Using Compressed Suffix-Arrays for a Compact Representation of Temporal-Graphs[☆]

Nieves R. Brisaboa[c], Diego Caro[a,b], Antonio Fariña[*,c], M. Andrea Rodriguez[d]

[a]*Data Science Institute, Faculty of Engineering, Universidad del Desarrollo, Chile.*
[b]*Telefónica I+D Fellow, Chile*
[c]*Database Laboratory, University of A Coruña, Spain.*
[d]*Department of Computer Science, University of Concepción, Chile.*

Abstract

Temporal graphs represent binary relationships that change along time. They can model the dynamism of, for example, social and communication networks. Temporal graphs are defined as sets of contacts that are edges tagged with the temporal intervals when they are active. This work explores the use of the Compressed Suffix Array (CSA), a well-known compact and self-indexed data structure in the area of text indexing, to represent large temporal graphs. The new structure, called Temporal Graph CSA (TGCSA), is experimentally compared with the most competitive compact data structures in the state-of-the-art, namely, EdgeLog and CET. The experimental results show that TGCSA obtains a good space-time trade-off. It uses a reasonable space and is efficient for solving complex temporal queries. Furthermore, TGCSA has wider expressive capabilities than EdgeLog and CET, because it is able to represent temporal graphs where contacts on an edge can temporally overlap.

Key words: Temporal Graphs, Compressed Suffix Array, Self-index

1. Introduction

The main assumption of static graphs is that the relationship between two vertexes is always available. However, this is not true in many real world situations. For example, consider how friendship relations evolve in an online social network, or how the connectivity in a communication network changes when users, with their mobile devices, move in a city. Temporal graphs deal with the time-dependence of relationships

[☆]Funded in part by European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941 (project BIRDS). D. Caro is partially funded by the Chilean government initiative CORFO 13CEE2-21592 (2013-21592-1-INNOVA PRODUCCION). M. A. Rodríguez is partially funded by Fondecyt [1170497] and the Complex Engineering Systems Institute (CONICYT: FBO16). N. R. Brisaboa and A. Fariña were partially funded by Xunta de Galicia/FEDER-UE [CSI: ED431G/01 and GRC: ED431C 2017/58]; by MINECO-AEI/FEDER-UE [Datos 4.0: TIN2016-78011-C4-1-R and ETOME-RDFD3: TIN2015-69951-R]; and by MINECO-CDTI/FEDER-UE [CIEN: LPS-BIGGER IDI-20141259 and INNTERCONECTA: uForest ITC-20161074]. An early partial version of this article appeared in *Proc. SPIRE'14* [3].

[*]Corresponding author

Email addresses: `brisaboa@udc.es` (Nieves R. Brisaboa), `dcaro@udd.cl` (Diego Caro), `fari@udc.es` (Antonio Fariña), `andrea@udec.cl` (M. Andrea Rodriguez)

between vertexes by representing these relationships as a set of *contacts* [36]. Each contact represents an edge (i.e., two vertexes) tagged with the time interval when the edge was active. For example, in a communication network, a contact may represent a call between users made from 4 pm to 4.05 pm.

The temporal dimension of edges adds a new constraint to the relationship between vertices not found in static graphs: two vertexes can communicate only if there is a time-respecting path (also called journeys [36]) between them [36, 46, 50, 47, 19]. For example, in Figure 1.b (corresponding to the time aggregation of the edges in the temporal graph of Figure 1.a), there are two paths connecting the vertexes a and d: one through the vertex b, and the other one through c. However, there is no such path when considering the temporal availability of the edges (a, b) and (a, c). Notice that the vertexes b and c are only reachable from the vertex a because the edges reaching d are not available. Therefore, taking into account the temporal dynamism of graphs allows us to exploit information about temporal correlations and causality, which would be unfeasible through a classical analysis of static graphs [36, 19, 32].

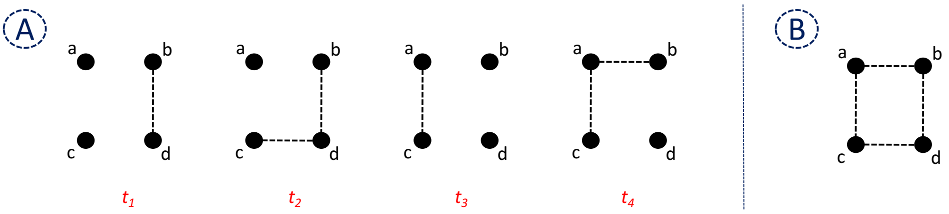


Figure 1: A temporal graph composed of four vertexes with a lifespan of four time-instants: a) A snapshot-based representation showing the available edges per time-instants, b) the time-aggregated graph of the temporal graph.

A direct approach to represent temporal graphs could be a time-ordered sequence of snapshots (Figure 1a), one for each time instant, showing the state of the temporal graph at a time instant as a static graph. Several centralized and distributed processing systems follow this approach (e.g. Pregel [29], Giraph[1], Neo4J[2], Trinity [44]), but without specific support for temporal extensions [24].

In temporal graphs where contacts are active during long time intervals (as in a social network), consecutive snapshots tend to become very similar. Thus, strategies based on a sequence of snapshots are space consuming because edges are duplicated in each snapshot. An alternative change-based approach represents the temporal graph by the differences between snapshots; that is, by the set of edges that appear/disappear along time. These differences can be calculated with respect to consecutive snapshots [15], or with respect to a derived graph that diminishes the number of stored edges [38, 23, 26, 42].

The change-based approach has also been used for pre-computing reachability queries [43, 42], as some paths may remain available for several time instants [2]. Although these works improve the time performance

[1] `http://giraph.apache.org/`

[2] `http://neo4j.org/`

of complex algorithms, they overlook the space cost, which becomes crucial for large temporal graphs. In this context, a compact representation can keep larger sections or even the whole temporal graph in memory and, in consequence, queries could become much more efficient by avoiding disk transfers.

Recently, some compact approaches to represent temporal graphs have been proposed [7, 8]. The work in [7] presents the *ck^d-tree*, a tree-shaped compact data structure based on the Quadtree [40], which represents a temporal graph as a point in a four dimensional space. This data structure was designed to reduce space usage at the expense of time access in sparse temporal graphs. EdgeLog (Time Interval Log per Edge) [8] uses a compressed inverted index, which also provides fast answers to different types of queries, in particular, when solving adjacency queries involving the recovery of active neighbors of a vertex at a specific time instant. CET (Compact Events ordered by Time) [8] uses a wavelet tree [34, 16] to represent temporal graphs and is the best alternative in the state-of-the-art to answer queries related to time-instant events that change the state of an edge.

Both EdgeLog and CET overcome the overload of storing a snapshot per each time instant by representing the temporal graph as a log of events. These events indicate when edges become active or inactive. Then, the activation state of a given edge can be recovered by counting how many events occurred on that edge during a time interval. If there is an even number of events, it means that the edge has been active and inactive several times. Conversely, if the edge has an odd number of events, it means that the last state of the edge is active. A detailed explanation of these data structures is available in Section 2.

A main drawback of the log-based structures, such as EdgeLog and CET, is that they do not allow the representation of time-overlapping contacts of an edge. For example, if a contact represents the data communication between two machines X and Y during a time interval, it is impossible to represent a second contact between X and Y during an overlapping time interval. This limitation arises because in these structures the event that represents the activation of the second contact would be interpreted as the deactivation event of the first contact.

The work in this paper presents and evaluates a data structure named Temporal Graph CSA (TGCSA). The TGCSA is a compact and self-indexed structure based on a modification of the well-known Compressed Suffix Array (CSA)[39], extensively used for text indexing. We focus on algorithms to process temporal-adjacency queries that recover the set of active neighbors of a vertex at a given time instant. These queries are basic blocks to solve time-respecting paths [32], which can be useful in the context of moving-object data [30, 25], and also when analyzing activity patterns as temporally ordered sequences of actions occurring at specific time instances or time intervals [27, 28].

We also present algorithms for answering queries that recover the snapshot of the graph at a time instant, as well as queries to recover the state of single edges. In addition, we include a complete experimental evaluation with real and synthetic data that compares TGCSA with EdgeLog and CET in terms of both space and time usage. The results of this evaluation show that TGCSA opens new opportunities for the application of suffix arrays [31, 39] in the context of graphs in general, and of temporal graphs in particular.

As discussed above, there are different fields where the application of our TGCSA, or other compact existing alternatives from the state of the art such as EdgeLog or CET, can be of interest. Among others, we can mention [45, 21]: (i) Social networks, where friendships establish connections between nodes that can vary along time. (ii) Biological networks, where function brain connections are dynamic. (iii) Communication networks, where nodes are connected while their exchange information. This applies to person-to-person and machine-to-machine communication. (iv) Transportation networks, where the connectivity between nodes can change due to scheduling and traffic conditions. In this context, one could also model movements on a network by considering that two nodes are connected if there exists an object that moves from one to the other node during a time interval.

The structure of this paper is as follows. Section 2 presents preliminary concepts about temporal graphs and relevant queries on them. To make the paper self-contained, Sections 2.2 and 2.3 provide a brief overview of both EdgeLog and CET. These are the state-of-the-art techniques we compare TGCSA with. Section 3 introduces TGCSA by showing how to modify a traditional CSA to create TGCSA. It also describes how TGCSA solves relevant queries for temporal graphs and provides pseudocode for such operations. Finally, this section presents a new representation of the Ψ array from CSA [17, 14], called in this work vbyte-rle, which increases the query performance of TGCSA. Section 4 provides the experimental evaluation that uses real and synthetic data. Final conclusions and future research directions are given in Section 5.

2. Preliminary concepts

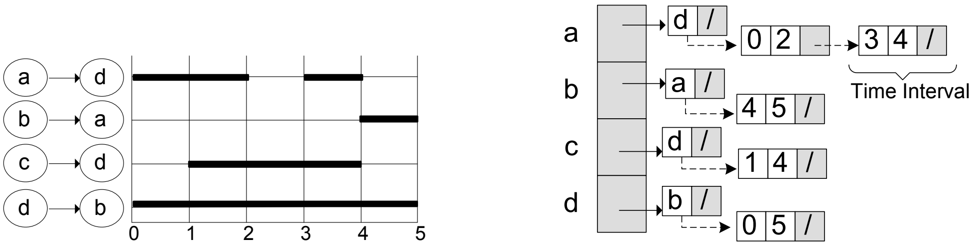
In this section we introduce temporal graphs and a classification of the relevant basic queries that could be of interest for most applications. We also revise previous compact representations of temporal graphs.

2.1. Temporal graph definition

Formally, a temporal graph is a set $\mathcal{C}$ of contacts that connect pairs of vertexes in a set V during a time interval defined over the set $\mathcal{T}$ that represents the *lifetime* of the graph. A *contact* in $\mathcal{C}$ of an edge $(u, v) \in E \subseteq V \times V$ is a 4-tuple $c = (u, v, t_s, t_e)$, where $[t_s, t_e) \in \mathcal{T} \times \mathcal{T}$ is the time interval when the edge (u, v) is active [36]. We say that an edge (u, v) is *active* at time t if there exists a contact $(u, v, t_s, t_e) \in \mathcal{C}$ such that $t \in [t_s, t_e)$. Note that this definition applies for directed graphs as we consider ordered pairs of vertexes.

We classify operations on temporal graphs into two categories: queries for checking the connectivity between vertexes and queries for retrieving the changes on the connectivity occurred along time. For the first category of queries, we define four operations: (1) activeEdge checks if an edge is active. (2) directNeighbor returns the active direct neighbors of a vertex. (3) reverseNeighbor gives the active reverse neighbors of a vertex. (4) snapshot returns all the active edges. For example, in the temporal graph of Figure 2.a, we know that at time instant $t = 1$ the edge (a, d) is active, the set of direct neighbors of c is $\{d\}$ and the set of reverse neighbors of d is $\{a, c\}$; whereas the snapshot at time $t = 3$ corresponds to the edges $\{(a, d), (c, d), (d, b)\}$.

For queries retrieving the changes on connectivity, we defined two operations: (1) activatedEdge returns the set of edges that were activated. (2) deactivatedEdge returns the set of edges that were deactivated. For example, given Figure 2.a at time instant $t = 4$, the edge $\{(b, a)\}$ was activated, and the edges $\{(a, d), (c, d)\}$ were deactivated.



(a) Set of contacts

(b) EdgeLog representation

Figure 2: A temporal graph of 4 vertexes and its EdgeLog representation. The reverse aggregated graph is omitted in (b).

Note that all previous queries have a time-instant or a time-interval version. In what follows, we concentrate on time-instant queries, which can be easily extended to answer time-interval queries, and they also serve as the building blocks for more complex temporal measures that are based on recovering time-respecting paths [32].

2.2. EdgeLog: Baseline representation

A simple temporal graph representation [6] stores the aggregated graph[3] as $|V|$ adjacency lists, one per each vertex, with a sorted list of time intervals attached to each neighboring vertex indicating when that edge is/was active. Figure 2.b shows a conceptual example.

To check if an edge (u, v) was active at time t, we first check if v appears within the adjacency list of vertex u. If v is found, then we need to check if t falls into one of the time intervals related to (u, v) that are represented in the time-interval list of that edge. Direct neighbors of vertex u at time t are recovered similarly. For each neighbor v in the adjacency list of u, we check if t is within the time intervals of the edge (u, v).

A simple representation of the aggregated graph and the temporal labels attached to vertices has two main drawbacks: (1) it uses much space; and (2) operation reverseNeighbor requires traversing all the

[3] The static graph including all the edges that were active at any time during the lifetime of the temporal graph.

adjacency lists. The data structure EdgeLog [8] addressed these weaknesses. On the one hand, since both the adjacency list and the time-interval list are sorted (i.e., they are of the form $\langle t_1, t_2, t_3, ..., t_l \rangle$, with $t_i < t_{i+1}$), they can be represented as d-gaps $\langle t_1, t_2 - t_1, t_3 - t_2, ..., t_l - t_{l-1} \rangle$, and those differences can be compressed using a variable-length encoding (e.g., *PForDelta* [52], *Simple16* [51], *Rice codes* [49]). On the other hand, to avoid traversing all the adjacency lists in reverseNeighbor queries, EdgeLog stores a reverse aggregated graph containing an adjacency list with all the reverse neighbors of each vertex. Therefore, to get the reverse neighbors of vertex v at time t, we first use the reverse adjacency list to obtain the candidate reverse neighbors of v. Then, for each candidate reverse neighbor u, we search for v in its adjacency list and, finally, check if the edge (u, v) is active at time t (using the time-interval list of the edge).

2.2.1. Strengths and weaknesses of EdgeLog

Although EdgeLog is a simple structure using well-known technology, it is expected to be extremely space-efficient when the temporal graph has a low number of edges per vertex and a large number of contacts per edge. In the opposite way, a low number of contacts per edge will have a negative impact on the compression obtained by EdgeLog (as d-gaps become large). Note also that, even with the reverse aggregated graph to find reverse neighbors, the performance is expected to be poor if the number of edges per vertex is high because all their adjacency lists will have to be checked.

EdgeLog was designed to be efficient for activeEdge, directNeighbor, and reverseNeighbor queries, but it could not efficiently answer queries such as: *"Find all the edges that have active contacts at time t"* or *"Find all the edges that have been active only once"*. This is because in such operations, all the adjacency lists must be processed. Also, the applicability of EdgeLog is limited to temporal graphs whose contacts do not temporally overlap; that is, it assumes that a contact of an edge ends before another contact of the same edge starts.

2.3. CET: Compact Events ordered by Time

In CET a temporal graph is a sequence of symbol pairs that represent the changes on the connectivity between vertexes. Each pair represents either the activation or deactivation of an edge along time. Note that a contact of the form (u, v, t_s, t_e) generates two changes: an activation of the edge (u, v) at time t_s, and a deactivation at time instant t_e. The sequence of pairs (S) is composed of the changes on the connectivity of edges (i.e., activations or deactivations produced by all the contacts in the temporal graph) grouped by time instant in increasing order. In Figure 3.a, we show how the sequence of changes of the temporal graph from Figure 2.a is built. We can see that the first two entries of S correspond to the edges (a, d) and (d, b) that are activated at time instant t_0. Next entry corresponds to the activation of the edge (c, d) at time instant t_1. The fourth and fifth entries of S are related to the edge (a, d), which is deactivated at time instant t_2 and activated again at t_3, respectively. The next three entries reflect the changes produced at t_4

when the edges (a, d) and (c, d) are deactivated and (b, a) is activated. Finally, the edges (b, a) and (d, b) are deactivated at time instant t_5.

The activation state of an edge at time instant t is computed by *counting* how many times the pair encoding the edge appears in the subsequence of changes within the time interval between 0 and t (in the closed time interval). As we assume that all edges are inactive at the beginning of the lifetime, the first occurrence of the pair means that the edge becomes active, the second occurrence means that the edge becomes inactive, and so on. In consequence, if the pair appears an odd number of times, it means that the state of the edge is active; otherwise, it is inactive. For example, we can see in Figure 3.a that, because the pair ad occurs three times within interval $[t_0, t_3)$, the edge (a, d) is active at time instant t_3. The direct neighbors of a vertex u at time t are also recovered using the counting strategy, but checking the frequency of the form $(u, *)$, i.e., the pairs whose first component is u. Similarly, the reverse neighbors of v are obtained by counting the pairs that end with v.

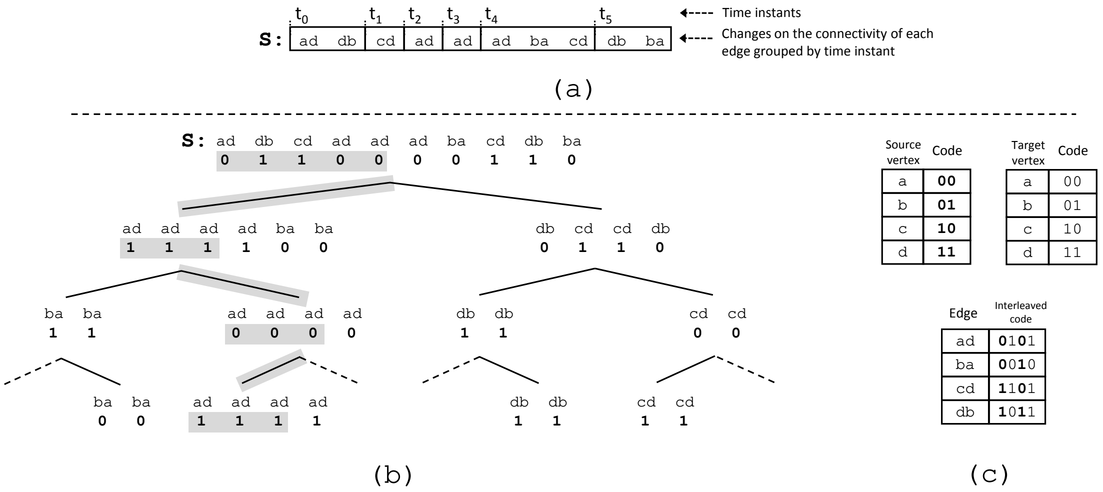
The sequence of pairs that composes S is represented in an Interleaved Wavelet Tree (IWT) [8], a variant of the Wavelet Tree [16, 18] capable of counting the number of occurrences of multidimensional symbols in logarithmic time, while keeping a reduced space. The Wavelet Tree is a balanced binary tree, whose leaves are labeled with symbols in an alphabet Σ, and whose internal nodes handle a range of the alphabet. Each node of the Wavelet Tree represents the sequence as a bitmap with 0s and 1s, depending on the binary code used to represent each symbol in the alphabet Σ. Figure 3.b shows the IWT representation for the sequence of changes S of the temporal graph in Figure 2.a. (For more details on the Wavelet Tree and its applications, refer to [34]).

In the IWT, the pairs of symbols in S are represented by an *interleaved* code that is the result of interleaving the bits (Morton Code [41]) of the codes corresponding to the source and target vertexes of each pair. Figure 3.c shows the interleaved bits for the pairs (corresponding to the edges) of the temporal graph in Figure 2.a. Note that the symbols in pair `ad` are given the codes $\underline{\mathtt{00}}$ and `11` respectively. Therefore, the interleaved code for pair `ad` is $\underline{\mathtt{0}}\mathtt{1}\underline{\mathtt{0}}\mathtt{1}$, and those four bits are represented along the wavelet tree by starting in the root node with the first $\underline{\mathtt{0}}$. Because that bit is a zero, we move to the left child in the next level where we use the second bit of such code. This second bit is `1` and appears at the first position in the bitmap. Subsequently, we move to the right child in the next level, and use the third bit of the code, which is the $\underline{\mathtt{0}}$ at the first position of the bitmap. Finally, we move again to the left child of the node and reach the last level where we set the last bit of the code of `ad`, which is `1`.

The counting operation of a symbol c in the sequence $S[1, i]$[4] is translated into counting operations over the bitmaps in the path of the symbol c. In order to show how the counting algorithm works, let us use the operation $\mathsf{rank}_b(B, i)$.[5] The algorithm works as follows. At the root node, if the first bit of symbol c

[4] For simplicity, we will use the notation $V[i, j]$ to refer to the sequence of elements $\langle V[i], \ldots, V[j]\rangle$.

[5] Given a bitmap B, $\mathsf{rank}_b(B, i)$ computes the number of occurrences of bit b in $B[1, i]$.



Source vertex	Code
a	00
b	01
c	10
d	11

Target vertex	Code
a	00
b	01
c	10
d	11

Edge	Interleaved code
ad	0101
ba	0010
cd	1101
db	1011

Figure 3: The CET data structure representing the temporal graph in Figure 2.a. The top part shows the sequence of changes S. The bottom-left part shows the Interleaved Wavelet Tree (IWT) representation of S. The bottom-right part shows the interleaving bits used to represent pairs of symbols in the IWT.

is 0 (1) we descend through the left (right) child of the node. At the child node, the position i is updated to $\mathsf{rank}_0(B_v, i)$ ($\mathsf{rank}_1(B_v, i)$), if the first bit of the symbol c is 0 (1). This process is recursively repeated until we reach a leaf node. At the leaf node, the number of occurrences of the symbol c corresponds to the updated value of i. In total, this counting strategy requires to answer $O(\log n)$ rank operations over the bitmaps in the path of a symbol. Figure 3.b shows, with a darker background, the bitmaps used to count how many times the symbol ad appears until the fifth position of the sequence.

2.3.1. Strengths and weaknesses of CET

One advantage of CET is its ability to retrieve reverse neighbors with the same time performance of direct neighbors, due to the bi-dimensional representation used for storing the events of activation/deactivation of edges. Indeed, we just need to update the retrieval range to $(*, v)$ to obtain the frequency of neighboring changes of the edges whose target vertex is v.

Another advantage is that the time performance in operations about vertexes and edges is independent of the number of contacts per query in the graph. This is because IWT allows the counting of events in logarithmic time with respect to the number of edges (instead of a sequential counting on the history of events). Due to the temporal arrangement of events of activation/deactivation of edges, operations regarding events on edges are easily obtained by extracting the subsequence related to the time instant of the query. For example, to obtain the edges that change their state at time instant t, we just need to recover the pairs of vertexes in the section related to events occurred at time t.

Despite the advantages of CET, its main weakness is related to the counting strategy used to recover the states of edges when contacts are active for short time intervals. For example, if we want to retrieve a snapshot at a time instant t in a graph where all the edges were activated and deactivated before t, we are forced to retrieve the frequency of all the edges (i.e., visiting each node of the IWT), although only a small fraction of them will actually be in the output. In addition, the frequency counting does not allow the representation of temporal graphs with overlapping contacts. This is because a symbol representing an overlapping contact will be interpreted as a symbol denoting the deactivation of the contact.

2.4. Improved representations of EdgeLog and CET

In the previous section, the descriptions of EdgeLog and CET are given for temporal graphs where edges can freely appear and disappear along time, with no restrictions on the number of contacts per edge. The representation of these data structures can be improved by taking into account properties of the graph being represented. In particular, properties such as the duration and the dynamism of contacts [19].

When all contacts last only one time instant, both EdgeLog and CET can be modified to only store the event that activates an edge because, by definition, all edges will only remain active for one time instant. This small modification invalidates the strategy used to check if the edge is active (i.e., the counting strategy in CET, and the check of the interval in EdgeLog). However, it enables a new strategy to check if an edge is active. For example, in EdgeLog, the list of time intervals per edge is replaced by a list of time instants when an edge was active. Thus, the updated algorithm for checking the activation state of an edge at time t is replaced by verifying if the new list of time instants contains t. Similarly in CET, the activation state of an edge is replaced by checking if the edge appears in the subsequence related with the events occurred at time instant t.

The data structures were also specialized for temporal graphs where each edge has only one contact, and once activated, this contact remains active until the end of the lifetime. In the literature, these graphs are called incremental graphs [13]. With this kind of temporal graphs, the modification is straightforward. As all contacts end at the same time instant (i.e., at the end of the lifetime), it is not necessary to explicitly store the events that deactivate the edges. Caro *et al.* [8] also used this strategy to improve the space cost of both EdgeLog and CET data structures, without the need of updating the query algorithms. Nevertheless, its usefulness depends on how many contacts effectively end at the last time instant of the graph.

3. CSA for Temporal graphs (TGCSA)

The Compressed Suffix Array for Temporal Graphs (TGCSA) is a new data structure adapted from Sadakane's Compressed Suffix Array (CSA) [39] to represent temporal graphs. Unlike EdgeLog and CET, it can represent contacts of the same edge that temporally overlap, what makes TGCSA a more general representation for temporal graphs.

Below we provide a brief presentation of the CSA. Then, we include a detailed description of TGCSA where we show how to create a TGCSA and we present a modification of the main structure (Ψ) of TGCSA

(Section 3.4) that targets at improving its efficiency. Finally, we also show how it solves the most relevant temporal queries.

3.1. Sadakane's Compressed Suffix Array (CSA)

Given a sequence $S[1,n]$ built over an alphabet Σ of length σ, the *suffix array* $A[1,n]$ built on S is a permutation of $[1,n]$ of all the suffixes $S[i,n]$ such that $S[A[i],n] \prec S[A[i+1],n]$ for all $1 \leq i < n$, being $\prec$ the lexicographic ordering [31]. In Figure 4.a, we show the suffix array A for the text $S =$`"abracadabra"`.[6]

Because A contains all the suffixes of S in lexicographic order, this structure permits to search for any pattern $P[1,m]$ in time $O(m \log n)$ with a simple binary search of the range $A[l,r]$ (i.e., $[l,r] \leftarrow binSearch(P)$) that contains pointers to all the positions in S where P occurs. The term m of the cost appears because, at each step of the binary search, one could need to compare up to m symbols from P with those in the suffix $S[A[i], A[i]+m-1]$. Unfortunately, the space needs of A are high.

To reduce the space needs, CSA [39] uses another permutation $\Psi[1,n]$ defined in [17]. For each position j in S pointed by $A[i] = j$, $\Psi[i]$ gives the position z such that $A[z]$ points to $j+1 = A[i]+1$. There is a special case when $A[i] = n$, in which case $\Psi[i]$ gives the position z such that $A[z] = 1$. In addition, two other structures are needed, a vocabulary array $V[1,\sigma']$ with all the different symbols that appear in S, and a bitmap $D[1,n]$ aligned to A so that $D[i] \leftarrow 1$ if $i = 1$ or if $S[A[i]] \neq S[A[i-1]]$ ($D[i] \leftarrow 0$; otherwise). Basically, a 1 in D marks the beginning of a range of suffixes pointed from A such that the first symbol of these suffixes coincides. Therefore, if the i^{th} and $(i+1)^{th}$ ones in D occur in $D[l]$ and $D[r]$, respectively, that is, if $select_1(D,i) = l$ and $select_1(D,i+1) = r$, it means that all the suffixes $S[A[l],n]$, $S[A[l+1],n]$,... $S[A[r-1],n]$ pointed from the entries $A[l,r-1]$ start by the same symbol of the vocabulary. The bitmap D is used to index the vocabulary array. Note that $V[rank_1(D,l)] = V[rank_1(D,x)]\ \forall x \in [l, r-1]$. Recall that $rank_1(D,i)$ returns the number of 1s in $D[1,i]$ and can be computed in constant time using $o(n)$ extra bits [22, 33], whereas $select_1(D,i)$ returns the position of the i^{th} 1 in D. In Figure 4.b, we show the components of the CSA for the text `"abracadabra"`.

By using Ψ, D, and V, it is possible to perform binary search without the need of accessing A or S. Note that, the symbol $S[A[i]]$ pointed by $A[i]$ can be obtained by $V[rank_1(D,i)]$, symbol $S[A[i]+1]$ can be obtained by $V[rank_1(D,\Psi[i])]$, symbol $S[A[i]+2]$ can be obtained by $V[rank_1(D,\Psi[\Psi[i]])]$, and so on. Recall that $\Psi[i]$ basically indicates the position in A that points to the symbol $S[A[i]+1]$. Therefore, by using Ψ, D, and V we can obtain the symbols $S[A[i], A[i]+m-1]$ that we could need to compare with $P[1,m]$ in each step of the binary search.

In principle, Ψ would have the same space requirements as A. Fortunately, Ψ is highly compressible. It was shown to be formed by σ subsequences of increasing values [17] and, therefore, it can be compressed to

[6] The \$ at the end of S is a terminator that must be lexicographically smaller than all the other symbols in S.

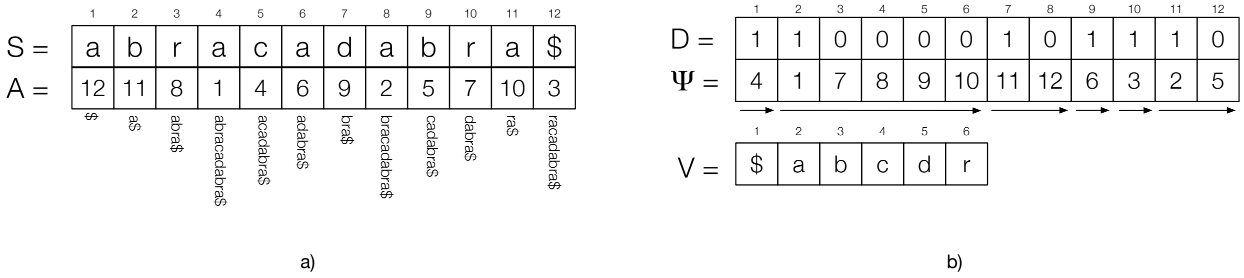


Figure 4: The Compressed Suffix Array for the text S ="abracadabra". The left part shows the Suffix Array (A). The right part depicts the permutation Ψ, the bitmap D, and the vocabulary V. Arrows under the elements of Ψ denote (highly compressible) increasing values. In addition, the inverse of the Suffix Array would be $A^{-1} = \langle 4, 8, 12, 5, 9, 6, 10, 3, 7, 11, 2, 1 \rangle$.

around the zero-order entropy of S [39], and by using δ-codes to represent the differential values, a space cost of $nH_0 + O(n \log \log \sigma)$ bits is obtained. Note that, in Figure 4.b, the arrows under Ψ denote the σ subsequences of increasing values in Ψ. In [35], they showed that Ψ can be split into $nH_k + \sigma^k$ (for any k) *runs* of consecutive values so that the differences within those runs are always 1. This permitted them to combine δ-coding of gaps with run-length encoding (of *1-runs*) yielding higher-order compression of Ψ. In addition, to maintain fast random access to Ψ, absolute samples at regular intervals are kept.

In [14], authors adapted CSA to deal with large (integer-based) alphabets and created the *integer-based CSA* (iCSA). They also showed that, in this scenario, the best compression of Ψ was obtained by combining differential encoding of runs with Huffman [20] and run-length encoding.

As said before, Ψ, D, and V are enough to simulate the binary search for the interval $A[l, r]$ where pattern P occurs without keeping A and S ($[l, r] \leftarrow CSA\_binSearch(P)$). Being $r - l + 1$ the number of occurrences of P in S, this permits to solve the well-known *count* operation. However, if one is interested in *locating* those occurrences in S, A is still needed. In addition, to be able to *extract* the subsequence $S[i, j]$, we also need to keep A^{-1} so that we know the position in A that points to $S[i]$. In practice, only sampled values of A and A^{-1} are stored. Non-sampled values $A[i']$ can be retrieved by applying $i' \leftarrow \Psi[i']$ k-times until a sampled position $A[x]$ is reached (then $A[i'] \leftarrow A[x] - k$). Similarly, sampled values of $A^{-1}[i]$ can be obtained by applying k-times $i' \leftarrow \Psi[i']$ from the previous sample $A^{-1}[x]$ (starting with $i' \leftarrow x$). In this case, $A^{-1}[i] \leftarrow A^{-1}[x] + k$. From this point, the CSA is a *self-index* built on S that replaces S (as any substring $S[i, j]$ could be extracted) and does not need A anymore to perform searches.

3.2. Modifying CSA to represent Temporal Graphs

Recall that a temporal graph is a set $\mathcal{C}$ of contacts of the form $c = (u, v, t_s, t_e)$, where u and v are vertexes (V) and a link or edge between them is active during a time interval $[t_s, t_e)$. Also $[t_s, t_e) \subset \mathcal{T} \times \mathcal{T}$, with $\mathcal{T}$ being the time instants representing the *lifetime* of the graph. In Example 1, we include a set of five contacts that we will use in our discussion below.

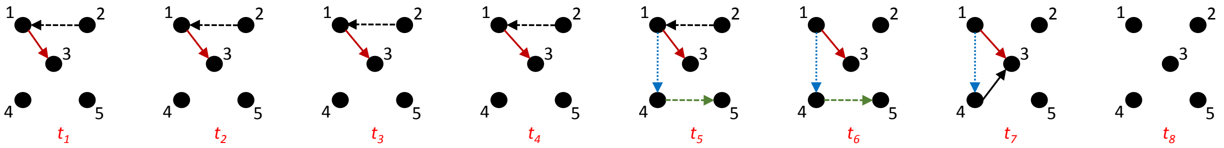


Figure 5: The temporal graph from Example 1.

Example 1. Let us consider the temporal graph in Figure 5 with $|V| = 5$ vertexes numbered $1 \dots 5$ and $|\mathcal{T}| = 8$ time instants numbered $1 \dots 8$. This graph contains the following five contacts: $(1, 3, 1, 8)$, $(1, 4, 5, 8)$, $(2, 1, 1, 6)$, $(4, 3, 7, 8)$, and $(4, 5, 5, 7)$. □

Targeting at using a CSA to obtain a self-indexed representation of a set of contacts (i.e. all their terms regarded as a unique sequence), we discuss in this section two adaptations that we performed. The first one, *using-disjoint-alphabets*, consists in assigning *ids* from disjoint alphabets to both vertexes and time instants. Then, when we perform a query for a given *id* (or a sequence of *ids*) within the CSA, that *id* will correspond either to a source vertex, a target vertex, a starting time instant, or an ending time instant. The second modification consists in *making* Ψ *cyclical* on the elements of the 4-tuple representing a contact. This will permit us to use the regular binary search procedure of the CSA to efficiently search for (and retrieve) those contacts matching some constraints on their terms.

3.2.1. Using disjoint alphabets

Given a set of n contacts, such as the one in Example 1, our procedure to create TGCSA starts by creating an ordered list of the n contacts, so that they are sorted by their first term, then (if they have the same first term) by the second term, and so on. After that, these sorted contacts are regarded as a sequence with $4n$ elements ($S[1, 4n]$), and a suffix array $A[1, 4n]$ is built over it. This is depicted in Figure 6.

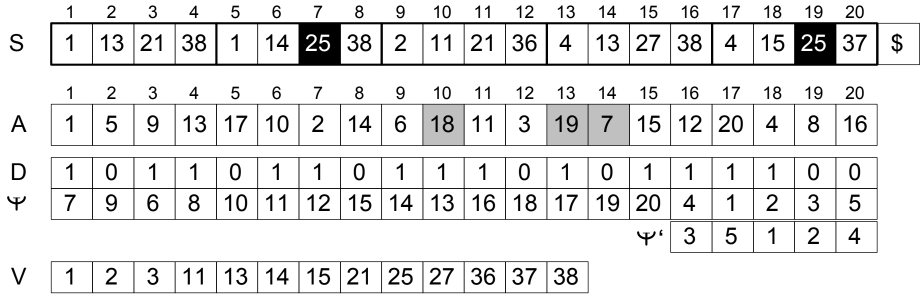
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
S	1	3	1	8	1	4	5	8	2	1	1	6	4	3	7	8	4	5	5	7	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	10	1	5	11	3	9	2	14	13	17	6	18	19	7	12	20	15	4	8	16

Figure 6: Suffix Array for the contacts from Example 1 using a unique alphabet $\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

If S were made up of text, A and S (or a CSA built on S) would be enough to perform searches for any word or text substring $P[1, m]$. In such case, if we looked for the occurrences of symbol 5 (i.e $P[1, 1] = \langle 5 \rangle$), $A[12, 14] = \langle 18, 19, 7 \rangle$ would indicate that there are 3 occurrences of symbol 5. They occur at $S[18]$, $S[19]$, and $S[7]$ respectively. However, in our scenario, when we search for symbol 5 (i.e. $P[1, 1] = \langle 5 \rangle$) we have to be able to distinguish among the source vertex 5, the target vertex 5, the starting time instant 5 and the ending time instant 5. This would require accessing all the entries $A[i]$, $\forall i \in [12, 14]$, and checking the positions in S they are pointing to. In practice, if $A[i] \mod 4 = 1$ then $A[i]$ points to a source vertex; otherwise, if $A[i] \mod 4 = 2$ then it points to a target vertex, and so on. However, this procedure would ruin the $O(m \log n)$ search time that would now become $O(m \log n + occ)$, where *occ* is the number of occurrences of the query pattern in S.

A simple workaround to the problem above consists in using disjoint alphabets for the four terms in a contact. In our case, we use alphabets $\Sigma_1, \Sigma_2, \Sigma_3$, and Σ_4 satisfying that $\Sigma_1 \prec \Sigma_2 \prec \Sigma_3 \prec \Sigma_4$ ($\prec$ indicates lexicographic order). Note that we can always replace vertexes and time instants in the original set of contacts by new *ids* satisfying this property. For example, in Figure 7, we have created a new sequence S where: (i) the *ids* of the source vertexes have been kept as they were initially ($\Sigma_1 = \{1, 2, 4\}$); (ii) the *ids* of the target vertexes have been added +10 ($\Sigma_2 = \{\underline{1}1, \underline{1}3, \underline{1}4, \underline{1}5\}$); (iii) the *ids* of the starting time instants have been added +20 ($\Sigma_3 = \{\underline{2}1, \underline{2}5, \underline{2}7\}$); and (iv) the *ids* of the ending time instants have been added +30 ($\Sigma_4 = \{\underline{3}6, \underline{3}7, \underline{3}8\}$). Now, when we build the suffix array for the new S, we can search for either the pattern $\langle 5 \rangle$, $\langle 15 \rangle$, $\langle 25 \rangle$, or $\langle 35 \rangle$, depending on if we want to find the occurrences of the term 5 that corresponds to a source vertex, target vertex, starting time, or ending time, respectively. For example, we can see in the figure that when we are searching for the starting time 5, we can simply add +20 to its *id* and actually use the suffix array (or the CSA) to look for $P = \langle \underline{2}5 \rangle$ obtaining its two occurrences pointed by $A[13]$ and $A[14]$. However, to search for the target vertex 5 we would add +10 to its *id* and found that $A[10]$ points to its unique occurrence in S. In any case, we retain the original $O(m \log n)$ search time as expected.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
S	1	13	21	38	1	14	25	38	2	11	21	36	4	13	27	38	4	15	25	37	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	1	5	9	13	17	10	2	14	6	18	11	3	19	7	15	12	20	4	8	16
D	1	0	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	1	0	0
Ψ	7	9	6	8	10	11	12	15	14	13	16	18	17	19	20	4	1	2	3	5
															Ψ‘	3	5	1	2	4

V	1	2	3	11	13	14	15	21	25	27	36	37	38

Figure 7: Suffix Array for the contacts from Example 1 using disjoint alphabets. The structures Ψ, D, and V for the corresponding CSA are also depicted.

An interesting by-product that arises from the use of disjoint alphabets is that, since values from Σ_i are smaller than those from Σ_j ($\forall i < j$), the first quarter of entries in A ($A[1, n]$) will point to the first terms of all the contacts ($S[1 + 4k], \forall k \in [0, n)$), the next n entries in A ($A[n + 1, 2n]$) to the second terms ($S[2 + 4k], \forall k \in [0, n)$), and so on. Consequently, the first quarter of entries of Ψ ($\Psi[1, n]$) will point to a position in the range $[n + 1, 2n]$, because in the indexed sequence S each symbol $u \in \Sigma_1$ is followed by a symbol $v \in \Sigma_2$, and so on. In this way, each entry in the last quarter of Ψ will point to a position in the range $[1, n]$, corresponding to the first quarter of entries in A.

In our example, recall we have $n = 5$ contacts. We can see that the entries in the four quarters of A discussed above match that: $\forall i \in [1,5], A[i] \mod 4 = 1$; $\forall i \in [6,10], A[i] \mod 4 = 2$; $\forall i \in [11,15], A[i] \mod 4 = 3$; and $\forall i \in [16,20], A[i] \mod 4 = 0$. In addition, in Figure 7, we have also included the Ψ structure that arises when we build the corresponding CSA. In this case, we can also verify that it holds that: $\forall i \in [1,5], \Psi[i] \in [6,10]$; $\forall i \in [6,10], \Psi[i] \in [11,15]$; $\forall i \in [11,15], \Psi[i] \in [16,20]$; and $\forall i \in [16,20], \Psi[i] \in [1,5]$. This property will be of interest in the following section.

3.2.2. Modifying Ψ to make it cyclical on the terms of each contact

Recall that in a regular CSA, once we know that the i^{th} entry in the underlying suffix array A points to a position $z = A[i]$ of the source sequence S, we can recover the entries $S[z], S[z+1], ...$ from the original sequence S as $S[z] = S[A[i]] \leftarrow V[rank_1(D,i)]$, the next symbol as $S[z{+}1] = S[A[i]{+}1] \leftarrow V[rank_1(D,\Psi[i])]$, the next symbol as $S[z+2] = S[A[i]+2] \leftarrow V[rank_1(D,\Psi[\Psi[i]])]$, and so on. Therefore, as shown in Section 3.1, by using Ψ, D, and V, we can binary search for any pattern P obtaining the range $[l,r]$ so that $\forall i \in [l,r], A[i]$ points to the positions in S where P can be found. Then, from those positions on, we could recover the source data of the suffixes $S[A[i], ...]$ that start with P. Unfortunately, this mechanism allows us to recover the source data only forward-wise (not backwards), and this is not enough in our scenario because we typically want to search for the contacts that match a given constraint and then we want to retrieve all their terms.

To clarify the issue above, consider, for example, when we look for the contacts whose target vertex is $v = 5$ ($P = \langle \underline{1}5 \rangle$), then we obtain its unique occurrence at the position 10 ($A[10]$). Consequently, to retrieve the terms of that contact (u, v, t_s, t_e), we would compute: $v \leftarrow V[rank_1(D,10)] = \underline{1}5$; $t_s \leftarrow V[rank_1(D,\Psi[10])] = \underline{2}5$; $t_e \leftarrow V[rank_1(D,\Psi[\Psi[10]])] = \underline{3}7$. However, $u' \leftarrow V[rank_1(D,\Psi[\Psi[\Psi[10]]])]$ would not recover the first term of the current contact, but the first term of the next contact in S. As in a regular CSA, to retrieve u, we would have to access $A[10] = 18$ to know that the target vertex v occurs at position $S[18]$, and consequently the source vertex u should be retrieved from $S[18-1]$. Now, because S is not actually kept in the CSA, to extract $S[17]$, we have to know the entry x in A such that $A[x] = 17$. We can use that $x = A^{-1}[17] = \mathbf{5}$.[7] Finally, by using $u \leftarrow V[rank_1(D,\mathbf{5})] = 4$ we have fully recovered the contact $(4, \underline{1}5, \underline{2}5, \underline{3}7)$ we were searching for. To sum up, the previous procedure would make it necessary to use not only Ψ, D, and V, but also A and A^{-1} as explained in Section 3.1. Fortunately, we can modify Ψ in such a way that it allows us to move circularly from one term to the next term within a given contact.

Recall that, due to our disjoint alphabets, if $A[i] (i \in [3n+1, 4n])$ points to the last term of the j^{th} contact, then $\Psi[i]$ would store the position in A pointing to the first term of the following $(j+1)^{th}$ contact ($A[i] + 1 = A[\Psi[i]]$), which would be in the range $[1,n]$. For TGCSA, we modified these pointers in the last

[7] Recall $A^{-1}[j] = x$ indicates which position x from A points to the j^{th} entry of S. That is, such that $A[x] = j$.

quarter of Ψ in such a way that, instead of pointing to the position $x = A[\Psi[i]]$ corresponding to the first term of the following contact, they point to the first term of the same contact; that is, $A[\Psi'[i]] = x - 1$ or $A[\Psi'[i]] = n$ if $x = 1$. The modified quarter of Ψ is depicted as Ψ' in Figure 7. In this way, starting at any entry i in Ψ, and following the pointers $\Psi[i]$, $\Psi[\Psi[i]]$, and $\Psi[\Psi[\Psi[i]]]$, all the elements of the current contact can be retrieved, but no entry from any other tuple will be reached. Due to this modification, in the example above, we can recover $u \leftarrow V[rank_1(D, \Psi[\Psi[\Psi[10]]])]$, and A and A^{-1} are no longer needed.

Note that it is not possible now to traverse the whole CSA by just using Ψ because consecutive applications of the Ψ function will cyclically obtain the four elements of the corresponding contact. However, this small change in Ψ to make it cyclical on the terms of each contact, brings additional interesting searching capabilities that we will exploit in Section 3.5.

3.3. Detailed construction of TGCSA

Once we have explained the need of using disjoint alphabets and the reason why we use a modified Ψ, in this section we explain the actual procedure to build our TGCSA. In Figure 8, we depict all the structures involved in the creation of a TGCSA representing the temporal graph in Example 1.

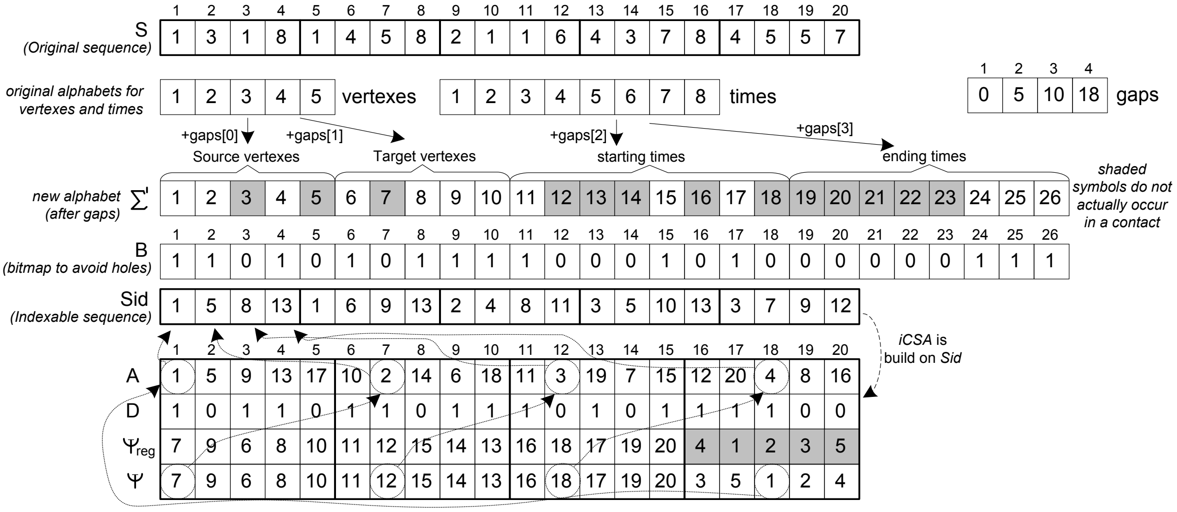
As indicated above, the first step to build a TGCSA is to create a sequence S with the ordered n contacts. Hence we obtain, $S[1, 4n] = \langle u^1, v^1, t_s^1, t_e^1, u^2, v^2, t_s^2, t_e^2, \ldots, u^n, v^n, t_s^n, t_e^n \rangle$.[8]

The second step involves defining a reversible mapping that enables us to use disjoint alphabets. Let us assume we have $\nu = |V|$ different vertexes and $\tau = |\mathcal{T}|$ time instants. It is possible to define a reversible mapping function that maps the terms of any original contact $c = (u, v, t_s, t_e)$ to $c' = (u, v + \nu, t_s + 2\nu, t_e + 2\nu + \tau)$. To achieve this, we define an array $gaps[1, 4] \leftarrow \langle 0, \nu, 2\nu, 2\nu + \tau \rangle$ and a set with elements $c'[i] \leftarrow c[i] + gaps[i]$, $\forall i = 1 \ldots 4$. This mapping defines four ranges of entries in an alphabet Σ' for both vertexes and time instants such that $|\Sigma'| = 2\nu + 2\tau$. Note that vertex i is mapped to either the integer i or $i + \nu$ depending on whether it is the source or target vertex of an edge. Similarly, the time instant t is mapped to either $t + gaps[3]$ or $t + gaps[4]$. This allows us to distinguish between starting/ending vertexes/time instants by simply checking the range where their value falls into.

Observe that even though vertex i always exists in the temporal graph, either source vertex $u' = i + gaps[1] = i$ or target vertex $v' = i + gaps[2]$ could actually not be used. Similarly, a time instant t' could not occur either as an initial or as an ending time of a contact, yet we could be interested in retrieving all the edges that were active at that time t'.

To overcome the existence of holes in the alphabet Σ', a bitmap $B[1, 2\nu + 2\tau]$ is used. We set $B[i] \leftarrow 1$ if the symbol i from Σ' occurs in a contact, and $B[i] \leftarrow 0$; otherwise. Therefore, each of the four terms

[8] Note that the ordering is not relevant because we have a *set* of contacts. Therefore, we will assume that contacts are sorted by the first term, then by the second one, and so on.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S (Original sequence)	1	3	1	8	1	4	5	8	2	1	1	6	4	3	7	8	4	5	5	7

original alphabets for vertexes and times: 1 2 3 4 5 vertexes; 1 2 3 4 5 6 7 8 times

	1	2	3	4
gaps	0	5	10	18

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
new alphabet Σ' (after gaps)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
B (bitmap to avoid holes)	1	1	0	1	0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Sid (Indexable sequence)	1	5	8	13	1	6	9	13	2	4	8	11	3	5	10	13	3	7	9	12
A	1	5	9	13	17	10	2	14	6	18	11	3	19	7	15	12	20	4	8	16
D	1	0	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	1	0	0
Ψ_{reg}	7	9	6	8	10	11	12	15	14	13	16	18	17	19	20	4	1	2	3	5
Ψ	7	9	6	8	10	11	12	15	14	13	16	18	17	19	20	3	5	1	2	4

Figure 8: Structures involved in the creation of a TGCSA for the temporal graph in Example 1.

within a contact (u, v, t_s, t_e) will correspond to a 1 in B. Then an alphabet Σ of size $\sigma = rank_1(B, 4n)$[9] is created containing the positions in B where 1 occurs. For each symbol $i \in \Sigma'$, a *mapID(i)* function assigns an integer $id \in \Sigma$ to i, so that $id \leftarrow$ *mapID(i)* $= rank_1(B, i)$ if $B[i] = 1$, and $0 \leftarrow$*mapID*(i) if $B[i] = 0$. The reverse mapping function is provided via *unmapID*$(id) = select_1(B, id)$.[10]

At this point, a sequence of ids $Sid[1, 4n]$ can be created by setting $Sid[i] \leftarrow$*mapID*$(S[i]+gaps[i \mod 4+1]), \forall i = 1 \ldots 4n$. Indeed, being $type = 1, 2, 3, 4$, respectively, the types of source vertexes, target vertexes, starting time instants, and ending time instants from the original sequence S, we can map any source symbol i from S into Sid by $id =$*getmap*$(i, type) \leftarrow rank_1(B, i + gaps[type])$. Similarly, the reverse mapping obtains $i =$*getunmap*$(id, type) \leftarrow select_1(B, id) - gaps[type]$.

Once we have made up our indexable sequence Sid, an iCSA is built over it.[11] Then, as discussed in Section 3.2.2, we modified the array Ψ in our TGCSA to allow Ψ to move circularly from one term to the next one within the same contact. To do this, we simply have to modify the last quarter of the regular Ψ array so that, $\forall i = 3n + 1 \ldots 4n,\ \Psi[i] \leftarrow ((\Psi[i] - 2) \mod n) + 1$. This small change brings an interesting property that allows us to perform a query for any term of a contact in the same way. We use the iCSA to binary search for a term of a contact(s), obtaining a range $A[l, r]$, and then by circularly applying Ψ up to three times, we can retrieve the other terms of the contact(s).

To sum up, TGCSA consists of a bitmap B, and the structures D and Ψ of the iCSA. In practice, B is

[9] Recall $rank_1(B, i)$ returns the number of 1s in $B[1, i]$.

[10] Recall $select_1(B, i)$ computes the position of the i^{th} 1 in B.

[11] We actually added four integers set to *zero* that make up a dummy contact (0,0,0,0) at the beginning of *Sid*. This is required to avoid limit-checks at query time.

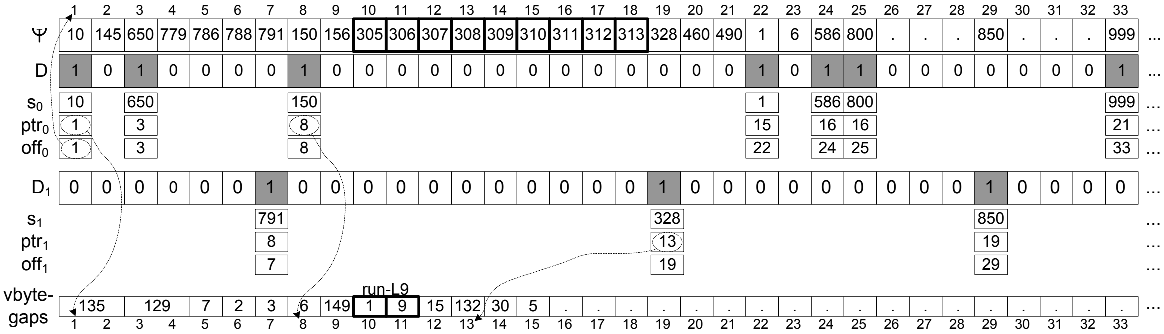


Figure 9: Example of vbyte-rle representation of Ψ assuming $t_\Psi = 4$.

compressed using Raman *et al.* strategy[12] [37], and for D we used a faster bitmap representation [14] using $1.375|D|$ bits. For the representation of Ψ we also used the best option (named huff-rle-opt) that samples Ψ at regular intervals and then differentially encodes the remaining values [14]. Yet, we also created an alternative representation for Ψ that is discussed in Section 3.4.

3.4. A more suitable representation of Ψ for temporal graphs: vbyte-rle *strategy*

The regular representation of Ψ is based on sampling the Ψ array at regular intervals (one sample every t_Ψ entries) and then, differentially encoding the remaining values between two samples. In [14], they studied different alternative encodings for the non-sampled values, and showed that the best space/time trade-off in a text-indexing scenario was reported by coupling run-length encoding of *1-runs* (sequences of $+1$ values) with bit-oriented Huffman (huff-rle-opt approach). In practice, they used t_Ψ Huffman codes to indicate the presence of 1-runs of length $1 \dots t_\Psi$. They also reserved n_{sv} Huffman codes to represent short gaps (where n_{sv} is a parameter typically set to 2^{14}). Finally, being ω the machine word size, $2 \times \omega$ additional Huffman codes are used as escape codes to mark the number of bits needed to either represent a large positive gap (g) or a negative gap ($-g$). In both cases, such a escape code is followed by g represented with $\lceil \log_2 g \rceil$ bits.

In this paper, we present a new strategy to represent Ψ, that we called vbyte-rle, where we try to speed up the Ψ access performance at the cost of using a little more space. An example of the structure for the resulting Ψ representation is shown in Figure 9. We also use sampling and differentially encode non-sampled values. Yet, we made some changes with respect to the traditional Ψ representations (i.e., huff-rle-opt), which are summarized as follows:

- We used *vbyte* (byte-aligned) codes [48] rather that bit-oriented Huffman codes to differentially encode non-sampled values. This should result in around one order of magnitude improvement in decoding speed when sequential values of Ψ are to be retrieved. Note that in the bottom part of Figure 9, we

[12] Raman et al strategy allows both $select_1$ and $rank_1$ in $O(1)$ time and requires $|B|\mathcal{H}_0(B) + o(|B|)$ bits.

include a sequence of byte-oriented codewords (either 1 or 2-byte codewords in our example) that are used to represent the gaps from the original Ψ structure. It can also contain a pair of codewords for the pair $\langle 1, L \rangle$ to encode a *1-run* of length L. Of course, using byte-aligned rather than bit-oriented codes will imply a loss in compression effectiveness.

- We do not sample Ψ at regular intervals. Instead of that, we keep samples aligned with the *ones* in bitmap D, that is, there is a sample at the beginning of the interval in $[l_c, r_c]$ corresponding to each symbol c. This modification brings three main advantages:

  (ii) We ensure that $\Psi[l_c]$ is always sampled, whereas with the traditional representation of Ψ the previous sampled position could be in the range $[l_c - t_\Psi + 1, l_c]$. Therefore, l_c was sampled with probability $1/t_\Psi$. Note that, in TGCSA, a typical access pattern to Ψ during searches (see Section 3.5) consists in traversing all the values $\Psi[l_c, r_c]$ once we know the interval $[l_c, r_c]$ corresponding to a given symbol c. This requires decoding gaps from the previous sample to l_c in huff-rle-opt to obtain synchronization at value $\Psi[l_c]$, and sequentially decoding gaps from there on. Since l_c is always sampled in vbyte-rle, we avoid that synchronization cost.

  (ii) While in the traditional representation of Ψ, the differential sequence $\Psi[j] - \Psi[j-1]$ ($j \in [2, 4n]$) could contain up to $\sigma/2$ negative values (when $i = l_c$ belongs to a symbol c and $j - 1 = r_{c-1}$ to symbol $c - 1$)[17], the vbyte-rle representation does not deal with negative values because $j = l_c$ is always a sampled position.

  (iii) We do not break *1-runs*. Recall that *1-runs* could occur mainly within the range $[l_c, r_c]$ corresponding to a given symbol c. Because our first-level sampling stores only a sample at position l_c, *1-runs* are no longer split. This is interesting for both space and access time because a unique codeword can be used to represent a large 1-run sequence. In our example, we can see that the codewords $\langle 1, 9 \rangle$ in vbytegaps$[10, 11]$ represent the *1-run* of length 9 within $\Psi[10, 18]$. That is, we do not break the *1-run* every $t_\Psi = 4$ values.

In Figure 9, we can see that samples consist of a triple of values $\langle s', \mathsf{ptr}', \mathsf{off}' \rangle$ that are aligned with the *ones* in D: s' indicates the absolute value, ptr' is a pointer to vbytegaps sequence, and off' indicates the index of the sampled position. In practice, these values are set in three arrays $s_0[1, \sigma]$, $\mathsf{ptr}_0[1, \sigma]$, and $\mathsf{off}_0[1, \sigma]$, respectively, such that if $\Psi[j] = s'$ is sampled, we set $s_0[rank_1[D, j]] = s'$, $\mathsf{off}_0[rank_1[D, j]] = j$, and $\mathsf{ptr}_0[rank_1[D, j]] = x$.

Note that the absolute values s' are kept explicitly in s_0 and are not represented within the sequence vbytegaps (exactly as in huff-rle-opt). For example, $\Psi[1] = 10$ is stored at the first entry of s_0, and the first codeword in vbytegaps represents value 135, which corresponds to the gap $\Psi[2] - \Psi[1]$. Hence, no codeword in vbytegaps is associated with the sampled value $\Psi[1]$. Note also that x is the position in vbytegaps that we have to access to recover values $\Psi[j+1,...]$. In our example, we can see that $\Psi[9]$ can be recovered by accessing the previous sampled value $s_0[3] = 150 = \Psi[8]$, then accessing sequence vbytegaps at position $x = \mathsf{ptr}_0[3] = 8$ to obtain the gap $\Psi[9] - \Psi[8]$ by $gap = decode\_vbyte(x) = 6$. Finally, we recover $\Psi[9] = 150 + 6 = 156$. As an important remark, observe that given a symbol c, we will use $\mathsf{off}_0[c]$ to obtain the starting sampled position l_c for the range $\Psi[l_c, r_c]$. We could skip storing array off_0 as we can compute $l_c = select_1(D, c)$. This introduces a space/time trade-off that we discuss in the next section.

Despite the advantages of the sampling structures described above, our representation has also a main drawback: we cannot parameterize the number of samples we want to use. Thus, we can be using a rather too dense sampling for infrequent symbols (consequently, we expect that compression will suffer in datasets with very large vocabularies ($\sigma \approx n$)), or we can be using a very sparse sampling for frequent symbols c, as they will have only one sample at the beginning of the corresponding interval $[l_c, r_c]$. This fact could slow down the access to an individual position $\Psi[j]$, with $j \in [l_c + 1, r_c]$. To overcome this, we added a second-level sampling where we sample the positions $l_c + t_\Psi, l_c + 2 \times t_\Psi, \ldots$ (t_Ψ is again the sampling interval). We use a bitmap D_1 (see Figure 9) to mark the positions of these samples in Ψ, and, aligned with the *ones* in D_1, arrays $s_1[1, n_1]$, $\mathsf{ptr}_1[1, n_1]$, and $\mathsf{off}_1[1, n_1]$ keep the sampling data (n_1 is the number of *ones* in D_1). This second-level sampling works exactly like the first-level one with the exception that sampled values are also retained in the vbytegaps sequence. This redundant data is kept to allow us to sequentially decode the whole values $\Psi[l_c + 1, r_c]$ belonging to a given symbol c without the need to access the second-level sampling data. This is of interest when we want to retrieve a range of consecutive values from Ψ instead of simply recovering an individual value.

3.4.1. Comparing the Space/time trade-off of vbyte-rle *with* huff-rle-opt.

We run experiments to compare the space/time trade-off obtained by huff-rle-opt against vbyte-rle and vbyte-rle-select (the latter is the variant of vbyte-rle where arrays off_0 and off_1 are not stored). We tuned these representations using four different sampling values for Ψ. In particular, we used values $t_\Psi \in \{256, 64, 16, 8\}$ (from sparser to denser sampling, respectively). In addition, we include in the comparison a non-compressed baseline representation for $\Psi[1, 4n]$ (we refer to it as plain) that represents each entry of Ψ with $\lceil \log 4n \rceil$ bits and provides direct access to any position.

In Figures 10 and 11, we compare the space (shown as the number of bits needed to represent each entry in Ψ) and time (in μs per entry reported) required to access all the values in Ψ for three different scenarios. In the plots labeled by [B1] and [B2], we assume that the ranges $[l_c, r_c]$ for all the symbols $c \in [1, \sigma]$ are known and we perform a buffered access to retrieve the values $\Psi[l_c, r_c]$ for all these symbols. In scenario [B2], we only retrieve those values $\Psi[l_c, r_c]$ for symbols occurring at least 8 times (hence $r_c - l_c - 1 \geq 8$). In these *buffered* scenarios, synchronization is done once to obtain $\Psi[l_c]$ (except in plain that has direct access and does not require synchronization at all) and from there on, we apply sequential decoding of subsequent values. In the last scenario (plot labeled [S1]), we show the cost of accessing Ψ at individual positions (hence synchronization, for the compressed variants, is required for each access to Ψ). We access sequentially all the positions in Ψ, $\forall j \in [1..4n]$.

We have run tests for all the datasets in Table 2 (described in Section 4) and show results here for datasets: I.Comm.Net, Powerlaw, Flickr-Data, and Wikipedia-Links. We do not show plots for ba* datasets because they obtain as fairly identical shapes as those for I.Comm.Net (yet with slightly different x-axis).

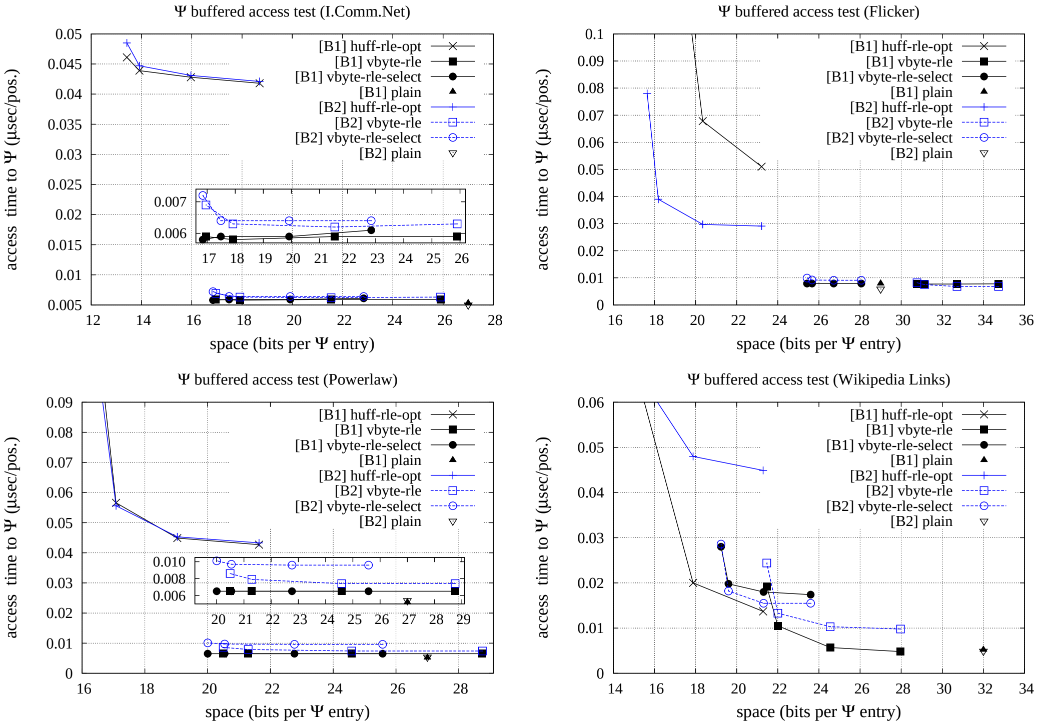


Figure 10: Space/time trade-off for buffered access to Ψ.

We can see that the cost of the synchronization required by huff-rle-opt and the slower decoding of bit-Huffman in comparison with vbyte make huff-rle-opt more than 5 times slower than vbyte-rle when decoding all the entries of Ψ corresponding to a given symbol c. In Section 3.5, we will see that this particular operation appears in most TGCSA query algorithms (a *for loop* after a binary search that returns the range of Ψ values for a given symbol). The shortcoming of this speed up at recovering Ψ values is that the overall size of Ψ increases by around 20-25%. As we expected, it can be seen that in the Flickr-Data dataset, due to the large vocabulary size of this dataset in comparison with the number of contacts, the vbyte-rle representation becomes unsuccessful because a plain representation of Ψ would even be smaller. We also include results for the vbyte-rle-select counterpart. In this case, we do not explicitly store arrays off_0 and off_1, and we require $select_1$ operations to know the position j in Ψ corresponding to the i-th sample. In general, when the number of synchronization operations is small (this occurs when σ is small), vbyte-rle-select offers an interesting space/time trade-off. In particular, we can see that it typically yields the same performance of plain baseline representation while requiring 5-40% less space.

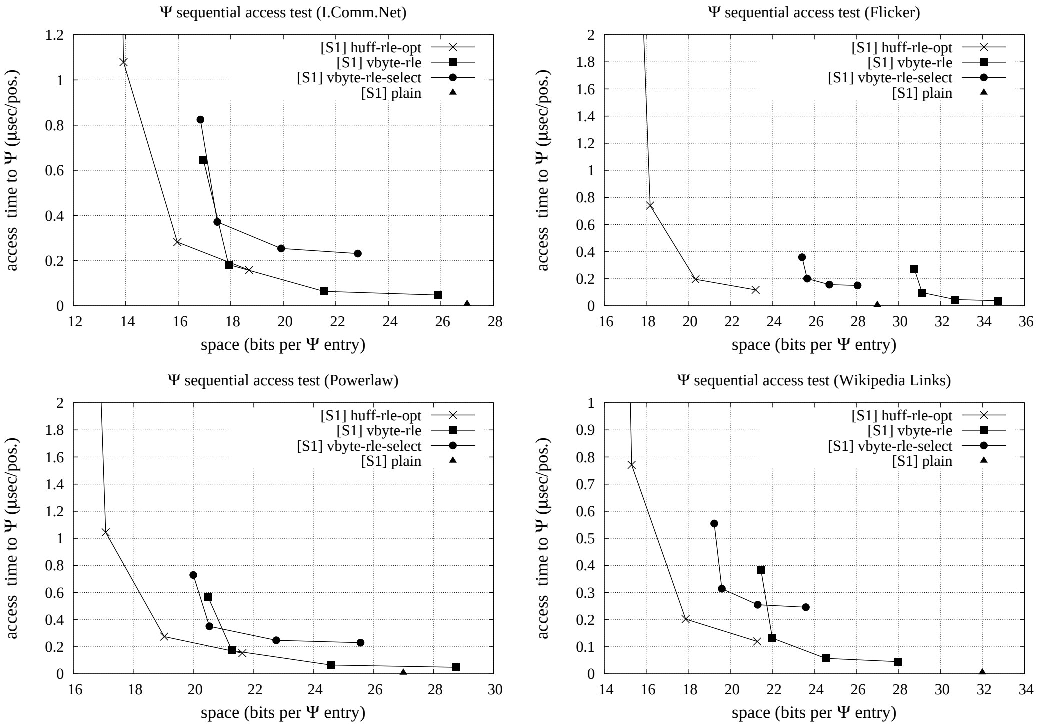


Figure 11: Space/time trade-off for sequential access to Ψ.

Unfortunately, not all the accesses to Ψ performed at query time will follow a sequential pattern in TGCSA. In that case, the previous buffered retrieval of Ψ values is not applicable, and we need to perform many random accesses to positions within Ψ. Accessing random positions implies that each access to $\Psi[j]$ must initially check if j is a sampled position. This is accomplished by checking if $j \mod t_\Psi = 0$ in huff-rle-opt or if $access(D, j) = 1$ in vbyte-rle.[13] In that case $\Psi[j] = s_0[\lfloor j/t_\Psi \rfloor]$ or $\Psi[j] = s_0[rank_1(D, j)]$, respectively. Yet, in vbyte-rle we could still have a sampled value if $access(D_1, j) = 1$ and we would obtain the sampled value by $\Psi[j] = s_1[rank_1(D_1, j)]$.

In Figure 11, we can see that when we access individual positions of Ψ, vbyte-rle and its two-level sampling approach is still able to improve the Ψ access time of huff-rle-opt. In general, huff-rle-opt using $t_\Psi = 8$ (very dense setup) obtains similar values than vbyte-rle with $t_\Psi = 64$ (a relatively sparse setup). Yet, in vbyte-rle

[13] $access(D, j)$ returns the value of the bit at position j in the bitmap D.

we still have room to decrease access time at the cost of using a denser tuning. As expected, in this scenario, vbyte-rle-select becomes unsuccessful, and plain is unbeatable due to its direct access capabilities.

3.5. Performing queries in TGCSA

We can take advantage of the iCSA capabilities at search time to solve all the typical queries in a temporal graph regarding *direct* and *reverse* vertexes from contacts that are active at a given time instant t (directNeighbor and reverseNeighbor queries, respectively). Basically, we binary search the range in $A[l, r]$ for the given source or target vertex, and for each position $i \in [l, r]$, we apply Ψ circularly up to the third or four ranges where we can check whether or not the starting-time and ending-time constrains hold. In Figure 12, we include the pseudocode of the algorithms to answer both directNeighbor and reverseNeighbor queries. Note that they are almost identical with the difference that, in the former, the search begins in the range $A[lu, ru]$ corresponding to the source vertex, whereas in the latter the starting range $A[lv, rv]$ corresponds to the target vertex being searched for.

Note that the accesses to Ψ in the *for* loop in line 8 traverse consecutive positions $i \in [lu, ru]$ (or $i \in [lv, rv]$ for reverse neighbors). Recall that we do not have *direct access* to all the values of Ψ, but only to sampled positions and the remaining values require accessing the previous sample (to gain synchronization on either the Huffman-compressed or Vbyte-compressed stream of gaps) and sequentially decoding gaps from there on up to the desired position (see Section 3.4 for more details). Therefore, although it is not stated in the pseudocode, we have boosted the access to consecutive positions in Ψ (i.e. $\Psi[lu, ru]$) by implementing a *buffered access* method to Ψ. By using this buffered access method to recover $\Psi[lu, ru]$, we only access the sample before position lu, then we synchronize at value $\Psi[lu]$,[14] and from there on, we sequentially decompress the remaining values in $\Psi[lu + 1, ru]$. The other accesses to Ψ (i.e., $\Psi[x]$ and $\Psi[y]$ in directNeighbor) are completely random and there is no room for optimization there. We will also apply this buffered access to Ψ in the loops on the following algorithms.

When comparing queries, activeEdge is expected to be faster than directNeighbor because we can binary search for a phrase $u \cdot v$ rather than by a unique vertex u, hence returning a much shorter initial range. The pseudocode for solving the activeEdge operation at a given time instant is included in Figure 13.

To solve snapshot queries given a time instant t, which return the set of active contacts (u, v, t_1, t_2) such that $t_1 \leq t < t_2$, we can binary search the starting and ending-time intervals: $[lt_s, rt_s] \leftarrow \mathit{CSA\_binSearch}(\mathit{getmap}(t, 3))$ and $[lt_e, rt_e] \leftarrow \mathit{CSA\_binSearch}(\mathit{getmap}(t, 4))$. All the contacts pointed by $A[2n + 1, rt_s]$ hold $t_1 \leq t$ and those in $A[rt_e + 1, 4n]$ hold $t_2 > t$. Therefore, $\forall i \in [2n + 1, rt_s]$, if $\Psi[i] > rt_e$, we recover the source and target vertexes by $\Psi[\Psi[i]]$ and $\Psi[\Psi[\Psi[i]]]$, respectively. The original values are obtained via *getunmap()*. Figure 14 includes the pseudocode to solve snapshot queries.

[14] Recall $\Psi[lu]$ is always sampled in vbyte-rle and no synchronization costs are involved.

```
**DirectNeighbors** $(vrtx, t)$ //neighbors $(v)$ of $vrtx$ in contact $(vrtx,v,t_1,t_2)$ s.t. $t_1 \leq t < t_2$
$u \leftarrow$ **getmap**$(vrtx,\ typeVertex = 1)$;   // maps into the final alphabet without holes
**if** $u = 0$ **then return** $\emptyset$;   // vertex does not appear as source vertex
$neighbors \leftarrow \emptyset$;
$t_s \leftarrow$ **getmap**$(t, typeStartTime = 3)$; $t_e \leftarrow$ **getmap**$(t, typeEndTime = 4)$;
$[lu, ru] \leftarrow$ **CSA_binSearch**$(u)$;   // range $A[lu, ru]$ for vertex $u$
$[lt_s, rt_s] \leftarrow$ **CSA_binSearch**$(t_s)$;   // range $A[lt_s, rt_s]$ for starting time $t_s$
$[lt_e, rt_e] \leftarrow$ **CSA_binSearch**$(t_e)$;   // range $A[lt_e, rt_e]$ for ending time $t_e$
**for** $i \leftarrow lu$ **to** $ru$   // checks time intervals for each occurrence of $u$
    $x \leftarrow \Psi[i]$;   // $x$ = position of target vertex
    $y \leftarrow \Psi[x]$;   // $y$ = position of starting time
    **if** $(y \leq rt_s)$ **then**
        $z \leftarrow \Psi[y]$;   // $z$ = position of ending time
        **if** $(z > rt_e)$ **then**
            $neighbors \leftarrow neighbors \cup \{$**getunmap**$(x, typeRevVertex = 2)\}$;
**return** $neighbors$;
```

```
**ReverseNeighbors** $(vrtx, t)$ //reverse neighbors $(u)$ of $vrtx$ in contact $(u,vrtx,t_1,t_2)$ s.t. $t_1 \leq t < t_2$
$v \leftarrow$ **getmap**$(vrtx,\ typeRevVertex = 2)$;   // maps into the final alphabet without holes
**if** $v = 0$ **then return** $\emptyset$;   // vertex does not appear as target vertex
$rev\_neighbors \leftarrow \emptyset$;
$t_s \leftarrow$ **getmap**$(t, typeStartTime = 3)$; $t_e \leftarrow$ **getmap**$(t, typeEndTime = 4)$;
$[lv, rv] \leftarrow$ **CSA_binSearch**$(v)$;   // range $A[lv, rv]$ for vertex $v$
$[lt_s, rt_s] \leftarrow$ **CSA_binSearch**$(t_s)$;   // range $A[lt_s, rt_s]$ for starting time $t_s$
$[lt_e, rt_e] \leftarrow$ **CSA_binSearch**$(t_e)$;   // range $A[lt_e, rt_e]$ for ending time $t_e$
**for** $i \leftarrow lv$ **to** $rv$   // checks time intervals for each occurrence of $v$
    $y \leftarrow \Psi[i]$;
    **if** $(y \leq rt_s)$ **then**
        $z \leftarrow \Psi[y]$;
        **if** $(z > rt_e)$ **then**
            $u \leftarrow \Psi[z]$;
            $rev\_neighbors \leftarrow rev\_neighbors \cup \{$**getunmap**$(u, typeVertex = 1)\}$;
**return** $rev\_neighbors$;
```

Figure 12: Obtaining the direct neighbors (directNeighbor) and the reverse neighbors (reverseNeighbor) of a vertex in a contact that is active at time t.

```
**activeEdge** $(vrtx_u, vrxt_v, t)$ //checks if exists $(vrtx_u, vrtx_v, t_1, t_2)$ s.t. $t_1 \leq t < t_2$
$u \leftarrow$ **getmap(**$vrtx_u,\ typeVertex = 1$**)**;    // maps into final alphabet without holes
$v \leftarrow$ **getmap(**$vrtx_v,\ typeRevVertex = 2$**)**;
**if** $u = 0$ **or** $v = 0$ **then return** $false$;    // edge does not exist
$t_s \leftarrow$ **getmap**$(t, typeStartTime = 3)$; $t_e \leftarrow$ **getmap**$(t, typeEndTime = 4)$;
$[l_{uv}, r_{uv}] \leftarrow$ **CSA_binSearch(**$uv$**)**;    // range $A[l_{uv}, r_{uv}]$ for edge $uv$
$[lt_s, rt_s] \leftarrow$ **CSA_binSearch(**$t_s$**)**;    // range $A[lt_s, rt_s]$ for starting time $t_s$
$[lt_e, rt_e] \leftarrow$ **CSA_binSearch(**$t_e$**)**;    // range $A[lt_e, rt_e]$ for ending time $t_e$
**for** $i \leftarrow l_{uv}$ **to** $r_{uv}$    // checks time intervals for each occurrence of $uv$
    $x \leftarrow \Psi[i]$;
    $y \leftarrow \Psi[x]$;
    **if** $(y \leq rt_s)$ **then**
        $z \leftarrow \Psi[y]$;
        **if** $(z > rt_e)$ **then**
            **return** $true$;
**return** $false$;
```

Figure 13: Checking if an edge is active at time instant t (activeEdge operation).

Queries regarding activation/deactivation events at a given time instant t in the graph can be solved very efficiently. A unique binary search allows TGCSA to find all the contacts that have an event at time t. In the case of the deactivatedEdge operation, the binary search looks for the range $[lt_e, rt_e] \subseteq [3n+1, 4n]$ corresponding to contacts (u, v, t_1, t_2) where $t_2 = t$, whereas for the activatedEdge operation we obtain an interval $[lt_s, rt_s] \subseteq [2n+1, 3n]$ corresponding to those contacts where $t_1 = t$. From these intervals, we apply Ψ circularly (twice or three times, respectively) up to reaching the values u and v corresponding to the source and target vertex of these contacts. In Figure 15, we include the pseudocode for the deactivatedEdge operation. Note that the activatedEdge operation would be similar but the loop would traverse positions $i \in [lt_s, rt_s]$ with $x \leftarrow \Psi[\Psi[i]]$ in line 5.

Taking a look at the pseudocodes presented for TGCSA query operations, we can see that we are using the following operations during searches: (i) *getmap* and *getunmap* calls that imply performing *rank* and *select* over B and can be solved in $O(1)$ time. (ii) A call to *CSA_binSearch*(p) that requires $O(\log n)$ time and returns a range $A[l, r]$ containing the occurrences of a pattern p. Up to two additional calls to *CSA_binSearch*(p') could be needed depending on the query, also requiring $O(\log n)$ time. (iii) A loop traversing the $L = r - l + 1$ entries in $A[l, r]$ that involves only $O(1)$ operations, typically *getunmap* and accesses to Ψ. The exception is the snapshot operation that traverses always $L \leftarrow rt_s - 2n$ entries. To sum up, the temporal queries in TGCSA can be solved in time $O(\log n + L)$.

Dealing with interval queries. As indicated in Section 2, we have shown how TGCSA handles directNeighbor, reverseNeighbor, activeEdge, snapshot, activatedEdge, and deactivatedEdge queries at a given time instant t. Yet, these operations could be easily extended in TGCSA to time intervals. In queries that refer to checking

```
**snapshot** ($t$) //returns all the edges ($u$,$v$) s.t. $\exists$ contact ($u$,$v$,$t_1$,$t_2$) where $t_1 \leq t < t_2$
$t_s \leftarrow$ **getmap**($t, typeStartTime = 3$);
$t_e \leftarrow$ **getmap**($t, typeEndTime = 4$);
$[lt_s, rt_s] \leftarrow$ **CSA_binSearch**($t_s$);   // range $A[lt_s, rt_s]$ for starting time $t_s$
$[lt_e, rte] \leftarrow$ **CSA_binSearch**($t_e$);   // range $A[lt_e, rt_e]$ for ending time $t_e$
$snap \leftarrow \emptyset$;
**for** $i \leftarrow 2n + 1$ **to** $rt_s$
    $z \leftarrow \Psi[i]$;
    **if** ($z > rt_e$) **then**
        $x \leftarrow \Psi[z]$;
        $y \leftarrow \Psi[x]$;
        $u \leftarrow$ **getunmap**($x$, $typeVertex = 1$);
        $v \leftarrow$ **getunmap**($y$, $typeRevVertex = 2$);
        $snap \leftarrow snap \cup \{(u, v)\}$;
**return** $snap$;
```

Figure 14: snapshot operation returns the edges that are active at time instant t.

```
**DeactivatedEdges** ($t$) //returns all the edges ($u$,$v$) s.t. $\exists$ contact ($u$,$v$,$t_1$,$t_2$) where $t_2 = t$
$t_e \leftarrow$ **getmap**($t, typeEndTime = 4$);
$[lt_e, rte] \leftarrow$ **CSA_binSearch**($t_e$);   // range $A[lt_e, rt_e]$ for ending time $t_e$
$edges \leftarrow \emptyset$;
**for** $i \leftarrow lt_e$ **to** $rt_e$
    $x \leftarrow \Psi[i]$;
    $y \leftarrow \Psi[x]$;
    $u \leftarrow$ **getunmap**($x$, $typeVertex = 1$);
    $v \leftarrow$ **getunmap**($y$, $typeRevVertex = 2$);
    $edges \leftarrow edges \cup \{(u, v)\}$;
**return** $edges$;
```

Figure 15: deactivatedEdge operation returns the edges that were deactivated at time t.

the connectivity between vertexes (the first three ones), one would be interested in contacts (u, v, t_1, t_2) occurring not only at a given time instant t, but during a whole time interval $[t, t')$; that is, $[t, t') \subseteq [t_1, t_2)$ (this is called *strong semantics* for intervals in the literature). A different option (referred to as *weak semantics*) consists in reporting those contacts occurring at least at some point of $[t, t')$; that is, such that it holds $[t_1, t_2) \cap [t, t') \neq \emptyset$. Note that for queries retrieving the changes on connectivity (activatedEdge and deactivatedEdge), it makes no sense to distinguish between *weak* and *strong semantics*, and we would be interested in simply checking if the connectivity changed at some point of the interval $[t, t')$.

If we focus on queries constrained to an interval $[t, t')$ under *strong semantics*, to solve directNeighbor queries, we should only adapt the temporal constraint so that contacts match $(y \leq rt_s)$ AND $(z > rt_e)$. Yet, in this case, rt_s and rt_e must be the right hand of the ranges $[lt_s, rt_s]$ and $[lt_e, rt_e]$ corresponding to t and

t', respectively. Therefore, we should modify line 4 in the pseudocode of Figure 12 to set $t_s \leftarrow getmap(t, 3)$ and $t_e \leftarrow getmap(t', 4)$; instead of $t_s \leftarrow getmap(t, 3)$ and $t_e \leftarrow getmap(t, 4)$. Algorithms reverseNeighbor (in Figure 12) and activeEdge (in Figure 13) could be adapted by simply modifying their line 4 in the same way.

Although not considered in previous works, we could also think of defining a snapshot operation to recover the contacts that were active during the interval $[t, t')$. Under *strong semantics*, this interval-wise snapshot could be defined such that it would retrieve the contacts that were activated before t and deactivated after t'. Therefore, we could see this operation as the *union* of the results of snapshot at a given time t_x, $\forall t \leq t_x < t'$. This case would only require modifying line 2 from Figure 14, to again set $t_e \leftarrow getmap(t', 4)$.

For deactivatedEdge queries at time interval $[t, t')$ (see Figure 15), we would have to replace lines $1-4$ by the following: First, we map both t and t' values to the ending times t_s and t_e; that is, $t_s \leftarrow getmap(t, 4)$ and $t_e \leftarrow getmap(t', 4)$. Then, we binary search for the corresponding intervals in TGCSA: $[lt_s, rt_s] \leftarrow$*CSA_binSearch(t_s)* and $[lt_e, rt_e] \leftarrow$*CSA_binSearch(t_e)*. And finally, all the ending time instants between lt_s and $lt_e - 1$ correspond to contacts deactivated within $[t, t')$. Therefore, we have to traverse the entries in that range, that is, we would iterate (line 4) **for** $i \leftarrow lt_s$ **to** $lt_e - 1$. A similar adaptation is possible for activatedEdge queries.

We can also deal with *weak semantics* in TGCSA. As an example, we show how to adapt directNeighbor queries to this scenario. The rest of operations can be adapted similarly. Now, a directNeighbor query for a given vertex u constrained to an interval $[t, t')$ must retrieve any vertex v from a contact (u, v, t_1, t_2) that were active at some time instant within $[t, t')$. Therefore, these contacts must match the time constraint $(t_1 < t')$ AND $(t_2 > t)$. Focusing on Figure 12, because we need to compare the starting time instant of the contacts (t_1) with t', and their ending time instant (t_2) with t, we would have to replace line 4 to set $t_s \leftarrow getmap(t', 3)$ and $t_e \leftarrow getmap(t, 4)$. Finally, the sentences in lines $11-14$ in the for-loop must be changed to modify the temporal condition. In practice, we replace them by:

```
(11) if  (($y < lt_s$) then
(12)     $z \leftarrow \Psi[y]$;
(13)     if  ($z > rt_e$) then
(14)         $neighbors \leftarrow neighbors \cup \{$getunmap$(x, typeRevVertex = 2)\}$;
```

3.6. Strengths and weaknesses of TGCSA

The strong expressive power of TGCSA is probably its main advantage with respect to other state-of-the-art representations such as EdgeLog and CET ([12, 8]). Recall TGCSA can really represent any set of contacts, including contacts of a given edge that temporally overlap.

Another important property is that it can answer queries over any term of a contact in the same way; that is, searching for all the contacts of a source node u is performed exactly with the same algorithm as searching for all the contacts starting in a specific time instant t: first a binary search is performed over one

Operation	CET	EdgeLog	TGCSA
directNeighbor	$O(d \log \nu)$	$O(d + c)$	$O(\log n + d\ t_\Psi)$
reverseNeighbor	$O(d \log \nu)$	$O(d^2 + c)$	$O(\log n + d\ t_\Psi)$
activeEdge	$O(\log \nu)$	$O(d + c')$	$O(\log n + c'\ t_\Psi)$
activatedEdge	$O(k \log \nu)$	$O(n)$	$O(\log n + k\ t_\Psi)$
deactivatedEdge	$O(k \log \nu)$	$O(n)$	$O(\log n + k\ t_\Psi)$
snapshot	$O(e \log \nu)$	$O(n)$	$O(\log n + n\ t_\Psi)$

Table 1: Comparison of the costs of the search operations in TGCSA, CET, and EdgeLog [8]. The term d denotes the degree of the vertex of the query in the aggregated graph. The term c (c') is the number of contacts related to the vertex (edge) in the query. The term k is the number of contacts starting or ending at the time instant of the queries activatedEdge and deactivatedEdge. Finally, e is the number of different edges in the aggregated graph.

of the four sectors of the array Ψ, depending on the term of the contact that is searched for (i.e., bounded in the query), to locate the area devoted to that value, and then, for each of the entries in that area, Ψ is applied three times to recover the other components of each contact. The overall search time is $O(\log n + L)$, where L is the length of the range reported by the initial binary search (with the exception of the snapshot operation). Although other data structures are more efficient for some types of queries, TGCSA has a more regular behavior over all types of queries. Table 1 compares the cost of the query operations in TGCSA with those of the most representative state-of-the-art counterparts: CET and EdgeLog. Furthermore, for graphs whose contacts last for only one time instant (*Point-contact Temporal Graphs*), the behavior of TGCSA improves because the suffix array only has three sections and Ψ has only to be applied twice to recover each contact.

Observe that within the section devoted to any symbol, in each of the four quarters of Ψ, all the pointers are always growing, which is a property that allows good compression. However, this property is also the main drawback of this representation. When there are few occurrences of the symbols in the vocabulary; that is, when the vocabulary is huge and there are few occurrences of each symbol, Ψ will not be very compressible. As shown in the experimental results, the compression in some synthetic collections is poor when the relative number of contacts per time instant is low or when the number of edges per node is low. In these cases, the increasing areas of Ψ are small. Therefore, the differences between pointer values are rather big, and consequently, not very compressible.

4. Experimental results

We ran several experiments with real and synthetic temporal graphs. Table 2 gives the main characteristics of these graphs including: the name of each dataset, the numbers of their vertexes, edges, and contacts, and the length of the graphs' lifetime. In addition, we show the numbers of contacts per vertex, edges per vertex, and contacts per edge, respectively. Finally, we show the space of a plain representation of the original datasets (in MiB) assuming that each contact was represented with four 32-bit integers ($Size^{u32}$), or with $2\lceil\log\nu\rceil + 2\lceil\log\tau\rceil$ bits ($Size^b$).

Dataset	Vertexes	Edges	Lifetime	Contacts	c/ν	e/ν	c/e	Size^{u32}	Size^b
	(ν) $\times10^3$	(e) $\times10^3$	(τ) $\times10^3$	(c) $\times10^3$				(MiB)	(MiB)
I.Comm.Net	10	15,940	10	19,061	1.2	1594.1	1.2	291	127
Flickr-Data	6,204	71,345	167,943	71,345	1.0	11.5	1.0	1,089	868
Powerlaw	1,000	31,979	1	32,280	1.0	32.0	1.0	493	231
Wikipedia-Links	22,608	564,224	414,347	731,468	1.3	25.0	1.3	11,161	9,417
ba100k10u1000	100	941	100	941,408	1000.0	9.4	1000.0	14,365	7,631
ba1M10p12	1,000	9,735	1,000	50,177	5.2	9.7	5.2	766	479
ba1M10u5	1,000	9,735	1,000	48,679	5.0	9.7	5.0	743	464
ba1M10u50	1,000	9,735	1,000	486,792	50.0	9.7	50.0	7,428	4,642

Table 2: Description of temporal graphs used in our experiments.

The dataset `I.Comm.Net` is a synthetic dataset where short communications between random vertexes are simulated. The dataset `Powerlaw` is also synthetic; it simulates a power-law degree graph, where few vertexes have many more connections than the other vertexes (following a power-law distribution), but with a short lifetime. `Flickr-Data` is a real dataset that consists in an incremental temporal graph that indicates the time instant in which two people became friends in the Flickr social network, with a temporal granularity given in seconds, and a lifetime that starts with the creation of Flickr and ends in April 2008. The dataset `Wikipedia-Links` contains the history of links between articles from the English version of the Wikipedia with a time granularity given also in seconds. This dataset corresponds to a history dump of the Wikipedia[15] downloaded on 2014-03-04. Other synthetic datasets were built by first setting a given degree distribution on the aggregated graph, and then assigning a number of contacts to each edge that follows a given distribution. The time interval of each edge was selected uniformly over the lifetime. We used the Barabási-Albert model [1] (see datasets ba* below) to generate a powerlaw degree distribution. Then we used a uniform (U) and a pareto (P) distribution to assign the number of contacts per edge. Pareto distributions were generated with $\alpha = 1.2$, whereas for the uniform distributions, we created graphs with $5, 50$, and 1000 contacts per edge.

Even though TGCSA allows us to deal with datasets where contacts could have overlapping times, in order to allow the comparison with EdgeLog and CET, the datasets above have contacts with no time overlapping. Yet, these datasets still allow us to show the behavior of TGCSA.

Our tests were run on a machine with two Intel(R) Xeon(R) Intel(R) E5620 CPUs @ 2.40GHz. They sum eight-cores (sixteen siblings), yet our experiments run in a single core. The system has 64GB DDR3 RAM @ 1066Mhz. The operating system was Ubuntu 12.04 (kernel Linux version 3.2.0-79-generic), and the compiler used was gcc 4.6.3 (option -O3). Time measures refer to CPU user-time.

In the following sections, we include experiments to compare both the space and time performance of

[15] Downloaded from `http://dumps.wikimedia.org/enwiki/`.

CET, EdgeLog, and TGCSA. In particular, we compare the time performance for the following queries: directNeighbor, reverseNeighbor, activatedEdge, deactivatedEdge, and snapshot at a given time instant.

For EdgeLog and CET we used the same source code as in [8]. Therefore, EdgeLog uses an implementation in *C* of *PForDelta* from the PolyIRTK project,[16] and the best space was obtained by tuning *PForDelta* block-size to 32 (rather than the usual 128 value). In addition, when the number of elements to compress is smaller than the block size, *PForDelta* is replaced either by the word-wise *Simple16* coding [51], when $\tau < 2^{28}$, or by *Rice codes* [49] when $\tau \geq 2^{28}$ (both are also available in the PolyIRTK project).

The Interleaved Wavelet Tree in CET is implemented as a Wavelet Matrix [11], which keeps a good space/time trade-off for sequences with large alphabets. Compressed bitmaps [37, 10] included in CET can be found in the Compact Data Structures Library (*libcds*[17]).

The implementation of TGCSA is an adaptation of the implementation of iCSA[18] [14]. The bitmap representation used by D is exactly the same than in iCSA, whereas bitmap B uses the same *libcds* implementation of Raman *et al.* [37] in CET. In addition, TGCSA uses huff-rle-opt strategy to represent Ψ. We will show results including three different configurations by setting the sampling parameter on Ψ to values $t_\Psi \in \{16, 64, 256\}$. Note that $t_\Psi = 16$ (Ψ_{16} in advance) corresponds to the densest sampling and Ψ_{256} to the most sparse one. We have also included results for TGCSA-VB, the variant of TGCSA that uses the vbyte-rle strategy to represent Ψ. Again, we set $t_\Psi \in \{16, 64, 256\}$ for the second-level sampling in TGCSA-VB.

A further detail is related to the `Flickr-Data` dataset. In this case, the *ending time* of all the contacts is set to the same value (the last time instant in the timeline). Therefore, we could avoid representing this value explicitly. We have adapted TGCSA, and also used adapted versions of CET and EdgeLog [8], in order to index only the first three elements of the contacts. This reduces (rather slightly) the size of the resulting structures, and also improves their overall performance. We will include both the regular TGCSA and the TGCSA built over 3-element contacts (TGCSA-3R) when showing time performance on the `Flickr-Data` dataset.

4.1. Space comparison

Table 3 shows the comparison of TGCSA and TGCSA-VB against CET, EdgeLog, and a *plain* baseline representation using $2\lceil\log\nu\rceil + 2\lceil\log\tau\rceil$ bits. Finally, we also include *gzip* in that table (run over the source plain-text-wise datasets) because this will allow us to compare the compressibility obtained by iCSA in our datasets with that originally obtained when dealing with text [14]. Note that for the `Flickr-Data` dataset we include two rows. The first one refers to the space obtained by the structures when we assume

[16] Available at `http://code.google.com/p/poly-ir-toolkit/`.

[17] Available at https://github.com/fclaude/libcds

[18] Available at `http://vios.dc.fi.udc.es/indexing/wsi`

Dataset	TGCSA			TGCSA-VB			CET	EdgeLog	*plain*	*gzip*
	Ψ_{16}	Ψ_{64}	Ψ_{256}	Ψ_{16}	Ψ_{64}	Ψ_{256}			*bit-wise*	*def*
I.Comm.Net	69.36	61.17	59.17	91.68	77.17	73.34	52.28	82.48	56.00	66.13
Flickr-Data	82.90	77.60	76.29	139.12	132.80	131.34	49.71	187.39	79.00	–
	89.65	81.01	78.84	–	–	–	–	–	102.00	97.89
Powerlaw	81.66	73.85	71.92	103.88	90.69	87.50	67.97	129.88	60.00	70.01
Wikipedia-Links	78.02	67.73	65.14	104.69	94.51	92.34	57.75	137.08	108.00	50.67
ba100k10u1000	74.62	64.47	61.93	96.52	79.65	75.18	43.63	18.22	68.00	49.88
ba1M10p12	87.42	79.51	77.54	109.03	93.96	92.04	56.35	65.77	80.00	69.63
ba1M10u5	92.74	85.10	83.22	115.70	100.68	98.82	61.37	67.98	80.00	72.34
ba1M10u50	89.20	80.18	77.98	112.41	95.97	91.51	56.56	37.26	80.00	68.24

Table 3: Space comparison shown as number of bits per contact (*bpc*). For Flick-Data the first row assumes contacts with no ending time.

contacts containing only three elements, hence excluding the final time instant (the plain baseline uses only $2\lceil \log \nu \rceil + \lceil \log \tau \rceil$ *bpc* $=$ 79*bpc*). In the case of TGCSA, this corresponds to the variant TGCSA-3R. The space needs are shown as the number of bits needed to represent each contact (*bpc*).

Even tough an iCSA-based self-index built on English text typically reached the compression of *gzip* [14], the compressibility of temporal graphs is not so good. Actually, the large number of 1-runs that appear in Ψ when dealing with text is now much smaller in the TGCSA, and we are not able to reach the compression levels of *gzip* in most cases. As expected, taking into account the experiments regarding the vbyte-rle representation of Ψ that we showed in Section 3.4, we typically obtain that TGCSA-VB requires around 20-30% more space than TGCSA. With the Flickr-Data dataset, the space usage of TGCSA-VB is huge due to the non-parameterizable first-level sampling and the large vocabulary in such dataset.

Focusing on EdgeLog, we see that it is also unsuccessful when the number of contacts per edge is very small. However, when there are few edges and the number of contacts per edge grows, it becomes very interesting because its inverted lists become highly compressible. TGCSA shows a more stable behavior, with reasonable space needs in most cases. It does not require as much space as EdgeLog when the number of contacts per edge is small, but it cannot cope with many contacts per edge because Ψ is irregular, as discussed above.

With respect to CET, we can see that CET obtains always a more compact representation than TGCSA, and becomes the best overall alternative if one aims at obtaining little space cost (with the exception of ba100k10u1000 and ba1M10u50 datasets). Yet, in the following sections we will show that TGCSA typically performs faster.

4.2. Time comparison: Direct and Reverse neighbors operations

This section presents the evaluation of the time performance to retrieve the set of direct and reverse neighbors that were active at a given time instant. To evaluate these operations, we generated $2,000$ queries

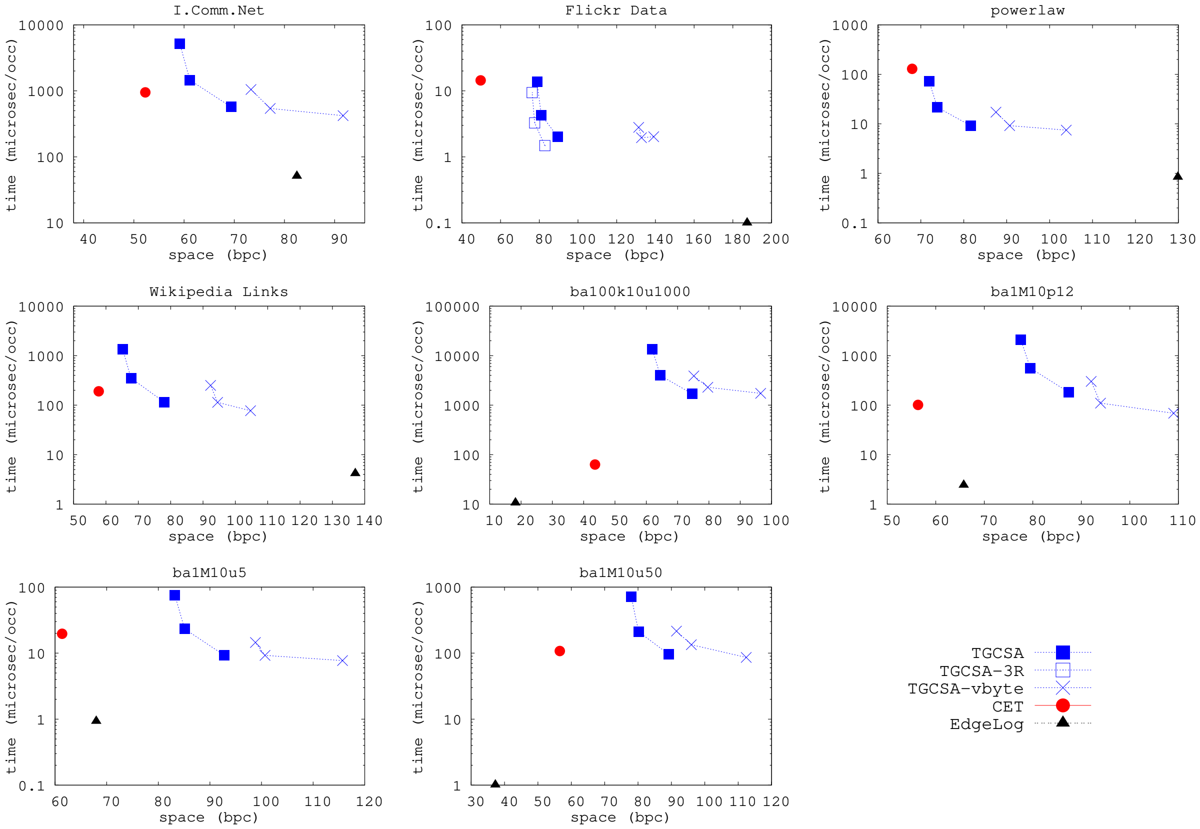


Figure 16: Space/time trade-off for directNeighbor queries.

by randomly choosing 2,000 contacts from each graph dataset. For each selected contact (u, v, t_s, t_e), we took the pairs (u, t_s) and (v, t_s) to create the query patterns to use for directNeighbor and reverseNeighbor, respectively. The time performance is measured in μs per contact reported and the space usage in bits per contact (as in Table 3).

Figures 16 and 17 show the results. Despite the fact that TGCSA uses always more space than CET to represent our temporal graphs, we can see that both techniques have similar performance at solving directNeighbor queries when the number of contacts per vertex is small. The only exception is the synthetic dataset `ba100k10u1000` where there are 1,000 direct neighbors for each vertex, which forces TGCSA to sequentially check a lot of probably unsuccessful direct neighbors. We can see that in the `Powerlaw` and `Flickr-Data` datasets, TGCSA clearly overcomes CET. Considering TGCSA-VB, it is typically faster (around 3-5 times) than TGCSA when using the densest sampling setup. Yet, assuming that we could tune TGCSA-VB and TGCSA to use similar space, TGCSA-VB would always be slower than TGCSA because it would use a very sparse sampling.

Finally, in the plot corresponding to the `Flickr-Data` dataset, we show the gain in both space and time that TGCSA-3R obtains with respect to TGCSA. As shown, it is worth not to explicitly represent the

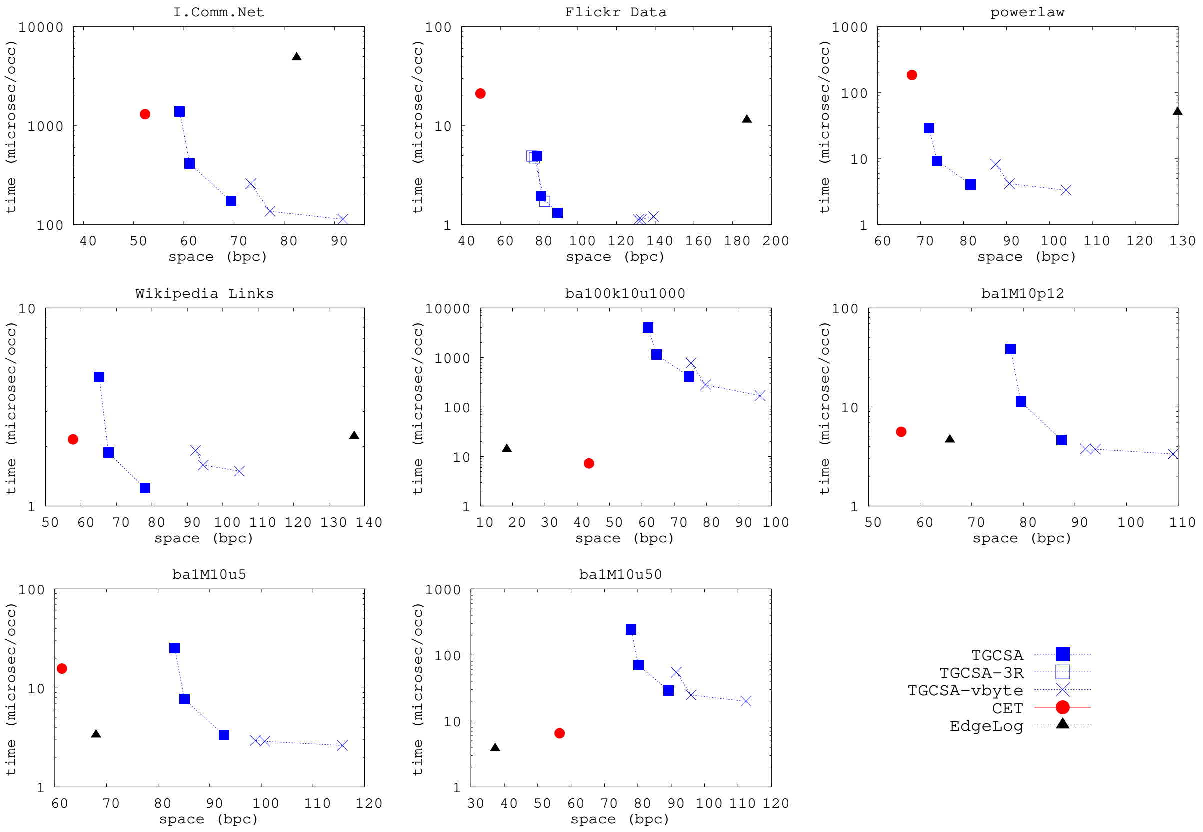


Figure 17: Space/time trade-off for reverseNeighbor queries.

fourth component (ending-time) of the contacts for incremental graphs. When comparing TGCSA-3R with EdgeLog, results show that solving directNeighbor queries is indeed one of the main strengths of EdgeLog, because EdgeLog only needs to traverse the corresponding adjacency list.

With respect to reverseNeighbor queries, we can see similar results as for directNeighbor queries when comparing CET with TGCSA. Yet, now we can see that TGCSA (and TGCSA-VB) are clearly faster to solve reverse- instead of direct-neighbors operations, whereas the results of CET are very similar for both types of operations.

It is easy to understand why TGCSA is faster at reverseNeighbor queries than at directNeighbor operations. Note that the time instants are the third and forth elements of the contacts, and the source vertex and target vertex are, respectively, the first and second elements. Therefore, in the case of directNeighbor operations TGCSA must traverse a range $[l, r]$ of source vertexes $i \in [l, r]$ and it has to apply $\Psi\Psi[i]$ and $\Psi^3[i]$, respectively, to reach the starting and ending time instants (in order to either accept or discard the contact due to the time constraints). In the case of reverseNeighbor operations, the traversal starts in the range of the target vertexes, and we save one application of Ψ to reach the time components of the contact (we apply $\Psi[i]$ and $\Psi[\Psi[i]]$, respectively, to reach the starting and ending time instants of the contact). Recall that in these operations, the first application of Ψ to obtain $\Psi[i]$ is performed over a range of consecutive positions $i \in [l, r]$, which benefits from the buffered access to Ψ. From there on, obtaining $\Psi\Psi[i]$ or $\Psi\Psi\Psi[i]$ requires, respectively, one or two (slower) additional random accesses to Ψ.

As expected, EdgeLog performance drastically worsens in reverseNeighbor queries. Yet, the use of the reverse aggregated graph still allows a good performance in most cases. The exception is in the `I.Comm.Net` graph, where the number of edges per vertex is high. In the other cases, the number of edges per vertex is relatively small (from 10 to 30) and the time performance does not degrade in excess.

4.3. Time comparison: Activation and deactivation at a given time instant

This section shows the performance of activatedEdge and deactivatedEdge queries; that is, retrieving the set of edges that have been either activated or deactivated at a given time instant. For the evaluation, we generated $2,000$ random time instants, uniformly distributed over the lifetime of the corresponding graph. Again, time measures are shown as the average time in μs per contact reported.

Figures 18 and 19 show the results. We can see that these types of operations are probably the best scenario for TGCSA because they are solved by a single binary search to find the given time instant. For example, in the case of deactivatedEdge queries at time t, the binary search returns an interval $[lt, rt]$ corresponding to all the contacts that are deactivated at time t. Therefore, for each $i \in [lt, rt]$, we apply Ψ circularly to recover the corresponding source vertex ($u \leftarrow \Psi[i]$) and target vertex ($v \leftarrow \Psi\Psi[i]$). Similarly, for activatedEdge queries at time instant t, we apply Ψ circularly from a starting interval within the third part of the suffix array in TGCSA.

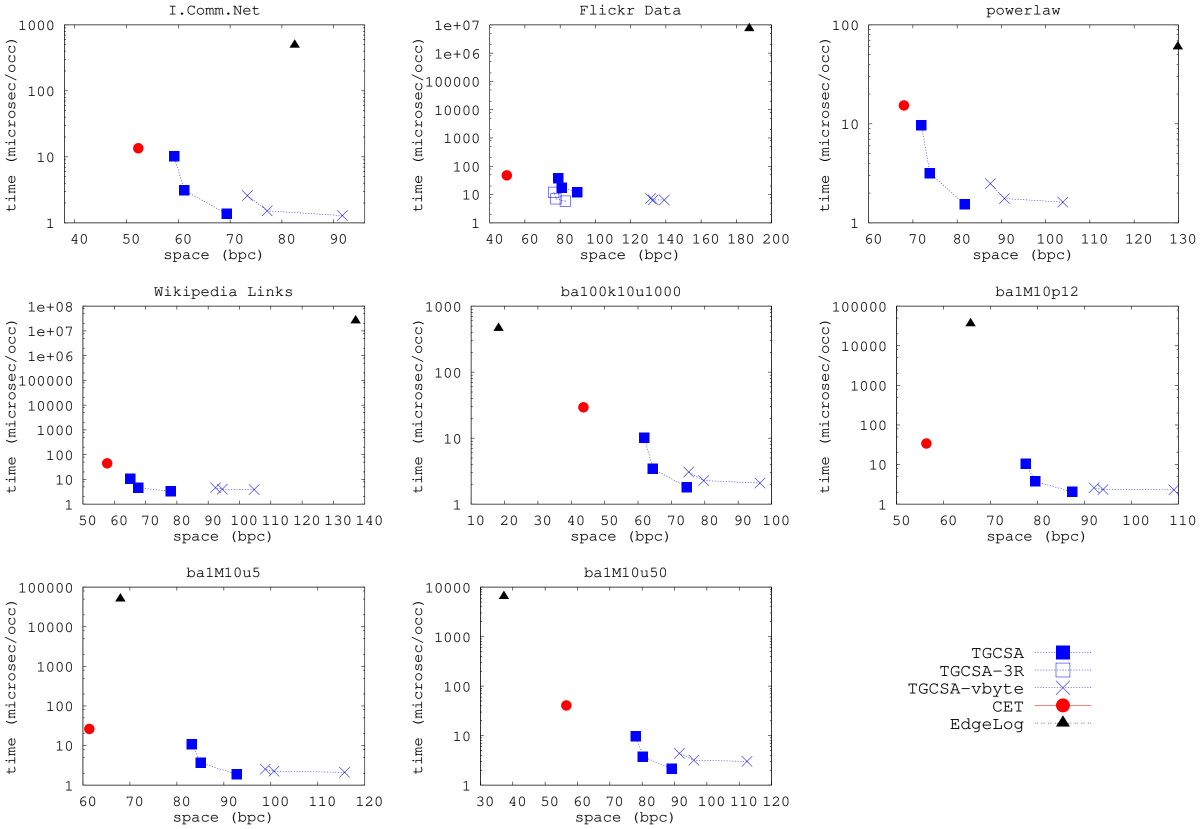


Figure 18: Space/time trade-off for activatedEdge operations.

Note that the time per contact reported of TGCSA for these operations is much better than for the directNeighbor and reverseNeighbor operations because now the traversal of the starting range and the application of Ψ always recover one contact. For the directNeighbor and reverseNeighbor operations, however, many checks (that implied applying Ψ to reach a starting or ending time instant) could discard a candidate contact and, consequently, TGCSA was doing unsuccessful work that increases the reported time per occurrence.

As expected, TGCSA reports the best time performance for activatedEdge and deactivatedEdge operations. With the densest configuration, TGCSA slightly overcomes TGCSA-VB (being 0-40% faster). Yet, when we set $t_\Psi = 256$, TGCSA-VB becomes around $2-4$ times faster than TGCSA.

CET still draws good results, yet it is is clearly overcome by TGCSA. We can also see that EdgeLog is by far the slowest technique. Finally, it is interesting to note that in the `Flickr-Data` graph, TGCSA-3R improves the times of TGCSA by around one third in activatedEdge queries. This is clearly expectable because TGCSA has to apply Ψ three times to recover the source and target vertexes of the edge, whereas TGCSA-3R requires only two Ψ applications.

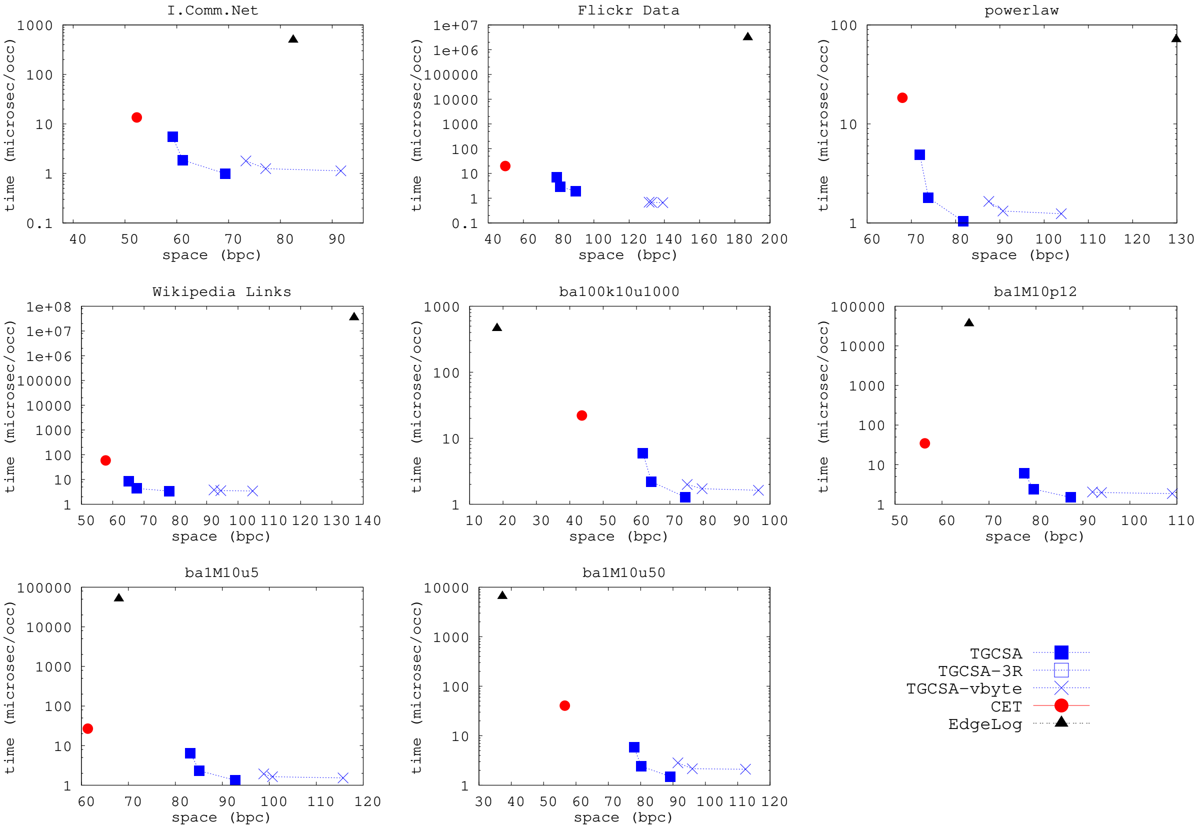


Figure 19: Space/time trade-off for deactivatedEdge operations.

4.4. Time comparison: Snapshot operation

We studied the performance obtained when retrieving the set of all the active edges at a certain time instant (snapshot operation). We compared the average retrieval time at five instants of the lifetime of the temporal graphs: the first and last ones, and those at the 25%, 50%, and 75% of the lifetime in each graph. Table 4 provides the average number of active edges per time instant, that is, the expected output size.

Timeline	0%	25%	50%	75%	100%
I.Comm.Net	19,997	19,991	19,997	19,999	19,996
Flickr-Data	2	17,428	2,313,193	17,586,575	71,345,977
Powerlaw	2,914,527	2,925,980	2,931,495	2,934,810	2,931,023
Wikipedia-Links	1	5,360,597	80,291,698	206,020,758	307,690,159
ba100k10u1000	18,847	470,948	470,824	18,786	470,061
ba1M10p12	90	4,121,832	4,866,245	4,121,871	95
ba1M10u5	90	4,864,776	4,866,275	4,863,160	94
ba1M10u50	988	4,866,241	4,866,821	4,866,351	937

Table 4: Number of contacts reported at each instant of the timeline for snapshot operations per each temporal graph.

Note that EdgeLog computes the snapshot operations with the application of directNeighbor queries over all the vertexes in the graph. CET computes this operation as a rangeReport operation in the underlying Wavelet Matrix [11] and its cost is logarithmic with respect to the total number of edges in the graph. TGCSA, instead, must check which contacts match the time constraints of the query for all the candidate contacts. As shown, this is done with a binary search to find the ranges within the suffix array with possible both valid starting and ending time instants. That is followed by a traversal of the valid starting times (buffered access to Ψ) to check if the end-time constraint is matched. In that case, we recover the source and target vertexes with one and two applications of Ψ, respectively.

Figure 20 shows the results. The time measures are shown in μs per edge reported. Overall, the results show that TGCSA overcomes CET in most cases and, in particular, in the non-synthetic datasets. TGCSA-VB draws also very good performance for *snapshot* operations and, as expected, it excels in ba100k10u1000 dataset due to its small vocabulary (few vertexes and short lifetime). This allows TGCSA-VB to exploit the faster sequential decoding of vbyte-rle when compared with the huff-rle-opt that is used in TGCSA. Note that, in this particular dataset, where CET clearly overcomes TGCSA, now TGCSA-VB is able to reach the same performance as CET.

For these types of queries, EdgeLog has a fast decoding of posting lists based on the use of *PForDelta*, but it must traverse all these lists for each source vertex. This leads to a very fast snapshot performance when the number of retrieved contacts is high, but it becomes very slow when we recover only a few contacts.

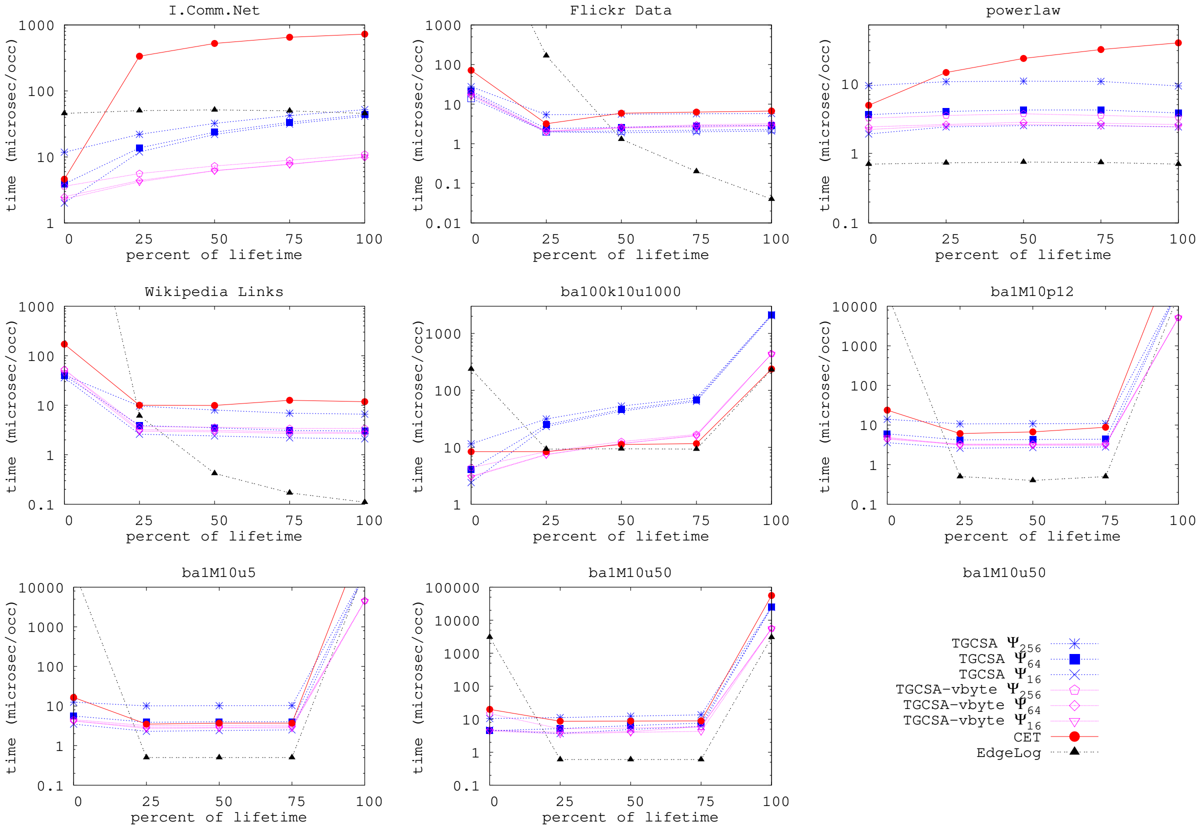


Figure 20: Performance of snapshot operations at different time instants (percent of lifetime).

5. Conclusions and future work

We presented TGCSA, a new representation for temporal graphs based on the well-known CSA. We showed how we can adapt the temporal graph so that it can be indexed with an iCSA self-index. Then, we proposed a modification of the regular Ψ structure in iCSA in such a way that it allows us to move circularly from one term to the other within each contact. This modification solves queries using the CSA mechanism to search for one or more terms of the contacts. This is both fast and flexible.

In addition, we explored a new way to increase the performance of iCSA based on replacing its traditional huff-rle-opt compressed representation of Ψ by a new representation that we called vbyte-rle. To improve access to Ψ values, our new technique uses byte-aligned codewords instead of bit-oriented Huffman (other traditional representations used delta and gamma codes, see [14] for more details). We also avoided sampling Ψ at regular intervals because it is done in traditional compressed representations of Ψ. In our case, since many operations in TGCSA imply recovering a sequence of consecutive values $\Psi[l_c, r_c]$ related to a given symbol c, we sampled the starting positions of Ψ ($\Psi[l_c]$) for all the different symbols c. We ran experiments that verified that our new representation is typically much faster than huff-rle-opt when we want to retrieve a buffer with consecutive values from Ψ. Yet, it is not so advantageous when accessing values at random positions. We created a variant of TGCSA, named TGCSA-VB, that uses the vbyte-rle approach to represent Ψ. TGCSA-VB is up to 5 times faster than TGCSA in some operations; however, it uses around 20-30% more space. Finally, we also adapted TGCSA to the particular case of temporal graphs where contacts have only three terms (an edge is never deactivated). This is the particular case of the Flickr-Data dataset. The resulting variant (referred to as TGCSA-3R) improved the results of TGCSA in both space and time.

The experimental results showed that TGCSA behaves reasonably well in space. In general, space needs are between 50-90 bits per contact. With respect to time performance, TGCSA is very successful for queries that can filter out many contacts from the dataset with an initial binary search in the TGCSA. This avoids the need for sequentially checking a large number of contacts.

We compared TGCSA with CET and EdgeLog. In directNeighbor and reverseNeighbor queries, EdgeLog is a hard rival because it is an inverted index designed to answer directNeighbor queries in a very efficient way and it also uses a reverse aggregated graph to support reverseNeighbor queries efficiently. However, even in this case, TGCSA solves most queries in less than 1 millisecond per contact reported. For queries about events (i.e., activatedEdge or deactivatedEdge), in constrast, EdgeLog performs poorly and TGCSA is clearly the fastest alternative. With respect to CET, we have shown that, even though CET typically uses less space than TGCSA, it is also usually slower. In particular, in activatedEdge and deactivatedEdge queries CET is around one order of magnitude slower than TGCSA.

An important feature of TGCSA is its expressive power. We can use it to represent any set of contacts without any limitation. For example, we could deal with contacts of an edge with overlapping time intervals.

Also, as it was indicated above, the indexing capabilities of the CSA allow us to perform most operations following the same structure: (i) performing an initial binary search in CSA to obtain one range (or more) $[l, r]$ corresponding either to the vertexes or the times in the contacts, and (ii) for all the entries in such range (each one corresponding to a different contact), we can apply Ψ circularly to either recover the other terms of the contacts, or to check a constraint about them.

As future work, we consider that there are two interesting lines we would like to explore in the scope of temporal graphs. On the one hand, our new vbyte-rle allows us to improve the performance of previous Ψ representations [14], but it requires a large amount of extra space. Likewise, the variant vbyte-rle-select uses less space but it also shows to be slower. Since Ψ is the most important structure in TGCSA (it uses around 80-90% of its space, and it is accessed profusely during searches), we still want to try other ways to represent Ψ. On the other hand, we are also interested in studying the applicability of other self indexes to the scope of this paper.

Finally, the variant of CSA shown in this paper is not only of interest in the field of temporal graphs, but it has also opened new opportunities for the application of suffix arrays in other fields. For example, it has obtained very good results when representing *RDF datasets* [4, 9]. In the future we are also planning to study its applicability to represent other types of networks. For example, we have obtained promising results when using a CSA-based approach to represent trajectories of moving objects constrained to a network [5]. We would expect that the flexibility of our approach could make it successful in other contexts.

References

[1] Albert, R., Barabási, A.-L., 2002. Statistical mechanics of complex networks. Rev. Mod. Phys. 74, 47–97.

[2] Bannister, M. J., DuBois, C., Eppstein, D., Smyth, P., 2013. Windows into Relational Events: Data Structures for Contiguous Subsequences of Edges. In: Proc. Symposium on Discrete Algorithms (SODA). pp. 856–864.

[3] Brisaboa, N. R., Caro, D., Fariña, A., Rodríguez, M. A., 2014. A compressed suffix-array strategy for temporal-graph indexing. In: Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE). LNCS 8799. pp. 77–88.

[4] Brisaboa, N. R., Cerdeira, A., Fariña, A., Navarro, G., 2015. A compact RDF store using suffix arrays. In: Proc. 22nd International Symposium on String Processing and Information Retrieval (SPIRE). LNCS 9309. pp. 103–115.

[5] Brisaboa, N. R., Fariña, A., Galaktionov, D., Rodríguez, M. A., 2016. Compact trip representation over networks. In: Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE). LNCS 9954. pp. 240–253.

[6] Buin-Xuan, B.-M., Ferreira, A., Jarry, A., 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. Int. J. Found. Comput. Sci. 14 (02), 267–285.

[7] Caro, D., Rodríguez, A., Brisaboa, N. R., Fariña, A., 2016. Compressed k^d-tree for temporal graphs. Knowl Inf Syst. 49 (2), 553–595.

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

[9] Cerdeira-Pena, A., Fariña, A., Fernández, J., Martínez-Prieto, M., 2016. Self-indexing RDF archives. In: Proc. Data Compression Conference (DCC). pp. 526–535.

[10] Claude, F., Navarro, G., 2009. Practical rank/select queries over arbitrary sequences. In: Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE). LNCS 5280. pp. 176–187.

[11] Claude, F., Navarro, G., Ordóñez, A., 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. Inf. Syst. 47, 15–32.

[12] de Bernardo, G., Brisaboa, N. R., Caro, D., Rodríguez, M. A., 2013. Compact data structures for temporal graphs. In: Proc. Data Compression Conference (DCC). p. 477.

[13] Demetrescu, C., Eppstein, D., Galil, Z., Italiano, G. F., 2010. Algorithms and Theory of Computation Handbook. Chapman & Hall/CRC, Ch. Dynamic Graph Algorithms, pp. 9:1–9:27.

[14] Fariña, A., Brisaboa, N. R., Navarro, G., Claude, F., Places, A. S., Rodríguez, E., 2012. Word-based self-indexes for natural language text. ACM Trans. Inf. Syst. 30 (1), 1:1–1:34.

[15] Ferreira, A., Viennot, L., 2002. A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Tech. rep., MASCOTTE - INRIA Sophia Antipolis / Laboratoire I3S , HIPERCOM - INRIA Rocquencourt.

[16] Grossi, R., Gupta, A., Vitter, J. S., 2003. High-order entropy-compressed text indexes. In: Proc. Symposium on Discrete algorithms (SODA). pp. 841–850.

[17] Grossi, R., Vitter, J., 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symposium on Theory of Computing (STOC). pp. 397–406.

[18] Grossi, R., Vitter, J. S., Xu, B., 2011. Wavelet trees: From theory to practice. In: Proc. International Conference on Data Compression, Communications and Processing (CCP). pp. 210–221.

[19] Holme, P., Saramäki, J., 2012. Temporal networks. Phys. Rep. 519 (3), 97–125.

[20] Huffman, D. A., 1952. A method for the construction of minimum-redundancy codes. Proc. IRE 40 (9), 1098–1101.

[21] Hulovatyy, Y., Chen, H., Milenković, T., 2015. Exploring the structure and function of temporal networks with dynamic graphlets. Bioinformatics 31 (12), i171–i180.

[22] Jacobson, G., 1989. Space-efficient static trees and graphs. In: Proc. Symposium on Foundations of Computer Science (FOCS). pp. 549–554.

[23] Khurana, U., Deshpande, A., 2013. Efficient snapshot retrieval over historical graph data. In: Proc. International Conference on Data Engineering (ICDE). pp. 997–1008.

[24] Kosmatopoulos, A., Giannakopoulou, K., Papadopoulos, A. N., Tsichlas, K., 2016. An overview of methods for handling evolving graph sequences. In: Revised Selected Papers of the 1st International Workshop on Algorithmic Aspects of Cloud Computing (ALGOCLOUD). LNCS 9511. pp. 181–192.

[25] Krogh, B. B., Pelekis, N., Theodoridis, Y., Torp, K., 2014. Path-based queries on trajectory data. In: Proc. 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL). pp. 341–350.

[26] Labouseur, A. G., Olsen, Jr, P. W., Hwang, J.-H., 2013. Scalable and Robust Management of Dynamic Graph Data. In: Proc. International Workshop on Big Dynamic Distributed Data (BD3@VLDB). pp. 43–48.

[27] Liu, Y., Nie, L., Han, L., Zhang, L., Rosenblum, D. S., 2016. Action2activity: Recognizing complex activities from sensor data. CoRR `http://arxiv.org/abs/1611.01872`.

[28] Liu, Y., Nie, L., Liu, L., Rosenblum, D. S., 2016. From action to activity: Sensor-based activity recognition. Neurocomputing 181, 108–115.

[29] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., Czajkowski, G., 2010. Pregel: A system for large-scale graph processing. In: Proc. of the International Conference on Management of Data. (SIGMOD). pp. 135–146.

[30] Mamoulis, N., Cao, H., Kollios, G., Hadjieleftheriou, M., Tao, Y., Cheung, D. W., 2004. Mining, indexing, and querying historical spatiotemporal data. In: Proc. 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD). pp. 236–245.

[31] Manber, U., Myers, G., 1993. Suffix arrays: a new method for on-line string searches. SIAM J. Comput. 22 (5), 935–948.

[32] Michail, O., 2016. An introduction to temporal graphs: An algorithmic perspective. Internet Math. 12 (4), 239–280.

[33] Munro, I., 1996. Tables. In: Proc. Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS 1180. pp. 37–42.

[34] Navarro, G., 2014. Wavelet trees for all. J. Discrete Algorithms 25, 2–20.

[35] Navarro, G., Mäkinen, V., 2007. Compressed full-text indexes. ACM Comput. Surv. 39 (1), article No. 2.

[36] Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., Latora, V., 2013. Temporal Networks. Springer Berlin Heidelberg, Ch. Graph Metrics for Temporal Networks, pp. 15–40.

[37] Raman, R., Raman, V., Satti, S. R., 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3 (4), article No. 43.

[38] Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R., 2011. On Querying Historical Evolving Graph Sequences. In: Proc. Very Large Databases Endowment (VLDB). pp. 726–737.

[39] Sadakane, K., 2003. New text indexing functionalities of the compressed suffix arrays. J. Algorithms 48 (2), 294–313.

[40] Samet, H., 1984. The quadtree and related hierarchical data structures. ACM Comput. Surv. 16 (2), 187–260.

[41] Samet, H., 2006. Foundations of Multidimensional And Metric Data Structures. Morgan Kaufmann.

[42] Semertzidis, K., Pitoura, E., 2016. Durable graph pattern queries on historical graphs. In: Proc. International Conference on Data Engineering (ICDE). pp. 541–552.

[43] Semertzidis, K., Pitoura, E., 2016. Time traveling in graphs using a graph database. In: Proc. International Conference on Extending Database Technology (EDBT).

[44] Shao, B., Wang, H., Li, Y., Jun. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In: Proc. International Conference on Management of Data. (SIGMOD). pp. 505–516.

[45] Shmueli, E., Altshuler, Y., Pentland, A., 2014. Temporal dynamics of scale-free networks. In: Proc. 7th International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction (SBP). LNCS 8393. Springer, pp. 359–366.

[46] Sizemore, A. E., Bassett, D. S., 2017. Dynamic graph metrics: Tutorial, toolbox, and tale. NeuroImage, In press .

[47] Tang, J., Leontiadis, I., Scellato, S., Nicosia, V., Mascolo, C., 2013. Applications of Temporal Graph Metrics to Real-World Networks. Springer Berlin Heidelberg, Berlin, Heidelberg, Ch. 7, pp. 135–159.

[48] Williams, H., Zobel, J., 1999. Compressing integers for fast file access. Comput. J. 42, 193–201.

[49] Witten, I., Moffat, A., Bell, T., 1999. Managing Gigabytes, 2nd Edition. Morgan Kaufmann.

[50] Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y., May 2014. Path problems in temporal graphs. Proc. VLDB Endowment 7 (9), 721–732.

[51] Zhang, J., Long, X., Suel, T., 2008. Performance of compressed inverted list caching in search engines. In: Proc. International Conference on World Wide Web (WWW). pp. 387–396.

[52] Zukowski, M., Heman, S., Nes, N., Boncz, P., 2006. Super-scalar RAM-CPU cache compression. In: Proc. International Conference on Data Engineering (ICDE). p. 59.