IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 23, 2019, accepted February 12, 2019, date of publication February 22, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2899921

StarZIP: Streaming Graph Compression Technique for Data Archiving

BATJARGAL DOLGORSUREN[1], KIFAYAT ULLAH KHAN[2], MOSTOFA KAMAL RASEL[1], AND YOUNG-KOO LEE[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Kyung Hee University, Seoul 130-701, South Korea
[2]Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad 75190, Pakistan

Corresponding author: Young-Koo Lee (yklee@khu.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MEST) under Grant 2018R1A2A2A05023669, and in part by the MSIT(Ministry of Science and ICT), South Korea, under the Grand Information Technology Research Center Support Program supervised by the IITP(Institute for Information and communications Technology Promotion) under Grant IITP-2018-2015-0-00742.

ABSTRACT The size of a streaming graph is possibly unbounded, and it is updated by a continuous sequence of edges over time. Due to numerous types of real-world interactions, the nature of edge arrival in a streaming graph is dynamic and holds different types of temporal subgraphs, such as stars, bipartite forms, cliques, and chains. The most current techniques find such subgraphs in each snapshot of a dynamic graph and use a dictionary or hash-based summary to compress the graph before applying a stitching technique to demonstrate its temporal behavior. However, it remains difficult to discover those subgraph structures from the continuous stream of edges found in large and rapidly changing dynamic graphs. In this paper, we propose a streaming graph compression algorithm, StarZIP, that uses a new encoding scheme. Our motivational factor is real-world graphs that contain an overwhelmingly large number of stars and a few other structures. We have observed that all subgraph structures can be represented as star-shaped subgraphs. Moreover, the star-shaped representation can easily be arranged in the form of an inverted index, which enables the application of different inverted list encoding techniques for compression. Therefore, we shatter a graph into a uniform representation of stars to compress it. The evaluation of StarZIP on real-world datasets shows that our proposed system reduced the size of a highly dense graph to 60 times less than its original size. Moreover, the experimental results indicate that StarZIP compression is 4 times better than the state-of-the-art techniques.

INDEX TERMS Encoding scheme, graph compression, streaming graph, star structure.

I. INTRODUCTION

Different types of networks, such as social media, e-mail, telephone, and web, can be modeled as streaming graphs for knowledge discovery. Because they create new relationships among a large number of entities over time, the graph size increases rapidly. Many studies have produced techniques such as sampling [1], [2], distances calculation [3] sketches computation [4], triangle listing [5], [6], and pattern matching [7], [8] to process and mine streaming graphs. However, the large size of many graphs is a barrier to the smooth execution of data mining algorithms. Similarly, large graphs have issues with data storage and visualization. Moreover, mining streaming data is challenging because of the massive amount of flooding data

Applying a static graph compression algorithm to a streaming graph is not straightforward because the data are implicit, unpredictable, and rapidly changing. Zaharia *et al.* [9], Sun *et al.* [10], Rasel *et al.* [11] used micro-batch processing to compress data by dividing the incoming events into batches, either by arrival time or window size. When a streaming graph is divided into multiple snapshots during batch processing, incremental encoding is well-suited to compressing each snapshot. A batch processing approach recently presented by Shah *et al.* [12] compresses a dynamic graph by finding a dictionary of temporal patterns, such as stars, bipartite forms, cliques, and chains. The authors summarized sub-structures to compress a graph. However, that algorithm,

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Benevides Da Costa.

Timecrunch, discovers candidate sub-structures only from individual static snapshots of a dynamic graph; it does not consider an accumulating streaming graph. Moreover, it labels the corresponding patterns using static identifiers to minimize the local encoding cost. The results of Timecrunch contain a higher frequency of star-shaped subgraphs than of the remaining types. That technique's limitations have motivated us to consider only the star structure for compression, rather than considering different types of structures over time.

In this paper, we propose a streaming graph compression technique, StarZIP, that uses the star structure as the only representative pattern for compression. StarZIP searches for the required set of star-shaped subgraphs, re-orders them to apply an encoding technique for compression, and then handles incremental updates to the underlying graph. The compression ratio of StarZIP differs depending on the structural properties of the underlying graph because of the high edge occurrence and graph density. Therefore, we focus on reducing the representative bits that show each streaming edge because they occur repeatedly, and we arrange (reorder) their node IDs in ascending order. Compressing a graph with respect to the original edges is rare but necessary [13], [14] for various real-life events. In contrast to Timecrunch, we introduce a new encoding scheme for data compression that preserves the edge weights, which we obtain from the number of edge occurrences. We performed extensive experiments on graphs of different types and sizes and found that StarZIP achieved a 62.6 times better compression ratio than the original data size. Furthermore, we evaluated our proposed algorithm against state-of-the-art compression techniques and found that our system achieved a 2.6 to 4 times better compression ratio and 3 to 25.7 times better execution performance.

Motivation: The star-shaped subgraphs existence is an average of 79.6% of all discovered subgraphs in 12 datasets, making them much more common than subgraph structures such as cliques, chains, and bipartite forms in Timecrunch [12] and VoG [15]. Moreover, any real-world graph can be visualized in the form of stars $G = \cup_{i=1}^{|c|} s_i$ because any new streaming edge predominantly connects with high-degree nodes. The high-degree nodes are important for understanding the evolution of a network and identifying the core nodes of star-shaped subgraphs. Furthermore, real-world graphs have the common property of scale-free power-law distribution. Networks evolve continuously by adding new nodes that connect to existing nodes with a probability proportional to their degree [16]. Therefore, we selected star-shaped structures for our proposed compression method.

Our contributions are as follows:

- We propose a novel algorithm, StarZIP, to find the optimal set of star subgraphs to maximize the compression ratio of a streaming graph.
- We introduce two different techniques, All-Core-First (ACF) and Core-First-Leaves (CFL), to label and reorder node IDs in a priority queue to apply our encoding technique.
- We propose a new encoding scheme, 0x1, to compress an inverted integer list.
- We performed experiments using five real-world datasets and achieved better results with our proposed method than with existing techniques.

II. RELATED WORK

We reviewed the existing work in the field of streaming-graph compression and categorized it into four different types: structural compression, encoding adjacency list, finding a suitable order of graph vertices, and edge sampling to estimate subgraphs.

A. STRUCTURAL COMPRESSION

These techniques find and merge repeating graph patterns. Recently Timecrunch [12] proposed a suitable lexicon for dynamic graphs; the author used the minimum description length principle (MDL) to label temporally coherent subgraphs, such as stars, chains, cliques, and bipartite forms, and effectively compress a large dynamic graph. GraphZip [7] uses a novel dictionary-based compression approach in conjunction with MDL to discover maximally compressed patterns in a graph stream. Similarly, VoG [15] uses MDL to label subgraphs in terms of a vocabulary of static graphs consisting of stars, (near) cliques, (near) bipartite cores, and chains.

B. ENCODING ADJACENCY LIST

Boldi and Vigna [23] proposed several compression techniques for web graphs, including compression by gap encoding, interval representation, and reference compression. Recently, Rossi *et al.* [24] proposed a fast parallel framework for graph compression based on the notion of cliques. Also, the authors introduced a new disk-resident and in-memory graph encoding technique, GraphZIP. We observed that those approaches provide useful graph compression techniques for static and dynamic cases but do not consider edge weight information.

C. FINDING A SUITABLE ORDER OF GRAPH VERTICES

Dhulipala *et al.* [25] have worked to improve the compression of graphs and inverted indexes using a reordering algorithm based on graph bisection. SlashBurn [26] is an efficient node-reordering approach to graph compression after recursive graph shattering. These approaches are not suitable for streaming graphs.

D. SAMPLING AND ESTIMATION OF SUBGRAPHS

Graph priority sampling for subgraph counting was studied in [1] and [2] to minimize the estimation variance counts of specified sets of subgraphs. In study [1], Ahmed *et al.* designed algorithms for triangle and wedge counting, and in [2]], they specialized in bipartite subgraphs. However, those studies proposed lossy compression techniques, and their results are unbiased. Therefore, even though they produce a high compression ratio, they do not offer an exact

TABLE 1. Comparison of all related compression techniques based on the properties of the input graph and the features of algorithm.

	Features of Algorithm				Input Graph		
System	Technique	Objective	Running Time	Lossless	Weighted	Directed	Dynamic
Timecrunch [12]	ranked list of subgraphs	to visualize temporal patterns	Linear	✗	✗	✓	✓
GraphZip by Parker [7]	dictionary-based compression to discover maximally compressing patterns	to retrieve the complex and insightful patterns in graph streams	Exponential	✗	✗	✓	✓
VoG [15]	structure list	patterns, visualization	Sub-Linear	✗	✗	✗	✓
WebGraph [23]	web graph compression	gap encoding, interval representation, and reference compression	Linear	✗	✓	✗	✓
GraphZIP by Rossi [24]	clique-based compression	to encode a large graph on disk-resident and in-memory graph	Linear	✗	✗	✗	✓
SlashBurn [26]	greedy hub selection and the hub ordering	to reduce nonzero blocks in a resulting adjacency matrix	Iterated logarithmic	✗	✗	✗	✓
GPS [1] [2]	sampling and estimation of bipartite subgraphs	to minimize the estimation variance of counts of triangle and wedge	Quadratic	✗	✗	✓	✗
StarZIP	star structure based compression	to achieve higher compression ratio (shattering, reordering,encoding)	Logarithmic	✓	✓	✓	✓

solution because no one prefers lossy information in daily life. The usual protection against lossy compression techniques is to exploit statistical redundancies to represent the data without actually losing any information. Most real-world graphs, such as email, internet, and calling graphs, do exhibit statistical redundancy, but we still need to preserve all the information for archiving and later analysis. Therefore, we need a lossless rather than a lossy compression algorithm. In this work, we use an encoding scheme to compress an inverted list that stores streaming graph data by leveraging a uniform star structure, and we also propose an efficient incremental algorithm for shattering graph snapshots and updating the encoded inverted list at any time point. Our proposed technique thus offers lossless compression; we can provide the exact results for queries from the decompressed graph. In contrast, the existing algorithms [1], [2] achieve a higher compression rate but with some data loss. Moreover, we studied another work on graph compression that produced lossy compression of weighted and dynamic graphs [36].That work used the Top-k approximation algorithm, which can lose some information during fraud detection. We provide the computational analysis for each aforementioned existing works in Table 1.

III. PRELIMINARIES

We consider a streaming graph model $G = (V, E, w)$, where $V = \{v_1, \ldots v_i, v_j, \ldots v_n\}$ is a set of vertices, $E = \{e(v_i, v_j, w) | v_i, v_j \in V;\ v_i \neq v_j; 1 \leq i, j \leq n\}$ is a set of edges that arrives in the form of a stream, and w is the edge weight that indicates the strength of association between any two vertices v_i and v_j. Degree d of $v \in V$ is the count of the neighborhood $N(v)$ of node v. We process the graph stream by batching the original stream into distinct sets of edges, where each set forms a streaming graph object called a *graph snapshot*. A graph snapshot $G_t(V_t, E_t)$ is represented by a subsequence of interconnected edges

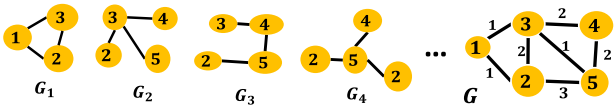


FIGURE 1. Instances of graph snapshots G_1, G_2, G_3, and G_4 of cumulated streaming graph G.

$E_t = \{e(v_i, v_j) | v_i, v_j \in V\}$ and $E_t \subseteq E$ that are ordered by their timestep t, where t is a non-negative integer.

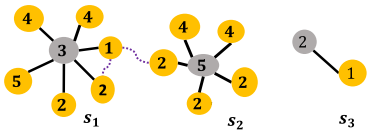
Definition 1 (Streaming Graph): We define a streaming graph as an undirected and weighted graph $G(V, E, w)$ that consists of a chronological sequence of edges $e(v_i, v_j) \in E_t$ corresponding to the timestep t to produce a snapshot G_t.

Figure 1 illustrates a streaming graph. Each graph snapshot G_t, $|t| = 4$ has the same number of edges, $|E_t| = 3$. The cumulated streaming graph G has 5 nodes and 7 edges. The edge between nodes 2 and 5 in G has weight 3, because this edge $e(2, 5)$ occurs once in snapshot 3 (G_3), and twice in snapshot 4 (G_4).

We transform a stream of edges first into graph snapshots and then into a set of stars. Therefore, it becomes possible to update each star incrementally by considering the incoming edges as a link.

Definition 2 (Star Graph): We define a star graph as $s = (c, L)$, where $c \in V$ is a core node, $L \subset V$ are leaf nodes, and $L = \{(e(c, l) | l \in L) \in E\}$. The size of a star is the total number of its member nodes n having $n - 1$ leaves. The degree of the core node c is $d(c) = n - 1$, where each leaf of the core $l \in L$ has the degree $d(l) = 1$.

In a star structure, a core is center node, and the leaves are its neighbor nodes. For any vertex, $v_i \in V$ can be used as a core or leaf to generate a corresponding star. A graph can be represented as a set of multiple star structures S. Figure 2a shows a set of star graphs for the sample streaming graph G in Figure 1.We aim to get an optimal solution by minimizing



(a)

core (c)	Leaf nodes $L(c)$
3	{1,2,2,4,4,5}
5	{2,2,2,4,4}
2	{1}

(b)

FIGURE 2. (a) A set of stars $S = \{s_1, s_2, s_3\}$. Grey nodes are the core nodes of each star and yellow nodes are leaves that connected to a core node. (b) Star representation.

the number of stars $|S|$ and maximizing the number of leaves $n-1$ for each star.

It is possible that the leaves of a star or two different stars are connected to each other by a new incoming edge in the underlying graph G. In that case, the new edge does not affect the existing stars, and we consider it as an individual star with two nodes and call it a *link edge*.

Definition 3 (Link Edge): A link, $lnk(l_i, l_j) \in E$, is an edge that connects the leaf nodes of two stars in the cumulated graph G.

For instance, in Figure 1, we can select any node as a core with the remaining two leaf nodes in G_1. Here, the edge that connects two leaf nodes is a link edge. Therefore, we observe that a link edge creates a cycle in the existing star, and it connects two different stars as well. Similarly, s_3 is a link edge that connects leaves of two different stars, s_1 and s_2, as shown by a dotted line in Figure 2a. The important benefit of the star structure is the ability to store it directly in the form of an inverted list.

Definition 4 (Inverted List): An inverted list $\mathcal{L}$ of a star s stores its core node c and a list of leaf nodes L, where leaf nodes are sorted in ascending order. It is represented as $\mathcal{L}(s) = c + L(c)$.

We illustrate an inverted list in Figure 2b for the stars in Figure 2a. We apply an inverted list compression algorithm to encode the node IDs, and thus the graph is modeled in the form of stars and stored in the inverted list. We compute the compression ratio cr for the input graph G using Equation 1. Compression ratio cr is a non-negative number that represents the ratio between the bits of the original graph size and the compressed graph size.

$$cr(G) = \frac{\text{Size of an original graph (G)}}{\text{Size of the compressed graph (G')}} \tag{1}$$

Definition 5 (TraceNode Table): A hash table $\mathcal{H}\langle Key, Value\rangle$ for tracking the vertex IDs consists of the new and old identifiers of a node.

We use a hash table $\mathcal{H}$, which can add and search a pair $\langle k, v\rangle$, to log vertex IDs re-ordering. We selected the hash table for its high performance in accessing, searching, and re-ordering node IDs because the number of nodes in a streaming graph is huge. For example, the *TraceNode* table of the graph G shown in Figure 3 is $\mathcal{H} = \{\langle 1,3\rangle, \langle 2,5\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 5,4\rangle\}$.

Problem statement. For a given streaming graph $G(V, E, w)$, we aim to find a set of optimal stars to maximize the compression ratio (cr) by minimizing the representative bit size using the function $f(\mathcal{L})$.

IV. THE PROPOSED SOLUTION

A compressed graph is an encoded version that preserves the overall structure of a streaming graph for a particular timestep t. The selection of a locally optimal set of stars guarantees a small number of lists and efficient re-ordering of node IDs, which reduces the number of bits for storage compared with the original size. Let $f(\mathcal{L}) = f(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \ldots, \mathcal{L}_{|C|})$ be a continuously differentiable function defined over a set of stars S. Then, a necessary condition for it to be a locally optimal for a point $\mathcal{L}(s) = \{l_1, l_2, \ldots, l_n\}$ to belong to star $s0$ is to minimize $f(\mathcal{L}(s)) = c + \sum_{i=2}^{d(c)} \delta(i)$, where c is the core node ID. The core node of a star is located in the start of its inverted list $\mathcal{L}(s)$, and $\delta(i) = l_i - l_{i-1}$ is the difference in the leaf node IDs. The final function to select an optimal list of stars is shown in Equation 2.

$$f(\mathcal{L}) = \sum_{i=1}^{|C|} f(\mathcal{L}(s_i)) \tag{2}$$

where, $|C|$ is the number of core nodes and is equal to the number of stars. To solve the problem, our proposed solution avoids computationally complex patterns and focuses on the single representative structure of a star in the graph snapshots. Moreover, we shatter the graphs using stars and re-order their nodes efficiently because we transmit only star information, which enables a small number lists in inverted list representation.

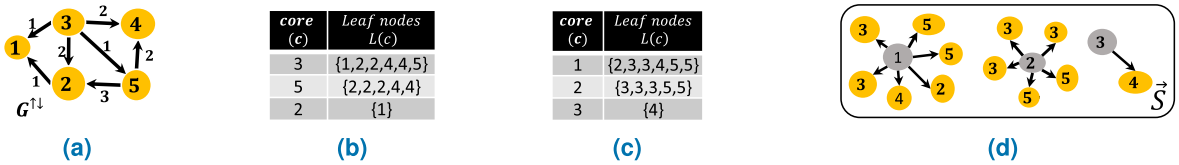
A. STAR-BASED GRAPH SHATTERING

In StarZIP, *shattering* is the process of arranging a graph in the form of stars and re-ordering the node IDs, which encodes the data using our proposed compression schemee (see Section IV-B). Shattering thus produces an inverted list of cores and leaves, as shown Figure 3c. The inverted list is an edge-disjoint decomposition of graph G into a set of stars S, which improves the query execution performance and increases the compression ratio. To shatter a graph, we use an edge direction rule [33], [34] that converts an undirected graph into a directed graph, as follows.

In the given input graph G, the edge direction rule assigns directions to each edge $e(u, v) \in E, f : E \rightarrow \{0, 1\}$, depending on degree of the node $d(v)$, resulting in a directed graph $G^{\uparrow\downarrow}$.

- If $d(v_i) > d(v_j)$, edge $e(v_i, v_j) \in E$ directs from v_i to v_j.
- If $d(v_i) = d(v_j)$, edge $e(v_i, v_j) \in E$ directs from v_j to v_i when v_i has a smaller ID than v_j.

Figure 3a depicts the directed graph produced by the direction rule for the graph in Figure 1. Similarly, we extract a set of stars from the directed graph $G^{\uparrow\downarrow}$, with each star consisting of a core node adjacent to all its out neighbors. We show the shattered stars from $G^{\uparrow\downarrow}$ in Figure 3d. After identifying the direction, each node with at least one outgoing edge serves as a core node. In other words, all the outgoing neighbors



core (c)	Leaf nodes L(c)
3	{1,2,2,4,4,5}
5	{2,2,2,4,4}
2	{1}

core (c)	Leaf nodes L(c)
1	{2,3,3,4,5,5}
2	{3,3,3,5,5}
3	{4}

FIGURE 3. Graph Shattering with Re-Ordering (a) Directed graph $G^{\uparrow\downarrow}$ (b) Star Representation S before re-ordering (c) Inverted List $\mathcal{L}$ as ordering node's old ID to new ID$\{3 \rightarrow 1, 5 \rightarrow 2, 2 \rightarrow 3, 1 \rightarrow 4, 4 \rightarrow 5\}$ (d) A set of extracted stars S after re-ordering.

of each core are selected as leaves, producing lossless graph compression. Therefore, we reverse the original graph G from directed graph $G^{\uparrow\downarrow}$ by deleting the direction of each edge.

We reorder the node IDs of the shattered subgraphs to efficiently apply delta encoding. However, we do not directly re-order the node IDs; instead, we define some priority rules for the core and leaf nodes. The objective of the priority rules is to assign the minimum possible value to each node ID. Thus, a node with higher priority gets a smaller node ID after re-ordering. We use the following priority rules.

1) A core has higher priority than its leaves
2) If a core has a higher out-degree, then it has a higher priority than the other core nodes.
3) If a leaf has more in-degrees, it has higher priority than the remaining leaves.

Our objective is to minimize the function $f(\mathcal{L})$ by prioritizing identifiers with new IDs for each vertex. The compression ratio $cr(G) \geq 0$ is highly effective due to the unique IDs of the vertices. When the node ID increases, the storage cost increases proportionally. Therefore, we select a vertex with a small ID as a core node because a core node ID cannot be encoded. Furthermore, the first leaf node of each star should have a small ID to minimize the differences between the IDs of the remaining leaf nodes.

We store the node IDs with their local information in the hash map, *TraceNode*, to improve locality because core nodes are likely to be accessed frequently. The graph reordering technique does not change the structure of the graph; it only affects the layout of the graph data representations $\mathcal{L}$ and $\mathcal{H}\langle Key, Value\rangle$. The reordering of the node IDs is triggered at the end of each timestep t and performed only in the case of new star creation; otherwise, we extend existing stars.

We propose two different techniques for reordering a core and its leaves: (i) All-Cores-First (ACF), in which we reorder all cores before all leaf nodes, and (ii) Core-First-Leaves (CFL), in which we reorder one core node at a time along with its leaves. For example, in Figure 3c, we use ACF to first assign the IDs 1,2, and 3 to cores 3, 5, and 2, respectively. Then, we assign IDs to the leaf nodes of all the cores, which have higher priority. For instance, in $\{2, 3, 3, \ldots\}$, we replace node 1 with its new ID 4 and replace node 4 with its new ID 5, and the resultant leaf node list for core node 1 is $\{2, 3, 3, 4, 5, 5\}$. If we use CFL instead of ACF, the new ID list will be $\{3 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5\}$.

B. ENCODING TECHNIQUES FOR GRAPH COMPRESSION

We propose the *0x1* scheme for our StarZIP compression technique to compress the graph with a high compression ratio. We transfer the inverted list L for each star into a delta list using the delta encoding technique. Therefore, we can execute an 8-bit packing technique to encode the deltas into binary, which we call the *0x1* scheme.

1) DELTA ENCODING

In general, an integer value (from 0 to 4294967295($2^{32}-1$)) is stored using a word of memory, which is typically 4 bytes. However, we aim to store the lightweight integer IDs dynamically as 8 bits, 16 bits, 32 bits. Therefore, we need to store the differences between the integers instead of the integers themselves to reduce the size of the representative bits.

As a result of shattering and reordering, the most frequent nodes in the list have smaller integer identities to improve compressibility. It is also beneficial to sort the leaves by in-degree because the higher in-degree leaf nodes are re-ordered first. Instead of storing the original array of sorted integers (vertex ID $v_1, v_2, \ldots$, with $v_i \leq v_{i+1}$ for all i), we keep only the differences between successive elements with the initial value ($v_1, \delta_1 = v_2 - v_1, \delta_2 = v_3 - v_2, \ldots$). Therefore, we compress the inverted list more efficiently and then decompress it by computing the prefix sum $v_j = v_1 + \sum_{i=1}^{j} \delta_i$ for data retrieval.

2) 0x1 SCHEME

We design a new compression scheme, named *0x1*, to be applied after the delta encoding. The output of the delta encoding, $\mathcal{D}$, contains of non-negative integers that are much smaller than the original integers, and it contains consecutive series of 1s, frequent 0s, and some other numbers. Therefore, we categorize deltas into three types, 0, 1 and x, as follows.

- *Frequent delta (0-type delta):* Each run of zeros is encoded with the previous x delta and *numberOf* (0) + 1 equal to the edge weight w.
- *Sequential delta (1-type delta):* Each run of ones is represented as sequential IDs of leaf nodes and encoded into a smaller number of bits in the compressed data.
- *Gap delta (x-type delta):* $x \neq 0$ and $x \neq 1$.

We use the 0x1 scheme to map the input delta list into 8 sequential bits, which outputs a smaller size of binary data than the original. This is a type of arithmetic coding that replaces each bit with a codeword to achieve a better compression rate. Therefore, the compression reduces the data size

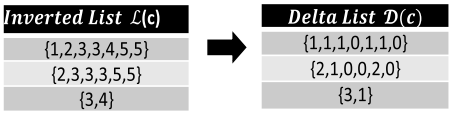


FIGURE 4. Example of delta encoding where the first value in the encoded list is the same as the core nodes in an inverted list $\mathcal{L}(c)$. Thereafter, each number is the difference between leaves.

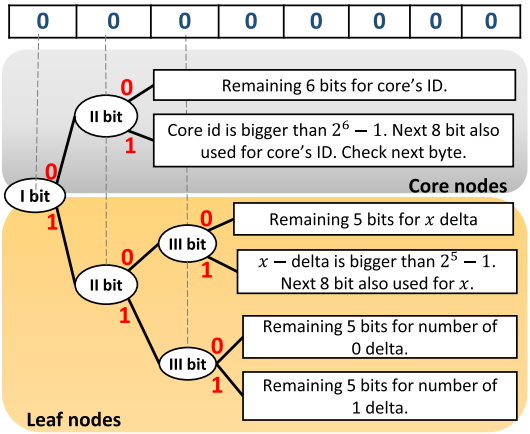


FIGURE 5. Illustration of proposed 0x1 scheme where encoding for 8 bits can be determined by tracing the path from root to leaf node.

using mathematical algorithms about data redundancies and localities, creating a fast and exact encoding scheme. In this case, we use 8 sequential bits to encode an integer or multiple 0s and 1s.

Figure 5 summarizes the logical encoding method of the $0x1$ scheme. To encode the core nodes, we use the first 2 fixed bits as *marker bits* and the last 6 bits as *data bits* to identify the core node. The first bit identifies 0 as a core node and 1 as a leaf node. The second bit of the core's byte is a continuation marker that identifies the continuation of a core node ID: if it is 0, then only the next 6 bits of that byte are used as the core ID, and if it is 1, then the next bytes will also be considered as the core ID. For leaf node encoding, the second bit is used to differentiate the delta types. If the second bit is 0, then the encoding delta is the x-type delta. The third bit is a continuation marker, and the remaining 5 data bits store the delta value. If the second bit is 1, then the next bit represents a 0 or 1-type delta, and the remaining 5 data bits store the frequency of the sequential 0 or 1 delta (the run length of 0s or 1s). For instance, the inverted list for the first star $\mathcal{L}(1) = \{1, 2, 3, 3, 4, 5, 5\}$ converts to the delta list $\mathcal{D}(1) = \{1, 1, 1, 0, 1, 1, 0\}$, which is in memory. In a bits sequence, these numbers are 0000001, 11100010, 11000001, 11100010, and 11000001. As a result, we can store seven numbers using 5 bytes instead of 28 bytes (7×4 bytes). Similar to the encoding of the second star, we compress the delta $\mathcal{D}(2) = \{2, 1, 0, 0, 2, 0\}$ into the following sequence of bits: 00000010, 11100001, 11000010, 10000010, and 11000001. Here, the fourth 8-bit sequence (10000010) is for an $x - type$ delta, which is decoded to 2. We know it easily because the second marker bit of this byte is equal to 0.

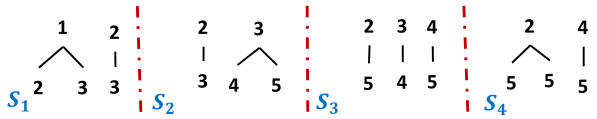


FIGURE 6. For each graph instance in Figure 1, we extract a set of sub-stars and link edges which are used to update the inverted list ($\mathcal{L}$) incrementally.

Algorithm 1 Streaming Graph Compression Using StarZIP

Require: an undirected graph G
Ensure: compressed graph CG
Step 1. Initial Graph Shattering and Encoding
Transfer G_1 into $G_1^{\uparrow\downarrow}$ according to direction identification rules
Extract a set of stars $S(c, L)$ from $G^{\uparrow\downarrow}$
Re-order each nodes and store $S(c, L)$ into inverted list $\mathcal{L}$
Apply Delta-Encoding Algorithm to $\mathcal{L}$
Compress the graph using 0x1 scheme
Step 2. Incremental Update for incoming G_i where $i = 1, \ldots, t$
for each edge $e(v_i, v_j) \in E_i$ **do**
  Check the types of nodes v_i and v_j //Core, Leaf, or New Node
  Identify whether core or leaf node extended
  if $e(v_i, v_j)$ extends core node **then** update an inverted list of $s(v)$ and encode this list only
  else Store the link edge $lnk(e)$ as a new star s and encode $\mathcal{L}(s)$
end for

C. INCREMENTAL ENCODING

For streaming graph compression, we avoid having to begin graph shattering and node re-encoding from the start as graph G changes. Upon the arrival of a new set of stars, StarZIP updates the already encoded corresponding stars, which have same core node IDs. It avoids repeated computation to reduce its computational cost. To evaluate an incoming edge $e(v_i, v_j)$ for encoding, we first need to check whether its nodes already exist, whether they are marked as core or leaf nodes, and whether they belong to the same or different stars. According to the power-law distribution, the new nodes connect to existing nodes with a probability proportional to the degree of the existing node. In other words, the high-degree core nodes, which are located on the top of the inverted list, are extended with new nodes. In this case, we need to find only the core node and insert the new nodes into its inverted list, which does not affect any other stars. In some cases, existing leaf nodes are connected by a new edge or extended by new nodes, and then we add those new edges as link edges $lnk(e)$. Each link edge is stored at the end of the inverted list ($\mathcal{L}$), which encodes only the link edges. Therefore, it has no effect on the existing stars.

Algorithm 1 shows the incremental streaming graph compression used in StarZIP.

The proposed algorithm produces lossless data compression that maintains the integrated original graph. The query results are exactly the same as results obtained from the original graph. Recently, the large amounts streaming data produced by social networks/web graphs have driven research toward lossless compression. Therefore, the scope of lossless graph compression is extensive, containing approximately 450 papers that use various approaches, techniques, algorithms, and applications [35]. We define our lossless graph compression algorithm as follows.

Definition 6 (Lossless Graph Compression): The compressed graph $G'(V', E')$ of original graph $G(V, E)$ is called a lossless compression if it can produce a compact resultant graph with less storage cost and without any loss of nodes or edges and if the graph G can be recovered when it is decompressed.

An exact replica of the original graph can be retrieved by decrypting the compressed graph G' because the proposed *0x1* scheme is reversible, as described in Algorithm 2.

Stream buffering: A stream is a sequence of data in term of bytes. We use a buffer that stores data in memory temporarily. In a streaming case, we dump incoming data into the buffer and then process the data for compression once the buffer is filled. We set the buffer size B to 30KB to store the compressed output stars of the StarZIP algorithm. We compress the delta list of stars obtained from the inverted list $\mathcal{L}$. The first node ID in $\mathcal{L}$ represents the core code of that star. We denote this new star as $s(c)$ and store its information into $l(c)$. Our proposed compression algorithm constructs $\mathcal{L}(s)$ until the occupied space exceeds the given buffer size B. Our StarZIP algorithm starts with an empty buffer that it uses to compress l into an encoded sequential 8 bit stream. When the next G_t item is read from the stream, we append $\mathcal{L}$ of G_t to the buffer. If the buffer B is filled, then we compress that buffer using our encoding scheme and measure the compressed size. The compressed data and metadata are stored onto the disk and cleared from the buffer to make it empty. If the buffer is not full, we continue on with the next G_t. If no more stars remain to be compressed and the buffer is not empty, we compress the remaining data and store it to the disk. In this way, the size of the compressed list is the same in memory and on the disk.

Algorithm 2 Decompression

Require: Sequence of 8 Bits
Ensure: Inverted list $\mathcal{L}$

Step 1. Decode bits into deltas

for each bytes **do**
  Check the first marker bit a_1
  if $a_1 == 0$, it encodes a core node ID **then**
    create a new list $\mathcal{D}(s)$ to store a delta list for a star
    check the second marker bit a_2 whether 0 or 1
    if $a_2 == 0$ **then**
      convert the remaining 6 bits to decimal number
    else
      check next bytes which represents core IDs also.
    end if
  else if $a_1 == 1$, it encodes a leaf node ID **then**
    check the second and third marker bits
    if $a_2 == 0$, the remaining 5 bits for x-type delta **then**
      check the next bits and convert to a number
    else
      check the third marker bit whether 0 or 1
      if $a_3 == 0$, it encodes 0-type delta **then**
        convert the remaining 5 bits to know the number of 0
      else
        convert the remaining 5 bits to know the number of consecutive 1
      end if
    end if
  end if
end for

Step 2. Decode deltas into node IDs

for each $\mathcal{D}(s)$ **do**
  find the node IDs as cumulating deltas
  create a new list $\mathcal{L}(s)$ and add node IDs
end for

V. EXPERIMENTAL EVALUATION

We evaluated our proposed solution against existing systems using real datasets with up to 28 million edges and 1.8 million nodes, as details in Table 2. We implemented our system in *Java* using Apache Spark Streaming [32] library. Apache Spark Streaming is a processing unit for the streaming data that enables scalable batch execution of iterative graph algorithms. The experiments were conducted on a 3.0 GHz Intel processor running Windows 10 with 32GB of RAM. Thus, the compact representation of each graph was loaded into the memory and stored on the disk of a single machine. In the case of the StarZIP algorithm, we set the memory bound to 4 GB and 8 GB for the AS and Wiki datasets, respectively. The experiments were performed using our proposed technique and existing graph compression algorithms to evaluate the space and time complexity of each technique.

A. COMPLEXITY ANALYSIS

1) SPACE COMPLEXITY.

In the worst case, the number of cores $|C|$ in an inverted list $\mathcal{L}$ that includes star information is equal to $|V| - 1$. The total leaf nodes IDs are equal to $|E|$, and each core and leaf node ID is represented by a word (4 bytes). Then space complexity for an inverted list of an undirected graph with large values for $|C|$ and $|E|$ is $O(|E| + |V| - 1)$ words. We summarize the space complexity of all comparable graph representations in Table 3.

We also compute the space complexity after applying the delta and our proposed *0x1* encoding techniques as follows.

TABLE 2. Summary of datasets.

Dataset	Zachary [27]	Email [28]	DBLP [29]	AS-Skitter [30]	Wiki-Tops [31]
Graph Properties					
$\|V\|$	34	986	317,080	1,696,415	1,791,489
$\|E\|$	78	332,334	1,046,866	11,095,298	28,511,807
$Density$	0.13	0.68	2.08E-05	7.71E-06	1.78E-05
Representation structure with Bytes					
$EdgeList$	624	2,658,672	8,374,928	88,762,384	228,094,456
$Adj.List$	760	2,662,616	9,643,248	95,548,044	235,260,412
$Adj.Matrix$	145	121,524	125×10^8	3.59E+11	4.01E+11
$Inv.List$	368	1,331,904	4,852,788	51,166,852	121,213,184

Algorithm 3 Stream Buffering

Require: Streaming Graph G
Ensure: Compressed Graph CG
1: Start with an empty buffer B
2: **for** each G_t from stream graph G **do**
3: Get inverted list $\mathcal{L}$ of G_t
4: Read each stars data one by one $l \in \mathcal{L}$
5: Append l to the compression buffer $B = B \backslash l$
6: Compress the buffer B
7: Measure the compressed size, (number of bytes)
8: **if** $sizeOf(B) \geq$ 30KB **then**
9: complete the block,
10: output metadata and encoded binary bytes on the disk
11: empty the buffer $B = \emptyset$
12: **else**
13: continue on with the next l
14: **end if**
15: **if** no more l in $\mathcal{L}$ to compress, but buffer is not empty $B! = \emptyset$ **then**
16: compress it and output encoded bit stream.
17: **end if**
18: **end for**

TABLE 3. Space complexity of data representations (words).

Representation	Space Complexity
EdgeList	$2 \times \|E\|$
Adj.List	$\|V\| + \|E\|$
Adj.Matrix	$(\|V\| \times \|V\|)/32$
Inv.List $\mathcal{L}$(Worst case)	$\|C\| + \|E\|$

Lemma 1: Let $\mathcal{L}(s)$ be a list of node IDs for a star s in the inverted list $\mathcal{L}$, and let x be the number of words used to represent $\mathcal{L}(s)$. Suppose $D(s)$ be a list of deltas after delta encoding of $\mathcal{L}(s)$. Let y and z be the number of words to represent $D(s)$ before and after applying 0x1 scheme, respectively. Then $x = y$, $x \leq z$ and $y \leq z$.

Proof: Let $\mathcal{L}(s)$ be a list of node IDs in which each node ID is represented by a word. Let $D(s)$ be a list of deltas obtained after using the delta encoding technique on $\mathcal{L}(s)$. The delta encoding reduces the variance of the values, and the created deltas in $D(s)$ are smaller than the node IDs in $\mathcal{L}(s)$. But there is no difference between the number of words required to represent $\mathcal{L}(s)$ and $\mathcal{D}(s)$. Thus, $x = y$ is true.

Let y_x, y_0 and y_1 be the number of representative words in $D(s)$ that are x-type, 0-type, 1-type deltas, respectively, in the delta encoded data. Let y and z be the number of words required to represent $D(s)$ before and after applying *0x1* scheme, respectively. Each of the x-type deltas encodes only a single node ID without edge weight; consecutive 1-type deltas encode multiple sequential node IDs, and a series of 0-type deltas encodes multiple node IDs into a single value (edge weight) for the previous x-type delta node IDs. Therefore, we obtain the following relations among the number of words in $\mathcal{L}(s)$ and $\mathcal{D}(s)$.

1) $y = y_x + y_0 + y_1$
2) $z = z_x + z_0 + z_1$
3) $y_x = z_x$
4) $z_0 \leq y_0$ and $z_1 \leq y_1$

If $D(s)$ contains neither consecutive 0-type deltas nor consecutive 1-type deltas, then $z_0 = 0$ and $z_1 = 0$. Thus, $y_0 = 0$ and $y_1 = 0$. Therefore, according to relations 1, 2, and 3, $z = y$. If $D(s)$ contains at least one 0-type or 1-type delta, then relations 1, 2, 3, and 4 indicate that $z \leq y$. Thus $z \leq y$ is always valid, and $z \leq x$ is also valid because $x = y$.

2) TIME COMPLEXITY.

We calculate the time complexity for four computational parts of StarZIP. First, the computational complexity for graph shattering is $O(|V| + |E|)$. In the implementation, the outer loop execution, $O(|V|)$, depends on the graph structure. If we have no edge at all, the outer loop will be executed for a single iteration with a complexity of $(O(1))$.The inner loop is executed for each edge; thus, that complexity is $O(deg(v))$, where $deg(v)$ is the degree of the current node v. Therefore, the runtime of a single iteration of the outer loop is $O(1 + deg(v))$. After accumulating all complexities, the graph shattering complexity is,

$$O(|V| \times 1 + deg(v1) + deg(v2) + \ldots) = O(|V| + |E|).$$

Second, the time complexity for node-reordering is $O(|V| \times log(|V|))$. We store the reordered new and old node IDs in a *TraceNode* table with $|V|$ elements, and we search it

TABLE 4. Comparison of StarZIP and VoG algorithms.

Graphs	Original (Bytes)	VoG			StarZIP		
		Compressed Size(Bytes)	# of Structures (st, ch, nb, nbc, fc, nc)	Runtime	Compressed Size(Bytes)	# of Stars	Runtime
Zachary	366	236	11	0.017s	214	14	0.001s
Email	4,050,944	1,691,648	3,568	25.05s	678,125	642	0.97s
Dblp	13,733,888	6,000,640	52,642	1h2m59s	6,479,109	166,331	20m58s
AS	149,106,688	27,942,082	149,906	20h53m34s	13,516,160	128,820	15h33m59s
Wiki	422,023,168	212,467,712	493,982	99h33m15s	112,467,712	566,331	21h30m14s

TABLE 5. Detail of VoG algorithms.

Graphs	Frequency of each structure type						# of Iterations
	st	ch	nb	nbc	fc	nc	
Zachary	7	1	0	1	2	0	2
Email	3,434	0	93	15	26	0	344
Dblp	34,314	328	3,094	2,058	7,550	5,298	3,375
AS	121,370	2,649	10,385	5,344	7,650	2,508	11,236
Wiki	470,968	2,030	6,333	2,962	9,558	2,131	47,363

for the reordering verification using a binary search. Therefore, the complexity is $O(log(|V|))$. Third, we iterate for each star in inverted List $\mathcal{L}$. The number of inverted list is $|C|$, and we spend $O(1)$ time to calculate the delta at the end of each list. Therefore, the time complexity for calculating deltas in the inverted list is $O(|C|)$. Finally, the complexity of *0x1* encoding is $O(|C| + 1)$. The *0x1* scheme works on the list that results from the delta encoding. Encoding its integers to binary can be more complicated, but the run-time will not differ too much. We need to eliminate the constants, which do not affect the long-term growth rate of functions, from the summed equation $O(|E| + |V| \times log(|V|) + 2 \times |C| + 1)$. Therefore, the total complexity of our StarZIP algorithm is

$$O(|E| + |V| \times log(|V|) + |C|).$$

B. COMPARISON WITH THE-STATE-OF-THE-ART

We compared our proposed graph shattering process with the graph decomposition step used in the state-of-the-art algorithm VoG [15], and the results are shown in Table 4. The execution time of VoG was up to 25.7 times (average 10 times) longer than that of the proposed algorithm because of the iteration numbers required. In the experiment, we observed the iteration number for finding a set of possibly overlapping subgraphs in the decomposition, as shown in Table 5. VoG finds several structure types and a total number of structures much larger than StarZIP uses. To evaluate all the edges in the AS dataset, VoG required 11,236 iterations and output 149,906 subgraphs of different types. To evaluate all the edges in the Wiki dataset, VoG used 47,363 iterations and output 493,982 sub-patterns within 100 hours. The number of stars found by StarZIP was highly effective for the compression ratio. The compressed size from using StarZIP is much smaller than that from VoG, which uses a model M with different encoded lengths per structure: $L(st)$, $L(ch)$, $L(nb)$, $L(nbc)$, $L(fc)$, and $L(nc)$.

C. COMPARISON WITH INVERTED INDEX-BASED COMPRESSION TECHNIQUES

We selected three well-known inverted list compression algorithms, Variable Byte (VB) [17], Simple9 [18], and PforDelta [19], to compare with the performance of our proposed algorithm. We also compared our proposed shattering method with two naive shattering approaches that shatter a graph into stars based on the identity of nodes (the lowest ID) and the degree of nodes (maximum degree of nodes).

The results in Figure 7-11 show that our proposed scheme provides better compression of the delta lists, up to 4 times higher than the existing methods. The delta computation is generally considered to be a minor operation, a negligible fraction of the total encoding and decoding time. Our proposed algorithm takes advantage of its small number of deltas. The evaluation results also show that a graph's structural property greatly affects the compression ratio and that the compression ratio has a considerable effect on the graph density: $D = \frac{2|E|}{|V|(|V|-1)}$. These results show the memory required to apply the three compression and four shattering methods.

The results shown in Figures 7a, 8a, 9a, 10a, and 11a indicate that our proposed technique achieved a higher compression ratio than the well-known bit compression techniques. Therefore, it needs to encode only a small number of deltas into 8-bit sequences as node identifiers. Figures 7b, 8b, 9b, 10b, and 11b report the compressed size of the graph after applying the different compression methods. We compressed original graphs to reduce memory footprint and then compared the existing compression methods in terms of their compression ratio and memory footprint. Recall that the compression ratio of StarZIP is always higher than other algorithms, and the larger compression ratio implies the smaller compressed size for the original data.

We selected the Zachary dataset as a benchmark because it contains nodes without joining multiple communities at the same time. However, two different communities are

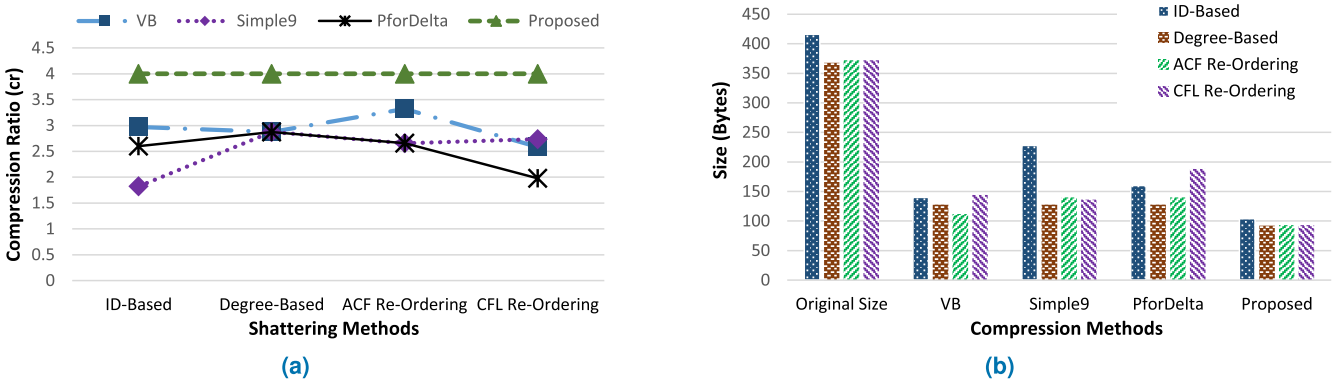


FIGURE 7. Zachary Dataset (a) Compression ratio (cr) (b) Memory footprint analysis achieved by different shattering and compression methods. It was computed based on the size of the original graph represented by Edge List.

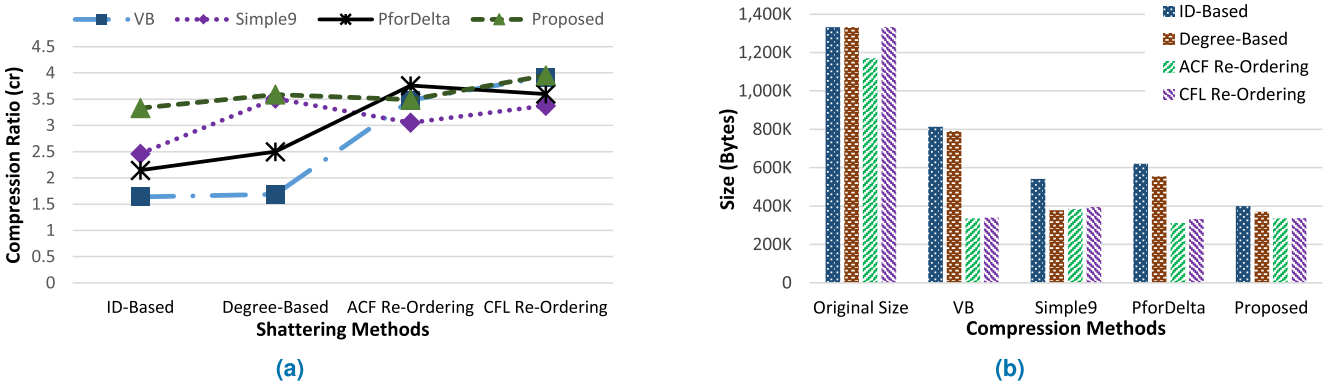


FIGURE 8. Email Dataset (a) Compression ratio using different Shattering Methods (b) Memory footprint analysis using different compression algorithms.

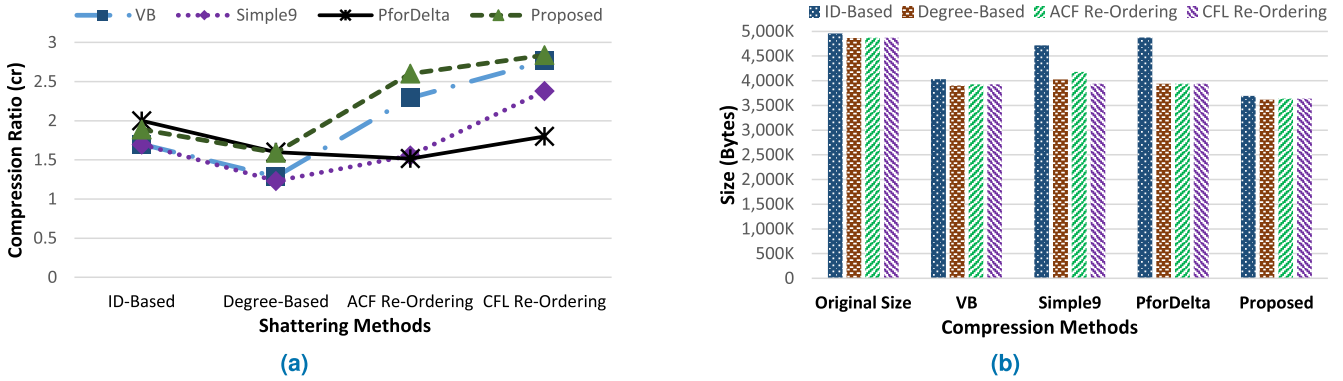


FIGURE 9. BLP Dataset (a) Comparison of compression ratio for different Shattering methods while applying different compression techniques (b) Memory footprint analysis.

connected by edges between the nodes of corresponding communities. This non-overlapping feature of nodes between communities is called a *ground truth community*. The ground truth community structure affects graph compression. We achieved a high compression ratio and easily visualized the graph by representing each community as a star structure. Figure 7 shows that our proposed technique compressed the graph 4 times better than the existing techniques.

In the Email graph shown in Figure 8, we compressed the original graph by 25% due to its degree distribution, which holds a strong power of the law property. For example, more than 10 thousand emails have been transferred to others. This result indicates that we obtained a smaller number of stars and assigned consecutive integer IDs to many nodes.

In Figure 9, the compression ratio of the DBLP graph is much smaller because of the structural behavior of the graph. In this graph, the nodes are authors, and an edge exists between them if they have written a paper together in a given period of time. In real life, a few authors publish together often, which implies that more intra-structures will be present

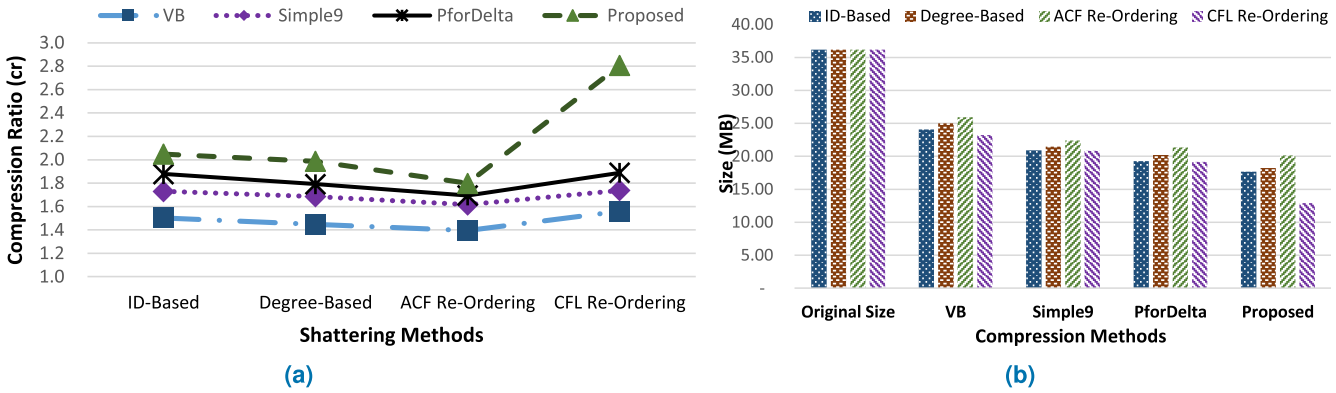


FIGURE 10. AS Skitter Dataset (a) Comparison of compression ratio for different Shattering methods while applying different compression techniques (b) Memory footprint analysis.

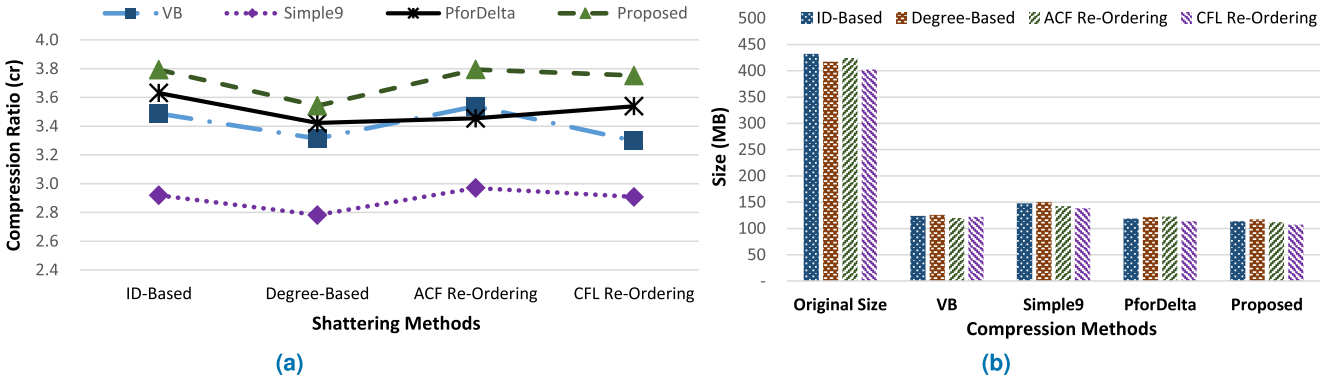


FIGURE 11. Wiki-Tops Dataset (a) Comparison of compression ratio for different Shattering methods while applying different compression techniques (b) The size of memory that reserved entirely for each compression process.

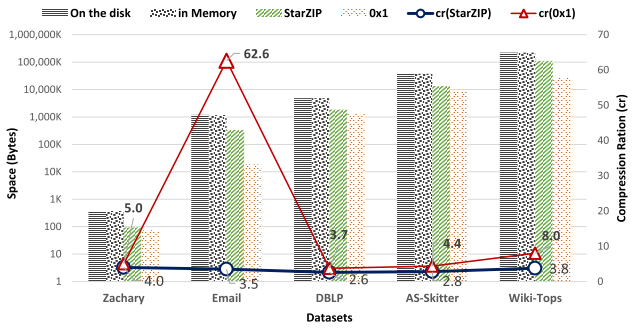


FIGURE 12. Compression ratio and space complexity effectiveness of using 0x1 scheme on different datasets.

in small subparts of the graph and is the main reason that our proposed technique was unable to achieve better compression. Clique-based or triangle mesh compression shows better results for this type of graph.

We also selected two large datasets,which have approximately the same number of nodes and different numbers of edges. The AS Internet topology graph was compressed by 33% compared with the original size, as shown in Figure 10. The compression rate of the Wiki dataset is 1.8 times higher than that of the AS because of its density; it has two times more edges than the AS graph, as shown in Figure 11. We observed that the compression ratio of StarZIP was close to 3 on large graphs with small-world properties, such as DBLP, AS, and Wiki. The StarZIP compression ratio was more than 4 on datasets with a density higher than 0.5 before we applied any encoding scheme.

Effect of 0x1 Scheme: We also evaluated the effectiveness of the proposed *0x1* scheme on the five datasets.

We observed that high edge occurrences are beneficial because they create many 0-type deltas, and sequential 1-type deltas also appeared as a consequence of re-ordering. In Figure 12, we show the size of the graph originally, compressed, and on disk after applying the proposed technique and compression scheme. We measured and show the compression ratio before and after applying the 0x1 scheme. Our proposed approach significantly improved the compression ratio $cr(Proposed)$ by 2.6 to 4 times. After applying the *0x1* scheme, the experimental results show a high variation in compression ratio, from 3.7 to 62.6 times depending on the type of dataset. For example, the *0x1* scheme achieved a compression ratio of more than 60 times for the Email dataset, which is the densest one tested. We calculated the $cr(Proposed)$ and $cr(0x1)$ as follows the equation can be derived, as shown at the bottom of next page:

D. COMPARISON WITH BITMAP COMPRESSION TECHNIQUES

We also compared our proposed scheme against three well-known bitmap compressions schemes: Word Aligned Hybrid

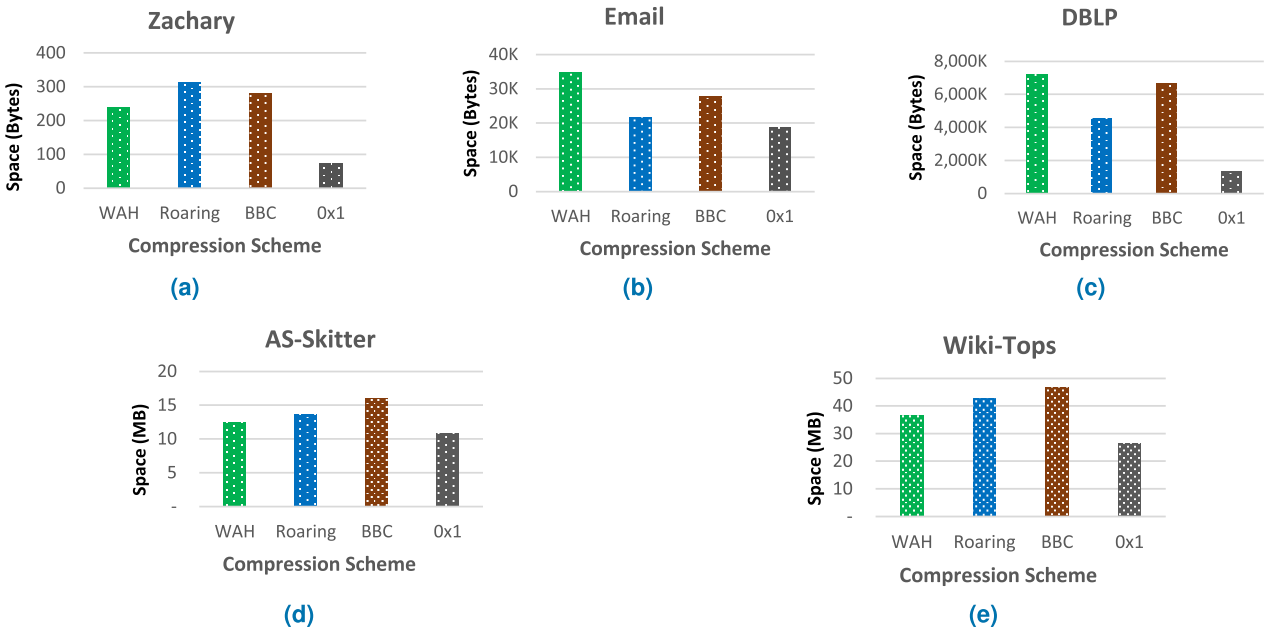


FIGURE 13. Size of memory after applying different bit compression schemes.

(WAH) [20], the Byte-aligned Bitmap Code (BBC) [21] and Roaring [22]. WAH and BBC are lossless run-length compression schemes that can be queried without decompression and fully support intersection (AND) and union (OR) directly from the compressed bitmaps. The representation of WAH is the size of a typical word, and the representation of BBC is the size of a single byte of data. Roaring uses packed arrays for compression instead of run length encoding and divides the entire list into different buckets. Our experimental results indicate that BBC achieved a better compression ratio than WAH on the large datasets such as Email and DBLP. On the other hand, the results from the *0x1* scheme were better than those from all the existing schemes, as shown in Figure 13. The proposed inverted list compression achieved a higher compression ratio than the other methods because it applies the *0x1* scheme to encode the bits sequences. By comparison, the existing bitmap compression schemes consume more space than the original list compression, as shown in Figure 13.

E. EVALUATION OF STARZIP ON A STREAMING GRAPH

Our goal with this set of experiments was to evaluate our algorithm with a streaming graph. Therefore, we divided the number of graph snapshots in each dataset equally. However, the number of edges for the incoming streams differed; therefore, the dataset size both varies and evolves. Figure 14 shows the changing behavior driven by the number of nodes and stars and the size of the compressed graph at each timestep. The size of the compressed graph increased linearly as we divided the graph into more timesteps, even though the number of stars increased slowly. These results also confirm that our technique batched the graph to update the compressed data incrementally. The algorithm updated more than 55% existing stars, which were then added to the previous timesteps.

Furthermore, we measured the execution time for each step (graph shattering, node reordering, delta encoding, and 0x1 encoding) in compressing the large AS and Wiki datasets, as shown in Figure 15. In that experiment, the number of edges in timestep t was set to 1 and 2 million for the AS and Wiki datasets, respectively. The shattering and reordering steps took 99% of the execution time. The encoding algorithm needed 0.1 to 10 seconds to compress the pre-prepared integer list for one million edges.

Comparison With Timecrunch: We compared our algorithm with Timecrunch, which is an effective approach for concisely summarizing large and dynamic graphs while leveraging the MDL. The authors proposed an effective and scalable algorithm to find coherent and temporal patterns in dynamic graphs. We used the original code of Timecrunch,[1] which is available publicly. Timecrunch finds the best model

[1] www.cs.cmu.edu/ neilshah/code/timecrunch.tar

$$cr(Proposed) = \frac{\text{Size of Original Graph}}{\text{Size of Compressed Graph(Proposed technique)}}$$

$$cr(0x1) = \frac{\text{Size of Original Graph}}{\text{Size of Compressed graph(Proposed technique+0x1 scheme)}}$$

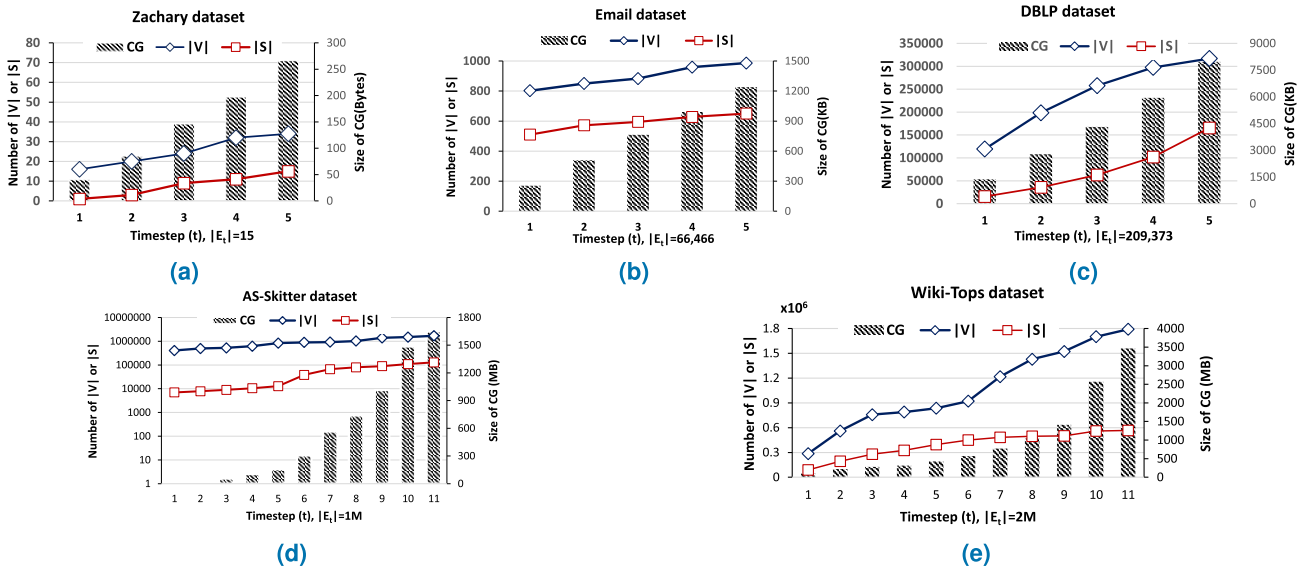


FIGURE 14. Comparing effect of different timesteps on performance of the proposed 0x1 scheme.

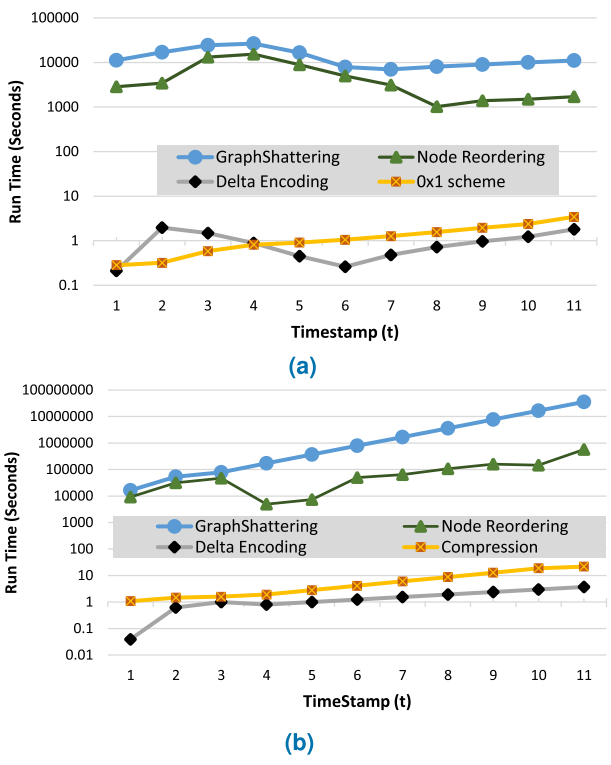


FIGURE 15. Execution time in detail (a) AS-Skitter, in million edges per timestep (b) Wiki dataset, in two million edges per timestep, the y-axis is in log scale.

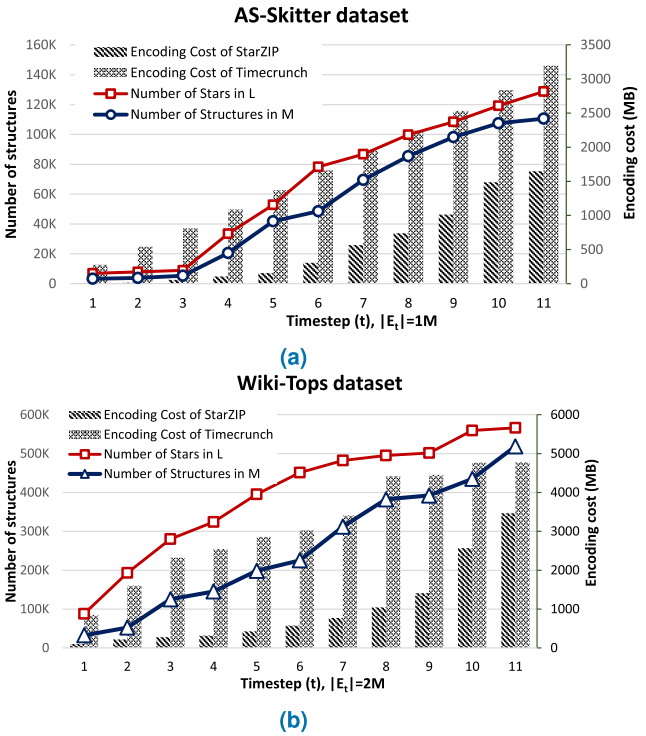


FIGURE 16. Comparison of StarZIP and TimeCrunch algorithms.

M that consists of all possible permutations of the subsets of C, where $C = \bigcup_v C_v$ and C_v denotes the set of all possible temporal structures $\Omega = \{st, fc, nc, bc, nb, ch\}$ of phrase $v \in \Phi$ over all the possible combinations of timesteps. In Figure 16, we represent the size of the compressed graph (CG) on the y-axis (right) using two different bars and the number of stars and structures in the CG on the y-axis (left) using lines. The x-axis of these figures shows the number of edges in different timesteps t, with 1 and 2 million edges per t for the AS and Wiki datasets, respectively. The encoding cost of the StarZIP algorithm is almost 10 times lower than that of Timecrunch for all timesteps. But the number of stars is larger than the number of structures in Timecrunch because the other substructures extract into more than two stars.

The encoding cost increases more slowly over time than for other non-constant polynomials. Therefore, the space complexity function is logarithmic. Logarithmic growing curves increase quickly at the start of execution, and then the

gains decrease and become more horizontal over time. The reason for this phenomenon is the large number of stars in the initial shattered graphs, which is extended by the incoming edges using only the existing stars. The logarithmic scale on the horizontal axis (t) allows for the same proportional increase in a variable to be represented by the same number of incoming edges $|E|$.

VI. CONCLUSION

Our proposed system, StarZIP, is an algorithm that works with a new encoding scheme, 0x1, to compress streaming graphs. Our motivation for StarZIP, which specifically targets the star subgraph structure, stemmed from the results of an existing technique using multiple subgraph types. As a result of this work, we can successfully compress a streaming graph with a better compression ratio than existing solutions can provide. Moreover, our compression scheme is especially beneficial for compressing large evolving graphs in which the edge weight is represented as the occurrence of edges. Previous studies ignored the edge weights during compression; thus, the proposed solution is up to 60 times more effective than existing techniques. Based on our success in this study, our future work will focus on implementing potential optimizations to this algorithm and scheme. We will also extend our study to a scalable distributed stream processing system to achieve runtime benefits.

REFERENCES

[1] N. K. Ahmed, N. Duffield, T. L. Willke, and R. A. Rossi, "On sampling from massive graph streams," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1430–1441, 2017.

[2] N. Ahmed, N. Duffield, and L. Xia, "Sampling for approximate bipartite network projection," in *Proc. IJCAI*, 2018, pp. 3286–3292.

[3] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "Graph distances in the data-stream model," *SIAM J. Comput.*, vol. 38, no. 5, pp. 1709–1727, 2008.

[4] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: Sparsification, spanners, and subgraphs," in *Proc. 31st ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2012, pp. 5–14.

[5] P. Wang, Y. Qi, Y. Sun, X. Zhang, J. Tao, and X. Guan, "Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage," *Proc. VLDB Endowment*, vol. 11, no. 2, pp. 162–175, 2017.

[6] Y. Lim and U. Kang, "MASCOT: Memory-efficient and accurate sampling for counting local triangles in graph streams," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 685–694.

[7] P. Charles, and H. B. Lawrence, "GraphZip: Mining graph streams using dictionary-based compression," in *Proc. SIGKDD Workshop Mining Learn. Graphs*, 2017. [Online]. Available: http://www.mlgworkshop.org/2017/paper/MLG2017_paper_18.pdf

[8] J.-S. Kao and J. Chou, "Distributed incremental pattern matching on streaming graphs," in *Proc. ACM Workshop High Perform. Graph Process.*, 2016, pp. 43–50.

[9] M. Zaharia *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[10] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "GraphScope: Parameter-free mining of large time-evolving graphs," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2007, pp. 687–696.

[11] M. K. Rasel, E. Elena, and Y.-K. Lee, "Summarized bit batch-based triangle listing in massive graphs," *Inf. Sci.*, vol. 441, pp. 1–17, May 2018.

[12] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos, "TimeCrunch: Interpretable dynamic graph summarization," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1055–1064.

[13] K. U. Khan, B. Dolgorsuren, T. N. Anh, W. Nawaz, and Y.-K. Lee, "Faster compression methods for a weighted graph using locality sensitive hashing," *Inf. Sci.*, vol. 421, pp. 237–253, Dec. 2017.

[14] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka, "Compression of weighted graphs," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 965–973.

[15] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, "Summarizing and understanding large graphs," *Stat. Anal. Data Mining*, vol. 8, no. 3, pp. 183–202, 2015.

[16] A. L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[17] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance," in *Proc. 13th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, 1989, pp. 405–411.

[18] V. N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," *Inf. Retr.*, vol. 8, no. 1, pp. 151–166, 2005.

[19] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *Proc. 22nd Int. Conf. Data Eng.*, Apr. 2006, p. 59.

[20] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, 2006.

[21] G. Antoshenkov, "Byte-aligned bitmap compression," in *Proc. DCC Data Compress. Conf.*, Mar. 1995, p. 476.

[22] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Softw.-Pract. Exper.*, vol. 46, no. 5, pp. 709–719, 2016.

[23] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 595–602.

[24] R. A. Rossi and R. Zhou, "GraphZIP: A clique-based sparse graph compression method," *J. Big Data*, vol. 5, no. 1, pp. 1–14, 2018.

[25] L. Dhulipala, I. Kabiljo, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1535–1544.

[26] Y. Lim, U. Kang, and C. Faloutsos, "SlashBurn: Graph compression and mining beyond caveman communities," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 12, pp. 3077–3089, Dec. 2014.

[27] *Zachary Karate Club*. Accessed: Sep. 4, 2018. [Online]. Available: http://konect.uni-koblenz.de/networks/ucidata-zachary

[28] *Email-Eu-Core Temporal Network*. Accessed: Aug. 24, 2018. [Online]. https://snap.stanford.edu/data/email-Eu-core-temporal.html

[29] *DBLP Collaboration Network and Ground-Truth Communities*. Accessed: Aug. 24, 2018. [Online]. https://snap.stanford.edu/data/com-DBLP.html

[30] *Autonomous Systems by Skitter*. Accessed: Aug. 24, 2018. [Online]. http://snap.stanford.edu/data/as-Skitter.html

[31] *Wikipedia Network of Top Categories*. Accessed: Sep. 3, 2018. [Online]. http://snap.stanford.edu/data/wiki-topcats.html

[32] *Apache Spark Streaming*. Accessed: Jul. 28, 2018. [Online]. Available: https://spark.apache.org/streaming/

[33] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 325–336.

[34] X. Hu, Y. Tao, and C. W. Chung, "I/O-efficient algorithms on triangle listing and counting," *ACM Trans. Database Syst.*, vol. 39, no. 4, p. 27, 2014.

[35] M. Besta, and T. Hoefler. (Jun. 2018). "Survey and taxonomy of lossless graph compression and space-efficient graph representations." [Online]. Available: https://arxiv.org/abs/1806.01799

[36] W. Henecka, and M. Roughan, "Lossy compression of dynamic, weighted graphs," in *Proc. 3rd Int. Conf. Future Internet Things Cloud*, Aug. 2015, pp. 427–434.

BATJARGAL DOLGORSUREN received the B.S and M.S degrees in computer science from the Mongolian University of Science and Technology, in 2010 and 2012, respectively. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Kyung Hee University, South Korea. She was a Lecturer with the Department of Computer Science, Mongolian University of Science and Technology, from 2010 to 2014. Her research interests include streaming graph mining and distributed computing. She has received the Best Paper Awards at BigDAS 2016 and the Best Presentation Award at BigComp 2017 International Conference for papers related to streaming graph mining and graph algorithms, respectively.

KIFAYAT ULLAH KHAN received the B.S. degree from Gomal University, Pakistan, in 2005, the M.S. degree from the University of Greenwich, U.K., in 2007, and the Ph.D. degree from Kyung Hee University, South Korea, in 2016. He was a Postdoctoral Fellow with the Department of Computer Science and Engineering, Kyung Hee University, South Korea, from 2016 to 2018. He is currently an Assistant Professor with the Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan. He has several publications in the form of patents, journals, and conferences of prestige, including some research works currently in progress/review. His research interests include data mining, data warehousing, graph summarization, artificial intelligence, the Internet of Things, and big data. He is an Active Researcher and working on various research assignments in collaboration with researchers from various parts of the globe.

MOSTOFA KAMAL RASEL received the B.S. degree in computer science and information technology from the Islamic University of Technology, Dhaka, Bangladesh, in 2009, and the M.S. degree in computer science and engineering from Kyung Hee University, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering. His research interests include graph compression, graph mining, and big data processing. He received the Best Paper Award from the 3rd International Conference on Big Data and Smart Computing, in 2016.

YOUNG-KOO LEE received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 1988, 1994, and 2002, respectively. From 2002 to 2004, he was a Postdoctoral Fellow with the Advanced Information Technology Research Center, KAIST, South Korea, and a Postdoctoral Research Associate with the Department of Computer Science, University of Illinois at Urbana Champaign, USA. Since 2004, he has been a Professor with the Department of Computer Engineering, College of Electronics and Information, Kyung Hee University, South Korea. His research interests include ubiquitous data management, data mining, activity recognition, bioinformatics, on-line analytical processing, data warehousing, database systems, spatial databases, and access methods.