On Compressing Temporal Graphs

Conference Paper · May 2022

CITATIONS
0

READS
306

4 authors:

Panagiotis Liakos
National and Kapodistrian University of Athens
22 PUBLICATIONS **130** CITATIONS
SEE PROFILE

Katia Papakonstantinopoulou
National and Kapodistrian University of Athens
14 PUBLICATIONS **59** CITATIONS
SEE PROFILE

Theodoros Stefou
National and Kapodistrian University of Athens
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Alex Delis
National and Kapodistrian University of Athens
172 PUBLICATIONS **1,830** CITATIONS
SEE PROFILE

On Compressing Temporal Graphs

Panagiotis Liakos
University of Athens
GR15703, Athens, Greece
p.liakos@di.uoa.gr

Katia Papakonstantinopoulou
Athens Univ. of Economics and Business†
GR10434, Athens, Greece
katia@aueb.gr

Theodore Stefou
University of Athens
GR15703, Athens, Greece
cs2190002@di.uoa.gr

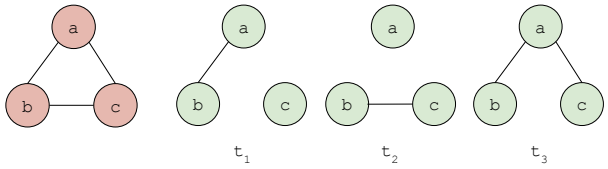
Alex Delis
University of Athens
GR15703, Athens, Greece
ad@di.uoa.gr

***Abstract*—Contemporary data-systems empowering the daily human activity are routinely represented with graphs. During the last decade, the volume growth of such systems has been unprecedented. This hinders the timely analysis of the formed networks due to existing physical memory limitations and significant I/O overheads. Graph compression techniques have managed to reduce memory requirements and allow for representing such networks using a few *bits-per-edge*. Respective approaches offer succinct mappings for social, biological, and information networks while allowing for the efficient access of sought graph elements. Despite their success, such methods mostly focus on static graphs, and predominantly offer access to either a snapshot or an aggregated view of a network. In reality however, networks change over time and, in many instances, we are interested in capturing and studying this *evolution*. In this paper we propose a framework for compressing *emerging temporal graphs* based on a dual-representation which articulates both network structure and corresponding temporal information. We empirically establish properties exhibited by community-networks regarding their time aspect(s) and harness these features in our proposed representation. Our experimental evaluation demonstrates that our approach for compressing temporal graphs readily outperforms competing techniques, attaining compression ratios that are on average *around 60%* of the space required by state-of-the-art techniques. Moreover, our memory-efficient representation yields more than 70% faster graph compression and orders of magnitude quicker retrieval of graphs' elements, especially when it comes to large-scale networks. Finally, our framework is the first effort we are aware of, that considers *actual time* instead of time steps. This helps us attain better control for the size of our representation and reap further memory savings.**

I. Introduction

The emergence of Big Data, the constantly growing WWW space and the explosive growth of social networks have led to systems that routinely work with massive volumes of data modeled as graphs. This trend becomes more than evident by the impressive rate with which Google's web-index grows, going from a billion pages in 2000, to more than a trillion pages in 2008 and 100 trillion in 2016.[1] Similarly, Facebook, with over 2.8 billion monthly active users as of the 4^{th}-quarter of 2020,[2] stands out as the biggest social network worldwide.

Both mining and analysis of graphs that portray networks of the above size pose major challenges as we have to circumvent costly I/O operations without skyrocketing the costs of necessary hardware. To this end, several graph compression approaches have been proposed and offer space-efficient in-memory representations of graph data. These methods allow for querying neighbors of nodes without the need for decompression, and thus, they substantially facilitate the analysis of large-scale networks. However, graphs, and consequently graph compression approaches of today, predominantly depict a *static view of networks* which can be a snapshot of a single moment or an aggregated time period [1]. Instead, the relationships among the entities of networks are rarely persistent over time. For example, phone calls between individuals last only for a finite interval of time and are often recurrent.



(a) Static graph. (b) Temporal graph.

Fig. 1: A static graph (a) lacks temporal information and flattens relationships between pairs of nodes in single edges. A temporal graph (b) portrays the evolution of the network through time (t_1 to t_3) more accurately.

Figure 1 depicts two representations of phone calls made between three individuals during a period of three time steps: t_1, t_2, t_3. The representation of Figure 1(a) offers an aggregated static view of the network showing that a call was made between nodes `a` and `b`, `a` and `c`, and `b` and `c`. In contrast, the representation of Figure 1(b) additionally captures the *evolution of the network*. More specifically, we see that a call was placed between nodes `a` and `b` at step t_1, `b` and `c` at step t_2, and nodes `a` and `b`, and `a` and `c` at step t_3. The static representation of Figure 1(a) in which the relationships between pairs of nodes are "flattened" in a single edge is clearly an oversimplifying approximation of the actual network. Attempting to uncover insights based on such a confined representation for networks that *evolve* over time is both *inaccurate* and *limited*.

Using a phone-call network again as an example, we may be interested in tracking the evolution of the groups a person belongs to, by applying community detection on a weekly basis [2], [3]. Similarly, in the case of the Web graph, we may wish to retrieve the historical state of the connectivity

†Theory, Economics and Systems Laboratory, Dept. of Informatics.

[1]https://www.google.com/search/howsearchworks/

[2]https://investor.fb.com/investor-news/press-release-details/2021/Facebook-Reports-Fourth-Quarter-and-Full-Year-2020-Results/default.aspx

between websites and measure how their PageRank values change over time [4]. Also, we are often interested in spotting atypical behavior, e.g., uncovering attacks by analyzing traffic in computer networks or identifying influential scientific articles attracting a lot of citations over a short period of time [5]. All the aforementioned use-cases point out to the fundamental need for space-efficient in-memory representations of *temporal* graphs that allow for quick neighbor-query execution. Indeed, the focus in the field of graph compression has recently shifted towards approaches that consider temporal graphs [6], [7], [8], [9]. Such approaches overcome the overhead of representing a snapshot of the graph for each time step by considering the *aggregated structure* of the respective network and focusing on the *changes* occurring over time. In this way, one only needs to store temporal information about recurring relationships between nodes, instead of wasting space to represent duplicate pairs of nodes [8]. In addition, sorting lists of relationships with regards to time, allows for saving space by storing the *gaps* between timestamps, instead of the actual values [7].

Despite the above efforts, the compression potential of real-world time evolving graphs is not yet fully utilized. Our conjecture is that properties temporal networks exhibit, regarding both their *structure* and occurring *edge updates* during their lifetime, can be further exploited to provide representations offering much enhanced memory-savings. Moreover, prior approaches focus on *time steps* instead of using an accurate *timestamp* for each event. These time steps reflect changes in the network, and the intervals between them are not necessarily *evenly-spaced*. However, in several cases we are interested in the exact time in which an event occurred, e.g., we may wish to know whether a call was placed between two specific individuals during the last hour. Furthermore, irregular intervals of time steps often hinder the analysis of network *evolution*. Therefore, we need representations that use timestamps, or equivalently evenly spaced time steps, and provide users with the flexibility to i) query the graph by specifying directly the *time of interest* and ii) apply their desired time aggregations, e.g., hourly or daily, to save further space.

In this paper, we propose a flexible, memory-efficient temporal graph compression framework, termed CHRONOGRAPH. CHRONOGRAPH breaks the storage of a temporal graph into two distinct representations: i) the network structure and ii) the relationships' timestamps. Restoring the neighbors of each node along with their respective timestamp becomes feasible as the two representations share the *same* ordering. The distinct representation of the structure allows for exploiting similarities between nodes as far as their neighbors are concerned. In this manner, we can reference a portion of the adjacency list of a node while we are storing the list of another. Similarly, by keeping a separate representation of the timestamps we reveal some newly discovered properties that offer impressive memory savings if properly exploited.

CHRONOGRAPH applies *gap* and *bit encodings* to achieve a compact timestamps' representation. Our framework also applies compression of multiple occurrences of edges, intervalisation, and referentiation, to efficiently store the network structure. CHRONOGRAPH allows for querying a node's neighbors –or the entire graph– at a user-specified time interval. Moreover, we can specify the time aggregation to be stored when compressing the graph, as CHRONOGRAPH can use any desired resolution available. Evidently, using a 1-*hour* resolution is more space-efficient than using a 1-*second* resolution.

We summarize here the contributions of our proposed CHRONOGRAPH framework. In particular, we:

- investigate, uncover and report the *power law distribution* that the timestamps of a node's neighbors follow under certain orderings, and exploit this property to represent them more *compactly*.
- significantly reduce the *overall memory requirements* for representing temporal graphs, in comparison to the state-of-the-art approaches. We show that our optimized structures sometimes need *less than half* of the space required by earlier approaches, offering improvements between 15.11% and 61.23% for any network investigated in our evaluation.
- offer memory savings that are more impressive when considering that we represent and allow for retrieving the *actual timestamps* of events instead of just using time steps. Our framework allows for querying at specific time intervals and enables more accurate tracking of the evolution of networks through time.
- allow for further reducing the resulting size, by specifying a desired time resolution / granularity and applying the respective aggregation of timestamps in intervals. The lower the desired resolution is set, the smaller the produced size of the compressed graph becomes.
- outperform state-of-the-art methods in terms of speed as far as both compression and access on the graph's elements are concerned. Our compression time is more than 70% faster, whereas query answering is usually quicker by orders of magnitude and scales much better for large graphs.

II. RELATED WORK

Our work lies in the intersection of compressed static graph representations and querying graphs that evolve over time. In this regard, we first outline here pertinent aspects of these two areas: i) state-of-the-art space-conscious representation of static real-world graphs and ii) systems capable of storing and querying series of graphs that handle the complexity and subtlety inherent in dynamic graphs efficiently. Additionally, we review earlier temporal graph compression approaches to point out their strengths and weaknesses and clarify how our proposed approach differs.

A. Static Graph Compression Approaches

The need for compact representation of graphs emerged with the explosion of the size of the worldwide web. In the last twenty years, graph compression has been a very active research area, particularly with regard to web and social network graphs. Some studies focus on computing the entropy of *random graphs*, using models such as Barabási–Albert [10]

or Watts–Strogatz [11], in an attempt to derive the compressibility of each such class of graphs. Most approaches, however, target graphs resulting from human activity and propose algorithms that exploit *empirically observed properties* of these *real-world graphs* to offer a good space/time trade-off [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. A pertinent review and classification of such static graph compression methods can be found in [22]. Among the techniques surveyed, the Webgraph framework [14] stands out. In this, several compression mechanisms were combined to exploit various types of redundancy encountered in web graphs. In spirit, our proposal follows the same rationale as that of [14] while seeking to create a compression method appropriate for *temporal graphs*. More specifically, several compression techniques proposed in the literature can be useful in the context of graphs that evolve over time as well, if properly applied. We employ -among others- (i) gap encoding [12], (ii) referentiation [13] (iii) intervalisation [14], and (iv) ζ codes [24].We will show that the aggregated network structure of a temporal graph that results when considering the edges of its entire lifetime, exhibits structural properties that allow us to obtain a high compression ratio. Moreover, we furnish empirical evidence that we can reduce the overall memory requirements for the representation of time evolving graphs. This is accomplished by exploiting pertinent temporal properties that such graphs demonstrate.

B. Querying Historical Graphs

Prior research efforts also dealt with the challenge of efficiently querying historical graph data [25], [26], [27]. These earlier approaches do consider compact representations that exploit the redundancy exhibited by temporal graphs. However, the focus is mostly on efficient query processing and support for complex queries on evolving graphs, rather than space-efficient representations. Our approach is different as our main focus is on optimizing memory usage to allow for handling large-scale graphs with *minimal* resources. We seek to offer neighbor-query-friendly compression of temporal graphs, to allow the execution a wide range of applications, including community detection, PageRank calculation and anomaly detection, in settings where it was not feasible before.

C. Temporal Graph Compression Approaches

Several compression approaches for temporal graphs consider a representation based on a chronological log of events. The Adjacency Log of Events (EveLog) [28] uses two separate lists per vertex. The first represents the time steps and is compressed with gap encoding. The second holds the respective edges and is compressed with a statistical model. CHRONOGRAPH also builds two structures to represent temporal graphs but we model our representation based on adjacency lists. Moreover, we apply several additional compression techniques for each structure and do accomplish notable improvements with regards to both space efficiency and access time.

Compact-Events-ordered-by-Time (CET) [28] also models its representations as a chronological log of events. CET uses a variant of the Wavelet Tree [29] called Interleaved Wavelet Tree [28]. The latter is a compact structure that allows for retrieving the number of occurrences of multidimensional symbols in logarithmic time. In this way, CET may determine how many times an edge appears in the graph. Compact-Adjacency-Sequence (CAS) [28] uses a Wavelet Tree to represent activation or deactivation of edges, and uses a sorted by vertice sequence in the log. In [7], Caro et al. propose a 4-dimensional data structure, termed Compressed k^d-trees (ck^d-trees) and based on quadtree [30]. Two dimensions are used to represent the structure of the graph whereas the remaining two help represent the time steps in which updates occurred. The 4-dimensional representation is compressed using a generalization of k^2-trees [16] to a d-dimensional space. The ck^d-trees method achieves considerable space-savings at the expense of time access in sparse temporal graphs.

Nelson et al. [9] consider an aggregated adjacency matrix that represents all the edges appearing in a temporal graph regardless of the time step. They subsequently compress each row of this aggregated matrix using compressed binary trees [31]. For every leaf node of this tree that represents an edge, a new tree is created to represent the time dimension of the edge. As the number of time steps in a temporal graph can be quite large, an alternating compressed binary tree technique is introduced, that is capable of efficiently representing runs of ones as well as zeros.

Time-interval Log per Edge (EdgeLog) [7] represents temporal graphs with the use of adjacency lists. For each neighbor of a node, EdgeLog maintains a sorted inverted list that stores the time steps at which an update concerning this neighbor occurred. Gap encoding is first applied to each inverted list, which is further compressed using a suitable variable-length encoding (e.g., PForDelta, Simple16, Rice codes). Clearly, EdgeLog is suitable for temporal graphs with a few edges per node that get frequently updated, which is often not the case in real-world graphs. CHRONOGRAPH also uses an adjacency list representation and applies gap encoding to reduce the size required to store the timestamps of the neighbors of each node. However, we consider gaps between the entire list of neighbors for each node, instead of the inverted list associated with a single edge. Moreover, we apply bit encodings on the resulting gaps that help us achieve a significantly improved trade-off between compressed size and access time.

In Table I we depict the approaches that are most applicable to our task. We see that, as far as temporal graph types are concerned, all earlier approaches provide compact representations for incremental, point, and interval temporal graphs. We will show, however, that our framework is significantly more space-efficient and faster. Table I also depicts that previous methods represent temporal graphs using time steps. Our CHRONOGRAPH can additionally store timestamps, and allows for graph queries targeting a particular time interval. Furthermore, we provide the flexibility of limiting the resulting size of the graph by specifying a smaller desired time granularity, as we can reduce the required space through storing aggregated temporal information.

TABLE I: Summary of features: state-of-the-art surveyed temporal graph compression approaches vs. CHRONOGRAPH.

		EveLog	EdgeLog	CET	CAS	ck^d-trees	T-ABT	CHRONOGRAPH
Type	**Incremental**	✓	✓	✓	✓	✓	✓	✓
	Point	✓	✓	✓	✓	✓	✓	✓
	Interval	✓	✓	✓	✓	✓	✓	✓
Time	**Time steps**	✓	✓	✓	✓	✓	✓	✓
	Timestamps							✓
	Aggregations							✓

III. PRELIMINARIES

A. Features of Temporal Graphs

At this time, an ever growing number of network systems and applications render services for which the *evolving nature* of their underlying data infrastructure is critical. Such artifacts are naturally modeled and studied via *temporal graphs* [32], i.e., graphs which change over time. Such graphs are also known as *dynamic* [33], *evolving* [34] and *time-varying* [35]. These graphs display sets of edges that receive changes over time as edges are constantly either added or removed.

Time-related information in temporal graphs pertains to events of edge creation or deletion and consists of the distinct points in time when edges appear or disappear from the graph. Each such piece of information is referred to as a *timestamp*. There are three distinct classes of graphs when it comes to managing timestamps in time evolving graphs: *point graphs, contact graphs* and *incremental graphs*.

- In *point graphs*, each edge between nodes u and v that is created at time t is represented by a triplet $e = (u, v, t)$.
- In *contact graphs*, the time-intervals in which each edge remains *active* are represented. A contact between nodes u and v that started at time t and lasted for δt is denoted by the quadruplet $e = (u, v, t, \delta t)$. Here, δt is expressed in appropriate temporal units.
- In *incremental graphs*, only additions of edges are allowed. Deletions are not permissible.

B. Compression of Real-world Graphs

Effective approaches for compressing *static* real-world graphs [14], [22] have exploited the redundancy derived by two properties frequently observed in various types of such graphs: the *locality of reference* and the *copy property*. In particular:

- the *locality of reference* describes the fact that nodes that are "close" in the network tend to have *similar labels*. This is obviously the case in web graphs, as it is a natural consequence of assigning labels to the web pages according to the lexicographical order of their URL. The same can also be observed in other types of graphs created by human activity including social network and citation graphs, after applying on their nodes a proper reordering algorithm [36], [37].
- the *copy (or similarity) property* describes the fact that nodes found in proximity in a network tend to have similar sets of neighbors. If we consider for instance a social network in which two individuals are friends, they will very likely have many mutual friends.

These two properties induce different types of redundancy in the graph. For every such type, there is also a proper encoding that helps minimize the compressed size of the graph. Note that since real-world graphs consist of millions or even billions of nodes, a sequence of node labels is expected to consist of very large integers and we need 8 bytes for storing each one of them. Due to the locality of reference, however, many of the labels are expected to have *similar* values. Therefore, we can use *gap encoding* [12] to obtain sequences of much smaller values and compress such sequences using variable-length instantaneous codes that assign shorter representations to smaller integers [24].

IV. OVERVIEW OF CHRONOGRAPH

A. Analyzing Timestamps

Designing an efficient compression technique for temporal graphs calls for analyzing time-related aspects as they appear in such graphs, highlighting their properties and selecting appropriate encodings for their representation. While working with the dataset outlined in Table III, we observed that the timestamps of the edges of each node display similar values. This is also true for many real networks outside our dataset, for which the intervals between edge creations follow a a power-law distribution with notable heavy tail [38], [39], [40], [41]. There are two sources of *locality* in temporal graphs. Consider for example the `Wiki-Edit` graph (Table III). When a user updates an article, with high probability she will continue her activity in the near future: she may either update the same article again, resulting to *locality with a specific neighbor* (i.e., contacts of similar timestamps with this neighbor), or update some other article, resulting to *locality with various neighbors* (i.e., contacts of similar timestamps with various neighbors). To expose this property, we examined three different gap strategies: the gap of each timestamp from (i) the minimum one, (ii) the most frequent one, and (iii) the previous one.

Figure 2 illustrates the cumulative frequency of the timestamp gaps of each of the above three strategies for the `Yahoo` graph. For the *frequent* and *previous* strategies we use a mapping of integers to natural numbers, as the timestamps are ordered according to the labels of their associated nodes, and these strategies may produce negative gaps. We see that the *previous* strategy stands out, as the frequency of small integers is much higher there compared to the other two strategies. This high concentration of small values becomes more evident in Figure 3, where we derive the distribution of timestamp gaps, and show results we obtain with additional real-world graphs. We observe that *all graphs* of our dataset exhibit *skewed distributions* that favor compression. The concentration of small

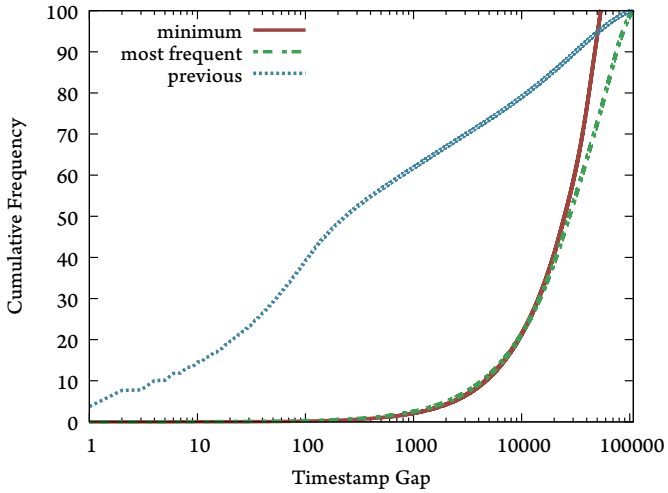


Fig. 2: Empirical study of the timestamps of Yahoo graph using 3 different gap strategies and resolution of a second.

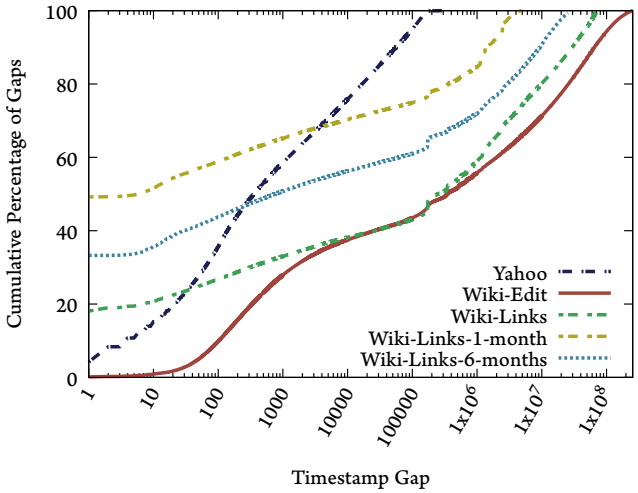


Fig. 3: Empirical study of the distribution of timestamp gaps when using the *previous* strategy for various real-world graphs.

values is higher for Yahoo, with 40% of its timestamp gaps being shorter than 100 seconds. This percentage is slightly above 20 for Wiki-Links and below 10 for Wiki-Edit. The reason is that the time span of Yahoo is much shorter than that of Wiki-Edit and Wiki-Links, which represent time evolution of several years. This becomes clearer when observing the distributions of two subgraphs of Wiki-Links in Figure 3, with time spans lasting one month and six months, respectively. We can see that as we shorten the time span, we come up with distributions that are more favorable for compression. The skewness that is evident when we apply the *previous* strategy, even for graphs that span many years, is a property our ChronoGraph builds upon, to reduce the memory required to represent time-related information.

The huge number of distinct timestamps in graphs with large time spans does not only hinder compression efforts; it also makes the visualization and processing of the results a rather challenging task. However, we are almost never interested in querying the graph about specific timestamps. In contrast, we are particularly interested in changes that occurred in some specific time interval. Therefore, we are actually keen in *aggregating* the timestamps within some predefined interval. In this context, we also investigated various levels of aggregation, depending on the size of each dataset.

Figure 4 shows the distribution of timestamp gaps in the Yahoo graph for three levels of aggregation: (i) per hour, (ii) per minute and (iii) per second (no aggregation). The necessary mapping of integers to natural values results in the formation of two lines for each aggregation, one for the positive gaps (upper) and one for the negative gaps (lower). Nevertheless, we observe that the resulting gaps are power-law distributed. Also, the three upper (or lower, respectively) lines actually follow the exact same distribution. The values are just divided by 60 as we move towards larger aggregations, so the distribution tail shifts to the left.

The nature of the redundancy due to time-related information found in temporal graphs is of paramount importance when it comes to the delivery of our approach. Having uncovered respective properties in the real-world graphs of our dataset and associated them with intuitive justifications which we expect to be true for networks generated by human activity, we can now build upon these properties to create space-efficient representations. Naturally, the performance of ChronoGraph will depend on the existence of these properties, as is the case with earlier approaches [6], [7], [8], [9] that also base their effectiveness on temporal locality, even though they do not identify the properties we uncover here. This is also true for non-temporal graph compression approaches [12], [13], [14], [16], [15], [19], [17], [18], [22], [23] that perform better when the dataset exhibits empirically observed properties such as the locality of reference and the copy property. The key novelty of our framework is that

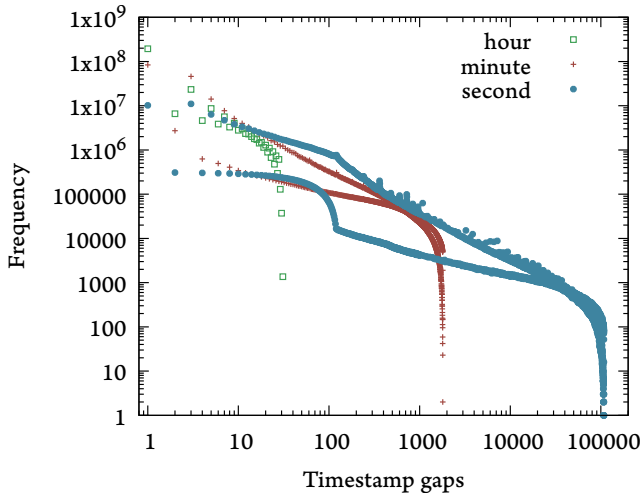


Fig. 4: Distribution of the timestamp gaps of the Yahoo graph for different levels of time granularity.

it entails two key yet separate parts to fully exploit the compression potential of temporal networks: the compression of timestamps and the compression of the network structure.

B. Compression of Timestamps

Our CHRONOGRAPH framework uses a dedicated structure holding the timestamps of relationships to compress the temporal information of a graph. For each node, we maintain a list of timestamps that are associated with a neighbor of a node. The order of the timestamps is defined by the labels of the nodes and the values of the timestamps. We empirically observed that when using this order, the gaps between timestamps of a node's neighbors follow a power law distribution, as depicted in Figure 3. Therefore, if the timestamps associated with the neighbors of a node u are $T(u) = (t_1, t_2, \ldots, t_n)$, we first calculate their gap sequence as $(t_1 - t_{min}, t_2 - t_1, \ldots, t_n - t_{n-1})$, where t_{min} is the smallest timestamp in our temporal graph. As we can see in the second column of Table II, such a sequence may contain negative integers. However, the set of codes we will use for compressing the gaps, cannot handle negative numbers. Therefore, we apply the following mapping to obtain a natural number to all but the first element of our sequence, which is not negative by definition:

$$f(x) = \begin{cases} 2x, & \text{if } x \geq 0 \\ 2|x| - 1, & \text{otherwise} \end{cases} \quad (1)$$

Eq. (1) ensures that gaps with *small* absolute values will be mapped to *small* natural numbers that favor compression. The third column of Table II contains the natural numbers that we come up with in our example when using Eq. (1). We compress this list of natural numbers using ζ codes [24], a set of flat codes that are particularly suitable for integers with a power law distribution in a certain exponent range. Below, we first offer the definitions of baseline unary and minimal binary codings. Secondly, based on these two required definitions, we designate ζ codes.

Let x denote a positive integer, b its binary representation and l the length of b:

1) *Unary coding*: the unary coding of x consists of $x - 1$ zeros followed by a 1, e.g., the unary coding of 2 is 01.
2) *Minimal binary coding* over an interval [24]: consider the interval $[0, z - 1]$ and let $s = \lceil \log z \rceil$. If $x < 2^s - z$ then x is coded using the x-th binary word of length $s - 1$ (in lexicographical order), otherwise, x is coded using the $(x - z + 2^s)$-th binary word of length s. As an example, the minimal binary coding of 8 in $[0, 56 - 1]$ is 010000, as $8 = 2^{\lceil \log 56 \rceil} - 56$ and therefore we need the $8 - 56 + 2^6 = 16$-th binary word of length 6.
3) ζ *coding* with parameter k [24]: given a fixed positive integer k, if $x \in [2^{hk}, 2^{(h+1)k} - 1]$, its ζ_k-coding consists of $h + 1$ in unary, followed by a minimal binary coding of $x - 2^{hk}$ in the interval $[0, 2^{(h+1)k} - 2^{hk} - 1]$. As an example, 16 is ζ_3-coded to 01010000, as $16 \in [2^3, 2^6 - 1]$, thus $h = 1$ and the unary of $h + 1 = 2$ is 01, and the minimal binary coding of $16 - 2^3$ over the interval $[0, 2^6 - 2^3 - 1]$ is 010000, as shown above.

Depending on the time span of a temporal graph, the distribution of timestamp gaps varies. Unfortunately, there does not exist a universal value of k that obtains the most succinct representation for any temporal graph [24]. However, our experimental evaluation discusses the impact of k and reports the codes that consistently work well.

C. Aggregations

The time granularity of a temporal graph may vary, depending on the domain. We usually expect that the exact second associated with a relationship is available in a temporal graph. However, such fine granularity is oftentimes not very useful in network analysis. Instead, typical use cases involve aggregated intervals of coarse granularity, e.g., on an hourly or daily level.

Earlier temporal graph compression approaches [6], [7], [8], [9] represent time using *evenly spaced* steps. Consequently, such approaches cannot perform meaningful aggregations. CHRONOGRAPH uses timestamps and offers the flexibility of specifying *desired granularity*. If the network analysis tasks at hand require fine granularity, we may use CHRONOGRAPH with the default configuration that does not perform any aggregations. When, however, we are interested in issuing queries using time intervals larger than the available granularity, we can specify the respective aggregation and obtain a much more memory-efficient graph representation. In particular, we use the quotient of the division of each timestamp with the desired aggregation expressed in seconds. Table II also shows the sequence we get when applying an hourly aggregation on the original timestamps. Clearly, the gaps that we come up with after applying aggregation are much more easily compressed, as the values fall into a much smaller space. Our experimental evaluation highlights this trade-off between granularity and space efficiency by comparing the memory requirements of temporal graphs when varying the applied level of aggregation.

D. Compressing the Network Structure

CHRONOGRAPH follows an adjacency list representation for the network structure, and maintains for each node a list of neighbors ordered according to their label. Due to *locality of reference* and *copy property*, this ordering brings about many opportunities for compression. In this section, we outline the 4 techniques we use to compress the lists of neighbors. The first technique is a novel contribution we suggest, whereas the other 3 have been earlier used for the compression of static graphs [13], [42], [14], [24].

1) Compression of Multiple Occurrences: Temporal graphs represent networks that evolve over time. In the case of *incremental* graphs, the changes exclusively involve insertions of new edges. Therefore, the aggregated graph that represents all the edges appearing in the network through its lifetime has no parallel edges. Nevertheless in *point* and *contact* graphs, it is common to encounter a specific edge several times. Thus, the respective representation with all the edges of the network can be modeled using a multigraph. To the best of our knowledge,

TABLE II: Gap encoding for the compression of timestamps associated with the neighbors of a node.

Without Aggregation	**Timestamps**	1209479772, 1209479933, 1209479965, 1209479822, 1209479825, 1209483450, 1209483446
	Gaps (Integers)	34637, 161, 32, -143, 3, 3625, -4
	Gaps (Natural)	34637, 322, 64, 285, 6, 7250, 7
Hourly Aggregations	**Timestamps**	335966, 335966, 335966, 335966, 335966, 335966, 335966, 335967, 335967
	Gaps (Integers)	10, 0, 0, 0, 0, 1, 0
	Gaps (Natural)	10, 0, 0, 0, 0, 2, 0

Node	Outdegree	Neighbors
1	16	2,3,3,3,5,6,7,8,9,11,12,13,14,17,17,33

(a) Neighbors' list of a node.

Multiple Intervals	Lengths	Gaps
(3,3,3), (17,17)	(3,3), (17,2)	(2,1), (13,0)

(b) Deduplication.

Intervals	Lengths	Gaps
(5,6,7,8,9), (11,12,13,14)	(5,5), (11,4)	(4,1), (0,0)

(c) Intervalisation.

Extra Nodes	Gaps
(2,33)	(1,30)

(d) Extra nodes.

Fig. 5: The neighbors' list of a node (a) and the steps of the different compression techniques of CHRONOGRAPH (b,c,d).

state-of-the-art static graph compression algorithms do not support multigraphs [14], [19], [18]. Consequently, one would be simply unable to represent the aggregated network structure of a temporal graph using an existing static graph compression approach.

We propose here a *deduplication* step, applied to the neighbors' list of each node, that enables CHRONOGRAPH to render support for multigraphs. More specifically, we iterate through the sorted list of neighbors of a node u, $N(u)$, and identify neighbors with more than one occurrence. For each identified node, we hold its value and the number of times it appears in the list. If the duplicate neighbors of node u are $D(u) = (d_1, d_2, \ldots, d_n)$ we will represent the sequence as $(u - d_1, d_2 - d_1 - 1, \ldots, d_n - d_{n-1} - 1)$. The first element of the ensued sequence may be negative, and so, we apply Eq. (1) to obtain a natural number. For example, Figure 5(a) shows the neighbors of node 1. In Figure 5(b), we see that neighbors 3 and 17 appear in the list three and two times, respectively. We only need to store the node label and the number of occurrences to represent this information. Additionally, we use gap encoding [12] to further reduce the memory requirements. Similarly, we can reduce the size required to represent the lengths of intervals by subtracting the minimum interval length, i.e., 2. As Figure 5(b) points out, we end up with the pairs $(2, 1)$ and $(13, 0)$, which we store using Elias γ coding [43].

We note that the presence of multiple occurrences of certain neighbors in the adjacency list of a node has a negative impact on the compression potential of the *copy property* (Section III-B). This phenomenon weakens the advantages of applying reference compression, as it becomes more difficult to identify candidate nodes that exhibit similarity. In the case of temporal graphs, we are interested in similarity expressed through both the lists of neighbors and the occurrences of neighbors, which is, in fact, rare. Therefore, the first step of the operation of CHRONOGRAPH is to remove all neighbors with multiple occurrences and compress them separately. Then, we are left with a list of remaining nodes and we examine whether we can find in other neighbor lists a good candidate to use as a reference.

2) Reference Compression: Similarities between neighbor lists of different nodes are frequent in real-world graphs. We can exploit such similarities by representing part of a node's neighbors through references to a particular, and preferably large, segment of the neighbors of a different node.

Reference compression is used in several forms in [13] and is theoretically analyzed in [42]. We represent the neighbors of a node u, $N(u)$, with respect to another node's neighbors, $N(v)$, as in [14] using:

- a sequence of $|N(v)|$ bits, termed the *copy* list specifying which of the neighbors of v are also neighbors of u and should be copied, and
- a list of the remaining neighbors of u, i.e., $N(u) \setminus N(v)$, called *extra* nodes.

Due to the locality of reference and the copy property, there are many *consecutive* nodes in neighbor lists which are common for nodes that are similar. Therefore, it is often the case that the copy list contains long runs of 1s and 0s. To further reduce the size of our representation by exploiting these empirically observed properties, we first split the copy list to blocks of alternating 1s and 0s. Then, we need to only store the number of the blocks and the size of every block.

We should point out that there is a trade-off between the compression ratio and access time. This trade-off depends on the choice of the *window* we consider when looking for the best reference of a node's neighbor list. The larger the value of this window, the better the compression we achieve, at the cost of slower compression and decompression. In this work, we adopt a window size of 7, as in [14].

3) Compression of Consecutive Neighbors: Due to the locality of reference property, we expect many of the remaining *ordered* neighbors of a node to be *consecutive*. Intervalisation [14] is a technique that isolates the consecutive nodes in a neighbors' list into *intervals*. We are interested in intervals whose length is larger than a minimum threshold, which we then compress in a way similar to our *deduplication* process. In this context, we only need to store the smallest node in each interval, along with the length of the interval. The other

nodes can be restored by starting with the smallest node and increasing by one the label of the last node we brought back, until we have restored as many nodes as the length specifies. Figure 5(c) shows the two intervals of consecutive neighbors we can find in the example of Figure 5(a). Instead of storing all neighbors, we just store the first one (5, and 11) and the length of the interval (5, and 4).

We reduce the necessary space for the smallest node in each interval through gap encoding, i.e., we store the differences between intervals. These differences are further reduced by the minimum interval length used plus one, as there must be at least one node between the end of an interval and the beginning of the next one [14]. Moreover, we reduce the size of the lengths of the intervals by storing how larger they are compared to the minimum interval length. Figure 5(c) shows that we end up with pairs $(4, 1)$ and $(0, 0)$ in our example. Finally, we use Elias γ coding [43] to store the differences between the intervals as well as the interval lengths. We set the minimum interval length to 4, as it is also suggested in [14].

4) Extra Neighbors: We reduce the space required for the remaining neighbors through gap encoding. Figure 5(d) shows the extra neighbors that are left uncompressed after applying all previous steps in the list of Figure 5(a). Using gap encoding, we end up with: $2-1=1$ and $33-2-1=30$. The resulting list of integers exhibits a power law distribution in a certain exponent range and we compress it effectively using ζ codes [24].

Algorithm 1: isNeighbor

```
1  Function isNeighbor (G, T, u, v, t_start, t_end)
2      start ← −1;
3      end ← −1;
4      neighbors ← G.neighbors(u);
5      for i=0; i < neighbors.length; i++ do
6          if neighbors[i] == v then
7              start ← i;
8              end ← i;
9              break;
10         if neighbors[i] > v then
11             return false;
12     if start ≥ 0 then
13         for j=i+1; j < neighbors.length; j++ do
14             if neighbors[j] == v then
15                 end++;
16             else
17                 break;
18     else
19         return false;
20     timestamps ← T.timestamps(u);
21     for i=start; i≤end; i++ do
22         if timestamps[i] ≥ t_start and
             timestamps[i] ≤ t_end then
23             return true;
24         return false;
```

E. Compressing Offsets

Efficiently retrieving the neighbors of nodes and their associated timestamps requires knowing where each adjacency or timestamp list starts, in the respective representation. To this end, we maintain a list of offsets for both representations that allow for random access to the graph's elements. However, maintaining these offsets in *uncompressed* form in main-memory can be very expensive.

The Elias—Fano encoding [44], [45] offers a compact representation for any non-decreasing monotone sequence $x_0, x_1, \ldots, x_n - 1$ of n natural numbers smaller than u, while supporting constant time access to the i-th element. Elias-Fano splits the binary representation of each element into two parts, the higher and the lower:

- we store the $l = \log \frac{u}{n}$ lower bits explicitly, and
- regarding the higher bits, we associate a counter to each possible value corresponding to the cardinality of that value. Then, we store the cardinality in unary coding, i.e., we append as many 1-bits as the counter of each value followed by a 0-bit.

The final representation requires at most $2 + \log \frac{u}{n}$ bits per element, which is very close to the information-theoretical lower bound of $\log e + \log \frac{u}{n}$.

The offsets for the adjacency list and the timestamp list of the graph's nodes are both cases of non-decreasing monotone sequences. Therefore, we employ Elias–Fano encoding to come up with a succinct representation for both these lists of offsets; this makes it possible to expend for each list, a number of bits roughly equal to the logarithm of the average length of the list.

F. Querying the Compressed Graph

Compressed graph representations reside in main-memory and offer rapid access to the graph's elements without decompression. Queries such as obtaining the neighbors of a node or investigating the existence of an edge enable the execution of many complex graph algorithms such as community detection, PageRank and anomaly detection.

CHRONOGRAPH maintains in main memory the compressed representations of the network structure and the timestamps as well as the two lists of offsets that point to these representations. These two lists provide *direct* access to the position where we keep the neighbors of a node and their associated timestamps. Thus, we can obtain in constant time the information we need to decompress in order to answer *neighbor* and *edge* queries for user-specified intervals or points in time. Starting with the compressed network structure, we retrieve *i)* the neighbors with multiple occurrences, *ii)* the neighbors described through a reference to another node, *iii)* the intervals with consecutive neighbors, and *iv)* the remaining nodes. While decompressing these neighbors, we have to restore their initial ordering so that they can be mapped to their respective timestamps. We do so 'on the fly', by using the neighbor with the smallest label from each of the four techniques used. The timestamps associated with the neighbors of a node are compressed uniformly, using gap encoding and

TABLE III: Dataset of our experimental setup, featuring six real-world and two synthetic large-scale graphs.

Graph	Type	Nodes	Edges	Contacts	Time steps	Lifetime	Granularity
Flickr	Incremental	2,302,925	33,140,017	33,140,017	134	134	day
Wiki-Edit	Point	21,504,191	122,075,170	266,769,613	134,075,025	304,002,799	second
Wiki-Links-sub	Interval	50,768,029	224,355,260	686,861,868	121,065,245	627,275,625	second
Wiki-Links-full	Interval	173,158,115	1,303,811,744	2,742,681,110	257,944,225	627,393,540	second
Yahoo-sub	Point	40,616,538	163,243,366	537,966,048	54,094	54,094	second
Yahoo-full	Point	103,661,225	776,456,413	3,023,733,076	181,292	181,292	second
Comm.Net	Interval	10,000	17,625,882	19,061,570	10,001	-	-
Powerlaw	Interval	1,000,000	32,238,517	32,280,816	1,001	-	-

a particular ζ code. Hence, we can retrieve the timestamps by simply applying the reverse operations on the compressed timestamps' representation. In cases when we are interested in a specific time interval or point, we additionally need to examine whether an edge should be included once we obtain both the neighbor and the respective timestamp.

Algorithm 1 outlines how we can answer whether a node v is included in the neighbors of another node u at a specific time interval $[t_{start}, t_{end}]$. We initially iterate through the neighbors of u. If we retrieve a neighbor that has a larger label than the one we are looking for or we run out of neighbors, we know that the node is not included in the neighbors. If we do find the node we are looking for, we count its occurrences and then retrieve the respective timestamps. Depending on the graph type and the timestamp values, we answer whether the edge was active during the specified interval.

Slight modifications to Algorithm 1 allow for finding points or periods in time when an edge was active. Again, we need not decompress the entire list of neighbors, as ChronoGraph keeps nodes ordered by their label and so we can skip edges in a way similar to Line 11 of Algorithm 1. Instead of answering whether we found a timestamp for the edge at the specified interval, we would simply return the relevant timestamps. Retrieving all neighbors of a node before, during, or after a time interval or point is accomplished by filtering out neighbors associated with timestamps that should not be included. This operation depends on the type of graph; for point and incremental graphs, we simply filter out the neighbors with timestamps outside the specified time frame, whereas for contact graphs, we need to keep track of whether the respective edge is active. Finally, to obtain a snapshot of the graph we simply retrieve the neighbors of all nodes during the time interval that we are interested in.

V. Experimental Evaluation

We implemented and tested our framework on a variety of real-world and synthetic large scale graphs. In Section V-A, we provide the specifications of the machine used for implementing and testing our framework and describe the dataset used. The remainder of the section outlines the results of our method's evaluation. We compare ChronoGraph with existing methods with respect to the compression ratio, compression times and access times in Sections V-B, V-E and V-D, and also provide comparisons of different aggregation levels and different encodings in Sections V-C and V-F respectively.

A. Technical Specifications & Dataset

We implemented and ran our algorithm using Java 11; our code, together with reproducible experiments, is publicly available.[3] The experiments were carried out on a computer running Xubuntu 18.04 OS with an Intel® Core™ i5-4590, with a CPU frequency of 3.30GHz and a 6MB L3 cache, a total of 16GB DDR3 1600MHz RAM and a hard disk drive.

The results for EdgeLog, CET, and CAS are produced using implementations made publicly available by the authors.[4] In cases where we report no results, the execution failed due to extensive memory use. The results we report for EveLog, ck^d-trees, and T-ABT are repeated from [7], [9], as there exist no publicly available implementations for these algorithms.

The dataset on which we applied and tested our compression techniques comprises the eight large-scale graphs listed in Table III. Six of these graphs represent real-world networks whereas the rest two are synthetic. Table III shows the number of nodes and edges of each graph, as well as the number of contacts that are created, the number of time steps used, the lifetime of each graph and the available time granularity. The origin and characteristics of our graphs are summarized below:

- `Flickr`[5] [46] is an incremental temporal graph, with a granularity of a day and a lifetime of 134 days, from November 2nd, 2006 to May 18th, 2007, representing friendships in the Flickr social network.
- `Wiki-Edit` is a point temporal graph composed of the history of linked articles before each edition, with a time granularity of a second. It corresponds to the history dump of Wikipedia, from its creation in January 17th, 2001 up to September 6th, 2010. Wiki-Edit is a bipartite graph with its two disjoint sets of vertices representing Wikipedia users and Wikipedia articles. Node labels are not unique in the two disjoint sets. However, as the article nodes have no outgoing edges, we use the original labels of the nodes, just as the earlier approaches have done [6], [7], [9], to ensure a fair comparison.
- `Wiki-Links` is an interval graph that represents the history of links among articles in the English version of Wikipedia

[3] https://github.com/panagiotisl/evolving-graph-compression
[4] https://github.com/diegocaro/temporalgraphs
[5] http://socialnetworks.mpi-sws.org/data-www2009.html

TABLE IV: Comparison with compression ratios of other methods. All ratios are measured in bits/contact. For CHRONOGRAPH we report the overall requirements in bold and the timestamp requirements in a parenthesis.

Graph	Raw	Gzip	EveLog	EdgeLog	CET	CAS	ck^d-trees	T-ABT	CHRONOGRAPH	Impr. (%)
Flickr	217.63	37.46	74.2 [7]	197.7	131.87	74.4	23.0 [7]	22.43 [9]	**19.04** (6.02)	15.11%
Wiki-Edit	213.45	51.78	84.8 [7]	127.4	54.1	60.6	41.7 [7]	51.67 [9]	**31.56** (23.48)	24.32%
Wiki-Links-sub	222.86	38.77	84.0 [7]	-	-	41.1	61.2 [7]	66.99 [9]	**32.87** (26.49)	20.00%
Wiki-Links-full	237.07	38.51		-	-	-			**33.12** (24.67)	45.88%
Yahoo-sub	202.61	35.08	103.7 [7]	-	55.65	49.5	34.0 [7]	44.07 [9]	**20.33** (14.8)	40.21%
Yahoo-full	207.53	35.07		-	-	-			**19.21** (14.18)	43.05%
Comm.Net	114.0	23.2	45.0 [7]	82.0	60.9	32.3	26.0 [7]	29.84 [9]	**10.08** (7.01)	61.23%
Powerlaw	136.87	34.89	77.5 [7]	132.4	86.0	45.6	31.9 [7]	32.53 [9]	**21.25** (4.46)	33.39%

from January 15th, 2001 to December 3rd, 2020. The original dataset used in [7], [9] could not be made available by the authors and was recreated from the original source data.[6] As the resulting graph is more than 3x larger than the one used in earlier efforts, we also generated a subgraph (`Wiki-Links-sub`) using part of the raw data.

- `Yahoo` contains communication patterns between end-users in the large Internet and Yahoo servers. A netflow record includes timestamp, source IP address, destination IP address. The Yahoo dataset used in [7], [9] could not be made available by the authors and was recreated from the original source data.[7] As the resulting graph is 3x larger than the one used in earlier efforts, we also generated a subgraph (`Yahoo-sub`) using the records of April 29th, 2008.
- `Comm.Net` is a synthetic dataset simulating a random network of nodes that establish short-life communications. The network is built in the context of the work in [9] using the Erdös-Rényi model and according to the instructions provided in [6]. The graph is synthetic so the lifetime and granularity have no meaning.
- `Powerlaw` is also synthetic and is built in the context of the work in [9] using the Barabási-Albert model and according to the instructions provided in [6]. Lifetime and granularity have no meaning, as Powerlaw is synthetic.

The above graphs vary in category and size, and therefore they are very good candidates for examining the effectiveness of our proposed method. Furthermore, these graphs have been used in the evaluation of previously proposed approaches and thus, allow for making comparisons against earlier methods.[8]

B. Compression Ratio Comparison

We commence the experimental study of our framework by investigating the memory requirements of CHRONOGRAPH. Table IV shows the bits per contact that several earlier approaches require for the graphs of our dataset, compared to our approach. For our results we give the overall compression ratio (using bold font) as well as the bits per contact required for the temporal information only (in parenthesis). Note that we were not able to compress all graphs using EdgeLog, CET and CAS. Moreover, we express the results in terms of bits per contact as the `Wiki-Links` and `Yahoo` datasets we use are not identical to the ones used in the results of EveLog, ck^d-trees, and T-ABT, originally reported in [7], [9].

Results in Table IV ascertain that CHRONOGRAPH significantly outperforms all earlier approaches for all graphs in our dataset. In particular, the improvement over the second best ratio for each graph ranges from 15.11% to an impressive 61.23%, with an average improvement of 35.4%. The increased savings of CHRONOGRAPH compared to individual approaches are actually even greater, as the second best result is not always produced by the same algorithm. More importantly, CHRONOGRAPH represents actual timestamps with the available granularity of the dataset, whereas earlier approaches simply represent time steps. In the cases of `Wiki-Edit`, `Wiki-Links-sub` and `Wiki-Links-full` the time steps used by other approaches are less than half of the graph's actual lifetime. Therefore, our substantial savings are achieved while also offering much more *accurate* temporal information.

C. Aggregation Level Comparison

We observe in Table IV that a great portion of CHRONOGRAPH space requirements is due to the representation of timestamps. More specifically, the storage requirements for temporal information are usually more than half of the overall needed space; the only exceptions here are that of `Flickr` and `Powerlaw` graphs which have very short lifetimes.

Such fine granularity with regards to time is not always necessary in network analysis tasks. Oftentimes an hourly or daily level is more than enough. Our framework provides flexibility in specifying a *desired granularity* in the representation of time by selecting the respective aggregation level. If the granularity of time in the source data is greater than what the network analysis task at hand calls for, we may choose an appropriate aggregation level to further minimize the overall space requirements.

Figure 6 illustrates the increased savings of CHRONOGRAPH when selecting an aggregation level that is larger than the available granularity, for all the real-world graphs of our dataset. For `Wiki-Edit`, `Wiki-Links` and `Yahoo` the granularity of the source dataset is in seconds. We see in Figure 6 that the space requirements decrease as we lower

[6]https://dumps.wikimedia.org/backup-index.html

[7]http://webscope.sandbox.yahoo.com/catalog.php?datatype=g

[8]The authors sincerely thank Prof. Sridhar Radhakrishnan and Prof. Diego Caro for providing guidelines, links and access to the graphs used in our experimental section.

TABLE V: Access (μs) and compress (s) time comparison against state-of-the-art approaches. CHRONOGRAPH is usually orders of magnitude faster and scales much better for large graphs. Measurements for EveLog, ck^d-trees, and T-ABT are repeated from [7], [9] as there exist no publicly available respective implementations.

	Algorithm	Flickr	Wiki-Edit	Wiki-Links sub	Wiki-Links full	Yahoo sub	Yahoo full	Comm.Net	PowerLaw
Neighbors (μs)	**EveLog [7]**	>100 [7]	$\gg 10^6$ [7]	$\gg$10,000 [7]		$\gg 10^6$ [7]		-	$\gg$1,000 [7]
	EdgeLog	80.5	28.76	-	-	-	-	253.9	7.9
	CET	4,530.1	12,709.0	-	-	344,309.4	-	2,340.5	109.8
	CAS	780.1	402.6	613.0	-	317,968.1	-	1,272	146.2
	ck^d-trees [7]	$\gg$100 [7]	$\gg$10,000 [7]	$\gg$10,000 [7]		$\gg$10,000 [7]		-	$\gg$1,000 [7]
	T-ABT [9]	9	950	1,200 [9]		110 [9]		**9** [9]	7 [9]
	CHRONOGRAPH	**3.4±0.2**	**6.3±0.5**	**10.2±1.0**	**12.8±0.5**	**5.3±0.4**	**9.3±2.5**	70.5±5.7	**0.37±0.11**
Edge (μs)	**EveLog [7]**	$\gg$10,000 [7]	$\gg 10^6$ [7]	$\gg 10^6$ [7]		$\gg 10^7$ [7]		-	$\gg$10 [7]
	EdgeLog	14.1	474.1	-	-	-	-	39.6	9.2
	CET	14.4	30.2	-	-	33.1	-	20.2	23.2
	CAS	42.3	70.8	53.3	-	44.6	-	42.3	26.8
	ck^d-trees [7]	>10 [7]	$\gg$10 [7]	$\gg$10,000 [7]		$\gg$10 [7]		-	>1,000 [7]
	T-ABT [9]	**0.3**	77	92 [9]		8.6 [9]		**0.7** [9]	0.3 [9]
	CHRONOGRAPH	1.1±0.1	**2.4±0.3**	**1.2±0.8**	**7.1±1.2**	**2.8±0.3**	**2.9±0.8**	2.9±0.7	**0.11±0.02**
Compress (s)	**EdgeLog**	236	1,337	-	-	-	-	121	221
	CET	630	452	-	-	1,514	-	**34**	79
	CAS	138	965	4,577	-	2,624	-	46	133
	T-ABT [9]	282	1,836	6,120 [9]		7,200 [9]		126 [9]	264 [9]
	CHRONOGRAPH	**22.71**	**200.69**	**463.83**	**2,141.5**	**417.37**	**2,069.73**	42.31	**77.98**

the granularity. The savings are particularly impressive as we go from a second to half an hour. We achieve even greater efficiency as we elongate the granularity further; however, the savings are not as important from a point on. Regarding `Flickr`, the available granularity is in days. We see in Figure 6 that lowering the granularity to two days offers negligible improvements.

The results of Figure 6 are in accordance with our discussion in Section IV-A. Applying aggregations helps us reduce the range of values for the gaps of timestamps while also preserving the skewness in their distribution. This results in a greater concentration of small integers in our representation for timestamps and improves our compression ratio.

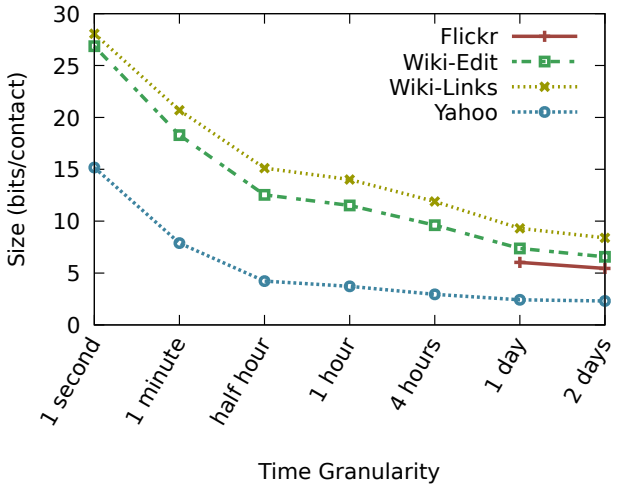


Fig. 6: Comparison for various aggregation levels. The resulting size is significantly reduced even when lowering the granularity. However, the impact weakens as the aggregation intervals grow.

D. Access Time Comparison

We have shown that our framework for temporal graph compression offers significantly improved space-efficiency over state-of-the-art approaches. We now focus on access time efficiency and investigate how quickly we can retrieve the graph's elements. In particular, we measure the time needed to i) retrieve *all neighbors* of a particular node and ii) check whether an *edge exists* during a specified time interval.

Table V features the CHRONOGRAPH performance against several state-of-the-art approaches for both these retrieval operations over all the graphs of our dataset. Results provided are averages of 10^8 random executions on the *same hardware*, except for EveLog, ck^d-trees and T-ABT for which we repeat results reported in [7] and in [9].

The results of V readily ascertain that CHRONOGRAPH is extremely responsive offering answers to both queries in just a few μs. We also observe that the access time of CHRONOGRAPH depends on the average degree of the graph and the degree distribution, but not the size of the graph, as we obtain the compressed neighbors and timestamps of a particular node in constant time, using our offset indices.

Our worst result is for retrieving the neighbors of `Comm.Net` that exhibits an unreal 1,906.16 average contacts per node. For the real-world graphs of our dataset that exhibit much smaller average degrees, our access times are significantly faster. Finally, regarding the `PowerLaw` synthetic graph the access time is extremely small, due to the power law degree distribution.

We also see in Table V that our ChronoGraph greatly outperforms all other approaches featured in our evaluation, that often require multiple tree traversals to acquire the compressed representation of a node's neighbors. The results of all EveLog, EdgeLog, CET, CAS, ck^d-trees and T-ABT appear to depend on the size of the graph, and provide orders of magnitude slower access to the graph's elements compared to our ChronoGraph for the large-scale graphs of our dataset. T-ABT is sometimes slightly faster than our approach for very small graphs. However, as the size of the graph grows the access time of T-ABT deteriorates quickly.

E. Compression Time Comparison

Another important aspect of graph compression approaches is the time required to produce the compact representation. Table V also features the time that state-of-the-art approaches and our ChronoGraph require to compress each graph of our dataset. Unfortunately, such results are not available for EveLog and ck^d-trees in [7]. We can see that the compression time grows with the size of the graph for all approaches. However, our ChronoGraph is clearly much faster, as we offer more than 70% improvement on average, as far as compression time is concerned. We remind the reader that our results for `Wiki-Links` and `Yahoo` consider significantly larger graphs than the results of [9] and the actual improvement over T-ABT is more accurately depicted when comparing with their subgraphs also shown in Table V. We also note the memory requirements of ChronoGraph while compressing the graph are very limited and are mostly related to the two offset indexes that our representation uses.

F. Comparing Codes

Our ChronoGraph framework compresses the timestamps associated with the neighbors of a node using i) gap encoding and ii) ζ codes, a set of flat codes that are particularly suitable for integers with a power law distribution in a certain exponent range [24]. Here, we investigate the space-efficiency of our timestamps' representation when varying the k parameter of ζ codes.

Figure 7 illustrates the overall memory requirements of ChronoGraph for the representation of timestamps of various graphs, when using granularity of a second and a minute, respectively. We report the space needed for storing the timestamps and their respective offsets' index, but not the network structure which remains an invariant factor. Our best results when applying minute aggregations are obtained using a k that is smaller than the one needed to achieve the best result without aggregation. The reason why larger values of k are better when using fine time granularity is evident in Figure 4. Applying aggregations reduces the size of the maximum gap in the distribution. Thus, codes that are extremely efficient in representing small values but less efficient for large integers, such as ζ_2, become the preferred option.

Furthermore, we see in Figure 7 that the optimal values of k for `Yahoo` are smaller than the optimal values for `Wiki-Edit` and `Wiki-Links`. This is due to the very large time spans of

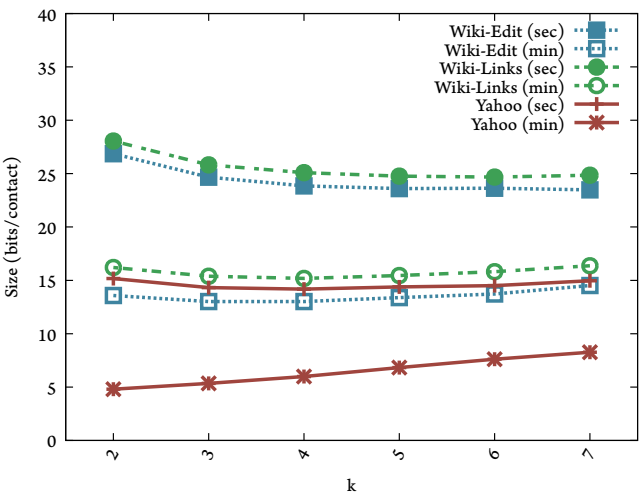


Fig. 7: Comparison for various ζ codes (2-7) for the compression of timestamps. The optimal k depends on the time granularity and the distribution of timestamp gaps.

the latter that last for several years, as opposed to `Yahoo` with a lifetime of a few days. The distributions of the timestamp gaps of these graphs, depicted in Figure 3, clearly show a larger concentration of small values and a much smaller maximum gap for `Yahoo`. Therefore, we expect that temporal graphs with significantly long lifetimes will be best compressed by using ζ_5 or ζ_6 when no aggregation is applied, and ζ_3 otherwise. For temporal graphs with shorter time spans we expect to obtain our best results using ζ_4 when no aggregation is applied, and ζ_2 otherwise.

VI. Conclusions and Future Work

In this paper we introduce the ChronoGraph compression framework for temporal graphs. ChronoGraph employs a dual representation to efficiently manage in memory the network structure and the temporal content of the graph. We uncover and exploit properties that real-world temporal graphs exhibit and offer a space-efficient in-memory representation that provides rapid access to the graph's elements. In addition, our approach uses actual time and allows for the proper designation of the granularity level of the representation. The experimental evaluation of ChronoGraph shows that it greatly outperforms state-of-the-art approaches by attaining a significantly lower, sometimes *less than half*, compression ratio, while at the same time ChronoGraph offers orders of magnitude faster compression and access time to the graph's elements. In this regard, our framework does empower the execution of algorithms on large-scale temporal graphs, in settings where the use of existing methods is prohibitive due to memory constraints or simply too inefficient.

We plan to extend our work by investigating the applicability of our techniques for algorithms based on the 'think like a vertex' programming paradigm. We expect that the properties of temporal graphs introduced here will also be effective in a distributed setting suitable for such algorithms [23].

REFERENCES

[1] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora, "Graph Metrics for Temporal Networks," in *Temporal Networks, Understanding Complex Systems*. Springer-Verlag, June 2013, pp. 15–40.

[2] M. Goldberg, M. Magdon-Ismail, S. Nambirajan, and J. Thompson, "Tracking and Predicting Evolution of Social Communities," in *Proc. of IEEE 3rd Int. Conf. on Privacy, Security, Risk and Trust & IEEE 3rd Int. Conf. on Social Computing*, Boston, MA, October 2011, pp. 780–783.

[3] P. Liakos, K. Papakonstantinopoulou, A. Ntoulas, and A. Delis, "Rapid detection of local communities in graph streams," *IEEE Trans. on Knowledge and Data Engineering*, 2020.

[4] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal, "PageRank on an Evolving Graph," in *Proc. of 18th ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, Beijing, PR China, 2012, p. 24–32.

[5] R. Hu, C. C. Aggarwal, S. Ma, and J. Huai, "An Embedding Approach to Anomaly Detection," in *IEEE 32nd Int. Conf. on Data Engineering (ICDE)*, Helsinki, Finland, May 2016, pp. 385–396.

[6] D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data Structures for Temporal Graphs Based on Compact Sequence Representations," *Information Systems*, vol. 51, pp. 1–26, July 2015.

[7] D. Caro, M. A. Rodríguez, N. R. Brisaboa, and A. Fariña, "Compressed *kd*-tree for Temporal Graphs," *Knowledge Information Systems*, vol. 49, no. 2, pp. 553–595, 2016.

[8] N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez, "Using Compressed Suffix-Arrays for a Compact Representation of Temporal-Graphs," *Information Sciences*, vol. 465, pp. 459–483, October 2018.

[9] M. Nelson, S. Radhakrishnan, and C. N. Sekharan, "Queryable Compression on Time-Evolving Social Networks with Streaming," in *Proc. of IEEE Int. Conf. on Big Data*, Seattle, WA, December 2018, pp. 146–151.

[10] F. Chierichetti, R. Kumar, S. Lattanzi, A. Panconesi, and P. Raghavan, "Models for the compressible web," in *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, 2009, pp. 331–340.

[11] I. Kontoyiannis, Y. H. Lim, K. Papakonstantinopoulou, and W. Szpankowski, "Symmetry and the entropy of small-world structures and graphs," in *IEEE International Symposium on Information Theory, ISIT 2021, Melbourne, Victoria, Australia, July 12-20, 2021*.

[12] K. Bharat, A. Z. Broder, M. R. Henzinger, P. Kumar, and S. Venkatasubramanian, "The Connectivity Server: Fast Access to Linkage Information on the Web," *Computer Networks*, vol. 30, no. 1-7, pp. 469–477, 1998.

[13] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener, "The Link Database: Fast Access to Graphs of the Web," in *Proc. of Int. Data Compression Conf. (DCC)*, Snowbird, UT, 2002, pp. 122–131.

[14] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proc. of the 13th Int. Conf. on World Wide Web*, Rio de Janeiro, Brazil, May 2004, pp. 595–602.

[15] Y. Asano, Y. Miyawaki, and T. Nishizeki, "Efficient Compression of Web Graphs," in *Proc. of 14th Int. Conf. on Computing and Combinatorics (COCOON)*. Dalian, China: Springer-Verlag, June 2008, pp. 1–11.

[16] N. Brisaboa, S. Ladra, and G. Navarro, "k2-Trees for Compact Web Graph Representation," in *Proc. of 16th Int. Symp. on String Processing and Information Retrieval (SPIRE)*. Saariselka, Finland: Springer-Verlag, August 2009, pp. 18–30.

[17] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On Compressing Social Networks," in *Proc. of Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, Paris, France, July 2009, pp. 219–228.

[18] A. Apostolico and G. Drovandi, "Graph Compression by BFS," *Algorithms*, vol. 2, no. 3, pp. 1031–1044, 2009.

[19] F. Claude and S. Ladra, "Practical Representations for Web and Social Graphs," in *Proc. of ACM Int.Conf. on Information and Knowledge Management (CIKM)*, October 2011, pp. 1185–1190.

[20] P. Liakos, K. Papakonstantinopoulou, and M. Sioutis, "On the effect of locality in compressing social networks," in *Advances in Information Retrieval - 36th European Conf. on IR Research, (ECIR), Amsterdam, The Netherlands. Proceedings*, April 2014, pp. 650–655.

[21] P. Liakos, K. Papakonstantinopoulou, and A. Delis, "Memory-optimized distributed graph processing through novel compression techniques," in *Proc. of the 25th ACM Int. Conf. on Information and Knowledge Management, (CIKM), Indianapolis, IN*, October 2016, pp. 2317–2322.

[22] P. Liakos, K. Papakonstantinopoulou, and M. Sioutis, "Pushing the Envelope in Graph Compression," in *Proc. of 23rd ACM Int. Conf. on Conference on Information and Knowledge Management (CIKM)*, Shanghai, PR China, November 2014, pp. 1549–1558.

[23] P. Liakos, K. Papakonstantinopoulou, and A. Delis, "Realizing Memory-Optimized Distributed Graph Processing," *IEEE Trans. on Knowledge Data Engineering*, vol. 30, no. 4, pp. 743–756, 2018.

[24] P. Boldi and S. Vigna, "Codes for the world wide web," *Internet Mathematics*, vol. 2, no. 4, pp. 407–429, 2005.

[25] U. Khurana and A. Deshpande, "Efficient Snapshot Retrieval over Historical Graph Data," in *Proc. of 29th IEEE Int. Conf. on Data Engineering (ICDE)*, Brisbane, Australia, 2013, pp. 997–1008.

[26] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han, "The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs," *Distributed Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2015.

[27] K. Semertzidis and E. Pitoura, "Durable Graph Pattern Queries on Historical Graphs," in *Proc. of 32nd IEEE Int. Conf. on Data Engineering (ICDE)*, Helsinki, Finland, May 2016, pp. 541–552.

[28] N. R. B. Diego Caroa, M. Andrea Rodríguez, "Data Structures for Temporal Graphs Based on Compact Sequence Representations," *Information Systems*, vol. 51, pp. 1–26, July 2015.

[29] R. Grossi, A. Gupta, and J. S. Vitter, "High-order Entropy-compressed Text Indexes," in *Proc. of 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Baltimore, MD, January 2003, pp. 841–850.

[30] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, no. 2, p. 187–260, June 1984.

[31] M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. N. Sekharan, "Queryable Compression on Streaming Social Networks," in *Proc. of IEEE Int. Conf. on Big Data*, Boston, MA, December 2017, pp. 988–993.

[32] V. Kostakos, "Temporal Graphs," *Physica A Statistical and Theoretical Physics*, vol. 388, no. 6, pp. 1007–1023, July 2008.

[33] F. Harary and G. Gupta, "Dynamic Graph Models," *Mathematical and Computer Modelling*, vol. 25, no. 7, p. 79–87, April 1997.

[34] P. Grindrod and D. Higham, "Evolving Graphs: Dynamical Models, Inverse Problems and Propagation," *Proceedings of the Royal Society A*, vol. 466, January 2009.

[35] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying Graphs and Dynamic Networks," *Int. Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[36] P. Boldi, M. Santini, and S. Vigna, "Permuting web and social graphs," *Internet Math.*, vol. 6, no. 3, pp. 257–283, 2009.

[37] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks," in *Proc. of 20th WWW Int. Conf.*, 2011, pp. 587–596.

[38] L. Speidel, R. Lambiotte, K. Aihara, and N. Masuda, "Steady state and mean recurrence time for random walks on stochastic temporal networks," *Phys. Rev. E*, vol. 91, p. 012806, Jan 2015. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.91.012806

[39] E. Ubaldi, A. Vezzani, M. Karsai, N. Perra, and R. Burioni, "Burstiness and tie activation strategies in time-varying social networks," *Scientific reports*, vol. 7, no. 1, pp. 1–11, 2017.

[40] K. Zhu, G. H. L. Fletcher, N. Yakovets, O. Papapetrou, and Y. Wu, "Scalable temporal clique enumeration," in *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*, 2019, pp. 120–129.

[41] K. Zhu, G. Fletcher, and N. Yakovets, "Competition-driven modeling of temporal networks," *EPJ Data Sci.*, vol. 10, no. 1, p. 30, 2021.

[42] M. Adler and M. Mitzenmacher, "Towards Compressing Web Graphs," in *IEEE Int. Data Compression Conf. (DCC)*, Snowbird, UT, March 2001, pp. 203–212.

[43] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[44] P. Elias, "Efficient Storage and Retrieval by Content and Address of Static Files," *Journal of ACM*, vol. 21, no. 2, p. 246–260, April 1974.

[45] R. M. Fano, "On the Number of Bits Required to Implement an Associative Memory – Memorandum 61," Computer Structures Group, Project MAC, MIT, Cambridge, MA, Tech. Rep., 1971.

[46] M. Cha, A. Mislove, and K. P. Gummadi, "A Measurement-Driven Analysis of Information Propagation in the Flickr Social Network," in *Proc. of the 18th Int. Conf. on World Wide Web (WWW)*, Madrid, Spain, April 2009, p. 721–730.