left), the BCC-equivalent graph (middle), and the refined BBF index (right). With processing order l, h, m, o, (h, x_4) and (o, x_5) are removed directly, avoiding redundant traversals.

optimization avoids repeated rewiring and significantly shortens the transformation process in practice, as shown in our experimental results.

6 Performance Studies

We evaluate our approaches against four baseline algorithms. All algorithms are implemented in C++ and compiled with g++ using the same optimization flag `-O3`. All experiments are conducted on a Linux server with an Intel Xeon 2.80 GHz CPU and 512 GB RAM.

Datasets. We use fifteen real-world temporal graph datasets, where each edge is associated with a timestamp, from the KONECT[3] and SNAP[4] collections, as listed in Table 1. Among them, `DW` and `SO` are the largest unipartite temporal graph datasets, while the remaining datasets are selected from diverse domains. For each dataset, the edges are sorted by timestamp in non-decreasing order to form an input stream for our algorithm. Self-loops are omitted.

[3]http://konect.cc/networks/
[4]https://snap.stanford.edu/data/

Dataset	$\|V\|$	$\|E\|$	$\|E^*\|$	D
DN, DNC-temporal[3]	2,029	39,264	4,384	8
EL, Elec[3]	7,118	103,675	100,693	7
SD, Slashdot[3]	51,083	140,778	116,573	17
MO, Mathoverflow[3]	24,818	506,550	187,986	9
EM, Email-Euall[4]	265,214	420,045	364,481	14
EP, Epinions[3]	131,828	841,372	711,210	16
LK, Lkml-reply[3]	63,399	1,096,440	159,996	63
SU, Superuser[3]	194,085	1,443,339	714,570	12
ER, Enron[3]	87,273	1,148,072	297,456	14
SW, Dynamic-simplewiki[3]	100,312	1,627,472	824,968	12
YT, Youtube-growth[3]	3,223,589	9,375,374	9,375,374	31
TW, Higgs-twitter[3]	456,626	14,855,842	12,508,413	12
BD, Baidu-Zhishi[3]	2,141,300	17,794,839	17,014,946	20
SO, Stackoverflow[3]	2,601,977	63,497,050	28,183,518	11
DW, Dynamic-dewiki[3]	2,166,669	86,337,879	39,705,981	10
LD, Ladder (synthetic)	80,000	120,040	120,040	2,001
DS, Dense Graph (synthetic)	2,001	1,000,500	1,000,500	2

Table 1. Dataset statistics. $|V|$ and $|E|$ denote the number of vertices and edges, respectively. $|E^*|$ denotes the number of edges with distinct endpoints, and D is the diameter.

We use two synthetic datasets. We stress-test the BBF updates using an adversarial pattern that targets the factors in our insertion complexity $O(h \cdot h_t)$. We generate a ladder graph (`LD`) by sequentially stacking l 4-edge cycles with one shared edge boundary, such that both h and h_t grow linearly with l[5]. We generate 40 such ladder graphs with $l = 1{,}000$ to initialize the first window. Then, we generate a random edge stream in which consecutive edges form cycles of random length at most 100. We evaluate query performance on a dense graph `DS` with edge density 0.5, where edges are randomly shuffled in the initial window. We generate a random edge stream for `DS` following the same pattern as `LD`.

Competitors. We evaluate the following algorithms.

- `HT`. A classic DFS-based online BCC computation method [32] with SOTA performance for sequential BCC computation [12].
- `GBBS` and `FBCC`. Two recent BCC computation methods [11, 12]. We use their public implementations and run them in single-thread mode for a fair comparison.
- `HDT`. The fully dynamic BCC algorithm [28], which deletes the expired edge first and then inserts the new edge. We use the top tree in [31] for `HDT`. As discussed in § 3.3, we select `HDT` as the baseline, since later fully dynamic methods extend `HDT` mainly at the theoretical level, without experimental evaluation.
- `BBF-Basic`. Our basic BCC maintenance algorithm, where we apply Algorithm 2 to remove the expired edge, and then Algorithm 4 to insert the newly arrived edge.
- `BBF`. Our final BCC maintenance algorithm, an improvement over `BBF-Basic`[6]. It replaces the Swing and Swipe operators in Algorithm 4 (lines 14–15) with Algorithm 5.

[5]For any i such that $i \bmod 4 = 0$, edges $(i, i+1)$, $(i, i+2)$, $(i+1, i+3)$, and $(i+2, i+3)$ arrive at timestamps i, $i+1$, $i+2$, and $i+3$, respectively.

[6]The implementation is available at https://github.com/azHUwFrAbj/mbcisg

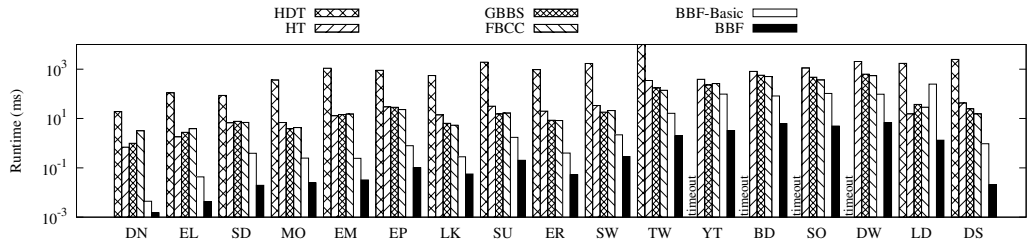


Fig. 9. Average sliding-window update time.

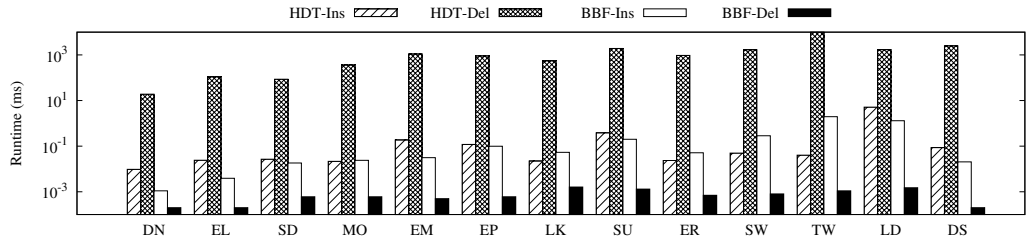


Fig. 10. Average insertion and deletion time.

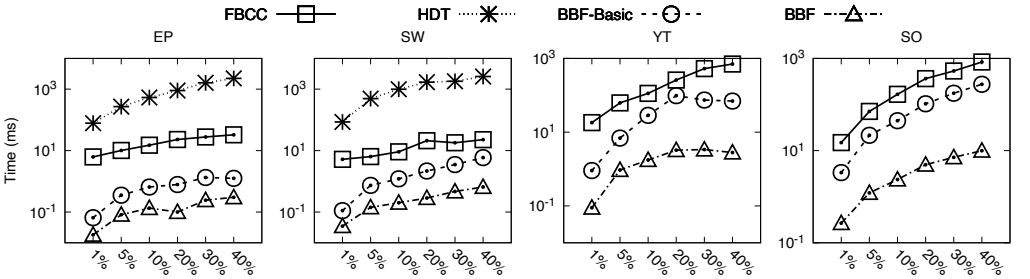


Fig. 11. Average update time under varying window sizes.

6.1 Update Efficiency

We compare the average runtime of sliding-window update operations by performing 10,000 updates. For each update, $t_{\max}$ increases by one; the edge with timestamp $t_{\max} - \theta$ is deleted, and the edge with timestamp $t_{\max}$ is inserted. These updates are conducted after loading the first θ edges into the window.

Average Case. We report the average update time under our default window size setting, where $\theta = 0.2 \times |E|$ for real-world datasets and $\theta = |E|$ for synthetic datasets. For some datasets, HDT is omitted because it failed to complete the experiment within 48 hours.

In Figure 9, our algorithm BBF consistently outperform the baselines. In particular, in the real-world dataset, BBF achieves up to 600× speedup over the fastest baseline on the EL dataset, and reduces update time by two orders of magnitude on many datasets. The adversarial synthetic input LD degrades the performance of BBF-Basic, making it slower than the static baselines. In contrast, BBF maintains its advantage over the best baseline under the adversarial input, demonstrating the effectiveness of the pruning strategy of BBF, which prevents BBF from degenerating into its worst-case behavior. The performance of HDT is worse than that of the online computation baseline. This observation is consistent with prior findings [3, 6, 34, 65], which show that HDT-based algorithms often fail to achieve their expected amortized bounds in practice. The HDT-based BCC maintenance is asymptotically the most expensive among the variants in [28]. Moreover, the constant factor of

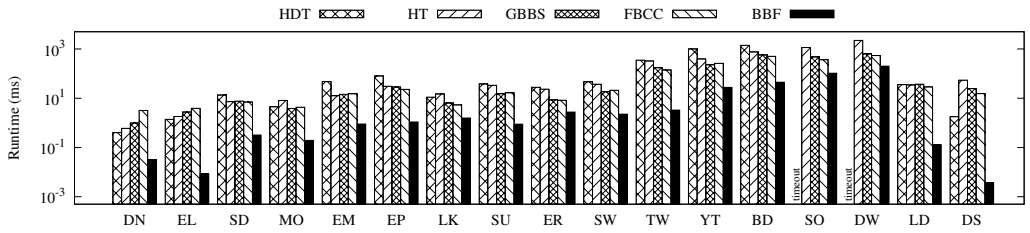


Fig. 12. Average query time for retrieving all BCCs.

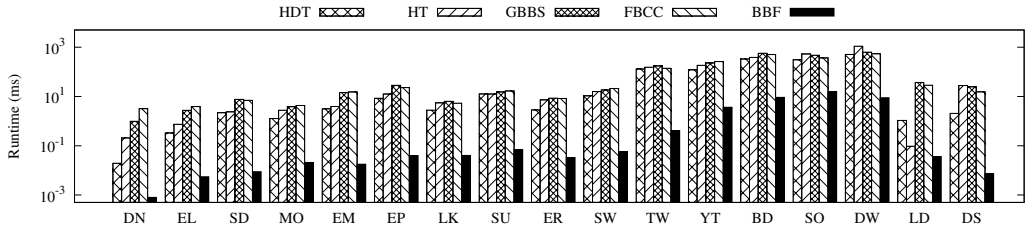


Fig. 13. Average time for searching the BCC(s) containing a given vertex.

HDT-based BCC maintenance is larger than that of other variants, since it relies on more complex data structures.

Figure 10 further shows that our method improves the overall efficiency by effectively eliminating the deletion bottleneck. Deletion dominates the runtime of HDT (labeled as HDT-Del), often exceeding insertion costs by several orders of magnitude. In contrast, deletion operations (BBF-Del) in our BBF algorithm are faster than insertions (BBF-Ins) and therefore no longer a bottleneck.

Varying Window Size. We evaluate the update performance under different window sizes (1%, 5%, 10%, 20%, 30%, and 40% of $|E|$) in Figure 11. We select FBCC as the representative static method, since its overall performance is the best among static baselines in Figure 9. As the window size increases, the update time of all algorithms generally increases. Nevertheless, BBF consistently achieves the lowest update time across all window sizes, as it eliminates the need to rescan the entire graph.

6.2 Query Processing

We now compare the runtime of querying all BCCs in a sliding window. The BCC query returns all biconnected components in the current window. After loading the first θ edges, we perform one query after each sliding-window update, and report the average time over 1,000 such queries. Note that the HDT paper [28] does not describe a BCC query algorithm. We implement an HDT-based query procedure that traverses a spanning tree via unvisited biconnected edges and collects the vertices of each BCC, where biconnectivity is determined by a native HDT procedure.

Average Case. Figure 12 reports the average BCC query time under our default window size, where $\theta = 0.2 \times |E|$ for real-world datasets and $\theta = |E|$ for synthetic datasets. Our index-based method (BBF) achieves the best performance across all datasets, as it only needs to traverse the $O(n)$-space forest structure to enumerate all BCCs. Our index-based method runs faster than HDT in practice, as our index directly encodes the existing BCCs instead of determining biconnectivity using auxiliary data structures. The static baselines must revisit non-tree edges for each query, resulting in $O(m)$ cost. Note that our $O(n)$ query algorithm is optimal in terms of answer size. As a result, the performance improvement over static baselines can be explained by the ratio m/n. The

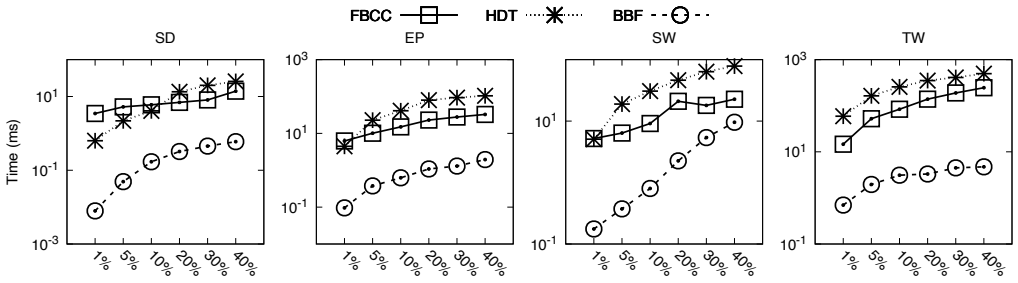
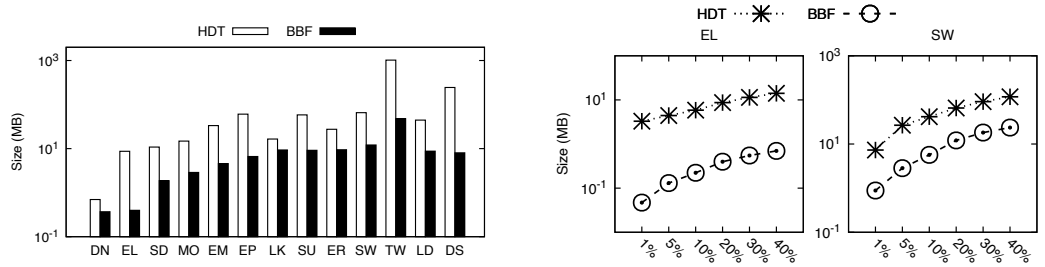


Fig. 14. Query time under varying window sizes.



(a) Index size under the default window size.

(b) Index size under varying window sizes.

Fig. 15. Index size under different window-size settings.

real-world datasets with relatively larger m/n ratios exhibit larger performance gains, such as EL and TW. On the synthetic dense graph DS, the advantage of our method becomes more pronounced.

Extension. Our index can support the vertex-based BCC search that retrieves the BCC(s) containing a given vertex [16]. With BBF, we first obtain the roots, which represent the result BCCs, of the leaf nodes corresponding to MST edges adjacent to the query vertex, and then retrieve the corresponding BCCs. We also extend HT and HDT into localized variants by traversing only from the query vertex. Figure 13 reports the average query time of 1,000 random vertex-based queries under the default window size. In this setting, the performance gap between our index-based search algorithm and the global recomputation widens, since our BBF is linear in the size of the retrieved BCC(s), whereas the static baselines may still need to scan all edges. For the dense graph DS, the query performance shows a mild decrease, since DS typically form a single large BCC, and the vertex-based search additionally traverses from the leaf to the root.

Varying Window Size. Figure 14 shows the average query time under four representative datasets. We select FBCC to present the static methods. As the window size increases, the cost for all algorithms increases, as more edges (and vertices) are retained within the window. Our BBF consistently outperforms the baselines.

6.3 Index Size

Following [6], we measure the memory footprint by summing the byte size of data structure elements, i.e., the index size. The static baselines are omitted since they need no index.

Figure 15a reports the index size after performing 10,000 updates on a fully loaded window of size $\theta = 0.2 \times |E|$ on the real-world datasets and $\theta = |E|$ on the synthetic datasets. Datasets that failed to complete all updates are omitted. Overall, our BBF index consistently consumes less space. This is because BBF requires only linear space in the number of vertices, whereas HDT has a higher space cost of $O(m + n \log^2 n)$.

Figure 15b reports the index size under increasing window sizes. We report two representative datasets (EL and SW), as similar trends are observed across other datasets. Across all window settings, BBF remains more space-efficient than HDT.

7 Conclusion

In this paper, we propose an index-based approach for the maintenance of biconnected components in streaming graphs under the sliding-window model. Our space-optimal index reformulates the maintenance process as an insertion-only procedure and supports constant-time deletion maintenance. For edge insertions, we develop a two-step update algorithm that first constructs an intermediate non-tree structure and then transforms it into an updated index through BCC-equivalence-preserving structural adjustments. Experiments demonstrate that our approach improves update efficiency, query efficiency, and memory footprint compared with the baselines.

Acknowledgments

Dong Wen is supported by ARC DP230101445 and ARC DE240100668. Wentao Li is supported by NSFC 62302417. Xuemin Lin is supported by NSFC U2241211, NSFC U20B2046, and Guangdong Basic and Applied Basic Research Foundation 2019B1515120048. Wenjie Zhang is supported by ARC DP230101445 and ARC FT210100303.

References

[1] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1997. Minimizing Diameters of Dynamic Trees. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1256)*, Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela (Eds.). Springer, 270–280. doi:10.1007/3-540-63165-8_184

[2] Ian Bogle, Karen Devine, Mauro Perego, Sivasankaran Rajamanickam, and George M Slota. 2019. A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[3] Giuseppe Cattaneo, Pompeo Faruolo, U Ferraro Petrillo, and Giuseppe F Italiano. 2010. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Applied Mathematics* 158, 5 (2010), 404–425.

[4] Meher Chaitanya and Kishore Kothapalli. 2016. Efficient multicore algorithms for identifying biconnected components. *International Journal of Networking and Computing* 6, 1 (2016), 87–106.

[5] Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. 2024. Querying structural diversity in streaming graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1034–1046.

[6] Qing Chen, Michael H Böhlen, and Sven Helmer. 2025. An Experimental Comparison of Tree-data Structures for Connectivity Queries on Fully-dynamic Undirected Graphs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[7] Zheng Chen, Feng Zhang, Yang Chen, Xiaokun Fang, Guanyu Feng, Xiaowei Zhu, Wenguang Chen, and Xiaoyong Du. 2024. Enabling Window-Based Monotonic Graph Analytics with Reusable Transitional Results for Pattern-Consistent Queries. *Proc. VLDB Endow.* 17, 11 (2024), 3003–3016. doi:10.14778/3681954.3681979

[8] Joseph Cheriyan and Ramakrishna Thurimella. 1991. Algorithms for parallel k-vertex connectivity and sparse certificates. In *Proceedings of the twenty-third annual ACM Symposium on Theory of Computing*. 391–401.

[9] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. 2013. The Open Graph Drawing Framework (OGDF). In *Handbook on Graph Drawing and Visualization*, Roberto Tamassia (Ed.). Chapman and Hall/CRC, 543–569.

[10] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. 2013. Dynamic graphs in the sliding-window model. In *Algorithms–ESA 2013: 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings 21*. Springer, 337–348.

[11] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[12] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably fast and space-efficient parallel biconnectivity. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 52–65.

[13] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM* 44, 5 (1997), 669–696. doi:10.1145/265910.265914

[14] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[15] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing graph algorithms. In *Proceedings of the 2021 International Conference on Management of Data*. 459–471.

[16] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392. doi:10.1007/S00778-019-00556-X

[17] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 513–527. doi:10.1145/3448016.3457263

[18] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Computing connected components with linear communication cost in pregel-like systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 85–96.

[19] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Long Yuan. 2018. Distributed computing connected components with linear communication cost. *Distributed and Parallel Databases* 36 (2018), 555–592.

[20] Greg N Frederickson. 1991. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proceedings of the 32nd annual symposium on Foundations of computer science*. 632–641.

[21] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (2003), 5–14. doi:10.1145/776985.776986

[22] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 645–657. doi:10.1145/3448016.3452800

[23] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2022. Recent advances in fully dynamic graph algorithms–a quick reference guide. *ACM Journal of Experimental Algorithmics* 27 (2022), 1–45.

[24] Frank Harary. 1991. *Graph theory*. Addison-Wesley.

[25] Monika Rauch Henzinger. 1995. Fully dynamic biconnectivity in graphs. *Algorithmica* 13 (1995), 503–538.

[26] Monika Rauch Henzinger and Valerie King. 1995. Fully dynamic biconnectivity and transitive closure. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 664–672.

[27] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1998. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 79–89.

[28] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)* 48, 4 (2001), 723–760.

[29] Jacob Holm, Wojciech Nadara, Eva Rotenberg, and Marek Sokołowski. 2025. Fully dynamic biconnectivity in $\tilde{O}(\log^2 n)$ time. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*. 156–165.

[30] Jacob Holm and Eva Rotenberg. 2024. Good r-divisions imply optimal amortized decremental biconnectivity. *Theory of Computing Systems* 68, 4 (2024), 1014–1048.

[31] Jacob Holm, Eva Rotenberg, and Alice Ryhl. 2023. Splay top trees. In *Symposium on Simplicity in Algorithms (SOSA)*. SIAM, 305–331.

[32] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.

[33] John E. Hopcroft and Robert Endre Tarjan. 1974. Efficient Planarity Testing. *J. ACM* 21, 4 (1974), 549–568. doi:10.1145/321850.321852

[34] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. 2001. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *Journal of Experimental Algorithmics (JEA)* 6 (2001), 4–es.

[35] Fuad T. Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2018. Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs. *IEEE Trans. Parallel Distributed Syst.* 29, 3 (2018), 659–672. doi:10.1109/TPDS.2017.2763951

[36] Matthew Johnston, Hyang-Won Lee, and Eytan H. Modiano. 2015. A Robust Optimization Approach to Backup Network Design With Random Failures. *IEEE/ACM Trans. Netw.* 23, 4 (2015), 1216–1228. doi:10.1109/TNET.2014.2320829

[37] Adam Karczmarz and Marcin Smulewicz. 2023. On fully dynamic strongly connected components. In *31st Annual European Symposium on Algorithms (ESA 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 68–1.

[38] Bogyeong Kim, Kyoseung Koo, Undraa Enkhbat, and Bongki Moon. 2022. Denforest: Enabling fast deletion in incremental density-based clustering over sliding windows. In *Proceedings of the 2022 International Conference on Management of Data*. 296–309.

[39] Bogyeong Kim, Kyoseung Koo, Juhun Kim, and Bongki Moon. 2021. Disc: Density-based incremental clustering by striding over streaming data. In *2021 IEEE 37th International conference on data engineering (ICDE)*. IEEE, 828–839.

[40] Janice Klaiber and Clemens van Dinther. 2024. Deep Learning for Variable Renewable Energy: A Systematic Review. *ACM Comput. Surv.* 56, 1 (2024), 1:1–1:37. doi:10.1145/3586006
[41] Johannes A La Poutré and Jeffery Westbrook. 1994. Dynamic two-connectivity with backtracking. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. 204–212.
[42] Wejdene Mansour and Martin Werner. 2024. Calculating Upstream Relation in Spatial Networks Under Path Constraints. In *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems*. 314–324.
[43] Andrew McGregor. 2014. Graph stream algorithms: a survey. *SIGMOD Rec.* 43, 1 (2014), 9–20. doi:10.1145/2627692.2627694
[44] Lingkai Meng, Long Yuan, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2025. Triangle Counting in Hypergraph Streams: A Complete and Practical Approach. *Proc. ACM Manag. Data* 3, 6 (2025), 1–28. doi:10.1145/3769837
[45] Renato EN Moraes and Celso C Ribeiro. 2013. Power optimization in ad hoc wireless network topology control with biconnectivity requirements. *Computers & Operations Research* 40, 12 (2013), 3188–3196.
[46] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. 2006. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 635–646. doi:10.1145/1142473.1142544
[47] MEJ Newman and Gourab Ghoshal. 2008. Bicomponents and the robustness of networks to failure. *Physical review letters* 100, 13 (2008), 138701.
[48] Charudatt Pachorkar, Meher Chaitanya, Kishore Kothapalli, and Debajyoti Bera. 2016. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 301–310.
[49] Jadyn Powell, Alex McCafferty-Leroux, Waleed Hilal, and S Andrew Gadsden. 2024. Smart grids: A comprehensive survey of challenges, industry applications, and future trends. *Energy Reports* 11 (2024), 5760–5785.
[50] Monika Rauch. 1992. Fully dynamic biconnectivity in graphs. In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 50–59.
[51] Monika Rauch. 1994. Improved data structures for fully dynamic biconnectivity. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*. 686–695.
[52] John H Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229–234.
[53] Zhenzhen Shao, Na Guo, Yu Gu, Zhigang Wang, Fangfang Li, and Ge Yu. 2020. Efficient Closeness Centrality Computation for Dynamic Graphs. In *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24-27, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12113)*, Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.). Springer, 534–550. doi:10.1007/978-3-030-59416-9_32
[54] George M Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 1–10.
[55] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On querying historical connectivity in temporal graphs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.
[56] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.
[57] Robert E Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* 14, 4 (1985), 862–874.
[58] Mikkel Thorup. 2000. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 343–350.
[59] Yung H Tsin and Francis Y Chin. 1984. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Comput.* 13, 3 (1984), 580–599.
[60] Mihir Wadwekar and Kishore Kothapalli. 2017. A fast GPU algorithm for biconnected components. In *2017 Tenth International Conference on Contemporary Computing (IC3)*. IEEE, 1–6.
[61] Zhi Wang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2025. On More Efficiently and Versatilely Querying Historical k-Cores. *Proc. VLDB Endow.* 18, 5 (2025), 1335–1347. https://www.vldb.org/pvldb/vol18/p1335-zhong.pdf
[62] Jeffery Westbrook and Robert E Tarjan. 1992. Maintaining bridge-connected and biconnected components on-line. *Algorithmica* 7, 1 (1992), 433–464.
[63] 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. doi:10.1145/3589315
[64] Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, and Xuemin Lin. 2024. Constant-time Connectivity Querying in Dynamic Graphs. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–23.
[65] Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, Yang Lu, and Xuemin Lin. 2025. Minimum Spanning Tree Maintenance in Dynamic Graphs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.

[66] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.
[67] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully dynamic depth-first search in directed graphs. *Proceedings of the VLDB Endowment* 13, 2 (2019), 142–154.
[68] 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. doi:10.14778/3476249.3476260
[69] Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. 2023. Rethinking the Expressive Power of GNNs via Graph Biconnectivity. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=r9hNv76KoT3
[70] Chao Zhang, Angela Bonifati, and M Tamer Özsu. 2024. Incremental Sliding Window Connectivity over Streaming Graphs. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2473–2486.
[71] Siyuan Zhang, Zhenying He, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. MWP: Multi-Window Parallel Evaluation of Regular Path Queries on Streaming Graphs. *Proc. ACM Manag. Data* 2, 1 (2024), 5:1–5:26. doi:10.1145/3639260

Received October 2025; revised January 2026; accepted February 2026