On Compressing Time-Evolving Networks

Sudhindra Gopal Krishna[1], Sridhar Radhakrishnan[1], Michael Nelson[1], Amlan Chatterjee[2] & Chandra Shekaran[3]
[1]School of Computer Science, University of Oklahoma, Norman, OK, {sudhi, sridhar, Michael.A.Nelson-1}@ou.edu
[2]Department of Computer Science, California State University Dominguez Hills, Carson, CA, {achatterjee}@csudh.edu
[3]Department of Computer Science, Loyola University at Chicago, Chicago, IL, {chandra}@cs.luc.edu

***Abstract*—A graph G = (V, E) is an ordered tuple where V is a non-empty set of elements called vertices (nodes), and E is a set of an unordered pair of elements called links (edges), and a time-evolving graph is a change in the states of the edges over time. Extremely large graphs are such graphs that do not fit into the main memory. One way to address the issue is to compress the data for storage. The challenge with compressing data is to allow for queries on the compressed data itself at the time of computation without incurring overhead storage costs. Our previous work on Compressed Binary Trees (CBT), which was shown to be efficient both in time and space, compresses each node and its neighbors (termed as row-by-row compression). This paper first provides encoding to store the arrays in the Compressed Sparse Row (CSR) data structure and extends the encoding to store time-evolving graphs in the form of CSR. The encoding also enables accessing a node without decompressing the entire structure, meaning the data structure is queryable. We have performed an extensive evaluation of our structures on synthetic and real networks. Our evaluations include time/space comparison with both time-evolving compressed binary tree and ck^d data structures, including the querying times.**

***Index Terms*—Compression, Network Science, Time-Evolving, Queries, Graphs**

I. Introduction

Graphs can be used to represent real-world data from a wide variety of domains. The relationships among the data are captured by the characteristics of the graph. For most real-world data, the relationships change over time. This results in the graph evolving from its initial state to the current one. A graph $G = (V, E)$ is represented by a set of vertices V and a set of edges E. For real-world data, the graph G evolves with time and can be statically represented using a series of graphs $G_t = (V_t, E_t)$ where the time t indicates an instant which is spread over a certain interval.

Therefore, a time-evolving graph can be defined as a graph that changes or evolves over time. Consider, as an example, pages on Wikipedia. Each page evolves over time with the addition and deletion of content. The current state of the page is the one that contains the content of the page at present. However, all the edit information is also saved for the page. Using the edit information, the state of the page at previous instants of time can be checked. Hence, having such information preserves the integrity of the page while being open to editing. Now, the information for the Wikipedia pages with the time-evolving data can be represented and stored as graphs. Storing such information is useful for performing various kinds of analyses. For example, one might want to know what changes have occurred to a document from beginning to the current state. Another related query can be, what changes occurred to a document within a certain interval of time; this would be specifically interesting to study if the document represents some current socio-economic or political events. Also relevant would be to know the number of changes that occur to the documents from any point in time to another.

Time-evolving graphs represent data from different domains such as social networks and communication networks. A variety of analyses can be performed on such data based on the availability of the same over time. For example, such graphs can be used to perform descriptive, diagnostic, predictive and prescriptive analytics, among others. Hence, to execute such operations on time-evolving graphs, the data has to be stored. Generally, graphs are stored in either of the three different representations: adjacency matrix, adjacency list and edge list. In adjacency matrix, the graph is represented as a matrix of n^2 elements, where $|V| = n$; edges are represented using 1's and lack of it as 0's. For an adjacency list representation, for each node $v \in V$, a list of adjacent nodes is stored. Finally, for an edge list representation, all edges are stored in a pair format (v_i, v_j) where an edge exists from v_i to v_j; the number of entries in the edge list is $|E|$. However, most real-world graphs are very large in size. Hence, the memory requirements for storing the data are significant. For example, if we consider a time-evolving graph, like Wiki-edits and Yahoo Netflow, the sizes of the data in the edge list format are 5.7 GB and 19 GB, respectively. With such sizes, the data might not fit on the main memory for analysis. Therefore, to store the data for time-evolving graphs and perform computation on the same, the data is required to be compressed.

In this paper, we propose techniques to perform compression on time-evolving graphs. There are normally two methods for compressing the adjacency information for a graph: one is to consider the entire graph together, the other considering portions of the graph at a time.

For our methods, we exploit row-by-row compression for node data separately. Specifically, we utilize two combinations of data structures to store the time and adjacency information. In the first one, the time tree provides information regarding the instants of time the graph evolved; for each node, there is an additional tree to store the adjacency information for each instant in the time tree which is CBT. Rather than storing the entire adjacency information for the nodes, a differential approach is leveraged to reduce the memory requirements. In the second approach, for each time frame the edges are stored in an unsigned bit array and similar to the CSR-CSR compression, the later information is stored in a differential approach to save the memory which is CSR. Our contributions also show that depending on the characteristics of the graph being compressed, using a combination of techniques rather than a single method for the data structures yields better compression.

The rest of the paper is organized as follows. In Section II, we discuss existing techniques for storing and managing time-evolving graph data. We propose our methods for compressing and storing time-evolving graphs in Section III. In Section IV, we examine the various algorithms that provide efficient compression for time-evolving graphs. We report the empirical results and analysis in Section V and the conclusion in Section VI.

II. Related Work

A time-evolving graph can be represented as a sequence of static graphs (snapshots), with each of the snapshots representing the graph at a particular point in time. Since a snapshot can be represented as a 2D matrix, a time-evolving graph can thus be represented as a 3D matrix, also known as a *presence matrix* [1].

In 2009, Chierichetti et al. [2] modified the web compression method developed by Boldi and Vigna's WebGraph [3] called Backlinks Compression (BLC). The compression is based on the social network's property of reciprocity. The compression technique makes use of intrinsic ordering heuristics based on shingles, which improves on the WebGraph format.

Nelson et al. [4] in 2017 introduced a compressed data structure as an indexed array of Compressed Binary Tree (CBT). The data structure eliminates the necessity of intermediate structure to create the compressed binary tree. The data structure also makes use of row-by-row compression which enables faster access to the edge existence, neighbor query and the streaming operation.

In 1976, Compressed Sparse Row (CSR) was first documented by Snay [5], and is one of the most common data structures used for representing a graph. Compressed sparse row is also a row-by-row compression which involves two arrays for the compression of each node. All the information is efficiently packed in the array for the quick traversal of the data structure. The first array shows the degree of each node, and the following array shows the edge incidence to each node. Here, the degree of a node v is the number of edges incident to v, and is denoted as $d(v)$.

In 2016, Caro et al. [6] developed $ck^d - trees$. They define a contact as a quadruplet (u, v, t_i, t_j) and then compress the 4D binary matrix corresponding to the time-evolving graph defined by a set of these contacts. It is done by representing the 4D matrix as a $k^d tree$ and then distinguishing white nodes as those without any contacts, black nodes as ones that only contain contacts, and gray nodes as those that contain only one contact. This work was preceded by Brisaboa et al's $k^2 - trees$ [7] in 2014.

G^* database [8] is a distributed index that solves the space issue of the presence matrix by only storing new versions of an arc when its state changes, i.e. as a log of changes. This is done by storing versions of the vertices as adjacency lists and maintaining pointers to each time frame. If an arc changes in the next frame, a new adjacency list is created for that vertex's arc and a pointer is added to the new frame.

Caro et al. [9] proposed a compressed adjacency log structure based on the *log of events* strategy called EveLogs. It consists of two separated lists per vertex - one for the time frames, and another for representing the arcs related to the event. The time frames are compressed using gap encoding, and the arc list is compressed with a statistical model. Caro et al. [6] shows that query times suffer with scanning the log sequentially.

Ren et al. [10], developed the FVF (Find-Verify-Fix) framework which includes a copy+log compression that also supports shortest-paths and closeness centrality queries. More preliminary work is done in [11] [12], which describe three different methods to index time-evolving graphs based on the *copy+log* strategy.

Two *log of events* strategies, CAS and CET, are proposed in [9] to address the problem of slow query times when processing a log. CAS orders the sequence by vertex and adds a Wavelet Tree [13] data structure to allow for logarithmic time queries. CET orders the sequence by time, and the authors develop a modified Wavelet Tree called Interleaved Wavelet Tree to also allow logarithmic time queries.

In 2014, Brisaboa et al. [14] adapted Compressed Suffix Arrays (CSA) as in [9] for use in temporal graphs (TGCSA) by treating the input sequence as the list of contacts. They use an alphabet consisting of the source/destination vertices and the starting/ending times.

III. Representation of time-evolving graphs

In this paper, we represent the time-evolving graphs based on the neighbors of each node over time. This requires two data structures, the first one stores the time information, which can be represented as a time array, and the second one stores the neighbors of the node at each of the instants given in the time array. Now, the time array can be thought of as a stream of 0 and 1 bits: a 0 indicating no change from the previous instant of time, and a 1 indicating changes in the neighborhood of the node from the previous time stamp. Since the time instants taken into account are finite and relatively small, the size of the time array could be in the range of 10,000 for an example graph. For the same graph, the size of the node array, which would contain the neighborhood information for the specific node over 10,000 time instants, could be 1,000,000,000 elements.

The density of the time array and the node array can be different. Given the time array and the node array are bit arrays, these can be compressed for storage using different methods. In this paper, we propose compressing the binary arrays using one of the two techniques: a) CBT, and b) CSR.

Now, depending on the size, and the sparsity of the graphs, the sizes of the compressed structure vary.

The term bit-packing works on the number of bits required to represent each number. For a given array of unsigned integers, represent each number of the array in bits and store them in as an unsigned bit array. For example, consider an array of unsigned integers 1, 3, 5, 10, 16, and 26. The maximum number of bits required to store each integer is the ceiling of the log of the maximum element in the array. An array location can store a maximum of 32 or 64 bits depending upon the system, and all the bits are stored in a little-endian format. Table I shows all the above numbers stores in a single unsigned bit array location.

TABLE I: THE SINGLE BIT ARRAY NEEDED TO REPRESENT ALL THE INTEGERS.

unsigned int	1	3	5	10	16	26
unsigned bit	00001 00011 00101 01010 10000 11010					

If the entire number does not fit into an array location, a part of the number can be stored in one array location and the rest of the number can be stored in the starting bits of the following memory location. This ensures that there are no unoccupied bits in the bit array. Table II shows the bits carry over for a 32-bit integer.

TABLE II: THE CARRY-OVER BIT ARRAY NEEDED TO REPRESENT ALL THE INTEGERS.

unsigned int	1	3	5	10	16	26	30
unsiged bit	00001 00011 00101 01010 10000 11010 11						
	110 00000 00000 00000 00000 00000 0000						

Algorithm 1 explains the working of the bitPacking method. The algorithm takes in an unsigned integer array, the number of elements in the array, and the number of bits required to represent each number in the array. The variable arraySize indicates the number of array locations needed to store the unsigned bits of all the numbers in the array. Line 7 indicates the start of converting the unsigned integer to the bit representation; m indicates the number of unsigned integers that an array location can accommodate. Lines 9 through 11 convert the unsigned integers to unsigned bits and store it in the array location k. The remaining bits in the array location k are filled by the most significant bits of the next number, as shown in line 13 through line 20.

IV. COMBINING CBT AND CSR

For every input of the time-evolving graphs G, the input is divided as an ordered triplet (u, v, T_τ), where u and v are the nodes which form an edge at time T_τ. If the edge appears again later at another time frame $T_{\tau+i}$, the edge is considered to be deactivated at the time frame. For the CSR-CSR and CBT-CSR combinations, we are assuming the datasets are sorted with respect to the time frames and then sorted by node numbers for each time frame. For the CSR-CBT and CBT-CBT combinations, the datasets are first sorted with respect to source node and then sorted with respect to time frame for each source nodes.

Algorithm 1: Algorithm for bitPacking

```
Input: An unsigned integer array (uArray), number of
       elements (numElements), and the number of
       bits (numBits) required to convert
Output: The converted bit array.
1  begin
2      totalBits = 64;
3      balance = 0;
4      arraySize = $\frac{numberofElements}{numberofBits} * totalBits$;
5      Initialize an unsigned bit array (bArray);
6      k = 0;
7      for $index = 0$ to $(index < numOfElements)$ do
8          m = $availBits/numBits$;
9          for $i = 0$ to $(i < m$ && $(index + i)$
             $< numElements)$ do
10             bArray[k] $|= (uArray[index + i] <<$
                 $((i * numBits) + balance)))$;
11             i++;
12         index $+= m$;
13         if $index < numElements$ then
14             remBits = $availBits\%numBits$;
15             if $((remBits > 0)\&\&(m <$
                 $numElements))$ then
16                 bArray[k] $|= (uArray[index] <<$
                     $(totalBits - remBits))$;
17                 availBits
                     $= totalBits - (numBits - remBits)$;
18                 k++;
19                 bArray[k] $|=$
                     $(uArray[index] >> remBits)$;
20                 balance $= (numBits - remBits)$;
21                 index++;
22             else
23                 k++;
24                 balance = 0;
25                 availBits = totalBits;
26     return $bArray$;
```

A. CSR-CSR

This is a novel combination. In this algorithm, we compress the graph based on the edges appearing in each time frame. For the time frame T_0, we compress the graph row-by-row in a conventional compressed sparse row format using Algorithm 2. For each row, the first array consists of each node's degree in the time frame T_0, and the second array consists of the upper triangular destination edge id v.

For time frame T_1 *to* T_τ, storing the edge in the conventional CSR format will cost more space as not all the edges from the original start edges change. To overcome this, we encoded all the edges using Algorithm 3. For every time frame T_i, CSR is made up of three arrays, where the first array is the unique source node u involved in the graph's changes, the

Algorithm 2: Algorithm for Compressed Sparse Row at time T_0

```
Input: Unsigned integer array of contacted nodes (v),
       unsigned integer array of degree of each nodes
       (u), maximum degree of the graph (δ(G))
Output: Unsigned bit array
begin
    logD = log2(maxDegree) + 1;
    logN = log2(numNodes) + 1;
    degreeBitArray = bitPacking(degreeArray,
      numNodes, logD);
    csrBitArray = bitPacking(vArray, numEdges,
      logN);
    finalBitArray.append(degreeBitArray);
    finalBitArray.append(csrBitArray);
    return finalBitArray;
```

Algorithm 3: Algorithm for Compressed Sparse Row from time T_1 to T_τ

```
Input: Unsigned integer array of contacted nodes (v),
       unsigned integer array of degree of each nodes
       (u), unsigned integer array of contact nodes (u)
Output: Unsigned bit array
begin
    for $time\ t = 1\ to\ t = \tau - 1$ do
        logU = log2(maxContactNode) + 1;
        logD = log2(maxDegree) + 1;
        logN = log2(maxContactedNode) + 1;
        bitUArray = bitPacking(uArray, uArray.size(),
          logU);
        bitDegreeArray = bitPacking(degreeArray,
          uArray.size(), logD);
        bitVArray = bitPacking(uArray, vArray.size(),
          logN);
        finalBitArray.append(bitUArray);
        finalBitArray.append(bitDegreeArray);
        finalBitArray.append(bitVArray);
    return finalBitArray;
```

second array consists of the degree of the source nodes, and the third array consists of the destination nodes v.

This yields the time complexity of $O(\tau * (n * log(\delta) + mlog(n)))$, where τ is the number of time frames, n is the number of nodes, m is the number of contacts, and δ is the maximum degree of the graph.

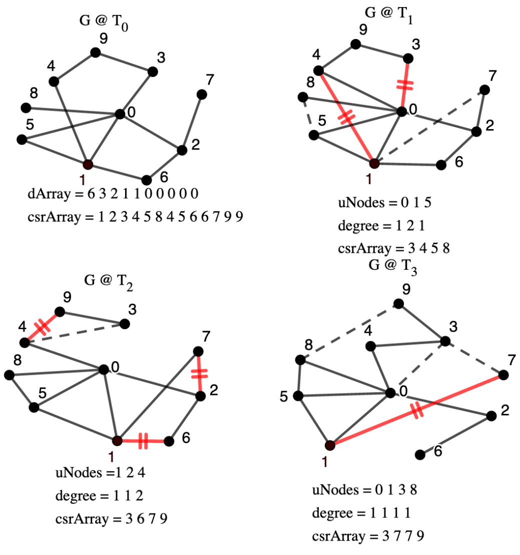
Figure 1 shows the overall structure of the CSR-CSR compression. The dotted line in the graph at each time frame represents the edge being added and the double-crossed red line represents the edge being deleted at the time frame.

B. CBT-CSR

This is a novel combination. In this combination, we compress the first time frame T_0 using the existing CBT algorithm

Algorithm 4: Compressed Binary Tree

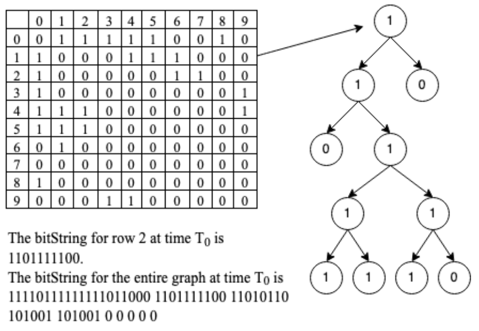
```
Input: An edge's time array, T, of size $\tau$
Output: The compressed edge as a bitstring
begin
    BitString s;
    Node node = root;
    Boolean flip = False;
    Visitor vtr = PreOrderTraversal(node);
    for $i = 0$ to $\tau - 1$ do
        while !node.isLeaf() do
            if (!flip ∧ node.spans(v)) ∨
            (flip ∧ !node.spans(v)) then
                s.AppendBit(1);
            else
                s.AppendBit(0);
                vtr.Ignore(node);
            node = vtr.VisitNext(node);
        s.AppendBit($T[i]$)
        if $T[i]$ != flip then
            flip = !flip;
    //fill rest of tree with 0s
    return s;
```



CSR as unsinged char: 01101100010010001000 100001001100001010100010010101110000100110000101010 000100101 100010100 110000101010001 100010001 100100010 0010111000010101000 0000100011000001 1111 1100111011101001

Fig. 1: The structure of the time-evolving CSR

4 [4] as shown in the Figure 2. The input to the algorithm [4] are the edges associated with time frame T_0. For the time frames $T_1\ to\ T_\tau$ we follow the method used in the CSR-CSR compression, as shown in Figure 1. This yields the time-complexity of $O(\tau * (d(v)log(\delta) + mlog(n)))$.



	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	1	1	0	0	1	0
1	1	0	0	0	1	1	1	0	0	0
2	1	0	0	0	0	0	1	1	0	0
3	1	0	0	0	0	0	0	0	0	1
4	1	1	1	0	0	0	0	0	0	1
5	1	1	1	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0
9	0	0	0	1	1	0	0	0	0	0

The bitString for row 2 at time T_0 is 1101111100.
The bitString for the entire graph at time T_0 is 1111011111111111011000 1101111100 11010110 101001 101001 0 0 0 0 0

Fig. 2: The structure of the time-evolving CBT at time frame T_1

C. CBT-CBT

In this combination, we followed the algorithm mentioned in [15]. The input to this algorithm is first sorted based on the source node u, and for each source node, the data is sorted based on the time T_i. With this type of input comes two arrays for each node, the first being the time frames at which the node has an edge, followed by all the destination node v for each time T_i. Therefore, each node's total number of trees will be the number of time frames for the node u and one tree to represent all the time frames. This yields the time-complexity of $O(\tau * (d(\tau)log(\delta) + mlog(n)))$, where $d(\tau)$ represents the degree of each time-frame.

D. CSR-CBT

This is a novel combination. In this combination, we follow the same input type as CBT-CBT combination, but here we first compress the time array using bit-packing algorithm 1, and to compress the destination edges for each time frame of the source node u, we follow the CBT algorithm 4 [4]. This yields the time-complexity of $O(\tau * (d(v)log(\delta) + mlog(n)))$, where $d(v)$ denotes the degree of each node v.

V. Experimental Results

For our analyses, we have the results of compression size and the time taken to compress from all the combinations using the datasets mentioned in Table III, which is compared with the results of $ck^d - tree$ [6] and CBT [15] as shown in Table IV.

TABLE III: The graph datasets, including the type, number of nodes, number of edges, time frames, and the size of the input file both in .txt and gzip format.

Graphs	Type	Nodes	Edges	Contacts	Time Frames
CommNet	Interval	10000	15940743	19061571	10001
PowerLaw	Interval	1000000	31979927	32280816	1001
Flickr-Days	Incremental	2585570	33140018	33140018	135
Wiki-edits	Point	21504191	561754369	266769613	134075025
Yahoo Netflow	Interval	32904819	122075170	1123508740	58735

If the edges in the graph exist from time $[t_i, t_j)$, then such graphs are called interval graphs. If the edges in the graph appear once and live till the last time-frame, such graphs are referred to as incremental graphs. If the edges appear for a single time-frame, then such graphs are referred to as point graphs.

TABLE IV: The compression size and the time taken to compress each dataset. Please note that ck^d does not allow streaming operations

Graphs	.txt	.txt.gz	CSR-CSR		CSR-CBT	
CommNet	271.6 M	51 M	34 M	10.25 s	16 M	56.19 s
PowerLaw	546.9 M	132 M	80 M	18.94 s	80 M	162.23 s
Flickr-Days	860 M	130 M	107 M	34.16 s	91 M	208.39 s
Wiki-edits	5.7 G	1.8 G	2.0 G	1158 s	1.8 G	2042.88 s
Yahoo Netflow	19 G	4.9 G	4.3 G	1372 s	3.2 G	1770.71 s

Graphs	.txt.gz	CBT-CSR		CBT-CBT		CkD	
CommNet	51 M	16 M	55.80 s	15.9 M	65.5 s	30 M	119 s
PowerLaw	132 M	70 M	141.21 s	73.80 M	149 s	128 M	254 s
Flickr-Days	130 M	82 M	120.015 s	73.8 M	179 s	89 M	235 s
Wiki-edits	1.8 G	2.0 G	1126.85 s	1.4 G	3081s	1.2 G	2059 s
Yahoo Netflow	4.9 G	4.2 G	1874.95 s	2.99 G	3506 s	2.5 G	5471 s

The CommNet graph and the PowerLaw graphs are synthetically generated datasets based on the data avaiable from [6]. CommNet graph simulates short communication between random vertices. PowerLaw graph simulates the powerlaw degree distribution in the graph.

Flickr dataset is an incremental graph [16]. This graph represents the user interaction derived from the Flickr social network for a span of days from 11-02-2006 to 05-18-2007.

Wiki-edits is a bipartite point graph [17]. This graph shows when the user edited an article in Wikipedia. The time is stored in seconds since the creation of Wikipedia.

The last dataset for our analysis is a Yahoo-Netflow graph [18]. This graph is an interval graph, where the data are the interaction between the users and the Yahoo server. The time is measured in seconds and the first occurrence of the data was on 04-29-2008.

All the experiments were run on an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (16 Cores) with 64 GB of RAM, and the programs are written in GNU C/C++.

The source code for this work is available to download at [19].

A. Compression Results

Table IV shows the compression results for the dataset in Table III over all the combinations and $ck^d - trees$. We have used compression size, time taken to compress each dataset, and querying times as metrics to evaluate.

Table IV clearly shows the space and time tradeoff between the compression results of CSR-CSR and CBT-CBT. While the CBT-CBT consumes 30% less space compared to CSR-CSR, CSR-CSR consumes around 40% less time for the Yahoo-Netflow graph [18]. The combination of CBT-CSR and CSR-CBT has shown similar or better results with datasets with fewer or no change in both compression size and time is taken to compress.

B. Querying Results

For 1000 randomly chosen vertices,

- Neighbor Query: What are the neighbors that exist at time T_i.
- Neighbor Query: What are all the possible neighbors of a node between time interval of T_i through T_j.

TABLE V: THE AVERAGE TIME NEEDED TO QUERY A NODE TO FETCH ALL THE NEIGHBORS AT A GIVEN TIME T_i AND THE TIME INTERVAL T_i THROUGH T_j

Graphs	CSR_Ti (ms)	CBT_Ti (ms)	CKD_Ti (ms)	CSR_Ti_Tj (ms)	CBT_Ti_Tj (ms)	CKD_Ti_Tj (ms)
CommNet	0.78 ± 0.005	1.33 ± 1.44	48.89 ± 11.56	0.93 ± 0.049	1.43 ± 0.47	64.46 ± 0.43
PowerLaw	2.07 ± 0.006	2.99 ± 0.55	374.23 ± 50.72	2.10 ± 0.012	5.70 ± 1.05	374.64 ± 50.66
Flickr-Days	1.31 ± 0.02	11.29 ± 8.52	35.34 ± 10.39	2.08 ± 0.31	38.49 ± 8.34	45.22 ± 5.78
Wiki-edits	0.40 ± 0.007	1.24 ± 1.911	3.0 ± 3.0	0.403 ± 0.001	1.42 ± 2.12	4.39 ± 0.72
Yahoo Netflow	2.19 ± 0.45	43.13 ± 0.263	231.9 ± 82.1	1.51 ± 0.06	51.21 ± 4.67	254 ± 92.06

TABLE VI: THE AVERAGE TIME NEEDED TO QUERY AN EDGE EXISTS BETWEEN TWO NODES AT A GIVEN TIME T_i AND THE TIME INTERVAL T_i THROUGH T_j

Graphs	CSR_Ti (ms)	CBT_Ti (ms)	CKD_Ti (ms)	CSR_Ti_Tj (ms)	CBT_Ti_Tj (ms)	CKD_Ti_Tj (ms)
CommNet	0.78 ± 0.001	0.39 ± 0.66	49.6 ± 3.4	0.82 ± 0.002	0.39 ± 0.55	49.7 ± 0.24
PowerLaw	2.06 ± 0.011	0.64 ± 0.12	216.0 ± 5.3	2.08 ± 0.06	1.6 ± 0.03	226.13 ± 14.38
Flickr-Days	1.31 ± 0.013	4.23 ± 2.81	35.2 ± 1.2	2.21 ± 0.2	5.44 ± 10.11	37.2 ± 2.3
Wiki-edits	0.39 ± 0.08	1.15 ± 0.18	2.62 ± 1.7	0.39 ± 0.008	1.15 ± 0.19	2.98 ± 0.25
Yahoo Netflow	1.38 ± 0.014	30.32 ± 2.36	211.8 ± 89.0	1.52 ± 0.042	31.2 ± 5.37	212.32 ± 71

- Edge Existence: Does an edge exist at time T_i.
- Edge Existence: Does an edge exist between the time interval of T_i through T_j.

From Tables V and VI, we can infer that the CSR takes the least amount of time to query a random node both for edge existence and to fetch all the neighbors.

VI. CONCLUSION

Valuable insights can be gained from the analysis of time-evolving graphs. However, due to the large size of such graphs, the memory requirements are significant and it is a challenge for computing using the main memory. Therefore, in this paper we propose compression techniques for time-evolving graphs.

Our techniques show that a significant reduction in memory requirements can be achieved by exploiting the topological characteristics of graphs, specifically the adjacency information for each node also known as row-by-row compression.

With the help of the characteristics of the graphs, we were also able to combine two identical or different techniques to compress a graph, as proposed in section IV. We also compare our compression results with state of the art compression CBT-CBT and $ck^d - trees$, as shown in Table IV.

We implement our algorithms on real-world datasets and show significant improvements in the time required to query edges or any node's neighbors at a given time over the existing techniques, as shown in the Tables V and VI, thereby showing a clear space/time tradeoff between the compression size and the querying time.

Our future work will focus on exploiting the parallelism in improving the timings for the compression techniques on a wider domain of graphs. Also, we plan use the faster querying on compressed structure to enable us an opportunity to develop graph algorithms with time constraints.

REFERENCES

[1] A. Ferreira and L. Viennot, "A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks," Research Report RR-4403, INRIA, 2002.

[2] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, (New York, NY, USA), p. 219–228, Association for Computing Machinery, 2009.

[3] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression Techniques," in *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, (New York, NY, USA), pp. 595–602, ACM, 2004.

[4] M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. Sekharan, "Queryable Compression on Streaming Social Networks," in *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE BigData '17, IEEE Computer Society, 2017.

[5] R. A. Snay, "Reducing the profile of sparse symmetric matrices," *Bulletin Géodésique*, vol. 50, no. 4, pp. 341–352, 1976.

[6] D. Caro, M. A. Rodriguez, N. R. Brisaboa, and A. Farina, "Compressed kd-tree for temporal graphs," *Knowl. Inf. Syst.*, vol. 49, pp. 553–595, Nov. 2016.

[7] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

[8] A. G. Labouseur, J. Birnbaum, P. W. Olsen, Jr., S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs," *Distrib. Parallel Databases*, vol. 33, pp. 479–514, Dec. 2015.

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

[10] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *PVLDB*, vol. 4, pp. 726–737, 2011.

[11] S. Álvarez-García, N. R. Brisaboa, G. d. Bernardo, and G. Navarro, "Interleaved K2-Tree: Indexing and Navigating Ternary Relations," in *2014 Data Compression Conference*, pp. 342–351, March 2014.

[12] G. D. Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodríguez, "Compact data structures for temporal graphs," in *2013 Data Compression Conference*, pp. 477–477, March 2013.

[13] R. Grossi, A. Gupta, and J. S. Vitter, "High-order Entropy-compressed Text Indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, (Philadelphia, PA, USA), pp. 841–850, Society for Industrial and Applied Mathematics, 2003.

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

[15] M. Nelson, S. Radhakrishnan, and C. Sekharan, "Queryable Compression on Time-Evolving Social Networks with Streaming," in *Big Data (Big Data), 2018 IEEE International Conference on*, IEEE BigData '18, IEEE Computer Society, 2018.

[16] "http://socialnetworks.mpi-sws.org/data-www2009.html," 03/2020.

[17] "http://konect.uni-koblenz.de/," 03/2020.

[18] "http://webscope.sandbox.yahoo.com/catalog.php?datatype=g," 03/2020.

[19] "https://github.com/sudhigopal/csr_cbt_paper," 02/2021.