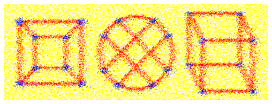

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 23, no. 2, pp. 191–226 (2019)
DOI: 10.7155/jgaa.00489

# Time Windowed Data Structures for Graphs

*Farah Chanchary*[1] *Anil Maheshwari*[1]

[1]School of Computer Science, Carleton University, Ottawa, ON, K1S 5B6, Canada

**Abstract**

We present data structures that can answer *time windowed* queries for a set of timestamped events in a relational event graph. We study the relational event graph as input to solve (a) time windowed decision problems for monotone graph properties, such as disconnectedness and bipartiteness, and (b) time windowed reporting problems such as reporting the minimum spanning tree, the minimum time interval, and the graph edit distance for obtaining spanning forests. We also present results of window queries for counting subgraphs of a given pattern, such as paths of length 2 (in general graphs) and paths of length 3 (in bipartite graphs), quadrangles and complete subgraphs of a fixed order or of all orders $\ell \geq 3$ (i.e., cliques of size $\ell$). These query results can be used to compute graph parameters that are important for social network analysis, e.g., clustering coefficients, embeddedness and neighborhood overlapping.

| Submitted: | Reviewed: | Revised: | Reviewed: | Revised: |
|---|---|---|---|---|
| December 2017 | May 2018 | July 2018 | October 2018 | October 2018 |
| | Accepted: | Final: | Published: | |
| | February 2019 | February 2019 | February 2019 | |
| | Article type: | | Communicated by: | |
| | Regular paper | | A. Lubiw | |

This research work was supported by NSERC Research Grants and Ontario Graduate Scholarship. Preliminary results for the subgraph counting data structures in RE graphs appeared in the 10th International Workshop on Algorithms and Computation (WALCOM 2016).

*E-mail addresses:* farah.chanchary@carleton.ca (Farah Chanchary) anil@scs.carleton.ca (Anil Maheshwari)

# 1 Introduction

A *relational event (RE) graph* $G = (V, E = \{e_1, e_2, \ldots, e_m\})$ is an undirected graph with a fixed set $V$ of $n$ vertices and a sequence of edges $E$ between pairs of vertices, where each edge has a unique positive timestamp [4]. Without loss of generality, we assume that $t(e_1) < t(e_2) < \ldots < t(e_m)$, where $t(e_i)$ is the timestamp of the edge $e_i$. In a *time windowed query* we are given a relational event graph $G$ and a predicate $\mathcal{P}$. We want to preprocess $G$ into a data structure such that given a query time interval $q = [i, j]$ with $1 \leq i < j \leq n$, it can answer time windowed queries based on the *graph slice* $G_{i,j} = (V, E_{i,j} = \{e_i, e_{i+1}, \ldots, e_j\})$ that matches $\mathcal{P}$.

This paper follows an *event based* approach [3], where each relational event (graph edge) appears at a specific instance in time, such that at some time $k$ there exists at most one event, i.e., the edge $e_k$. Therefore, when we consider an interval in time $[i, j]$, we get a set of events $\{e_i, \ldots, e_j\}$ whose timestamps fall in that interval. As mentioned above, given a query time interval our data structures answer queries using only the set of events that exist in this time interval.

In this paper, we present new results for time windowed decision problems on monotone properties of relational graph events, such as disconnectedness and bipartiteness, and problems on minimum spanning trees (MSTs) within a given query interval, e.g., reporting the existence of an MST, the minimum spanning interval (i.e., minimum time required to obtain an MST), and graph edit distance ($\mathcal{GED}$) to convert a graph slice into a spanning forest. We also present window data structures for counting all occurrences of subgraphs that match with a given pattern, e.g., paths of length 2 (2-paths), paths of length 3 (3-paths), quadrangles (cycles of length 4) and all complete subgraphs of a given order $\ell$, where $\ell \geq 3$. The triangle counting problem is fundamental to many graph applications. This problem has been studied in various contexts, for example as a base case for counting complete subgraphs of given orders [24], in minimum cycle detection problem [23], and as a special case of counting given length cycles [2]. Counting the total number of triangles and quadrangles are also essential for analyzing large networks (such as WWW, social networks, bipartite graphs or two-mode networks) as they are used to compute important network structures, such as clustering coefficients [32, 33] and transitivity coefficients [19]. Complete subgraphs counting problems have applications in combinatorics and network analysis (see e.g., [30, 31]). We also present some applications to show how various graph parameters can be computed using subgraph counting data structures that are particularly useful for social network analysis.

## 1.1 Previous Work

A relational event (RE) graph generally represents communication events between pairs of entities in an underlying network that occurred at some specific times. In recent studies, RE graphs have been used to model social networks

for various computational analysis and are also named *dyadic event data* [7] and *contact sequences* [22]. Bannister et al. [4] were the first to consider this model for preprocessing into data structures that can answer time windowed queries using timestamped events. They presented data structures that can count number of connected components, number of components containing cycles, number of vertices having degrees equals to some predefined value, and number of influenced vertices on a time-increasing paths [4]. They showed techniques to reduce time windowed problems into a matroid rank problem so that ranks of the query graph slices can be answered by a dominance counting query. Hence they obtained sub-logarithmic query times for time windowed problems. However, the required preprocessing time and space for all problems presented in [4] (see Table 1) are the time and space requirements for the reduction only, and do not consider those of the dominance counting data structures.

Chanchary et al. [10] presented techniques for building time window data structures for RE graphs by using a colored range searching approach [6, 16, 17]. Their results include reporting several graph parameters that are essential for social network analysis, such as graph density, $h$-index, $k$-stars, embeddedness, neighborhood overlap (for both general and bipartite graphs) and the total number of influenced vertices in the queried graph slices (see Table 1).

Recently, window queries have been extended towards solving geometric problems such as reporting skyline and proximity relations of point sets [3], finding all maximal subsequences that hold some hereditary property for a set of points [5], solving convex hull area decision problem, diameter and width decision problem [9], intersection decision problems for objects (e.g., line segments, triangles and convex $c$-gons) and problems related to dominant points and maximal layers [11].

## 1.2 New Results

The main contributions of this paper are listed below, and are also summarized in Table 1. Let $G$ be a (weighted) relational event graph consisting of $m$ edges and $n$ vertices.

1. *Decision problems on monotone graph properties*: A graph property $\mathcal{P}$ is called monotone if every subgraph of a graph with property $\mathcal{P}$ also has property $\mathcal{P}$. Given a dynamic algorithm $\mathcal{D}$ that maintains a monotone graph property $\mathcal{P}$ (e.g., disconnectedness and bipartiteness) having update time $U(n)$, query time $Q(n)$, and space $S(n)$, the time windowed decision problem can be reduced to a 2-dimensional range searching query in time $O(m \cdot (U(n) + Q(n)))$ using space $O(S(n) + m)$ that can answer queries in time $O(\log n)$.

2. *Problems on minimum spanning trees*: Given a weighted RE graph $G$ that has an MST with weight $\omega^*$:

   - We can preprocess $G$ into a data structure of size $O(m \log n)$ such that given a query time interval $[i, j]$ we can report whether there exists

Table 1: Summary of previous and new results using relational event graph $G = (V, E)$. Here $n = |V|$, $m = |E|$, $\mathcal{W} = j - i + 1$ is the width of the query window $[i, j]$, $r$ (respectively, $f$) is the number of past (respectively, future) neighbors of an edge, $h$ is the $h$-index (the largest number $h$ such that the graph contains at least $h$ vertices of degree at least $h$), $z$ is the size of the output, $s$ is the number of edges having neighboring edges, $t$ is the number of edges that are contained in some triangles, $p$ is the number of vertex pairs having some common neighbors, $k$ is the number of vertices having some neighbors in $G_{i,j}$, $\epsilon$ is a small positive constant, $X$ and $Y$ are set of output edges, $a(G) = O(\sqrt{m}))$ is the arboricity of $G$ (the minimum number of edge-disjoint spanning forests into which $G$ can be partitioned), $\gamma \leq min\{\binom{n}{2}, a(G)m\}$, $\ell$ is the order of a complete subgraph, $w$ is the number of a specific subgraph (i.e., 2-path, 3-path (in bipartite graphs), quadrangle, or complete subgraph) in the query interval, and $\mathcal{K}$ is the total number of the same subgraph in $G$. All results are in 'big-oh' notations. Results of this paper are marked with $(*)$.

| **Problem** | **Prep. time** | **Space** | **Query time** | **Ref.** |
|---|---|---|---|---|
| Number of (reciprocated) edges, degree-$k$ vertices, Reachable vertices, Connected components, tree components | $m + n$ | $m + n$ | $\log \mathcal{W} / \log \log m$ | [4] |
| | $m \log n$ | $m + n$ | $\log \mathcal{W} / \log \log m$ | [4] |
| Edges with bounded number of neighbors | $(r + f)m$ | $m + n$ | $\log \mathcal{W} / \log \log m$ | [4] |
| Triad closure | $hm$ | $m + n$ | $\log \mathcal{W} / \log \log m$ | [4] |
| Number of vertices, Graph density | $m + n$ | $m + n$ | $\log n$ | [10] |
| Degrees of vertices | $m + n$ | $m + n$ | $\log^2 n + z$ | [10] |
| $k$-stars | $m + n$ | $m + n$ | $\log n + z$ | [10] |
| $h$-index (approx.) | $n \log^2 n$ | $n \log n$ | $\log^2 n + h \log n$ | [10] |
| Embeddedness | $a(G)m$ | $a(G)m$ | $\log^2 n + t \log n$ | [10] |
| $NOver(G_{i,j})$ | $mn$ | $mn$ | $\log^2 n + (t + s) \log n$ | [10] |
| $NOver(G_{i,j})$-bipartite | $a(G)m$ | $a(G)m$ | $\log^2 n + p \log n + k$ | [10] |
| Influenced vertices | $m \log n$ | $m \log n$ | $\log n + z$ | [10] |
| Monotone properties* | | | | |
| Bipartiteness* | $m \log^3 n$ | $m + n \log n$ | $\log n$ | Th. 1 |
| Disconnectedness* | $m \log^2 n$ | $m + n \log n$ | $\log n$ | Th. 2 |
| MST* | $m \log^4 n$ | $m \log n$ | $\log n$ | Th. 6 |
| Min spanning interval* | $m \log^4 n$ | $m \log^\epsilon n$ | $\log \log n$ | Th. 7 |
| $\mathcal{GED}$ (Forest)* | $m \log n$ | $m \log n$ | $\log n + \lvert X \rvert$ | Th. 8 |
| 2-paths* | $m$ | $m$ | $n$ | Th. 9 |
| | $a(G)m$ | $m + n + \mathcal{K}$ | $\log \mathcal{W} / \log \log \mathcal{K}$ | Th. 3 |
| 3-paths* | $m + n$ | $m + n$ | $m + n$ | Th. 9 |
| | $n^4$ | $m + n + \mathcal{K}$ | $\log \mathcal{W} / \log \log \mathcal{K}$ | Th. 3 |
| Complete Subgraphs (fixed $\ell \geq 3$)* | $a(G)^{\ell-2}m$ | $m + n + \mathcal{K}$ | $\log \mathcal{W} / \log \log \mathcal{K}$ | Th. 10 |
| Complete Subgraphs (all $\ell \geq 3$)* | $a(G)^{\ell-1}m$ | $m + n + \mathcal{K} \log \mathcal{K} / \log \log \mathcal{K}$ | $\log \mathcal{K} / \log \log\ \mathcal{K})^2$ | Th. 11 |
| Quadrangles* | $a(G)m \log n$ | $a(G)m \log n$ | $\gamma \log n + w$ | Th. 12 |

an MST $T = (V, E')$ of $G$ with weight $\omega^*$ such that $E' \subseteq \{e_i, \cdots, e_j\}$ in $O(\log n)$ time. Preprocessing requires $O(m \log^4 n)$ time.

- We can preprocess $G$ into a data structure of size $O(m \log^\epsilon n)$, where $\epsilon$ is a small constant, such that given a query time interval $[i, j]$ we can report the minimum spanning interval (i.e., the smallest time interval $[\alpha, \beta]$ such that $i \leq \alpha$, $\beta \leq j$ and $\beta - \alpha$ is the smallest time required to obtain an MST of $G$) in $G_{i,j}$ in $O(\log \log n)$ time. Preprocessing requires $O(m \log^4 n)$ time.
- We can preprocess $G$ into data structures of size $O(n+m)$ such that given a query time interval $[i, j]$ we can report the graph edit distance ($\mathcal{GED}$) to convert $G_{i,j}$ into a spanning forest of $G$ in $O(\log n + |X|)$ time, where $X$ is the set of the minimum number of edges such that $G_{i,j} \setminus X$ is a forest. Preprocessing requires $O(m \log n)$ time.

3. *Problems on counting subgraphs*: We can preprocess $G$ into time windowed data structures so that given a query time interval $[i, j]$ we can count the total number of the following subgraphs in $G_{i,j}$.

   - *Paths of length 2 (2-paths)*: Preprocessing takes $O(n+m)$ time using $O(n + m)$ space, and queries can be answered in $O(n)$ time.
   - *Paths of length 3 (3-paths) (in bipartite graphs)*: Preprocessing takes $O(n + m)$ time using $O(n + m)$ space, and queries can be answered in $O(n + m)$ time.
   - *Complete subgraphs of a fixed order $\ell \geq 3$*: Preprocessing takes $O(a(G)^{\ell-2}m)$ time and $O(m + n + \mathcal{K})$ space, and queries can be answered in $O(\log \mathcal{W} / \log \log \mathcal{K})$ time, where $\mathcal{W}$ is the width of the query window and $\mathcal{K}$ is the total number of complete subgraphs of a fixed order in $G$.
   - *Complete subgraphs of all orders $\ell \geq 3$*: Preprocessing takes $O(a(G)^{\ell-1}m)$ time and $O(m+n+\mathcal{K} \log \mathcal{K} / \log \log \mathcal{K})$ space, and queries can be answered in $O((\log \mathcal{K} / \log \log \mathcal{K})^2)$ time, where $\mathcal{K}$ is the total number of complete subgraphs in $G$ of orders 3 and more.
   - *Quadrangles*: Preprocessing takes $O(a(G)m \log n)$ time and $O(a(G)m \log n)$ space. Queries can be answered in $O(\gamma \log n + w)$ time, where $\gamma \leq min\{\binom{n}{2}, a(G)m\}$ and $w$ is the number of reported quadrangles.

## 1.3 Organization

The rest of this paper is organized as follows. Section 2 provides preliminaries to the paper. Sections 3 and 4 present results of time windowed data structures for problems on monotone graph properties and on minimum spanning trees, respectively. In section 5 we present window data structures for subgraph counting problems for paths, complete subgraphs and quadrangles. In Section 6 we present some applications of window data structures. Section 7 concludes the paper.

# 2 Preliminaries

## 2.1 Relational Event Graph

A relational event (RE) graph $G$ is defined to be a simple graph with a set of $n$ vertices $V$ and a set of $m$ edges (or relational events) $E = \{e_k \mid 1 \leq k \leq m\}$ between pairs of vertices. We assume that the graph is undirected so the pairs are unordered. We also assume that each edge or relational event has a unique timestamp. We denote the timestamp of an edge $e_k \in E$ by $t(e_k)$. Without loss of generality, we assume that $t(e_1) < t(e_2) < \ldots < t(e_m)$, and that the timestamps follow the sequence $1, 2, \ldots, m$.

Given a relational event graph $G$, for a pair of integers $1 \leq i < j \leq m$, we define the *graph slice* $G_{i,j} = (V, E_{i,j} = \{e_i, e_{i+1}, \ldots, e_j\})$. $N_{i,j}(v)$ denotes the set of neighbors of vertex $v$ in $G_{i,j}$ and $deg_{i,j}(u)$ is the number of edges adjacent to vertex $u$ in $G_{i,j}$. Figure 1(a) illustrates an RE graph $G$ with 5 edges. Figures 1(b) and 1(c) represent, respectively, the graph slices $G_{2,4}$ and $G_{2,5}$.

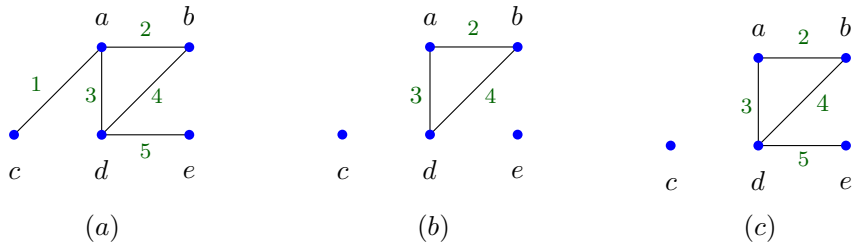



Figure 1: (a) An RE graph $G$ with five edges (the integer numbers on the edges are their timestamps), (b) a graph slice $G_{2,4}$ and (c) a graph slice $G_{2,5}$. Examples of open triplets: $\langle bd, de\rangle$, $\langle ad, de\rangle$ in $G$ and $G_{2,5}$; $\langle ca, ab\rangle$, $\langle ca, ad\rangle$ in $G$. Examples of closed triplets: $\langle ab, bd, da\rangle$ in $G$, $G_{2,4}$ and $G_{2,5}$.

A *triplet* is a connected subgraph consisting of three distinct vertices that are connected by either three edges (closed triplet) or two edges (open triplet). Note that any triangle consists of three closed triplets, one centered on each of the vertices. Figure 1(a) shows a triangle $(a, b, d)$ consisting of three open triplets with time stamps $\langle 2, 3\rangle$, $\langle 2, 4\rangle$ and $\langle 3, 4\rangle$ where each of these triplets are closed by a third edge with timestamps 4, 3 and 2, respectively.

A *high* (respectively, *low*) event in any RE graph slice $G_{i,j}$ is defined to be the timestamp of an edge $e_k$, where $e_k$ is the edge with the highest (respectively, the lowest) timestamp among all edges that form a particular subgraph in $G_{i,j}$, such as paths of a fixed length, quadrangles or complete subgraphs. We refer to Figure 2 and suppose we are interested in counting how many quadrangles ($C_4$) are in the query slice $[i, j]$. For a given slice $G_{1,12}$, edges $\{e_1, .., e_{12}\}$ do not form any $C_4$, thus no *high* or *low event* occurs in this slice. However, edges $e_9, e_{10}, e_{12}$ and $e_{13}$ create a $C_4 = (e_9, e_{10}, e_{12}, e_{13})$ in $G_{1,13}$. Therefore, edges $e_9$ and $e_{13}$ become the *low* and the *high event* respectively for this $C_4$. It is possible

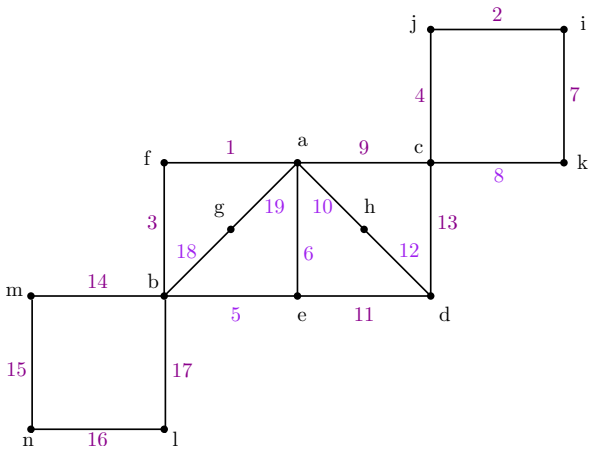



Figure 2: An RE graph with 14 vertices and 19 edges. The timestamps are mentioned as the integer numbers on the graphs edges.

that some edges participate in multiple subgraphs and thus become *high* or *low events* for more than one subgraphs in the same slice. There are three $C_4$'s in $G_{6,13}$, $(e_6, e_9, e_{11}, e_{13}), (e_6, e_{10}, e_{11}, e_{12})$ and $(e_9, e_{10}, e_{12}, e_{13})$. For two of these subgraphs, $e_{13}$ is the *high event*, though each of them have different *low events*.

Throughout the paper, adjacency linked lists are used to represent an RE graph $G$. There will be two copies of each edge $(u, v)$ for each endpoints $u$ and $v$. Each node of the linked list for $u$ stores its neighbour $v$ and the timestamp of the edge $(u, v)$.

Arboricity $a(G)$ of a graph $G = (V, E)$ having $m = |E|$ edges and $n = |V|$ vertices is the minimum number of edge-disjoint spanning forests into which $G$ can be partitioned [18]. Chiba and Nishizeki [13] gave an upper bound on $a(G)$ for a general graph $G$ as $a(G) \leq \lceil (2m+n)^{1/2}/2 \rceil$. Thus, for a connected graph $G$, $a(G) = O(\sqrt{m})$. We state the following lemma from their paper as this result will be used later in the section for subgraph counting.

**Lemma 1** *(Lemma 2.1 in [13]) If graph $G = (V, E)$ has $n$ vertices and $m$ edges, then $\sum_{(u,v)\in E} min\{deg(u), deg(v)\} \leq 2a(G)m$, where $deg(x)$ denotes the degree of vertex $x$ in $G$.*

## 2.2 Geometric Data Structures

The $d$-dimensional dominance counting problem for a set $S$ of $d$-dimensional points is to store $S$ in a data structure such that given a query point $q$ the points in $S$ that are dominated by $q$ can be counted quickly. Let $p = (p_x, p_y)$ and $q = (q_x, q_y)$ be two points in plane. We say $q$ *dominates* $p$, if $p_x \leq q_x$ and $q_y \leq p_y$. Note that this is a non-traditional definition of dominance. The standard dominance counting data structure considers dominance relationship to be $q_x \geq p_x$ and $q_y \geq p_y$, see Lemmas 1 and 2 in [1].

We can use this data structure to query for dominated points by mirroring the coordinates of the points. Given a point set $S$ with $n$ points in plane, the dominance counting query is to determine the total number of points of $S$ dominated by a query point $q$. See Figure 3(a) for an example.

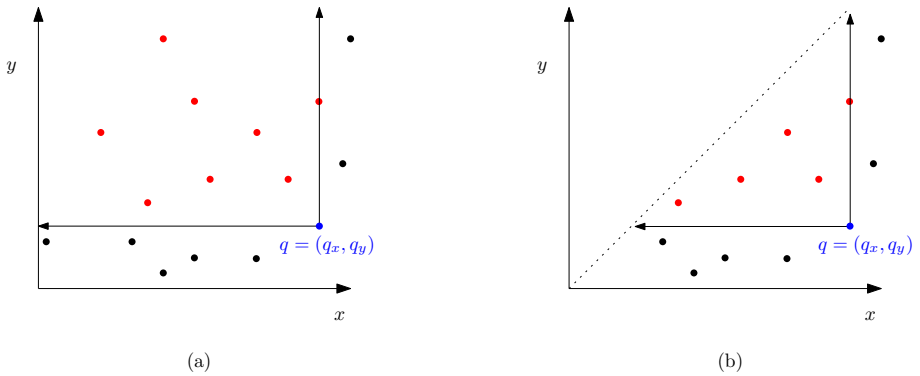



Figure 3: (a) All red points are dominated by a query point $q$. A red point $p = (p_x, p_y)$ is dominated by $q = (q_x, q_y)$, where $p_x \leq q_x$ and $q_y \leq p_y$. (b) A special case where all points are below the main diagonal of a grid.

We state the following results from [14] and [1] on dominance counting, range tree and interval tree data structures.

**Theorem 1** *[1, Theorem 2] Let $S$ be a set of $n$ $d$-dimensional points, where $d \geq 2$ is a constant. Then there exists a data structure for the $d$-dimensional dominance counting problem using $O(n(\log n/\log\log n)^{d-2})$ space such that queries can be answered in $O((\log n/\log\log n)^{d-1})$ time.*

The following result is from [4] that presents a data structure for window sensitive dominance queries on a set of $n$ points, where the integer coordinates of each point are in the range from 1 to $n$. Here, it is assumed that the points are below the main diagonal, see Figure 3(b).

**Theorem 2** *[4, Theorem 9] Let $S$ be a set of $O(n)$ points below the main diagonal of an $n \times n$ grid, then there exists a data structure of size $O(n)$ that can perform dominance queries for which the query point $(i, j)$ is at distance $d = (j-1)\sqrt{2}$ from the main diagonal in time $O(\log d/\log\log n)$ time, with $1 \leq i < j \leq n$.*

A $d$-dimensional *range tree* is a data structure for rectangular range queries, where each query is composed of $d$ 1-dimensional sub-queries respectively on the $x_1, x_2, \ldots, x_d$ - coordinates of the points.

**Theorem 3** *[14, Theorem 5.11] Let $P$ be a set of $n$ points in $d$-dimensional space, where $d \geq 2$. A range tree for $P$ uses $O(n\log^{d-1} n)$ space and it can*

*be constructed in $O(n \log^{d-1} n)$ time. With this range tree one can report the points in $P$ that lie in a rectangular query range in $O(\log^{d-1} n + k)$ time, where $k$ is the number of reported points.*

An *interval tree* data structure stores a set of $n$ axis-parallel line segments or intervals on the line.

**Theorem 4** *[14, Theorem 10.4] An interval tree for a set $I$ of $n$ intervals uses $O(n)$ storage and can be built in $O(n \log n)$ time. Using the interval tree we can report all intervals that contain a query point in $O(\log n + k)$ time, where $k$ is the number of reported intervals.*

# 3 Monotone Graph Properties

In this section, we present data structures to solve *time windowed decision problems* under some monotone graph property $\mathcal{P}$.

**Definition 1** *A graph property $\mathcal{P}$ is monotone if every subgraph of a graph with property $\mathcal{P}$ also has property $\mathcal{P}$.*

In other words, a graph property is monotone if it is closed under the removal of edges. For example, every subgraph of a planar graph is planar. Other examples of monotone graph property includes disconnectedness and bipartiteness.

*Problem statement:* Given an RE graph $G = (V, E)$ and a monotone graph property $\mathcal{P}$, we want to preprocess $G$ so that for any query time interval $[i, j]$ with $1 \leq i < j \leq n$, we can answer quickly whether the graph slice $G_{i,j} = (V, E_{i,j})$ satisfies $\mathcal{P}$.

## 3.1 Overview of the Algorithm

We present a general approach to reduce time windowed decision problems under monotone graph properties $\mathcal{P}$ to standard range searching problems using dynamic data structures that maintain $\mathcal{P}$. Suppose, there exists a dynamic data structure $\mathcal{D}$ that maintains some monotone graph property $\mathcal{P}$ and requires $S(n)$ space, $U(n)$ update time and $Q(n)$ query time. We further assume that, $\mathcal{D}$ accepts update operations such as edge insertions and deletions and allows queries that test whether the current graph satisfies $\mathcal{P}$. We preprocess edges of $G = (V, E = \{e_1, e_2, \ldots, e_m\})$ using $\mathcal{D}$ to find all the maximal subsequences of edges that satisfy $\mathcal{P}$. To be more specific, starting with $e_1$, we insert edges of $E$ according to the increasing timestamps of edges to $\mathcal{D}$ and check whether the subgraph formed by edges added so far satisfies property $P$. If it does, we continue adding edges to $\mathcal{D}$ until for the first time we find a graph slice $G_{1,b+1}$ that does not satisfy $P$ for some $b \geq 1$ (see Figure 4).

Now, for this first maximal subsequence of edges $e_1, e_2, \ldots, e_b$ that satisfies $\mathcal{P}$, we store a point $p = (1, b) \in \mathbb{R}^2$. Next, we keep deleting edges from $\mathcal{D}$ starting with $e_1$ and following the same sequence of edges as they have been

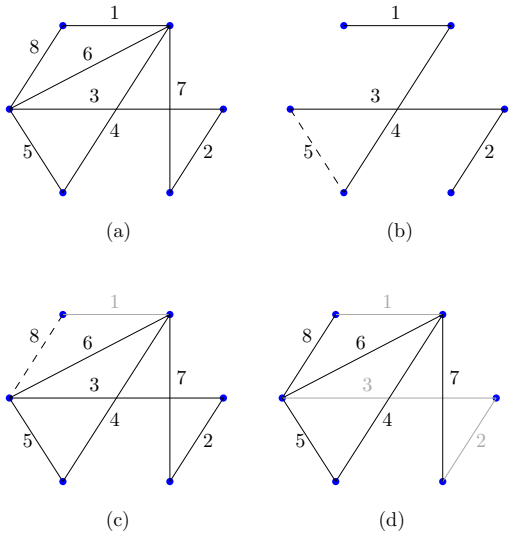



Figure 4: (a) An RE graph $G$ with eight edges. Timestamp of each edge is mentioned as an integer value. $G$ has three maximal subsequences of edges that satisfy the disconnectedness monotone property; they are (b) $e_1, \ldots, e_4$ ($e_5$ connects $G$), (c) $e_2, \ldots, e_7$ ($e_8$ connects $G$), and (d) $e_4, \ldots, e_8$ (preprocessing ends).

inserted until we find some $e_a$ such that $e_a, e_{a+1}, \ldots, e_{b+1}$ satisfies $\mathcal{P}$. To find the next maximal subsequence, we start inserting edges from $e_{b+2}$ and repeat the process. We continue processing edges until we scan all $m$ edges of $G$. At the end of this process, we will have a set of points $S$ in plane, where each point $(a, b) \in S$ represents a graph slice having a maximal contiguous edge set $\{e_a, e_{a+1}, \ldots, e_b\}$ that satisfies property $\mathcal{P}$. During preprocessing, each edge of $G$ is updated (inserted and deleted) and queried exactly once using $\mathcal{D}$. So the total time required for this preprocessing step is $O(m \cdot (U(n) + Q(n)))$.

Each point $(a, b) \in S$ obtained from this preprocessing step represents a time interval $[a, b]$, where $b > a$, such that $G_{a,b}$ satisfies $\mathcal{P}$ (see Figure 5(a)). Let $I_S$ be the set of all intervals found from $S$ in this way. A query time interval $[i, j]$ satisfies $\mathcal{P}$ if and only if $[i, j]$ is contained in some interval $[a, b] \in I_S$. We define the North-West quadrant of the point $(i, j)$ as $NW(i, j) = (-\infty, i] \times [j, \infty)$. Now we reduce this problem to range emptiness problem as stated in the following lemma.

**Lemma 2** *A query time interval $[i, j]$ is contained in some interval $[a, b] \in I_S$ if and only if $NW(i, j) \cap S \neq \emptyset$.*

**Proof:** For the '*if*' part, note that all points $(a, b) \in S$ are above the main diagonal as $b > a$. Therefore, when some query interval $[i, j]$ is contained in $[a, b] \in S_I$, it implies that $a \leq i < j \leq b$ and $(a, b) \in NW(i, j)$ (see Figure 5(b)).

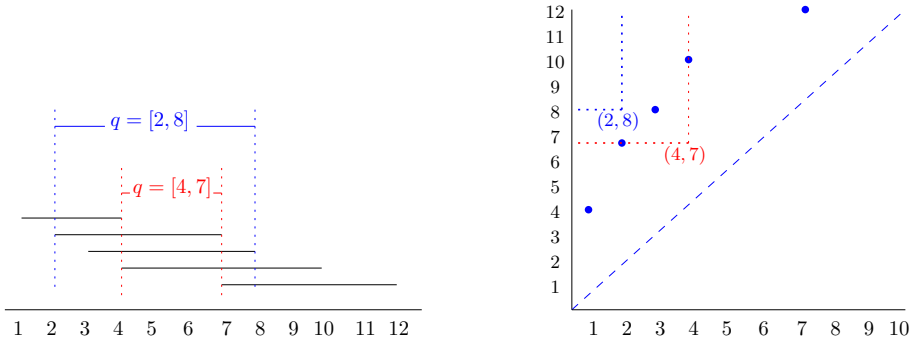



Figure 5: (a) An example illustrating a set of time intervals representing graph slices that satisfy some monotone property $\mathcal{P}$, (b) Query: for $q = [2, 8]$ (blue line) there is no point in $(-\infty, 2] \times [8, +\infty)$, hence $G_{2,8}$ does not satisfy $\mathcal{P}$, and for $q = [4, 7]$ (red line) there are points in $(-\infty, 4] \times [7, +\infty)$, hence $G_{4,7}$ satisfies $\mathcal{P}$.

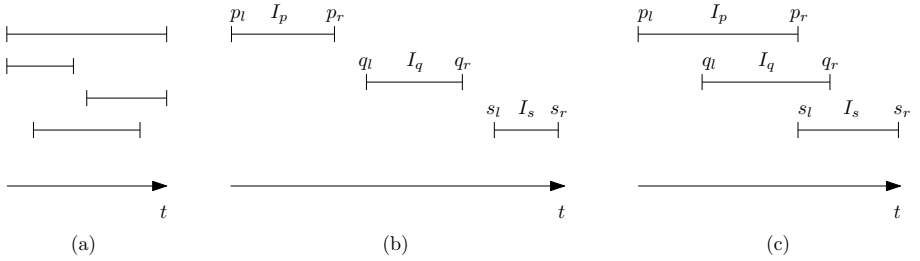



Figure 6: (a) Subsequences cannot be contained in other maximal subsequences. Permissible orientations of subsequences: (b) Non-overlapping subsequences, (c) Partially overlapping subsequences.

Now we prove the '*only if*' part. First we observe that $I_S$ can never contain two subsequences such that one is totally contained in another. See Figure 6(a). Our algorithm will always keep the subsequence with the maximum length. Moreover, there can be only two possible orderings of all subsequences of $I_S$ where two maximal subsequences will either partially overlap each other or they do not overlap at all, see Figures 6(b) and (c).

Suppose our algorithm does not identify a valid maximal subsequence $I_q$. Two cases have to be verified here.

Case (a): when $I_q = (e_{q_\ell}, \ldots, e_{q_r})$ does not overlap any other subsequences and it is in between two subsequences $I_p = (e_{p_\ell}, \ldots, e_{p_r})$ and $I_r = (e_{r_\ell}, \ldots, e_{r_r})$. I.e., the ordering of the timestamps of the edges are $p_r < q_\ell < q_r < r_\ell$. Note that $I_p$ is identified as a valid maximal subsequence because $e_{p_\ell}, \ldots, e_{p_r}$ satisfies some monotone property while $e_{p_\ell}, \ldots, e_{p_{r+1}}$ does not. So our algorithm will keep deleting edges from $e_{p_\ell}$ and check for satisfiability of the property. In this process all edges from $e_{p_\ell}$ to $e_{q_{\ell-1}}$ go through the insertion, verification and

deletion process. Finally edge $e_{q\ell}$ will be verified and if $I'$ is valid then all edges in the sequence $(e_{q_\ell}, \ldots, e_{q_r})$ must be identified by our algorithm.

Case (b): when $I_q$ partially overlaps $I_p$ and $I_r$ and suppose the ordering of the timestamps of the edges are $q_\ell < p_r < s_\ell < q_r < s_r$. As before our algorithm identifies that $e_{p_\ell}, \ldots, e_{p_r}$ satisfies some monotone property but $e_{p_\ell}, \ldots, e_{p_{r+1}}$ does not. Now if $I_q$ is valid, when edges $e_{p_\ell}, \ldots, e_{q_{\ell-1}}$ are deleted then $e_{q_\ell}, \ldots, e_{p_{r+1}}$ must satisfy the property and do not get deleted. In this step our algorithm must keep adding next edges in the sequence and identifies $e_{q_\ell}, \ldots, e_{q_r}$ as the valid maximal subsequence $I_q$. □

For any monotone property, this approach generates $O(m)$ maximal subsequences. So for any RE $G$, this preprocessing step produces a point set $S$, with $|S| \leq m$. We build a 2-dimensional priority search tree (PST), see [25], on the point set $S$. To answer queries of the form '*Given a query time interval* $[i, j]$ *s.t.* $i \leq j$*, does the graph slice* $G_{i,j}$ *satisfy property* $\mathcal{P}$*?*', we query this data structure using a grounded query rectangle $q = (-\infty, i] \times [j, +\infty)$. We report that $G_{i,j}$ satisfies $\mathcal{P}$ if the query returns a positive count. Total space required by a 2-dimensional PST is linear. Thus the total space requirement of this approach is $O(S(n) + m)$. A 2-dimensional PST on $m$ points can answer each grounded query in time $O(\log m)$. If $G$ is completely connected then $m = O(n^2)$ and therefore $O(\log m) = O(\log n)$. We will use this assumption through out the paper. We summarize the result in the following theorem.

**Theorem 5** *Suppose, $\mathcal{D}$ is a dynamic algorithm that maintains a monotone graph property $\mathcal{P}$ using $S(n)$ space, and requires $U(n)$ time per update and $Q(n)$ time per query. Given an RE graph with $n$ vertices and $m$ edges and an arbitrary query time interval $[i, j]$, we can reduce the time windowed decision problem for reporting whether $G_{i,j}$ satisfies a monotone property $\mathcal{P}$ to a 2-dimensional range searching query in $O(m \cdot (U(n) + Q(n))$ time using $O(S(n) + m)$ space that can answer queries in $O(\log n)$ time.*

## 3.2 Bipartiteness

A graph is bipartite if the set of its vertices can be decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. We directly use Henzinger and King's [20] dynamic algorithm for maintaining bipartiteness of a graph that supports each update in $O(\log^3 n)$ time and each query can be answered in $O(1)$ time. Their data structure uses $S(n) = O(m + n \log n)$ space. Therefore we obtain the following result.

**Corollary 1** *We can preprocess an RE graph $G$ with $n$ vertices and $m$ edges into a data structure of size $O(m + n \log n)$ in $O(m \log^3 n)$ time such that a time windowed bipartiteness decision query can be answered in $O(\log n)$ time.*

## 3.3 Disconnectedness

Let $G = (V, E)$ be a graph such that $G$ is not connected. Then observe that for any subset $E' \subseteq E$, $G' = (V, E')$ is also disconnected. Thus the property of being

disconnected is monotone. Following our algorithm, as described in Section 3.1, we first identify all maximal subsequences of edges $e_a, e_{a+1}, \ldots, e_{b-1}, e_b$ such that $G$ is not connected by the edges of $G_{a,b}$. We use the deterministic dynamic connectivity algorithm by Holm et al. [21, Theorem 3] to answer connectivity queries. This dynamic data structure $\mathcal{D}$ uses $S = O(m + n \log n)$ space, amortized $U(n) = O(\log^2 n)$ update time and worst case $Q(n) = O(\log n / \log \log n)$ query time to answer whether two vertices $u$ and $v$ are connected in a given graph $G$.

However, our time windowed queries ask for the connectivity of the entire graph and not just any two vertices. We describe here how to use the same data structure $\mathcal{D}$ to answer the full connectivity query by executing an additional check. We keep a *count* that stores the number of connected components of $G$. Initially, *count* is equal to total number of vertices since no edges have been inserted into the data structure so far, and thus each vertex represents one component. Suppose at some time during preprocessing $G$, we finished processing edge $e_{k-1}$ and now we want to insert an edge $e_k = (u, v)$ into this structure. Also assume that vertices $u$ and $v$ are not connected in $\mathcal{D}$. Since inserting $e$ into $\mathcal{D}$ will connect two previously disconnected components containing vertices $u$ and $v$, respectively, into one component, the total number of components maintained by $\mathcal{D}$ will be reduced by one. So *count* is decreased by 1. By a similar argument, every time an edge $e$ is deleted from $\mathcal{D}$ the total number of components maintained by this structure will be increased by one. We update *count* accordingly during processing each edge. Thus, we know that a graph slice is connected only when *count* equals 1. Updating *count* takes $O(1)$ time per edge insertion and deletion.

In total, we can find at most $O(m)$ maximal subsequences of edges that satisfy the connectivity property, or equivalently, $O(m)$ points to be stored in the dominance counting structure. So, our windowed data structure for disconnectedness requires $O(m + n \log n)$ space and $O(m(\log^2 n + \log n / \log \log n)) = O(m \log^2 n)$ time for preprocessing. Using Theorem 5, we obtain the following result for the windowed query for disconnectedness.

**Corollary 2** *We can preprocess an RE graph $G$ with $n$ vertices and $m$ edges into a data structure of size $O(m+n \log n)$ in $O(m \log^2 n)$ time such that a time windowed decision query for disconnectedness can be answered in $O(\log n)$ time.*

# 4 Problems on Minimum Spanning Trees

In this section we solve three window query problems using the minimum spanning tree (MST) of a weighted RE graph. In particular, we first show an application of dynamic algorithm to construct a data structure to report whether an MST exists in the query time interval. Next we report the minimum spanning interval of an MST for any query time interval. Then, we show how to report the graph edit distance to transform a query graph slice into a spanning forest.

## 4.1 Weight of the MST

Suppose a weighted RE graph $G = (V, E)$ is given where each edge of $G$ has a positive numerical weight. Let the weight of the MST of $G$ be denoted by $\omega^*$. For a query time interval $[i, j]$, we want to report whether there exists some MST $T = (V, E')$ such that $E' \subseteq \{e_i, \cdots, e_j\}$ with weight $\omega^*$.

We utilize two dynamic data structures to preprocess the edges of $G$. The first dynamic structure (due to Holm et al. [21]) is used to maintain the minimum spanning forest (MSF) of $G$ using the edges inserted so far. We call this data structure $\mathcal{D}_1$. We require a second dynamic structure to verify whether the MSF maintained by $\mathcal{D}_1$ is connected, i.e., it is also the MST of $G$. We use the dynamic connectivity algorithm that we have used to solve the time windowed problem for disconnectedness (see Section 3.3) as our second data structure, namely $\mathcal{D}_2$.

Note that $\mathcal{D}_1$ maintains the MSF of $G$. We augment $\mathcal{D}_1$ so that every time an edge $e$ is inserted into $\mathcal{D}_1$, it reports which edge has been added to the MSF and which edge (if any) has been deleted from the MSF. After each update, $\mathcal{D}_1$ also updates the total weight of the current MSF. We have the following cases to consider.

1. The newly inserted edge $e$ does not create any cycle with the existing tree edges and therefore is added to the current MSF. $\mathcal{D}_1$ reports that $e$ has been added to the MSF.

2. The newly inserted edge $e$ creates a cycle with the existing tree edges. If the weight of $e$ is less than that of any other edge of the cycle then $e$ replaces the edge with the highest weight in that cycle. Then $\mathcal{D}_1$ reports that $e$ has been added to the MSF and the edge with the highest weight in the corresponding cycle has been deleted from MSF.

3. Otherwise, the MSF does not change and $\mathcal{D}_1$ does not report anything.

These simple augmentations can be done without making any changes to the preprocessing and space time bounds of the original data structure.

Now we discuss the preprocessing. For $k = 1$ to $m$, we insert edge $e_k$ into $\mathcal{D}_1$ according to their increasing timestamps and check whether $e_k$ becomes a new tree edge in $\mathcal{D}_1$. We insert $e_k$ into $\mathcal{D}_2$ in two cases; (a) when $e_k$ becomes a new tree edge and (b) when $e_k$ replaces an existing tree edge. Every time a new tree edge $e_k$ is inserted into $\mathcal{D}_2$ we check the connectivity of the current subgraph using $\mathcal{D}_2$. Every time a new tree edge $e_k$ replaces an old tree edge $e_{k'}$ in $\mathcal{D}_1$, we insert $e_k$ into $\mathcal{D}_2$ and delete $e_{k'}$ from $\mathcal{D}_2$. We keep repeating this process until for the first time we find a subsequence of edges $e_1, \ldots, e_q$ that has the the MST of $G$ with weight $\omega^*$. Observe that in case where $q > n - 1$, some of the edges in this subsequence with higher weights have been replaced by some other lighter edges from the same subsequence. We start deleting from edge $e_1$ and check whether $e_2, \ldots, e_q$ still holds the MST of weight $\omega^*$. We keep deleting edges according to the same sequence of edge insertion until we find

the minimal subsequence $I = e_p, \ldots, e_q$ where the MST exists. We store a point $(p, q) \in \mathbb{R}^2$ marking that there exists an MST of $G$ in the interval $[p, q]$. Now we delete edge $e_p$, insert edge $e_{q+1}$ and repeat the whole process until we finish scanning all edges of $G$. At the end of the preprocessing we have a set of points $S' \in \mathbb{R}^2$, where each point $(p.q) \in S'$ represents a minimal subsequence in which an MST with weight $\omega^*$ exists. Let $I_{S'}$ be the set of all intervals obtained by this process. Now the following lemma holds.

**Lemma 3** *A query time interval $[i, j]$ contains some interval $[p, q] \in I_{S'}$ if and only if $SE(i, j) \cap S' \neq \emptyset$, where $SE(i, j) = [i, +\infty) \times (-\infty, j]$ is the South-East quadrant of the point $(p, q)$.*

The proof is similar to the one presented for Lemma 2 and hence omitted.

**Theorem 6** *Suppose a weighted RE graph $G$ is given with $m$ edges and $n$ vertices, and let $G$ has an MST of weight $\omega^*$. Given a query time slice $[i, j]$, the problem of reporting whether $G_{i,j}$ contains an MST of $G$ with weight $\omega^*$ can be reduced to a 2-dimensional range searching query in $O(m \log^4 n)$ time using $O(m \log n)$ space. Queries can be answered in $O(\log n)$ time.*

**Proof:** We observe that no two intervals in $I_{S'}$ can have same start time. Therefore, there can be at most one interval starting from any time $k$ with $1 \leq k \leq n$. So the number of MSTs that can possibly be generated over $m$ edge updates is at most $O(m)$. Holm et al.'s algorithm maintains an MSF using amortized $O(\log^4 n)$ time per update and $O(m \log n)$ space [21, Theorem 8]. As mentioned before, the dynamic connectivity algorithm by the same authors provides $O(\log^2 n)$ update time using $O(m + n \log n)$ space. Similar to the data structure of Section 3.1 we build a 2-dimensional PST on $O(m)$ points using linear space and query using a grounded query rectangle $[i, +\infty) \times (-\infty, j]$ in $O(\log n)$ time. We report there exists an MST of $G$ in $G_{i,j}$ if the query returns a positive count. □

## 4.2 Minimum Spanning Interval

We define a *spanning interval* as the time difference $t(e_q) - t(e_p)$ such that there exists an MST $T = (V, E' \subseteq \{e_p, \cdots, e_q\})$ of $G$. The motivation for solving this problem is the fact that given a query time interval $[i, j]$ multiple MSTs of $G$ can exist in the graph slice $G_{i,j}$, and we are interested to find a minimum spanning interval that contains an MST. To solve this problem we change the data structure described in Section 4.1 as follows. The length of the minimal subsequence $e_p, \ldots, e_q$ that holds an MST is $q - p + 1$. So now we consider each point $a = (p, q)$ as a weighted point and initialize the weight of each point to $w(a) = q - p + 1$. Therefore the problem of finding the minimum spanning interval within $G_{i,j}$ can be reduced to the orthogonal range minimum problem on $O(m)$ points. According to Chan et al. [8], orthogonal range minimum problem can be solved in $O(\log \log n)$ time using $O(m \log^\epsilon n)$ space, where $\epsilon$ is a small positive constant. We summarize the result here.

**Theorem 7** *Given an RE graph $G$ with $m$ edges and $n$ vertices, and a query time slice $[i, j]$, the problem of reporting the minimum spanning interval in $G_{i,j}$ can be reduced to the orthogonal range minimum problem in $O(m \log^4 n)$ time using $O(m \log^{\epsilon} n)$ space. Queries can be answered in $O(\log \log n)$ time.*

## 4.3 Graph Edit Distance for Target Class Forest

**Definition 2** *Given a set of graph edit operations (insertion or deletion of graph edges), the graph edit distance $\mathcal{GED}(G, H)$ between a source graph $G$ and a target graph $H$ is defined as follows.*

$$\mathcal{GED}(G, H) = \min\{c(S) | S \text{ is a sequence of operations transforming } G \text{ into } H\}$$

In this definition, $S = (s_1, s_2, \ldots, s_k)$ is a sequence of operations that transforms $G$ into $H$. The cost of a sequence $S = (s_1, s_2, \ldots, s_k)$ is given by $c(S) = \sum_{i=1}^{k} c(s_i)$, where $c(s_i)$ is the cost of the operation $s_i$. The goal of the graph edit distance is to find the minimum cost of the operations that makes the transformation possible. We consider *edge deletion* as the only permitted graph edit operation to solve our problem. We assume that for unweighted graphs, each edit operation has a unit cost. In this section, we want to solve the following problem.

Given an RE graph $G = (V, E)$ and a query time interval $[i, j]$, we want to compute the $\mathcal{GED}(G_{i,j}, H)$ where $H = (V, E')$ is a spanning forest of $G_{i,j}$ and $E' \subseteq \{e_i, e_{i+1}, \ldots, e_j\}$. This is equivalent to saying that we want to find a set of edges $X$ such that $G_{i,j} \setminus X$ is a forest and $|X|$ is minimum.

First we present the following observation.

**Lemma 4** *The total cost $\mathcal{C}$ of $\mathcal{GED}(G_{i,j}, H)$ is equivalent to the number of chordless cycles in $G_{i,j}$, where $H$ is the spanning forest of $G_{i,j}$.*

**Proof:** Let $C$ be a simple cycle without any chords. Observe that by deleting any edge of $C$, we obtain a spanning forest (tree) of $C$. Also note that graph edit distance must report the minimum cost of operations. Since one edge deletion is required for every chordless cycle and every edge deletion operation has unit cost, the total cost $\mathcal{C}$ of $\mathcal{GED}(G_{i,j}, H)$ is the same as the total number of edge deletions in $G_{i,j}$. Therefore, reporting $\mathcal{GED}(G_{i,j}, H)$ is equal to the number of chordless cycles in $G_{i,j}$. Therefore the lemma holds. □

*Algorithm:* We maintain a link-cut tree $T$, see [29], to store the vertices of $G$. A link-cut tree allows edge insertions and deletions in amortized time $O(\log n)$. This data structure also supports standard aggregate functions (e.g., max, min, sum or increment) over all the edges from a vertex $v$ to the root of $T$ in time $O(\log n)$ [29]. We store the timestamp of an edge as its weight so that we can query $T$ to find the edge with the minimum timestamp that is on the path from $v$ to the root.

For $k = 1$ to $m$, we process each edge $e_k$ in increasing order of the timestamp and add $e_k$ to $T$ unless it creates a cycle with the existing edges of $T$. Suppose $e_k = (u_k, v_k)$ is an edge with $u_k$ and $v_k$ as its two end points, that creates a cycle in $T$. Then there must be an existing path from $u_k$ to $v_k$ in $T$. We query $T$ to find the edge $e_\ell$ with the minimum timestamp $t(e_\ell)$ on this path. We store a point $(k, \ell) \in \mathbb{R}^2$ with label $\ell$. We delete $e_\ell$ from $T$ and insert $e_k$ to $T$. We repeat these steps until we finish processing all edges in $E$. At the end of this process we obtain a set of labelled points $P$ with $|P| = O(m)$ in $\mathbb{R}^2$. Figure 7 illustrates an example of our algorithm for computing $\mathcal{GED}$ for forests.

**Lemma 5** *The number of points in $P \cap ([i, j] \times [i, j])$ is equal to the $\mathcal{GED}$ of $G_{i,j}$.*

**Proof:** Note that every point $(k, \ell) \in \mathbb{R}^2$ that represents a chordless cycle in the interval $[\ell, k]$ lies below the main diagonal. The number of points $|X| = P \cap ([i, j] \times [i, j])$ gives us the total number of chordless cycles that exist in time interval $[i, j]$. By Lemma 4, the number of points in $|X|$ is equal to the $\mathcal{GED}$ of $G_{i,j}$. □

**Theorem 8** *Given an RE graph $G$ with $m$ edges and $n$ vertices, and a query time slice $[i, j]$, the problem of reporting the $\mathcal{GED}$ for target class forest can be reduced to the range searching problem in $O(m \log n)$ time using $O(m+n)$ space. Queries can be answered in $O(\log n + |X|)$ time using $O(m \log n)$ space, where $X$ is the set of edges such that $G_{i,j} \setminus X$ is a forest and $|X|$ is minimum.*

**Proof:** The preprocessing step ensures that when an edge $e_k$ creates a cycle such that $e_\ell$ is an edge in that cycle with the smallest timestamp, the interval $[\ell, k]$ contains exactly one chordless cycle. According to Lemma 4 this chordless cycle in $G_{\ell,k}$ must contribute a unit cost to the graph edit distance. So a point $(k, \ell) \in \mathbb{R}^2$ with label $\ell$ represents that a cycle exists in time interval $[\ell, k]$ and $e_\ell$ is the edge that must be deleted to maintain the forest. It is also necessary to delete $e_\ell$ physically from $T$ to ensure that the next cycle found by the scanning process is chordless. Labels are required only if we want to report the edges. Otherwise if we want to report the value of $\mathcal{GED}$ only (i.e., the total number of edges to be deleted) then we can ignore labels.

Every edge will be inserted and deleted from $T$ at most once in this process. Queries are also made at most once for each edge. Therefore, total preprocessing time is $O(m \log n)$ using $O(n + m)$ space [29]. As mentioned above $P$ has $O(m)$ labelled points in $\mathbb{R}^2$. $P$ can be stored using a standard range search tree that can be built in $O(m \log n)$ time using $O(m \log n)$ space [14].

For a given query time interval $[i, j]$, we count the number of points $|X| = P \cap q$ that intersect with the query rectangle $q = [i, j] \times [i, j]$ in $O(\log n)$ time. By Lemma 5 we report $|X|$ as the $\mathcal{GED}$ for converting $G_{i,j}$ to a forest, i.e., $|X|$ is the number of edges needed to be deleted from $G_{i,j}$ such that the resulting graph slice becomes a forest. If the query is to report the set of edges that are needed to be deleted, we can answer in $O(\log n + |X|)$ time to report the labels of the points (i.e., timestamps of the deleted edges), where $X$ is the set of deleted edges. □

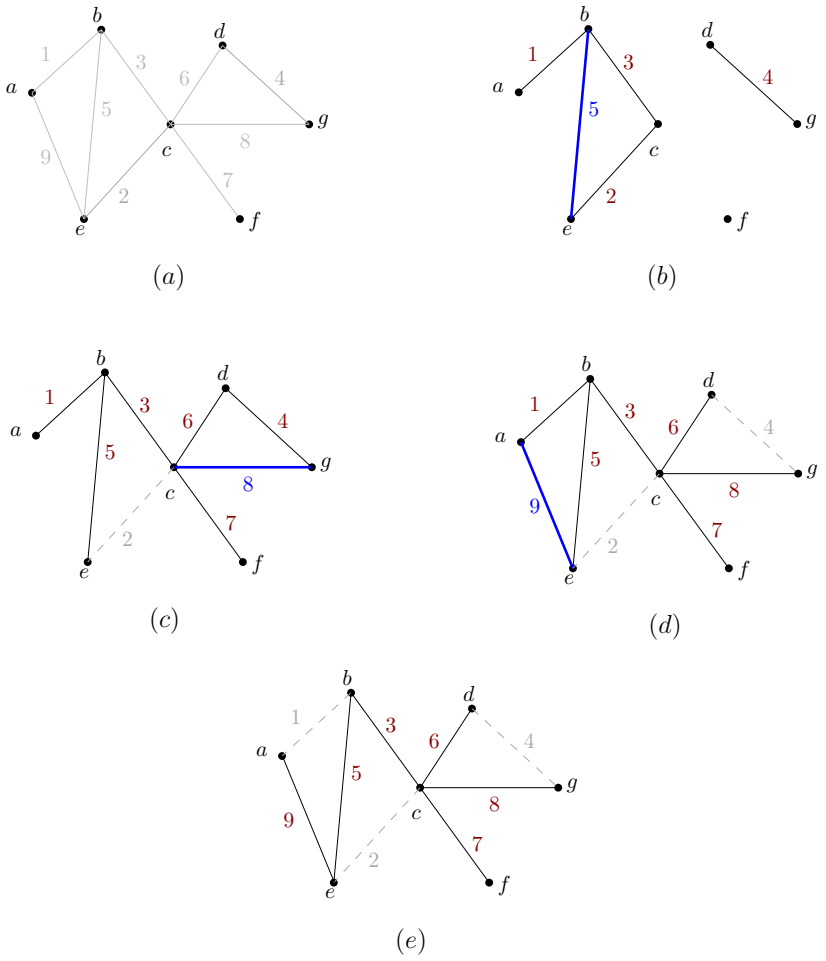



Figure 7: (a) An RE graph $G$ with seven vertices stored in a link-cut tree $T$. Edges are not inserted yet. (b) Starting with $e_1$ edges are inserted to $T$ according to the increasing timestamps until $e_5$ creates a cycle $C_1 = \langle e_2, e_3, e_5 \rangle$ (timestamp of each edge is mentioned as an integer value). (c) The edge with the lowest timestamp in $C_1$ is $e_2$ and a point $(5, 2) \in \mathbb{R}^2$ with label 2 is stored. Edge $e_2$ is deleted and $e_6, \ldots, e_8$ are inserted to $T$. (d) Edge $e_8$ creates another cycle $C_2 = \langle e_4, e_6, e_8 \rangle$. Similarly the edge with the lowest timestamp $e_4$ is deleted from $T$ and a point $(8, 4) \in \mathbb{R}^2$ with label 4 is stored. Edge $e_9$ is inserted to $T$. (e) $e_9$ creates cycle $C_3 = \langle e_1, e_5, e_9 \rangle$. Again the edge with the lowest timestamp $e_1$ in $C_3$ is deleted from $T$ and a point $(9, 1) \in \mathbb{R}^2$ with label 1 is stored. (e) $\mathcal{GED}(G_{1,9}, H) = 3$, where $H$ is a spanning forest of $G$. Reported edges are 1, 2 and 4. Bold blue edges create cycles, solid black edges are part of $H$ and dashed grey edges are deleted from $G_{1,9}$.

# 5 Problems on Counting Subgraphs

In this section we consider the problems of counting subgraphs of some fixed patterns in a queried graph slice $G_{i,j}$. The patterns for which we present data structures are 2-paths (paths of length two in general graphs), 3-paths (paths of length three in bipartite graphs), all complete subgraphs of size 3 and more, and quadrangles ($C_4$) or simple cycles of length 4.

## 5.1 Counting 2-paths and 3-paths

First, we consider the case of counting 2-paths. Let $G = (V, E)$ be an RE graph with $n$ vertices and $m$ edges. We maintain a set of $n$ lists, one for each vertex $v_k \in V$, where $1 \leq k \leq n$. The list $L_{v_k}$ stores timestamps of $v_k$'s incident edges, sorted in increasing order. The length of this list is $deg(v_k)$, where $deg(v_k)$ is the degree of $v_k$ in $G$. Therefore, the total space over all the lists is at most $\sum_{v_k \in V} deg(v_k) \leq 2m = O(m)$.

Next we analyze the complexity of computing all 2-paths in $G$. For a query time slice $[i, j]$, suppose, $i'$ and $j'$ are the indices of the elements in $L_{v_k}$ such that $L_{v_k}[i']$ is the smallest element $\geq i$, and $L_{v_k}[j']$ is the largest element $\leq j$. Then the total number of 2-paths centering $v_k$ in $G_{i,j}$ will be $\binom{j'-i'+1}{2}$. For each vertex $v_k$, the number of 2-paths where $v_k$ is the 'center' vertex can be computed by performing a binary search in the list $L_{v_k}$ to locate $i'$ and $j'$ as discussed above. This requires $O(\log deg(v_k))$ time. Thus we can report all 2-paths in $G_{i,j}$ in $O(\sum_{k=1}^{n} \log deg(v_k)) = O(n \log n)$ time. Using *fractional cascading* data structure, see [12], we can improve the query time to $O(n)$ time without incurring any increase in space as follows.

Fractional cascading data structure is generally used to solve *iterated search* problem, where many search problems can be solved by first solving a subproblem whose size is a constant fraction of the original problem size and then using this solution to obtain the solution of the original problem. We provide a brief description of how this data structure can be built and used in our case. We define $Y_{v_n} = L_{v_n}$. Suppose we take a sample of size $\lfloor n/2 \rfloor$ from $Y_{v_n}$ by selecting every alternate elements. Then elements from this sample list are merged with $L_{v_{n-1}}$ to obtain $Y_{v_{n-1}}$. For each element $k$, with $1 \leq k \leq n$, in $Y_{v_{n-1}}$ we use two pointers that points respectively to the smallest integer $p$ such that $Y_{v_n}[p] \geq k$ and to the first element of $Y_{v_{n-1}}$ that comes from $L_{v_{n-1}}$ and appears after $k$. We repeat this process of using $L_{v_{n-2}}$ and $Y_{v_{n-1}}$ to obtain $Y_{v_{n-2}}$ until we obtain $Y_{v_1}$. Total space required is $O(n)$. Now the iterative query can be answered by first using a binary search in some $L_{v_k}$ with keys $i'$ and $j'$, where $1 \leq k \leq n$. For the subsequent steps we can find each pair of $L_{v_{k+1}}[i'']$ and $L_{v_{k+1}}[j'']$ from the information of $L_{v_k}[i']$ and $L_{v_k}[j']$ in constant time. Therefore queries can be answered in $O(n + \log n) = O(n)$ time.

Now we present data structure for counting all 3-paths in a bipartite RE graph $G = (V = \{A \cup B\}, E = \{(u, v) : u \in A, v \in B\})$. We maintain two cascading structures. The first structure $\mathcal{D}_1$ maintains all 2-paths centering each vertex $u_k \in A$ with $1 \leq k \leq |A|$. $\mathcal{D}_2$ is a similar structure for each vertex

$v_\ell \in B$ with $1 \leq \ell \leq |B|$. We also maintain an array, namely $count[1..|A|]$, that initially stores zero in each $count[k]$ with $1 \leq k \leq |A|$. We update this array during the query with the following information. For query interval $[i, j]$, each $count[k]$ will store $deg_{i,j}(u_k)$, i.e., the number of edges adjacent to $u_k$ in $G_{i,j}$, for all $u_k \in A$, see Figure 8.

The data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ are adjacency linked lists. In $\mathcal{D}_1$, every node on $u_k$'s list contains two fields, a vertex $v$ and the timestamp $t(u_k, v)$ such that $(u_k, v) \in E$, $u_k \in A$ and $v \in B$ (see Figure 8(b)). In $\mathcal{D}_2$'s structure, every node on $v_\ell$'s list also has two similar fields and an extra pointer field that points to $count[k]$, if the edge $(v_\ell, u_k)$ appears in $G_{i,j}$ (see Figure 8(c)). Total preprocessing takes $O(m + n)$ time.

The query is processed in two steps. Given a query time interval $[i, j]$, we first perform a binary search on $\mathcal{D}_1$ using key values $i$ and $j$. This process is same as counting 2-paths. For every $u_k \in A$, with $1 \leq k \leq |A|$, we store the number of edges adjacent to $u_k$ within time interval $[i, j]$ in position $count[k]$. This step takes $O(n)$ time using fractional cascading.

Next, we do a similar binary search on $\mathcal{D}_2$, and similar to the process described above for 2-paths, let $i'$ and $j'$ be the indices of the elements in $L_{v_\ell}$ such that $L_{v_\ell}[i']$ is the smallest element $\geq L_{v_\ell}[i]$, and $L_{v_\ell}[j']$ is the largest element $\leq L_{v_\ell}[j]$. This time, we walk along each element from $L_{v_\ell}[i']$ to $L_{v_\ell}[j']$ for every $v_\ell \in B$, and follow the pointer of each $u_k \in N(v_\ell)$ to reach $count[k]$.

The number of 3-paths in $G_{i,j}$ of the form $\langle a, v_\ell, u_k, b\rangle$, such that $v_\ell \in B$ is a fixed vertex and $u_k \in A$ is a fixed neighbour of $v_\ell$, are equal to $(deg_{i,j}(v_\ell) - 1) \times (count[k] - 1)$, given that $deg_{i,j}(v_\ell) > 1$ and $count[k] > 1$. Otherwise no 3-path exists of this form. So, the total number of 3-paths passing through all the neighbours $u_k \in N(v_\ell)$ of a fixed $v_\ell$ in $G_{i,j}$ will be

$$(deg_{i,j}(v_\ell) - 1) \times \sum_{\forall k: u_k \in N_{i,j}(v_\ell)} (count[k] - 1).$$

Where $N_{i,j}(v_\ell)$ denotes neighbors of $v_\ell$ in $[i, j]$. Therefore, the count of all 3-paths in $G_{i,j}$ will be

$$\sum_{\ell=1}^{|B|} \Big( (deg_{i,j}(v_\ell) - 1) \times \sum_{\forall k: u_k \in N_{i,j}(v_\ell)} (count[k] - 1) \Big).$$

We can find the number of adjacent edges of all $u_k \in A$ with $i \leq k \leq j$, and of all $v_\ell \in B$ with $i \leq \ell \leq j$, in $O(n)$ time using fractional cascading on $\mathcal{D}_1$ and $\mathcal{D}_2$. Since scanning neighbours of all $v_\ell$, with $1 \leq \ell \leq |B|$, requires at most $O(m)$ time, total query time to find all 3-paths in $G_{i,j}$ is $O(n + m)$. We have used an array of size at most $n$, and two copies of the similar data structure that has been used for counting 2-paths. Thus the total space requirement is $O(m + n)$.

**Theorem 9** *(a) Let $G = (V, E)$ be an RE graph with $n$ vertices and $m$ edges. $G$ can be preprocessed into a data structure in $O(n + m)$ time using $O(n + m)$*

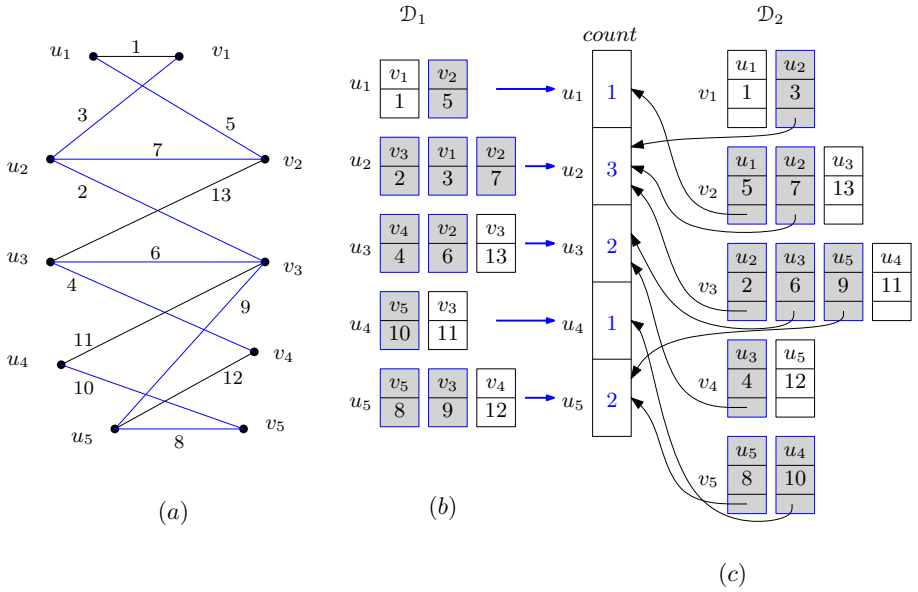



Figure 8: (a) A bipartite RE graph $G$, (b) Fractional cascading structure $\mathcal{D}_1$, (c) Fractional cascading structure $\mathcal{D}_2$. An example illustrating how $\mathcal{D}_1$ and $\mathcal{D}_2$ are queried with time interval [2,10].

*space such that the total number of 2-paths in $G_{i,j}$ can be counted in $O(n)$ time. (b) Let $G = (V, E)$ be a bipartite RE graph with $n$ vertices and $m$ edges. $G$ can be preprocessed into a data structure in $O(n + m)$ time using $O(n + m)$ space such that the total number of 3-paths in $G_{i,j}$ can be counted in $O(n + m)$ time.*

## 5.2 Counting Complete Subgraphs of a Fixed Order $\ell \geq 3$

In this section we present the general overview of the technique to count all complete subgraphs of order $\ell \geq 3$ in an RE graph $G = (V, E)$. The *order* of $G$ is the number of vertices in $G$. A complete subgraph $K_\ell$ of $G$ is a subgraph induced by a subset of the vertices $V$ such that every two distinct vertices are adjacent in $K_\ell$, where $\ell$ is the order of $K_\ell$.

*Preprocessing step*: We modify the edge-scanning algorithm presented by Chiba and Nishizeki [13] that originally reports all complete subgraphs of a fixed order $\ell \geq 3$ in a graph as follows. Vertices of $G$ are sorted in non-increasing order of their degrees and without loss of generality, let $deg(v_1) \geq deg(v_2) \geq \cdots \geq deg(v_n)$, where $deg(v)$ is the degree of a vertex $v$ in graph $G$. Starting with vertex $v_1$ the algorithm marks the subgraph induced by the neighbours $N(v_1)$ of $v_1$, and finds all instances of a specified complete subgraph that contains $v_1$. For each of these instances, we compute the *high* and the *low* values by comparing the timestamps of participating edges. Recall that *high* (or *low*) is the highest

(or the lowest) timestamp of all the edges involved in a subgraph. The interval $[low, high]$ represents the timespan of that subgraph in $G$. Each subgraph is then represented by a point $(high, low)$ in $\mathbb{R}^2$. Once all subgraphs containing $v_1$ are stored as a set of points, $v_1$ is deleted from $G$ to avoid duplication and the process continues with the next vertex in the sequence. At the end of the process a set of point $P \in \mathbb{R}^2$ representing all complete subgraphs of order $\ell$ is obtained.

In [13], the algorithm finds a complete subgraph $K_\ell$ containing a vertex $v$ by detecting $K_{\ell-1}$ in the subgraph induced by the neighbours of $v$. We modify their algorithm (see modified Algorithms 1 and 2 as follows. An initially empty global stack $S$ is used that now stores $(\ell - k)$ elements of the form $(v_i, min_{v_i}, max_{v_i})$ at the recursion level $(\ell - k)$. All vertices $v_i$ stored in $S$ are pairwise adjacent in $G$.

Algorithm 2 (procedure CS$(k, G_k)$) finds all complete subgraph of order $k$ in $G_k$. Each of these complete subgraphs forms a $K_\ell$ together with $(\ell - k)$ vertices stored in $S$. When procedure CS$(k, G_k)$ is executed, at a recursive call of depth $(\ell - k)$, we compute *low* and *high* events (denoted with $min_{v_i}$ and $max_{v_i}$, respectively) that are respectively the minimum and the maximum timestamps within the complete subgraph consisting vertices $(v_1 \dots v_i)$. When $k$ becomes 2, for each edge $(x, y)$ left in $G_2$, we list vertices $\{x, y\} \cup S$ that form $K_\ell$. At this stage, we use the timestamps of edges formed by connecting $x$ and $y$ with each vertex $z \in S$ to update the *low* and the *high* events of $K_\ell$. A point $(high, low)$ is stored in $\mathbb{R}^2$ that represents a $K_\ell$ in the time interval $[low, high]$. The graph representation requires linear space using adjacency lists data structure. The linear space implementation of counting complete subgraphs of fixed order using the same data structure is available in [13]. Thus total space requirement becomes $O(m + n + \mathcal{K})$, where $\mathcal{K}$ is the number of complete subgraphs of the fixed order $\ell$. We obtain the following results.

**Theorem 10** *Given an RE graph $G = (V, E)$ with $m$ edges and $\mathcal{K}$ complete subgraphs of a fixed order $\ell$ ($\geq 3$), the problem of determining the number of complete subgraphs of order $\ell$ in the query time interval $[i, j]$ can be reduced to dominance counting in $O(a(G)^{\ell-2}m)$ time using $O(m+n+\mathcal{K})$ space. The query takes $O(\log \mathcal{W} / \log\log \mathcal{K})$ time to count the total number of complete subgraphs of order $\ell$ in $G_{i,j}$, where $\mathcal{W}$ is the width of the query window.*

**Proof:** We follow the proof technique used in [13]. Let $n = |V|$ and $m = |E|$. The function PreprocessCS$(\ell, G)$ recursively calls procedure CS$(k, G_k)$ with $k = \ell$ and $G_k = G$. Let $T(k, m, n)$ be the time required by procedure CS$(k, G_k)$ to find all $K_k$'s in $G_k$.

When $k = 2$: for each edge in $G_2$ we update *low* and *high* values of $K_k$ by comparing timestamps of edges connecting at most $\ell$ vertices. This time is upper bounded by $O(m + n)$. So, $T(2, m, n) = O(m + n)$.

Next, for $k \geq 3$: in the $i$'th iteration of the **for** loop (lines 4 to 16), we find the subgraph $G_{k-1}$ induced by the neighbours of the vertex with the current highest degree $v_i$ in $O\big(d_k(v_i) + \sum_{u \in N(v_i)} d_k(u)\big)$ time (line 5). Line 13

is a recursive call that requires $T\big(k-1, (\sum_{u\in N(v_i)} d_k(u))/2, d_k(v_i)\big)$ time to find and store all complete subgraphs containing $v_i$. All updates of $min_{v_i}$ and $max_{v_i}$ values with respect to the timestamps of edges connecting $v_i$ with $(\ell-k)$ vertices stored in stack $S$, is again upper bounded by the time required to compute $G_{k-1}$ in line 5. All stack operations (lines 12 and 14) require constant time.

Thus, the total time required for each $v_i$ is,

$$O\Big(d_k(v_i) + \sum_{u\in N(v_i)} d_k(u)\Big) + O(1) + T\Big(k-1, (\sum_{u\in N(v_i)} d_k(u))/2, d_k(v_i)\Big)$$

which is the same as was obtained in [13]. Then following the analysis of [13, Theorem 3] we obtain $T(k,m,n) = O(a(G_k)^{\ell-2}m+n)$ with $k \geq 3$. Note that we do not explicitly report the complete subgraphs. Therefore, when $k = \ell$ finding all $K_\ell$'s in $G$ requires at most $O(a(G)^{\ell-2}m)$ time.

Points in $P = p_1, \ldots, p_{\mathcal{K}}$ can be stored in a window sensitive dominance data structure $\mathcal{D}$ of linear size, see [4]. Given a query interval $q = [i,j]$ we query $\mathcal{D}$ using $(-\infty, j] \times [i, +\infty)$. Since $high > low$ for every point $(high, low) \in P$, all points of $P$ lie below the main diagonal, (see Figure 3(b)). This condition is required by the window sensitive dominance structure $\mathcal{D}$. Thus $\mathcal{D}$ can report the number of points dominated by our query in $O(\log \mathcal{W}/\log\log \mathcal{K})$ time, where $\mathcal{W} = j - i + 1$ is the width of the query interval and $\mathcal{K}$ is the total number of complete subgraphs of order $\ell$. □

**Algorithm 1:** PreprocessCS$(\ell, G)$

**Input :** Relational event graph $G$ and the order of complete subgraph $\ell$.

**Output:** A set of points $P$ in $\mathbb{R}^2$.

1 Set stack $S \leftarrow \emptyset$.
2 Set $mt \leftarrow$ maximum timestamp in $G$.
3 Set min $\leftarrow mt + 1$, and max $\leftarrow -1$.
4 Let $G_\ell = G$.
5 Call CS$(\ell, G_\ell)$.

## 5.3 Counting All Complete Subgraphs of Orders $\ell \geq 3$

We extend our algorithm for computing all complete subgraphs of orders between 3 to $\ell$. That is, if we are given an order $\ell$, we can find all complete subgraphs $K_k$ in $G_{i,j}$, where $3 \leq k \leq \ell$. For each order $3 \leq k \leq \ell$, the previous algorithm of a fixed order $k$ can be run once in total $O\big(m\sum_{k=3}^{\ell} a(G)^{k-2}\big) = O(a(G)^{\ell-1}m)$ time to obtain corresponding timespans $[low, high]$ of all complete subgraphs. To represent complete subgraphs of all orders, we modify our algorithm so that, for each complete subgraph of order $k$, we store a point

**Algorithm 2:** CS$(k, G_k)$

**Input :** Graph slice $G_k$ and parameter $k$.
**Output:** A set of points $P$ in $\mathbb{R}^2$.

**if** $k > 2$ **then**
  Set $j \leftarrow$ order of $G_k$.
  Sort vertices $v_1, v_2, \ldots, v_j$ in non-increasing order of degrees. Without loss of generality, let $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_j)$.
  **for** $i = 1$ *to* $j$ **do**
    Let $G_{k-1} \subseteq G_k$ be the subgraph induced by the neighbours of $v_i$.
    **if** *stack* $S \neq NIL$ **then**
      Update $\min_{v_i}$ and $\max_{v_i}$ with respect to $t(v_i, z)$ where $z \in S$.
      **else**
        Set $(\min_{v_i} \leftarrow \min)$ and $(\max_{v_i} \leftarrow \max)$.
      **end**
    **end**
    Add $(v_i, \min_{v_i}, \max_{v_i})$ to stack $S$.
    Call CS$(k-1, G_{k-1})$.
    Delete $(v_i, \min_{v_i}, \max_{v_i})$ from stack $S$.
    Delete $v_i$ from $G_k$ and without loss of generality, let $G_k$ be the resulting graph.
  **end**
**end**
**else**
  **if** $k = 2$ **then**
    **for** *each edge* $(x, y)$ *of* $G_2$ **do**
      Set $(low \leftarrow \min_{v_i})$ and $(high \leftarrow \max_{v_i})$.
      Update $low$ and $high$ with $t(x, y)$.
      Update $low$ and $high$ with respect to $t(x, z)$ and $t(y, z)$ where $z \in S$.
    **end**
  **end**
**end**

$p = (a, b, k)$ in $\mathbb{R}^3$ where $a = high$, $b = low$ and $k =$ order in any standard dominance counting data structure, for example see [1]. This results in the following theorem.

**Theorem 11** *Given an RE graph $G$ with $m$ edges and an integer $\ell \geq 3$ , the problem of determining the total number of complete subgraphs $K_k$ of all orders $3 \leq k \leq \ell$ in the query interval $[i, j]$ can be reduced to dominance counting in $O(a(G)^{\ell-1}m)$ time using $O(m + n + \mathcal{K} \log \mathcal{K} / \log\log \mathcal{K})$ space, where $\mathcal{K}$ is the total number of complete subgraphs in $G$. The query takes $O((\log \mathcal{K} / \log\log \mathcal{K})^2)$ time to report the total number of complete subgraphs in $G_{i,j}$.*

**Remarks.** We can also reduce the problem of finding all 2-paths and 3-paths in $G_{i,j}$ to dominance counting as follows. For each vertex $u \in V$, we find its neighbours $v \in N(u)$, and successively find all neighbours of $v$, denoted as $w \in N(v)$. For each 2-path starting at $u$, that is $\langle uv, vw\rangle$, we store a point $(high, low)$ in $\mathbb{R}^2$, where $high$ and $low$ are respectively the maximum and minimum timestamps between $t(u, v)$ and $t(v, w)$. During this search, we will also find a path $\langle wv, vu\rangle$ starting from vertex $w$ while exploring neighbours of $w$, which is the same path as $\langle uv, vw\rangle$. Since each 2-path will be identified exactly twice during the preprocessing, we keep only one point per 2-path to avoid duplication. The time taken to find all 2-paths in $G$ will be upper bounded by $a(G)m$, which is the time needed to identify all triangles in $G$ (see Theorem 10). Now, we can count the number of 2-paths in $G_{i,j}$ using our dominance counting structure.

Let $G = (V = \{A \cup B\}, E = \{(u, v) : u \in A, v \in B\})$ be a bipartite RE graph. A complete bipartite graph can have at most $O(n^4)$ 3-paths. Each vertex $u \in A$ in a 3-path shares edges with two neighbours $v_1$ and $v_2$ such that $v_1, v_2 \in B$, and vice versa. Thus, reducing the problem of counting all 3-paths in $G$ to the problem of dominance counting requires an exhaustive search for all alternating edge adjacency between the vertices of $A$ and the vertices of $B$. To find a 3-path $\langle u_1, v_1, u_2, v_2\rangle$, we start from each vertex $u_1 \in A$, and find its neighbour $v_1 \in B$. Successively, we find $v_1$'s neighbour $u_2 \in A$, and $u_2$'s neighbour $v_2 \in B$. For every new edge added to the path, we update $(high, low)$ so that each 3-path can be stored as a point in $\mathbb{R}^2$. Thus, preprocessing all 3-paths in $G$ requires $O(n^4)$ time in the worst case.

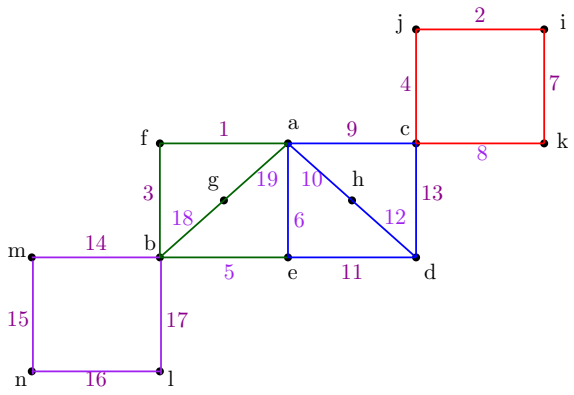
**Corollary 3** *Given an RE graph $G$ with $m$ edges, the problems of counting 2-paths can be reduced to window sensitive dominance counting in $O(a(G)m)$ time. Given a bipartite RE graph $G = (V = A \cup B, E)$ with $n$ vertices, the problems of counting 3-paths can be reduced to window sensitive dominance counting in $O(n^4)$ time. Each query can be answered in $O(\log \mathcal{W} / \log\log \mathcal{K})$ time, where $\mathcal{W}$ is the width of the query window and $\mathcal{K}$ is the number of paths in and $G$.*

## 5.4 Counting Quadrangles

In this section we present an algorithm for counting quadrangles, i.e., cycles of length 4, in an RE graph $G$. An edge searching algorithm is presented in [13] where a set of quadrangles can be implicitly represented by a tuple $(y, z, \{a_1, a_2, a_3, ...\})$ in $O(a(G)m)$ time and space, where $y$ and $z$ are vertices on two opposite sides of all quadrangles of this set and each vertex $v \in \{a_1, a_2, a_3, ..\}$ shares edges with both $y$ and $z$. Within this setting, any two vertices from $\{a_1, a_2, a_3, ..\}$ together with $y$ and $z$ represent a quadrangle.

Figure 9(c) illustrates a relational event graph with eight quadrangles. The search algorithm represents these quadrangles using four tuples, $(a, b, \{e, f, g\})$, $(a, d, \{c, e, h\})$, $(b, n, \{m, \ell\})$ and $(e, i, \{j, k\})$. We can see that the first tuple $(a, b, \{e, f, g\})$ contains three quadrangles $(a, e, b, f)$, $(a, f, b, g)$ and $(a, e, b, g)$.

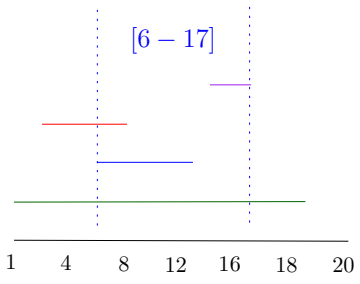
We can use the dominance counting data structure to count the total number of quadrangles using a similar algorithm presented in Section 5.2 by explicitly



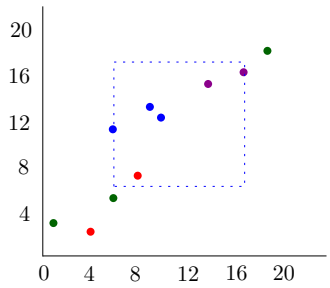

(a)

| Tuple | $(a, b, \{e, f, g\})$ | $(a, d, \{c, e, h\})$ | $(b, n, \{m, l\})$ | $(e, i, \{j, k\})$ |
|---|---|---|---|---|
| Interval $(x_i)$ | $[1 - 19]$ | $[6 - 13]$ | $[14 - 17]$ | $[2 - 8]$ |
| Pointset $(P)$ | $(1, 3)$ | $(9, 13)$ | $(14, 15)$ | $(4, 2)$ |
| | $(6, 5)$ | $(8, 12)$ | $(17, 16)$ | $(8, 7)$ |
| | $(19, 18)$ | $(6, 11)$ | | |

(b)




(c)




(d)

Figure 9: Counting quadrangles in an RE graph. The integer numbers on the graphs edges are their timestamps. (a) Quadrangles in $G$ are highlighted. (b) Tuple representation for each of the quadrangles, and their corresponding time intervals and point sets in $\mathbb{R}^2$. (c) Querying the interval tree for the valid intervals in $q = [6 - 17]$. (d) Querying the range tree for the valid quadrangles.

representing all quadrangles and computing the corresponding *high* and *low* values for each quadrangle. This requires $O(n^2)$ time and space as there can potentially be $O(n)$ vertices in a tuple $(y, z, \{a_1, a_2, a_3, ..., a_{n-2}\})$ such that each vertex in $\{a_1, a_2, a_3, .., a_{n-2}\}$ shares edges with both $y$ and $z$. To improve the overall space requirement we now present an output sensitive range searching data structure for counting quadrangles in $G$. We still use the tuple representation of quadrangles discussed above and modify the algorithm as follows (see Algorithm 3).

We need the following additional data structures.

- For each tuple $(y, z, \{a_1, a_2, a_3, ...\})$, we use two linked lists to store edges adjacent to $y$ and $z$, respectively, i.e., $E = \{(y, a_1), (y, a_2), (y, a_3), ..\}$ and $E' = \{(z, a_1), (z, a_2), (z, a_3), ..\}$. Each node in $E$ points to a node in $E'$ that contains the edge having one common endpoint, i.e., $(y, a_1)$ points to $(z, a_1)$, $(y, a_2)$ points to $(z, a_2)$, and so on.

- For each tuple, we use a 2-dimensional range tree to store a set of points $\{(t(e_1), t(e'_1)), (t(e_2), t(e'_2)), ..., (t(e_p), t(e'_p))\}$, where $t(e_i)$ is the timestamp of the edge $e_i$, and $e_i \in E$, $e'_i \in E'$.

- We store the timespan of each tuple using an interval $x$ of the form $[min(t(e_1), t(e'_1)), max(t(e_p), t(e'_p))]$. For the given graph $G$, we use an interval tree to store the set of horizontal intervals $I = \{x_1, x_2, \cdots\}$, where each interval represents the timespan of a tuple.

The quadrangle counting algorithm consists of a *preprocessing* step (Algorithm 3) and a *query* step (Algorithm 4). We describe these steps below.

***Preprocessing* Step:** The preprocessing step takes an RE graph $G$ consisting of $n$ vertices and $m$ edges as input. $G$ is processed by starting with the vertex having the highest degree. So, the vertices are first sorted in non-increasing order of their degrees and without loss of generality, let $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$. Following our search process, once all quadrangles containing $v_i$ are identified correctly, where $1 \leq i \leq n$, $v_i$ is deleted from $G$ to avoid duplication and the loop continues with the next vertex in the sequence. We first describe two major components of this procedure.

*a) Finding Quadrangles:* For each vertex $y \in V$, we apply the following technique to find all quadrangles containing $y$. For each vertex $z \in V$ at distance 2 from $y$, we find all the vertices that are adjacent to both $y$ and $z$. We store these vertices in a set $U[z]$. Thus, the tuple $(y, z, U[z])$ represents the set of quadrangles, where every quadrangle has vertices $y$ and $z$ as two opposite corner points.

*b) Finding Intervals:* Recall that we want to count the number of quadrangles within a query time slice $[i, j]$, but all we have is the representation of a set

**Algorithm 3:** PreprocessQuad($G$)

**Input :** A relational event graph $G = (V, E)$.
**Output:** 2D range trees for each tuple of $G$ and an interval tree for $G$.

Sort the vertices according to their degrees in non-increasing order. Without loss of generality, let $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$.
Create an empty interval tree $I = \emptyset$.
**for** *each vertex* $v \in V$ **do**
  Set $U[v] \leftarrow \emptyset$.
**end**
**for** *each vertex* $v_i$ **do**
  Set $y \leftarrow v_i$.
  **for** *each vertex* $u$ *adjacent to* $y$ **do**
    **for** *each vertex* $z \neq y$ *adjacent to* $u$ **do**
      Set $U[z] \leftarrow U[z] \cup \{u\}$.
    **end**
    **for** *each vertex* $z$ *with* $|U[z]| \geq 2$ **do**
      Let $U[z]$ stores vertices $\{a_1, a_2, \cdots, a_p\}$.
      Store $(y, z, \{a_1, a_2, \cdots, a_p\})$ using linked lists $E = \{(y, a_1), (y, a_2), \cdots, (y, a_p)\}$ and $E' = \{(z, a_1), (z, a_2), \cdots, (z, a_p)\}$.
      Let $e_1 = (y, a_1), e_2 = (y, a_2), \cdots, e_p = (y, a_p)$.
      Let $e'_1 = (z, a_1), e'_2 = (z, a_2), \cdots, e'_p = (z, a_p)$.
      Create a 2D range tree with point set $P = \{(t(e_1), t(e'_1)), (t(e_2), t(e'_2)), \cdots, (t(e_p), t(e'_p))\}$.
      Sort $E = (e_1, e_2, \cdots, e_p)$ and $E' = (e'_1, e'_2, \cdots, e'_p)$ according to increasing order of timestamps. Without loss of generality, let $t(e_1) < t(e_2) < \cdots < t(e_p)$ and $t(e'_1) < t(e'_2) < \cdots < t(e'_p)$.
      Create an interval segment $x = [\min(t(e_1), t(e'_1)), \max(t(e_p), t(e'_p))]$.
      Insert $x$ into interval tree $\mathcal{T}$.
    **end**
    **for** *each vertex* $z$ *with* $U[z] \neq \emptyset$ **do**
      Set $U[z] \leftarrow \emptyset$.
    **end**
  **end**
  Delete $y$ from $G$ and let $G$ be the resulting graph.
**end**

**Algorithm 4:** QueryQuad($\mathcal{T}, [i, j]$)

**Input :** Interval tree $\mathcal{T}$, query interval $[i, j]$.
**Output:** Total number of quadrangles $\#Quad$ within the given query interval.

```
Query T using [i, j] horizontal segment and find interval set I' within
  the query range.
Set #Quad ← 0.
for each segment x ∈ I' do
    Query 2D range tree with query rectangle [i, i]x[j, j] to find valid set
      S of edges.
    if |S| ≥ 2 then
        Set #Quad ← (|S| choose 2) + #Quad.
    end
end
```

of quadrangles as tuples. So, for each such tuple we mark its timespan and maintain some geometric data structures so that we can answer the query.

First, we store each tuple $(y, z, U[z] = \{a_1, a_2, a_3, \ldots\})$ using two linked lists $E = \{(y, a_1), (y, a_2), (y, a_3), ..\}$ and $E' = \{(z, a_1), (z, a_2), (z, a_3), ..\}$ according to the order of the vertices in $\{a_1, a_2, a_3, ..\}$. We also represent each tuple with a point set $P = \{(t(y, a_1), t(z, a_1)), (t(y, a_2), t(z, a_2)), ..\}$, where $t(y, a_i)$ is the timestamp of the edge $(y, a_i)$ for all $i = 1, 2, ..$ . We store $P$ using a 2-dimensional range tree.

Next, we compute the timespan of each tuple and store it as an interval using an interval tree. We sort linked lists $E$ and $E'$ according to the timestamps of their edges in non-decreasing order. Without loss of generality, let $E = (e_1, e_2, \ldots, e_p)$ and $E' = (e'_1, e'_2, \ldots, e'_p)$, where $t(e_1) \leq t(e_2) \leq \ldots \leq t(e_p)$ and $t(e'_1) \leq t(e'_2) \leq \ldots \leq t(e'_p)$. Now we can create an interval segment $x = [\min(t(e_1), t(e'_1)), \max(t(e_p), t(e'_p))]$ for each tuple to mark its timespan. Following this technique, we compute segments for all tuples and store them in an interval tree $\mathcal{T}$.

For an illustration, see the example presented in Figure 9(a). It shows an RE graph $G$ with four tuples, each tuple is highlighted with a different color. We first store the tuple $(a, b, \{e, f, g\})$ using linked lists $E = (ae, af, ag)$ and $E' = (be, bf, bg)$. Then we create a point set $P = \{(6, 5), (1, 3), (19, 18)\}$ and store it in a two-dimensional range tree. Next, we sort edges in $E = (af, ae, ag)$ and $E' = (bf, be, bg)$ according to their timestamps in increasing order. Thus, we store an interval segment $[\min\{1, 3\}, \max\{19, 18\}] = [1, 19]$ for this tuple in the interval tree. Interval segments and point sets for all tuples are shown in Figure 9(b).

***Query* Step:** We query the interval tree $\mathcal{T}$ with a horizontal query segment $[i, j]$ and obtain a set of valid interval segments $I' \subseteq I$. We consider each segment $x_i \in I'$ valid, if it intersects with the query slice indicating that $x_i$ might contain

quadrangles that exist within the query slice. To obtain the exact number of quadrangles ($\#Quad$), we need to know which of these valid segments contain at least two sets of paired edges $\{(y, v'), (z, v')\}$ and $\{(y, v''), (z, v'')\}$ such that it makes a quadrangle $(y, v', z, v'')$. Recall that for each segment we have already stored a set of points $P$ that contain information of these paired edges. So for each valid segment, we perform a 2-D rectangular range query on $P$ using a query rectangle with four corner points $(i, i), (i, j), (j, j)$ and $(j, i)$. It returns a set of edges $S$. If $|S| \geq 2$ we add $\binom{|S|}{2}$ to $\#Quad$.

Continuing with the same example, Figure 9(c) shows all the valid segments after querying an interval tree with query point $q = [6, 17]$. Finally a rectangular query shows that tuples $(a, d, \{c, e, h\})$ and $(b, n, \{m, \ell\})$ have sets of edges that fall within the query range (Figure 9(d)). Therefore, we obtain $\binom{3}{2} + \binom{2}{2} = 4$ quadrangles ($\#Quad$) within the query time interval [6,17].

**Preprocessing Analysis.** In lines 1-5, sorting vertices with respect to their degrees and setting up the lists for all vertices take $O(m+n)$ time. The outer **for** loop (starting from line 6) identifies all quadrangles containing $v_i$ by traversing its neighbours in $O(\sum_{v_i \in V} O(d(v_i) + \sum_{u \in N(v_i)} d(u))$ time, where $N(v_i)$ is the set of neighbours of $v_i$. So, except for the time needed to construct the range trees and the interval tree, total time required is $O(m + n) + \sum_{v_i \in V} O(d(v_i) + \sum_{u \in N(v_i)} d(u))$. This can be bounded by $O(a(G)m)$ by Lemma 1.

Since we can have at most $a(G)m$ tuples in $G$, there can be at most $a(G)m$ intervals to store in the interval tree. This is a weak upper bound on the number of intervals. For example, for any complete graph $a(G) = \lceil n/2 \rceil$, whereas for planar graphs $a(G) = O(1)$. Since, each interval considers unique pair of vertices on opposite sides of a quadrangle, there can be at most $\binom{n}{2}$ quadrangles in $G$. So the number of intervals can be bounded as $\alpha = \min\{a(G)m, \binom{n}{2}\} \leq a(G)m$. We maintain 2-dimensional range trees for every tuple identified in $G$ (line 16). We observe that, the total number of points that can be stored in these range trees can be $O(\sum_{v_i \in V} O(d(v_i) + \sum_{u \in N(v_i)} d(u))) = O(a(G)m)$. Therefore, by Theorem 3, total time required to build these range trees is $O(\alpha \log \alpha + a(G)m \log(a(G)m)) = O(a(G)m \log n)$.

Using similar reasoning, in lines 18-19, the interval tree for $\alpha$ intervals can be created in time, $O(\alpha \log \alpha) = O(\alpha \log n)$ (by Theorem 4). Therefore, the total time required by Algorithm 3 is $O\big(a(G)m + a(G)m \log n + \alpha \log n\big) = O(a(G)m \log n)$.

**Query Analysis.** Suppose given a query time interval $q = [i, j]$, we find $I'$ as the set of all valid interval segments that intersects with $q$ (line 1 of Algorithm 4). Let $\gamma = |I'|$, where $\gamma \leq \min\{\binom{n}{2}, a(G)m\}$. Reporting all $\gamma$ segments from the interval tree requires $O(\log(\min\{\binom{n}{2}, a(G)m\}) + \gamma) = O(\log n + \gamma)$ time [14]. Rectangular range queries on $\gamma$ range trees (line 3-8) take time $O\big(\sum_{i=1}^{\gamma} \log n + k_i\big) = O(\gamma \log n + w)$, where $k_i$ is the number of reported quadrangles in segment $i$ and $w$ is $\#Quad$ in $G_{i,j}$.

**Space:** Total number of intervals in our problem is bounded by $O(a(G)m)$. By Theorem 4, the interval tree uses $O(a(G)m)$ space. Therefore, total required space is dominated by the space required by range trees, i.e., $O(a(G)m \log(a(G)m)) = O(a(G)m \log n)$ (Theorem 3). The following theorem summarizes the results for quadrangle counting in a relational event graph.

**Theorem 12** *Given an RE graph $G$ with $m$ edges, the number of quadrangles in the query time slice $[i, j]$ can be determined in $O(\gamma \log n + w)$ time, where $\gamma \leq \min\{\binom{n}{2}, a(G)m\}$ and $w$ is the number of reported quadrangles. Preprocessing takes $O(a(G)m \log n)$ time and $O(a(G)m \log n)$ space.*

# 6 Applications

We now discuss some of the applications of these data structures. The problem of finding whether an MST exists in a queried graph slice can be directly applied in the design of any type of connected networks in a query time interval, such as, telecommunication, transportation, computer networks or electrical grids. The problem of graph edit distance for spanning forests provides a cost effective measure (i.e., the number of operations required) to maintain the connectivity within the connected components of a network. We find that the applications of subgraph counting data structures are specifically pertinent to some useful social network analyses. So we further elaborate them below.

## 6.1 Clustering Coefficient

*Clustering coefficient* is also known as network transitivity and is an important model for extracting community structure from social networks [27]. Multiple definitions are available for clustering coefficient depending on the context in which it is being used and the type of network is being studied. In this paper we define the clustering coefficient of a graph $G$ as the measure of the degree to which vertices in $G$ tend to cluster together [28]. It is formulated as follows,

$$CC(G) = \frac{\text{total number of closed triplets}}{\text{total number of triplets}} = \frac{3 \times \text{total number of triangles}}{\text{total number of 2-paths}}$$

Since clustering coefficient of any graph is the ratio of total number of closed triplets over all the open and closed triplets, its value ranges between 0 and 1. For an illustration, see Figure 1. The clustering coefficient of $G_{1,5}$ in Figure 1 (a) is $CC(G_{1,5}) = \frac{3\times1}{7} = 0.43$. Similarly, $CC(G_{2,4}) = \frac{3\times1}{3} = 1$ and $CC(G_{2,5}) = \frac{3\times1}{5} = 0.6$. For bipartite graphs, the clustering coefficient is based on the number of quadrangles ($C_4$). More formally, the clustering coefficient of a bipartite graph is the ratio of the number of quadrangles to the number of 3-paths [28], i.e.,

$$CCB(G) = \frac{4 \times \text{total number of quadrangles } (C_4)}{\text{total number of 3-paths}}$$

Thus, for general graphs we obtain the following results.

**Corollary 4** *Let $G$ be an RE graph consisting of $m$ edges. Let $\mathcal{K}$ be the total number of 2-paths in $G$. The problem of computing the clustering coefficient of $G_{i,j}$ can be reduced to dominance counting in $(a(G)m)$ time using $O(m+n+\mathcal{K})$ space and each query can be answered in $O(\log \mathcal{W}/\log\log \mathcal{K})$ time, where $\mathcal{W}$ is the width of the query window.*

For clustering coefficient of a bipartite graph slice $G_{i,j}$, we can count the total number of 3-paths in $O(m+n)$ time and the number of quadrangles in $O(\gamma \log n + w)$ time, where $w$ is the number of total quadrangles in $G_{i,j}$ and $\gamma \leq \min\{\binom{n}{2}, (a(G)m)\}$. Note that the total preprocessing time is dominated by the quadrangle processing time. The following corollary summarizes the result.

**Corollary 5** *Let $G = (V, E)$ be a bipartite RE graph with $n$ vertices and $m$ edges. The graph $G$ can be preprocessed in $O(a(G)m \log n)$ time and the clustering coefficient of a bipartite graph slice $G_{i,j}$ can be computed in $O(\gamma \log n + w)$ time, where $w$ is the total number of quadrangles in $G_{i,j}$ and $\gamma \leq \min\{\binom{n}{2}, (a(G)m)\}$.*

## 6.2 Embeddedness and Neighborhood Overlapping

*Embeddedness* of an edge $(u, v)$, denoted as $emb(u, v)$, in a graph $G$ is the number of common neighbors the two endpoints $u$ and $v$ have, i.e., $emb(u, v) = |N(u) \cap N(v)|$[15]. Embeddedness of an edge $(u, v)$ in a graph represents the trustworthiness of its neighbors, and the confidence level in the integrity of the transactions that take place between two vertices $u$ and $v$. Note that the embeddedness of an edge $(u, v)$ represents the number of triangles that share $(u, v)$.

The value of embeddedness is also used to compute the *neighborhood overlap* of an edge $(u, v)$, denoted as $NOver(u, v)$, and defined as the ratio of the number of vertices who are neighbors of both $u$ and $v$, and the number of vertices who are neighbors of only one of them [15].

$$NOver(u, v) = \frac{emb(u, v)}{|N(u) \cup N(v)| - emb(u, v) - 2}$$

Neighborhood overlap of an edge represents the strength (in terms of connectivity) of that edge in its neighborhood. The neighborhood overlap of an entire graph $G$ is defined as the average of the neighborhood overlap values of all the edges of $G$, i.e., $NOver(G) = \frac{1}{|E|} \sum_{k=1}^{|E|} NOver(e_k)$ [26].

The result for computing the neighborhood overlap of a query graph slice has been presented in [10] and is also stated below. These results use the time windowed data structures presented in this paper and also include colored range searching data structures [6, 16, 17].

**Theorem 13** *(Theorem 6 in [10]) Given an RE graph $G = (V, E)$ the problem of computing the average neighborhood overlap of $G_{i,j}$ can be reduced to the colored range counting in $O(mn)$ time. For a query time slice $[i, j]$, $NOver(G_{i,j})$*

*can be computed in* $O(\log^2 n + (t+s)\log n)$ *time, where* $t$ *is the number of edges in* $G_{i,j}$ *with positive embeddedness and* $s$ *is the number of edges having some neighboring edges in* $E_{i,j}$*.*

# 7 Conclusion

In this paper, we present time window data structures to answer various queries involving both decision and reporting problems based on relational event graphs. In the first part of the paper, we provide dynamic algorithm based data structures that can answer time windowed decision problems under monotone graph properties, such as disconnectedness and bipartiteness, and can report the weight of a minimum spanning tree, the minimum spanning interval and the graph edit distance for obtaining a spanning forest for a query graph slice. In the second part of the paper, we present window data structures for counting subgraphs of a given pattern. We consider 2-paths (for general graphs), 3-paths (for bipartite graphs), complete subgraphs of order $\ell \geq 3$ and quadrangles as valid patterns. We also show some applications of our subgraph counting results for computing graph parameters that are important for social network analysis, such as clustering coefficients, embeddedness and neighborhood overlapping.

**Acknowledgements.** We would like to thank the reviewers sincerely for their insightful comments.

## References

[1] Algorithms and computation, 15th international symposium, ISAAC 2004, hong kong, china, december 20-22, 2004, proceedings. volume 3341 of *Lecture Notes in Computer Science*. Springer, 2004. doi:10.1007/b104582.

[2] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi:10.1007/BF02523189.

[3] M. J. Bannister, W. E. Devanny, M. T. Goodrich, J. A. Simons, and L. Trott. Windows into geometric events: Data structures for time-windowed querying of temporal point sets. In *Proceedings of the 26th Canadian Conference on Computational Geometry, CCCG 2014, Halifax, Nova Scotia, Canada, 2014*, 2014. URL: http://www.cccg.ca/proceedings/2014/papers/paper02.pdf.

[4] M. J. Bannister, C. DuBois, D. Eppstein, and P. Smyth. Windows into relational events: Data structures for contiguous subsequences of edges. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 856–864, 2013. doi:10.1137/1.9781611973105.61.

[5] D. Bokal, S. Cabello, and D. Eppstein. Finding all maximal subsequences with hereditary properties. In *31st International Symposium on Computational Geometry, SoCG 2015, June 22-25, 2015, Eindhoven, The Netherlands*, volume 34 of *LIPIcs*, pages 240–254. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.SOCG.2015.240.

[6] P. Bozanis, N. Kitsios, C. Makris, and A. K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*, pages 464–474. 1995. doi:10.1007/3-540-60084-1_97.

[7] U. Brandes, J. Lerner, and T. A. B. Snijders. Networks evolving step by step: Statistical analysis of dyadic event data. In *2009 International Conference on Advances in Social Network Analysis and Mining, ASONAM 2009, 20-22 July 2009, Athens, Greece*, pages 200–205, 2009. doi:10.1109/ASONAM.2009.28.

[8] T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the ram, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10, 2011. doi:10.1145/1998196.1998198.

[9] T. M. Chan and S. Pratt. Two approaches to building time-windowed geometric data structures. In *32nd International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, volume 51 of *LIPIcs*, pages 28:1–28:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.SoCG.2016.28.

[10] F. Chanchary, A. Maheshwari, and M. H. M. Smid. Querying relational event graphs using colored range searching data structures. In *Algorithms and Discrete Applied Mathematics - Third International Conference, CALDAM 2017, Sancoale, Goa, India, February 16-18, 2017, Proceedings*, volume 10156 of *Lecture Notes in Computer Science*, pages 83–95. Springer, 2017. doi:10.1007/978-3-319-53007-9_8.

[11] F. Chanchary, A. Maheshwari, and M. H. M. Smid. Window queries for problems on intersecting objects and maximal points. In *Algorithms and Discrete Applied Mathematics - 4th International Conference, CALDAM 2018, Guwahati, India, February 15-17, 2018, Proceedings*, volume 10743 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2018. doi:10.1007/978-3-319-74180-2_17.

[12] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440.

[13] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985. doi:10.1137/0214017.

[14] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. URL: http://www.worldcat.org/oclc/227584184.

[15] D. A. Easley and J. M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge University Press, 2010. URL: http://www.cambridge.org/gb/knowledge/isbn/item2705443/?site_locale=en_GB.

[16] P. Gupta, R. Janardan, S. Rahul, and M. Smid. Computational geometry: Generalized (or colored) intersection searching. *Handbook of Data Structures and Applications, 2nd Edition, (Dinesh Mehta and Sartaj Sahni, editors), CRC Press, Boca Raton, Chapter 67*, pages 1042–1057, 2018.

[17] P. Gupta, R. Janardan, and M. H. M. Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *J. Algorithms*, 19(2):282–317, 1995. doi:10.1006/jagm.1995.1038.

[18] F. Harary. *Graph theory*. Addison-Wesley, 1991.

[19] F. Harary and H. J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 6(2):199–210, 1979.

[20] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

[21] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

[22] P. Holme. Network reachability of real-world contact sequences. *Physical Review E*, 71(4):046119, 2005.

[23] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978. doi:10.1137/0207033.

[24] T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. *Inf. Process. Lett.*, 74(3-4):115–121, 2000. doi:10.1016/S0020-0190(00)00047-8.

[25] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985. doi:10.1137/0214021.

[26] N. Meghanathan. A greedy algorithm for neighborhood overlap-based community detection. *Algorithms*, 9(1):8, 2016. doi:10.3390/a9010008.

[27] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003. doi:10.1137/S003614450342480.

[28] T. Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks*, 35(2):159–167, 2013. doi:10.1016/j.socnet.2011.07.001.

[29] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

[30] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976. doi:10.1145/321921.321925.

[31] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979. doi:10.1137/0208032.

[32] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442, 1998.

[33] P. Zhang, J. Wang, X. Li, M. Li, Z. Di, and Y. Fan. Clustering coefficient and community structure of bipartite networks. *Physica A: Statistical Mechanics and its Applications*, 387(27):6869–6875, 2008.