# Graph Summarization: Compactness Meets Efficiency

DEMING CHU, University of New South Wale, Australia
FAN ZHANG[*], Guangzhou University, China
WENJIE ZHANG, University of New South Wale, Australia
YING ZHANG, University of Technology Sydney, Australia
XUEMIN LIN, ACEM, Shanghai Jiao Tong University, China

As the volume and ubiquity of graphs increase, a compact graph representation becomes essential for enabling efficient storage, transfer, and processing of graphs. Given a graph, the *graph summarization* problem asks for a compact representation that consists of a summary graph and the corrections, such that we can recreate the original graph from the representation exactly. Although this problem has been studied extensively, the existing works either trade summary compactness for efficiency, or vice versa. In particular, a well-known greedy method provides the most compact summary but incurs prohibitive time cost, while the state-of-the-art algorithms with practical overheads are more than 20% behind in summary compactness in our comparison with the greedy method.

This paper presents *Mags* and *Mags-DM*, two algorithms that aim to bridge the compactness and efficiency in graph summarization. *Mags* adopts the existing greedy paradigm that provides state-of-the-art compactness, but significantly improves its efficiency with a novel algorithm design. Meanwhile, *Mags-DM* follows a different paradigm with practical efficiency and overcomes its limitations in compactness. Moreover, both algorithms can support parallel computing environments. We evaluate *Mags* and *Mags-DM* on graphs up to billion-scale and demonstrate that they achieve state-of-the-art in both compactness and efficiency, rather than in one of them. Compared with the method that offers state-of-the-art compactness, *Mags* and *Mags-DM* have a small difference ($< 0.1\%$ and $< 2.1\%$) in compactness. For efficiency, *Mags* is on average 11.1x and 4.2x faster than the two state-of-the-art algorithms with practical overheads, while *Mags-DM* can further reduce the running time by 13.4x compared with *Mags*. This shows that graph summarization algorithms can be made practical while still offering a compact summary.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**.

Additional Key Words and Phrases: Graph Summarization; Graph Algorithms; Massive Graphs;

**ACM Reference Format:**
Deming Chu, Fan Zhang, Wenjie Zhang, Ying Zhang, and Xuemin Lin. 2024. Graph Summarization: Compactness Meets Efficiency. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 140 (June 2024), 26 pages. https://doi.org/10.1145/3654943

[*] Fan Zhang is the corresponding author.

Authors' addresses: Deming Chu, University of New South Wale, Australia, ned.deming.chu@gmail.com; Fan Zhang, Guangzhou University, China, zhangf@gzhu.edu.cn; Wenjie Zhang, University of New South Wale, Australia, wenjie.zhang@unsw.edu.au; Ying Zhang, University of Technology Sydney, Australia, ying.zhang@uts.edu.au; Xuemin Lin, ACEM, Shanghai Jiao Tong University, China, xuemin.lin@sjtu.edu.cn.

ACM 2836-6573/2024/6-ART140
https://doi.org/10.1145/3654943

## 1 INTRODUCTION

Graphs abstract the relationships between entities, e.g., web links [5, 7, 34] and friendships among people [4, 9, 11]. People are increasingly interested in graphs nowadays and are producing more graph data than ever before. Large-scale graphs thus have become ubiquitous across different domains [10, 33, 42, 44].

A compact representation is essential for handling large graphs. It can save precious hardware resources (e.g., memory, disk, and network I/O) and speed up algorithms by allowing a large portion of the graph to reside in the cache [7, 15, 35]. This motivates a large body of techniques for compressing graphs [1–5, 7–9, 14, 17, 20–30, 32, 34–37, 39, 45, 46], including relabeling node [1, 5, 9, 14], compressing adjacency lists by reference encoding [5], encoding graph patterns [17, 23, 32], etc. Among these works, *graph summarization* [2, 21, 22, 24, 25, 27, 30, 34, 45] is a line of works that are particularly effective in compressing graphs and discovering structural patterns. According to our experiments, graph queries can be answered efficiently on the summary graph (see Section 6.6).

Given a graph $\mathcal{G}$, the *graph summarization* problem asks for a minimum-sized representation $R = (S, C)$ that contains a summary graph and the edge corrections. The *summary graph* $S$ aggregates the nodes with similar neighbor structures. The super-nodes in $S$ are disjoint sets of nodes in $\mathcal{G}$, while a super-edge in $S$ means linking all node pairs between the two sets of nodes. The *edge corrections* $C$ contains the edges to insert and remove, when we restore $\mathcal{G}$ from the summary graph exactly or with an accuracy guarantee (i.e., lossless or lossy summarization). Our paper mainly focuses on lossless graph summarization.

Navlakha et al. [30] are the first to formulate the problem of graph summarization, and they present a greedy method for graph summarization (referred to as *Greedy*). This method is well accepted for its simplicity and effectiveness, but it is prohibitive in time cost. Specifically, it runs in $O(n \cdot {d_{avg}}^3 \cdot (d_{avg} + \log m))$ time [30], where $n$ and $m$ are the number of nodes and edges in the graph, and $d_{avg}$ is the average degree (i.e. $2m/n$). Empirically, it cannot terminate in two days on a 3-million-edge graph (Amazon0601) [34], due to a large hidden constant factor in time complexity.

The inefficiency of *Greedy* has driven a plethora of algorithms [21, 22, 24–27, 34, 45] that aim to reduce the overhead of graph summarization. Shin et al. [34] present an $O(T \cdot m)$ time algorithm for graph summarization which can scale to billion-scale graphs, where $T$ is a pre-set iteration number. Shin et al.'s seminal paradigm has motivated *LDME* [45] and *Slugger* [25], two state-of-the-art algorithms with practical computation time. Those algorithms, however, are far behind in summary compactness. In particular, *Greedy* returns a summary whose size is on average 21.7% smaller than *LDME* and 30.2% smaller than *Slugger* (see the experiments in Section 6.2).

In short, no existing graph summarization algorithm can scale to billion-scale graphs while still providing a summary as compact as *Greedy* (Navlakha et al.'s greedy approach [30]). Hence, any practitioner who conducts graph summarization on sizable networks can only sacrifice summary compactness for efficiency, or vice versa.

**Contributions.** This paper presents *Mags* and *Mags-DM* (Multi-phase algorithms for graph summarization), and the latter follows a Divide-and-Merge paradigm. In particular, *Mags* borrows ideas from the *Greedy* method [30] but significantly improves its efficiency without sacrificing the summary compactness. This is achieved by a novel algorithm design that can largely reduce unpromising search and wasted updates in *Greedy*, while still retaining necessary search space for a compact summary. In the meantime, *Mags-DM* adopts the divide-and-merge paradigm of Shin et al. [34]. We propose a series of novel techniques that can improve the performance of the paradigm, including an improved dividing strategy, as well as three merging strategies for improving node

Table 1. Summary of Notations

| Notation | Definition |
|---|---|
| $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ | an undirected graph with nodes $\mathcal{V}$ and edges $\mathcal{E}$ |
| $n;\ m$ | the number of nodes/edges in $\mathcal{G}$ (assume $m > n$) |
| $d_{avg}$ | the average degree, i.e. $2m/n$ |
| $P_u$ | the nodes contained in a super-node $u$ |
| $N_u$ | the neighbor super-nodes of a super-node $u$ |
| $\mathcal{N}_u$ | the neighbors of a node (or super-node) $u$ in $\mathcal{G}$ |
| $s(u,v)$ | the saving of a super-node pair $(u,v)$, see Equation 4 |
| $mh(u,v)$ | a MinHash-based similarity measure, see Equation 5 |
| $T$ | the number of iterations |
| $\theta(t)$ | *SWeG*'s merge threshold [34] |
| $\omega(t)$ | our merge threshold, see Equation 6 |

selection, similarity measure, and merge threshold. Moreover, *Mags* and *Mags-DM* can support parallel environments.

We evaluate *Mags* and *Mags-DM* on a variety of graphs, and experimentally compare them with the state-of-the-art algorithms for graph summarization, i.e., *Greedy* [30], *LDME* [45], and *Slugger* [25]. For summary compactness, *Mags* and *Mags-DM* have a small difference ($< 0.1\%$ and $< 2.1\%$ on average respectively) with *Greedy* when we use small graphs that *Greedy* can process. In terms of efficiency, *Mags* is on average 11.1x faster than *LDME* and 4.2x faster than *Slugger* over all datasets in the experiments. Moreover, our *Mags-DM* is on average 13.4x faster compared with *Mags*. To our knowledge, this is the first time that compactness and efficiency are met in graph summarization, while existing solutions can only achieve state-of-the-art in one of them.

The principal contributions of this paper are as follows:

1. We propose a greedy algorithm *Mags* for graph summarization that runs in $O(T \cdot m \cdot (d_{avg} + \log m))$ time. Compared with the previous greedy method, it reduces the theoretical and practical time cost without affecting the summary compactness.
2. We design a divide-and-merge algorithm *Mags-DM* for graph summarization. The method takes an $O(T \cdot m)$ running time, and it outperforms existing divide-and-merge methods in summary compactness and practical efficiency.
3. We devise the parallel implementations of *Mags* and *Mags-DM* when shared-memory parallel computations are available.
4. Extensive experiments on large graphs demonstrate that *Mags* and *Mags-DM* achieve state-of-the-art compactness and efficiency, enabling practical use of graph summarization algorithms.

**Organizations.** We introduce the problem and existing solutions in Section 2. Then, we propose two novel methods in Sections 3 and 4. We detail the implementations and parallelism in Section 5. The experimental results are given in Section 6. We introduce other related works in Section 7 and conclude the paper in Section 8.

## 2 PRELIMINARIES

In this section, we first define the graph summarization problem. Next, we present an overview of the solutions from Navlakha et al. [30] and Shin et al. [34], and the limitations of the state-of-the-arts. Table 1 summarizes the notations frequently used in the paper.

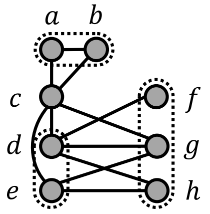


Fig. 1. Graph $\mathcal{G}$

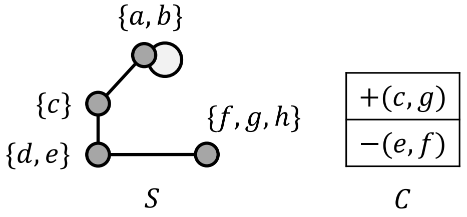


Fig. 2. Representation $R = (S, C)$

## 2.1 Problem Definition

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected network with nodes $\mathcal{V}$ and edges $\mathcal{E}$, where $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$. Given $\mathcal{G}$, we seek a representation $R = (S, C)$ that consists of a summary graph $S$ and the set of edge corrections $C$. The *summary graph* $(P, E)$ is built on a set of super-nodes $P$ where $P$ is a *partition* over $\mathcal{V}$ (i.e., disjoint subsets of $\mathcal{V}$). Each super-node $u \in P$ represents a set of nodes $P_u$ in $\mathcal{G}$, and each super-edge $(u, v) \in E$ means all pair of nodes in the cartesian product $P_u \times P_v$ are linked. The set of *edge corrections* $C$ stores the edges to insert and remove when recreating $\mathcal{G}$ from the summary graph, annotated as '$+e$' and '$-e$' respectively for each edge $e \in C$. Intuitively, the summary graph $(P, E)$ finds multiple sets of vertices with similar neighbors and groups them into super-nodes, while the edge corrections $E$ enables a lossless recreation of $\mathcal{G}$ from $(P, E)$.

**Example 1.** Consider the graph $\mathcal{G}$ in Figure 1 and its compact representation $R$ in Figure 2. Initially, $\mathcal{G}$ contains 11 edges. After we summarize $\mathcal{G}$ into $R$, the number of edges drops to 6, i.e., 4 summary graph edges and 2 edge corrections. The super-edge between $\{d, e\}$ and $\{f, g, h\}$ means linking all 6 node pairs among the two sets, while the correction $-(e, f)$ removes the edge $(e, f)$. This correctly restores the 5 edges between $\{d, e\}$ and $\{f, g, h\}$ in graph $\mathcal{G}$. For $\{c\}$ and $\{f, g, h\}$, there exists no super-edge but the correction $+(c, g)$ restores the edge $(c, g)$ in $\mathcal{G}$. After we process all super-edges and corrections as explained above, we can restore $\mathcal{G}$ from $R$ exactly.

**Definition 1** (Lossless Graph Summarization [30])**.**

- **Given:** a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$;
- **Find:** a summary graph $S = (P, E)$ and a set of corrections $C$;
- **to Minimize:** the representation cost of $R = (S, C)$, i.e.,

$$c(R) \;=\; |E| + |C|; \tag{1}$$

- **Subject to:** $\mathcal{G}$ can be losslessly recreated from $R$.

In the next section, we show that the problem can be rewritten as follows: Given $\mathcal{G}$, we ask for a set of super-nodes $P$ that minimizes the representation cost $c(R)$.

## 2.2 Optimal Encoding

The best possible set of super-nodes $P$ is hard to decide. However, given a fixed $P$, we can easily decide the optimal encoding, i.e., the best possible sets $E$ and $C$ with the minimum cost, according to [30]. Consider two super-nodes $u$ and $v$ from a set of super-nodes $P$. We define $\Pi_{uv} = P_u \times P_v$ as the cartesian product between $P_u$ and $P_v$, that is, $\Pi_{uv}$ contains all edges of $\mathcal{G}$ that may exist between the two super-nodes. Then, we let $E_{uv} \subseteq \Pi_{uv}$ be the edges *actually* exist in the original graph $\mathcal{G}$, i.e., $E_{uv} = \Pi_{uv} \cap \mathcal{E}$. After that, we can encode any $E_{uv}$ with the summary and correction structures:

1. if $|E_{uv}| > (1 + |\Pi_{uv}|)/2$, then we add the super-edge $(u, v)$ to $E$ and add '$-e$' to $C$ for each $e \in \Pi_{uv} \setminus E_{uv}$;

2. if $|E_{uv}| \leq (1 + |\Pi_{uv}|)/2$, then we add '$+e$' to $C$ for each $e \in E_{uv}$.

The costs of the above two ways are $|\Pi_{uv}| - |E_{uv}| + 1$ and $|E_{uv}|$ respectively. Such an encoding of $E_{uv}$ will always lead to the minimum cost among the two ways, that is,

$$c_{uv} \; = \; \min\{|\Pi_{uv}| - |E_{uv}| + 1, |E_{uv}|\}. \tag{2}$$

Therefore, we decide the optimal encoding of $E$ and $C$ as follows. Given $P$, for every pair of super-nodes $(u, v)$, we add the super-edge to $E$ if and only if $|E_{uv}| > (1 + |\Pi_{uv}|)/2$, and then insert the corrections into $C$ accordingly. Let $N_u$ be the set of *neighbor super-nodes* of a super-node $u$, i.e., $N_u = \{x \mid x \in P \wedge E_{ux} \neq \emptyset\}$. Then, based on the optimal encoding, Equation 1 can be rewritten as:

$$c(R) \; = \; \sum_{u \in P} \sum_{v \in N_u} c_{uv}. \tag{3}$$

Thus, the lossless graph summarization problem is equivalent to the following: Given $\mathcal{G}$, we seek a set of super-nodes $P$ that minimizes the cost $c(R)$ in Equation 3.

### 2.3 Navlakha et al.'s Greedy Solution

In a nutshell, Navlakha et al.'s greedy solution [30] (referred to as *Greedy*) starts with a set of super-nodes $P$ where each super-node contains an individual node, then iteratively merges the super-node pair $(u, v)$ that leads to the largest *saving* $s(u, v)$.

The saving $s(u, v)$ measures the normalized cost reduction after merging super-nodes $u$ and $v$. Recall that $N_u$ is the neighbor super-node set of $u$ s.t. the edge set $E_{ux}$ is not empty for any $x \in N_u$. Given a super-node $u$, we define the cost $c_u$ as the sum of costs for $u$'s neighbor set $N_u$, i.e., $c_u = \sum_{x \in N_u} c_{ux}$. After that, when we merge $u$ and $v$ into a new super-node $w$, the cost is reduced by $c_u + c_v - c_w$, and the saving $s(u, v)$ is defined as the cost reduction divided by the sum of costs for $u$ and $v$ before the merge, i.e.,

$$s(u, v) \; = \; \frac{c_u + c_v - c_w}{c_u + c_v}. \tag{4}$$

The saving takes the normalized value instead of the absolute cost reduction, as the latter is biased to high-degree vertices[30]. By definition, only those pairs with a positive saving can lead to a cost reduction, and such pairs must share at least one common neighbor, i.e., they are 2 hops apart. Therefore, *Greedy* maintains any positive-saving pair that is 2-hop apart, during its iterative merge of the super-nodes. Specifically, *Greedy* runs in three steps:

1. **Initialization**: This phase computes the saving for every 2-hop-apart pair, and then puts those with a positive saving into a priority queue $H$. It also initializes the set of super-nodes $P$ where each super-node contains a single node.
2. **Greedy Merge**: This phase iteratively merges the pair $(u, v)$ with the largest saving. After each merge, it iterates over any pair $(x, y)$ affected by the merge, and then re-computes its saving $s(x, y)$ and updates it in $H$. The process terminates when there is no positive-saving pair in $H$.
3. **Output**: This phase decides $R$ from the set of super-nodes $P$ (by Section 2.2) and returns $R$ as the final result, where $P$ is obtained from the greedy merge.

*Greedy* is concise in methodology, but it incurs prohibitive overheads due to the $O(n \cdot {d_{avg}}^3 \cdot (d_{avg} + \log m))$ time complexity [30], where $d_{avg}$ is the average degree of the input graph, i.e., $2m/n$. The saving update of Step 2 dominates the overall time cost. In particular, when merging $(u, v)$ into $w$, *Greedy* iterates over $x \in \{w\} \cup N_w$ (1-hop) and any $y$ in 2 hops. This identifies any pair $(x, y)$ affected by the merge, and costs roughly 3-hops. Then, for any $(x, y)$ visited, it re-computes

$s(x, y)$ by iterating over the neighbors of $x$ and $y$ (one more hop), and it also updates the saving of $(x, y)$ in $H$ using $\log |H|$ time. Note that we have $|H| \approx n \cdot {d_{avg}}^2$, i.e., $H$ contains the pairs between a node and any node in its 2-hop neighborhood. In summary, the saving update after each merge requires $O({d_{avg}}^3 \cdot (d_{avg} + \log(n \cdot {d_{avg}}^2)))$ time, and there are at most $n$ such merges. This leads to an $O(n \cdot {d_{avg}}^3 \cdot (d_{avg} + \log m))$ time complexity.

## 2.4 Shin et al.'s Divide-and-Merge Solution

Shin et al. overcome the inefficiency of *Greedy* and propose a new method called *SWeG* [34]. Similar to the initialization of *Greedy*, *SWeG* starts with a set of super-nodes $P$ where each super-node contains a single node. Then, *SWeG* runs in $T$ rounds, and the $t$-th round consists of two steps:

1. **Dividing**: This phase divides all super-nodes into disjoint groups $S^{(1)}, \cdots, S^{(d)}$, each of which contains a set of super-nodes with similar connectivity.
2. **Merging**: This phase merges some super-node pairs within each group. For each group $S^{(i)}$, it iteratively removes a random node $u \in S^{(i)}$, and then picks the node $v \in S^{(i)}$ whose neighbor set is the most similar to $u$ using Super-Jaccard. The pair $(u, v)$ is merged if the saving $s(u, v)$ is larger than a threshold $\theta(t)$.

Compared with *Greedy*, *SWeG* avoids a great number of saving computations. That is, *SWeG* limits the nodes we consider merging with a super-node $u$, which shrinks the search scope from the 2-hop neighborhood to the assigned group. On the other hand, *SWeG* ensures compactness by a greedy-like method, i.e., it first merges the pairs with a large saving and defers other pairs to subsequent rounds. To achieve this, *SWeG* only merges the pairs with a saving larger than $\theta(t)$ in the $t$-th round, where the merge threshold $\theta(t) = 1/(t + 1)$ gradually decreases as $t$ increases. Shin et al. show that *SWeG* runs in $O(T \cdot m)$ time, and that the algorithm can be extended to parallel and distributed computing [34].

## 2.5 Limitations of SOTA Solutions

Due to the efficiency of *SWeG* [34], this seminal work has motivated *LDME* [45] and *Slugger* [25], two state-of-the-art graph summarization algorithms with practical overheads built on *SWeG*. *LDME* and *Slugger* can process graphs of near billion scale, but they cannot output a compact summary like *Greedy*. Specifically, *Greedy* returns a summary that is on average 21.7% smaller than *LDME* and 30.2% smaller than *Slugger*, when we run the methods on small graphs that *Greedy* can process (Section 6.2). On the other hand, *Greedy* is not practical for large graphs, e.g., *Greedy* cannot terminate in two days on a 3-million-edge graph (Amazon0601) [34]. In summary, despite a large volume of existing works for graph summarization, none of them can efficiently process large graphs while returning a summary that is as compact as *Greedy*.

To bridge the compactness and efficiency for graph summarization, we propose two novel solutions: a greedy solution *Mags* that scales *Greedy*'s paradigm [30] to billion-scale graphs; a divide-and-merge solution *Mags-DM* that adopts *SWeG*'s paradigm [34] and returns a highly compact summary in a much shorter time.

# 3 PROPOSED GREEDY METHOD

As we mention in Section 2.5, *Greedy* [30] incurs prohibitive computation overheads but returns a highly compact summary. The main reason for *Greedy*'s inefficiency is that it computes the saving of any pair of nodes with 2-hop apart, and it updates the saving of any affected pair after every merge. Intuitively, most of those pairs are *unpromising* and most of the saving updates are *wasted*, since we are only interested in promising pairs (whose saving is high) and their saving updates.

```
Algorithm 1: Mags ($\mathcal{G}, T, k$)
$CP \leftarrow$ Generate up to $k \cdot n$ candidate pairs;                    // Alg. 2
Initialize a set of super-nodes $P \leftarrow \{\{u\} \mid u \in \mathcal{V}\}$;
for $t = 1..T$ do                                                               // Alg. 3
    Select a set of high-saving pairs from $CP$; Merge them in $P$;
    Update saving for any pair in $CP$ affected by the merges;
Decide the optimal $R$ from $P$;                                                // Alg. 4
return $R$;
```

This section presents *Mags*, a greedy method that borrows ideas from *Greedy* [30] but avoids its inefficiency with a novel algorithm design. Algorithm 1 outlines the basic idea of *Mags*. At a high level, *Mags* consists of three phases as follows:

1. **Candidate Generation (Algorithm 2).** This phase generates $k \cdot n$ candidate pairs. For each node $u \in \mathcal{V}$, we generate $k$ candidate pairs that both contain $u$ and provide a high saving.
2. **Greedy Merge (Algorithm 3).** This phase repeats a greedy merge for $T$ iterations. The $t$-th iteration merges a set of candidate pairs, and then updates saving for any candidate pair that is affected by the merges.
3. **Output (Algorithm 4).** This phase decides the optimal $R$ from the set of super-nodes $P$, according to the method we described in Section 2.2.

The main drawback of *Greedy* is that it considers plenty of node pairs that are not promising to merge, and it frequently updates the saving of unpromising pairs during the greedy merge. In comparison, our *Mags* uses the candidate generation phase to focus on promising (candidate) pairs rather than unpromising ones, and uses a new design of greedy merge phase to reduce both the times of saving updates and the updates of unpromising node pairs.

Different from *Greedy*'s initialization phase, the candidate generation phase of *Mags* samples a *pre-set* number (i.e., $k$) of high-saving candidate pairs for each node, instead of taking any node in 2-hops as a candidate. This avoids the unpromising pair issue in *Greedy*, because we ensure that the number of candidate pairs is up to $k \cdot n$ while still retaining a sufficient number of promising pairs. Nevertheless, it is challenging to select promising candidate pairs from all 2-hop-apart pairs. To this end, we design an efficient algorithm for generating candidate pairs (Algorithm 2).

Compared with *Greedy*, *Mags* largely reduces the times of saving updates and the number of pairs to update in each saving update. Recall that *Greedy* updates the saving of all 2-hop-apart pairs (up to $n \times {d_{avg}}^2$ pairs) after each merge (up to $n$ times). Meanwhile, *Mags* updates the saving of candidate pairs (up to $n \times k$ pairs) in each iteration (up to $T$ times). As a result, *Mags* is much faster than *Greedy* while it keeps the high-quality merges of *Greedy*.

In what follows, we detail the candidate generation phase of *Mags*, followed by the greedy merge phase, and then the output phase.

## 3.1 Candidate Generation Phase

We aim to generate $k$ pairs for each node $u$ such that they contain $u$ and provide a high saving. A naive approach is to compute saving $s(u, v)$ for any $v$ from $u$'s 2-hop neighborhood, and then select the top-$k$ pairs with the highest saving. This approach takes $O(n \cdot d_{avg}^2 \cdot (d_{avg} + \log k))$ time. In particular, for any node $u$, we iterate over any $v$ that is 2-hop apart from $u$, and then compute saving $s(u, v)$ in $O(d_{avg})$ time and insert the pair into a heap in $O(\log k)$ time, where the heap maintains

**Algorithm 2**: CandidateGeneration ($\mathcal{G}, k$)

```
1  Initialize $h$ hash functions $f^{(1)}, \cdots, f^{(h)}$;
2  Compute MinHash $f_{\min}^{(i)}(u)$ for each $u \in \mathcal{V}$ and each $i = 1..h$;
3  Initialize a candidate pair set $CP \longleftarrow \varnothing$;
4  for each $u \in \mathcal{V}$ do
5      Sample a random subset $S$ of $b$ nodes from $\mathcal{N}_u$;
6      Let $2Hop \longleftarrow \mathcal{N}_u \cup \bigcup_{w \in S} \mathcal{N}_w$;
7      for each $v \in 2Hop$ do
8          Compute $mh(u, v)$ by MinHash (Equation 5);
9      Select $k$ nodes with the highest $mh(u, v)$;
10     Put $(u, v)$ into $CP$ for each selected $v$;
11 return $CP$;
```

the top-$k$ pairs. Such a naive approach is cost-prohibitive, so we design an improved algorithm that can drastically speed up the candidate generation.

Algorithm 2 presents the candidate generation phase of *Mags*. We let $\mathcal{N}_u$ be the neighbor set of a node $u$ in the original graph, i.e., $\mathcal{N}_u = \{v \mid (u, v) \in \mathcal{E}\}$. Given a node $u$, instead of considering all 2-hop neighbors, we sample a subset of 2-hop neighbors ($2Hop$ in Line 6) that are likely to provide a high saving when merging with $u$. After that, we use a MinHash-based [6] scoring $mh(u, v)$ to select $k$ nodes from $2Hop$, and then insert $(u, v)$ into $CP$ for any selected $v$ (Lines 9-10). In other words, Algorithm 2 speeds up the candidate generation by two strategies: (i) sampling promising nodes from the 2-hop neighbors, and (ii) approximating the exact saving by the Jaccard similarity and MinHash (Jaccard is highly correlated with saving as we show later).

Intuitively, Algorithm 2 works as follows. For each node $u$, we sample $k$ nodes with the largest value of $mh(u, v)$ from the 2-hop neighbors of $u$ (i.e. $2Hop$ in Algorithm 3), where $mh(u, v)$ is the empirical probability that $u$ and $v$ will have the same MinHash value of neighbor set among $h$ hash functions. This empirical probability is an unbiased estimation of the Jaccard similarity between the neighbor sets of $u$ and $v$. Therefore, these $k$ nodes are very likely to be similar to $u$, and we add a candidate pair $(u, v)$ for any $v$ from these $k$ nodes.

In Lines 5-6, we randomly sample $b$ neighbors of $u$ (i.e., $S$). Then, we unionize $u$'s neighbor set and the neighbor sets of each node in $S$, and take this union (i.e., $2Hop$) as an approximation of $u$'s 2-hop neighbors. Compared with directly obtaining the 2-hop neighbors of $u$, the sampling above reduces the number of neighbor sets to unionize from $|\mathcal{N}_u| + 1$ to $b + 1$. Meanwhile, the sampling will retain the nodes whose neighbor set is similar to $u$ with a high probability, and we formalize this idea with the following theorem:

Theorem 1. *Given two nodes $u$ and $v$, we assume that the Jaccard similarity between $\mathcal{N}_u$ and $\mathcal{N}_v$ equals to $p$. Let $2Hop(x)$ be the $2Hop$ (Algorithm 2 Line 6) when $u = x$. Then, the following condition holds with at least $1 - (1 - p)^{2b}$ probability:*

$$u \in 2Hop(v) \quad \vee \quad v \in 2Hop(u).$$

Proof. The Jaccard similarity between $\mathcal{N}_u$ and $\mathcal{N}_v$ is defined as $J(\mathcal{N}_u, \mathcal{N}_v) = |\mathcal{N}_u \cap \mathcal{N}_v| / |\mathcal{N}_u \cup \mathcal{N}_v| = p$, and it follows that $|\mathcal{N}_u \cap \mathcal{N}_v| / |\mathcal{N}_u| \geq p$. If we sample a node $w$ from $\mathcal{N}_u$, then $v \in \mathcal{N}_w$ (i.e., $w \in |\mathcal{N}_u \cap \mathcal{N}_v|$) holds with probability at least $p$. If we sample a subset $S$ of $h$ nodes from $\mathcal{N}_u$, then we have $v \in 2Hop(u)$ (i.e., $v \in \bigcup_{w \in S} \mathcal{N}_w$) with probability at least $1 - (1 - p)^b$. Similarly, we have $u \in 2Hop(v)$ with probability at least $1-(1-p)^b$. Therefore, the condition $u \in 2Hop(v)$ or $v \in 2Hop(u)$ holds with at least $1-(1-p)^{2b}$ probability. □

By Theorem 1, given two nodes $u$ and $v$, if the Jaccard similarity between $\mathcal{N}_u$ and $\mathcal{N}_v$ is non-trivial, then Algorithm 2 identifies $(u,v)$ as a candidate pair (Lines 5-6) with a high probability. We set $b=5$ empirically, as it retains a pair whose Jaccard similarity is 0.2 with at least 89% probability. Our sampling strategy is effective in practice as the probability bound in Theorem 1 is loose, e.g., $|\mathcal{N}_u \cap \mathcal{N}_v|/|\mathcal{N}_u|$ can be much larger than $p$, and the nodes similar to $u$ are likely to be $u$'s neighbors (i.e., found by $\mathcal{N}_u$ in Line 6).

Next, we show the correlations between Jaccard similarity and saving (Equation 4). Given a pair of super-nodes $(u,v)$, the Jaccard similarity between $\mathcal{N}_u$ and $\mathcal{N}_v$ equals $J(\mathcal{N}_u,\mathcal{N}_v) = |\mathcal{N}_u \cap \mathcal{N}_v|/|\mathcal{N}_u \cup \mathcal{N}_v|$. Assume that $u$ and $v$ both contain one single node. By definition (Equation 2), it follows that $c_u = |\mathcal{N}_u|$ and $c_v = |\mathcal{N}_v|$. When we merge $u$ and $v$ into a new node $w$, then $c_w$ equals $|\mathcal{N}_u \cup \mathcal{N}_v| - 1$ if $u \in \mathcal{N}_v$, and $|\mathcal{N}_u \cup \mathcal{N}_v|$ otherwise. If roughly taking $c_w \approx |\mathcal{N}_u \cup \mathcal{N}_v|$, then we can rewrite Equation 4 as

$$s(u,v) = \frac{c_u + c_v - c_w}{c_u + c_v} \approx \frac{|\mathcal{N}_u| + |\mathcal{N}_v| - |\mathcal{N}_u \cup \mathcal{N}_v|}{|\mathcal{N}_u| + |\mathcal{N}_v|} = \frac{|\mathcal{N}_u \cap \mathcal{N}_v|}{|\mathcal{N}_u| + |\mathcal{N}_v|}.$$

The r.h.s of the above equation is the Dice similarity of $\mathcal{N}_u$ and $\mathcal{N}_v$ multiplied by two. The Dice and Jaccard similarity can approximate each other relatively and absolutely [16], so Jaccard is highly correlated with saving. Therefore, *Mags* selects $k$ nodes (Lines 9-10) with the MinHash, a technique for estimating the Jaccard similarity.

**MinHash Background.** The MinHash [6] is a type of Locality Sensitive Hashing (LSH) [19] that can quickly estimate the Jaccard similarity between sets. Given a hash function $f(\cdot)$ that maps a node to an integer, we define the MinHash of $u$ as the minimum hash value among $u$'s neighbors, i.e., $f_{\min}(u) = \min_{v \in \mathcal{N}_u}\{f(v)\}$. Notably, the Jaccard similarity between two sets is equal to the probability of identical MinHash, that is, $\Pr\{f_{\min}(u) = f_{\min}(v)\} = J(\mathcal{N}_u,\mathcal{N}_v)$.

Therefore, we can estimate Jaccard by the *empirical* probability of identical MinHash. In particular, given $h$ hash functions $f^{(1)},\cdots,f^{(h)}$, we define $mh(u,v)$ as the empirical probability of identical MinHash among $h$ functions, i.e.,

$$mh(u,v) \quad = \quad \frac{1}{h}\sum_{i=1}^{h}\left[f_{\min}^{(i)}(u) = f_{\min}^{(i)}(v)\right]. \tag{5}$$

Here, the notion $[\cdot]$ returns 0 if the condition within is true, and 0 otherwise. By the fact that $\Pr\{f_{\min}(u) = f_{\min}(v)\} = J(\mathcal{N}_u,\mathcal{N}_v)$, it follows that the expectation $\mathbb{E}\left[mh(u,v)\right] = J(\mathcal{N}_u,\mathcal{N}_v)$, and we can estimate the Jaccard similarity by $mh(u,v)$ without bias.

**Usage of MinHash and $mh(\cdot)$.** Given $h$, we initialize $h$ hash functions $f^{(1)},\cdots,f^{(h)}$ (Line 1), and each function is a permutation of $1,\cdots,n$. Then, we iterate over every hash function $f^{(i)}$, and compute the MinHash $f_{\min}^{(i)}(u)$ for each node $u$ (Line 2). After that, Line 8 can compute $mh(\cdot)$ with the MinHash according to Equation 5, where each estimation of $mh(\cdot)$ takes $O(h)$ time.

The purpose of LSH design in *Mags* is different from existing solutions. In short, *Mags* uses LSH to generate high-quality candidate node pairs for the greedy merge, while *LDME* [45] (and *SWeG* [34]) uses LSH to divide nodes into disjoint groups and then merge the nodes in each group with Super-Jaccard. The LSH in our *Mags* can help produce more compact summaries, because *Mags* will output a summary with similar compactness to *Greedy* as long as *Mags* can generate high-quality candidate pairs with LSH and greedily merge the candidate pairs (like *Greedy*).

**Algorithm 3**: GreedyMerge ($\mathcal{G}, T, CP$)

```
1  $P \leftarrow \{\{u\} \mid u \in \mathcal{V}\}$;   $H \leftarrow$ a priority queue (key=saving);
2  for each $(u, v) \in CP$ do Insert $(s(u, v), u, v)$ into $H$;
3  for $t = 1..T$ do
4      for each $(s, u, v) \in H$ in deceasing $s$ do
5          Go on when $s \geq \omega(t)$, otherwise break;
6          if renewed $s(u, v) \geq \omega(t)$ then
7              Merge $u$ and $v$ into $w$ in $P$, where $w \leftarrow u \cup v$;
8              Replace $u$ and $v$ by $w$ in $CP$ and $H$;
9      Let $U$ be the union of $\{w\} \cup N_w$ for any $w$ from Line 7;
10     for each $u \in U$ do
11         for each $(u, v) \in CP$ do
12             Remove $(s, u, v) \in H$, and insert back $(s(u, v), u, v)$;
13 return $P$;
```

Theorem 2. *Given small constants $b$ and $h$, and we set $k = c \cdot d_{avg}$ where $c$ is constant, Algorithm 2 runs in $O(m \cdot \log d_{avg})$ time.*

Proof. Lines 1-2 take $O(hm)$ time, as it initializes $h$ hash functions (node permutations) and computes the MinHash of each node in $O(m)$ time for each function. Lines 5-6 run in $O(bn \cdot d_{avg})$ time. Notice that we union $b + 1$ neighbor sets for each node (Line 6), and the expected size of $2Hop$ is $(b + 1) \cdot d_{avg}$. After that, Lines 7-10 require $O(bn \cdot d_{avg} \cdot (h + \log k))$ time. That is, for each $u$ and each $v \in 2Hop$, we compute $mh(u, v)$ and maintain the top-$k$ by a heap in $O(h + \log k)$ time. Overall, the time complexity is $O(hm + bn \cdot d_{avg} \cdot (h + \log k))$. By setting $k = c \cdot d_{avg}$, and $b$ and $h$ as small constants, we can rewrite the time complexity as $O(m \cdot \log d_{avg})$. □

### 3.2 Greedy Merge Phase

Algorithm 3 presents the greedy merge phase of *Mags*. Given $\mathcal{G}$, the number of iterations $T$, and a set of candidate pairs $CP$, the algorithm aims to return a set of super-nodes $P$ that minimizes the representation cost. Lines 1-2 initialize a set of super-nodes $P$ where each super-node contains a single node, and a priority queue $H$ that contains every candidate pair and its saving. The subsequent part of the algorithm consists of $T$ iterations (Lines 3-12), and each iteration contains two parts:

The first part of the iteration (Lines 4-8) merges a set of candidate pairs whose saving is larger than $\omega(t)$ (this threshold is defined later). We merge pairs in decreasing order of saving $s$ (Line 4). As we defer saving updates to the second part of the iteration (Lines 9-12), the saving stored in $H$ ($s$ in Line 5) may be *out-of-date* due to the previous merges. To avoid this issue, if and only if the saving since the last update ($s$ in Line 5) and the renewed saving $s(u, v)$ in Line 6 are both larger than $\omega(t)$, we merge the pair in $P$, $CP$ and $H$ (Lines 7-8). Note that if a qualified $(u, v)$ pair ($s(u, v) > \omega(t)$) is skipped by Line 5 due to the old $s$ value, it will still be considered in the later iterations as $\omega(t)$ decreases.

In the second part of the iteration (Lines 9-12), we update saving for the candidate pairs that are affected by the merges in the first part. Assume that $u$ and $v$ are merged into a new super-node $w$. Similar to *Greedy*, the saving of a pair $(x, y)$ may change when $x$ or $y$ is contained in $w$'s neighborhood, i.e., $\{w\} \cup N_w$. We put $\{w\} \cup N_w$ into $U$ for any newly formed $w$ (Line 9), and then update saving for any candidate pair containing $u \in U$ (Lines 10-12).

**Algorithm 4**: Output $(\mathcal{G}, P)$

```
1 $E \leftarrow \emptyset$;  $C \leftarrow \emptyset$;
2 for each super-node $u \in P$ do
3     for each super-node $v \in N_u$ (i.e., $E_{uv} \neq \emptyset$) do
4         if $|E_{uv}| > (|\Pi_{uv}| + 1)/2$ then
5             Add '$-e$' to $C$ for $e \in \Pi_{uv} \setminus E_{uv}$;  Add $(u, v)$ to $E$;
6         else Add '$+e$' to $C$ for $e \in E_{uv}$;
7 return $R \leftarrow (S, C)$ where $S = (P, E)$;
```

**Merge Threshold.** Equation 6 formulates the *merge threshold* $\omega(t)$ used in Algorithm 3. We set $\omega(1) = 0.5$, as the saving $s(u, v)$ is equal to (and is up to) 0.5 when $u$ and $v$ have the identical neighbor set (Equation 4). We empirically set $\omega(T) = 0.005$, as it implies that $u$ and $v$ only share 1% of their neighbors. When $1 < t < T$, we set up a geometric sequence with a ratio $r = \sqrt[T-1]{0.01}$.

$$\omega(t) = \begin{cases} 0.5 \times r^{t-1} & \text{if } t < T, \text{ where } r = \sqrt[T-1]{0.01} \\ 0.005 & \text{if } t = T \end{cases} \tag{6}$$

The merge threshold gradually decreases as $t$ increases, e.g., when $T = 50$, $\omega(t)$ forms a sequence $0.5, 0.455, 0.414, \cdots, 0.005$ ($r \approx 0.912$ in Equation 6). Recall that we only merge a pair whose saving is larger than $\omega(t)$ in the $t$-th iteration. Using our well-designed merge threshold, *Mags* can merge the high-saving pairs with a moderate order of decreasing thresholds.

Theorem 3. *Given we set $k = c \cdot d_{avg}$ where $c$ is a small constant, Algorithm 3 runs in $O(T \cdot m \cdot (d_{avg} + \log m))$ time complexity.*

Proof. Lines 1-2 require $O(|H| \cdot (d_{avg} + \log |H|))$ time, as it takes $O(d_{avg})$ time to compute the saving of a candidate pair and $O(\log |H|)$ time to put the pair into $H$. After that, the $T$ iterations take $O(T \cdot |H| \cdot (d_{avg} + \log |H|))$ time. Specifically, in each iteration, we re-compute the saving of any pair in $H$ (Lines 6 and 12) by iterating over the neighbors (one hop), and then update the saving in $H$ using $O(\log |H|)$ time (Line 12). These saving computations take $O(T \cdot |H| \cdot (d_{avg} + \log |H|))$ time. The cost of merges (Lines 7-8) is $O(n \cdot (d_{avg} + k))$, as there are up to $n$ merges and each merge requires $O(d_{avg} + k)$ time. The overall time is $O(T \cdot |H| \cdot (d_{avg} + \log |H|))$. By setting $k = c \cdot d_{avg}$, and the fact that $|H| \leq k \cdot n = c \cdot d_{avg} \cdot n = c \cdot m$, we can rewrite the time complexity to $O(T \cdot m \cdot (d_{avg} + \log m))$. □

## 3.3 Output Phase

Algorithm 4 decides the optimal $R$ from $P$ (see Section 2.2). That is, if and only if $|E_{uv}| > (|\Pi_{uv}| + 1)/2$, we add the super-edge $(u, v)$ to $E$ and add correction '$-e$' for each $e \in \Pi_{uv} \setminus E_{uv}$; otherwise, we add correction '$+e$' for each $e \in E_{uv}$.

Theorem 4. *Algorithm 4 runs in $O(m)$ time.*

Proof. The number of operations in Lines 4-6 equals $2 \times c(R)$, because each super-edge (resp. each correction) has at most 2 copies. By the fact that the representation cost is no more than the original cost (i.e., $c(R) \leq m$), Algorithm 4 takes $O(m)$ running time. □

## 3.4 Putting Them Together

In summary, *Mags* algorithm works as follows. *Mags* needs two parameters $k$ and $T$, where $k$ is decided from $d_{avg}$, and $T$ is user-defined. Given $\mathcal{G}$, $k$, and $T$, *Mags* first feeds $\mathcal{G}$ and $k$ as input to Algorithm 2, and obtains a set of candidate pairs $CP$ in return. Then, *Mags* gives $\mathcal{G}$, $T$, and $CP$ as input to Algorithm 3, and receives a set of super-nodes $P$. Finally, *Mags* decides the best representation $R$ from $P$ using Algorithm 4, and returns $R$ as the final result.

By Theorems 2, 3, and 4, *Mags* takes an $O(T \cdot m \cdot (d_{avg} + \log m))$ running time when we set parameters according to the theorems. In practice, we set $b = 5$ and $h = \min\{10 \cdot d_{avg}, 50\}$ for Algorithm 2, and set $k \leftarrow \min\{5 \cdot d_{avg}, 30\}$ for Algorithm 3. Note that these parameters have a limited impact on the result, as shown in Section 6.5. The only parameter that users need to specify is $T$, and a larger $T$ implies a more compact summary and higher time overhead.

Recall that *Greedy* [30] runs in $O(n \cdot {d_{avg}}^3 \cdot (d_{avg} + \log m))$ time and it provides the highest summary compactness. On the theory side, our *Mags* improves the time complexity to $O(T \cdot m \cdot (d_{avg} + \log m))$. On the practice side, *Mags* can scale to billion-scale graphs while providing a summary as compact as *Greedy* (Section 6.3).

## 4 PROPOSED DIVIDE-AND-MERGE METHOD

Shin et al.'s divide-and-merge solution [34] (i.e., *SWeG*) runs in practical time but the summary of which is less compact. In what follows, we first revisit *SWeG* and present our insights on the method. After that, we describe our proposed *Mags-DM* method.

Recall in Section 2.4 that *SWeG* runs in $T$ iterations, and the $t$-th iteration consists of two phases. The dividing phase generates a hash function $f(\cdot)$ and computes the MinHash $f_{\min}(u)$ for each super-node $u$, then it divides super-nodes into groups $S^{(1)}, \cdots, S^{(d)}$ by the MinHash value (see Section 3.1 for MinHash). The merging phase works as follows. For each group $S^{(i)}$, it iteratively removes a random node $u \in S^{(i)}$ from the group, and picks the node $v \in S^{(i)}$ with the highest Super-Jaccard similarity to $u$, and then merge $(u, v)$ if its saving $s(u, v)$ is larger than a merge threshold $\theta(t) = 1/(1 + t)$.

Despite *SWeG*'s efficiency, the method is sub-optimal in summary compactness. Recall that *SWeG* divides nodes into groups by MinHash in the dividing phase, and selects promising pairs by Super-Jaccard in the merging phase. MinHash and Super-Jaccard are both *approximate measures* of saving, and their precision is critical to the summary compactness of the algorithm. To enhance these approximate measures in precision and efficiency, our *Mags-DM* substantially improves *SWeG* in dividing and merging strategies:

- **Merging Strategy 1 (Node Selection)**. During the merging phase, given a group and a random node $u$ in the group, *SWeG* picks the node $v$ that is the most similar to $u$, and then attempt to merge the pair $(u, v)$. In comparison, *Mags-DM* selects $b$ nodes that are the most similar to $u$, and then the node $v$ that provides the largest saving $s(u, v)$ among the $b$ nodes. This improves the summary compactness of the algorithm.
- **Merging Strategy 2 (Similarity Measure)**. *Mags-DM* uses a MinHash-based similarity measure $mh(\cdot)$ (Eq. 5) instead of *SWeG*'s Super-Jaccard. This new measure provides a more compact summary and is faster to compute.
- **Merging Strategy 3 (Merge Threshold).** *Mags-DM* adopts our merge threshold $\omega(t)$ (Section 3.2) rather than *SWeG*'s $\theta(t)$, and $\omega(t)$ provides a more compact summary.
- **Dividing Strategy**: We divide the nodes with a set of hash functions and ensure that the maximum size of each group is not large, while *SWeG* divides nodes by a single hash function. This improves the efficiency of the algorithm on large graphs.

**Algorithm 5**: *Mags-DM* ($\mathcal{G}, T$)

```
1  Initialize $h$ hash functions $f^{(1)}, \cdots, f^{(h)}$;
2  Compute MinHash $f^{(i)}_{\min}(u)$ for each $u \in \mathcal{V}$ and each $i = 1..h$;
3  Initialize a set of super-nodes $P \leftarrow \{\{u\} \mid u \in \mathcal{V}\}$;
4  for $t = 1..T$ do
5      Divide the nodes in $P$ into $S^{(1)}, \cdots, S^{(d)}$ by MinHash;
6      for each $S^{(i)} \in \{S^{(1)}, \cdots, S^{(d)}\}$ do
7          while $S^{(i)}$ is non-empty do
8              Pick (and remove) a random node $u$ from $S^{(i)}$;
9              Let $Q$ be the $b$ nodes from $S^{(i)}$ whose $mh(u, w)$ (Equation 5) is the largest for any
               $w \in Q$;
10             Let $v \leftarrow \arg\max_{v \in Q} s(u, v)$;
11             if $s(u, v) \geq \omega(t)$ then
12                 Merge $u$ and $v$ into $w$ in $P$, where $w \leftarrow u \cup v$;
13                 for $i = 1..h$ do $f^{(i)}_{\min}(w) \leftarrow \min\{f^{(i)}_{\min}(u), f^{(i)}_{\min}(v)\}$;
14 Decide the optimal $R$ from $P$;                                      // Alg. 4
15 return $R$;
```

### 4.1 Our Method: *Mags-DM*

This section presents *Mags-DM*, a graph summarization method built on the paradigm of *SWeG* but aims to improve its performance. Algorithm 5 overviews the pseudo-code of *Mags-DM*. The algorithm runs $T$ iterations (Line 4), and the $t$-th iteration contains two phases:

1. **Dividing (Line 5)**: This phase divides the nodes into groups $S^{(1)}, \cdots, S^{(d)}$ by the MinHash values of multiple hash functions, such that the size of each group is not large. We initialize hash functions and the MinHash before the iterations (Lines 1-2), and dynamically update them during the merges (Line 13).

2. **Merging (Lines 6-13)**: This phase iterates over each group. For each group $S^{(i)}$, it iteratively picks a random node $u$, then selects $b$ nodes that are the most similar to $u$ by $mh(u, w)$ (Equation 5) and the node $v$ with the largest saving from the $b$ nodes. After that, we merge $u$ and $v$ if the saving $s(u, v) > \omega(t)$, where $\omega(t)$ is the merge threshold (Eq. 6).

**Usage of MinHash.** The algorithm first initializes $h$ hash functions and the MinHash of functions in Lines 1-2 (see Section 3.1 for MinHash), where each hash function is a permutation of $1, \cdots, n$. Recall that $mh(\cdot)$ (Equation 5) relies on the MinHash values of $h$ hash functions, and the MinHash is unknown for a newly merged super-node $w$ (Line 12). In this case, we define the MinHash of a super-node $w$ as $f_{\min}(w) = \min_{x \in P_w}\{f_{\min}(x)\}$, and maintain it during the merges. That is, when we merge $u$ and $v$ into $w$, we take the minimum value of the MinHash of $u$ and $v$, and set it to the MinHash of $w$ for every hash function (Lines 13-13).

**Procedure of Dividing Phase.** In Line 5, we first generate a random permutation of $f^{(1)}, \cdots, f^{(h)}$. Let the id of those functions be $s_1, \cdots, s_h$. Then, we group any node $u \in P$ by the MinHash value $f^{(s_1)}_{\min}(u)$. For those groups whose size is larger than a constant $M$ (spec., $M = 500$), we recursively divide the groups using functions $f^{(s_2)}(\cdot), \cdots, f^{(s_{10})}(\cdot)$, until the size of each group is below $M$.

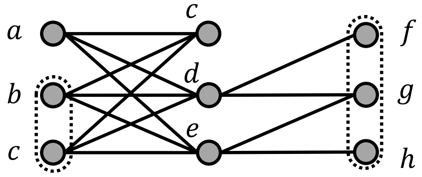


Fig. 3. Super-Jaccard would merge $\{b, c\}$ with $\{f, g, h\}$, while our $mh(\cdot)$ would merge $\{b, c\}$ with $\{a\}$

Note that the recursion depth is limited to a constant (i.e., 10). Such a setting is also adopted in previous works [24, 25].

**Procedure of Merging Phase.** During the merging phase, given a group $S^{(i)}$ and a random node $u$ from $S^{(i)}$, we identify a node $v$ that is the most similar to $u$ as follows. Specifically, we first compute $mh(u, w)$ by Equation 5 for any $w \in S^{(i)}$, then pick the $b$ nodes with the largest $mh(u, w)$, and put them into a set $Q$ (Line 9). After that, we iterate over any $v \in Q$ and pick the node $v$ with the largest value of saving $s(u, v)$ (Line 10). In the $t$-th iteration, we merge $(u, v)$ if and only if $s(u, v)$ is larger than a merge threshold $\omega(t)$.

In Line 11, we use our merge threshold $\omega(t)$ (see Equation 6) rather than *SWeG*'s $\theta(t)$ [34] , as our $\omega(t)$ provides a more compact summary. To explain, $\theta(t) = 1/(1 + t)$ forms a sequence $0.5, 0.33, 0.25, \cdots, 0.0196$ when $T = 50$, while our merge threshold $\omega(t)$ forms $0.5, 0.455, 0.414, \cdots, 0.005$ ($r \approx 0.912$ in Equation 6). Intuitively, the algorithm is more likely to merge promising pairs before unpromising ones when we use $\omega(t)$, as $\omega(t)$ decreases more slowly for small $t$. For example, assume there are three nodes $u, v, w$ where $s(u, v) = 0.46$, $s(u, w) = 0.34$, and $s(u \cup v, w) = -0.05$. Clearly, it is desired to merge $(u, v)$ and avoid merging $(u, w)$. Due to $\theta(2) = 0.33$ and the dividing strategy, *SWeG* may merge $(u, w)$ before $(u, v)$ for any $t \geq 2$. Meanwhile, by $\omega(2) = 0.455$ and $\omega(6) = 0.313$, our *Mags-DM* will only consider merging $(u, v)$ when $2 \leq t \leq 5$. Our $\omega(t)$ thus provides a more compact summary compared with *SWeG*'s $\theta(t)$.

**Comparison of Super-Jaccard and $mh(\cdot)$.** Given a super-node $u$, we let $\mathcal{N}_u$ be the neighbor node set of $u$ in the original graph, i.e., $\mathcal{N}_u = \{y \mid x \in P_u \wedge (x, y) \in \mathcal{E}\}$. Then, given super-nodes $u$ and $v$, the Super-Jaccard [34] of $u$ and $v$ is defined as

$$SJ(u, v) \quad = \quad \frac{\sum_{x \in \mathcal{N}_u \cup \mathcal{N}_v} \min\{w(u, x), w(v, x)\}}{\sum_{x \in \mathcal{N}_u \cup \mathcal{N}_v} \max\{w(u, x), w(v, x)\}}, \tag{7}$$

where $w(u, x) = |\{y \in P_u \mid \{x, y\} \in \mathcal{E}\}|$ is the number of nodes in super-node $u$ that is adjacent to a node $x \in \mathcal{V}$ in the original graph. The Super-Jaccard is a weighted Jaccard similarity of $\mathcal{N}_u$ and $\mathcal{N}_v$, where the weight is specified by $w(u, x)$ and $w(v, x)$. In comparison, our $mh(u, v)$ (Eq. 5) is an unbiased estimator of $J(\mathcal{N}_u, \mathcal{N}_v)$, that is, the (unweighted) Jaccard similarity between $\mathcal{N}_u$ and $\mathcal{N}_v$.

In *Mags-DM*, we replace the Super-Jaccard by our $mh(\cdot)$, because the Super-Jaccard is biased to the super-nodes that contain more nodes. In the example below, we show this drawback of the Super-Jaccard and how it may impair algorithm performance.

**Example 2.** Assume that we aim to merge the super-node $\{b, c\}$ with some other super-nodes in Figure 3. The best choice is to merge $\{b, c\}$ with $\{a\}$ because they have the same connectivity. But the Super-Jaccard prefers $\{f, g, h\}$ more, as we have $SJ(\{b, c\}, a) = 3/6$ and $SJ(\{b, c\}, \{f, g, h\}) = 4/6$. Note that $\{f, g, h\}$ contains three nodes and this leads to a weight larger than $\{a\}$. In comparison, the unweighted Jaccard and our $mh(\cdot)$ prefers $\{a\}$, that is, $J(\{b, c\}, a) = 3/3$ and $J(\{b, c\}, \{f, g, h\}) = 2/3$. In summary, the Super-Jaccard tends to prioritize some super-nodes with a large number of nodes, while our $mh(\cdot)$ can provide a better result compared with the Super-Jaccard in practice.

Theorem 5. *Given small constants $b$ and $h$, Algorithm 5 runs in $O(T \cdot m)$ time.*

Proof. Lines 1-2 take $O(hn)$ time to initialize $h$ hash functions and $O(hm)$ time for the MinHash. The dividing phase (Line 5) requires $O(n)$ time, as $P$ contains at most $n$ super-nodes and the MinHash values are available. Let $s_{max}$ be the maximum size of the group (Line 6), i.e., $s_{max} = \max\{|S^{(i)}|, \cdots, |S^{(d)}|\}$. During the merge phase (Lines 6-13), for each $u$, Line 9 computes $mh(u, w)$ for any $w \in S^{(i)}$ and maintain the top-$b$ in $O(s_{max} \cdot (h + \log b))$ time, then Line 10 takes $O(b \cdot d_{avg})$ time to compute the saving for any $v \in Q$ ($|Q| \leq b$), and finally the merge of $(u, v)$ requires $O(h)$ time (Lines 11-13). Given $u$ (Line 8), Lines 9-13 run in $O(s_{max} \cdot (h + \log b) + b \cdot d_{avg} + h)$ time, and we can rewrite it to $O(h + b \cdot d_{avg})$ if we divide $P$ finely such that the maximum size of each group (i.e., $s_{max}$) is smaller than a constant. The merge phase will process at most $n$ different $u$ (Line 8), thus, the complexity of it is $O(n \cdot (h + b \cdot d_{avg}))$. The output phase (Line 14) takes $O(m)$ time by Theorem 4. After summing up the cost of initialization, dividing, merging, and output, the total time complexity of Algorithm 5 equals $O(hm + Tn \cdot (h + b \cdot d_{avg}))$. By setting $b$ and $h$ as constants, we can rewrite the complexity to $O(T \cdot m)$ time. □

## 4.2 Putting Them Together

Our *Mags-DM* needs a user-defined parameter $T$. Given $T$, Algorithm 5 takes $T$ as an input, and returns $R$ as the final result. By Theorem 5, *Mags-DM* runs in $O(T \cdot m)$ time complexity when we set the parameters according to the theorem. In practice, we can set $b = 5$ and $h = 40$ for Algorithm 5. The feasible values of these parameters are within a wide range and they have a limited impact on the performance, as shown in Section 6.5.

*LDME* [45] and *Slugger* [25] are the state-of-the-art works using the divide-and-merge paradigm. *LDME* and *Slugger* are built on *SWeG* [34] and they both run in $O(T \cdot m)$ time. In particular, *LDME* uses a weighted LSH to find good node merges, while *Slugger* represents a graph using hierarchical super-nodes that contain each other and it greedily merges nodes into super-nodes while exploiting their hierarchy. Different from these works, our *Mags-DM* improves *SWeG* with three novel merging strategies and one dividing strategy (see the beginning of Section 4). As a result, our *Mags-DM* can better divide nodes into promising groups and merge high-quality node pair, which leads to a more compact summary and a higher efficiency (see Section 6.4).

## 5 IMPLEMENTATION AND PARALLELISM

In this section, we detail the data structures used and the parallel implementations of our algorithms.

### 5.1 Implementation of *Mags*

**Data Structures of *Mags*.** We implement the set of super-nodes $P$ using disjoint set union [38]. For each super-node $u$, we store a table $W_u$ where $W_u(v)$ equals the numbers of edges between $u$ and $v$ (i.e., $|E_{uv}|$). During the merges, we update $W$ dynamically such that we can quickly compute $c_{uv}$ (Equation 2) and $s(u, v)$ (Equation 4) based on $W_u$ and $W_v$. The candidate pair set $CP$ is implemented as a series of tables, where $CP_u$ stores the candidate pairs containing $u$ and $CP_u(v)$ returns the saving of the pair $(u, v)$. As a result, we can quickly update the saving of a given pair $(u, v)$ in the priority queue $H$. That is, we can remove the pair from $H$ by $(CP_u(v), u, v)$ (Algorithm 3 Line 12), and then insert $(s(u, v), u, v)$ back into $H$. Each candidate pair $(u, v)$ is stored in both $CP_u$ and $CP_v$.

**Parallel Implementation of *Mags*.** *Greedy* [30] is hard to make parallel, as every greedy merge depends on the result of the previous one, forming a dependency chain (see Section 2.3). In comparison, our *Mags* processes a batch of merges and saving updates in each iteration, which makes *Mags* feasible to parallelize.

There are three steps in *Mags*. The first step to make parallel is candidate generation (Algorithm 2). We initialize each hash function (Line 1) and its MinHash (Line 2) in parallel, and then initialize

Table 2. Dataset Statistics

| Dataset | $n$ | $m$ | $d_{avg}$ | Type |
|---:|---:|---:|---:|---:|
| Caida (CA) | 26,475 | 53,381 | 4.0 | Internet |
| Email-Enron (EN) | 36,692 | 183,831 | 10.0 | E-Mail |
| Brightkite (BK) | 58,228 | 214,078 | 7.4 | Geo-Social |
| Email-Eu-All (EA) | 265,009 | 364,481 | 2.8 | E-Mail |
| Slashdot-0922 (SL) | 82,168 | 504,230 | 12.3 | Social |
| DBLP (DB) | 317,080 | 1,049,866 | 6.6 | Co-author |
| Amazon0601 (AM) | 403,394 | 2,443,408 | 12.1 | Co-purchase |
| CNR-2000 (CN) | 325,557 | 2,738,969 | 16.8 | Web |
| Youtube (YT) | 1,134,890 | 2,987,624 | 5.3 | Social |
| Skitter (SK) | 1,696,415 | 11,095,298 | 13.1 | Internet |
| IN-2004 (IN) | 1,382,867 | 13,591,473 | 19.7 | Web |
| EU-2005 (EU) | 862,664 | 16,138,468 | 37.4 | Web |
| Eswiki-2013 (ES) | 970,327 | 21,184,931 | 43.7 | Web |
| LiveJournal (LJ) | 3,997,962 | 34,681,189 | 17.3 | Social |
| Hollywood-2011 (HO) | 1,985,306 | 114,492,816 | 115.3 | Collaboration |
| Indochina-2004 (IC) | 7,414,758 | 150,984,819 | 40.7 | Web |
| UK-2005 (UK) | 39,454,463 | 783,027,125 | 39.7 | Web |
| IT-2004 (IT) | 41,290,648 | 1,027,474,947 | 49.8 | Web |

$W_u$ in parallel for each node $u$. After that, we generate candidate pairs for every node $u \in \mathcal{V}$ concurrently (Line 4), where we put a candidate pair $(u, v)$ into $CP_u$ and $CP_v$ with lock protection (Line 10). Then, we insert all candidate pairs into a priority queue $H$ serially. The second step is greedy merge (Algorithm 3). We first group all pairs with $s \geq \omega(t)$ by connectivity (Line 4), and then decide the pairs to merge concurrently for each group (Lines 6-8). During the merges among Lines 4-8, the update of $P$ is serial, the updates of $W_u$ and $W_v$ require lock protections, while we update $CP$ and $H$ in a batch after handling all node merges. Next, we update the saving for each $u \in U$ in parallel (Line 10). Given $u$ and $v \in CP_u$, we compute the saving $s(u, v)$ and update it in the candidate pair tables ($CP_u$ and $CP_v$) and the priority queue $H$. In practice, we only update $CP_u$ in real-time, as this leads to conflict-free updates for different $u \in U$. Then, we update $CP_v$ and $H$ in a batch at the end of the iteration. Finally, in the output phase, we build the super-edges for each super-node (Line 2 of Algorithm 4) in parallel.

## 5.2 Implementation of *Mags-DM*

**Data Structures of *Mags-DM*.** Like *Mags*, we implement the set of super-nodes $P$ using disjoint set union, and we maintain the number of edges between super-nodes (i.e., $W$) during the merges.

**Parallel Implementation of *Mags-DM*.** We make Algorithm 5 parallel as follows. Each hash function (Line 1) and its MinHash (Line 2) are initialized in parallel, while $W_u$ is initialized in parallel for each node $u$. During the dividing phase (Line 5), we use parallel sorting to order all nodes in $P$ by MinHash, and then segment the sequence into node groups with identical MinHash value. In the merging phase (Lines 6-13), we process different groups of super-nodes (Line 6) in parallel, while the updates of $P$ and $W$ are synchronized. Note that the updates of the MinHash are conflict-free, as different groups are disjoint from each other. In the output phase, we process each super-node in parallel (Algorithm 4 Line 2).

## 6 EXPERIMENTS

### 6.1 Experimental Setup

The source code of this paper is available on GitHub[1].

**Datasets.** Table 2 lists the datasets used in our experiments, and these datasets are widely used in the literature of graph summarization [22, 25, 34, 45]. Some of the datasets (CN, EU, IC, IT) are available from Network Repository[2], some (EW, HW, UK) are from LAW[3], while the rest are from SNAP[4]. We remove all edge directions, duplicated edges, and self-loops in the datasets.

**Algorithms.** We compare our algorithms with *Greedy* [30], *Slugger* [25], and *LDME* [45]. In particular, *Greedy* provides state-of-the-art summary compactness on small graphs, while *Slugger* and *LDME* are the state-of-the-art methods that scale to large graphs. We implement our algorithms and *Greedy* in C++, and we adopt the Java code open-sourced by the authors of *Slugger* and *LDME*.

**Parameters.** In each experiment, we repeat the method three times and report the average result. We terminate the program when it cannot finish in 24 hours. Unless otherwise specified, we use the following parameters in our experiments:

(1) *Mags*, *Mags-DM*: the parallel version (40 cores) with $T = 50$.
(2) *Greedy* [30]: no parameter.
(3) *LDME* [45]: $T = 50$ and $k = 5$ (signature length).
(4) *Slugger* [25]: $T = 50$.

**Compactness Measure.** Let $R$ be a representation that contains a summary graph $S = (P, E)$ and edge corrections $C$. We evaluate the compactness of $R$ by its *relative size* to the original edge set $\mathcal{E}$, i.e., $(|E| + |C|)/|\mathcal{E}|$. Recall that $|E| + |C|$ is the representation cost in Equation 1. This compactness measure is widely adopted in the literature of graph summarization [30, 34, 45] except for *Slugger* [25]. Note that *Slugger* [25] adopts a hierarchical graph summarization model, so it uses a different compactness measure. Given a graph $\mathcal{G}$, *Slugger* asks for a representation $R^H = (S, P^+, P^-, H)$, and the relative size of which equals $(|P^+| + |P^-| + |H|)/|\mathcal{E}|$ (see [25]).

**Environments.** We perform experiments on a server with two Intel Xeon Gold 6342 CPUs and 512GB memory, running Ubuntu 22.04 system. The server can run up to 96 threads concurrently. We implement algorithms (*Greedy* [30], *Mags*, and *Mags-DM*) in C++. The C++ code is compiled with g++ 10.2.0, and we use OpenMP for multi-threading programming. We compile the open-source Java code of *LDME* [45] and *Slugger* [25] with OpenJDK 11.

### 6.2 Results on Different Graphs

We divide graphs into small graphs (CA-DB) that *Greedy* can process in 24 hours, and large graphs (AM-IT) that *Greedy* runs out of time.

**Results on Small Graphs (CA-DB).** Figure 4 shows that *Greedy* consistently outperforms *LDME* and *Slugger* on small graphs, and it returns a summary that is 21.7% smaller than *LDME* and 30.2% smaller than *Slugger* on average. Our *Mags* and *Mags-DM* return a summary that is as compact as *Greedy*, and the difference in relative size is on average $< 0.1\%$ and 2.1% respectively. Figure 6 reports the running time on small graphs. In particular, *Mags* outperforms *Greedy* by 2-4 orders of magnitude in efficiency, outperforms *LDME* by 3.88x on average, and *Slugger* by 3.84x. Moreover, *Mags-DM* can further reduce the running time by 7.22x compared with *Mags*.

[1] https://github.com/nedchu/mags-release
[2] http://networkrepository.com
[3] https://law.di.unimi.it/datasets.php
[4] http://snap.stanford.edu

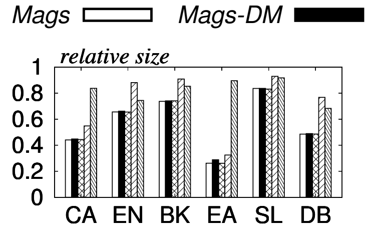


Fig. 4. Compactness on Small Graphs.

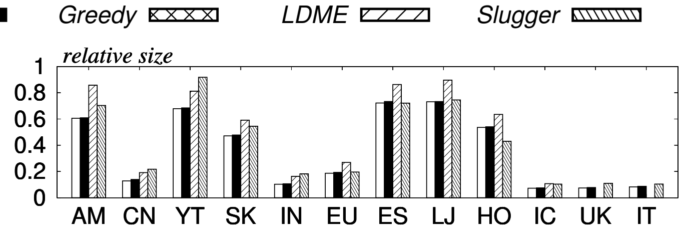


Fig. 5. Compactness on Large Graphs.

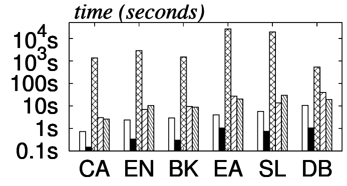


Fig. 6. Running Time on Small Graphs.

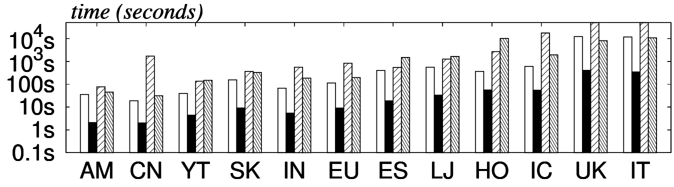


Fig. 7. Running Time on Large Graphs.

**Results on Large Graphs (AM-IT).** We omit the results of *Slugger* on datasets UK and IT, because *Slugger* cannot terminate within 24 hours. Figure 5 presents the summary compactness on large graphs. The figure shows that *Mags* significantly outperforms existing methods in compactness, and it can return a summary that is on average 24.9% smaller than *LDME*, 16.9% smaller than *Slugger*. At the same time, the summary compactness of *Mags* and *Mags-DM* is close, i.e., *Mags* outperforms *Mags-DM* by about 2.8%. Surprisingly, *Slugger* returns a summary more compact than *Mags* on dataset HO. The reason is that HO contains a 2208 clique and a hierarchy around the clique, and this structure well fits the hierarchical graph summarization model of *Slugger*. Figure 7 presents the running time on large graphs. On these graphs, *Mags* is on average 15.4x faster than *LDME* and 4.4x faster than *Slugger*, while *Mags-DM* can further improve the efficiency by 16.4x compared with *Mags*.

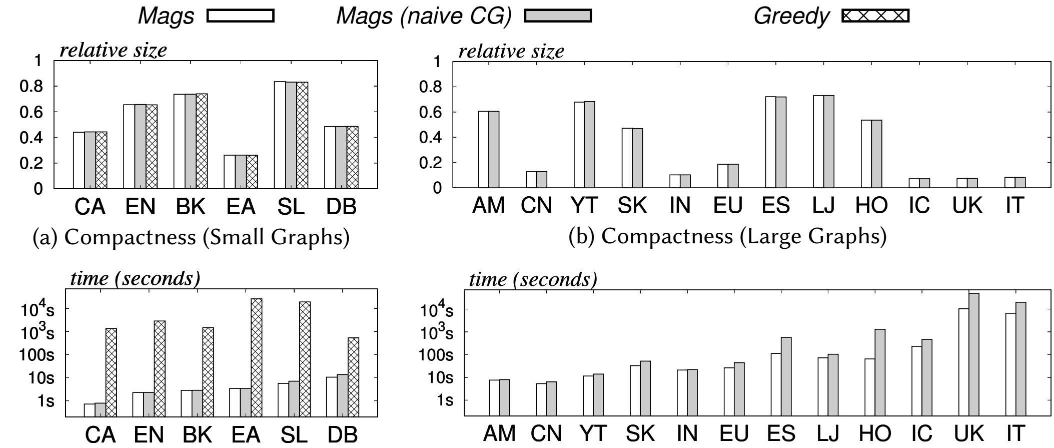
**Results on All Graphs.** Our *Mags* and *Mags-DM* return a summary as compact as *Greedy* on small graphs (the difference in compactness is $< 0.1\%$ and 2.1% respectively), while they are faster than *Greedy* by orders of magnitude. Compared with *LDME* (resp. *Slugger*), *Mags* returns a summary that is 23.7% (resp. 21.4%) more compact using 11.1x (resp. 4.2x) less time. When we compare *Mags* with *Mags-DM*, the latter takes 13.4x less time and returns a slightly larger summary (2.6% difference). In summary, *Mags* and *Mags-DM* achieve state-of-the-art compactness and efficiency, while existing works can only achieve state-of-the-art in one of them.

### 6.3 Technique Effectiveness of *Mags*

We compare the following three methods related to *Mags*, and report their performance in Figure 8.

(1) *Mags*: parallel version (40 cores) with $T = 50$.
(2) *Mags (naive CG)*: a variant of (1) where we use the naive candidate generation discussed at the start of Section 3.1.
(3) *Greedy* [30]: the baseline greedy algorithm.

**Technique from (3) to (2).** We first justify the technique in Section 3.2, where we avoid the unpromising pair and wasted update issues of *Greedy* with a novel algorithm design. Figure 8a shows that the compactness of our methods is close to *Greedy*, and the difference in compactness is <0.5% on average. According to Figure 8c, our methods outperform *Greedy* by 2-4 orders of



(a) Compactness (Small Graphs)

(b) Compactness (Large Graphs)

(c) Total Running Time (Small Graphs)

(d) Running Time of Candidate Generation (Large Graphs)

Fig. 8. *Mags* outperforms its variant and *Greedy*.

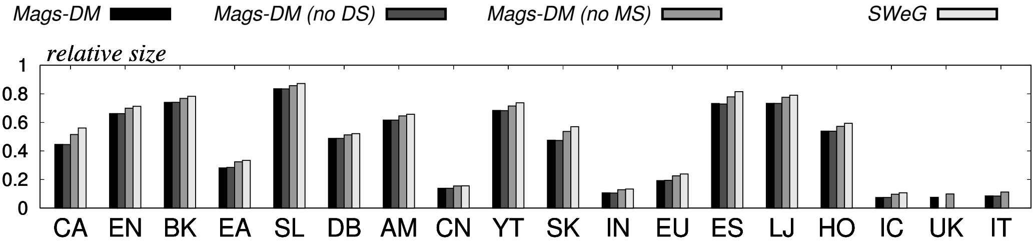


Fig. 9. *Mags-DM* outperforms its variants (Summary Compactness)

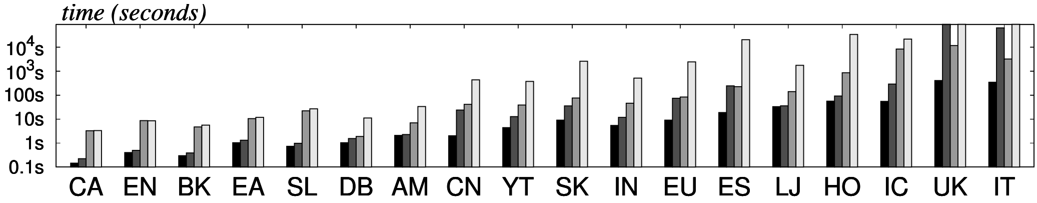


Fig. 10. *Mags-DM* outperforms its variants (Running Time)

magnitude in running time. Also, our methods can scale to large graphs that greedy cannot terminate within a reasonable time (24 hours).

**Technique from (2) to (1).** Next, we evaluate the technique in Section 3.1, i.e., we switch from the naive candidate generation (CG) approach to the improved candidate generation of *Mags*. According to the performance reports in Figures 8b and 8d, the improved CG is on average 3.68x faster than the naive CG in running time, while the difference in summary compactness is negligible. When we improve from (2) to (1), the speedup of the total running time is on average 2x over all datasets, while the improved approach achieves a higher speedup on large graphs (e.g., UK and IT) and the graphs with a high average degree (e.g., HO and ES).

## 6.4 Technique Effectiveness of *Mags-DM*

In this section, we compare our *Mags-DM* with the following variants to validate the effectiveness of our design choices.

(1) *Mags-DM*: parallel version (40 cores) with $T = 50$.
(2) *Mags-DM (no DS)*: a variant of (1) without our dividing strategy.
(3) *Mags-DM (no MS)*: a variant of (1) without our merging strategy.
(4) *SWeG* [34]: the serial version without both strategies.

Figures 9-10 report the performance of these methods. Compared with *Mags-DM (no DS)*, our *Mags-DM* provides a similar compactness using 14.4x less time on average. This is because the dividing strategy of *Mags-DM* can finely divide the nodes into small groups and speed up the merging computation within those groups.

Compared with *Mags-DM (no MS)*, our *Mags-DM* takes 21.3x less time and returns a summary that is 12.8% more compact. Recall in Section 4 that *Mags-DM* adopts three merging strategies, and we also compare *Mags-DM* with the version that removes one of the strategies. The results show that the first (node selection) can increase the compactness by 3.1%; the second (similarity measure) can increase the compactness by 2.8% and efficiency by 11.4x; and the third (merge threshold) can increase the compactness by 1%.

We also implement *SWeG* [34] and compare it with our algorithm. The results show that our *Mags-DM* takes on average 202x less time and is able to return a more compact summary on every dataset. This validates the design choices of our *Mags-DM*.

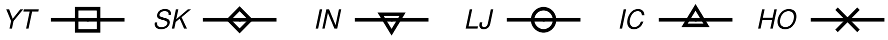
### 6.5 Parameter Analysis

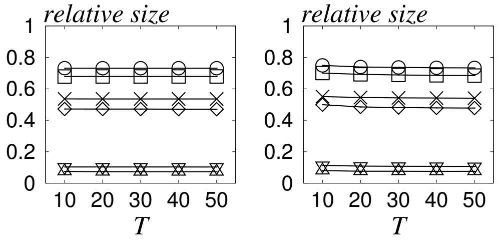
**Different Numbers of Iterations** $T$**.** Figures 11 and 12 present the performance of *Mags* and *Mags-DM* with different $T$. When we increase $T$ from 10 to 50, the output relative size of *Mags* (resp. *Mags-DM*) is decreased by on average 0.5% (resp. 2%), while the running time is increased by 35% (resp. 37%). In other words, the summary compactness of *Mags* and *Mags-DM* quickly converges when $T$ is small (e.g., $T = 20$), while adopting a larger $T$ will slightly improve the compactness at the cost of more running time. Our algorithms are robust to the number of iterations, because our proposed merge threshold $\omega(t)$ in Equation 6 can automatically adjust the step size (i.e., $r$) based on $T$, such that $\omega(t)$ decrease steadily with the increase of $t$.

**Different Numbers of Threads** $p$**.** Given the number of thread $p$, we evaluate the parallel speedup ratio as $T_1/T_p$, where $T_1$ (resp. $T_p$) equals the running time when executing the parallel algorithm with 1 thread (resp. $p$ threads). Figure 13 reports the parallel speedup when we run *Mags* and *Mags-DM* with different numbers of threads, varying $p$ in $\{1, 5, 10, 20, 40\}$. We observe that *Mags* only provide an average speedup of 3.4x when we run *Mags* using 40 cores. In comparison, *Mags-DM* can provide an average speedup of 12.1x using 40 cores. In other words, the parallelism of *Mags* is limited, while *Mags-DM* is effective for parallel computing. This is because *Mags* faces a data race when merging a batch of node pairs concurrently, while *Mags-DM* divides nodes into groups that are independent of each other. Besides, *Mags* and *Mags-DM* both provide a better speedup on large graphs, e.g., they achieve about 5x and 15-20x speedup on datasets IC and HO, respectively.

**Other Parameters (i.e.,** $b, h, k$**).** Recall that *Mags* sets $b, h$ as small constants and $k$ as $c \cdot d_{avg}$ (see Theorems 3 and 2), and that *Mags-DM* sets $b, h$ as small constants (see Theorem 5). Figures 14-16 report the performance of *Mags* and *Mags-DM* with different $b$, $h$, and $k$. Overall, these parameters have a limited impact on summary compactness (the difference is less than 0.5% on average), and they have a wide range of feasible values. By default, we set $k = \min\{5 \cdot d_{avg}, 30\}$, $b = 5$, and $h = 40$.

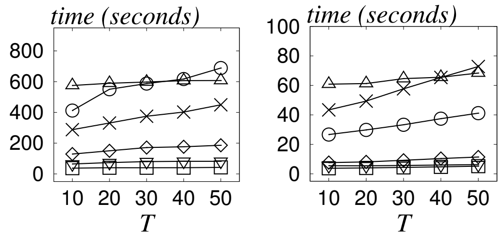
*Mags* and *Mags-DM* can adopt a wide range of values of $b, h, k$, and the reason is as follows. For *Mags*, we can generate a sufficient number of promising candidate pairs when $b$, $h$, and $k$ reach certain values, then the greedy merge of *Mags* will always return a compact summary. For *Mags-DM*, the algorithm uses MinHash to find promising pairs within groups, where $b$ and $h$ can hardly improve the precision of finding pairs when the parameters reach a certain level.





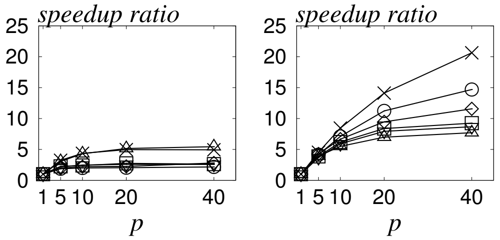
(a) *Mags* (b) *Mags-DM*

Fig. 11. Compactness vs. $T$.



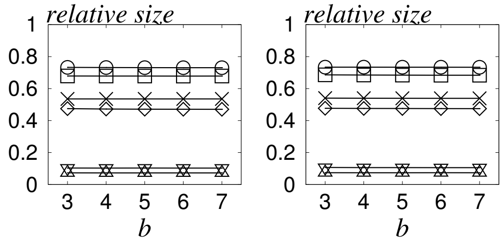
(a) *Mags* (b) *Mags-DM*

Fig. 12. Running time vs. $T$.



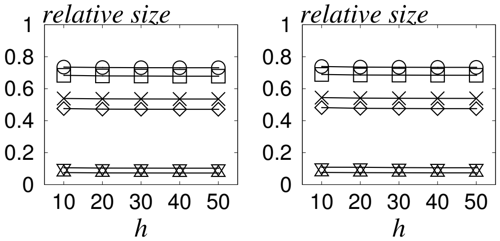
(a) *Mags* (b) *Mags-DM*

Fig. 13. Speedup vs. #threads $p$.



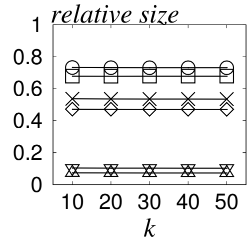
(a) *Mags* (b) *Mags-DM*

Fig. 14. Compactness vs. $b$.



(a) *Mags* (b) *Mags-DM*

Fig. 15. Compactness vs. $h$.



(a) *Mags*

Fig. 16. Compactness vs. $k$.

**Algorithm 6**: Neighbor Query of $q$ on Summary Graph

```
Input  : node $q$, summary graph $S = (P, E)$, corrections $C$
Output: $q$'s neighbor set $\hat{\mathcal{N}}_q$
$\hat{\mathcal{N}}_q \leftarrow \emptyset$;  $u \leftarrow$ the super-node such that $q \in P_u$;
for each super-node neighbor $v$ of $u$ do  $\hat{\mathcal{N}}_q \leftarrow \hat{\mathcal{N}}_q \cup P_v$;
Let $C_q^+ = \{x \mid +(q,x) \in C\}$ and $C_q^- = \{x \mid -(q,x) \in C\}$;
$\hat{\mathcal{N}}_q \leftarrow (\hat{\mathcal{N}}_q \cup C_q^+) \setminus C_q^-$;
return $\hat{\mathcal{N}}_q$;
```

### 6.6 Graph Query Processing

In what follow, we show that our representation (see Definition 1) can handle general-case graph queries (e.g., neighbor queries) and speed up specific graph queries (e.g., PageRank). In the future, we will investigate other graph queries.

Table 3. Running Time of the PageRank Computation (in seconds, bold is better).

| | SL | DB | AM | CN | YT | SK | IN | EU | ES | LJ | HO | IC | UK | IT | avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Input Graph Query | **0.09** | **0.34** | **0.65** | 0.41 | **1.3** | **2.7** | **1.9** | 2.0 | **4.0** | **9.6** | **15.5** | 15.6 | 103 | 109 | 19.0 |
| Summary Graph Query (Alg. 7) | 0.13 | 0.62 | 1.2 | **0.40** | 2.6 | 4.7 | **1.9** | **1.9** | 7.0 | 20.9 | 17.9 | **12.2** | **88** | **94** | **18.1** |
| Relative Size of Summary | 0.84 | 0.48 | 0.61 | 0.13 | 0.68 | 0.47 | 0.10 | 0.19 | 0.72 | 0.73 | 0.54 | 0.07 | 0.07 | 0.08 | 0.41 |

```
Algorithm 7: Compute PageRank on Summary Graph
   Input  : damping factor $d$, iteration number $T$
            summary graph $S = (P, E)$, corrections $C$
   Output: the PageRank of all nodes $PR$
1  Let $PR_0$ be a size-$n$ vector of 1;
2  for $t = 1..T$ do
3      Let $PR_t$ be a size-$n$ vector of $(1 - d)$;
4      for each super-node $u$ in $P$ do  $A_u \leftarrow \sum_{y \in P_u} d \cdot \frac{PR_{t-1}(y)}{|\mathcal{N}_y|}$;
5      for each super-node $u$ in $P$ do
6          $B_u \leftarrow \sum_{(u,v) \in E} A_v$;
7          for each $x$ in $P_u$ do  $PR_t(x)$ += $B_u$;
8      for each $+(x, y) \in C$ do  $PR_t(x)$ += $d \cdot \frac{PR_{t-1}(y)}{|\mathcal{N}_y|}$;
9      for each $-(x, y) \in C$ do  $PR_t(x)$ −= $d \cdot \frac{PR_{t-1}(y)}{|\mathcal{N}_y|}$;
10 return $PR_T$;
```

**Neighbor Queries.** Given a node $q$, we aim to return its neighbor set $\mathcal{N}_u$. Algorithm 6 presents how to answer neighbor queries on summary graph $S = (P, E)$ and corrections $C$. The expected time complexity of Algorithm 6 is $O(1.12 \cdot d_{avg})$ where $d_{avg}$ is the average degree (i.e., $2m/n$). This result is also reported in Shin et al. [34].

The time complexity is obtained as follows. Among all datasets in Table 2, when we run our *Mags* or *Mags-DM*, the number of negative corrections $|C^-|$ is no more than 6% of the number of edges in the input graph, i.e., $|C^-| \leq 0.06 \cdot m$. The time complexity of Algorithm 6 is in proportional to $|\mathcal{N}_u| + 2 \cdot |C_q^-|$ when we use hash tables. Thus, the expected time complexity of Algorithm 6 equals

$$\mathbb{E}\{|\mathcal{N}_u| + 2 \cdot |C_q^-|\} = \frac{|\mathcal{E}| + 2 \cdot |C^-|}{n/2} \leq \frac{m + 0.12 \cdot m}{n/2} = 1.12 \cdot d_{avg}$$

**PageRank Computation.** PageRank is an algorithm for measuring the importance of website pages. Let $d$ be the damping factor. Then, we define the PageRank of a node $x$ at timestamp $t$ as

$$PR_t(x) = \begin{cases} 1 & \text{if } t = 0 \\ (1 - d) + d \cdot \sum_{y \in \mathcal{N}_x} \frac{PR_{t-1}(y)}{|\mathcal{N}_y|} & \text{if } t > 0 \end{cases} \tag{8}$$

Given $d$ and an iteration number $T$, we aim to compute the PageRank for any $t \leq T$. Algorithm 7 describes the algorithm for computing the PageRank on the summary graph $S = (P, E)$ and corrections $C$. Specifically, given a super-node $u$ that contains a node $x$, rather than summing over $x$'s neighbor set $\mathcal{N}_x$ (see Equation 8), we sum over the super-node neighbor of $u$ (Lines 4-7) and adjust the values using the corrections that contain $x$ (Lines 8-9).

The running time of Algorithm 7 is $O(T \cdot (|E| + |C|))$, while the query on the input graph using Equation 8 takes $O(T \cdot m)$ time. By the fact that $|E| + |C| \leq m$, the PageRank computation on the summary is asymptotically faster than that on the input graph.

Table 3 reports the running time of the two methods. The results show that Algorithm 7 has a smaller average running time, and it outperforms the PageRank query on the input graph for 8 out of 18 datasets. For the remaining datasets, the time cost of Algorithm 7 is no better than the input graph query as the operations in Algorithm 7 incur a large constant factor.

**Discussion.** The experiments show that we can achieve similar (or even slightly better) running time when we run the algorithm on the summary graph, and a more compact summary usually leads to faster graph processing. However, it is not guaranteed that the algorithm on the summary graph is faster, because the algorithm on the original graph may be simpler in its implementation and the decompression is essentially required for some operations.

## 7 RELATED WORKS

**Graph Summarization.** There has been a large body of literature on graph summarization (see [2, 21, 22, 24, 25, 27, 30, 34, 45]). Navlakha et al. [30] are the first to propose the graph summarization problem. They propose a greedy method and a randomized method (i.e., *Greedy* and *Randomized*). Shin et al. [34] overcome the inefficiency of previous works, and they devise a novel algorithm named *SWeG* that can scale to billion-scale graphs. The efficiency of Shin et al.'s paradigm [34] has motivated a line of follow-up works, e.g., Yong et al. [45] design an algorithm *LDME* that improves over *SWeG* in performance, and Lee et al. [25] propose a hierarchical graph summarization problem along with an algorithm *Slugger* for the problem. Besides, *Mosso* [22] is a graph summarization method for dynamic graph streams; Liu et al. [27] study the graph summarization problem under a distributed setting. In Section 6, the experiments show that our methods (*Mags* and *Mags-DM*) outperform the state-of-the-art methods (*Greedy*, *LDME*, and *Slugger*) in compactness and efficiency.

**Other Techniques for Summarizing Graphs.** In this paper, we refer to "graph summarization" as the problem in Definition 1, but this term is also used in many other problems that seek a compact graph representation while discovering the structural patterns of the graph. Liu et al. [28] provide a comprehensive overview of these problems. For example, Tian et al. [39] summarize graphs based on node attributes and different edge; Zhang et al. [46] design a discovery-driven graph summarization; Koutra et al. [23] describe a graph with a set of assumed vocabulary (i.e., cliques, stars, chains, and bipartite cores); Song et al. [36] develop a parallel summarization algorithm for large knowledge graphs; Ke et al. [20] study the summarization of multi-relation networks; Tang et al. [37] propose a sketch for summarizing graph streams in a sub-linear space; Cebiric et al. [8] survey the summarization of semantic graphs.

Another line of studies aims to produce a graph summary of a certain size (typically lossy) which minimizes the reconstruction error, e.g., Riondato et al. [31] aim to produce a summary that can be stored in main memory with a quality guarantee; Toivonen et al. [40] study how to compress weighted graphs with relatively small compression errors; Lefevre et al. [26] summarize a graph specifically for a series of graph queries. Some works introduced in the above subsections [2, 24, 27] also belong to this category. These graph summarization methods can be adapted to a lossless scenario by adding a correction set, but they are not originally conceived for a lossless setting.

The above methods are unable to provide a summary as compact as the result of the graph summarization problem (Definition 1), because the problem objectives are different. For instance, *LDME* [45] outperforms Koutra et al.'s method [23], and a method [24] for lossy graph summarization (the lossy version of Definition 1) outperforms Lefevre et al.'s method [26].

**Other Techniques for Compressing Graphs.** The graph summarization techniques discussed above seek a small graph representation that can capture structural patterns. In comparison, a graph compression technique seeks the absolutely smallest representation, e.g., relabeling node [1, 5, 14], compressing adjacency lists with reference encoding [5], investigating a graph representation that asymptotically matches the storage lower bound [18]. Besta et al. [3] survey over those graph compression techniques.

This line of research complements (and is orthogonal to) the studies on graph summarization. According to Shin et al. [34], *SWeG* (as well as our *Mags* and *Mags-DM*) can be combined with any compression technique to improve its compression performance. In other words, we can feed the output of our *Mags* or *Mags-DM* to another graph compression method, and compress it further. Due to the prevalence of large-scale graph analysis [10, 12, 13, 33, 41, 43, 47, 48], both graph summarization and graph compression are useful in practice.

## 8 CONCLUSION

This paper presents *Mags* and *Mags-DM*, two algorithms for graph summarization. In particular, *Mags* is a greedy algorithm that runs in $O(T\cdot m\cdot(d_{avg}+\log m))$ time, and it significantly improves over the existing greedy method in efficiency without affecting its summary compactness. We also devise a divide-and-merge algorithm *Mags-DM* that takes $O(T \cdot m)$ running time, and it improves over existing divide-and-merge methods in summary compactness and practical efficiency. In addition, both *Mags* and *Mags-DM* can be accelerated by parallel computing. Our experiments show that *Mags* and *Mags* provide state-of-the-art summary compactness and efficiency, but existing methods can only achieve state-of-the-art in one of them. This is the first time that graph summarization algorithms are made practical while still offering a compact summary. For future work, we may extend *Mags* and *Mags-DM* to lossy summarization when higher compactness is required, e.g., we allow a bounded error in the representation. It is also interesting to study the extension of *Mags* and *Mags-DM* to dynamic graphs that are frequently updated.

## ACKNOWLEDGMENTS

Fan Zhang is partially supported by NSFC (62002073) and Guangdong Basic and Applied Basic Research Foundation (2023A1515012603, 2024A1515011501). Deming Chu is supported by the scholarship of China Scholarship Council No. 202006140012. Wenjie Zhang is supported by ARC DP230101445 and ARC FT210100303. Xuemin Lin is supported by NSFC U2241211 and NSFC U20B2046. This project is also supported by ARC LP210301046 and ARC DP230101445. We would like to thank the anonymous reviewers for their valuable and constructive feedback.

## REFERENCES

[1] Alberto Apostolico and Guido Drovandi. 2009. Graph compression by BFS. *Algorithms* 2, 3 (2009), 1031–1044.

[2] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. 2018. Scalable approximation algorithm for graph summarization. In *PAKDD*. Springer, 502–514.

[3] Maciej Besta and Torsten Hoefler. 2018. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799* (2018).

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*. 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. 595–602.

[6] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 2000. Min-Wise Independent Permutations. *J. Comput. Syst. Sci.* 60, 3 (2000), 630–659.

[7] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *WSDM*. 95–106.

[8] Šejla Čebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *VLDBJ* 28 (2019), 295–327.

[9] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *KDD*. 219–228.

[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *VLDB* 8, 12 (2015), 1804–1815.

[11] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the best k in core decomposition: A time and space optimal solution. In *ICDE*. IEEE, 685–696.

[12] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2024. Discovering and Maintaining the Best k in Core Decomposition. *TKDE* (2024).

[13] Deming Chu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2022. Hierarchical core decomposition in parallel: From construction to subgraph search. In *ICDE*. IEEE, 1138–1151.

[14] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *KDD*. 1535–1544.

[15] Songyun Duan, Achille Belly Fokoue-Nkoutche, Anastasios Kementsietsidis, Wangchao Le, Feifei Li, and Kavitha Srinivas. 2015. Scalable summarization of data graphs. US Patent 8,977,650.

[16] Tom Eelbode, Jeroen Bertels, Maxim Berman, Dirk Vandermeulen, Frederik Maes, Raf Bisschops, and Matthew B Blaschko. 2020. Optimization for medical image segmentation: theory and practice when evaluating with dice score or jaccard index. *IEEE Transactions on Medical Imaging* 39, 11 (2020), 3679–3690.

[17] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2021. Making graphs compact by lossless contraction. In *SIGMOD*. 472–484.

[18] Arash Farzan and J Ian Munro. 2013. Succinct encoding of arbitrary graphs. *Theoretical Computer Science* 513 (2013), 38–52.

[19] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*. 604–613.

[20] Xiangyu Ke, Arijit Khan, and Francesco Bonchi. 2022. Multi-relation graph summarization. *TKDD* 16, 5 (2022), 1–30.

[21] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. 2015. Set-based approximate approach for lossless graph summarization. *Computing* 97 (2015), 1185–1207.

[22] Jihoon Ko, Yunbum Kook, and Kijung Shin. 2020. Incremental lossless graph summarization. In *KDD*. 317–327.

[23] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. 2014. Vog: Summarizing and understanding large graphs. In *ICDM*. SIAM, 91–99.

[24] Kyuhan Lee, Hyeonsoo Jo, Jihoon Ko, Sungsu Lim, and Kijung Shin. 2020. Ssumm: Sparse summarization of massive graphs. In *KDD*. 144–154.

[25] Kyuhan Lee, Jihoon Ko, and Kijung Shin. 2022. Slugger: lossless hierarchical summarization of massive graphs. In *ICDE*. IEEE, 472–484.

[26] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *ICDM*. SIAM, 454–465.

[27] Xingjie Liu, Yuanyuan Tian, Qi He, Wang-Chien Lee, and John McPherson. 2014. Distributed graph summarization. In *SIGMOD*. 799–808.

[28] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph summarization methods and applications: A survey. *CSUR* 51, 3 (2018), 1–34.

[29] Antonio Maccioni and Daniel J Abadi. 2016. Scalable pattern matching over compressed graphs via dedensification. In *KDD*. 1755–1764.

[30] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. 2008. Graph summarization with bounded error. In *SIGMOD*. 419–432.

[31] Matteo Riondato, David García-Soriano, and Francesco Bonchi. 2017. Graph summarization with quality guarantees. *Data mining and knowledge discovery* 31 (2017), 314–349.

[32] Ryan A Rossi and Rong Zhou. 2018. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data* 5, 1 (2018), 1–14.

[33] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *VLDB* 11, 4 (2017), 420–431.

[34] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. 2019. Sweg: Lossless and lossy summarization of web-scale graphs. In *WWW*. 1679–1690.

[35] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*. IEEE, 403–412.

[36] Qi Song, Yinghui Wu, Peng Lin, Luna Xin Dong, and Hui Sun. 2018. Mining summaries for knowledge graph search. *TKDE* 30, 10 (2018), 1887–1900.

[37] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph stream summarization: From big bang to big crunch. In *SIGMOD*. 1481–1496.

[38] Robert E Tarjan and Jan Van Leeuwen. 1984. Worst-case analysis of set union algorithms. *JACM* 31, 2 (1984), 245–281.

[39] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. 2008. Efficient aggregation for graph summarization. In *SIGMOD*. 567–580.

[40] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. 2011. Compression of weighted graphs. In *AKDD*. 965–973.

[41] Kai Wang, Gengda Zhao, Wenjie Zhang, Xuemin Lin, Ying Zhang, Yizhang He, and Chunxiao Li. 2023. Cohesive Subgraph Discovery over Uncertain Bipartite Graphs. *TKDE* (2023).

[42] Yiqi Wang, Long Yuan, Zi Chen, Wenjie Zhang, Xuemin Lin, and Qing Liu. 2023. Towards efficient shortest path counting on billion-scale graphs. In *ICDE*. IEEE, 2579–2592.

[43] Jiadong Xie, Zehua Chen, Deming Chu, Fan Zhang, Xuemin Lin, and Zhihong Tian. 2024. Influence Maximization via Vertex Countering. *VLDB* 17, 6 (2024), 1297–1309.

[44] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[45] Quinton Yong, Mahdi Hajiabadi, Venkatesh Srinivasan, and Alex Thomo. 2021. Efficient graph summarization using weighted lsh at billion-scale. In *SIGMOD*. 2357–2365.

[46] Ning Zhang, Yuanyuan Tian, and Jignesh M Patel. 2010. Discovery-driven graph summarization. In *ICDE*. IEEE, 880–891.

[47] Gengda Zhao, Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Yizhang He. 2022. Efficient computation of cohesive subgraphs in uncertain bipartite graphs. In *ICDE*. IEEE, 2333–2345.

[48] Zhongxin Zhou, Wenchao Zhang, Fan Zhang, Deming Chu, and Binghao Li. 2022. VEK: a vertex-oriented approach for edge k-core problem. *World Wide Web* 25, 2 (2022), 723–740.

Received October 2023; revised January 2024; accepted February 2024