Efficient Processing of Growing Temporal Graphs

Huanhuan Wu, Yunjian Zhao, James Cheng, Da Yan

Department of Computer Science and Engineering, The Chinese University of Hong Kong

{hhwu,yjzhao,jcheng,yanda}@cse.cuhk.edu.hk

***Abstract*—Temporal graphs are useful in modeling real-world networks. For example, in a phone call network, people may communicate with each other in multiple time periods, which can be modeled as multiple temporal edges. However, the size of real-world temporal graphs keeps increasing rapidly (e.g., considering the number of phone calls recorded each day), which makes it difficult to efficiently store and analyze the complete temporal graphs. In this paper, we propose a new model, called *equal-weight damped time window model*, to efficiently manage temporal graphs. In this model, each time window is assigned a unified weight. This model is flexible as it allows users to control the tradeoff between the required storage space and the information loss. It also supports efficient maintenance of the windows as new data come in. We discuss two applications using the model for analyzing temporal graphs and propose efficient algorithms for these applications. Our experiments demonstrated that we can handle massive temporal graphs efficiently with limited space requirement.**

I. Introduction

A temporal graph is a graph in which the relationship between vertices is not just modeled by an edge between them, but the time period when the relationship happens is also recorded. For example, the records that two persons A and B talked on the phone in time periods $[t_1, t_2]$ and $[t_3, t_4]$ are modeled as two temporal edges, $(A, B, [t_1, t_2])$ and $(A, B, [t_3, t_4])$. An example of a temporal graph is shown in Figure 1(a).

Graphs are used ubiquitously to model relationships between objects in real world. However, the graph data in many applications are actually better to be modeled as temporal graphs. For example, in communication networks, including online social networks, messaging networks, phone call networks, etc., people communicate with each other in different time periods. Temporal graphs collected from these applications carry rich time information, and have been shown to possess many important time-related patterns that cannot be found from non-temporal graphs [1], [2], [3], [4], [5], [6], [7].

However, existing work overlooks one serious problem presented by temporal graphs in real world applications, that is, the number of temporal edges (or temporal records) can be extremely huge so that it becomes overly expensive to store and process a temporal graph. For example, in a temporal graph that models phone-call records, a person may talk on the phone many times in different time periods in a day, where each phone call is represented by a temporal edge with the corresponding time period. The total number of temporal edges accumulated over time for all persons can easily become overwhelming. Note that while the number of temporal edges usually increases at a steady rate over time, the number of vertices, on the other hand, does not increase too much over time after passing the growth stage.

The problem in the above example is actually a real problem presented to us by a telecommunications operator, who collects phone-call and messaging records represented as a temporal graph that becomes too large over time for them to manage (millions to tens of millions of new temporal edges added each day). While analyzing only a short recent window of the data is useful, the telecom operator is also very keen in storing and analyzing the temporal graph over a long period of time (e.g., in recent years), and possibly the entire history, in an efficient way. Motivated by this, we propose a new model to efficiently manage a temporal graph.

Our new model considers the input temporal graph as a continuous stream, which captures how the temporal graph is collected in real-life applications (e.g., new call/messaging records are accumulated in the order of the calling/messaging time). Apparently, the sheer size of the stream over the entire time history will render the analysis (and even storage) of the original temporal graph impractical. To address this problem, we consider a *damped time window model* (also called *tilted time window*) [8], where a decay function is applied to depreciate the importance of records in an older window. However, the windows defined by existing damped time window models do not have a unified weight and hence the importance of records in different windows cannot be easily compared. For example, it is difficult to determine which of the following patterns is more important: a pattern that A and B communicated 10 times in a recent window (e.g., last week), or a pattern that A and B communicated 10,000 times in an older window (e.g., last year)?

We design a new damped time window model that gives a unified weight to each time window, called *equal-weight damped time window*, and represents the temporal graph falling into each window (i.e., a time period) as a weighted graph. The weighted graphs from different time windows can then be compared and analyzed.

The main contributions of our work are summarized as follows:

- Our equal-weight damped time window distributes a unified weight to each time window, which makes it easy to compare different time windows.
- Our model can handle massive temporal graphs with limited space requirement, and support efficient graph analysis with little information loss.
- The equal-weight design in our model also leads to natural and efficient update maintenance of the entire window (within a bounded storage space) as new data come in.
- We propose two applications of analyzing a temporal graph under this new model, including connectivity analysis and community finding. We devise efficient algorithms for solving these problems. Then we verified the effectiveness and efficiency of our method by extensive experiments on large temporal graph datasets.

The rest of the paper is organized as follows. Section II presents the equal-weight time window model. Section III discusses two applications based on equal-weight time window model. Section V reports experimental results. Section VI discusses related work. Section VII concludes the paper.

II. Equal-Weight Damped Time Window

Different window models have been proposed for processing a data stream. Among which, the *landmark window model* [9] considers the entire history of a stream without distinguishing the importance of recent and old records, while the *sliding window model* [10] focuses on the most recent window only. Our work is motivated by application needs from a telecom operator that requires to analyze historical data while giving more importance to recent data. For this purpose, the *damped time window model* [8] seems to suit the requirement. We introduce our damped time window model in this section, and discuss its difference with existing ones.

We first define the notations related to a temporal graph. Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a temporal graph, where $\mathbb{V}$ is the set of vertices and $\mathbb{E}$ is the set of edges in $\mathbb{G}$. An edge $e \in \mathbb{E}$ is a quadruple $(u, v, [t_i, t_j])$, where $u, v \in \mathbb{V}$, and $[t_i, t_j]$ is the time that e is *active*. The *duration* of the edge e is given by $t_j - t_i$. We focus our discussion on undirected temporal graphs, while we note that it is not difficult to extend our method to directed temporal graphs.

A. The Weight Function

In a damped time window model, a decaying weight function is used to depreciate the importance of a record over time. In the setting of a temporal graph, we use such a function to assign weight to temporal edges in the graph. We first present the weight density function as follows.

Definition 1 (Weight density function): Let t_τ be the current time. The weight density of a record at time t (with respect to t_τ) is defined as

$$f(t) = e^{\lambda(t - t_\tau)},$$

where $\lambda \geq 0$ is a decaying constant.

Note that $t \leq t_\tau$, and t is a time in the past if $t < t_\tau$.

In Definition 1, $f(t)$ is an exponentially decaying function, which is used in the discussion throughout the paper. In general, the class of exponentially decaying functions has been widely adopted [11], [12], as it fits most real application scenarios. But we note that $f(t)$ can also be defined differently (e.g., as a linear decaying function) depending on the application. Based on $f(t)$, we define our weight function as follows.

Definition 2 (Weight function): The weight of a temporal edge $(u, v, [t_1, t_2])$ is given as the integral

$$F(t_1, t_2) = \int_{t_1}^{t_2} f(t)dt.$$

Let $W = [t_x, t_y]$ be a given time window. With the weight function, we represent the part of a temporal graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ that falls into W as a weighted graph G_W defined as follows.

Definition 3 (Weighted graph): The weighted graph of a temporal graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ within a time window $W = [t_x, t_y]$ is given by $G_W = (V_W, E_W, \Pi_W)$, where:

- $V_W = \mathbb{V}$,
- $E_W = \{(u, v) : (u, v, [t_i, t_j]) \in \mathbb{E}_W\}$, where $\mathbb{E}_W = \{(u, v, [t_i, t_j]) \in \mathbb{E} : [t_i, t_j] \cap [t_x, t_y] \neq \emptyset\}$,
- Π_W is a function that assigns each edge $e = (u, v) \in E_W$ a weight $\Pi_W(e) = \sum_{(u,v,[t_i,t_j]) \in \mathbb{E}_W} F(\max(t_i, t_x), \min(t_j, t_y))$.

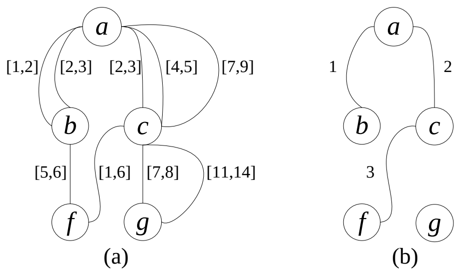


Figure 1. **(a)** a **temporal graph** $\mathbb{G}$, and **(b)** the **weighted graph** $G_{[2,5]}$

The following example shows a weighted graph.

Example 1: Figure 1(a) shows a temporal graph and Figure 1(b) shows the corresponding weighted graph $G_{[2,5]}$ within the time window $[2, 5]$. For simplicity, we assume that $\lambda = 0$ and hence $f(t) = 1$. Thus, the weight of edge (a, b) is $F(2, 3) = 1$, the weight of edge (a, c) is $F(2, 3) + F(4, 5) = 2$, and the weight of edge (c, f) is $F(2, 5) = 3$. The weight of all other edges is 0.

B. The Equal-Weight Window Model

Next, we determine the size of each window in a data stream given the weight function.

Existing damped time window model [8] usually sets the sizes of the windows in a stream by an exponentially increasing function (e.g., 2^0T, 2^1T, 2^2T, 2^3T, ..., where the windows are disjoint and the most recent window has a size 2^0T), or by the lengths of conventional time units (e.g., hour, day, month, year, ...). These window size settings may seem to be intuitive, but they are primarily designed for mining frequent itemsets from a stream and are not suitable for our problem of handling a temporal graph stream (see more discussion in "Advantages of the new model" at the end of this subsection). We introduce an equal-weight scheme as follows.

Let $[t_0, t_\tau]$ be the time period of the entire stream up to the current time t_τ. To limit the space requirement for handling a large temporal graph, we divide the stream into θ windows for a given constant number θ. We first define the equal-weight window condition as follows.

Definition 4 (Equal-weight window condition): Consider that the probability distribution of any edge being active at any time follows a uniform distribution. Under this distribution, the equal-weight condition is satisfied if the stream is divided into θ windows such that the weighted graph of each window is the same in expectation.

Intuitively, Definition 4 states that *if the probability of any edge being active does not change over time, then the weight of the edge should not change in any of the θ windows.*

Let the time periods of the θ windows be $[t_0, t_1]$, $[t_1, t_2]$,

..., $[t_{\theta-1}, t_\tau]$. Then, applying Definition 4, we have

$$\int_{t_0}^{t_1} f(t)dt = \int_{t_1}^{t_2} f(t)dt = ... = \int_{t_{\theta-1}}^{t_\tau} f(t)dt = \frac{1}{\theta}\int_{t_0}^{t_\tau} f(t)dt.$$

Take $f(t) = e^{\lambda(t-t_\tau)}$. We first determine t_1 as follows:

$$\int_{t_0}^{t_1} f(t)dt = \frac{1}{\theta}\int_{t_0}^{t_\tau} f(t)dt$$

$$\Rightarrow \frac{1}{\lambda}(e^{\lambda(t_1-t_\tau)} - e^{\lambda(t_0-t_\tau)}) = \frac{1}{\theta\lambda}(e^{\lambda(t_\tau-t_\tau)} - e^{\lambda(t_0-t_\tau)})$$

$$\Rightarrow \theta(e^{\lambda t_1} - e^{\lambda t_0}) = e^{\lambda t_\tau} - e^{\lambda t_0}$$

$$\Rightarrow e^{\lambda t_1} = \frac{e^{\lambda t_\tau} + (\theta-1)e^{\lambda t_0}}{\theta}$$

$$\Rightarrow t_1 = \frac{1}{\lambda}\ln\frac{e^{\lambda t_\tau} + (\theta-1)e^{\lambda t_0}}{\theta}.$$

Similarly, we obtain t_i, where $1 \le i \le \theta - 1$, as follows:

$$t_i = \frac{1}{\lambda}\ln\frac{i \times e^{\lambda t_\tau} + (\theta-i)e^{\lambda t_0}}{\theta}.$$

Based on the above analysis, we define *equal-weight damped time window model* as follows.

Definition 5 (Equal-weight damped time window model): Given a stream that spans the time period $[t_0, t_\tau]$, and an integer θ, *equal-weight damped time window model* divides the stream into θ windows spanning time periods $[t_0, t_1]$, $[t_1, t_2]$, ..., $[t_{\theta-1}, t_\tau]$, where $t_i = \frac{1}{\lambda}\ln\frac{i\times e^{\lambda t_\tau}+(\theta-i)e^{\lambda t_0}}{\theta}$, for $1 \le i \le \theta - 1$.

Based on Definition 5, we obtain θ weighted graphs derived from the temporal graph that falls into each of the θ windows in the stream. The value of θ is determined by users, which controls the space requirement and the efficiency of graph analysis, as well as the degree of information loss (from the original temporal graph to the θ weighted graphs). The larger the value of θ, the finer is the granularity of the windows and the less is the information loss, but also the more is the memory space needed.

Advantages of the new model. The equal-weight damped time window model has the following advantages: (A1) it is a generalization of existing damped time window models; (A2) it gives equal importance to each window, which makes it easy to compare the graphs from different windows; and (A3) it provides a systematical way for update maintenance of the windows.

For (A1), by defining an appropriate weight density function, we can apply our proposed equal-weight scheme to compute the size of each window for existing damped time window models. Take the logarithmic tilted-time window model as an example, where an exponentially increasing function (e.g., 2^0T, 2^1T, 2^2T, 2^3T, ...) is used. Assume that the time span of the entire stream is $[0, 2^\theta - 1]$, then the weight density function is defined as follows:

$$f(t) = \begin{cases} 1, & 0 \le t < 2^{\theta-1} \\ 2, & 2^{\theta-1} \le t < 2^{\theta-1} + 2^{\theta-2} \\ ..., & ... \\ 2^{\theta-1}, & 2^\theta - 2 \le t \le 2^\theta - 1 \end{cases}$$

For (A2), if the weights of an edge (u, v) in two different windows W_1 and W_2 are the same, then it implies that the probability of (u, v) being active remains the same in W_1 and W_2. Now if the probability of (u, v) being active is higher in W_1, then apparently the weight of (u, v) in W_1 is also higher than that in W_2. This may not be true if existing damped time windows are used unless we define an appropriate $f(t)$ function for them, and apply our scheme proposed in this section to determine the window sizes.

For (A3), we show that our model provides a systematical way for update maintenance of the windows in the following subsection.

C. Window Maintenance

As time goes on, new temporal edges are created and the windows need to be updated. We devise an update scheme for our window model as follows.

Let $[t_0, t_1]$, $[t_1, t_2]$, ..., $[t_{\theta-1}, t_\theta]$ denote the θ existing windows, and $[t_\theta, t_{\theta+1}]$ denote the new window. As the current time changes from $t_\tau = t_\theta$ to $t'_\tau = t_{\theta+1}$, the weight density function $f(t)$ changes from $f(t) = e^{\lambda(t-t_\theta)}$ to $f(t) = e^{\lambda(t-t_{\theta+1})}$. The following lemmas state the change needed.

Lemma 1: If the current time changes from t_τ to t'_τ, for any temporal edge whose weight w is last updated at time t_τ, the weight should be updated as follows:

$$w \leftarrow w \times e^{\lambda(t_\tau - t'_\tau)}.$$

Proof: Given an edge $(u, v, [t_i, t_j])$, its weight computed at time t_τ is $\int_{t_i}^{t_j} e^{\lambda(t-t_\tau)}dt$, and its weight at time t'_τ is $\int_{t_i}^{t_j} e^{\lambda(t-t'_\tau)}dt = \int_{t_i}^{t_j} e^{\lambda(t-t_\tau)}dt \times e^{\lambda(t_\tau-t'_\tau)}$. ■

Lemma 2: Given a weighted graph $G = (V, E, \Pi)$ of any window, if the current time changes from t_τ to t'_τ, the weight w of each edge in E which is computed at time t_τ should be updated as follows

$$w \leftarrow w \times e^{\lambda(t_\tau - t'_\tau)}.$$

Proof: The proof follows directly from Lemma 1. ■

Lemma 2 shows that it is simple to update the weighted graphs of the existing windows as new windows are created in the stream. However, we still need to determine at what point a new window, i.e., $[t_\theta, t_{\theta+1}]$, should be created in the stream, which is to determine $t_{\theta+1}$. Following our discussion in Section II-B, we have

$$\int_{t_{\theta-1}}^{t_\theta} f(t)dt = \int_{t_\theta}^{t_{\theta+1}} f(t)dt$$

$$\Rightarrow e^{\lambda t_\theta} - e^{\lambda t_{\theta-1}} = e^{\lambda t_{\theta+1}} - e^{\lambda t_\theta}$$

$$\Rightarrow t_{\theta+1} = \frac{1}{\lambda}\ln(2e^{\lambda t_\theta} - e^{\lambda t_{\theta-1}}).$$

Similarly, we can also compute the windows that are to follow in the stream, i.e., $[t_{\theta+1}, t_{\theta+2}]$, ..., and so on. However, in this way, the number of windows keeps increasing, and the size of a new window (i.e., the time span of the window) becomes smaller and smaller. To solve these issues, we propose to merge windows to bound the number of windows in the stream within the range $[\theta, 2\theta]$. Specifically, when the number of windows reaches 2θ, we merge every two consecutive windows into one window. In this way, every window in the stream after merging still satisfies the equal-weight window condition. In fact, we can also merge more than two windows into a single window if necessary.

III. Window-Based Network Analysis

We now discuss network analysis based on the equal-weight damped time window model, which we illustrate using two applications in this section, while we remark that there are a list of open problems that are interesting and useful for studying and analyzing large real-world temporal graphs under our model. We give a list of open problems in Section IV.

Let G_1, G_2, ..., G_θ denote the θ weighted graphs derived from the θ windows in the stream.

A. Connectivity Analysis

Given a weighted graph $G = (V, E, \Pi)$ of a window in the stream (defined in Definition 3, and here the window W is omitted for simplicity), we define a measure of connectivity between two vertices u and v in G as follows.

Definition 6 (Connectivity): Let $\mathbb{P}(u, v) = \{P(u, v) : P(u, v) \text{ is a path from } u \text{ to } v \text{ in } G\}$. The *connectivity* of a path $P(u, v)$, denoted by $\gamma(P(u, v))$, is defined as the minimum edge weight among the edges on $P(u, v)$. The *connectivity* between u and v, denoted by $\gamma(u, v)$, is defined as $\gamma(u, v) = \max\{\gamma(P(u, v)) : P(u, v) \in \mathbb{P}(u, v)\}$.

Since the weight of each edge in a weighted graph indicates the strength of relationship (or interaction, communication, etc.) between the two end points in the corresponding temporal graph within the time span of the window, the value of $\gamma(u, v)$ reflects the connectivity between u and v within the window, for example, the amount of information that can be passed between u and v via any path within the time span.

Given a *connectivity query* $\gamma(u, v)$, we can answer it using an algorithm similar to Dijkstra's algorithm, as shown in Algorithm 1. Algorithm 1 uses a maximum priority queue Q to keep the current largest connectivity value, $c[x]$, of a path from u to a visited vertex $x \in V$. The algorithm starts from one of the query vertices, u, greedily grows the paths by extending to u's neighbors, and then further grows to the neighbors' neighbors until reaching the other query vertex v. During the greedy process, the $c[x]$ value of a vertex x is updated whenever a larger connectivity value from u to x is found. At each iteration, the vertex with the maximum $c[.]$ value is extracted from Q to update the $c[.]$ values of its neighbors.

The following theorem proves the correctness and complexity of Algorithm 1.

Theorem 1: Algorithm 1 correctly computes the connectivity value $\gamma(u, v)$ in $O((|E| + |V|) \log |V|)$ time.

Proof: If $\mathbb{P}(u, v) \neq \emptyset$, it is easy to show that there exists a path $P = \langle u, u_1, u_2, \ldots, u_j = v\rangle \in \mathbb{P}$ such that every prefix subpath of P, $P_i = \langle u, u_1, u_2, \ldots, u_i\rangle$, gives the correct connectivity value $\gamma(u, u_i)$, i.e., $\gamma(u, u_i) = \gamma(P(u, v))$, for $1 \leq i \leq j$. The correctness follows from (proof by induction on i) that Algorithm 1 computes $c[u_i] = \gamma(u, u_i)$, for $1 \leq i \leq j$.

The complexity is the same as Dijkstra's algorithm. ■

The complexity of Algorithm 1, even if Fibonacci heap is used, is too high to process a connectivity query online. One may pre-compute the connectivity values for all pairs of vertices. However, the space complexity of this method is $O(|V|^2)$, and the pre-computation requires $O((|E|+|V|)|V| \log |V|)$ time, both of which are impractical for handling a large graph. We propose a more efficient way to process connectivity queries, with linear index space.

Algorithm 1: Compute $\gamma(u, v)$

```
Input  : A weighted graph $G = (V, E, \Pi)$, two query
         vertices $u$ and $v$
Output : $\gamma(u, v)$
1  Initialize $c[u] \leftarrow \infty$, $c[x] \leftarrow 0$ for every vertex $x \in V \setminus \{u\}$;
2  Let $Q$ be a maximum priority queue, where an element of $Q$
   is a pair $(x, c[x])$ and $c[x]$ being the key;
3  Initialize $Q$ by inserting a single element $(u, c[u])$;
4  while $Q$ is not empty do
5      $(x, c[x]) \leftarrow$ Extract-Max($Q$);
6      if $x = v$ then
7          Goto Line 12;
8      foreach neighbor vertex, $y$, of $x$ do
9          if $c[y] < \min(c[x], \Pi(x, y))$ then
10             $c[y] \leftarrow \min(c[x], \Pi(x, y))$;
11             If $y$ is not in $Q$, push $(y, c[y])$ into $Q$; otherwise,
               update $c[y]$ in $Q$;
12 return $\gamma(u, v) = c[v]$;
```

First, we compute a *maximum spanning tree*, denoted by *MaxST*, of the weighted graph G. Without loss of generality, we assume G is connected. If not, we can consider each connected component of G separately. A MaxST has the cut property. A *cut* is a partition of the vertex set of a graph into two disjoint subsets. We say that *an edge crosses the cut* if it has one endpoint in each subset of the partition. The *cut property* states that for any cut C in the graph, if the weight of an edge e crossing C is larger than the weights of all the other edges crossing C, then e must be contained in every MaxST.

Given a MaxST, T, there is a unique path connecting any two vertices in T. Let $\gamma_T(u, v)$ denote the connectivity value between u and v in the MaxST T. Based on the cut property, we have the following lemma.

Lemma 3: Given a MaxST T of a weighted graph G, $\gamma(u, v) = \gamma_T(u, v)$, for any pair of vertices u and v in G.

Proof: For any pair of vertices u and v, there is a unique path $P_T(u, v)$ connecting u and v in T. Let e be the edge with minimum weight among the edges on $P_T(u, v)$. If we remove e, the tree T will be divided into two disjoint components T_1 and T_2. Let V_1 (V_2) be the set of vertices in T_1 (T_2). By the cut property, among the edges crossing the cut (V_1, V_2), e is the one with the largest weight, since if there exists another edge e' in the cut (V_1, V_2) with larger weight than e, then e' should be included in the MaxST instead of e. Thus, $\gamma(u, v) \leq \Pi(e) = \gamma_T(u, v)$. Also, by Definition 6, we have $\gamma(u, v) \geq \gamma_T(u, v)$ since $P_T(u, v) \in \mathbb{P}(u, v)$. Thus, $\gamma(u, v) = \gamma_T(u, v)$. ■

Based on Lemma 3, a connectivity query $\gamma(u, v)$ can be answered by first finding the unique path between u and v in the MaxST T, and then returning the minimum edge weight on the path. The query time complexity is $O(|V|)$, which is much better than that of Algorithm 1. Next, we show that we can further reduce the querying time complexity to $O(1)$ time.

We first introduce the concept of *Cartesian tree* [13], which is a binary tree derived from a sequence of numbers. Given an array A of n numbers ($A[0]$ to $A[n-1]$), the root of

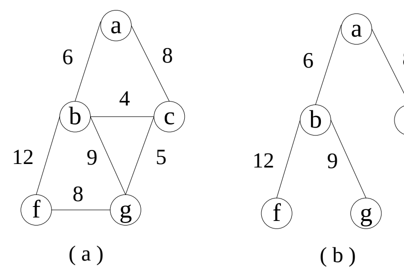


Figure 2. **(a)** a **weighted graph** G, and **(b)** a **MaxST** T

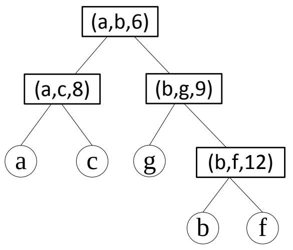


Figure 3. The **Cartesian tree** C_T of T in Figure 2(b)

the Cartesian tree is the minimum number among all the n numbers. Let $A[i]$ be the minimum number, i.e., the root. Then, its left subtree is computed recursively on the numbers $A[0]$ to $A[i-1]$, while its right subtree is computed recursively on the numbers $A[i+1]$ to $A[n-1]$.

We construct a Cartesian tree, C_T, based on a MaxST T. The root node of C_T is the edge with the minimum weight among all the edges in T. Then, by removing this edge, T will be partitioned into two subtrees. Following a similar procedure, we can recursively construct the left and right subtrees of the root node. When removing an edge (u, v), if u (and/or v) is not an end point of any remaining edges in T, then we also create a leaf node u (and/or v) as the child of the node (u, v) in C_T. Thus, the set of leaf nodes in the tree C_T corresponds to the set of vertices in T.

Based on the Cartesian tree C_T, given a connectivity query $\gamma(u, v)$, we can first find the lowest common ancestor (LCA) of the two leaves u and v in C_T, and then return the weight of the edge in T that corresponds to the LCA.

The following example demonstrates the concepts of MaxST, Cartesian tree, and how to answer a connectivity query.

Example 2: Figure 2(a) shows a weighted graph G and Figure 2(b) shows a MaxST T of G. It is easy to verify $\gamma(u, v) = \gamma_T(u, v)$. For example, $\gamma(c, f) = 6$ in G and $\gamma_T(c, f) = 6$ in T. Figure 3 shows the Cartesian tree C_T of T. The root node of C_T is the edge $(a, b, 6)$, since this edge is the one with the minimum weight in T. Removing $(a, b, 6)$ partitions T into two components $\{a, c\}$ and $\{b, f, g\}$. Following a similar procedure recursively, we obtain C_T. The leaves of C_T are the vertices in T. Then, to find the connectivity value between any two vertices, we find the LCA of these two vertices in C_T. For example, given a connectivity query $\gamma(f, g)$, we find that the edge $(b, g, 9)$ is the LCA of the leaves f and g in C_T. Thus, we return 9 as the answer for $\gamma(f, g)$. It is easy to verify that the answer is correct.

Now, we give the complexity of processing a connectivity query and of constructing the index.

Theorem 2: A connectivity query $\gamma(u, v)$ can be answered in $O(1)$ time with an index using $O(|V|)$ space, and the index construction time is $O((|E| + |V|) \log |V|)$.

Proof: According to [14], an LCA query can be answered in $O(1)$ time with an index that has linear size and can be constructed in linear time. According to [15], if the edge weights of T are sorted, the Cartesian tree C_T can be constructed in linear time. Thus, the overall time complexity for index construction is bounded by the computation of MaxST, which is $O((|E|+|V|) \log |V|)$. Since both the MaxST and the index for answering LCA queries have $O(|V|)$ size, the overall index space is $O(|V|)$. ■

Given the θ weighted graphs G_1, G_2, ..., G_θ from the θ windows in the stream, we define the connectivity between u and v in the entire θ windows as $\Gamma(u, v) = \min\{\gamma_1(u, v), \ldots, \gamma_\theta(u, v)\}$, where $\gamma_i(u, v)$ is the connectivity value $\gamma(u, v)$ in the weighted graph G_i, for $1 \leq i \leq \theta$. Since θ is a constant, the query $\Gamma(u, v)$ can be answered in constant time with indexes of size linear to the number of vertices.

B. Core Community

Let H be an induced subgraph of a weighted graph $G = (V, E, \Pi)$. The weight of H, denoted by $\Pi(H)$, is defined as the sum of the weight of the edges in H. Let $den(H)$ denote the density of H, defined as $den(H) = \frac{\Pi(H)}{|H|}$.

Density has been used to define communities in a graph such as densest subgraphs of a graph [16], [17]. However, it is possible that within a community, some vertices are loosely connected to other vertices, while the density of the community is still high. To make sure that every vertex in the community is well connected, we give the definition of η-community as follows.

Let $\Pi(v, G)$ denote the weight of a vertex $v \in V$ in G, which is defined as the sum of the weights of the edges that are incident to v, that is, $\Pi(v, G) = \sum_{(u,v) \in E} \Pi(u, v)$. Let $G[S]$ be the subgraph of G induced by $S \subseteq V$.

Definition 7 (η-community): Given a weighted graph $G = (V, E, \Pi)$ and a number $\eta > 0$, we say that a set of vertices $S \subseteq V$ forms a *η-community*, if $\Pi(v, G[S]) \geq \eta$ for every $v \in S$.

Intuitively, Definition 7 makes sure that every vertex in a community should actively interact with their neighboring vertices. The number η serves as a threshold value to control the level of interaction between a vertex with its neighbors.

Given a value η, an *η-community query* is to find the *largest* η-community.

A naive solution to answer an η-community query is to enumerate all the vertex subset of V, and check whether it satisfies Definition 7. Apparently, the naive solution is not practical. We propose an efficient solution to answer an η-community query for any given η as follows.

Our solution is based on the concept of weighted k-core. A subgraph H of G is a *weighted k-core* [18], [19] if for every vertex v in H, $\Pi(v, H) \geq k$. Note that if H is a weighted k'-core, where $k' > k$, then H is also a weighted k-core (since $\Pi(v, H) \geq k' > k$). The *weighted core number* of a vertex v, denoted by $\phi(v)$, is defined as $\phi(v) = k$ such that v is in a weighted k-core and v is not contained in any weighted

Algorithm 2: Weighted core decomposition

```
Input  : An undirected weighted graph $G = (V, E, \Pi)$
Output : The weighted core number, $\phi(v)$, of each $v \in V$
1 Initialize $w(v) \leftarrow \Pi(v, G)$ for each $v \in V$;
2 Sort the vertices in $V$ in ascending order of $w(.)$ value;
3 foreach $v \in V$ in sorted order do
4     $\phi(v) \leftarrow w(v)$;
5     foreach neighbor vertex, $u$, of $v$ do
6         if $w(u) > w(v)$ then
7             $w(u) \leftarrow \max\{w(u) - \Pi(u, v), w(v)\}$;
8             Re-order $u$ in $V$;
9     Remove $v$ from $V$;
```

k'-core, where $k' > k$. Clearly, an η-community is a weighted η-core.

Weighted core decomposition in a weighted graph G is to compute the largest non-empty weighted k-core of G for every $k > 0$. Alternatively, we can compute $\phi(v)$ for every vertex $v \in V$, since the largest weighted k-core is simply the subgraph of G induced by the vertex set $\{v : v \in V, \phi(v) \geq k\}$.

The algorithm for weighted core decomposition is simple, as shown in Algorithm 2. The idea is to recursively remove from G the vertex with the lowest weight. When a vertex v is removed, its incident edges should also be removed, and hence the weight of its neighbors is updated as in Line 7 if the neighbor vertex has larger weight than v. The weighted core number of a vertex v is simply its weight at the time when v is removed, since the weight of every vertex in G is larger than or equal to v at that time, and v cannot be contained in any weighted core with a larger core number.

The following theorem gives the complexity of Algorithm 2.

Theorem 3: Algorithm 2 correctly computes $\phi(v)$ for each vertex $v \in V$ in a weighted graph $G = (V, E, \Pi)$ using $O((|V| + |E|) \log |V|)$ time.

Proof: The correctness is easy to see from the definition of weighted core number. The time complexity is bounded by Line 2 and 8 of Algorithm 2, which take $O((|V|+|E|) \log |V|)$ time. ■

Then, given an η-community query, the query answer is simply the set of vertices S such that $\forall v \in S$, $\phi(v) \geq \eta$.

Given the θ weighted graphs G_1, G_2, …, G_θ from the θ windows in the stream, we want to find the largest *common* η-community in all the θ weighted graphs. We call this query a (θ, η)-community query. To answer this query, we first define (θ, η)-community as follows.

Definition 8 ((θ, η)-community): Given θ weighted graphs $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, …, $G_\theta = (V, E_\theta, \Pi_\theta)$, a set of vertices $S \subseteq V$ is a *(θ, η)-community*, if $\forall i \in [1..\theta]$, $G_i[S]$ is an η-community of G_i.

Similar to weighted core number, we define the *θ-weighted core number* of a vertex v, denoted by $\Phi(v)$, as $\Phi(v) = k$ such that v is in a (θ, k)-community and v is not in any (θ, k')-community where $k' > k$.

Example 3: Figures 4(a) and (b) show two weighted graphs G_1 and G_2. In this case, $\theta = 2$. The set $\{a, b, c\}$ is a $(2, 6)$-community since $\{a, b, c\}$ is a 6-community in both G_1 and G_2. The θ-weighted core number of vertex b is 6 since there is no $(2, k)$-community containing b, where $k > 6$. On the contrary, we can find the weighted core number of b is 9 in both G_1 and G_2, which is larger than 6.

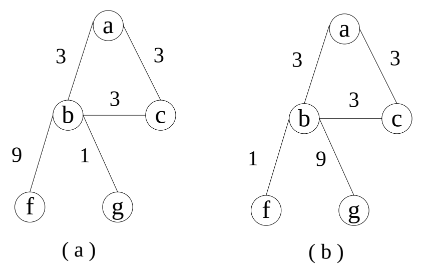


Figure 4. **(a)** a **weighted graph** G_1, and **(b)** a **weighted graph** G_2

Algorithm 3: θ-weighted core number computation

```
Input  : $\theta$ weighted graphs: $G_1 = (V, E_1, \Pi_1)$,
         $G_2 = (V, E_2, \Pi_2)$, …, $G_k = (V, E_\theta, \Pi_\theta)$
Output : The $\theta$-weighted core number, $\Phi(v)$, of each $v \in V$
1 For each $v \in V$, initialize $w_i(v) \leftarrow \Pi(v, G_i)$, for $1 \leq i \leq \theta$;
  set $w(v) \leftarrow \min\{w_1(v), w_2(v), \ldots, w_\theta(v)\}$;
2 Sort the vertices in $V$ in ascending order of $w(.)$ value;
3 foreach $v \in V$ in sorted order do
4     $\Phi(v) \leftarrow w(v)$;
5     Let $U$ be the union of the set of neighbor vertices of $v$ in
      $G_i$, for $1 \leq i \leq \theta$;
6     foreach vertex $u \in U$ do
7         if $w(u) > w(v)$ then
8             $w_i(u) \leftarrow \max\{w_i(u) - \Pi_i(u, v), w(v)\}$, for
              $1 \leq i \leq \theta$;
9             $w(u) \leftarrow \min\{w_1(u), w_2(u), \ldots, w_\theta(u)\}$;
10            Re-order $u$ in $V$;
11    Remove $v$ from $V$;
```

According to Definition 8, a simple algorithm to compute $\Phi(v)$ is to find a (θ, k)-community that gives the maximum value of k to $\Phi(v)$ as follows. For each subset of V that contains v, compute the maximum k for each such subset S such that $G_i[S]$ is an η-community of each G_i for $1 \leq i \leq \theta$. Then among all these subsets, assign $\Phi(v)$ as the maximum value of k obtained.

Apparently, the above-described algorithm is infeasible since there are $2^{|V \setminus \{v\}|}$ subsets of V that contains v. We propose an efficient solution as shown in Algorithm 3.

The algorithm uses a similar recursive procedure as in Algorithm 2, but it is not trivial to see the correctness, which we prove in the following theorem.

Theorem 4: Given θ weighted graphs, $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, …, $G_\theta = (V, E_\theta, \Pi_\theta)$, Algorithm 3 correctly computes $\Phi(v)$ for each vertex $v \in V$ in $O((|V| + \sum_{1 \leq i \leq \theta} |E_i|) \log |V|)$ time.

Proof: We first prove the correctness. In Algorithm 3, $w(v)$ keeps the minimum value among $\{w_1(v), w_2(v), \ldots, w_\theta(v)\}$, for each vertex $v \in V$. The vertices are sorted in ascending order of their $w(.)$ value. When a vertex v is removed, $w(v)$ is the minimum among the remaining vertices in V, which indicates that v is in a $(\theta, w(v))$-community. On the other hand, v cannot be in any (θ, k)-community with $k > w(v)$. Thus, Algorithm 3 correctly computes $\Phi(v)$ for every vertex $v \in V$.

The time complexity is bounded by Lines 2 and 10 of Algorithm 3, which take $O(|V| \log |V|)$ and $O((\sum_{1 \leq i \leq \theta} |E_i|) \log |V|)$ time, respectively. ■

Then, given a (θ, η)-community query, the query answer is simply the set of vertices S such that $\forall v \in S$, $\Phi(v) \geq \eta$.

C. *Queries on a Random Window*

Besides the need of analysis on a temporal graph in the whole time window, users may also be interested in analyzing the graph in any time period. For example, user A is interested in time window $[1, 20]$, while user B is interested in time range $[10, 40]$. To satisfy each user's need, the naive way is to store the complete temporal graph and extract the temporal subgraph from the required time range, which is not practical due to the massive size of the complete graph. We discuss how to efficiently obtain a weighted graph of any time period based on the equal-weight damped time window model.

Given a random window $W = [t_x, t_y]$, we are required to return $G_W = (V, E_W, \Pi_W)$. Given θ weighted graphs, $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, …, $G_\theta = (V, E_\theta, \Pi_\theta)$, we return an approximate weighted graph G'_W of G_W as follows.

Let $t_i < t_x \leq t_{i+1}$ and $t_j \leq t_y < t_{j+1}$. First, we return an approximate weighted graph $G'_{[t_x,t_{i+1}]}$ of $G_{[t_x,t_{i+1}]}$. $G'_{[t_x,t_{i+1}]} = (V, E'_{[t_x,t_{i+1}]}, \Pi'_{[t_x,t_{i+1}]})$ is computed as follows:

- $E'_{[t_x,t_{i+1}]} = E_{i+1}$,
- $\Pi'_{[t_x,t_{i+1}]}(e) = \Pi_{i+1}(e) \times \frac{\int_{t_x}^{t_{i+1}} f(t)dt}{\int_{t_i}^{t_{i+1}} f(t)dt}$, for each $e \in E'_{[t_x,t_{i+1}]}$.

In other words, $G'_{[t_x,t_{i+1}]}$ is computed based on $G_{i+1} = (V, E_{i+1}, \Pi_{i+1})$ in expectation. Similarly, we compute an approximate weighted graph $G'_{[t_j,t_y]}$ of $G_{[t_j,t_y]}$. Then, we have $G'_W = (V, E'_W, \Pi'_W)$ as follows:

- $E'_W = E'_{[t_x,t_{i+1}]} \cup E_{i+2} \cup \ldots \cup E'_{[t_j,t_y]}$,
- $\Pi'_W(e) = \Pi'_{[t_x,t_{i+1}]}(e) + \Pi_{i+2}(e) + \ldots + \Pi'_{[t_j,t_y]}(e)$, for each $e \in E'_W$.

Next, let us consider the two applications discussed in Sections III-A and III-B, by focusing on a random time window $W = [t_x, t_y]$. For example, if we query for the connectivity between u and v in G_W, or the weighted core number of a vertex v in G_W, we can first compute an approximate graph G'_W of G_W, and then apply online search to answer the queries. However, the query time of this approach is expensive, and there is no bound on the quality of the query answer. We present a better way to answer the queries for a random time window $W = [t_x, t_y]$ as follows.

First, we discuss the connectivity query. Let us first assume that t_x and t_y are chosen from $\{t_0, t_1, \ldots, t_\theta\}$. In this case, the window $[t_x, t_y]$ may consist of multiple consecutive windows or a single window of the θ windows. Since θ is normally a small number, we can construct indexes as described in Section III-A for each weighted graph of the window $[t_i, t_j]$, where $0 \leq i < j \leq \theta$. The index size is $O(\theta^2|V|)$, which is acceptable for small θ. Then, we can answer any connectivity query for any consecutive windows in constant time.

Next, we consider the case that t_x and/or t_y is not chosen from $\{t_0, t_1, \ldots, t_\theta\}$. In this case, we can find a *minimal window* $W_1 = [t_{i_1}, t_{j_1}]$ to cover $[t_x, t_y]$, and a *maximal window* $W_2 = [t_{i_2}, t_{j_2}]$ that is covered by $[t_x, t_y]$, where $0 \leq i_1 \leq j_1 \leq \theta$ and $0 \leq i_2 \leq j_2 \leq \theta$. Then, we can obtain a upper bound and a lower bound on the query answer based on the existing indexes on W_1 and W_2, which can be answered in constant time.

If θ is not so small, we can construct indexes for less number of weighted graphs of consecutive windows. For example, we can construct indexes for weighted graphs of each single window, every two consecutive windows ($\{[t_0, t_1], [t_2, t_3], \ldots\}$), every four consecutive windows ($\{[t_0, t_4], [t_4, t_8], \ldots\}$), and so on. In this way, the total index size is $O(\theta|V|)$, and we can also find a lower bound and a upper bound for any connectivity query. Similarly, we can also efficiently handle the query of the weighted core number of a vertex in a random window.

IV. Open Problems

In this section, we discuss a list of open problems based on the equal-weight damped time window model, each of them has important applications.

A. *Densest Community*

Dense subgraphs are useful in detecting communities, finding compressed representations of graphs, etc. [20], [21], [22].

Given a weighted graph $G = (V, E, \Pi)$, let H be an induced subgraph of G. The densest subgraph problem in G is to find an induced subgraph H which has the maximum density, $den(H)$, among all the induced subgraphs of G, where $den(H)$ is defined in Section III-B.

Next, we extend to the definition of densest θ-weighted community.

Definition 9 (Densest θ-weighted community): Given θ weighted graphs $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, …, $G_\theta = (V, E_\theta, \Pi_\theta)$, define the θ-weighted density of a set of vertices $S \subseteq V$ as $Den(S) = \min\{den(H_1), den(H_2), \ldots, den(H_\theta)\}$, where $H_i = G_i[S]$ for $1 \leq i \leq \theta$. The *densest θ-weighted community* is the subset of V that has the maximum θ-weighted density among all the vertex subsets of V.

The definition can also be extended to at-least-k or at-most-k densest θ-weighted community, for which we require the size of the subset to contain at least k or at most k vertices. These problems are all NP-hard, but approximation solutions can be studied. For example, there is an efficient 1/2-approximation algorithm for the densest subgraph problem [23], and k-core has been applied to obtain an efficient 1/3-approximation solution for the at least k densest subgraph problem [16].

B. *Subgraph Matching*

Subgraph matching is a fundamental operation in subgraph mining, network analysis, etc. [24], [25].

The subgraph matching problem is to find the subgraphs of a data graph that are isomorphic to a given query graph. For subgraph matching in a weighted graph, we not only require isomorphism between the query graph and a matching subgraph, but also set a constraint on the matching of the edge weights, as we define below.

Definition 10 (θ-weighted subgraph matching): Given θ weighted graphs $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, …, $G_\theta = (V, E_\theta, \Pi_\theta)$, a query graph $G_q = (V_q, E_q)$, and an edge weight constraint c_e, the problem of *θ-weighted subgraph matching* is to find all the vertex mapping from a subset of

Table I. REAL TEMPORAL GRAPHS ($\mathrm{K} = 10^3$)

Dataset	$\lvert\mathbb{V}\rvert$	$\lvert\mathbb{E}\rvert$	$d_{avg}(v, \mathbb{G})$	$\lvert T_{\mathbb{G}}\rvert$
phone	1,237	338,008,540	273,248.62	3,369
arxiv	28,094	9,193,606	327.24	2,337
elec	8,298	214,028	25.79	101,063
enron	87,274	2,282,904	26.16	220,364
facebook	46,953	1,730,624	36.86	867,939
lastfm	174,078	38,254,660	219.76	17,498,009
email	168	164,613	979.84	57,842
conflict	118,101	5,903,522	49.99	312,457

vertices to V_q, let $map(v)$ denote the mapping vertex in V of a vertex $v \in V_q$, which satisfies:

- $map(v) \in V$, for each $v \in V_q$;
- $(map(v_1), map(v_2)) \in E_i$, for each $e = (v_1, v_2) \in E_q$, $1 \leq i \leq \theta$;
- Each edge $(map(v_1), map(v_2))$ satisfies weight constraint c_e, for each $e = (v_1, v_2) \in E_q$, $1 \leq i \leq \theta$.

The edge weight constraint c_e is defined by an application, for example, it can be a threshold on the weight so that every matching edge must have weight above the threshold.

C. Shortest Path

The shortest path problem is a fundamental problem with numerous applications, and also serves as a building block of many algorithms. Given an undirected weighted graph, a shortest path between vertex u and v is a path with minimum total weight among all the paths between u and v. We extend the definition to the case when we have θ weighted graphs as follows.

Definition 11 (θ-weighted shortest path): Given θ weighted graphs $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, ..., $G_\theta = (V, E_\theta, \Pi_\theta)$, for any two vertices $u, v \in V$, we say that a path P is a *θ-weighted shortest path* between u and v if $len(P) = \max\{len_1(P), len_2(P), \ldots, len_\theta(P)\}$ is the minimum among all the paths between u and v, where $len_i(P)$ is the total weight of the edges of P in G_i, for $1 \leq i \leq \theta$.

Note that for the problem of θ-weighted shortest path, the weight function Π_i of each graph G_i is different from the setting in Section III-A. Here, the smaller $len_i(P)$ value, the closer is the relationship between u and v in G_i, for $1 \leq i \leq \theta$.

D. Minimum Cut

Minimum cut is a classical problem in both graph theory and real applications. A cut is a partition of the vertices into two disjoint subsets by removing a set of edges. A minimum cut in a weighted graph is a cut that has the smallest sum of weights of the cut edges. We extend the definition to the case when we have θ weighted graphs as follows.

Definition 12 (θ-weighted minimum cut): Given θ weighted graphs $G_1 = (V, E_1, \Pi_1)$, $G_2 = (V, E_2, \Pi_2)$, ..., $G_\theta = (V, E_\theta, \Pi_\theta)$, let $\Pi_i(C)$ be the total weight of the edges of a cut C in G_i, where $1 \leq i \leq \theta$. For any two vertices $s, t \in V$, we say that a cut C is a *θ-weighted minimum s-t cut* if $\max\{\Pi_1(C), \Pi_2(C), \ldots, \Pi_\theta(C)\}$ is the minimum among all the cuts that put s and t into two different subsets.

V. Experimental Results

We evaluated the usefulness of our equal-weight window model by showing the quality of the θ weighted graphs obtained based on the model, and the efficiency and quality of graph analysis based on these weighted graphs. We also verified the efficiency of dynamic update maintenance and the scalability of our method. All the experiments were run on a Linux machine with an Intel 3.3GHz CPU and 16GB RAM. All the programs were implemented in C++ and complied using G++ 4.8.2.

We used 8 real temporal graphs for our experiments, as shown in Table I, where we list the number of vertices and edges in each graph $\mathbb{G}$, the average degree in $\mathbb{G}$ (denoted by $d_{avg}(v, \mathbb{G})$), and the number of distinct time instances in $\mathbb{G}$ (denoted by $|T_{\mathbb{G}}|$). The phone graph consists of call records in Ivory Coast [26], where the call records were collected over a span of 150 days. The other 7 graphs were obtained from the Koblenz Large Network Collection (http://konect.uni-koblenz.de/), where one large temporal graph was selected from each of the following 7 categories: arxiv-HepPh (arxiv) from the arxiv networks; elec from the network of English Wikipedia; enron from the email networks; facebook-links (facebook) from the facebook network; lastfm-band (lastfm) from the music website last.fm; radoslaw-email (email) from the internal email communication network between employees of a mid-sized manufacturing company; wikiconflict (conflict) indicating conflicts between users of Wikipedia.

A. Results on Weighted Graph Construction

In this experiment, we evaluated the space requirement and the construction time of the θ weighted graphs for each of the temporal graphs, and then we measured the quality of the weighted graphs. We tested θ from 10 to 50. We set the value of $\lambda = 10^{-x}$ for the weight density function given in Definition 1, where $10^x \leq |T_{\mathbb{G}}| < 10^{x+1}$, that is, $\lambda = 10^{-\lfloor \log_{10} |T_{\mathbb{G}}| \rfloor}$. For example, for the phone graph, $\lambda = 10^{-3}$.

Space requirement. We first report the space requirement for the θ weighted graphs, as a percentage of the original temporal graph shown in Figure 5. As the value of θ increases, the total size of the θ weighted graphs also increases. However, the rate of increase is slow. For graphs with high average degree, the total size of the θ weighted graphs is only a small percentage of the original temporal graph. For example, for the phone graph, even the total size of 50 weighted graphs is less than 10 percentage of the original temporal graph. We emphasize that for temporal graphs, the set of vertices remains relatively stable while the number of temporal edges grows linearly over time, and thus the result verifies that our method can handle large temporal graphs as they grow over time, with small space requirement.

Construction time. Table II reports the time taken to read the temporal graphs from disk and construct the corresponding θ weighted graphs, for different values of θ. The construction is fast for all graphs as we only need to scan the graphs once, regardless of the value of θ. The construction time increases as θ increases because more weighted graphs need to be constructed, but the rate of increase is slow as scanning the original temporal graph dominates the cost.

Quality of results. Next, we examine the quality of the weighted graphs. To do this, we constructed a weighted graph, G_W, directly from the original temporal graph within a time window W, as defined in Definition 3. We also constructed an

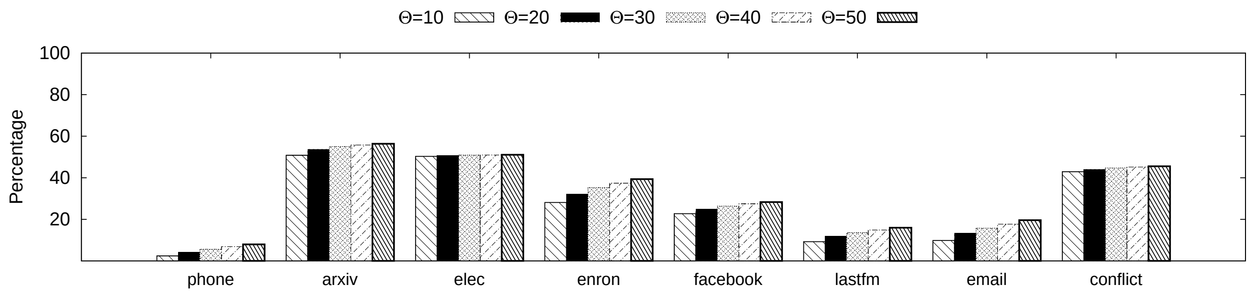


Figure 5. The total size of the θ weighted graphs compared with the original temporal graph $\mathbb{G}$

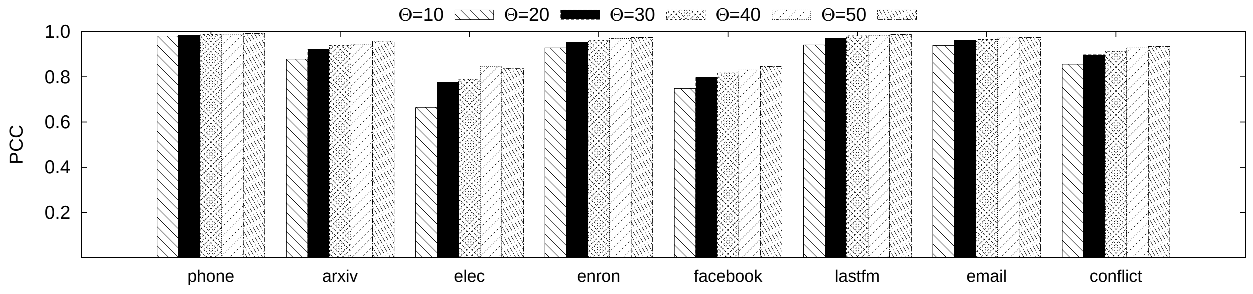


Figure 6. PCC between G'_W and G_W for different θ

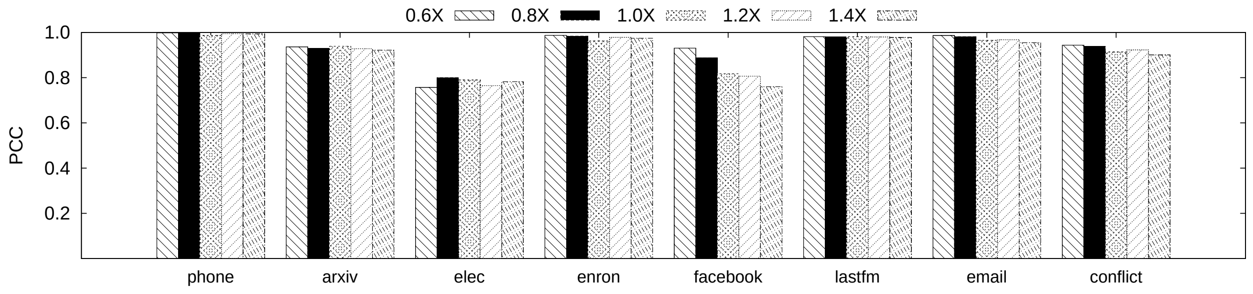


Figure 7. PCC between G'_W and G_W for different λ ($\theta = 30$)

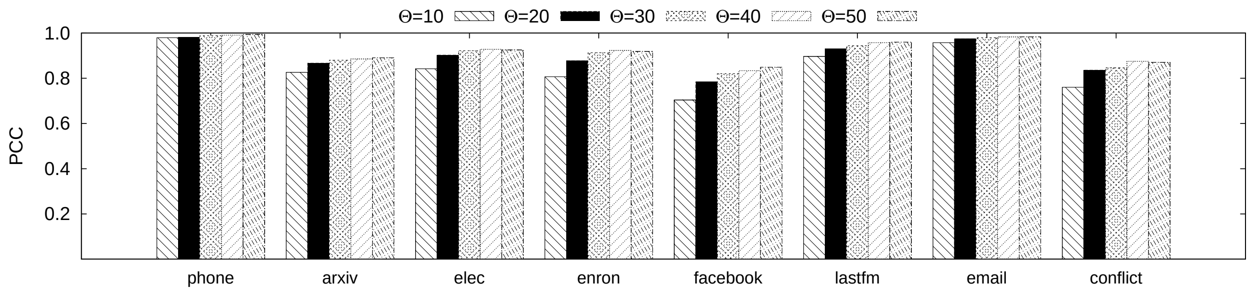


Figure 8. PCC between the connectivity computed from G'_W and that computed from G_W

Table II. CONSTRUCTION TIME OF θ WEIGHTED GRAPHS (IN SECONDS)

Dataset	$\theta = 10$	$\theta = 20$	$\theta = 30$	$\theta = 40$	$\theta = 50$
`phone`	130.3067	137.6559	143.6432	148.6535	153.1762
`arxiv`	4.0591	4.2070	4.3718	4.4772	4.5788
`elec`	0.1110	0.1168	0.1229	0.1266	0.1292
`enron`	0.8419	0.9031	0.9600	1.0041	1.0473
`facebook`	0.6245	0.6743	0.6996	0.7325	0.7581
`lastfm`	12.6525	13.2842	14.0147	14.6400	15.5061
`email`	0.0511	0.0548	0.0575	0.0607	0.0617
`conflict`	2.8762	2.9693	3.0447	3.1341	3.2189

approximate weighted graph G'_W of G_W from the θ weighted graphs as discussed in Section III-C. Then, we compared G_W and G'_W.

We computed G_W and G'_W for 100 randomly generated windows, $W = [t_x, t_y]$, where we ensured that W is a valid window by ensuring $t_x < t_y$. We use *Pearson correlation coefficient* (*PCC*) to measure the degree of linear correlation between G'_W and G_W, and report the results in Figure 6.

The result shows that we obtain high PCC values in most of the cases, which implies that analysis conducted on the approximate graph G'_W shares similar patterns/trends with that

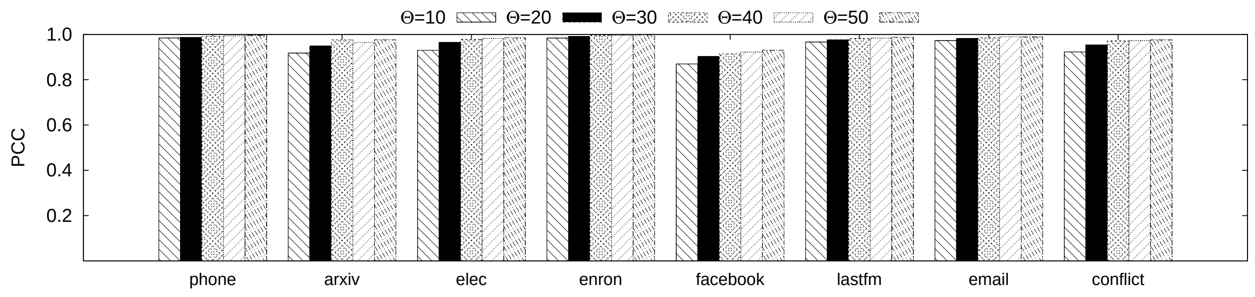


Figure 9. PCC between weighted core numbers computed from G'_W and that computed from G_W

conducted on the exact graph G_W (we will further verify this point in Figures 8 and 9). As θ increases from 10 to 50, the PCC values also increase, verifying that a larger θ leads to less information loss and hence higher correlation between G'_W and G_W. For a number of graphs, the PCC values are close to 1. The results are particularly impressive for the `phone` graph[1], for which the space requirement is also very small as shown in Figure 5.

Next, we tested the effect of different values of λ. In all the other experiments, we set $\lambda = 10^{-\lfloor \log_{10} |T_{\mathbb{G}}| \rfloor}$ as default. In this experiment, we tested λ at 0.6, 0.8, 1.0, 1.2, and 1.4 of its default value, and fixed $\theta = 30$. The result, as reported in Figure 7, shows that the PCC values are not much affected by the change in λ, and in all cases the PCC values are high.

We further evaluated the quality of using the approximate graph G'_W for graph analysis, compared with the results obtained from the exact graph G_W. We first processed 1000 randomly generated connectivity queries in each G'_W and G_W. We computed the PCC between the connectivity computed from G'_W and that computed from G_W. Figure 8 reports the result, which shows a high PCC value between the results computed from G'_W and G_W. The result reveals that the mutual relationship between pairs of vertices is similar in G'_W and G_W; in other words, given a vertex v, the vertices closely connected to v are similar in G'_W and G_W.

Then, we computed the weighted core number of each vertex in each G'_W and G_W. We report the PCC between the set of weighted core numbers computed from G'_W and that computed from G_W in Figure 9. For all datasets, the PCC between the results computed from G'_W and G_W is very high, indicating that the rank or the importance of each vertex in G'_W is similar to that in G_W.

B. *Efficiency of Graph Analysis*

In this experiment, we evaluated the efficiency of using the θ weighted graphs for connectivity analysis and core community analysis. We varied θ from 10 to 50.

We first tested 1000 randomly generated connectivity queries. We used the index presented in Section III-A to answer the queries, and compared with the online algorithm given in Algorithm 1. We denote these two methods by *Index* and *Online*, respectively. Table III reports the average processing time per query. The result shows that *Index* is more than 3 orders of magnitude faster than *Online*, verifying the efficiency of our method. The index construction time and the index size are also small, which are linear to the number of vertices (as shown Table I).

Table IV. COMPUTATION TIME FOR θ-WEIGHTED CORE DECOMPOSITION (IN SECONDS)

Dataset	$\theta = 10$	$\theta = 20$	$\theta = 30$	$\theta = 40$	$\theta = 50$
`phone`	1.9543	3.4277	4.6500	5.7795	6.7841
`arxiv`	1.4948	1.5417	1.5895	1.6313	1.6693
`elec`	0.0462	0.0494	0.0550	0.0591	0.0633
`enron`	0.2712	0.3384	0.4045	0.4573	0.5024
`facebook`	0.1561	0.1805	0.2143	0.2404	0.2659
`lastfm`	1.1827	1.5900	1.8855	2.1598	2.3740
`email`	0.0043	0.0058	0.0068	0.0066	0.0084
`conflict`	0.9882	1.0630	1.1235	1.1807	1.2394

Next, we computed the θ-weighted core number of each vertex from the θ weighted graphs. We report the total running time in Table IV. The result shows that our algorithm is very efficient. Even when $\theta = 50$, the total running time is only a few seconds for all the datasets. The running time increases slowly as θ increases. This is because the total size of the θ weighted graphs increases slowly as θ, as shown in Figure 5.

C. *Performance on Dynamic Updating*

In this experiment, we evaluated the performance of dynamic update maintenance of the θ weighted graphs in our equal-weight window model. We set $\theta = 15$. Since our update scheme keeps the number of windows in the stream within the range of $[\theta, 2\theta]$, we started with 15 windows initially and increased the number of windows as new temporal edges come in. As shown in the first row of Table V, the number of windows increases from 16 to 30. When the number of windows reached 30, we merged every two consecutive windows to reduce the number of windows back to 15.

Table V reports the average updating time for each edge insertion during the period from the creation of a new window to the creation of the next window. The average updating time also includes the time for constructing the weighted graph of the new window, as well as the time for updating the existing weighted graphs. In other words, the average updating time is the amortized cost per edge insertion. When θ reaches 30, the time for merging every two windows is also included in the updating time. The last column, denoted by "whole", is the amortized updating time over the whole period when the number of windows increases from 15 to 30 and then merged (back to 15 windows).

From Table V, we can see that the updating time is very short and stable over the entire cycle when the number of

[1]This is actually one of the main reasons why our telecom collaborator is interested in our work (see motivation in Section I). Our method is more effective on their dataset because their dataset is much larger, and hence the total size of the θ weighted graphs is much smaller than the size of the original temporal graph, while accurate analytic results can be obtained based on the θ weighted graphs. We cannot use their dataset in this paper as it involves the privacy of customers.

Table III. Average query processing time of connectivity queries (in milliseconds)

	$\theta = 10$		$\theta = 20$		$\theta = 30$		$\theta = 40$		$\theta = 50$	
	Index	Online	Index	Online	Index	Online	Index	Online	Index	Online
phone	0.0041	24.0973	0.0059	42.0291	0.0082	56.7214	0.0095	71.6784	0.0116	85.7897
arxiv	0.0045	21.6051	0.0081	19.8686	0.0127	18.9631	0.0160	18.3361	0.0188	17.8703
elec	0.0029	0.5434	0.0057	0.5460	0.0083	0.6127	0.0108	0.6866	0.0140	0.7909
enron	0.0049	5.6191	0.0103	6.2276	0.0132	6.8081	0.0186	7.7741	0.0230	8.7257
facebook	0.0052	5.7762	0.0095	5.5650	0.0141	5.6486	0.0185	5.9857	0.0231	6.4586
lastfm	0.0062	31.5103	0.0122	34.3018	0.0201	40.0901	0.0249	43.6900	0.0331	46.0317
email	0.0004	0.1239	0.0008	0.1856	0.0016	0.2295	0.0022	0.2649	0.0030	0.3012
conflict	0.0052	15.5628	0.0098	14.3276	0.0149	13.6799	0.0195	14.3760	0.0255	15.2124

Table V. Average updating time (in milliseconds)

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	whole
phone	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0011	0.0011	0.0011	0.0013	0.0011	0.0012	0.0015	0.0012	0.0018	0.0012
arxiv	0.0007	0.0006	0.0006	0.0006	0.0006	0.0007	0.0007	0.0007	0.0006	0.0007	0.0008	0.0008	0.0009	0.0008	0.0007	0.0007
elec	0.0011	0.0009	0.0010	0.0011	0.0011	0.0013	0.0013	0.0013	0.0014	0.0014	0.0015	0.0015	0.0017	0.0017	0.0053	0.0017
enron	0.0010	0.0009	0.0010	0.0012	0.0011	0.0013	0.0015	0.0014	0.0014	0.0015	0.0017	0.0014	0.0014	0.0016	0.0052	0.0018
facebook	0.0016	0.0017	0.0018	0.0020	0.0022	0.0022	0.0023	0.0024	0.0025	0.0028	0.0029	0.0030	0.0031	0.0032	0.0074	0.0025
lastfm	0.0007	0.0006	0.0006	0.0007	0.0007	0.0007	0.0007	0.0007	0.0007	0.0008	0.0008	0.0008	0.0008	0.0008	0.0015	0.0008
email	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	0.0006	0.0006	0.0006	0.0006	0.0006	0.0006	0.0006	0.0006	0.0016	0.0007
conflict	0.0014	0.0011	0.0011	0.0014	0.0012	0.0013	0.0015	0.0024	0.0020	0.0018	0.0018	0.0018	0.0019	0.0017	0.0050	0.0023

windows increases from θ to 2θ, and merged to become θ windows again. The result demonstrates that our equal-weight window model is suitable and efficient for update maintenance.

D. Scalability

In this experiment, we tested the scalability of our method. We generated two datasets, one from the `phone` graph and the other by a graph generator.

Synthetic `phone` graphs. We first generated synthetic phone call datasets based on the `phone` graph. The `phone` graph covers a span of 150 days, and we generated 5 synthetic graphs based on the statistics of the `phone` graph to cover 300, 450, 600, 750, and 900 days. For the synthetic graphs, we used the first 150 days as in the original `phone` graph, and for each day starting the 151-st day and 900-th day, we randomly replicated the records of one day taken from the original `phone` graph.

We computed the θ weighted graphs from the `phone` graph and the 5 synthetic phone graphs, by varying θ from 10 to 50. Figure 10 reports the construction time of the θ weighted graphs, which increases linearly as the size of temporal graph increases. The increase in the value of θ only increases the construction time slightly.

Figure 11 shows the space requirement for the θ weighted graphs for the 6 phone graphs. The result shows that the total size of the θ weighted graphs is only a small percentage of the size of the original temporal phone graph, and the percentage further decreases for larger original phone graphs. This result shows that our model can be an effective concise representation of a temporal graph as its size increases over time. The information loss of our method is also small, which is very similar to the results of the `phone` graph reported in Figures 6, 8 and 9.

We also computed the θ-weighted core number of each vertex in the graphs. Figure 12 reports the total running time, which again shows good scalability as the sizes of the graphs increase.

Synthetic power-law graphs. We used a synthetic power-law graph generator [27] to generator 2 sets of temporal graphs. The first set varies the average vertex degree $d_{avg}(v, \mathbb{G})$ from 100 to 400, while fixing $T_{\mathbb{G}} = 100,000$. The second set varies the number of time instances $T_{\mathbb{G}}$ from 50,000 to 400,000, while fixing $d_{avg}(v, \mathbb{G}) = 200$. We set $|\mathcal{V}| =$ 200,000.

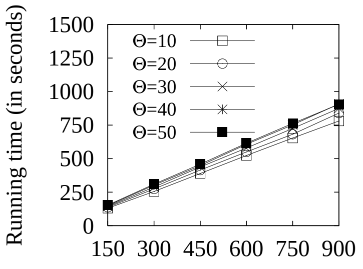


Figure 10. Construction time of θ weighted graphs for synthetic phone graphs

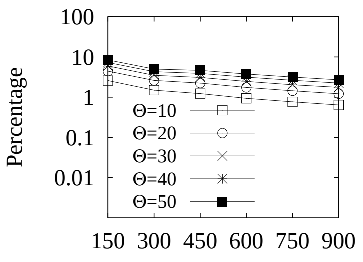


Figure 11. The total size of the θ weighted graphs compared with the original synthetic phone graphs

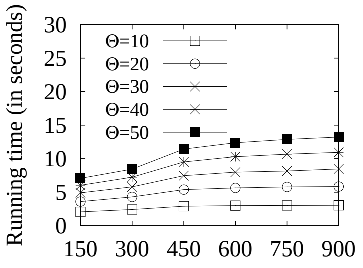
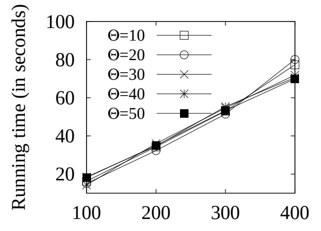
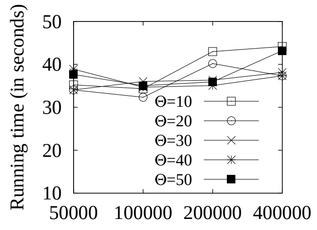


Figure 12. Computation time for θ-weighted core decomposition for synthetic phone graphs

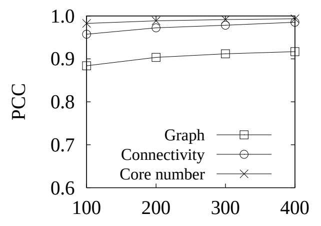


(a) Effect of $d_{avg}(v, \mathbb{G})$

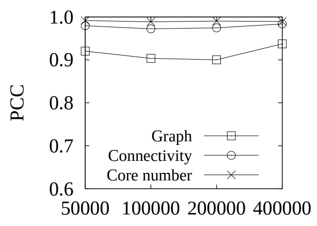


(b) Effect of $T_{\mathbb{G}}$

Figure 13. Construction time of θ weighted graphs for synthetic power-law graphs

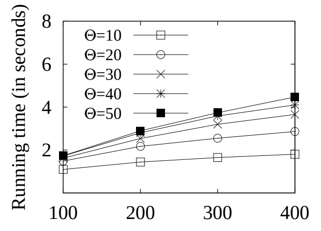


(a) Effect of $d_{avg}(v, \mathbb{G})$

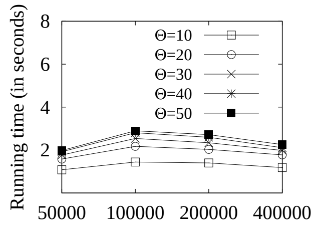


(b) Effect of $T_{\mathbb{G}}$

Figure 14. PCC between G'_W and G_W for synthetic power-law graphs ($\theta = 30$)



(a) Effect of $d_{avg}(v, \mathbb{G})$



(b) Effect of $T_{\mathbb{G}}$

Figure 15. Computation time for θ-weighted core decomposition for synthetic power-law graphs

Figure 13 shows that the time taken to construct the θ weighted graphs, for θ from 10 to 50. The construction time increases linearly as the number of edges in the temporal graph increases, while the effect of different values of $T_{\mathbb{G}}$ on the construction time shows no clear pattern.

We also computed G_W and G'_W for 100 randomly generated windows as in the experiment in Section V-A. We also computed the PCC values for the following: (1) between G'_W and G_W, (2) between the connectivity computed from G'_W and that computed from G_W, and (3) between weighted core numbers computed from G'_W and that computed from G_W. We report the PCC values in Figure 14. The result shows that all the above three types of PCC values are high, which further confirms the conclusions drawn from the results in Section V-A and verifies that our method is scalable.

We also computed the θ-weighted core number of each vertex in the graphs. Figure 15(a) shows that when $d_{avg}(v, \mathbb{G})$ increases, the running time increases sub-linearly. Figure 15(b) shows that the value of $T_{\mathbb{G}}$ does not affect the running time significantly. When $T_{\mathbb{G}}$ increases from 100,000 to 400,000, the running time even decreases, mainly because the graphs become more sparse in each window as $T_{\mathbb{G}}$ increases.

VI. Related Work

In this section, we discuss related work on temporal graphs and classical window models.

A. Temporal Graphs

Many works have been done on temporal graphs, also called time-varying graphs, or timetable graphs. Most of them [28], [2], [3], [29], [4], [30], [31], [32], [7], [6] are related to temporal paths. Temporal paths have been applied to study the connectivity of a temporal graph [2], the information latency in a temporal network [3], small-world behavior [5], and to find temporal connected components [33], [32]. Temporal paths have also been used to define metrics for temporal network analysis, such as temporal efficiency and temporal clustering coefficient [31], [32], and temporal betweenness [30] and closeness [4], [30]. Most of the existing works were focused on concepts and measures for studying temporal graphs, while computational issues were ignored. Among these works, only [6], [7] discussed algorithms for computing temporal paths. In [34], an index technique was proposed to answer temporal path queries, but the indexing method is not scalable and its efficiency was only verified on small datasets. In [35], minimum spanning tree is defined in temporal graphs. In [36], temporal graphs are used to model users' long-term and short-term preferences, which is useful for recommendation. Readers can also refer to more comprehensive surveys on temporal graphs [28], [1], [37].

There are also some works on storing temporal graphs in a compact way [38], [39], [40]. In [38], a compressed suffix array strategy was proposed to store temporal graphs. In [39], two data structures, compact adjacency sequence and compact events ordered by time, were proposed to represent temporal graphs. However, all these methods need to store each temporal edge. The performance of these methods is not better than the *gzip* compression.

B. Window Models

Various window models [8], [10], [9] were proposed for mining frequent patterns from data streams. There are three types of time window models: *landmark window model*, *sliding window model*, and *damped window model*. In the landmark window model [9], there is a specific time point called landmark point. The analysis is done on the window between the landmark point and the present. When there is no landmark point, the analysis is on the complete window. In the sliding window model [10], the size of the window is fixed. Let s_w denote the size of the sliding window, t_τ denote the current time. Then, the analysis is on the window $[t_\tau - s_w, t_\tau]$, and the window slides as time goes on. Data that are older than time $(t_\tau - s_w)$ are discarded. In the damped time window model (also called titled window model) [8], the more recent windows are at a finer granularity, while the older windows are at a coarser granularity. It is based on the assumption that recent data are more valuable than older ones, and the importance of data decreases exponentially as time goes on. Our window model is also a damped time window model.

VII. Conclusions

We proposed a novel time window model, called *equal-weight damped time window*, for processing massive growing temporal graphs. Our model allows users to set the number of windows to trade off between the required space and the information loss. Based on this model, we presented two applications, connectivity analysis and core community, to analyze the temporal graph. We conducted comprehensive experiments to verify the usefulness and efficiency of our method for analyzing large temporal graphs. We also showed that our method supports efficient dynamic update maintenance.

References

[1] P. Holme and J. Saramäki, "Temporal networks," *CoRR*, vol. abs/1108.1780, 2011.

[2] D. Kempe, J. M. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 820–842, 2002.

[3] G. Kossinets, J. M. Kleinberg, and D. J. Watts, "The structure of information pathways in a social communication network," in *KDD*, 2008, pp. 435–443.

[4] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Phys. Rev. E*, vol. 84, p. 016105, 2011.

[5] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora, "Small-world behavior in time-varying graphs," *Physical Review E*, vol. 81, no. 5, p. 055101, 2010.

[6] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *PVLDB*, vol. 7, no. 9, pp. 721–732, 2014.

[7] B.-M. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 2, pp. 267–285, 2003.

[8] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, "Multi-dimensional regression analysis of time-series data streams," in *VLDB*, 2002, pp. 323–334.

[9] C. Perng, H. Wang, S. R. Zhang, and D. S. P. Jr., "Landmarks: a new model for similarity-based pattern querying in time series databases," in *ICDE*, 2000, pp. 33–42.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1794–1813, 2002.

[11] J. Lai, C. Wang, and P. S. Yu, "Dynamic community detection in weighted graph streams," in *SDM*, 2013, pp. 151–161.

[12] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas, "Dynamic interaction graphs with probabilistic edge decay," in *ICDE*, 2015, pp. 1143–1154.

[13] J. Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, no. 4, pp. 229–239, 1980.

[14] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," in *LATIN*, 2000, pp. 88–94.

[15] E. D. Demaine, G. M. Landau, and O. Weimann, "On cartesian trees and range minimum queries," *Algorithmica*, vol. 68, no. 3, pp. 610–625, 2014.

[16] R. Andersen and K. Chellapilla, "Finding dense subgraphs with size bounds," in *WAW*, 2009, pp. 25–37.

[17] S. Khuller and B. Saha, "On finding dense subgraphs," in *ICALP*, 2009, pp. 597–608.

[18] M. Eidsaa and E. Almaas, "s-core network decomposition: A generalization of k-core analysis to weighted networks," *Physical Review E*, vol. 88, no. 6, p. 062819, 2013.

[19] A. Garas, F. Schweitzer, and S. Havlin, "A k-shell decomposition method for weighted networks," *CoRR*, vol. abs/1205.3720, 2012.

[20] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense implicit communities in the web graph," *TWEB*, vol. 3, no. 2, 2009.

[21] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *VLDB*, 2005, pp. 721–732.

[22] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Trawling the web for emerging cyber-communities," *Computer Networks*, vol. 31, no. 11-16, pp. 1481–1493, 1999.

[23] G. Kortsarz and D. Peleg, "Generating sparse 2-spanners," *J. Algorithms*, vol. 17, no. 2, pp. 222–236, 1994. [Online]. Available: http://dx.doi.org/10.1006/jagm.1994.1032

[24] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.

[25] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *SIGMOD*, 2014, pp. 625–636.

[26] V. D. Blondel, M. Esch, C. Chan, F. Clérot, P. Deville, E. Huens, F. Morlot, Z. Smoreda, and C. Ziemlicki, "Data for development: the D4D challenge on mobile phone data," *CoRR*, vol. abs/1210.0137, 2012.

[27] D. A. Bader and K. Madduri, "GTgraph: A synthetic graph generator suite," *Atlanta, GA, February*, 2006.

[28] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[29] V. Kostakos, "Temporal graphs," *Physica A: Statistical Mechanics and its Applications*, vol. 388, no. 6, pp. 1007–1023, 2009.

[30] N. Santoro, W. Quattrociocchi, P. Flocchini, A. Casteigts, and F. Amblard, "Time-varying graphs and social network analysis: Temporal indicators and metrics," *CoRR*, vol. abs/1102.0629, 2011.

[31] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Temporal distance metrics for social network analysis," in *WOSN*, 2009, pp. 31–36.

[32] ——, "Characterising temporal distance and reachability in mobile and online social networks," *Computer Communication Review*, vol. 40, no. 1, pp. 118–124, 2010.

[33] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, "Components in time-varying graphs," *CoRR*, vol. abs/1106.2134, 2011.

[34] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015, pp. 967–982.

[35] S. Huang, A. W. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *SIGMOD*, 2015, pp. 419–430.

[36] L. Xiang, Q. Yuan, S. Zhao, L. Chen, X. Zhang, Q. Yang, and J. Sun, "Temporal recommendation on graphs via long- and short-term preference fusion," in *KDD*, 2010, pp. 723–732.

[37] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. D. Zaroliagis, "Timetable information: Models and algorithms," in *ATMOS*, 2004, pp. 67–90.

[38] N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez, "A compressed suffix-array strategy for temporal-graph indexing," in *SPIRE*, 2014, pp. 77–88.

[39] D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," *Inf. Syst.*, vol. 51, pp. 1–26, 2015.

[40] G. de Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodríguez, "Compact data structures for temporal graphs," in *DCC*, 2013, p. 477.