

# Pool Compression for Undirected Graphs

MUHAMMAD IRFAN YOUSUF<sup>1</sup> AND SUHYUN KIM<sup>2</sup> 

<sup>1</sup>Department of Computer Science, University of Engineering and Technology (New Campus), Lahore 39161, Pakistan

<sup>2</sup>Center for Artificial Intelligence, Korea Institute of Science and Technology, Seoul 02792, Republic of Korea

Corresponding author: Suhyun Kim (suhyun\_kim@kist.re.kr)

This work was supported in part by the National Research Council of Science & Technology (NST) Grant by the Korea Government [Ministry of Science and ICT (MSIT)] under Grant CRC-20-02-KIST, and in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korea Government (MSIT) (Development of Ultra-High Speech Quality Technology for Remote Multi-Speaker Conference System) under Grant 2021-0-00456.

**ABSTRACT** We present a new graph compression scheme that intrinsically exploits the similarity and locality of references in a graph by first ordering the nodes and then merging the contiguous adjacency lists of the graph into blocks to create a pool of nodes. The nodes in the adjacency lists of the graph are encoded by their position in the pool. This simple yet powerful scheme achieves compression ratios better than the previous methods for many datasets tested in this paper and, on average, surpasses all the previous methods. The scheme also provides an easy and efficient access to neighbor queries, e.g., finding the neighbors of a node, and reachability queries, e.g., finding if node  $u$  is reachable from node  $v$ . We test our scheme on publicly available graphs of different sizes and show a significant improvement in the compression ratio and query access time compared to the previous approaches.


**INDEX TERMS** Graph compression, merging adjacency lists, node ordering, Elias-Gamma encoding.

## I. INTRODUCTION

In the last few years, big graphs have become a focus of intense research activities, performed by both academic and industrial research centers. Big graphs include Web graphs, Online Social Networks, Collaboration Graphs, Technological Networks, and many more. Needless to say, big graphs may contain billions of nodes and the efficient processing of these huge objects is becoming increasingly important for different research domains. Compressing such objects can accelerate graph processing by reducing the amount of I/O accesses and memory requirements to store it. In the past, compression techniques were developed specifically for Web graphs that rely on a specific ordering of the nodes (lexicographical URL ordering) and produce good compression ratios [1], [2].

Two properties of lexicographical URL ordering have been observed to hold:

- Similarity: proximal pages in the lexicographic ordering tend to have common neighbors.
- Locality: a page is likely to point to pages nearby in the lexicographic ordering because of intra-domain links.

The associate editor coordinating the review of this manuscript and approving it for publication was Sun-Yuan Hsieh .

These observations were exploited by compression techniques [1]–[3] to compress a Web graph down to a few bits per link for storage. With the success of lexicographical URL ordering, it was presumed that node ordering plays a crucial role in the performance of a compression scheme. However, the lexicographic ordering of URLs for Web graphs is both natural and crucial. The question is: can we find such orderings for other graphs, in particular, for social networks?

The two state-of-the-art techniques [4], [5] developed for social graphs try to answer this question by introducing shingle ordering [4] and ordering based on Layered Label Propagation [5]. Chierichetti *et al.* [4] address social networks and prove hardness results for different node orderings. They propose BackLinks compression scheme that extends Boldi and Vinga's BV scheme [1] and targets directed networks in their work. BackLinks scheme takes advantage of reciprocal edges in a directed graph. Their main contribution is the shingle ordering for social networks that preserves both locality and similarity. Boldi *et al.* [5] propose Layered Label Propagation (LLP), a compression-friendly vertex ordering for social networks. LLP is iterative and produces a sequence of node orderings; at each iteration, the propagation algorithm is run with a suitable value of its parameter and the resulting labeling is then turned into an ordering of the graph that keeps nodes with the same label close to one another. LLP ordering

also preserves both locality and similarity in social networks. Both of these techniques take advantage of reference compression [1] and compress the adjacency list of a node with reference to the adjacency list of another node in the ordering.

In this paper, we propose a new scheme for compressing undirected social graphs (e.g., friendship graphs, collaboration graphs). We go beyond the reference compression and make a pool of nodes by merging the adjacency lists of nodes within a window (at least in this aspect our paper resembles this study [6]). However, our approach differs in two aspects. 1) The authors in [6] assumed that the order of nodes in a graph is irrelevant but we observe that a good node ordering preserves locality and similarity in graphs and when we merge the adjacency lists of ordered nodes, we can remove more duplicate values than in the case when the nodes are unordered. 2) Instead of using the flag bits (see [6]) to reconstruct the original lists, if we use the position of nodes in the merged lists we can not only achieve a better compression ratio but can also reduce the time to answer a neighbor query by many folds. Based on these two observations we propose a new graph compression algorithm that creates a pool of nodes by merging the adjacency lists of the graph and we encode the position of nodes in the pool to reconstruct the original graph. We call this algorithm Pool Compression and show that it compresses a graph to a lesser number of bits than the previous approaches and improves the query access time by many folds.

The rest of the paper is organized as follows. In section 2, we present our approach to compressing graphs in detail. In section 3, we give the details of the real-world datasets used and the baseline methods for comparison. In section 4, we present the experimental results of compressing real-world social graphs by our scheme and compare it with the previous techniques. In section 5, we discuss the related work from literature. We conclude the paper in section 6.

## II. POOL COMPRESSION

Pool Compression (PC) incorporates two main ideas. First, it orders the nodes in the graph using a node ordering scheme, e.g., Breadth First Ordering (BFS), Depth First Ordering (DFS), Shingle Ordering [4] etc. Second, it merges the adjacency lists of nodes in a window and creates a pool of nodes so that we can encode the position of nodes in the pool to compress a graph. Algorithm 1 presents the pseudo-code of PC. We detail the algorithm as follows:

**1. Input:** We read the input graph in the form of an adjacency list. We assume that the node IDs are integers.

**2. Order:** We order the nodes in the graph. We will identify each node in the graph with its position in the ordering.

**3. Pool of Nodes:** We merge the adjacency lists of nodes within a window to create a block of integer IDs. Instead of storing the integer IDs, we store the difference between them. Next, we apply Elias Gamma encoding [7] and combine the blocks in a pool. While encoding, each block is preceded

---

### Algorithm 1 Pool Compression (PC)

---

**Input:** Graph (G), Window size (w)

**Output:** Pool of nodes (Pool), Position of nodes (Pos)

```

1: Variables:
   list : list of neighbors of a node
   block : block of merged lists
    $|G|$  : size of G
    $e_j$  : jth element of a list
2: Pool = null
3: Pos = null
4: Order the nodes in G
5: for ( $i = 1; i \leq |G|; i = i + w$ ) do
6:    $q = \lceil i/w \rceil$ 
7:    $block_q = list_i \cup list_{i+1}, \dots, \cup list_{i+w-1}$ 
8:    $block_q = RemoveDuplicate(block_q)$ 
9:    $block_q = DeltaEncode(block_q)$ 
10:   $block_q = EliasGamma(block_q)$ 
11:  Pool = Pool  $\cup$   $block_q$ 
12: end for
13: for ( $i = 1; i \leq |G|; i = i + 1$ ) do
14:    $q = \lceil i/w \rceil$ 
15:   for ( $j = 1; j \leq |list_i|; j = j + 1$ ) do
16:      $Pos_j = GetPosition(e_j, block_q)$ 
17:   end for
18:    $Pos_j = DeltaEncode(Pos_j)$ 
19:    $Pos_j = EliasGamma(Pos_j)$ 
20:   Pos = Pos  $\cup$   $Pos_j$ 
21: end for

```

---

by the total number of entries in the block to make it self-delimiting.

**4. Position of Nodes:** Given the adjacency list of a node, we get the position of each node in the corresponding block in the pool. We store the difference between the positions of nodes and apply Elias Gamma encoding. We also encode the degree of a node to make it self-delimiting.

**5. Output:** PC outputs the pool of nodes and the position of nodes in the pool. In order to reproduce the original graph from the compressed PC graph, we need these two files.

Table 1 shows the working of PC with window size  $w = 4$ . The toy graph consists of 8 nodes and 20 edges. Table 1(a) shows the original graph in the adjacency list format whereas Table 1(b) shows the same graph after shingle node ordering. We then merge the adjacency lists of nodes within the window to create blocks and combine them into a pool of nodes as shown in Table 1(c). We apply Delta Encoding to the members of the pool in each block. In order to make the pool self delimiting, we also encode the number of entries in each block. Finally, the graph is compressed by getting the position of each node in the corresponding block in the pool and by applying Delta Encoding as shown in Table 1(d). The degree of nodes is also encoded to make the position list self-delimiting.

TABLE 1. Working of pool compression.

(a) Original graph			(b) After Shingle node ordering		
Node	Degree	Neighbors	Node	Degree	Neighbors
1	4	2 3 6 7	3	3	1 2 4
2	3	1 3 4	2	3	1 3 4
3	3	1 2 4	7	3	1 6 8
4	3	2 3 5	6	2	1 7
5	1	4	1	4	2 3 6 7
6	2	1 7	4	3	2 3 5
7	3	1 6 8	5	1	4
8	1	7	8	1	7

(c) Pool Data				
Block	Total Entries	Pool Members	Delta Encoding	Bits required for Elias Gamma Encoding of 2nd and 4th column
1	7	1 2 3 4 6 7 8	1 1 1 1 2 1 1	5+1+1+1+1+3+1+1
2	6	2 3 4 5 6 7	2 1 1 1 1 1	5+3+1+1+1+1+1

(d) Position Data				
Node	Degree	Position in Pool	Delta Encoding	Bits required for Elias Gamma Encoding of 2nd and 4th column
3	3	1 2 4	1 1 2	3+1+1+3
2	3	1 3 4	1 2 1	3+1+3+1
7	3	1 5 7	1 4 2	3+1+5+3
6	2	1 6	1 5	3+1+5
1	4	1 2 5 6	1 1 3 1	5+1+1+3+1
4	3	1 2 4	1 1 2	3+1+1+3
5	1	3	3	1+3
8	1	6	6	1+5

As mentioned in the third and fourth step of the algorithm above, we apply Elias Gamma encoding [7] to encode and store the graph. To represent a number  $x$ , Elias Gamma uses  $2 * \log_2 x + 1$  bits. In Table 1(c), we also present the number of bits required for Elias Gamma encoding of the pool data. It should be noted that the overlapped numbers in the pool belong to different blocks. For example, node 2, 3, 4, 6, and 7 belong to both block 1 and 2 in Table 1(c). We encode and store them twice as shown in the table. Similarly, Table 1(d) shows the number of bits required for Elias Gamma encoding of the position data. We need a total of 93 bits to store the pool and position data under Elias Gamma encoding. Therefore, we need  $93/20 = 4.65$  bits per edge to store this graph under PC scheme. To give a theoretical estimate, we consult Elias Gamma encoding method and note that encoding integer 1 needs 1 bit, encoding next two integers, i.e., 2,3 needs 3 bits, encoding next four integers, i.e., 4,5,6,7 needs 5 bits and so on. Therefore, we can write

$$\begin{aligned}
 \text{Total Bits Required} &= (\# \text{ of } 1s) + 3(\# \text{ of } 2s, 3s) \\
 &\quad + 5(\# \text{ of } 4s, 5s, 6s, 7s) \\
 &\quad + 7(\# \text{ of } 8s \text{ to } 15s) + \dots \quad (1)
 \end{aligned}$$

Equation 1 shows that as  $x$  increases, we need more number of bits to encode it. It seems that a compression scheme that orders the nodes in such a way that two consecutive node IDs have a smaller gap is likely to give better compression ratios. If we apply eq. 1 to our example on Table 1(c & d), we find that we need  $24 + 3(13) + 5(6) = 93$  bits to encode this toy graph. It is clear that the greater the number of 1s we have in our data the lesser number of bits are required to encode it.

This is the reason we apply this Elias Gamma encoding to the gaps or differences between numbers.

To produce the original graph from the compressed graph, we need 1) pool data and 2) position data. To continue with the toy example of Table 1, we need Elias Gamma encoding of the last columns of Table 1(c & d). It will have all the required information such as total entries in the pool and its members, degree of a node and the position of its neighbors in the pool. We identify a node by its position in the ordering, e.g., node 7 in Table 1(d) is on 3rd position. Similar to [8], we use a hash table for this purpose. A hash table  $H(\text{key}, \text{value})$  returns the value (i.e., position of a node in the ordering) when given the key (i.e., node ID). For example, to reverse node 7, we get its position in node ordering using the hash table and find that it is on 3rd position. Since the window size is 4 in Table 1, therefore, we calculate that node 7 (3rd position in ordering) is in block 1. Using position data, we get that its degree is 3 and we also get the position of its neighbors in the pool. Consulting pool data, we find that nodes 1, 6 and 8 are on position 1, 5 and 7 in block 1 of pool data and return the same as the neighbors of node 7. Similarly, using Elias Gamma encoded data, we can reverse construct the whole graph.

### III. DATASETS AND BASELINE METHODS

We perform our experiments on 15 real graphs available at [9], [10]. The basic properties and description of these graphs are given in table 2. In these graphs, there is a co-purchase network (Amazon), a lexical network (WordNet), a technical network (Skitter) and a co-authorship network (DBLP). All other networks are social networks.

TABLE 2. Real-world graphs used in the experiments.

Dataset	Total Nodes	Total Edges	Description
WordNet	146,005	656,999	This is the lexical network of words from WorldNet Database. Nodes are Words and links are relationships e.g., synonyms, antonyms. [9]
Gowalla	196,591	950,327	This undirected network contains user–user friendship relations from Gowalla, a former location-based social network where user shared their locations. [9]
DBLP	317,080	1,049,866	This is the co-authorship network of the DBLP computer science bibliography. An edge between two authors represents a common publication. [9]
Amazon	334,863	925,872	This is the co-purchase network of Amazon based on the "customers who bought this also bought" feature. [9]
FourSquare	639,014	3,214,986	Foursquare is a location-based online social network. The dataset contains a list of all of the user-to-user links. [10]
Digg	770,799	5,907,132	Friendship links of users of Digg website. Nodes are users and edges represent their friendships. [9]
Twitter	1,112,702	2,278,852	Nodes are Twitter users and edges are retweets. These were collected from various social and political hashtags. [10]
Last.fm	1,191,805	4,519,330	This is a social network of friendship relations of last.fm music website. Nodes are users and edges are friendship relations. [10]
Hyves	1,402,673	2,777,919	This is the social network of Hyves, a Dutch online social network. Nodes are users and edges are friendship relations. [10]
Skitter	1,696,415	11,095,298	This is the network of autonomous systems on the Internet connected to each other, from the Skitter project. [9]
Flickr	1,715,255	15,550,782	This is the social network of users and their connections on Flickr, a popular image sharing website. [9]
Flixster	2,523,386	7,918,801	This is the social network of Flixster, a movie rating site on which people can meet others with a similar movie taste. [10]
Facebook	2,937,612	20,959,854	This is a friendship network on Facebook. Nodes are users and edges are friendship relations. [10]
YouTube	3,223,585	9,375,374	This is the friendship network of the video-sharing site YouTube. Nodes are users and edges are their friendship relations. [9]
LiveJournal	5,204,175	48,709,621	This is the social network of LiveJournal users and their connections. [9]

## A. BASELINE METHODS

We use the following state-of-the-art compression schemes as our baselines to compare against PC. We chose to compare against these techniques because these are technically re-ordering schemes and also exploit the similarity and locality of nodes in a graph.

### 1) LIST MERGING (LM)

The List Merging(LM) compression scheme [6] merges the adjacency lists to create a long list and then uses flag bits to describe to which input lists a given integer on the output list belongs. We implement the LM-bitmap variant of LM. Unless specified otherwise, we use 64 for the chunk size (or window) parameter, as recommended in the original work [6]. As mentioned in the introduction, PC resembles LM in the list merging part, however, PC uses position bits instead of flag bits to describe the position of a node in the merged list. We will see in the experimental section that replacing flag bits with the position bits improves the compression ratio a lot.

### 2) BACKLINKS (BL)

The Backlinks (BL) compression scheme presented in [4]. Briefly, BL orders nodes in shingle ordering and applies BV format [1] to compress a graph. We use double shingle ordering as it gives better results [4]. There is one minor change in our implementation of the BL scheme: we do not have *reciprocal edges* because we have undirected graphs.

### 3) LAYERED LABEL PROPAGATION (LLP)

The compression scheme based on Layered Label Propagation (LLP) presented in [5]. Briefly, LLP attempts to obtain a labeling that considers clusters of various resolutions. It then labels and orders nodes according to the clusters they belong to. This way LLP preserves both locality and similarity.

### 4) SLASHBURN (SB)

The Slashburn (SB) compression scheme presented in [11] proposes a compression-friendly ordering of nodes by exploiting the hubs and spokes of the hubs. With this ordering, we can find a compact representation of the adjacency matrix, which in turn leads to good compression. SB uses information theoretic lower bound for encoding the bits for storage.

## IV. EXPERIMENTS

In this section, we present the experimental results to answer the following questions:

- 1) How much can we improve by ordering the nodes before applying LM.
- 2) How much improvement can we obtain by replacing the flags bits with the position bits in LM.
- 3) What is the effect of window size on PC.
- 4) How well does PC compress graphs compared to the baseline methods.
- 5) How efficiently can we process neighbor queries and reachability queries on the compressed graphs.

All the experiments were run on a machine equipped with an Intel Core i7 CPU and 64 GB of RAM. All presented

algorithms were implemented in Java and launched on the 64-bit JVM 7. All the experiments were run and compiled in the same environment. As common in graph compression, we present the compression ratios in bits per edge. While presenting results, we will use the notation  $PC(w, \text{order})$  where  $w$  is the window size, e.g., 8, 16, 32 etc., and order means the node ordering, e.g., BFS, DFS, Shingle etc.

### A. LIST MERGING WITH NODE ORDERING

We perform an experiment in which we order the nodes before applying LM. We set the window size (or chunk size in [6])  $w = 64$  and order the nodes using DFS, BFS and Shingle ordering. We present the results in Table 3 in the form of bits per edge for all the datasets along with the percentage improvements relative to the original LM [6]. The table shows that ordering the nodes in the graph before merging lists improves the original LM. For all datasets, LM with a node ordering compresses a graph to a lower number of bits per edge than the original LM. On average, DFS, BFS and Shingle ordering improve over LM by 6%, 5% and 8% respectively. This experiment supports our first observation, as mentioned in the introduction, that a good node ordering preserves locality and similarity in graphs and when we merge the adjacency lists of ordered nodes, we can save more number of bits by removing more duplicate values than in the case when the nodes are unordered.

### B. LIST MERGING WITH POSITION BITS

In this experiment, we replace the flag bits of LM with the position bits. In other words, we merge the lists as per the LM compression scheme, however, we describe a node by its position in the long merged list and encode it as described in Algorithm 1. The window size is set to  $w = 64$ . We present the results in Table 4. We see that position bits improve LM in all the datasets except WordNet and Skitter. On average, LM with position bits improves over LM with flag bits by 5%. This experiment supports our second observation that many bits in the flag sequence could not be set and hence a large number of bits in the flag sequence go in waste. It has been pointed out in the original LM that a large number of list indicators have only one set bit in the flag sequence [6]. When we use flag bits, we need  $w$  number of bits to represent each entry in the long merged list. By replacing flag bits with the position bits and storing the difference between successive positions, we need far less than  $w$  bits per entry in the merged list and as a result, we can save a significant number of bits per edge when storing the graph.

### C. THE EFFECT OF WINDOW SIZE

In this experiment, we test Pool Compression at different window size. We set the window size  $w = 8, 16, 32, 64$  and apply different node ordering in PC. We find that DFS node ordering gives better results, though marginally, than BFS and Shingle ordering, therefore, we show the results in Table 5 for DFS ordering only. The results show that as the window size increases, generally, PC needs a lesser number of bits per

**TABLE 3.** The number of bits per edge achieved when the nodes are ordered before applying List Merging compression scheme. The window size is 64, i.e., we merge 64 adjacency lists. The table also shows the percentage improvements relative to List Merging without any node ordering, i.e., original LM. Boldface values are the best results.

Dataset	LM	LM + DFS	LM + BFS	LM + Shingle
WordNet	9.807	<b>9.394</b> (-5%)	10.034 (+2%)	9.427 (-4%)
Gowalla	14.878	13.169 (-12%)	13.241 (-12%)	<b>13.098</b> (-12%)
DBLP	14.134	<b>10.362</b> (-27%)	11.355 (-20%)	10.803 (-24%)
Amazon	16.520	11.952 (-28%)	13.692 (-18%)	<b>11.912</b> (-28%)
FourSquare	9.952	9.941 (-1%)	<b>9.504</b> (-5%)	9.994 (0%)
Digg	11.916	12.529 (+5%)	11.815 (-1%)	<b>11.675</b> (-3%)
Twitter	20.507	18.472 (-10%)	<b>18.241</b> (-12%)	18.274 (-11%)
LastFM	14.694	15.178 (+3%)	14.716 (0%)	<b>14.518</b> (-2%)
Hyves	12.801	12.904 (0%)	12.557 (-2%)	<b>12.318</b> (-4%)
Skitter	8.871	<b>7.922</b> (-11%)	8.868 (-1%)	8.847 (-1%)
Flicker	11.495	11.672 (+1%)	11.479 (-1%)	<b>11.452</b> (-1%)
Flixster	12.820	12.907 (0%)	12.812 (-1%)	<b>12.346</b> (-4%)
Facebook	16.233	16.954 (+4%)	16.127 (-1%)	<b>16.018</b> (-2%)
YouTube	15.797	16.372 (+3%)	16.093 (+1%)	<b>15.429</b> (-3%)
LiveJournal	21.535	21.248 (-2%)	21.642 (0%)	<b>20.834</b> (-4%)
Average	14.131	13.399 (-6%)	13.478 (-5%)	<b>13.131</b> (-8%)

**TABLE 4.** The number of bits per edge achieved when the flag bits are replaced with the position bits in List Merging compression scheme. The window size is 64, i.e., we merge 64 adjacency lists. The table also shows the percentage improvements relative to List Merging with flag bits, i.e., original LM. Boldface values are the best results.

Dataset	LM	LM + Position Bits
WordNet	<b>9.807</b>	10.111 (+3%)
Gowalla	14.878	<b>14.366</b> (-4%)
DBLP	14.134	<b>13.393</b> (-6%)
Amazon	16.520	<b>15.484</b> (-7%)
FourSquare	9.952	<b>9.471</b> (-5%)
Digg	11.916	<b>11.586</b> (-3%)
Twitter	20.507	<b>19.013</b> (-8%)
LastFM	14.694	<b>13.745</b> (-7%)
Hyves	12.801	<b>12.224</b> (-5%)
Skitter	<b>8.871</b>	8.898 (0%)
Flicker	11.495	<b>10.952</b> (-5%)
Flixster	12.820	<b>12.071</b> (-6%)
Facebook	16.233	<b>15.551</b> (-5%)
YouTube	15.797	<b>14.725</b> (-7%)
LiveJournal	21.535	<b>20.828</b> (-4%)
Average	14.131	<b>13.494</b> (-5%)

edge for the same graph. The window size of  $w = 32$  seems the best for PC because when  $w > 32$ , the size of the block of merged lists increases and hence we need more bits to represent the position of a node somewhere far in that block. It would be interesting to mention that, unlike LM, the window size in PC could be any integer number greater than one. It should be noted that the choice of power of 2 window size for experimentation is simply arbitrary and mainly for the purpose of direct comparison with the LM compression scheme as LM has limitation of power of 2 window size. We can use any integer number greater than one for window size. This experiment sheds some light on the relation between windows size and the number of bits needed to represent a graph. It could be an interesting future extension to find an optimal window size that works for all types of graphs.

### D. COMPARISON WITH THE BASELINE METHODS

In this experiment, we compare PC with the baseline methods BL [4], LLP [5] and SB [11]. For LM, we use a window

**TABLE 5.** The effect of window size on the number of bits per edge in Pool Compression with DFS ordering. Boldface values are the best results.

Dataset	PC(8, DFS)	PC(16, DFS)	PC(32, DFS)	PC(64, DFS)
WordNet	8.012	<b>7.707</b>	7.773	8.046
Gowalla	12.291	11.743	<b>11.731</b>	11.847
DBLP	9.078	<b>8.718</b>	8.782	9.097
Amazon	9.589	9.084	<b>9.075</b>	9.356
FourSquare	9.848	9.409	9.226	<b>9.216</b>
Digg	12.079	11.716	<b>11.593</b>	11.624
Twitter	14.762	14.362	<b>14.353</b>	14.577
LastFM	16.448	16.063	<b>15.959</b>	16.032
Hyves	14.153	<b>13.689</b>	13.692	14.106
Skitter	8.282	7.665	<b>7.445</b>	7.482
Flicker	12.043	11.572	<b>11.522</b>	11.941
Flixster	14.885	14.481	<b>14.621</b>	14.722
Facebook	18.039	<b>17.555</b>	17.662	17.742
YouTube	16.015	15.405	<b>15.382</b>	15.643
LiveJournal	17.128	<b>16.371</b>	16.374	16.461
Average	12.843	12.369	<b>12.346</b>	12.526

**TABLE 6.** Comparison of Pool Compression with the previous compression schemes. For LM, we present the best results achieved with a window size of 64. For PC, we order the nodes in DFS order and use a window size of 32 for the best results. The other methods are implemented as presented in their original work. Boldface values are the best results.

DataSet	LM	PC (32, DFS)	BL	LLP	SB
WordNet	9.807	<b>7.773</b>	9.114	8.975	11.881
Gowalla	14.878	<b>11.731</b>	15.558	19.024	11.832
DBLP	14.134	<b>8.782</b>	10.788	18.486	12.621
Amazon	16.520	<b>9.075</b>	11.127	21.864	14.042
FourSquare	9.952	<b>9.226</b>	11.689	11.776	10.671
Digg	11.916	11.593	12.561	12.924	<b>9.823</b>
Twitter	20.507	14.353	16.724	21.111	<b>13.495</b>
LastFM	14.694	15.959	18.268	18.439	<b>12.882</b>
Hyves	<b>12.801</b>	13.692	15.582	16.114	13.902
Skitter	8.871	<b>7.445</b>	11.487	11.622	12.751
Flicker	11.495	11.522	13.398	13.519	<b>10.012</b>
Flixster	<b>12.820</b>	14.621	15.535	16.225	13.062
Facebook	16.233	17.662	18.385	18.714	<b>15.293</b>
YouTube	15.797	<b>15.382</b>	17.979	18.342	17.032
LiveJournal	21.535	<b>16.374</b>	18.908	21.261	20.981
Average	14.131	<b>12.346</b>	14.474	16.560	13.352

size  $w = 64$  and for PC we use DFS ordering and  $w = 32$ , i.e., PC(32, DFS) as these settings give the best results for these compression schemes. We present the results in Table 6. We see that PC outperforms in eight datasets whereas SB performs the best in five datasets and LM in two. The reason that SB performs better than PC in some datasets is that SB works on the adjacency matrix of a graph and uses information theoretical lower bound to compress the adjacency matrix. On average, PC gives the lowest number of bits per edge among these methods and gives a nearly 13% improvement over LM in compressing a graph.

### E. QUERY PROCESSING TIME

In general, a graph query is a computable function that returns some specific information about the graph. In this experiment, we consider two classes of queries commonly used in practice. 1) neighborhood queries, to find nodes connected to a given node in a graph and 2) reachability queries, to answer whether node  $u$  is reachable from node  $v$  in  $G$  in Boolean form. In the first version of this experiment, we measure the time to process a neighbor query in PC(32, DFS), LM and

BL compressed graphs. A typical query seeks the immediate neighbors of a node, therefore, we aim to answer a simple query such as: *who are the immediate neighbors of node  $u$ ?* Since, BL and LLP differ only in the ordering of nodes, therefore, their query handling ability is very close to each other and we only consider BL results for comparison. We drop SB because it does not support neighbor queries rather it was designed to boost the performance of matrix-vector multiplication of graph adjacency matrices [11]. The results are presented in table 7. The table shows the average time taken in microseconds ( $\mu s$ ) to process a query where the time is averaged over 10,000 random queries. We see that PC is many folds efficient than LM and BL for handling queries. On average, PC takes  $1.69\mu s$  to process a query while LM and BL need  $38.35\mu s$  and  $8.72\mu s$  respectively to answer such queries. In other words, PC can access data nearly 22 times faster than LM and 5 times faster than BL. The reason is that PC can directly access the neighbors of a node by using the position of nodes in the pool whereas, in the case of LM, we need to decompress the flag sequence of every entry in the long merged list to check if it belongs to the node in question. In the case of BL, we need to query all the preceding nodes in the prototype chains until a node is not prototyped resulting in the slow serving of adjacency queries [4]. This experiment shows that PC does not sacrifice time for better compression ratios. Such neighbor queries can be extended easily to answer Breadth First and Depth First searches.

In the second version of this experiment, we execute reachability queries on PC graphs. We define a reachability query on a graph  $G$  as a Boolean query that asks whether node  $u$  is reachable from node  $v$  in  $G$ . We perform Breadth First Search to find the connected components in the graph and then execute reachability queries. The connected components of an undirected graph can be identified in linear time. The results are presented in Table 8. The table shows the average time taken in microseconds to process a query where the time is averaged over 10,000 random queries. Again, we see that PC surpasses other compression schemes in executing reachability queries. On average, PC processes a query in  $22.14\mu s$  whereas LM and BL take  $564.38\mu s$  and  $124.49\mu s$  respectively.

### V. RELATED WORK

A large portion of research in graph compression is dedicated to compressing Web graphs while recently research on compressing social graphs is also getting its pace. The authors in [2] introduced the idea of finding pages with similar sets of neighbors in the context of compressing Web graphs and applied the Huffman-based scheme to in-degrees to compress web graphs using reference encoding and log-encoding. Randall *et al.* [3] suggested lexicographic ordering as a way to obtain good Web graph compression, utilizing both similarity and locality. Raghavan and Garcia-Molina [12] considered a hierarchical view of the Web graph to achieve compression while Suel and Yuan [13] adopted a structural approach to compress Web graphs. One of the major contributions in

**TABLE 7.** Average time in microseconds ( $\mu s$ ) to process a neighbor query in graphs compressed with different compression schemes.

Dataset	PC	LM	BL
WordNet	0.612	26.45	6.318
Gowalla	0.862	26.44	6.875
DBLP	0.963	36.18	7.817
Amazon	1.028	34.94	7.982
FourSquare	2.042	31.99	7.766
Digg	1.253	38.17	8.082
Twitter	1.183	30.67	7.814
LastFM	0.927	29.84	6.738
Hyves	0.981	34.35	7.181
Skitter	2.173	43.43	9.728
Flicker	2.163	43.73	9.193
Flixster	3.162	34.59	8.541
Facebook	2.364	46.93	10.678
YouTube	2.483	50.76	11.026
LiveJournal	3.185	66.81	15.061

**TABLE 8.** Average time in microseconds ( $\mu s$ ) to process a reachability query in graphs compressed with different compression schemes.

Dataset	PC	LM	BL
WordNet	8.36	395.34	78.57
Gowalla	10.57	413.56	93.56
DBLP	12.58	506.56	110.35
Amazon	14.74	537.35	104.58
FourSquare	25.34	450.73	112.47
Digg	17.92	560.29	115.34
Twitter	14.87	450.36	128.45
LastFM	13.12	445.68	98.35
Hyves	12.78	510.46	95.21
Skitter	27.12	628.47	147.46
Flicker	24.73	645.86	131.04
Flixster	40.35	520.45	126.39
Facebook	30.35	675.45	154.68
YouTube	33.46	744.85	167.49
LiveJournal	45.76	980.24	203.46

this direction was offered by Boldi and Vigna [1], who both developed a generic Web graph compression framework that takes into account the locality and similarity of Web pages and obtained very strong compression performance. Next, Boldi *et al.* [14] test several existing vertex permutations. They also propose two new permutations based on the Gray ordering. Moreover, Boldi *et al.* [5] observe that it is important for high compression ratios to use an ordering of vertex IDs such that the vertices from the same host are close to one another and propose Layered Label Propagation (LLP), a compression-friendly vertex ordering targeting social networks. Buehrer and Chellapilla [15] used the frequent pattern mining approach to compress Web graphs and generate virtual nodes from frequent itemsets in the adjacency data to compress graphs effectively. Anh and Moffat [16] propose a hierarchical scheme for compressing web graphs. The key idea is to partition the adjacency arrays into groups of consecutive arrays. Then, sequences of consecutive integers in each of the arrays are replaced with new symbols to the graph. Similarly, the authors in [17] work on contiguous blocks of adjacency lists and merge the block into a single ordered list for a better compression ratio. Maneth and Peternek [18] recently proposed a scheme that recursively detects substructures occurring multiple times in the same graph and uses

grammar rules to represent them. Moreover, they show that some queries (e.g., reachability between two nodes) can be resolved in linear time when they are executed over the grammar, enabling speedups proportional to the compression ratio. The main idea due to Asano *et al.* [19] is to identify identical blocks in the adjacency matrix of a graph and then represent it with a sequence of blocks combined with some metadata information on the block type. Chierichetti *et al.* [4] provide three contributions targeted at social network compression. First, they prove hardness results about several types of vertex reordering. Second, they propose the BL compression scheme that extends the BV scheme from the WebGraph framework [1]. BL takes advantage of reciprocal links. Third, the notable contribution is the shingle ordering that preserves both locality and similarity. Liakos *et al.* [20], [21] use the fact that LLP reordering [5] enhances the locality with a large “stripe” around the diagonal that groups a large fraction of edges. They use a bit-vector to represent these edges and ultimately reduce space to store a network. Zhang *et al.* [22] propose the bound-triangulation algorithm. The main idea is to use a data structure that stores triangles efficiently. The motivation is that many web graphs and social networks contain a large number of triangles, thus priority placed over storing this motif efficiently reduces the required storage. Maserrat and Pei [23] argue that social networks should be compressed in a way that they still can be queried efficiently without decompression. Especially, neighbor queries, which search for all neighbors of a query vertex, are the most essential operations on social networks. They develop an effective social network compression approach achieved by a novel Eulerian data structure using multi-position linearizations of directed graphs.

GraphZip [24] is a graph compression and encoding framework based on the study that real-world graphs often form many cliques of a large size. Using this as a foundation, the authors proposed a technique to decompose a graph, especially a sparse graph, into a set of large cliques, which is then used to compress and represent the graph succinctly. Star-based graph compression method [25] compresses a graph by shrinking a collection of disjoint sub-graphs called stars. The authors find that compressing a graph into the optimal star-based compressed graph with the highest compression ratio is NP-complete and propose a greedy compression algorithm called StarZip. Another compression scheme [8] with the same name, StarZip, was motivated by the observation that all subgraph structures such as stars, bipartite forms, cliques, and chains etc. can be represented as star-shaped sub-graphs. 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. The authors apply this idea on streaming graphs and shatter a graph into a uniform representation of stars to compress it. The work [26] introduces a representation of graphs that uses one of the graphs automorphisms to describe a set of its edges. Detecting automorphisms is a natural way to identify redundant information presented in structured data.

The work explores two different classes of graphs to capture automorphisms, i.e., symmetry-compressible graphs and near-symmetry compressible graphs. The authors develop two algorithms that can be used to compress real-world graphs. Seo *et al.* [27] combine graph summarization and graph compression approaches to propose a greedy algorithm that reduces the size of a large graph by applying both the techniques. They also propose a cost model for calculating the compression ratio considering both the strategies. The proposed algorithm applies compression after summarizing the nodes from the graph. An approach called Zuckerli [28] improves multiple aspects of WebGraph [1] by using advanced compression techniques and novel heuristic graph algorithms. First, Zuckerli entropy-encodes the integers instead of zeta-encoding of WebGraph. Second, Zuckerli splits the nodes of a graph into chunks. Inside each chunk, degrees of the nodes are stored and represented by delta encoding. Thirdly, Zuckerli uses reference lists and blocks in the same way as WebGraph but with more sophistication. Fourth, it uses run-length encoding of zero gaps for interval representation. Finally, Zuckerli modifies the representation of the residuals, which are stored via delta encoding. The latest framework called Partition and Code [29] entails three steps to compress a graph. First, a partitioning algorithm decomposes the graph into subgraphs. Second, these are mapped to the elements of a small dictionary on which we learn a probability distribution. Third, an entropy encoder translates the representation into bits.

Besides web and social graphs, the researchers also worked on compressing Biological graphs, RDF graphs, Chemistry graphs and Geological graphs etc. Interested readers are referred to this recent survey [30] for details.

The main idea of vertex or graph relabeling is to change the initial IDs of vertices so that the new IDs, when stored, use less space. After relabeling, we can apply an encoding scheme such as gap encoding or delta encoding. Khalili *et al.* [31] relabel vertices so that similar vertices have closer IDs. Second, they group similar vertices and collapse edges between groups into single superedges. To keep track of the collapsed edges they use an auxiliary data structure. Dhulipala *et al.* [32] note that optimal compression-friendly relabeling of vertices is NP-hard. They extend the work of [4]. They recursively bisect the graph and, once the size of the partitions is small enough, compute a selected reordering for each partition. Finally, these partial results are combined to obtain the solution for the whole graph. Blandford *et al.* [33] relabel vertices based on recursive partitioning of the input graph to achieve compactness. Lim *et al.* [11] propose SlashBurn: a scheme that exploits high degree vertices and their neighbors to achieve high compression ratios. They also propose vertex relabeling that uses this observation and results in a space-efficient representation of the adjacency matrix.

## VI. CONCLUSION

We presented Pool Compression (PC), a graph compressor based on the merging of adjacency lists of nodes in

a graph and presenting the nodes by their positions in the pool of merged lists. PC is a simple yet effective compressor that achieves better compression results than the previous methods. PC not only beats its closest competitor, List Merging [6], but also outperforms previous state-of-the-art compression schemes. We also show that PC does not trade space for time and the query access time in PC is many folds lower than its competitors.

## REFERENCES

- [1] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Conf. World Wide Web (WWW)*, 2004, pp. 595–602.
- [2] M. Adler and M. Mitzenmacher, "Towards compressing web graphs," in *Proc. DCC Data Compress. Conf.*, Mar. 2001, pp. 203–212.
- [3] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener, "The link database: Fast access to graphs of the web," in *Proc. DCC Data Compress. Conf.*, Apr. 2002, pp. 122–131.
- [4] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2009, pp. 219–228.
- [5] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web (WWW)*, 2011, pp. 587–596.
- [6] S. Grabowski and W. Bieniecki, "Tight and simple web graph compression for forward and reverse neighbor queries," *Discrete Appl. Math.*, vol. 163, pp. 298–306, Jan. 2014.
- [7] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. IT-21, no. 2, pp. 194–203, Mar. 1975.
- [8] B. Dolgorsuren, K. U. Khan, M. K. Rasel, and Y.-K. Lee, "StarZIP: Streaming graph compression technique for data archiving," *IEEE Access*, vol. 7, pp. 38020–38034, 2019.
- [9] J. Kunegis, "Konect—The Koblenz network collection," in *Proc. Int. Conf. World Wide Web Companion*, 2013, pp. 1343–1350. [Online]. Available: <http://konect.uni-koblenz.de/>
- [10] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293. [Online]. Available: <http://networkrepository.com>
- [11] 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.
- [12] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *Proc. 19th Int. Conf. Data Eng.*, Mar. 2003, pp. 405–416.
- [13] T. Suel and J. Yuan, "Compressing the graph structure of the web," in *Proc. Data Compress. Conf.*, Mar. 2001, pp. 213–222.
- [14] P. Boldi, M. Santini, and S. Vigna, "Permuting web and social graphs," *Internet Math.*, vol. 6, no. 3, pp. 257–283, Jan. 2009.
- [15] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proc. Int. Conf. Web Search Web Data Mining (WSDM)*, 2008, pp. 95–106.
- [16] V. N. Anh and A. Moffat, "Local modeling for WebGraph compression," in *Proc. Data Compress. Conf.*, 2010, p. 519.
- [17] S. Grabowski and W. Bieniecki, "Merging adjacency lists for efficient web graph compression," in *Man-Machine Interactions*. Berlin, Germany, Feb. 2011.
- [18] S. Maneth and F. Peternek, "Compressing graphs by grammars," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 109–120.
- [19] Y. Asano, Y. Miyawaki, and T. Nishizeki, "Efficient compression of web graphs," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E92-A, no. 10, pp. 2454–2462, 2009.
- [20] P. Liakos, K. Papakonstantinou, and M. Sioutis, "On the effect of locality in compressing social networks," in *Proc. Eur. Conf. Inf. Retr.*, Apr. 2014, pp. 650–655.
- [21] P. Liakos, K. Papakonstantinou, and M. Sioutis, "Pushing the envelope in graph compression," in *Proc. 23rd ACM Int. Conf. Conf. Inf. Knowl. Manage.*, Nov. 2014, pp. 1549–1558.
- [22] L. Zhang, C. Xu, W. Qian, and A. Zhou, "Common neighbor query-friendly triangulation-based large-scale graph compression," in *Proc. Int. Conf. Web Inf. Syst. Eng.*, Oct. 2014, pp. 234–243.



- [23] H. Maserrat and J. Pei, "Neighbor query friendly compression of social networks," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 533–542.
- [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] F. Li, Z. Zou, J. Li, and Y. Li, "Graph compression with stars," in *Proc. 23rd Pacific-Asia Conf. Adv. Knowl. Discovery Data Mining (PAKDD)*, Macau, China, Apr. 2019, pp. 449–461.
- [26] U. Čibej and J. Mihelič, "Graph automorphisms for compression," *Open Comput. Sci.*, vol. 11, no. 1, pp. 51–59, Jan. 2021.
- [27] H. Seo, K. Park, Y. Han, H. Kim, M. Umair, K. U. Khan, and Y.-K. Lee, "An effective graph summarization and compression technique for a large-scaled graph," *J. Supercomput.*, vol. 76, no. 10, pp. 7906–7920, Oct. 2020.
- [28] L. Versari, I.-M. Comsa, A. Conte, and R. Grossi, "Zuckerli: A new compressed representation for graphs," *IEEE Access*, vol. 8, pp. 219233–219243, 2020.
- [29] G. Bouritsas, A. Loukas, N. Karalias, and M. M. Bronstein, "Partition and code: Learning how to compress graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 18603–18619.
- [30] M. Besta and T. Hoefler, "Survey and taxonomy of lossless graph compression and space-efficient graph representations," 2018, *arXiv:1806.01799*.
- [31] A. Y. H. Khalili and F. Oroumchian, "Web-graph pre-compression for similarity based algorithms," in *Proc. 3rd Int. Conf. Modeling, Simulation Appl. Optim.*, 2009, pp. 1–7.
- [32] L. Dhulipala, I. Kabiljo, B. Karrer, 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*, Aug. 2016, pp. 1535–1544.
- [33] D. K. Blandford, G. E. Blelloch, and I. A. Kash, "Compact representations of separable graphs," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 679–688.



**MUHAMMAD IRFAN YOUSUF** received the B.E. degree in electrical engineering from the University of Engineering & Technology, Lahore, the M.S. degree in information technology from the Pakistan Institute of Engineering and Applied Sciences, Islamabad, Pakistan, and the Ph.D. degree from the University of Science and Technology, Daejeon, South Korea. He worked as a Postdoctoral Researcher at the Imaging & Media Research Center, Korea Institute of Science and Technology, Seoul, South Korea, before joining the Department of Computer Science, University of Engineering and Technology (New Campus), Lahore, Pakistan. His research interests include peer-to-peer networks, graph data science, social network analysis, and compressing big graphs.



**SUHYUN KIM** received the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2005. Prior to joining the Korea Institute of Science and Technology (KIST), in 2007, he spent two years at IBM T. J. Watson Research Center. His current research interests include deep learning, data analysis, and peer-to-peer systems.

...