An Optimized Lossless Graph Summarization for Large-Scale Graphs

Meiquan Lai[†], Yaqi Huang[†], Zhidan Liu[†*], Kaishun Wu[‡]
[†]*Shenzhen University,* [‡]*Hong Kong University of Science and Technology (Guangzhou)*
Emails: {laimeiquan2020, huangyaqi2021}@email.szu.edu.cn, liuzhidan@szu.edu.cn, wuks@hkust-gz.edu.cn

***Abstract*—Graphs have been widely used for modeling large-scale data generated from real-world applications, while compact representation of such graphs is beneficial for efficient storage and effective graph analysis. As a promising solution, lossless graph summarization can compactly represent a given graph as a summary graph, which consists of supernodes (*i.e.*, sets of nodes) and superedges (edges between supernodes), and the correction edge sets, which together with summary graph can exactly reconstruct the original graph. Although many research efforts have been devoted to develop graph summarization methods, existing works are still inefficient in terms of computation efficiency and representation compactness. To address their limitations, we propose `optGS` that includes a set of optimization techniques, including computation-oriented supernode re-dividing, degree-aware approximation metric for selecting the best merge, and redundant computation avoidance, to improve current advances. Extensive experiments on a variety of large graph datasets demonstrate the computation efficiency and compression effectiveness of our `optGS`, *e.g.*, improving the representation compactness by up to 20.28% and achieving $3.53\times$ speedup in running time than the state-of-the-art methods.**

I. Introduction

Graphs have been frequently used to model the relationship between entities in many real-world applications [2], *e.g.*, web pages [1], social networks [4], and the transportation networks [12]. Such graphs are usually large and continuously growing. Consequently, it is crucial to find a storage-efficient manner to represent these large graphs. In addition to reducing the storage cost, a compact graph representation allows a large graph to fit in the main memory of one single machine for effective processing and analysis [11].

Among various graph compression solutions, a widely used technique is known as *graph summarization*, which takes as input a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and outputs a more compact representation consisting of a summary graph $\overline{\mathcal{G}}$ and correction edge sets $\mathcal{C}$ [11], [15]. Specifically, the summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ is a compressed graph, where each node in $\mathcal{S}$ represents a disjoint subset of nodes in the original graph $\mathcal{G}$ and each edge in $\mathcal{P}$ indicates the edges between all pairs of nodes in the two subsets. For clarity, we name nodes and edges in a summary graph as *supernodes* and *superedges*, respectively. In addition, graph summarization introduces correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ to achieve lossless compression, where $\mathcal{C}^+$ contains the edges to be inserted and $\mathcal{C}^-$ specifies the edges to be removed when reconstructing the original graph $\mathcal{G}$ from the summary graph $\overline{\mathcal{G}}$. Compared to other graph compression techniques, graph summarization owns several valuable properties [16], *e.g.*, be compatible with other techniques to further compress the graph and queryable on the summary graph. Since a summary graph captures the high-level structure of the original graph, the output summary graph can be used for insightful graph visualization as well [15], [18].

Many efforts have been made to design various graph summarization methods [6], [11]. As the pioneer work, Navlakha *et al.* first formally define the correction set based graph summarization problem, and propose methods to iteratively search for the best pair of nodes, which can reduce the storage cost at the most, for merging over the entire graph [15]. Although the proposed methods can derive highly compact representations, they incur huge computation overheads and thus cannot scale to large graphs. The state-of-the-art methods, *e.g.*, *SWeG* [16] and *LDME* [18], have been designed based on the same algorithmic framework consisting of three steps, *i.e.*, *dividing*, *merging*, and *encoding*. These advanced methods divide supernodes into smaller groups prior to merging, and utilize an approximation metric for choosing candidate supernode pairs to merge. Compared with traditional methods, they can achieve a significant speedup on the overall execution with an acceptable loss in representation compactness.

Despite huge advantages, the state-of-the-art methods are still inefficient in terms of running time and compactness of outputs. Although *SWeG* [16] greatly alleviates the computation challenge, its dividing step usually leads to unbalanced groups, where some groups are relatively larger and seriously slow down the speed. In addition, the approximation metric in *SWeG* is not capable of finding the best candidate supernodes to merge, and as a result, the final representation compactness is declined. *LDME* [18] is faster than *SWeG* by leveraging a novel dividing strategy to reduce group sizes, while it may separate the best supernodes for merging into different groups, resulting in a poor compression ratio. For example, we conduct experiments on graph EU (see more details of graph datasets in Section IV-A) to compare their performance, and find that *LDME* runs $19.30\times$ faster than *SWeG*, while its output takes 76.23% more spaces. *LDME* is adjustable to balance speed and compactness, however, it is hard to set the best parameter for each graph in advance.

[*] Corresponding author: Zhidan Liu.

To address the limitations of existing works, we empirically study the correction set based lossless graph summarization problem, and propose an *optimized graph summarization* (optGS) method to further advance current studies. With several optimization techniques, optGS can obtain high representation compactness with a shorter running time. Specifically, we test the impact of different settings of group size, and propose a computation-oriented diving method that further divides groups generated by *SWeG* or *LDME* into subgroups of proper size. In addition, we enhance *SWeG*'s approximation metric by considering the storage cost of supernodes, which can better measure the contribution of a candidate pair to the reduction of storage cost. Lastly, we observe that there are abundant computations involved for the same candidate pairs across multiple iterations. We thus exploit bloom filters to efficiently record such supernode pairs, and skip computations for the pairs by querying bloom filters.

In summary, the contributions of our work are as follows:

- We empirically investigate the impact of group size on graph summarization, and present a dividing method to split large groups into proper subgroups.
- We discover the inefficiency of *SWeG*'s approximation metric, and propose an improved metric for finding the best candidate pairs to merge.
- We observe repeated computations for the same candidate pairs, and devise a bloom filter based redundant computation avoidance strategy to eliminate unnecessary computation overheads.
- We conduct extensive experiments with seven large graph datasets to evaluate the performance of optGS. Compared to the state-of-the-art methods, optGS improves *SWeG* and *LDME* up to 20.28% in representation compactness, and achieves a $3.53\times$ speedup in running time.

The rest of this paper is organized as follows. The preliminary is presented in Section II. We elaborate and evaluate the design of optGS in Section III and Section IV, respectively. Finally, Section V concludes this paper.

II. Preliminary and Related Works

A. Problem Definition

Following previous works [15], [16], [18], we also focus on the *correction set based graph summarization* since it can be easily extended for both lossless and lossy graph compression. We introduce the involved concepts as follows.

Consider a given undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a set $\mathcal{V}$ of nodes and a set $\mathcal{E}$ of edges. Each edge $(u, v) \in \mathcal{E}$ is an unordered pair of distinct nodes $u, v \in \mathcal{V}$. The set of neighbors of each node u in $\mathcal{G}$ is denoted by $\mathbb{N}_u = \{v | (u, v) \in \mathcal{E}\}$. For the sake of reducing storage cost and enabling effective graph analysis, a graph summarization solution aims to turn graph $\mathcal{G}$ into a compact representation that includes a *summary graph* $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and *correction edge sets* $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$.

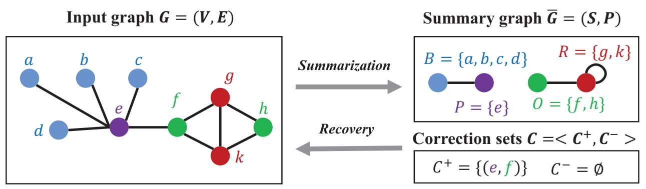

Fig. 1. Illustration of correction set based graph summarization and recovery.

- Summary graph $\overline{\mathcal{G}}$ consists of a set $\mathcal{S}$ of *supernodes* and a set $\mathcal{P}$ of *superedges*.
  - Each supernode $A \in \mathcal{S}$ is a disjoint set of nodes in $\mathcal{V}$, and for any two different supernodes $A, B \in \mathcal{S}$, we have $A \cap B = \emptyset$. Meanwhile, each node $u \in \mathcal{V}$ is contained in exactly one supernode A in $\mathcal{S}$. We denote the supernode that each node u belongs to as A_u.
  - Each superedge $(A, B) \in \mathcal{P}$ represents full connects between any two distinct nodes $u \in A$ and $v \in B$, *i.e.*, $\{(u, v) | u \in A, v \in B\}$. If $A = B$, then $(A, B) = (A, A)$ indicates the self-loop at supernode $A \in \mathcal{S}$.
- Correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ consist of a set $\mathcal{C}^+$ of edges to be inserted and a set $\mathcal{C}^-$ of edges to be removed when recovering original graph $\mathcal{G}$ from the summary graph $\overline{\mathcal{G}}$.

Correction set based graph summarization can reconstruct a graph $\widetilde{\mathcal{G}} = (\mathcal{V}, \widetilde{\mathcal{E}})$ from a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ through three steps: ① For each superedge $(A, B) \in \mathcal{P}$, adding edges formed by all pairs of distinct nodes in A and B to $\widetilde{\mathcal{E}}$; ② Adding each edge in $\mathcal{C}^+$ to $\widetilde{\mathcal{E}}$; ③ Removing each edge in $\mathcal{C}^-$ from $\widetilde{\mathcal{E}}$. In particular, if $\mathcal{E} = \widetilde{\mathcal{E}}$, graph $\mathcal{G}$ is lossless summarized by $\overline{\mathcal{G}}$ and $\mathcal{C}$.

Based on above definitions, the correction set based lossless graph summarization problem is formally defined as follows.

Definition 1: (**Correction set based lossless graph summarization problem**) Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we aim to compute a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, such that the output representation cost (*i.e.*, Eq. (1)) of the original graph $\mathcal{G}$ can be minimized.

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

The objective in Eq. (1) aims to minimize the sum of the number of superedges in the summary graph and the number of edges in the correction sets. Similar to previous works [16], [18], we exclude all self-loops in $\mathcal{P}$ because they can be encoded using a single bit, as a result, their representation cost is negligible. Besides, we only consider lossless graph summarization, as the lossy case can be easily implemented by dropping certain edges in the correction edge sets $\mathcal{C}$ [15].

Figure 1 illustrates an example of graph summarization and graph recovery. The input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with 9 nodes and 10 edges can be compactly represented as a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ with only 4 supernodes and 3 superedges and correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, where $\mathcal{C}^+$ contains only one edge (e, f) and no edge is stored in $\mathcal{C}^-$. With summary graph $\overline{\mathcal{G}}$ and correction sets $\mathcal{C}$, we can recover the original graph $\mathcal{G}$ in a lossless manner.

Algorithm 1: Algorithmic framework for lossless graph summarization

```
Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of iterations $T$
Output: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, correction sets
        $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$
1 Initialize each node $v \in \mathcal{V}$ as a supernode in $\mathcal{S}$;
2 for $it = 1 \to T$ do
3     Divide $\mathcal{S}$ into disjoint groups $\{\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \cdots, \mathcal{S}^{(m)}\}$;
4     Perform merges in each disjoint group;
5 Encode edges $\mathcal{E}$ into superedges $\mathcal{P}$ and correction sets $\mathcal{C}^+$,
  $\mathcal{C}^-$;
6 return $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C}$;
```

B. Algorithmic Framework of State-of-the-Art Works

According to whether nodes are aggregated into supernodes or not, previous graph summarization studies can be classified into two categories, *i.e.*, *grouping* [7], [8], [9], [15], [16], [18] and *non-grouping* [3], [5], [10], [14], [17]. In this paper, we focus on the research works of *grouping based graph summarization with correction edge sets*.

The state-of-the-art lossless graph summarization methods [16], [18] generally rely on the same algorithmic framework, as shown in **Algorithm 1**, which consists of three important steps, *i.e.*, *dividing*, *merging*, and *encoding*. Given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and number T of iterations, an expected method summarizes graph $\mathcal{G}$ repeatedly over T iterations by performing sequences of supernode merges per iteration. Initially, each node in $\mathcal{V}$ is initialized as a unique supernode, and all supernodes form the initial set $\mathcal{S}$. Then in each iteration, supernodes in $\mathcal{S}$ are divided into disjoint groups, and suitable supernode pairs are merged within each group. We will describe each step in detail as follows.

• ***Dividing***. To address the computation bottleneck of traditional graph summarization methods [15], *SWeG* [16] proposes to divide all supernodes into disjoint groups according to their similarity on the node connectivity. Specifically, *SWeG* divides supernodes using a function called as *shingle*. For a regular node $u \in \mathcal{V}$, its shingle $f(u)$ is defined as

$$f(u) = \min_{v \in \mathbb{N}_u \ or \ v = u} h(v),$$

where h is a random bijective function $h : \mathcal{V} \to \{1, 2, \cdots, |\mathcal{V}|\}$. The *shingle* function can be easily extended for supernode $A \in \mathcal{S}$ as $F(A) = \min_{u \in A} f(u)$. As a result, all supernodes in $\mathcal{S}$ can be divided into disjoint groups $\{\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \cdots, \mathcal{S}^{(m)}\}$, where supernodes in each group have the same shingle value.

Different from *SWeG*, another state-of-the-art method, *i.e.*, *LDME* [18], proposes to use a weighted locality sensitive hashing (LSH) technique to produce a hash signature of length k for each supernode $A \in \mathcal{S}$, and divides supernodes based on their signature values into disjoint groups, where supernodes within each group have the same signature value.

• ***Merging***. This step is performed within each group $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \cdots, \mathcal{S}^{(m)}\}$. Specifically, it merges supernodes by selecting a random supernode A as the *anchor* and then determining the best candidate B for A in the same group $\mathcal{S}^{(i)}$. Supernodes A and B will be merged if the result of the merger reduces Eq. (1) by a sufficient amount. Before dividing into the details, we present some definitions.

Definition 2: (**Cost of superedge**) For any superedge $(A, B) \in \mathcal{P}$, the cost of superedge (A, B), given current supernode set $\mathcal{S}$, is defined as:

$$Cost^{\mathcal{S}}(A, B) = \min\{1 + \pi_{AB} - \varepsilon_{AB}, \varepsilon_{AB}\}, \tag{2}$$

where $\varepsilon_{AB} = \{(u, v) \in \mathcal{E} | u \in A, v \in B\}$ is the set of edges connecting supernode A and B, and π_{AB} is the set of all pairs of nodes in A and B.

Definition 3: (**Cost of supernode**) For any supernode $A \in \mathcal{S}$, the cost of supernode A, given current supernode set $\mathcal{S}$, is defined as:

$$Cost^{\mathcal{S}}(A) = \sum_{B \in \mathbb{N}_A \cup A} Cost^{\mathcal{S}}(A, B), \tag{3}$$

where $Cost^{\mathcal{S}}(A, B)$ is the cost of superedge (A, B), and $\mathbb{N}_A = \bigcup_{u \in A} \mathbb{N}_u$ represents the set of nodes that have edge adjacent to any node in A.

Specifically, the cost of a supernode A implies how A contributes to Eq. (1) based on its connectivity to the neighbors. Based on the cost definitions, we then define the concept of $Saving$ due to the merger of two supernodes.

Definition 4: (**Saving of a merge**) The saving of a merge between supernodes A and B ($A \neq B \in \mathcal{S}^{(i)}$), given current supernode set $\mathcal{S}$, is defined as:

$$Saving^{\mathcal{S}}(A, B) = 1 - \frac{Cost^{(\mathcal{S} - \{A, B\}) \cup \{A \cup B\}}(A \cup B)}{Cost^{\mathcal{S}}(A) + Cost^{\mathcal{S}}(B) - Cost^{\mathcal{S}}(A, B)}, \tag{4}$$

where $Cost^{\mathcal{S}}(A) + Cost^{\mathcal{S}}(B) - Cost^{\mathcal{S}}(A, B)$ is the cost of supernodes A and B before their merge, and $Cost^{(\mathcal{S} - \{A, B\}) \cup \{A \cup B\}}(A \cup B)$ is the cost after merging.

Essentially, $Saving^{\mathcal{S}}(A, B)$ in Eq. (4) is the ratio of the cost reduction due to the merge of supernodes A and B and the cost before their merge. Based on this metric, traditional methods [15] iteratively search for the pair of supernodes, which can produce the largest saving over the entire graph, and merge them into one new supernode in a greedy manner. However, *SWeG* [16] claims that computing $Saving$ is computationally expensive and proposes an approximation metric known as $SuperJaccard$ similarity to approximate $Saving$ calculations.

Definition 5: (**SuperJaccard similarity**) The $SuperJaccard$ similarity between any two supernodes A and B is defined as:

$$SuperJaccard(A, B) = \frac{\sum_{v \in \mathbb{N}_A \cup \mathbb{N}_B} \min(w(A, v), w(B, v))}{\sum_{v \in \mathbb{N}_A \cup \mathbb{N}_B} \max(w(A, v), w(B, v))} \tag{5}$$

where $w(A, v) = |\{u \in A | (u, v) \in \mathcal{E}\}|$ is the number of nodes in supernode $A \in \mathcal{S}$ adjacent to node $v \in \mathcal{V}$.

Because $SuperJaccard(A, B)$ can measure the similarity of A and B in term of their connectivity, *SWeG* thus employs it to search the best merge candidate. After identifying the best merge candidate B for A by using *SuperJaccard* similarity, then $Saving^S(A, B)$ is computed only once in *SWeG* to decide whether to merge or not. More specifically, if $Saving^S(A, B) \geq \theta(it)$ in the it-th iteration, then A and B are merged; Otherwise they are not merged in current iteration. In general, the merging threshold $\theta(it)$ is defined as

$$\theta(it) = \frac{1}{1 + it}, \quad 1 \leq it \leq T. \tag{6}$$

Since $\theta(it)$ decreases along with time, more merging would be performed in the later iterations.

Instead of using $SuperJaccard$ to find the candidate for merging, *LDME* [18] calculates the $Saving^S(A, B)$ value for each supernode $B \in \mathcal{S}^{(i)}$ with respect to anchor A, and merges them if $Saving^S(A, B)$ is above the threshold $\theta(it)$. *LDME* employs hashtable-of-hashtables structure to index edges between supernodes, which can accelerate the computations of $Cost$ and $Saving$ values. However, this data structure introduces extra storage space and needs maintenance.

• ***Encoding***. After executing the steps of dividing and merging supernodes for T times, encoding step takes supernodes $\mathcal{S}$ from merging step and encodes edges $\mathcal{E}$ of input graph $\mathcal{G}$ into superedges $\mathcal{P}$ and correction edge sets $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$. Specifically, we either (*i*) encode a superedge $(A, B) \in \mathcal{P}$ and add the extraneous edges to $\mathcal{C}^-$; or (*ii*) add existing edges to $\mathcal{C}^+$ without creating a superedge. In the former case, we encode a superedge (A, B) and as a result, may introduce edges that were not in the original graph. We thus add these extraneous edges to $\mathcal{C}^-$, implying edges to be removed during recovery.

In both *SWeG* [16] and *LDME* [18], they encode superedges according to the same rules. For each pair of supernodes $A, B \in \mathcal{S}$, where $\varepsilon_{AB} \neq \emptyset$, if $\varepsilon_{AB} \leq \frac{\pi_{AB}}{2}$ then we do not encode superedge (A, B) and ε_{AB} is merged into $\mathcal{C}^+$; Otherwise, we encode superedge (A, B) and edges in $\{\pi_{AB} - \varepsilon_{AB}\}$ are merged into $\mathcal{C}^-$. In the special case where $A = B$, the condition is that if $\pi_{AA} \leq \frac{\varepsilon_{AB}}{2}$, we do not encode a loop superedge of A; otherwise, we encode a loop superedge.

III. Optimized Design

Although the state-of-the-art methods, *i.e.*, *SWeG* [16] and *LDME* [18], have greatly improved the traditional methods [15], we still find their deficiencies in terms of computation and representation compactness. Therefore, we present `optGS` that includes a set of optimization techniques to further advance current graph summarization researches.

A. Computation-oriented Supernode Dividing

Current advances improve traditional methods [15] by dividing all supernodes into disjoint groups and merging supernodes within each group individually. Despite certain losses in representation compactness, they significantly accelerate

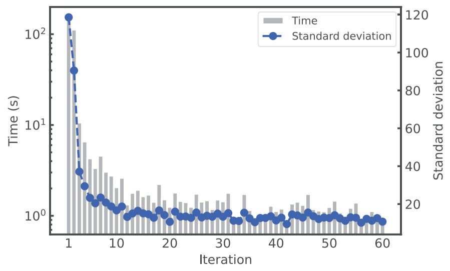

Fig. 2. Running time and standard deviation of group sizes for each iteration.

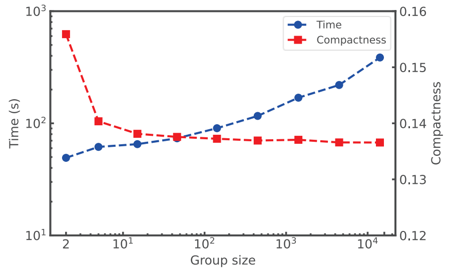

Fig. 3. Impact of group sizes on running time and representation compactness.

the whole process of graph summarization. According to our experiments, however, supernode groups derived by *shingle* function in *SWeG* [16] vary greatly in their sizes. In particular, some groups have much more supernodes than others, which could potentially lead to unbalanced computation workloads among groups, which will slow down the whole process. We run *SWeG* on a representative graph CN (Please see more details about the graph datasets in Section IV-A) for $T = 60$ iterations, and record the running time and standard deviation of all supernode groups for each iteration. Figure 2 shows that there exists a strong correlation between the deviation of group sizes and the running time of an iteration.

In theory, smaller groups will trigger much fewer computations as there are fewer supernodes to be evaluated for searching the best candidate within each group, while also degrading the representation's compactness. We run *SWeG* on graph CN by varying the group sizes. Figure 3 shows the experiment results on running time and representation compactness, which is the ratio between storage cost for summary graph and corrections expressed in Eq. (1) and the number $|\mathcal{E}|$ of all edges. The results comply with our analysis on the impact of group sizes. Another state-of-the-art *LDME* [18] can control group sizes by tuning parameter k, while it is hard to determine the best k for each graph without trial.

Based on the above experimental observations, we thus propose a computation-oriented supernode dividing method that aims to split a large group into multiple proper subgroups. Consider that the number of computations involved for a group of size n is about $\mathcal{O}(n^2)$, we estimate the group size for a given computation budget and present the results in Table I (see the second row). In addition, we run *SWeG* on a modern server and record the running time for each group size setting, as shown in the third row of Table I. The experiment result in Table I shows that when group size n increases, more running time will be taken. While we find the running time for groups smaller than

TABLE I
NUMBER AND TIME OF COMPUTATIONS INVOLVED FOR DIFFERENT GROUP SIZE.

Budget/$\mathcal{O}(n^2)$	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Size/n	2	5	15	46	142	448	1415	4473	14143
Time (ms)	10^{-3}	10^{-2}	10^{-1}	10^{-1}	10^0	10^1	10^2	10^3	10^4

46 to be relatively stable. Therefore, we empirically split a group with size greater than the threshold $\mu = 46$ into smaller subgroups. For a given supernode group $\mathcal{S}^{(i)}$ with size $|\mathcal{S}^{(i)}|$, we determine the size of subgroups (denoted by $\phi(|\mathcal{S}^{(i)}|)$) using the following equation:

$$\phi(|\mathcal{S}^{(i)}|) = \arg\min_{2<n\leq \frac{|\mathcal{S}^{(i)}|}{2}} \frac{|\mathcal{S}^{(i)}|}{n} \times \tau_n, \tag{7}$$

where n is the candidate group size listed in the second row of Table I and τ_n represents the running time for a group of size n that can be referred to from the third row of Table I. We exclude the case of splitting a group into extremely small subgroups consisting of only 2 supernodes, which will lead to terrible compression performance.

Therefore, we will divide a large group $\mathcal{S}^{(i)}$, derived from the dividing step of *SWeG* or *LDME*, with size $|\mathcal{S}^{(i)}| > \mu$ into $\lceil \frac{|\mathcal{S}^{(i)}|}{\phi(|\mathcal{S}^{(i)}|)} \rceil$ subgroups. For simplicity, supernodes in $\mathcal{S}^{(i)}$ are randomly assigned to these subgroups. It is worth noting that the best setting of μ for different computing hardware can be experimentally found by investigating various group size settings and deriving a result table like Table I, which can be easily obtained for each computing server.

B. Degree-aware Approximation Metric

During the supernode merging step, *SWeG* [16] employs $SuperJaccard(A, B)$ (*i.e.*, Eq. (5)) instead of $Saving(A, B)$[1] (*i.e.*, Eq. (4)) to find the best candidate supernode B for anchor A, where $B \neq A \in \mathcal{S}^{(i)}$. This is because $SuperJaccard(A, B)$ is cheaper than $Saving(A, B)$ on computations, and intuitively $Saving(A, B)$ tends to be high when A and B have similar connectivity that could be approximated by their $SuperJaccard$ value.

However, $SuperJaccard$ based merging will bring about a certain loss in representation compactness, as the definition of $SuperJaccard$ in Eq. (5) only considers the number of edges between A and B, while neglecting the key information about the size of supernodes. By comparing the definitions of $Saving$ and $SuperJaccard$ in Eq. (4) and Eq. (5), we suspect that the approximation metric cannot work well in some cases. For example, for a common node v of supernodes A and B, if v just connects few nodes of one supernode but most nodes of the other supernode, *e.g.*, $w(A, v) = 1$ and $w(B, v) = |B|$ (or $w(A, v) = |A|$ and $w(B, v) = 1$) where $|A|$ and $|B|$ are the number of nodes in supernodes A and B respectively, then $SuperJaccard(A, B)$ tends to be low while $Saving(A, B)$ is still high if A and B have such common nodes. As a result,

[1]We omit the superscript $\mathcal{S}$ if the context is clear.

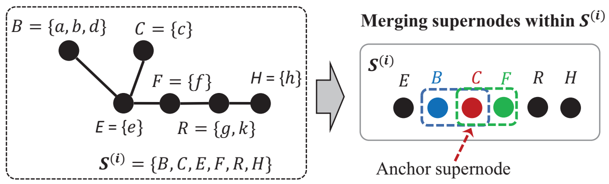

Fig. 4. An example to demonstrate the deficiency of $SuperJaccard$.

the candidate chosen by $SuperJaccard$ may not be the same one as the one selected by using $Saving$ value.

To fill the gap, we correct the $SuperJaccard$ calculation by taking the size of supernodes into account, and using the degree-aware approximation metric $\overline{SuperJaccard}$ value for supernodes A and B is defined as

$$\overline{SuperJaccard}(A, B) = \frac{\sum_{v\in\mathbb{N}_A\cup\mathbb{N}_B} \min(\frac{w(A,v)}{|A|}, \frac{w(B,v)}{|B|})}{\sum_{v\in\mathbb{N}_A\cup\mathbb{N}_B} \max(\frac{w(A,v)}{|A|}, \frac{w(B,v)}{|B|})}. \tag{8}$$

Example: As shown in Figure 4, given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with 9 nodes (*i.e.*, $\mathcal{V} = \{a, b, c, d, e, f, g, h, k\}$) and 10 edges between these nodes, assuming that in some iteration supernodes B, C, E, F, R, H are divided into the same group $\mathcal{S}^{(i)}$. Then, we select supernode C as the anchor supernode, and try to search for the best candidate within $\mathcal{S}^{(i)}$ for the possible merging with anchor C. If we use $Saving$ as the metric to compare supernodes B and F, we have $Saving(C, F) = \frac{1}{3} < Saving(C, B) = \frac{1}{2}$, which indicates that merging C with B can obtain more compactness than F. However, the approximation metric $SuperJaccard$ suggests that supernode F is a better candidate than B, because $SuperJaccard(C, F) = \frac{1}{2} > SuperJaccard(C, B) = \frac{1}{3}$. As a comparison, we use our degree-aware approximation metric $\overline{SuperJaccard}$ to evaluate the two pairs of supernodes, and have $\overline{SuperJaccard}(C, F) = \frac{1}{2} < \overline{SuperJaccard}(C, B) = 1$, which leads to the same merging suggestion as the $Saving$ metric, *i.e.*, merging C with B will bring more storage cost reduction than F.

C. Redundant Computation Avoidance

As shown in **Algorithm 1**, the overall execution is iteratively performed T times over supernodes $\mathcal{S}$. Each iteration contains a dividing step and a merging step, where the dividing step tends to partition supernodes into groups while the merging step computes the best merge within each group. A merge happens when two supernodes can gain considerable profit on compactness by comparing with a threshold $\theta(it)$, which is gradually decreased and predictable as shown in Eq. (6). At the initial iterations, finding a candidate supernode that satisfies the merging condition is very difficult, resulting in

TABLE II
SUMMARY OF REAL-WORLD GRAPH DATASETS FOR EXPERIMENTS.

Name	#Nodes	#Edges	Summary
cnr-2000 (CN)	325,557	5,565,380	Hyperlinks
in-2004 (IN)	1,382,908	27,560,356	Hyperlinks
eu-2005 (EU)	862,664	32,778,363	Hyperlinks
dblp-2011 (DB)	986,324	6,707,236	Collaboration
hollywood-2009 (HO)	1,139,905	113,891,327	Collaboration
amazon-2008 (AM)	735,323	5,158,388	Co-purchase
frwiki-2013 (FR)	1,352,053	34,378,431	Hyperlinks

many supernode pairs repeatedly computing the $Saving$ values for multiple iterations. The computations are redundant, and we thus propose to record such supernode pairs and avoid the redundant computations by using bloom filters.

Bloom filter is a space-efficient probabilistic data structure, which is used to test whether an element is a member of a set [13]. In `optGS`, we employ a bloom filter to record the supernode pairs, which still cannot contribute to the reduction of storage cost in the next iteration given their current $Saving$ value. For a supernode pair (A, B) that fails to meet the merging condition $\theta(it)$ at the it-th iteration, we will insert (A, B) into the bloom filter if $Saving(A, B) < \theta(it+1)$. We refer to such a candidate supernode pair as *ineffective pair*. By recording and querying ineffective pairs with bloom filter, we can avoid repeated computation in the next iteration, and thus speed up the execution without loss of compactness.

A more aggressive approach is that we could capture more ineffective pairs for the next κ iterations if $Saving(A, B) < \theta(it+j)$, where $j = 1, 2, \cdots, \kappa$. To this end, we utilize κ-level bloom filters, denoted by $\mathcal{L}_{it} = \{\mathcal{L}^{(1)}, \mathcal{L}^{(2)}, \cdots, \mathcal{L}^{(\kappa)}\}$, to record the ineffective pairs given their $Saving$ values at the it-th iteration. Each bloom filter $\mathcal{L}^{(j)}$ records the ineffective pairs for the $(it+j)$-th iteration.

In practice, given a supernode pair (A, B) that is possibly merged at the it-th iteration, we will first lookup whether the pair (A, B) exists in the bloom filter $\mathcal{L}_{it}$. If (A, B) indeed exists, we simply skip the computation of $Saving(A, B)$ and will not merge them. If (A, B) is not recorded in $\mathcal{L}_{it}$, we then compute $Saving(A, B)$, and merge them if $Saving(A, B) \geq \theta(it)$; Otherwise, we insert pair (A, B) into the j-th bloom filter $\mathcal{L}^{(j)}$ if $Saving(A, B) < \theta(it+j)$, $j = 1, 2, \cdots, \kappa$. It is worth noting that bloom filters are continuously generated and discarded. For example, after the it-th iteration, we will discard the first bloom filter $\mathcal{L}^{(1)}$ in $\mathcal{L}_{it}$, and meanwhile create a new bloom filter to record ineffective pairs for the $(it+\kappa)$-th iteration. However, the number κ of bloom filters should be carefully selected because the graph structural information may change and the $Saving$ values of recorded pairs will consequently change as well. We experimentally examine the settings of κ in Section IV-B.

IV. Performance Evaluation

A. Experimental Setup

We compare the performance of `optGS` with two state-of-the-art methods using a set of real-world graph datasets.

Datasets: We utilize seven graph datasets downloaded from the Laboratory of Web Algorithmics[2] for the experiments. We remove all edge directions, duplicated edges, and self-loops. Table II shows the characteristics of each graph.

Baseline methods: We compare our `optGS` with the current state-of-the-art *SWeG* [16] and *LDME* [18]. Specifically, we have the following five methods for performance comparisons. (a) `optGS`-*S* that uses the $Shingle$ function of *SWeG* to derive initial supernode groups; (b) `optGS`-*L* that adopts the LSH technique of *LDME* to obtain initial supernode groups, where we set $k = 1$ as the signature length; (c) *SWeG* [16]; (d) *LDME5* with signature length $k = 5$; and (e) *LDME20* with signature length $k = 20$. In particular, `optGS`-*S* and `optGS`-*L* mainly differ in the way of obtaining the initial supernode groups. As variants of *LDME* [18], *LDME20* runs faster while *LDME5* can produce more compact output.

Evaluation metrics: Given a correction set based graph summarization representation $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and correction edge set $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ for the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In addition to the *running time*, we also use the *compactness* of output representation, defined as Eq. (9), to evaluate the compression performance of each graph summarization method.

$$compactness = \frac{|\mathcal{P}| + |\mathcal{C}^+| + |\mathcal{C}^-|}{|\mathcal{E}|} \tag{9}$$

The metric of *compactness* measures the relative size of output, where the numerator in Eq. (9) is the objective function of graph summarization (*i.e.*, Eq. (1)) while the denominator is the storage cost of the input graph $\mathcal{G}$, which is constant for all methods. The lower the *compactness* value is, the better the compression performance of the method provides.

Implementation: We implement the five graph summarization methods in Java 1.8. We directly adopt the open-sourced implementations of *SWeG* [16] and *LDME* [18] for the experiments, and tune the parameters to achieve their best performance respectively. We set the length of each bloom filter as 7298440 bits, and operate bloom filters with 5 hash functions. Such an implementation can efficiently insert and lookup 1000000 ineffective pairs with an extremely low false positive rate. By default, we use $\kappa = 2$ bloom filters to achieve the best redundant computation avoidance. All experiments are performed on a powerful server with 3.8GHz AMD Ryzen 9 3900X CPUs (with 12 cores) and 64 GB memory. We run each method for $T = 60$ iterations, and encode the graph every 5 iterations for computing metrics of *compactness* and *running time*. Each experiment is performed 5 trials, and we report the average values in terms of execution speed and compactness of output representation.

B. Results

Comparisons on *running time*. As shown in Figure 5, our proposed methods, *i.e.*, `optGS`-*S* and `optGS`-*L*, have similar

[2]Laboratory of Web Algorithmics: http://law.di.unimi.it/dataset.php.

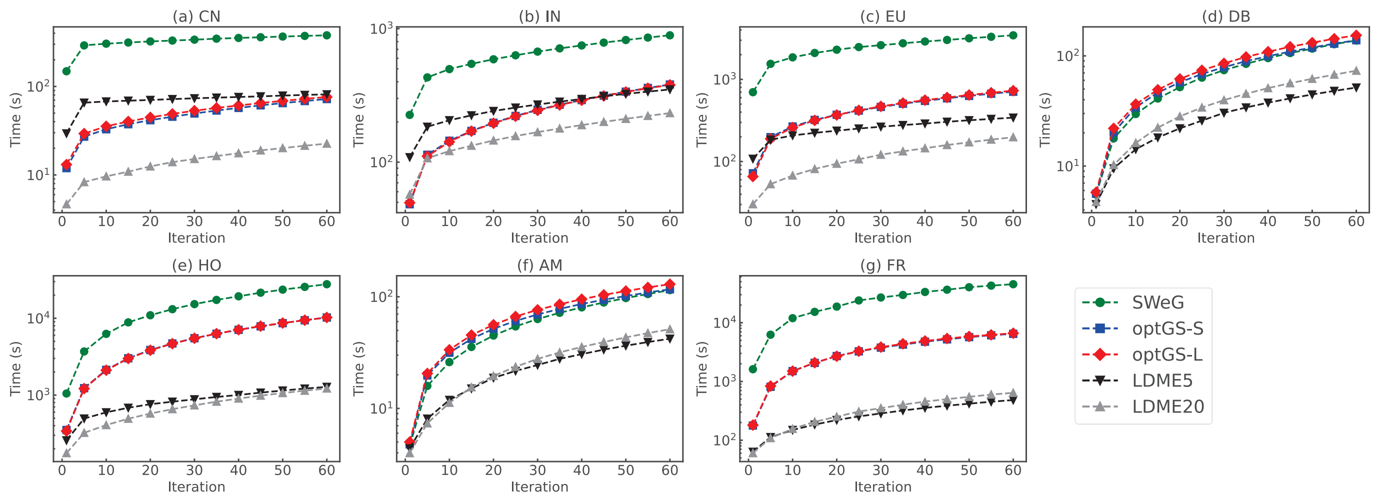

Fig. 5. Comparison among our methods, *i.e.*, `optGS`-*S* and `optGS`-*L*, and state-of-the-art methods in term of *running time* (in seconds) over 60 iterations.

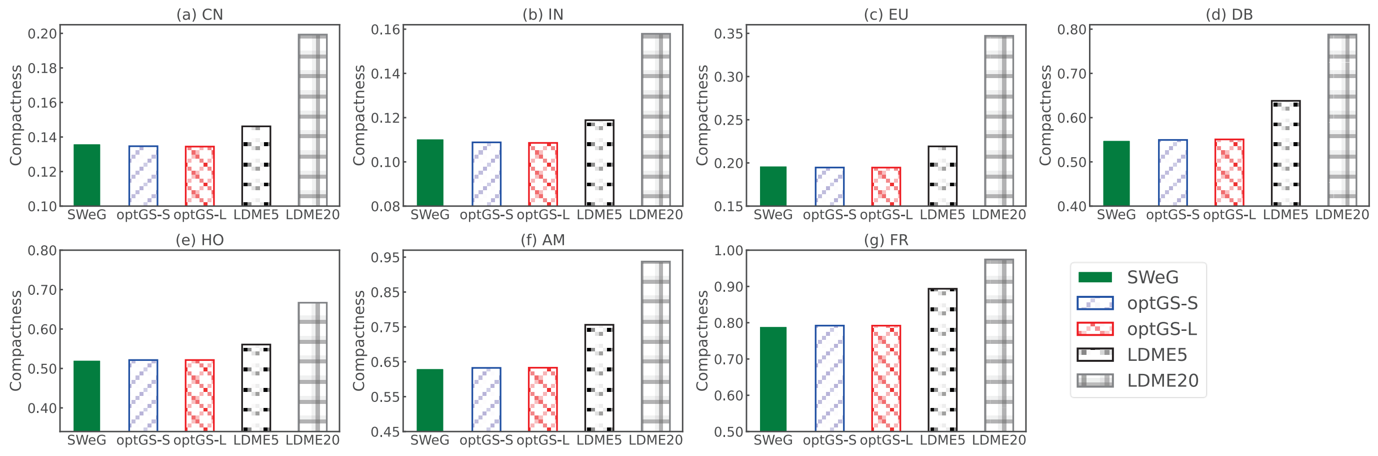

Fig. 6. Comparison among our methods, *i.e.*, `optGS`-*S* and `optGS`-*L*, and state-of-the-art methods in term of representation *compactness* over 60 iterations.

performance on the metric of running time across all graphs, and greatly outperform *SWeG* on most of the graphs. Overall, our methods achieves $2.59\times$ to $6.99\times$ speedup over *SWeG*. More specifically, our methods outperform *SWeG* on graph CN by $5.31\times$, IN by $2.60\times$, EU by $5.31\times$, HO by $2.71\times$, and FR by $6.91\times$. In addition, we also observe that our methods run a bit slower than *SWeG* on graphs DB and AM. The possible reason could be that their graph structures lead to less unbalanced groups, and as a result, our methods achieve similar performance as *SWeG* in running time.

From Figure 5, we see that *LDME5* (*LDME20*) can achieve $2.67\times$ $(1.88\times)$ to $93.67\times$ $(73.60\times)$ speedup over *SWeG*, respectively. *LDME20* achieves the best speedup performance because the size of supernode groups is very small, which potentially misses a large amount of candidate supernode pairs and will affect the final compactness of representation (as discussed later). On the contrary, the size of groups generated by *LDME5* is relatively larger than *LDME20*. The speedup performance of *LDME* is determined by the signature length k, but the optimal setting of k varies among different graphs. As a result, it is difficult to determine k for a given input graph in advance. Different from *SWeG* and *LDME*, our methods speed up the computations by limiting the size of each group, where the size threshold μ is relatively generic to most graphs.

Comparisons on *compactness*. As shown in Figure 6, our methods, *i.e.*, `optGS`-*S* and `optGS`-*L*, perform similarly in the metric of *compactness*, and they achieve comparable final compression performance as *SWeG* across the seven graphs. We even find that our methods outperform *SWeG* with a slight improvement on graph for CN (by 1.33%), IN (by 1.61%) and EU (by 1.20%). Such improvements in the representation compactness are mainly attributed to the proposed approximation metric $\overline{SuperJaccard}$, which helps find the best candidate supernode on reducing storage cost. We also observe that the five methods get more compact representations for graphs CN, IN, and EU, which are hyperlinks to web pages, than the other four graphs. It may be that web graphs exist much more similar connections among nodes than other kinds of graphs.

Although *LDME* runs fast, we find that both *LDME5* and *LDME20* perform poorly on deriving compact representations on all graphs, as shown in Figure 6. Specifically, *LDME20* has the worst compactness performance among the five methods. It is because a long signature (*i.e.*, $k = 20$) will divide all supernodes into much smaller groups and thus miss many

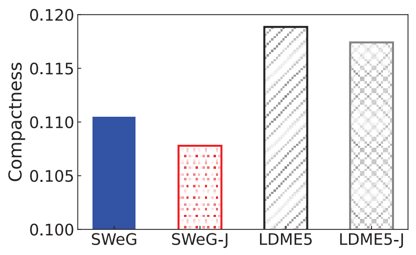

Fig. 7. Compression comparisons between different approximation metrics.

proper supernode pairs for merging. As a trade-off for compression, *LDME5* runs slower than *LDME20* but can get better compactness of outputs. Compared to *LDME5* and *LDME20*, our methods perform much better in the terms of compactness, with improvements of 10.63% and 29.94%, respectively.

Summary. According to these experiment results in Figure 5 and Figure 6, we find that both *SWeG* and *LDME* cannot well balance the performance of *running time* and *compactness*. On the contrary, our methods can achieve equal or even better compactness than *SWeG*, but have much speedier execution. On average, our methods improve *SWeG* by $3.53\times$ in running time and 0.53% in representation *compactness*. In addition, although our methods run slower than *LDME*, `optGS` greatly improves *LDME* in compactness by 20.28% on average.

Effectiveness of improved approximation metric. We replace the approximation metric used in *SWeG* and *LDME5* with our improved $\overline{SuperJaccard}$ for finding candidate supernodes, and the two variants are denoted by *SWeG-J* and *LDME5-J*, respectively. We compare their compression performance on all graphs, and report the results on graph IN in Figure 7. We find that *SWeG-J* and *LDME5-J* improve their original versions in *compactness* by 2.45% and 1.18%, respectively. We observe similar results on other graphs, but omit them due to space limitations. This experiment shows the effectiveness of our degree-aware approximation metric that can select the best candidate for merging.

Effectiveness of bloom filter. We combine the κ-level bloom filter based redundant computation avoidance strategy with *SWeG* and vary the number of bloom filters used. Similarly, we only report the experiment results on graph IN in Figure 8. Compared to the original *SWeG* design (*i.e.*, $\kappa = 0$), we find that the bloom filter is useful to avoid unnecessary $Saving$ computations of ineffective pairs, and thus reduce the running time. However, the speedup performance is weakened when more bloom filters (*i.e.*, $\kappa = 3$) are used. This could be because querying time on multiple bloom filters outweighs the benefits of computation avoidance. Therefore, we suggest $\kappa = 2$ to obtain the best speedup performance.

V. Conclusion

In this paper, we present `optGS` that incorporates a set of optimization techniques to improve state-of-the-art lossless graph summarization methods, namely *SWeG* and *LDME*. Specifically, we have devised a computation-oriented dividing method and a redundant computation avoidance strategy to reduce the computation overhead. Furthermore, we present a better approximation metric to select the best candidate pairs for merging. Extensive experiments on seven large-scale graph datasets demonstrate that `optGS` provides 0.53% and 20.28% higher compactness of representation than *SWeG* and *LDME*, respectively. In addition, our method achieves $3.53\times$ speedup in running time than *SWeG*.

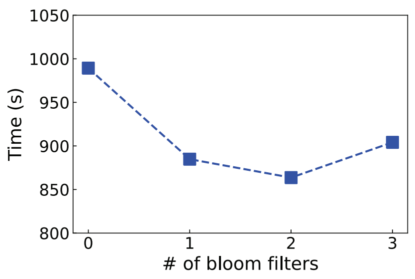

Fig. 8. Impact of number of bloom filters used for *SWeG* on running time.

Acknowledgment

This work was supported in part by China NSFC Grant (No.62172284) and the grant of Guangdong Basic and Applied Basic Research Foundation (No.2022A1515010155).

References

[1] N. Aspert, V. Miz, B. Ricaud, and P. Vandergheynst. A graph-structured dataset for wikipedia research. In *WWW*, 2019.
[2] D. Chakrabarti and C. Faloutsos. Graph mining: laws, generators, and algorithms. *ACM Computing Surveys*, 38(1):1–69, 2006.
[3] X. Gou, L. Zou, C. Zhao, and T. Yang. Fast and accurate graph stream summarization. In *IEEE ICDE*, 2019.
[4] B. Huang, Z. Liu, and K. Wu. Accelerating pagerank in shared-memory for efficient social network graph analytics. In *IEEE ICPADS*, 2020.
[5] C. Hübler, H.-P. Kriegel, K. Borgwardt, and Z. Ghahramani. metropolis algorithms for representative subgraph sampling. In *IEEE ICDM*, 2008.
[6] A. Khan, S. S. Bhowmick, and F. Bonchi. Summarizing static and dynamic big graphs. *Proceedings of the VLDB Endowment*, 10(12):1981–1984, 2017.
[7] J. Ko, Y. Kook, and K. Shin. Incremental lossless graph summarization. In *ACM SIGKDD*, 2020.
[8] K. A. Kumar and P. Efstathopoulos. Utility-driven graph summarization. *Proceedings of the VLDB Endowment*, 12(4):335–347, 2018.
[9] K. Lee, H. Jo, J. Ko, S. Lim, and K. Shin. Ssumm: sparse summarization of massive graphs. In *ACM SIGKDD*, 2020.
[10] E. Liberty. Simple and deterministic matrix sketching. In *ACM SIGKDD*, 2013.
[11] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: a survey. *ACM Computing Surveys*, 51(3):1–34, 2018.
[12] Z. Liu, P. Zhou, Z. Li, and M. Li. Think like a graph: real-time traffic estimation at city-scale. *IEEE Transactions on Mobile Computing*, 18(10):2446–2459, 2019.
[13] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo. Optimizing bloom filter: challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.
[14] A. Maccioni and D. J. Abadi. Scalable pattern matching over compressed graphs via dedensification. In *ACM SIGKDD*, 2016.
[15] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD*, 2008.
[16] K. Shin, A. Ghoting, M. Kim, and H. Raghavan. Sweg: lossless and lossy summarization of web-scale graphs. In *WWW*, 2019.
[17] N. Tang, Q. Chen, and P. Mitra. Graph stream summarization: from big bang to big crunch. In *ACM SIGMOD*, 2016.
[18] Q. Yong, M. Hajiabadi, V. Srinivasan, and A. Thomo. Efficient graph summarization using weighted LSH at billion-scale. In *ACM SIGMOD*, 2021.