Efficient Graph Summarization using Weighted LSH at Billion-Scale

Quinton Yong
University of Victoria
Victoria, Canada
quintonyong@uvic.ca

Mahdi Hajiabadi
University of Victoria
Victoria, Canada
mhajiabadi@uvic.ca

Venkatesh Srinivasan
University of Victoria
Victoria, Canada
srinivas@uvic.ca

Alex Thomo
University of Victoria
Victoria, Canada
thomo@uvic.ca

ABSTRACT

Summarizing graphs is of paramount importance due to diverse applications of large-scale graph analysis. A popular family of summarization methods is the group-based approach. The general idea consists of merging nodes of the original graph into supernodes of the summary graph, encoding original edges into superedges/correction set edges, and dropping certain superedges or correction set edges (for lossy summarization). The current state of the art has several steps in its computation that are serious bottlenecks in terms of running time and scalability. In this work, we propose algorithm LDME, a correction set based graph summarization algorithm that produces compact output representations in a fast and scalable manner. To achieve this, we introduce (1) weighted locality sensitive hashing to drastically reduce the number comparisons required to find good node merges, (2) an efficient way to compute the best quality merges that produces more compact outputs, and (3) a new sort-based encoding algorithm that is faster and more robust. More interestingly, our algorithm provides performance tuning settings to allow the option of trading compression for running time. On high compression settings, LDME achieves compression equal to or better than the state of the art with up to 53x speedup in running time. On high speed settings, LDME achieves up to two orders of magnitude speedup with only slightly lower compression.

KEYWORDS

Graph Summarization, Weighted LSH, Jaccard Similarity

1 INTRODUCTION

Large scale graphs are widely used in many important real-world applications and represent web, communication, social, and transaction networks. These graphs contain as many as several billion nodes and edges, and are continually growing at a tremendous rate. As such, it is crucial to be able to represent them in a compact way using a method that is both efficient and scalable. Compressing graphs into a smaller form is an essential step when working with them in scenarios such as storing graphs, processing graphs, answering queries, and data visualization.

The solution is a graph compression technique known as *graph summarization*, which takes an input graph and produces a more compact representation of the input called the summary graph. The summary graph not only decreases the footprint of the original graph, but also allows us to do tasks such as efficiently answer queries [7, 28, 32] or perform more effective and insightful data visualization [5, 6, 16, 21, 40]. There are many general methods of graph summarization such as grouping nodes into supernodes based on some similarity or logical relation metric [9, 19, 24, 28, 29, 32, 33], reducing the bits required to represent graphs [2, 4, 30, 31], and removing unimportant nodes and edges [26, 36].

The most popular of the above is the group-based approach (c.f. [25]), which is also the focus of our work. In this approach, as defined in [28, 32], the output representation consists of a summary graph and correction set. The correction set is used to reconstruct the original graph from the summary graph either perfectly (lossless) or with some loss in information (lossy). Correction set based graph summarization algorithms consist of three broad steps: merge nodes of the original graph into supernodes of the summary graph, encode the original edges into superedges of the summary graph and correction set, and drop some edges from the summary graph and correction set to yield a more compact output (for lossy case).

The current state of the art correction set based graph summarization algorithm is SWeG of [32]. SWeG is faster than all of its competitors, yields better compression than other methods, and can also run in a distributed setting. SWeG improves upon the original framework of [28] by adding a dividing step that divides the nodes into smaller groups prior to merging (for parallelizability and efficiency) and introducing an approximation metric for finding nodes to merge. Despite the impressive performance of SWeG compared to other algorithms, there are several steps in the algorithm which bottleneck its performance. In particular, the merging algorithm is quadratic in the size of groups, so its running time suffers due to the dividing step not creating small enough groups of nodes. The merging step also uses an approximation metric to find good merges since [32] did not present an efficient way to directly compute the true best merges in a group. Finally, the encoding algorithm also becomes a bottleneck since it scales quadratically based on the number of supernodes, making it perform poorly for larger graphs.

This work proposes algorithm LDME (**L**ocality Sensitive Hashing **D**ivide **M**erge **E**ncode), an efficient and scalable correction set based graph summarization method which makes optimizations in each step of SWeG. In particular, LDME introduces weighted locality sensitive hashing to reduce the amount of computation during the merge phase, uses an efficient method of computing the best merges, and implements a faster and more scalable encoding algorithm. Our optimizations are such that we can now handle large datasets requiring only a single machine without the need for expensive clusters of machines. Additionally, LDME includes the benefit of being able to tune its performance to trade off compression for running time. In particular, LDME with high compression settings achieves up to 53x speedup with equal or better compression than SWeG and with high speed settings achieves up to two orders of magnitude speedup with only a small loss in compression.

Contributions

- New node dividing algorithm which introduces weighted locality sensitive hashing to significantly improve the running time of the bottleneck merging step
- Efficient method to directly compute the best nodes to merge that gives better overall compression and is faster than the approximation metric used in SWeG
- New edge encoding algorithm that scales better based on only the number of edges in the original graph and is up to 26x faster than SWeG (especially on very large graphs)
- A performance tuning technique to allow the choice of more compressed output representation or faster running time
- Extensive experimental results showing the speedup and scalability of our approach that is able to handle billion-scale datasets on a single machine.

2 CORRECTION SET BASED GRAPH SUMMARIZATION

Here we describe the framework of correction set based graph summarization (CGS). In this framework, we are given a simple undirected input graph $G = (V, E)$, and the output consists of a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and corrections sets C^+ and C^- which contain edges to be inserted and deleted respectively. The goal is to reconstruct G using the summary graph and correction sets. We denote the reconstructed graph by $\hat{G} = (V, \hat{E})$. If $\hat{G} = G$, we call the summarization lossless; otherwise, the summarization is lossy. We denote the set of neighbours of node v in G ($\hat{G}$) by N_v ($\hat{N}_v$).

Problem Definition. We begin by formally describing the method used to obtain the reconstructed graph $\hat{G}$ from the output of CGS, namely the summary graph and the correction set. Given a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-, we build $\hat{G} = (V, \hat{E})$ as follows:

(1) For each superedge $(A, B) \in \mathcal{P}$, add all pairs of nodes (a, b) as edges to $\hat{E}$ where $a \in A$ and $b \in B$.
(2) Add each edge in C^+ to $\hat{E}$
(3) Remove each edge in C^- from $\hat{E}$

The *graph summarization problem* (an optimization problem) that CGS algorithms aim to solve is defined below.

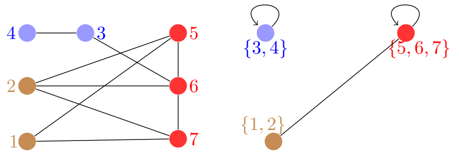
Graph Summarization Problem
Input: Graph $G = (V, E)$
Output: Summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-
Minimizing

$$|\mathcal{P}| + |C^+| + |C^-| \tag{1}$$

Such that the restored graph $\hat{G} = (V, \hat{E})$ satisfies the constraint

$$|N_v \setminus \hat{N}_v| + |\hat{N}_v \setminus N_v| \leq \epsilon |N_v|, \ \forall v \in V \tag{2}$$

The objective function (Eq. (1)), which we want to minimize, is the sum of the number of superedges in the summary graph and the number of edges in the correction sets. In our implementation, we only count the non-loop superedges since self loops can be encoded using a single bit, so their total size is negligible. Note that the constraint in Eq. (2) applies only to lossy summarization which is orthogonal to the contributions in this work. Figure 1 shows graph summarization using correction sets. It losslessly summarizes the input graph with 7 nodes and 9 edges to the summary with 3 supernodes, 3 superedges and 3 correction edges (1 insertion and 2 deletions). SWeG [32] is the current state of the art of CGS for large-



$C^+ = \{(3, 6)\}, C^- = \{(5, 7), (1, 6)\}$

Figure 1: The scheme of correction set based graph summarization

scale graphs that builds on the algorithm by Navlahka et al. [28] and provides approximate solutions to the graph summarization problem. SWeG contains several optimization techniques over [28], namely an efficient approximation metric for determining merge pairs and a dividing step to speed up the algorithm and parallelize the code for a distributed computing environment. The goal of our work is to introduce new techniques to summarize large scale graphs faster than SWeG by significantly reducing the amount of computation, while maintaining comparable output compactness.

Algorithmic Framework. To better describe our algorithm, we will first outline the CGS approach of [28, 32].

At a high level, the input to the algorithm is a graph $G = (V, E)$, number of iterations T, and an error bound ϵ; the output is a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-. Firstly, the set of supernodes $\mathcal{S}$ is initialized such that every supernode contains exactly one vertex of V and every vertex of V is in exactly one supernode. Then, $\mathcal{S}$ is repeatedly updated over T iterations by performing sequences of supernode merges per iteration. In the case of SWeG, in each iteration, $\mathcal{S}$ is divided into disjoint groups prior to the merging step and merges are performed within each group.

Once the supernodes are identified, the original edges E are encoded into superedges $\mathcal{P}$ and correction sets C^+ and C^-. Finally, in the case of lossy summarization (if $\epsilon > 0$), some superedges/edges are dropped from $\mathcal{P}$, C^+, and C^- while maintaining the constraint in Eq. (2). The individual steps in the algorithm will now be described in further detail.

Dividing step. The dividing step divides $\mathcal{S}$ into disjoint groups of supernodes such that supernodes which are similarly connected are placed into the same group. This step is an optimization technique introduced by [32] for the purpose of speed, memory efficiency, and parallelizability. The division is performed by grouping supernodes based on their *shingle*. The *shingle* $f(v)$ of a regular node $v \in V$ is defined as $f(v) := min_{u \in N_v \text{ or } u=v}\, h(u)$ where h is a random bijective function $h : V \rightarrow \{1, \ldots, |V|\}$. The shingle $F(A)$ of a supernode $A \in \mathcal{S}$ is defined as as $F(A) := min_{v \in A} f(v)$. $\mathcal{S}$ is then divided into disjoint groups $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(m)}\}$ where supernodes in each group have the same shingle value.

Merging supernodes. Let $\mathcal{S}$ be the entire set of supernodes. In the case of [32], set $\mathcal{S}$ to be each $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(m)}\}$ from the dividing step and perform the following for each group. (In a parallel implementation, each group is processed in parallel).

The merging step merges supernodes in $\mathcal{S}$ by selecting a random node A from $\mathcal{S}$, determining A's best merge candidate supernode B (in terms of minimizing Eq. (1)) from $\mathcal{S}$, then merging A and B if the result of the merge reduces Eq. (1) by a sufficient amount. Formally, this is done as follows:

- Initialize temporary set *temp* to $\mathcal{S}$

- While *temp* is not empty:
  - Randomly remove a supernode A from *temp*
  - Find the best merge candidate B for A from *temp*
  - If the *savings* (defined below) obtained from merging A and B is above some threshold, merge A and B then replace B in *temp* with the merged result

To define *savings*, we need the notion of *cost* described as follows. The cost of each supernode $A \in \mathcal{S}$ is denoted by $Cost(A, \mathcal{S})$ and is defined to be the number of superedges in $\mathcal{P}$ and edges in C^+ and C^- that A contributes to Eq. (1). This is computed by performing a temporary edge encoding step (described later) relative to A on the current state of $\mathcal{S}$. In particular, to compute $Cost(A, \mathcal{S})$ for some supernode A, we would calculate the number of edges between A and every adjacent supernode, then use this to calculate how many superedges and correction set edges would be encoded. Similarly, $Cost(A \cup B)$ is computed by looking at edges between merged supernode $A \cup B$ and all adjacent supernodes.

The *savings* obtained by merging two supernodes $A \neq B \in \mathcal{S}$, denoted $Saving(A, B, \mathcal{S})$, describes the "benefit" of merging A and B by calculating the inverse of the ratio between the cost of merged supernode $A \cup B$ and the sum of A and B's separate cost. Savings is formally defined as

$$Saving(A, B, \mathcal{S}) := 1 - \frac{Cost(A \cup B, (\mathcal{S} \setminus \{A, B\}) \cup \{A \cup B\})}{Cost(A, \mathcal{S}) + Cost(B, \mathcal{S})}$$

[32] claims that computing *Saving* is computationally expensive, and uses an approximation metric known as *SuperJaccard* similarity to approximate *Saving*. The *SuperJaccard* similarity between two supernodes is defined as

$$SuperJaccard(A, B) = \frac{\sum_{v \in N_A \cup N_B} min(w(A, v), w(B, v))}{\sum_{v \in N_A \cup N_B} max(w(A, v), w(B, v))} \quad (3)$$

where N_A is the set of nodes adjacent to the nodes inside supernode $A \in \mathcal{S}$ and $w(A, v)$ is the number of nodes in supernode $A \in \mathcal{S}$ adjacent to node $v \in V$, formally defined as $w(A, v) := |\{u \in A : \{u, v\} \in E\}|$. *SuperJaccard* aims to measure the similarity between two supernodes based on the similarity of their connectivity and is used to approximate the best merge candidate. After the best merge candidate B is identified using *SuperJaccard*, then $Saving(A, B, \mathcal{S})$ is computed once to decide whether or not to merge. Formally, if $Saving(A, B, \mathcal{S}) \geq$ *merging threshold* $\theta(t)$, for iteration $1 \leq t < T$, then A and B are merged. Here, $\theta(t)$ is defined as $1/(1 + t)$ so that more merge opportunity is allowed in the later iterations.

Encoding edges. The encoding step takes the supernodes $\mathcal{S}$ from the merging step and encodes the edges E of the original graph into superedges $\mathcal{P}$ and corrections C^+, C^-. This is done by iterating over all pairs of supernodes (A, B) where the set of edges in E between the nodes in A and nodes in B is not empty. Then we either (1) choose to not encode a superedge between A and B or (2) choose to encode a superedge between A and B. In case (1), since we do not introduce a superedge, we would lose all the edges between A and B in the reconstruction step, so we must add all these edges to C^+. In case (2), we add a superedge (A, B) to $\mathcal{P}$ and as a result we could potentially introduce extraneous edges that were not in the original graph; we add those edges to C^-. For each supernode pair $A, B \in \mathcal{S}$, E_{AB} is the set of edges in E that are between the nodes inside supernodes A and B, and F_{AB} is the set edges between all pairs of nodes in A and B. Formally,

$$E_{AB} := \{\{u, v\} \subset V : u \in A, v \in B, u, v \in E\}$$
$$F_{AB} := \{\{u, v\} \subset V : u \in A, v \in B\}$$

For each pair of supernodes $A, B \in \mathcal{S}$ where $E_{AB} \neq \emptyset$, if $|E_{AB}| \leq \frac{|F_{AB}|}{2} = \frac{|A| \cdot |B|}{2}$ then we do not encode E_{AB} as a superedge, so E_{AB} is merged into C^+. Otherwise, E_{AB} is encoded as a superedge, so edge (A, B) is added to superedges $\mathcal{P}$ and $(F_{AB} \setminus E_{AB})$ is merged into C^-. In the special case where $A = B$, the condition is such that we do not encode a superedge (superloop) if $|E_{AA}| \leq \frac{|F_{AA}|}{2} = \frac{|A| \cdot (|A|-1)}{4}$. Otherwise, we do encode a superloop.

Dropping edges. In [28, 32] there is also an optional post processing step (when $\epsilon > 0$), called the *dropping step*, where some edges are removed from the summary graph and correction sets so that the summary is more compact. The dropping step ensures that Eq. (2) is satisfied by verifying the constraint is still met as each edge is removed. We do not discuss this step further as its running time is negligible and it is orthogonal to the main approach.

3 PROPOSED METHOD

Since the correction set based summarization method of [32] is the current state of the art, we will discuss our proposed methods of improvement with respect to SWeG. Our goal is to optimize the steps of SWeG by reducing the amount of computation performed.

The merging step is, computationally, the most challenging stage of the algorithm and takes a significant fraction of the total running time, making it the overall bottleneck step for most graphs. This is because the process is quadratic in the size of groups. Here we propose an improved dividing step that reduces the size of groups, thus significantly speeding up the merge phase. Additionally, we propose an efficient algorithm to compute Saving directly during merge eliminating the need to use an approximation metric. For some graphs, the encoding step becomes a problematic step since it iterates through all pairs of supernodes, inducing a large amount of overhead required for computing/remembering the edges between pairs of supernodes. For graphs with a high number of supernodes, the overhead causes the encoding step to run much slower and in some cases not complete within reasonable time. We present a new sort-based encoding step which has an improved practical running time and is more robust than the one in [32]. We show the overall structure of our approach in Algorithm 1. It consists of the divide, merge and encode steps as outlined in Section 2.

Algorithm 1: Algorithm Overview

```
Input: input graph $G = (V, E)$, number of iterations $T$
Output: summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$, corrections $C^+$ and $C^-$
1: initialize supernodes $\mathcal{S}$ to each vertex in $V$
2: for $t = 1...T$ do
3:     compute weighted LSH signature of each supernode in $\mathcal{S}$
4:     divide $\mathcal{S}$ into disjoint groups based on their signature
5:     perform merges in each disjoint groups
6: encode edges $E$ into superedges $\mathcal{P}$ and correction edges $C^+$ and $C^-$
7: return summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and corrections $C^+, C^-$
```

Speeding up the merging step. In the merging step of [32], after picking a random supernode A from the group, the merge partner B for A is determined by checking every other supernode in the group and selecting the best candidate. Overall, the merging step takes time quadratic in the number of supernodes in a group. Conceptually, we can improve the running time of the merging step by reducing the size of each group $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$. We achieve this by introducing a refined divide step which uses *SuperJaccard* similarity for grouping supernodes. Note that SWeG uses *SuperJaccard* as an approximation of Saving in the merge phase. We instead propose to use *SuperJaccard* in the divide step. Namely, we want to create groups such that nodes with high *SuperJaccard* similarity end up in the same group. However, we will aim to avoid computing *SuperJaccard* for every pair of supernodes as this would be prohibitive. Instead we want to devise a locality sensitive hashing scheme for *SuperJaccard* and then hash the supernodes to the proper groups.

Locality sensitive hashing (LSH) is a technique used to group similar items. A hash function (or hash function family) is used to assign items to buckets, and the items in each bucket are "similar" to each other with high probability. Typically, items are considered to be sets and LSH schemes are designed for well-known set similarity measures, such as simple Jaccard, Hamming, and Cosine similarity. In the following, we show that *SuperJaccard* can be casted as weighted Jaccard similarity, for which there exist locality sensitive hash functions.

The weighted Jaccard similarity $J_w(X, Y)$ between two vectors X and Y of equal length and with integers weights is defined is

$$J_w(X, Y) := \frac{\sum_v min(X_v, Y_v)}{\sum_v max(X_v, Y_v)}$$

Note that when X and Y are Boolean vectors, the above equation gives the simple Jaccard similarity between two sets represented by vectors X and Y.

Weighted LSH based dividing step. We assign a "supervector" V_S of size n to each supernode $S \in \mathcal{S}$, where n is the number of nodes in V. Each index u in V_S (where $1 \le u \le n$) represents a node in V, and the weight of V_S at index u for each node u is $w(S, u)$. Recall, from Section 2, that $w(S, u)$ is the number of nodes in S adjacent to u.

We claim that for two supernodes A and B, the weighted Jaccard similarity between their supervectors V_A and V_B is equal to the SuperJaccard similarity between A and B. Namely, we note that

$$J_w(V_A, V_B) = \frac{\sum_v min((V_A)_v, (V_B)_v)}{\sum_v max((V_A)_v, (V_B)_v)} = \frac{\sum_v min(w(A, v), w(B, v))}{\sum_v max(w(A, v), w(B, v))}$$

is equal to $SuperJaccard(A, B)$ as defined in Eq. (3). For any supernode S, the non-zero values in its assigned supervector V_S correspond exactly to the nodes in N_S. So, in $J_w(V_A, V_B)$, only the indices $v \in N_A \cup N_B$ contribute to the overall value. Therefore, $J_w(V_A, V_B) = SuperJaccard(A, B)$.

Now, we can use a weighted LSH scheme (to be described shortly) in order to split $\mathcal{S}$ in the dividing step and have the property that supernodes with high SuperJaccard similarity have a high probability of being in the same bucket. Compared to the simple single-shingle based approach of [32], weighted LSH is a more precise metric that divides S into more groups of smaller size while ensuring that supernodes with similar connectivity are in the same group. Furthermore, we can control the performance of our algorithm by tuning the precision level of the hash function in order to trade compression for running time.

Updated Dividing Step. As a weighted LSH scheme, we use Densified One Permutation Hashing (DOPH) [35], which takes as input a binary vector I of length $|V|$ and uses a single permutation $h : \{1, \ldots, |V|\} \to \{1, \ldots, |V|\}$ to produce a hash signature H_I for I of length k. We use the fact from [34] that for any sparse weighted vectors V_A and V_B, the probability that the binarized forms of V_A and V_B have the same DOPH signature is approximately the weighted Jaccard similarity between V_A and V_B (non binarized).

Given a binary vector I of length $|V|$, a random permutation $h : \{1, \ldots, |V|\} \to \{1, \ldots, |V|\}$, a hash signature length k, and a random binary vector D of length k (where D_i, $1 \le i \le k$, is set to 0 or 1 independently and uniformly at random), DOPH signature H_I is computed as follows (see Algorithm 2):

- Permute the bits in I by re-indexing based on h. (Line 1)
- Separate I into k equal length bins. We denote bin i as b_i, $1 \le i \le k$. (Line 2)
- Define H_{b_i} as the first index within the bin b_i that contains a non-zero value. If the bin contains no non-zero values, H_{b_i} is "empty". Set H_{b_i} to be the value of the signature H_I at position i. (Lines 3 - 7)
- For each empty H_{b_i}, we define the value to be the first non-empty signature index either to the left or right (with wraparound at the endpoints). The choice of left or right is determined by the bit D_i. (Lines 8 - 12)
- Return $H_I = \{H_{b_1}, ..., H_{b_k}\}$. (Line 13)

We use DOPH as a new metric for dividing supernodes $\mathcal{S}$ into disjoint groups, where each group of supernodes has the same DOPH signature. For each supernode $A \in \mathcal{S}$, we binarize the supervector V_A (supervector defined in the previous section) by converting each non-zero entry to 1 and compute A's hash signature H_A. We then divide $\mathcal{S}$ into groups by hash signature value. The algorithm is illustrated in Algorithm 3.

Algorithm 2: Densified One Permutation Hashing (DOPH)

```
Input: Binary vector $I$, random permutation
  $h : \{1, \ldots, |V|\} \to \{1, ..., |V|\}$, hash signature length $k$, random
  binary vector $D$ of length $k$
Output: Hash signature $H$
1:  permute the values of $I$ using $h$
2:  divide $V$ into $k$ sequential bins of equal size (right pad $V$ with zeroes if
    $k$ does not divide $|V|$)
3:  for each bin $b_i$, $1 \le i \le k$ do
4:      if $b_i$ has a non-zero entry (i.e. $b_i$ is non-empty) then
5:          $H_{b_i} \leftarrow$ index of first non-zero entry in $b_i$, $1 \le H_{b_i} \le |b_i|$
6:      else
7:          $H_{b_i} \leftarrow \emptyset$
8:  for each $H_{b_i}$ where $H_{b_i} = \emptyset$, $1 \le i \le k$
9:      if $D_i = 1$ then
10:         $H_{b_i} \leftarrow H_{b_j}$, $j$ is index of first non-empty bin to the right
11:     else
12:         $H_{b_i} \leftarrow H_{b_j}$, $j$ is index of first non-empty bin to the left
13: return $H = \{H_{b_1}, ..., H_{b_k}\}$
```

Algorithm 3: Weighted LSH Divide

```
Input: Graph $G = (V, E)$, current supernodes $\mathcal{S}$, signature length $k$
Output: Disjoint groups of supernodes: $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$
1: generate a random permutation $h : \{1, ..., |V|\} \to \{1, ..., |V|\}$
2: generate random binary vector $D$ of length $k$
3: for each supernode $A \in \mathcal{S}$ do
4:     compute supervector $V_A$ (as previously defined)
5:     $I_A \leftarrow binarize(V_A)$
6:     $DOPH(I_A, h, k, D)$  ▷ compute hash signature $H_A$ (Algorithm 2)
7: divide supernodes in $\mathcal{S}$ into $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ by their hash signature
8: return $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$
```

Tuning the performance. Using DOPH to divide $\mathcal{S}$ allows us to tune the performance of the overall algorithm. In particular, we can obtain a more compact output with more running time or obtain a less compact output with less running time. Tuning is done through modifying the k value in DOPH which specifies the number of bins that we divide the input vector V into and the length of the hash signature. Increasing k means that vectors must be more similar to have the same signature, which results in $\mathcal{S}$ being divided into more groups of smaller size with higher weighted Jaccard similarity. Conversely, reducing k results in less groups but of larger size. As k increases, the running time of the merge algorithm significantly decreases since the merge algorithm receives groups of smaller size. Amount of compression also decreases due to the probabilistic nature of LSH since groups of smaller size result in a higher likelihood of good potential merge pairs being placed in different groups. Being able to tune k allows the option of trading compression for running time (depending on which is more important for a particular application). The number of groups increasing with k can also be explained combinatorially by analyzing the number of possible signatures for any given k, which we observe is $(\frac{n}{k} + 1)^k$. This is because there are $\frac{n}{k} + 1$ possible values per bin (namely $\frac{n}{k}$ indices and the *empty* value) and k total bins in a DOPH signature. Hence, the number of possible groups grows exponentially with k, compared to the shingle method in which the number of possible groups is fixed at n.

Efficiently computing Saving. As previously discussed, the merging step in [32] computes the SuperJaccard between supernodes to find the best merge partner for a particular supernode. SuperJaccard is an approximation for Saving, where Saving is the true amount of decrease in Eq. (1) when merging two supernodes. Here we present a method to directly compute the Saving between supernodes in running time no more than computing SuperJaccard.

For each group $\mathcal{S}^{(i)}$ of supernodes, we create a hashtable-of-hashtables data structure W. The first level hashtable of W is keyed by the supernodes in the group $\mathcal{S}^{(i)}$. A second level hashtable, denoted by W_A for some supernode A in $\mathcal{S}^{(i)}$, contains key-value pairs (B, val) where the key B is a supernode in $\mathcal{S}$ such that there exist edges in E between the nodes within A and B and the value val is the number of edges in E between A and B. This data structure enables us to find out the number of actual edges between any pair of supernodes in expected constant time and consequently compute its Cost and Saving with other supernodes. Algorithm 4 shows how to calculate Saving for a pair of supernodes A and B. The algorithm works along the lines of the decision process inside the encoding step described in Section 2. For this, it needs the numbers of edges between supernode pairs, which are conveniently retrieved from the hashtable structure we described (see lines 4,5,8,9,11).

Algorithm 4: Compute Saving

```
Input: Hashtables $W_A$ and $W_B$
Output: $Saving(A, B, \mathcal{S})$
1: initialize $Cost(A) = 0, Cost(B) = 0, Cost(A \cup B) = 0$
2: for each supernode $C \in$ keyset of $W_A$ do
3:     if keyset of $W_B$ does not contain $C$ then
4:         $Cost(A) \mathrel{+}= \min\left(\frac{|A| \cdot (|C|-1)}{2}, W_A[C]\right)$
5:         $Cost(A \cup B) \mathrel{+}= \min\left(\frac{(|A|+|B|) \cdot (|C|-1)}{2}, W_A[C]\right)$
6: for each supernode $C \in$ keyset of $W_B$ do
7:     if keyset of $W_A$ does not contain $C$ then
8:         $Cost(B) \mathrel{+}= \min\left(\frac{|B| \cdot (|C|-1)}{2}, W_{BC}\right)$
9:         $Cost(A \cup B) \mathrel{+}= \min\left(\frac{(|A|+|B|) \cdot (|C|-1)}{2}, W_B[C]\right)$
10:    else
11:        $Cost(A \cup B) \mathrel{+}= \min\left(\frac{(|A|+|B|) \cdot (|C|-1)}{2}, W_B[C] + W_A[C]\right)$
12: return $1 - \frac{Cost(A \cup B)}{Cost(A) + Cost(B)}$
```

After merging any two supernodes A and B, we update W by iterating over the keyset of the hashtable corresponding to the supernode with smaller size (say B). For each key-value pair $(C, W_B[C])$ in W_B (i.e. C shares an edge in E with B) we do the following steps:

(1) If key C exists in W_A (i.e. C shares an edge in E with A), we set $W_A[C] = W_A[C] + W_B[C]$ (i.e we add the number of edges between B and C to the count of edges between A and C). Otherwise, we add a new pair $(C, W_B[C])$ to W_A.
(2) If W_C is in W (i.e. C belongs to the same group as A and B) and key A exists in W_C, we set $W_C[A] = W_C[A] + W_C[B]$ and remove B from W_C. If key A does not exist in W_C, we create a new entry for A, set $W_C[A] = W_C[B]$, and remove B from W_C.

Finally, we remove B from W since the supernode B no longer exists after the merge. We note that computing Saving using Algorithm 4 requires only iterating over supernodes in $\mathcal{S}$ (in contrast to computing SuperJaccard which requires iterating over nodes in V). This helps us to speed up the merge step while eliminating the need of approximating the Saving computation.

Improving the encoding step. The encoding algorithm of [32] requires iterating over all supernodes and for each supernode A, identifying all supernodes $B \in \mathcal{S}$ for which there is at least one edge in E between the nodes within A and B (i.e. $E_{AB} \neq \emptyset$). Implementing this step requires a significant amount of computational overhead especially for summary graphs with many supernodes. In a simple implementation, all pairs of supernodes (A, B) would be checked and if $E_{AB} \neq \emptyset$, we would perform the rest of the encoding algorithm. However, this implementation would take time that is quadratic in the number of supernodes leading to poor scalability. In a more careful implementation, for each supernode A, we only iterate over the supernodes B where $E_{AB} \neq \emptyset$. To do this, we require a preprocessing step where we iterate over the nodes in V within A, compute the edges incident to these nodes to obtain the supernodes B that shares an edge in E with A, and save these edges in a lookup table for the subsequent steps. However, there is a significant overhead due to computing, storing, and looking up incident edges for each supernode.

Nevertheless, even the more careful implementation above performs poorly when the number of supernodes is large and in some cases caused the algorithm to not complete within reasonable time. Here we will introduce a restructured encoding step algorithm that has faster practical performance, is more consistent, and requires little computational overhead aside from reading the edges in E.

Updated Encoding Step. In our encoding algorithm (Algorithm 5), for each edge $e \in E$, we add a 2-tuple (s, e) to a list L. In each 2-tuple, $e = (u, v)$ corresponds to the original edge and $s = (A, B)$ is a "candidate superedge" where A and B are the supernodes containing nodes u and v respectively. The candidate superedge identifies which pair of supernodes an edge is between. We then group L into $\{L^{(1)}, ..., L^{(k)}\}$ by their candidate superedge value s so that any supernode pair (A, B) where $E_{AB} \neq \emptyset$, which have edges to be encoded, have the edges between them grouped together (line 5 in Algorithm 5). Each group $L^{(i)} \in \{L^{(1)}, ..., L^{(k)}\}$ is associated with a pair of supernodes (A, B) and contains exactly the edges between A and B (namely E_{AB}) that are needed for the encoding step. Thus, for each group, we can look at the respective E_{AB} to decide whether or not to encode a superedge using the same conditions as in [28, 32]. Line 5 in Algorithm 5 can be done efficiently by lexicographically sorting the 2-tuples in L by their candidate superedge value s. Effectively, this groups all the edges in E by their supernode endpoints. Iterating through each group (line 6) can be done by linear scanning L in sorted order and a group change is detected by a change in the candidate superedge s of a 2-tuple during the scan. The edges between the pair of supernodes of a group are obtained by temporarily saving each edge e in the 2-tuple scan until the group changes, at which point the remainder of the encoding step can be performed on the saved edges.

Algorithm 5: Updated Encoding Step

```
Input: Input graph $G = (V, E)$, supernodes $\mathcal{S}$
Output: Summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$, corrections $C^+$ and $C^-$
1:  $L = \{\}$
2:  for each edge $e = (u, v)$ in $E$ do
3:      $A \leftarrow$ supernode containing $u$, $B \leftarrow$ supernode containing $v$
4:      $s \leftarrow$ candidate superedge $(A, B)$, $L \leftarrow L \cup \{(s, e)\}$
5:  group $L$ into $\{L^{(1)}, ..., L^{(k)}\}$ by candidate superedge $s = (A, B)$
6:  for each $L^{(i)} \in \{L^{(1)}, ..., L^{(k)}\}$ do
7:      $(A, B) \leftarrow$ candidate superedge $s$ of $L^{(i)}$
8:      $E_{AB} \leftarrow$ set of edges $e$ in $L^{(i)}$ (denoted $E_{AA}$ if $A = B$)
9:      if $A \neq B$ then
10:         if $|E_{AB}| \leq \frac{|A| \cdot |B|}{2}$ then $C^+ \leftarrow C^+ \cup E_{AB}$
11:         else $\mathcal{P} \leftarrow \mathcal{P} \cup \{(A, B)\}$, $C^- \leftarrow C^- \cup (F_{AB} \setminus E_{AB})$
12:     else
13:         if $E_{AA} \leq \frac{|A| \cdot (|A|-1)}{4}$ then $C^+ \leftarrow C^+ \cup E_{AA}$
14:         else $\mathcal{P} \leftarrow \mathcal{P} \cup \{(A, A)\}$; $C^- \leftarrow C^- \cup (F_{AA} \setminus E_{AA})$
15: return $\overline{G} = (\mathcal{S}, \mathcal{P})$ and $C^+, C^-$
```

In summary, we significantly reduce the running time of the encoding algorithm of [28, 32] in practice by directly working with the edges in E. This reduces the overhead that comes from iterating over pairs of supernodes and computing the associated edges.

Time Complexity. The time complexity of LDME is dominated by the merge phase, which is $O((n/|\mathcal{S}^*|) \cdot |\mathcal{S}^*|^2) = O(n \cdot |\mathcal{S}^*|)$, where $\mathcal{S}^*$ denotes the largest group in $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ from the divide phase. Note that this is similar to the time complexity of the merge phase in SWeG, however the largest group size in LDME is much smaller than in SWeG, thus resulting in a significant improvement in running time in practice.

Space Complexity. The space requirement for storing $G, \overline{G}, C^+$, and C^- is $O(|V| + |E|) = O(|E|)$. The hashtable-of-hashtables W, created for each group, stores the number of edges between supernodes in the group and their adjacent supernodes, which in the worst case is $O(|E|)$. However, LDME's divide phase creates small groups, especially as k increases, thus the space requirement will be much less in practice. The other data structures used, such as the signatures and the encoding edge list, are all $O(|E|)$.

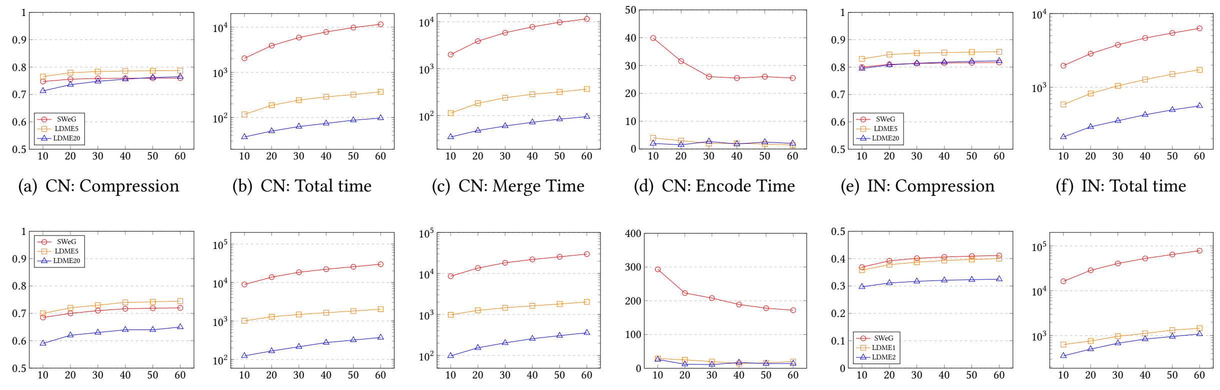
Parallel Implementation Description. Similar to SWeG, LDME is highly parallelizeable and can run in a distributed environment for higher speed and scalability. In the dividing step, the DOPH signature of each supernode can be computed in parallel (lines 5 and 6 of Algorithm 3). Then, the merging step can be performed on each group in $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ in parallel (line 5 in Algorithm 1). Finally, each supernode A in the encoding step can be processed in parallel, so line 2 of Algorithm 5 only reads in the edges incident to each A. Processing each supernode in parallel also removes the need for grouping candidate superedges via sorting in line 5.

4 EXPERIMENTS

In our experiments, we compare our approach to SWeG [32], Mosso [14], and VoG [15] (we do not compare versus [28] because it is superseded by SWeG). In our experiments, we wish to test how our algorithmic improvements translate to gains in running time compared to the mentioned algorithms. We evaluate the implementations of all above algorithms on a single-threaded machine where we can better observe our main goal of reducing the amount of computation. We also compare the performance of the parallel implementation of LDME and SWeG in a distributed setting.

We experiment with two versions of our approach, one where we use DOPH signature length $k = 5$ in our dividing step and the other where $k = 20$. We call our general approach LSH-based Divide-Merge-Encode (LDME), and the two versions LDME5 and LDME20 for $k = 5$ and $k = 20$ respectively. Both versions demonstrate a significant speedup over the compared algorithms (often an order of magnitude), with LDME20 faster than LDME5. With respect to compression, LDME5 is very similar to SWeG and Mosso, whereas LDME20 shows some moderate reduction in compression as trade-off for its speedier execution over LDME5.

We use the datasets in Table 1 from Laboratory of Web Algorithmics (http://law.di.unimi.it/datasets.php). The sizes of the graphs we use are as follows: cnr-2000 is a (relatively) small graph, in-2004, eu-2005, and hollywood-2009 are medium graphs, and hollywood-2011, indochina-2004, uk-2002, and arabic-2005 are large graphs. Table 1 shows the characteristics of each graph. We note that the



(a) CN: Compression (b) CN: Total time (c) CN: Merge Time (d) CN: Encode Time (e) IN: Compression (f) IN: Total time

(g) EU: Compression (h) EU: Total time (i) EU: Merge Time (j) EU: Encode Time (k) H1: Compression (l) H1: Total time

Figure 2: The comparison between original SWeG, LDME5 and LDME20 in terms of compression, total time (seconds), divide/merge time (seconds), encode time (seconds) over 60 iterations.

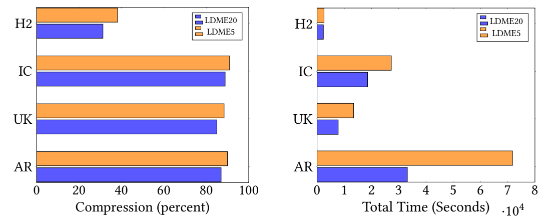


Figure 3: Compression and total time at final iteration 60 of LDME5/20 on graphs that SWeG could not complete within 1 day.

Graph	Abbr	Nodes	Edges
cnr-2000	CN	325,557	5,565,380
in-2004	IN	1,382,908	27,560,356
eu-2005	EU	862,664	32,778,363
hollywood-2009	H1	1,139,905	113,891,327
hollywood-2011	H2	2,180,759	228,985,632
indochina-2004	IC	7,414,866	304,472,122
uk-2002	UK	18,520,486	529,444,615
arabic-2005	AR	22,744,080	1,116,651,935

Table 1: Summary of datasets

number of edges shown in Table 1 is after symmetrization, where we add the reverse of directed edges if they do not already exist.

LDME v.s SWeG. We evaluate the difference between SWeG and LDME5/LDME20 using four different metrics: (1) compression (2) total running time, (3) dividing and merging time, and (4) encoding time. For each metric, we ran SWeG and LDME5/LDME20 for $T = 10, 20, 30, 40, 50, 60$ iterations. The amount of compression was computed using the complement of the ratio between the number of superedges + correction set edges and number of original edges. Specifically: $Compression = 1 - \frac{|\mathcal{P}|+|C^+|+|C^-|}{|E|}$

Figure 2 illustrates the compression and total running time of SWeG and LDME5/LDME20 on graphs for which all algorithms could complete within a reasonable time of 1 day (CN, IN, EU, and H1). In terms of compression, LDME5 demonstrates a similar final compression as SWeG; CN, IN, and EU achieved 2% to 4% increase at iteration 60 and H1 had a 1% decrease. LDME20 demonstrates a final compression that matches or is only slightly lower than SWeG; CN and IN achieved 0.5% increase at iteration 60 while EU and H1 had a 7% and 8% decrease respectively. In terms of total running time, LDME5 demonstrates a 3.6x to 31x speedup over SWeG on CN, IN and EU, and is 53x faster on H1. LDME20 shows an even more significant speedup ranging from 11x (IN) to 96x (CN) faster than SWeG. Thus, the expected effect of increasing k from 5 to 20 in LDME is clearly demonstrated in both compression and running time.

Figure 2 also shows the comparison of SWeG and LDME's divide/merge and encode times on CN and EU (we do not show this for IN and H1 since the behavior is similar). Since the merging step dominates the total running time of the algorithms, the divide/merge time and total running time are similar. The breakdown of encoding time illustrates the difference between our encoding algorithm and SWeG's encoding algorithm. LDME's encoding time is rather uniform through all iterations, while SWeG's starts high and decreases over iterations as the number of supernodes is compressed (since it scales based on $|\mathcal{S}|$). On CN, IN, EU, and H1, LDME's encoding time is 7x to 26x faster than SWeG, and for larger datasets (eg. UK, AR), SWeG's encoding could not complete within reasonable time while our encoding algorithm stayed consistently small.

To illustrate the superior scalability of LDME over SWeG, Figure 3 shows the final compression and running time of LDME5/20 on larger graphs H2, IC, UK, and AR, on all of which SWeG could not complete within reasonable time (1 day). SWeG ran overtime on these graphs due to its slow merging step and in some cases also due to its inefficient encoding step. Similar to the results in Figure 2, LDME20's compression is slightly lower than that of LDME5, but achieves a faster running time. AR, having over 1 billion edges, also shows that our algorithm can successfully summarize billion edge scale graphs using only a single machine.

Results of tuning k. Figure 4 illustrates the number of groups created in the dividing step and the size of the largest group for k = {5, 10, 15, 20} on graphs CN, H1, and H2. As k increases, there is a clear significant increase and decrease in number of groups and max group size respectively.

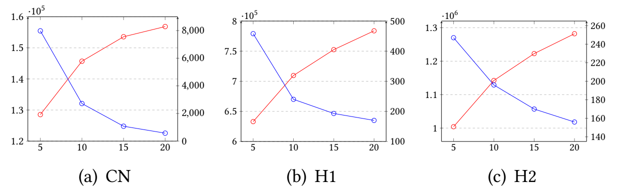


Figure 4: The number of groups (red, left y-axis) and the largest group size (blue, right y-axis) for increasing values of k.

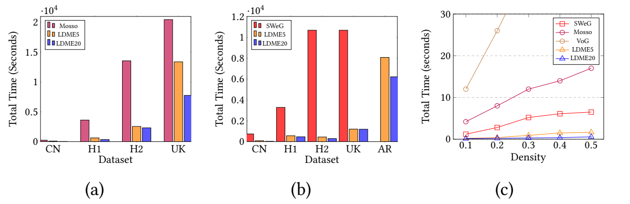


Figure 5: Running time of (a) LDME vs. Mosso on a single machine, (b) LDME vs. SWeG in a distributed environment, and (c) SBM experiments for LDME, SWeG, Mosso, and VoG on a single machine

LDME vs. Mosso and VoG. Figure 5 (a) shows the running time for Mosso and LDME5/20 (for 10 iterations) on CN, H1, H2, and UK. VoG was also run, but it was over 40x slower than LDME on all datasets, hence its running time is not displayed. We use the configuration ($e = 0.3, c = 120$) for MoSSo as used in [14], where e is the escape probability and c is the sample size of each trial. LDME5 and LDME20 achieved a 1.5x to 5.7x and 2.6x to 10.2x speedup over Mosso respectively. Mosso could also not complete AR within reasonable time (1 day) while both LDME5/20 could (see Figure 3).

Distributed Environment Experiments: Figure 5 (b) illustrates the running time of the parallel implementations of SWeG and LDME5/20, both run for 10 iterations, in a distributed environment. The implementation of both algorithms were done using Apache Spark and they were run on Amazon EMR clusters with the following specifications: 8 m5xlarge (4 vCore, 16GB memory, 64GB EBS storage) instances for CN/H1/H2 and 8 m5.2xlarge (8 vCore, 32GB memory, 128GB EBS Storage) instances for UK/AR. LDME's significant improvement in running time also translates to a distributed setting, where LDME5 achieved 3.0x to 23.8x speedup and LDME20 achieved 3.1x to 36.0x speedup on the experimented datasets. LDME also achieves higher scalability, as illustrated by SWeG being unable to complete AR within reasonable time (12 hours).

Stochastic Block Model Experiments: Stochastic block model requires two parameters for generating random graphs, the number of nodes in each community and the block matrix which shows the level of interaction between communities. We generate different random graphs with 3 communities, 300 nodes in each community and 900 nodes in total. We gradually increase the level of interactions between/within communities to generate more dense random graphs. We compare the running time of LDME5/20 with MoSSo, SWeG and VoG. Figure 5 (c) shows the performance of each algorithm. VoG goes off the figure and MoSSo running time increases substantially as the density of graph increases. SWeG and LDME5/20 are quite resilient with respect to density and LDME5 is up to 8x faster than SWeG in some cases.

5 RELATED WORK

Graph summarization is an active area of research and studied in a variety of settings (see [11, 25] for detailed surveys). Previous work on this topic can be classified into two categories, grouping [7, 13, 17–19, 28, 29, 32, 38] and non-grouping [1, 8, 10, 20, 22, 23, 26, 27, 37, 39, 40]. Grouping based methods received more attention in the last few years and they are divided into non-correction set based methods such as [7, 8, 15, 17–19, 29, 38] and correction set based methods [13, 14, 28, 32].

Navlakha et al. [28] introduced a novel framework in which a graph is represented compactly as a summary graph along with correction sets. Their algorithm, RANDOMIZED, picks a random supernode and identifies the supernode that gives the best savings with it among possible candidate merges from supernodes that are 2-hops away. SAGS [12] uses simple locality sensitive hashing instead of Saving or SuperJaccard in the merge phase to choose the best pair among pairs that are 2-hop away. VoG [15] uses existing clustering algorithms for finding important candidate subgraphs to summarize. These works, however, are unable to achieve strong compression while maintaining scalability.

The state of the art algorithm for correction set based graph summarization is SWeG studied by Shin et al. [32] which achieves strong compression and scales an order of magnitude better than RANDOMIZED and SAGS. MoSSo is a recent incremental algorithm for summarizing dynamic graphs using correction set [14] which we also include in our comparisons. SlimGraph [3] is a programming framework, where various summarization algorithms can be plugged-in (such as SWeG), rather than an algorithmic contribution, so there is no direct avenue to compare it with our work.

6 CONCLUSION

We proposed LDME, a correction set based graph summarization method that highlights the usefulness of weighted LSH to graph compression. LDME is able handle large datasets using only a single machine and improves each step of SWeG, namely, the dividing, merging, and encoding steps. Furthermore, using weighted locality sensitive hashing in the dividing step allows for performance tuning of LDME where compression can be traded off for running time. With high compression settings, LDME achieves up to 53x times faster running time while maintaining compression rates as good as SWeG. With high speed settings, LDME achieves up to two orders of magnitude speedup while allowing a small loss in compression.

REFERENCES

[1] Ahmed, A., Shervashidze, N., Narayanamurthy, S., Josifovski, V., and Smola, A. J. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web* (2013), pp. 37–48.

[2] Apostolico, A., and Drovandi, G. Graph compression by bfs. *Algorithms 2*, 3 (2009), 1031–1044.

[3] Besta, M., Weber, S., Gianinazzi, L., Gerstenberger, R., Ivanov, A., Oltchik, Y., and Hoefler, T. Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2019), SC '19, Association for Computing Machinery.

[4] Boldi, P., and Vigna, S. The webgraph framework i: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (2004), pp. 595–602.

[5] Cook, D. J., and Holder, L. B. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research 1* (1993), 231–255.

[6] Dunne, C., and Shneiderman, B. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), pp. 3247–3256.

[7] Fan, W., Li, J., Wang, X., and Wu, Y. Query preserving graph compression. In *Proceedings of the 38th ACM SIGMOD International Conference on Management of Data* (2012), pp. 157–168.

[8] Gou, X., Zou, L., Zhao, C., and Yang, T. Fast and accurate graph stream summarization. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)* (2019), pp. 1118–1129.

[9] Hassanlou, N., Shoaran, M., and Thomo, A. Probabilistic graph summarization. In *International Conference on Web-Age Information Management* (2013), Springer, pp. 545–556.

[10] Hübler, C., Kriegel, H.-P., Borgwardt, K., and Ghahramani, Z. Metropolis algorithms for representative subgraph sampling. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)* (2008), pp. 283–292.

[11] Khan, A., Bhowmick, S. S., and Bonchi, F. Summarizing static and dynamic big graphs.

[12] Khan, K. U. Set-based approach for lossless graph summarization using locality sensitive hashing. In *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops* (2015), pp. 255–259.

[13] Khan, K. U., Nawaz, W., and Lee, Y.-K. Set-based approximate approach for lossless graph summarization. *Computing 97*, 12 (2015), 1185–1207.

[14] Ko, J., Kook, Y., and Shin, K. Incremental lossless graph summarization. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Jul 2020).

[15] Koutra, D., Kang, U., Vreeken, J., and Faloutsos, C. Vog: Summarizing and understanding large graphs. *CoRR abs/1406.3411* (2014).

[16] Koutra, D., Kang, U., Vreeken, J., and Faloutsos, C. Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal 8*, 3 (2015), 183–202.

[17] Kumar, K. A., and Efstathopoulos, P. Utility-driven graph summarization. *Proceedings of the VLDB Endowment 12*, 4 (2018), 335–347.

[18] Lee, K., Jo, H., Ko, J., Lim, S., and Shin, K. Ssumm: Sparse summarization of massive graphs. In *KDD: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (2020), pp. 144–154.

[19] LeFevre, K., and Terzi, E. Grass: Graph structure summarization. In *Proceedings of the 10th SIAM International Conference on Data Mining (SDM)* (2010), pp. 454–465.

[20] Leskovec, J., and Faloutsos, C. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2006), pp. 631–636.

[21] Li, C., Baciu, G., and Wang, Y. Modulgraph: modularity-based visualization of massive graphs. In *Proceedings of the SIGGRAPH Asia 2015 Visualization in High Performance Computing* (2015), pp. 1–4.

[22] Li, C.-T., and Lin, S.-D. Egocentric information abstraction for heterogeneous social networks. In *Proceedings of the 1st International Conference on Advances in Social Network Analysis and Mining* (2009), pp. 255–260.

[23] Liberty, E. Simple and deterministic matrix sketching. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2013), pp. 581–588.

[24] Liu, X., Tian, Y., He, Q., Lee, W.-C., and McPherson, J. Distributed graph summarization. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (2014), pp. 799–808.

[25] Liu, Y., Dighe, A., Safavi, T., and Koutra, D. A graph summarization: A survey. *CoRR abs/1612.04883* (2016).

[26] Maccioni, A., and Abadi, D. J. Scalable pattern matching over compressed graphs via dedensification. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), pp. 1755–1764.

[27] Maiya, A. S., and Berger-Wolf, T. Y. Sampling community structure. In *Proceedings of the 19th International Conference on World Wide Web* (2010), pp. 701–710.

[28] Navlakha, S., Rastogi, R., and Shrivastava, N. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2008), pp. 419–432.

[29] Riondato, M., García-Soriano, D., and Bonchi, F. Graph summarization with quality guarantees. In *Proceedings of the 14th IEEE International Conference on Data Mining (ICDM)* (2014), pp. 947–952.

[30] Rossi, R. A., and Zhou, R. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data 5*, 1 (2018), 10.

[31] Shah, N., Koutra, D., Jin, L., Zou, T., Gallagher, B., and Faloutsos, C. On summarizing large-scale dynamic graphs. *IEEE Data Eng. Bull. 40*, 3 (2017), 75–88.

[32] Shin, K., Ghoting, A., Kim, M., and Raghavan, H. Sweg: Lossless and lossy summarization of web-scale graphs. In *Proceedings of the 28th International Conference on World Wide Web* (2019), pp. 1679–1690.

[33] Shoaran, M., Thomo, A., and Weber-Jahnke, J. H. Zero-knowledge private graph summarization. In *2013 IEEE International Conference on Big Data* (2013), IEEE, pp. 597–605.

[34] Shrivastava, A. Simple and efficient weighted minwise hashing. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2016), pp. 1498–1506.

[35] Shrivastava, A., and Li, P. Improved densification of one permutation hashing. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI)* (2014), pp. 732–741.

[36] Spielman, D. A., and Srivastava, N. Graph sparsification by effective resistances. *SIAM Journal on Computing 40*, 6 (2011), 1913–1926.

[37] Tang, N., Chen, Q., and Mitra, P. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1481–1496.

[38] Tian, Y., Hankins, R. A., and Patel, J. M. Efficient aggregation for graph summarization. In *Proceedings of the 34th ACM SIGMOD International Conference on Management of Data* (2008), pp. 567–580.

[39] Yan, N., Hasani, S., Asudeh, A., and Li, C. Generating preview tables for entity graphs. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1797–1811.

[40] Zeqian Shen, Kwan-Liu Ma, and Eliassi-Rad, T. Visual analysis of large heterogeneous social networks by semantic and structural abstraction. *IEEE Transactions on Visualization and Computer Graphics 12*, 6 (2006), 1427–1439.