A Compressed Suffix-Array Strategy for Temporal-Graph Indexing *

Nieves R. Brisaboa[2], Diego Caro[1], Antonio Fariña[2], and M. Andrea Rodríguez[1]

[1] Dept. Comp. Sci., University of Concepción, Chile. {diegocaro;andrea}@udec.cl
[2] Database Lab., University of A Coruña, Spain. {brisaboa;fari}@udc.es

Abstract. Temporal graphs represent vertexes and binary relations that change over time. In this paper we consider a temporal graph as a set of 4-tuples $\langle v_s, v_e, t_s, t_e \rangle$ indicating that an edge from a vertex v_s to a vertex v_e is active during the time interval $[t_s, t_e)$. Representing those tuples involves the challenge of not only saving space but also of efficient query processing. Queries of interest for this graphs are both direct and reverse neighbors constrained by a time instant or a time interval. We show how to adapt a Compressed Suffix Array (*CSA*) to represent temporal graphs. The proposed structure, called Temporal Graph *CSA* (*TGCSA*), was experimentally compared with a compact data structure based on compressed inverted lists. Our experimental results are promising. *TGCSA* obtains a good space-time trade-off. It owns wider expressive capabilities than other alternatives, it obtains reasonable space usage, and it is efficient even when performing the most complex temporal queries.

1 Introduction

There is an increasing need to handle large graphs that change over time and where not only the current state but also the past state is of interest. For example, consider the evolution of friendship relations when a user adds or removes friends in online social networks, how the citation network grows when new scientific articles are published, how connectivity between mobile devices evolves through time when their base station changes, or how links appear and disappear on the Web graph. The compact representation of temporal graphs is then a relevant problem since direct/reverse-neighboring queries constrained by time instant/interval could benefit from keeping as much data as possible in main memory.

A temporal graph can be seen as a set of 4-tuples of the form $\langle v_s, v_e, t_s, t_e \rangle$, indicating that an edge from a vertex v_s to a vertex v_e is active during the

* Founded in part by Fondef [D09I1185] and Fondecyt [1140428] (for the Chilean group); and, for the Spanish group, by MINECO (PGE and FEDER) [TIN2013-46238-C4-3-R, TIN2013-47090-C3-3-P]; CDTI, AGI, MINECO [CDTI-00064563/ITC-20133062]; ICT COST Action IC1302; and by Xunta de Galicia (co-founded with FEDER) [GRC2013/053].

interval $[t_s, t_e)$. A compact representation for this set of 4-tuples, called *EdgeLog* uses an adjacency list to represent edges and lists of time points marking when each edge is turned on and off. *EdgeLog* can use existing compact representations for inverted lists [16], d-gaps, or k^2-trees [3].

This paper proposes *Temporal Graph CSA* (*TGCSA*), a novel compact data structure based on the Compressed Suffix Array (*CSA*) [14]. We discuss how *TGCSA* opens new opportunities for the application of suffix arrays that are worth exploring both for temporal and general graphs.

Previous work in this area is still incipient. In [7], they represented a temporal graph as several static graphs (or snapshots), storing the active edges for each time point in the lifetime of the graph. Its main drawback is the amount of space used even if edges state (active or not) does not vary for a long time. Storing differences between some snapshots (carefully chosen) saves space but requires processing them at query time [13, 9]. This has the advantage of storing only what changes between consecutive time points and of answering queries about the active direct and reverse neighbors of a vertex. Data structures for temporal graphs based on adjacency lists [4] and on distributed environments [8, 10] exist; however, they have focused on improving time performance neglecting space cost. Recently, we found a preliminary effort to define efficient compact structures for temporal graphs [2]. However, their results apply only to medium size graphs and show that there is much work to do in this area.

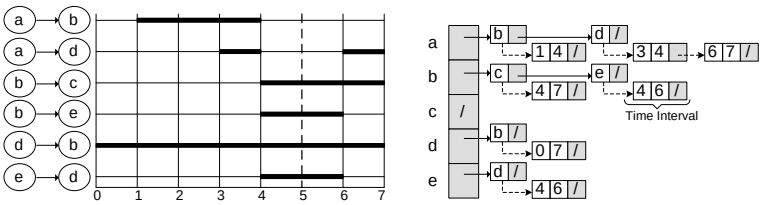
The structure of this paper is as follows. Section 2 presents preliminary concepts and the *EdgeLog* as a baseline to compare with the proposed structure. Section 3 describes *TGCSA*, which is followed in Section 4 by the experimental evaluation using real and synthetic data. Conclusions and future research directions are given in Section 5.

2 Preliminary concepts

Temporal graph definition. Formally, a temporal graph is a set $\mathcal{C}$ of contacts between a set of vertexes V during a set of time points $\mathcal{T}$ representing the *lifetime* of the graph. A *contact* of an edge $(u, v) \in E \subseteq V \times V$ is a 4-tuple $c = (u, v, t, t')$, where $[t, t') \subset \mathcal{T} \times \mathcal{T}$ is the time interval when the edge (u, v) is active [11]. 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 $t \in [t_s, t_e)$. We refer to an *aggregated graph* as the static graph composed by all edges that are active at a time point t within the lifetime of the temporal graph.

For the purpose of this paper, we define four operations on the temporal graph for a given time point t: (1) *Edge existence at* t checks if an edge is active at t. (2) *Direct neighbors at* t returns the adjacent active

neighbors at a given time point t. (3) *Reverse neighbors at t* gives the active inverse adjacent vertexes at time t. (4) *Snapshot at t* returns all active edges at a time point t. For example, given Figure 1a, the snapshot at $t = 5$ corresponds to the edges $\{(b, c), (b, e), (d, b), (e, d)\}$.



(a) Set of contacts (b) *EdgeLog* representation

Fig. 1. A temporal graph of 5 vertexes. The dashed line corresponds to time point $t = 5$. Reverse aggregated graph is omitted in (b). (Figure adapted from [11]).

EdgeLog*: Baseline representation.* A simple temporal graph representation [4] stores the aggregated graph as $|V|$ adjacency lists, one per vertex, with a sorted list of time intervals attached to each neighbor indicating when that edge is active. Figure 1b shows a conceptual example.

To check if an edge (u, v) is active at time t, we first check if v appears within the adjacency list of vertex u. If the edge 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-point 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).

This simple representation has two main drawbacks: (1) it uses much space; and (2) reverse neighbors operation requires traversing all adjacency lists. Both issues are overcome in what we call *EdgeLog*. (1) Since both the adjacency lists and the time-interval lists are sorted (i.e., they are of the form $\langle t_1, t_2, t_3, ..., t_l \rangle$, with $t_i \leq t_{i+1}$, 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 compressed using variable-length encoding for the differences (e.g., PForDelta [17], S16 [15]). Also, (2) to avoid traversing all adjacency lists in reverse-neighbor queries, *EdgeLog* stores the reverse aggregated graph containing an adjacency list with the reverse neighbors of each vertex. Therefore, to get the reverse neighbors of vertex v at time t, we first use the reversed adjacency list to obtain the candidate reverse neighbors of v. Then, for each candidate reverse neighbor u, we move to 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).

Strong and weak points in *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 achieved 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 is designed to be efficient for queries of the type *Edge existence at t* and Direct and Reverse neighbors at t, but it could not answer efficiently queries such as: *"Find all the edges that have active contacts at time t"* or *"Find all the edges that have only been active once"*. Finally, it must be pointed out that the applicability of the *EdgeLog* is limited to temporal graphs where edges can not have overlapping contacts in time.

3 CSA for Temporal graphs (*TGCSA*)

Our *Temporal Graph CSA* (*TGCSA*) is an adaptation of Sadakane's Compressed Suffix Array (*CSA*)[14]. More precisely, it is based on the *integer-based CSA* (*iCSA*) that allows *CSA* to deal with large (integer-based) alphabets (see [6] for details). Recall that *CSA* consists of three main elements to support searches: i) The symbols of the source alphabet S; ii) a bitmap D of size n to mark the positions of the suffix array A where the first symbol of the suffixes pointed to changes; and iii) an array Ψ such that $\Psi[i]$ marks, for each position i in A the position $z = \Psi[i]$ such that $A[z]$ points to the position $A[i] + 1$.

There is an important difference between the standard *CSA* and our implementation that we conceptually describe here. Let us assume that all the terms in a contact are made up from four disjoint alphabets $\Sigma_1, \Sigma_2, \Sigma_3$, and Σ_4 such that $\Sigma_1 \prec \Sigma_2 \prec \Sigma_3 \prec \Sigma_4$ ($\prec$ indicates lexicographic order). Our procedure starts by creating an ordered list of n contacts, so that the contacts are sorted by their first term, then (if they have the same first term) by the second component, and so on. Now, those sorted contacts are regarded as a sequence with $4n$ elements, and a suffix array $A[1, 4n]$ is built over it. Since the values from $\Sigma_i \prec \Sigma_j$ $(\forall i < j)$, the first 25% entries in A ($A[1, n]$) will point to the first terms of all the contacts, the next n entries ($A[n+1, 2n]$) to the second terms, and so on. Consequently, the first 25% entries of Ψ ($\Psi[1, n]$) will point to a position in the range $[n + 1, 2n]$, because in the indexed sequence each symbol $u \in \Sigma_1$ is followed by a symbol $v \in \Sigma_2$, and so on.

Note that, in the standard CSA, 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 in the ordered list $(A[i]+1 = A[\Psi[i]])$, that would be in the range $[1, n]$. However, we modified those pointers in the last 25% of Ψ, because we want that, instead of pointing to the position $x = A[\Psi[i]]$ corresponding to the first term of the following contact, we want them to point to first term of the same contact. That is, $A[\Psi'[i]] = x - 1$, or $A[\Psi'[i]] = n$ if $x = 1$.

By starting at any entry i in Ψ and following the pointers $\Psi[\Psi[\Psi[\Psi[i]]]]$, all the elements of the current contact can be retrieved, but no entry from any other tuple will be reached. With our modification, it is not possible to traverse the whole CSA just using Ψ because consecutive applications of Ψ will cyclically obtain the four elements of the same contact.

3.1 Detailed construction of the *TGCSA*

As indicated above, the first step to build a *TGCSA* is to create a sequence S with the ordered n contacts from $\mathcal{C}$. Hence we obtain, $S[1, 4n] = \langle u^1, v^1, t_s^1, t_e^1, u^2, v^2, t_s^2, t_e^2, \ldots, u^n, v^n, t_s^n, t_e^n \rangle$.[3]

Let us assume we have $\nu = |V|$ different vertexes and $\tau = |\mathcal{T}|$ periods of time. It is possible to define a reversible mapping function that maps the terms of any original contact $c = (u, v, t_s, t_e)$ into $c' = (u, v+\nu, t_s+2\nu, t_e+2\nu+\tau)$. To achieve this, we define an array $gaps[1, 4] \leftarrow [0, \nu, 2\nu, 2\nu+\tau]$ and $c'[i] \leftarrow c[i] + gaps[i]\ \forall i = 1 \ldots 4$. This mapping defines four ranges of entries in an alphabet Σ' for both vertexes and times 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 will permit us to distinguish between starting/ending vertexes/times by simply checking the range where their value falls in.

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

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

within a contact (u, v, t_s, t_e) will correspond to a 1 in B. Now an alphabet Σ of size $\sigma = rank_1(B, 4n)$[4] is created containing the positions in B where a 1 occurs. For each symbol $i \in \Sigma'$ a *mapID(i)* function is defined that assigns it an integer $id \in \Sigma$, 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 is provided via a $unmapID(id) = select_1(B, id)$ function[5].

At this point, a sequence of ids $Sid[1, 4n]$ can be created by setting $Sid[i] \leftarrow mapID(S[i] + gaps[((i-1) \mod 4) + 1]) \forall i = 1 \ldots 4n$.

Indeed, being $type = 1, 2, 3, 4$, respectively, the types of source vertexes, target vertexes, starting times and ending times from the source sequence S, any source symbol i from S can be mapped into Sid as $id = getmap(i, type) \leftarrow rank_1(B, i + gaps[type])$. Similarly, the reverse mapping obtains $i = getunmap(id, type) \leftarrow select_1(B, id) - gaps[type]$.

Finally, an *iCSA* is built over Sid.[6] Note that since the vocabularies of the *ids* associated with the four terms of any contact are disjoint, the corresponding suffix array A will have four ranges of length n so that $A[(j-1)n+1, jn], j = 1..4$. Pointers in each range point to suffixes starting with a source vertex, a target vertex, a starting time, or an ending time, respectively. Similarly, values in $\Psi[1, n]$ in the range of source vertexes, will point to the range of target vertexes $[n+1, 2n]$. Values in $\Psi[n+1, 2n]$ will point to the range of initial times $[2n+1, 3n]$. Those in $\Psi[2n+1, 3n]$ will point to the range of ending times $[3n+1, 4n]$. And finally, those in $\Psi[3n+1, 4n]$ will point to the range of source vertexes $[1, n]$. Indeed, if $A[3n+1]$ points to the ending time of the k^{th} contact of the collection, $z \leftarrow \Psi[3n+1]$ will indicate the position such that $A[z]$ points to the source vertex (first term) of the $(k+1)^{th}$ contact. This is how Ψ works in a regular *iCSA*.

As discussed above, we modified the Ψ array 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 values in the regular Ψ so that, $\forall i = 3n+1 \ldots 4n$, $\Psi[i] \leftarrow ((\Psi[i] - 2) \mod n) + 1$. This small change brings the interesting property of enabling to perform a query for any term of a contact in the same way. We use the *iCSA* to binary search for any term of a contact obtaining a range $A[l, r]$, and then by circularly applying Ψ up to three times, we can retrieve the other terms of the answered-contacts.

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

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

[6] 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.

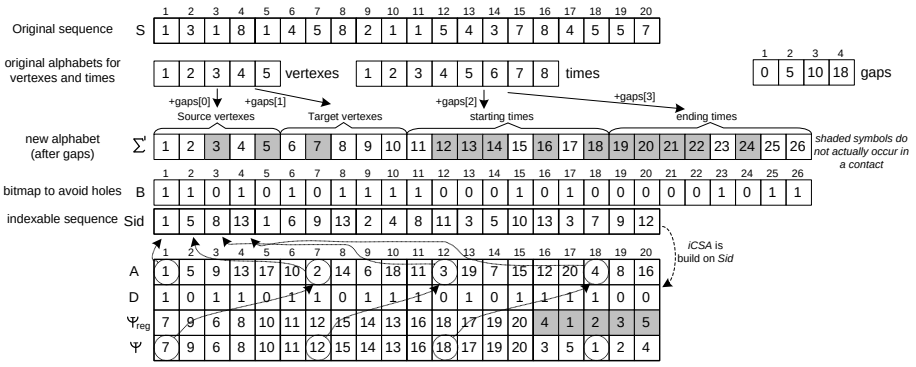


Fig. 2. Structures involved in the creation of a *TGCSA* for the graph in Example 1.

Example 1. Let us assume we have a temporal graph with $|\nu| = 5$ vertexes numbered $1 \dots 5$ and $|\tau| = 8$ time instants numbered $1 \dots 8$. The graph contains the following five contacts: $(1, 3, 1, 8)$, $(1, 4, 5, 8)$,$(2, 1, 1, 5)$, $(4, 3, 7, 8)$, and $(4, 5, 5, 7)$. Figure 2 depicts all the structures involved in the creation of a *TGCSA* that represents the temporal graph. □

To sum up, the *TGCSA* representation consists of bitmap B, and the structures D and Ψ from the *iCSA*. B is compressed with Raman et al. strategy [12]. For D we used a faster bitmap from [6] using $1.375|D|$ bits.

3.2 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 point t. 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 if starting-time and ending-time constraints either hold or not. In Figure 3 we show the pseudocode of the algorithm to obtain direct neighbors.

Edge operations consisting in checking if an edge (u, v) is active at time t are expected to be faster than direct neighbor queries as we can binary search for a phrase $u \cdot v$ rather than by a unique vertex u, hence returning a much shorter initial range. Finally, to solve *snapshot* queries returning the set of active contacts (u, v, t_1, t_2) such that $t_1 \leq t < t_2$, we can binary search $[lt_s, rt_s] \leftarrow$ *CSA_binSearch*$(getmap(t, 3))$ and $[lt_f, rt_f] \leftarrow$ *CSA_binSearch*$(getmap(t, 4))$. All the contacts pointed by $A[2n + 1, rt_i]$ hold $t_s \leq t$, and those in $A[rt_f + 1, 4n]$ hold $t_2 > t$. Therefore, $\forall i \in [2n + 1, rt_s]$, if $\Psi[i] > rt_f$ we recover the source and target vertexes as $\Psi[\Psi[i]]$ and $\Psi[\Psi[\Psi[i]]]$. The original values are obtained via *getunmap()*.

```
DirectNeighbors (vrtx, t) //neighbors of vrtx in contact (vrtx,v,t1,t2) s.t. t1 ≤ t < t2
(1)  u ← getmap(vrtx, typeVertex = 1);    // map into final alphabet without holes
(2)  if u = 0 then return ∅;    // vertex does not appear as a source vertex
(3)  neighbors ← ∅;
(4)  ts ← getmap(t, typeStartTime = 3); te ← getmap(t, typeEndTime = 4);
(5)  [lu, ru] ← CSA_binSearch(u);    // range A[lu, ru] for vertex u
(6)  [lts, rts] ← CSA_binSearch(ts);    // range A[lts, rts] for starting time ts
(7)  [lte, rte] ← CSA_binSearch(te);    // range A[lte, rte] for ending time te
(8)  for i ← lu to ru    // checks time intervals for each occurrence of u
(9)       x ← Ψ[i]; y ← Ψ[x];
(10)      if (y ≤ rts) then
(11)           z ← Ψ[y];
(12)           if (z > rtf) then
(13)                neighbors ← neighbors ∪ {getunmap(x, typeRevVertex = 2)};
(14) return neighbors;
```

Fig. 3. Obtaining the direct neighbors of a vertex in a contact that is active at time t.

3.3 Strengths and weak points in *TGCSA*

One advantage of *TGCSA* with respect to other representations such as those in [2], or our baseline *EdgeLog* is that it actually represents the whole set of 4-tuples. Therefore, it has the same (strong) expressive power as if the set is stored in a database. Note that *TGCSA* can represent temporally overlapping contacts for one edge with no limitations.

Another important property is that *TGCSA* can answer queries over any component of a contact with the same mechanism. That is, searching for the contacts of a source vertex u is done in the same way as searching for the contacts starting at a specific time t. Note that other data structures are designed to answer efficiently some types of queries but they are not efficient at others, whereas *TGCSA* has a more regular behavior.

Note also that inside the section devoted to any given symbol, in any of the four sectors of Ψ, all the pointers are always growing, which is a good property to allow compression. Unfortunately, this becomes a weakness of *TGCSA* when the vocabularies are huge and symbols occur few times. In this case, Ψ will not be highly compressible. As shown in our experiments, compression in some synthetic datasets we created is poor when the relative number of contacts per time instant is low, or when the number of edges per vertex is low. In those cases, the increasing areas of Ψ are small and the gaps between pointers are compressed poorly.

4 Experimental evaluation

We evaluated *TGCSA*[7] on real and synthetic datasets. We compared its space needs with *gzip* compressor and with the baseline *EdgeLog*.[8] We

[7] We used three different settings of *TGCSA* that differ in the sample-rate for Ψ.
[8] *EdgeLog* was configured to use PForDelta with $b = \{32, 128\}$.

also included a comparison in both space and time for some query types. Table 1 describes the experimental datasets. The "base size" is the size of all the uncompressed graphs, representing terms in the contacts with 32-bit integers (128 bits/contact).

The real *Flickr-Days* and *Flickr-Seconds* datasets are well-known temporal graphs where contacts indicate the time-points when two people become friends. Note that each edge has only one contact that ends at the end of the lifetime of the temporal graph. Flickr-Days [5] has a granularity of time in *days*, with a lifetime of 135 days. Flickr-Seconds captures time in *seconds*.

For the synthetic datasets we created an aggregated graph with a uniform degree distribution (following Erdõs-Rényi model [1]), and then assigned a fixed number of contacts to each edge. We used different combinations of the parameters (number of edges per vertex and number of contacts per edge and per instant time) to understand how they affected the compression and behavior of both *TGCSA* and *EdgeLog*. We present a summary of our results.

Dataset	Vertexes (×1000)	Edges (×1000)	Lifetime (×1000)	Contacts (×1000)	avg.contacts/ vertex	edges/ vertex	contacts/ edge	Base size (MB)
Flickr-Seconds	6,204	71,346	167,944	71,346	12	12	1	1,089
Flickr-Days	2,586	33,140	0.135	33,140	13	13	1	506
Erdos1	1,000	10,002	1,000	10,002	10	10	1	153
Erdos5	1,000	10,002	1,000	50,008	50	10	5	763
Erdos50	1,000	10,002	1,000	500,079	500	10	50	7,631
Erdos50R10	1,000	50,001	1	500,079	500	50	10	7,631

Table 1. Temporal graph datasets.

The tests were run on a machine with processors Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz, quad-core, and 64GB DDR3 RAM. The operating system was Ubuntu 12.04 and the compiler gcc 4.6.3 (option -O3).

4.1 Space comparison

In Table 2, we show three configurations of *TGCSA* (Ψ_{16} corresponds to a dense sampling and Ψ_{256} to a sparser one) and compare them with *gzip* as a baseline to show the compressibility of the source dataset. Even tough an *iCSA*-based self-index built on English text typically reached the compression of *gzip* [6], the compressibility of temporal graphs is not so good. Actually, the large number of 1-runs that appeared 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*.

Dataset	gzip default	Edgelog Pfor32	Edgelog Pfor128	TGCSA Ψ_{16}	TGCSA Ψ_{64}	TGCSA Ψ_{256}
FlickrSecs	61.51	161.57	161.53	96.30	87.65	85.48
FlickrDays	30.19	102.06	101.13	60.34	51.06	48.71
Erdos1	83.43	185.91	187.63	105.29	99.07	97.51
Erdos5	63.02	70.32	82.09	94.09	86.52	84.63
Erdos50	53.71	36.76	35.67	89.87	80.93	78.65
Erdos50R10	38.37	24.69	24.10	72.62	62.67	60.19

Table 2. Comparison on space usage. Space in bits per contact.

Focusing in *EdgeLog*, we see that it is completely 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 successful as its inverted lists become highly compressible. *TGCSA* shows a more regular behavior, and 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.

4.2 Performing queries

We chose the real *Flickr-Secs* and the synthetic *Erdos50R10* datasets to show the main features of *TGCSA* when answering typical temporal-graph queries such as retrieving the active direct and reverse neighbors at a given time t, checking if an edge is active at time t,[9] and recovering all the source contacts that are active at a given time instant.

Dataset	Edgelog Pfor32	Edgelog Pfor128	TGCSA Ψ_{16}	TGCSA Ψ_{64}	TGCSA Ψ_{256}	contactsReported
FlickrSecs.DirNei	0.02	0.02	1.48	4.26	9.44	960,364
FlickrSecs.RevNei	9.03	8.50	0.91	1.35	3.35	799,273
FlickrSecs.Edge	5.43	5.17	8.12	13.49	43.20	2,000
FlickrSecs.Snapshot	0.02	0.02	1.22	1.63	3.64	71,345,977
Erdos50R.DirNei	0.61	0.57	49.76	112.23	350.18	10,973
Erdos50R.RevNei	21.61	19.24	23.41	43.47	126.22	9,847
Erdos50R.Edge	4.24	4.12	3.48	5.32	14.48	2,000
Erdos50R.Snapshot	0,45	0.41	3.67	4.53	7.90	5,437,058

Table 3. Comparison on query performance. CPU-user times in μsec/contact reported.

Results in Table 3 show that *TGCSA* is very fast at retrieving direct-neighbors, and the time to recover a contact is close to 1 μsec when using a dense sampling in Ψ (Ψ_{16}) and many contacts are retrieved. Yet, snapshot time per contact reported degrades if many contacts (to check) start before the last time but only a few of them are active at that instant. Note that *EdgeLog* is much faster when answering direct-neighbors as it

[9] We used average times for 2000 queries with random values of t.

is designed for these queries. It is also very fast at the snapshot operation. However, its advantage is reduced drastically on edge operations (where the *TGCSA* is able to binary search directly the interval where the queried edge occurs in A).

As expected, reverse neighbors is the worst case for *EdgeLog*. In particular, its performance degrades when many reverse neighbors need to be checked. Yet, even for this type of queries, in the synthetic collection with less that 50 edges per node, *EdgeLog* was still faster than *TGCSA*.

It is interesting to point out that *TGCSA* is faster when performing reverse neighbor queries than at the direct ones. For reverse queries, we binary search A for a target vertex v (the second term of the contact), and a single application of Ψ permits us to reach the corresponding starting time of that contact (the third term of the contact). With an additional access to Ψ, we can also obtain the ending time. However, when we perform direct-neighbor queries, we start at the first term of the contact, and we need to access Ψ twice and three times to reach the starting and ending time of the contact respectively.

Flexibility to support special queries. *TGCSA* can give support to other query types that could be interesting in some domains. In particular, those queries including exact time-instants or edges, can benefit from searching more than one term in the initial binary search in the *TGCSA*. For example, in a temporal graph representing phone calls from a given user to another, starting and ending at a given time, it could be interesting to perform queries such as: *i) "who phoned user A exactly at time t_s?"*, or *ii) "who received a phone call from B that started at time t_s and ended at t_e?"*. They could be implemented in the *TGCSA* as an initial binary search for $(A \cdot t_s)$ and $(t_s \cdot t_e \cdot B)$, respectively. Then, for the entries in the returned ranges $A[l, r]$, two or one accesses to Ψ, respectively, would be needed to retrieve the caller for the first query, and the receiver in the latter one. Note also that, in the second query, the initial binary search would report a unique entry in A, hence the query is answered almost instantaneously.

5 Conclusions and future work

The experimental results showed that *TGCSA* has reasonable space usage, and succeeds when performing queries that filter out many contacts from the dataset with a single binary search in the *TGCSA*. This avoids the need for sequentially checking a large number of contacts. In particular,

our best trade-off between space and query performance was obtained in the real *Flicker-Days* dataset. In general, space needs are between 50-100 bits per contact, and most queries are solved in less than 1 millisecond per contact reported.

As future work, we want to try more Ψ compression alternatives to those in [6]. Since Ψ represents around 80-90% of the size of *TGCSA*, it is almost the only way to reduce space needs. We are also interested in studying the applicability of other self-indexes to the scope of this paper.

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Modern Physics 74, 47–97 (2002)
2. Bernardo, G.D., Brisaboa, N.R., Caro, D., Rodriguez, M.A.: Compact Data Structures for Temporal Graphs. In: Proc. DCC'13. p. 477 (2013)
3. Brisaboa, N., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. Inf. Systems 39(1), 152–174 (2014)
4. Buin-Xuan, B.M., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. Int. J. Found. Comput. Sci 14(02), 267–285 (2003)
5. Cha, M., Mislove, A., Gummadi, K.P.: A measurement-driven analysis of information propagation in flickr social network. In: Proc. WWW'09. pp. 721–730 (2009)
6. Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A., Rodríguez, E.: Word-based self-indexes for natural language text. ACM TOIS 30(1), article 1 (2012)
7. Ferreira, A., Viennot, L.: A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Tech. rep., MASCOTTE - INRIA Sophia Antipolis / Laboratoire I3S , HIPERCOM - INRIA Rocquencourt (2002)
8. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: Proc. ICDE'13. pp. 997–1008 (2013)
9. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J.H., Han, W.S.: The G* graph database: efficiently managing large distributed dynamic graphs. Distributed and Parallel Databases (2014)
10. Labouseur, A.G., Olsen, Jr, P.W., Hwang, J.H.: Scalable and Robust Management of Dynamic Graph Data. The VLDB Journal pp. 1–6 (2013)
11. Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., Latora, V.: Graph metrics for temporal networks. In: Temporal Networks, pp. 15–40. Springer (2013)
12. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. SODA'12. pp. 233–242 (2002)
13. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. PVLDB 4(11), 726–737 (2011)
14. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)
15. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proc. WWW'08. pp. 387–396 (2008)
16. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2) (Jul 2006)
17. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Super-scalar ram-cpu cache compression. In: Proc. ICDE'06. p. 59 (2006)