SWeG: Lossless and Lossy Summarization of Web-Scale Graphs

Kijung Shin
School of Electrical Engineering, KAIST, Daejeon, South Korea
kijungs@kaist.ac.kr

Amol Ghoting
LinkedIn Corporation, Mountain View, CA, USA
aghoting@linkedin.com

Myunghwan Kim
LinkedIn Corporation, Mountain View, CA, USA
mykim@cs.stanford.edu

Hema Raghavan
LinkedIn Corporation, Mountain View, CA, USA
hraghavan@linkedin.com

ABSTRACT

Given a terabyte-scale graph distributed across multiple machines, how can we summarize it, with much fewer nodes and edges, so that we can restore the original graph exactly or within error bounds?

As large-scale graphs are ubiquitous, ranging from web graphs to online social networks, compactly representing graphs becomes important to efficiently store and process them. Given a graph, *graph summarization* aims to find its compact representation consisting of (a) a *summary graph* where the nodes are disjoint sets of nodes in the input graph, and each edge indicates the edges between all pairs of nodes in the two sets; and (b) *edge corrections* for restoring the input graph from the summary graph exactly or within error bounds. Although graph summarization is a widely-used graph-compression technique readily combinable with other techniques, existing algorithms for graph summarization are not satisfactory in terms of speed or compactness of outputs. More importantly, they assume that the input graph is small enough to fit in main memory.

In this work, we propose SWeG, a fast parallel algorithm for summarizing graphs with compact representations. SWeG is designed for not only shared-memory but also MapReduce settings to summarize graphs that are too large to fit in main memory. We demonstrate that SWeG is **(a) Fast:** SWeG is up to *5400× faster* than its competitors that give similarly compact representations, **(b) Scalable:** SWeG scales to graphs with *tens of billions* of edges, and **(c) Compact:** combined with state-of-the-art compression methods, SWeG achieves up to *3.4*× better compression than them.

ACM Reference Format:
Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. 2019. SWeG: Lossless and Lossy Summarization of Web-Scale Graphs. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558.3313402

1 INTRODUCTION

Large-scale graphs are everywhere. Graphs are natural representations of the web, social networks, collaboration networks, internet topologies, citation networks, to name just a few. The rapid growth of the web and its applications has led to large-scale graphs of unprecedented size, such as 3.5 billion web pages connected by 129 billion hyperlinks [36], online social networks with 300 billion connections [11], and 1.15 billion query-URL pairs [30].

Consequently, representing graphs in a storage-efficient manner has become important. In addition to the reduction of hardware costs, compact representation may allow large-scale graphs to fit in main memory of one machine, eliminating expensive I/O over the network or to disk. Moreover, it can lead to performance gains by allowing large fractions of the graphs to reside in cache [7, 48].

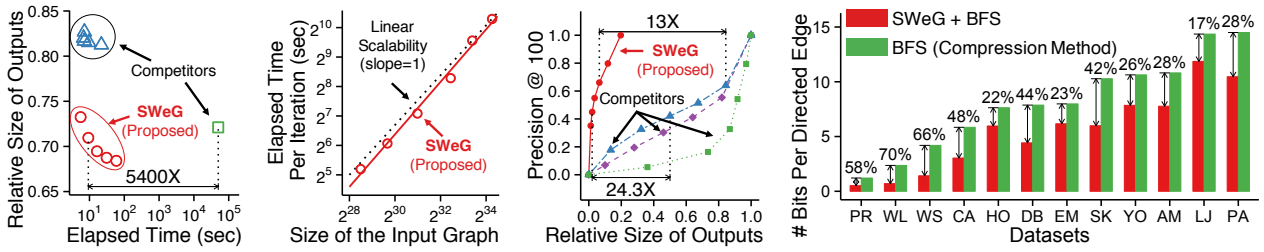
To compactly represent graphs, a variety of graph-compression techniques have been developed, including relabeling nodes [2, 5, 9, 11], employing integer-sequence encoding schemes (e.g., reference, gap, and interval encodings) [5], and encoding common structures (e.g., cliques, bipartite-cores, and stars) with fewer bits [7, 22, 43]. These techniques are either *lossless* and *lossy* depending on whether the input graph can be reconstructed exactly from their outputs.

Graph summarization [38] is a widely-used graph-compression technique. It aims to find a compact representation of a given graph $\mathcal{G}$ consisting of a summary graph and edge corrections. The *summary graph* $\overline{\mathcal{G}}$ is a graph where the nodes are disjoint sets of nodes in $\mathcal{G}$ and each edge indicates the edges between all pairs of nodes in the two sets. The *edge corrections* (i.e., edges to be inserted, and edges to be deleted) are for restoring $\mathcal{G}$ from $\overline{\mathcal{G}}$ exactly (in cases of lossless summarization) or within given error bounds (in cases of lossy summarization). The outputs can be considered as graphs: (a) the summary graph, (b) the graph consisting of the edges to be inserted, and (c) the graph consisting of the edges to be deleted.

As a graph-compression technique, graph summarization has the following desirable properties: **(a) Combinable**: since graph summarization produces three graphs, as explained in the previous paragraph, each of them can be further compressed by any other graph compression technique, **(b) Adjustable**: the trade-off between compression rates and information loss can be adjusted by given error bounds, and **(c) Queryable**: neighbor queries (i.e., finding the neighboring nodes of a given node) can be answered efficiently on the outputs of graph summarization (see Appendix A).

However, existing algorithms for graph summarization are not satisfactory in terms of speed [38, 44] or compactness of output representations [20]. These serial algorithms either have high computational complexity or significantly sacrifice compactness of outputs for lower complexity. Moreover, they assume that the input graph is small enough to fit in main memory. The largest graph used in the previous studies has just about 10 million edges [20, 38, 44]. As

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW '19, May 13–17, 2019, San Francisco, CA, USA
© 2019 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313402



(a) Fast and Compact (Lossless Summarization)

(b) Scalable (Lossless Summarization)

(c) Compact and Accurate (Lossy Summarization)

(d) Compact (Lossless Compression)

Figure 1: SWeG is fast and scalable with compact and accurate representations. (a) SWeG is faster with more compact representations than the other lossless-summarization methods. (b) SWeG scales linearly with the size of the input graph, successfully scaling to graphs with over 20 billions edges. (c) The lossy version of SWeG yields more compact and accurate representations than the other lossy-summarization methods. (d) Combining SWeG and an advanced compression technique [2] yields up to 3.4× more compact representations than using the technique alone. See Section 4 for details.

large-scale graphs often do not fit in main memory, improving the scalability of graph summarization remains an important challenge.

To address this challenge, we propose SWeG (**S**ummarizing **We**b-Scale **G**raphs), a fast parallel algorithm for summarizing large-scale graphs with compact representations. SWeG is designed for both shared-memory and MapReduce settings. In a nutshell, SWeG repeats (a) dividing the input graph into small subgraphs and (b) processing the subgraphs in parallel without having to load the entire graph in main memory. SWeG also adjusts, in each iteration, how aggressively it merges nodes within subgraphs, and this turns out to be crucial for obtaining compact representations.

Extensive experiments with 13 real-world graphs show that SWeG significantly outperforms existing graph-summarization methods and enhances the best compression techniques, as shown in Figure 1. Specifically, SWeG provides the following advantages:

- **Speed**: SWeG is up to *5, 400× faster* than its competitors that give similarly compact representations (Figure 1(a)).
- **Scalability**: SWeG scales to graphs with *tens of billions* of edges, showing near-linear data and machine scalability (Figure 1(b)).
- **Compression**: Combined with advanced graph-compression methods, SWeG yields up to *3.4× better* compression than the methods (Figure 1(d)).

In Section 2, we provide notations and a formal problem definition. In Section 3, we present our proposed algorithm, SWeG. In Section 4, we provide our experimental results. After discussing related work in Section 5, we offer conclusions in Section 6.

2 NOTATIONS AND PROBLEM DEFINITION

Notations and Concepts. See Table 1 for frequently-used symbols and Figure 2 for an illustration of concepts. Consider a simple undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$. We denote each node in $\mathcal{V}$ by a lowercase (e.g., v) and each edge in $\mathcal{E}$ by an unordered pair (e.g., $\{u, v\}$). We denote the set of neighbors of a node $v \in \mathcal{V}$ in $\mathcal{G}$ by $N_v \subset \mathcal{V}$.

A *summary graph* of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, denoted by $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, is a graph whose nodes are a partition of $\mathcal{V}$ (i.e., disjoint and exhaustive subsets of $\mathcal{V}$). That is, each node $v \in \mathcal{V}$ belongs to exactly one node in $\mathcal{S}$. We call nodes and edges in $\overline{\mathcal{G}}$ *supernodes* and *superedges*. We denote each supernode by an uppercase (i.e., A). $\mathcal{P}$ may include self-loops, and we let $\mathcal{P}^* \subset \mathcal{P}$ be the set of non-loop superedges.

Given a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and *corrections* $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, where $\mathcal{C}^+$ denotes the set of edges to be inserted and $\mathcal{C}^-$ denotes the set of edges to be deleted, a graph $\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$, which we call a *restored graph*, is created by the following steps:

(1) For each superedge $\{A, B\} \in \mathcal{P}$, all pairs of distinct nodes in A and B (i.e.,. $\{\{u, v\} : u \in A, v \in B, u \neq v\}$) are added to $\hat{\mathcal{E}}$,
(2) Each edge in $\mathcal{C}^+$ is added to $\hat{\mathcal{E}}$,
(3) Each edge in $\mathcal{C}^-$ is removed from $\hat{\mathcal{E}}$.

We let the neighbors of each node $v \in \mathcal{V}$ in $\hat{\mathcal{G}}$ be $\hat{N}_v \subset \mathcal{V}$. On $\overline{\mathcal{G}}$ and $\mathcal{C}$, neighbor queries (i.e., finding $\hat{N}_v$ for a query node $v \in \mathcal{V}$) can be answered rapidly without restoring entire $\hat{\mathcal{G}}$ (see Appendix A).

Problem Definition. The large-scale graph summarization problem, which we address in this work, is defined in Problem 1.

Problem 1 (Large-scale Graph Summarization).

(1) **Given**: *a large-scale graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which may or may not fit in main memory, and an error bound ϵ (≥ 0)*
(2) **Find:** *a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$*

Table 1: Table of frequently-used symbols.

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	input graph with nodes $\mathcal{V}$ and edges $\mathcal{E}$
N_v	set of neighbors of node v in $\mathcal{G}$
$\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$	edge corrections (i.e., edges insertions and deletions)
$\mathcal{C}^+$	set of edges to be inserted
$\mathcal{C}^-$	set of edges to be deleted
$\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$	summary graph with supernodes $\mathcal{S}$ and superedges $\mathcal{P}$
$\mathcal{P}^*$	non-loop superedges in $\mathcal{P}$
$\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$	graph restored from $\overline{\mathcal{G}}$ and $\mathcal{C}$
$\hat{N}_v$	set of neighbors of node v in $\hat{\mathcal{G}}$
ϵ	error bound
T	number of iterations
$\theta(t)$	merging threshold in the t-th iteration

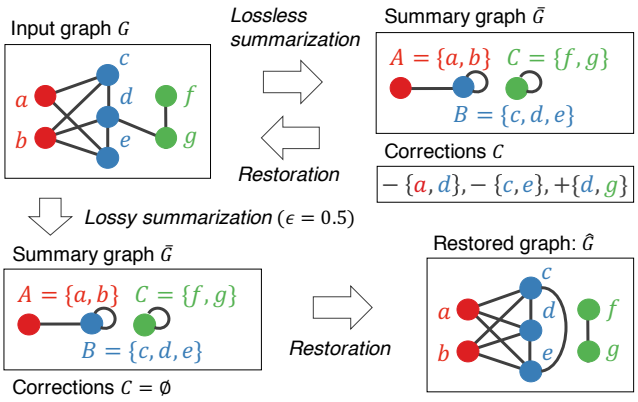


Figure 2: Illustration of graph summarization. Lossless summarization of the input graph (upper left) yields a summary graph and corrections (upper right) from which the input graph is restored exactly. Lossy summarization of the input graph (upper left) yields a summary graph and corrections (lower left). The restored graph (lower right) satisfies Eq. (2). Note that the outputs of the lossless and lossy graph summarization have fewer edges than the input graph.

(3) **to Minimize:**

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

(4) **Subject to:** *the restored graph* $\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$ *satisfies*

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

The objective (Eq. (1)), which we aim to minimize, measures the size of the output representation by the count of non-loop (super) edges. We exclude all self-loops in $\mathcal{P}$ from the objective since they can be encoded concisely using 1 bit per supernode regardless of their count. The constraints (Eq. (2)) [38] states that the neighbors N_v of each node $v \in \mathcal{V}$ in the input graph and the node's neighbors $\hat{N}_v$ in the restored graph $\hat{\mathcal{G}}$ should be similar enough so that the size of their symmetric difference (i.e., $(N_v \cup \hat{N}_v) - (\hat{N}_v \cap N_v)$) is at most a certain proportion of the size of the node's neighbors N_v in the input graph. The proportion is given as a parameter ϵ, which we call an *error bound*, and it controls the trade-off between compression rates and the amount of information loss. If $\epsilon = 0$, then $N_v = \hat{N}_v$ holds for every node $v \in \mathcal{V}$ and thus the restored graph $\hat{\mathcal{G}}$ is equal to the input graph $\mathcal{G}$. Thus, Problem 1 is **lossless summarization** if $\epsilon = 0$ and **lossy summarization** if $\epsilon > 0$. See Figure 2 for lossless and lossy summarization of a toy graph.

3 PROPOSED ALGORITHM: SWEG

We present our proposed algorithm, SWeG (**S**ummarizing **We**b-Scale **G**raphs). SWeG provides approximate solutions to the lossless and lossy graph summarization problems (i.e., Problems 1). In Section 3.1, we provide an overview of SWeG then describe each step of it in detail. In Sections 3.2 and 3.3, we discuss parallelizing SWeG in shared-memory and MapReduce settings. In Section 3.4, we analyze its time complexity and memory requirements. In Section 3.5, we present SWeG+, an algorithm for further compression.

Algorithm 1 Overview of SWeG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of iterations T, error bound ϵ
Output: summary graph $\bar{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections C
1: initialize supernodes $\mathcal{S}$ to $\{\{v\} : v \in \mathcal{V}\}$
2: **for** $t = 1 \ldots T$ **do**
3: divide $\mathcal{S}$ into disjoint groups ▷ Algorithm 2
4: merge some supernodes within each group ▷ Algorithm 3
5: encode edges $\mathcal{E}$ into superedges $\mathcal{P}$ and corrections C ▷ Algorithm 4
6: **if** $\epsilon > 0$ **then** drop some (super) edges from $\mathcal{P}$ and C ▷ Algorithm 5
7: **return** $\bar{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and C

Algorithm 2 Dividing Step of SWeG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, current supernodes $\mathcal{S}$
Output: disjoint groups of supernodes: $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(k)}\}$
1: generate a random bijective hash function $h : \mathcal{V} \to \{1, \ldots, |\mathcal{V}|\}$
2: **for each** supernode $A \in \mathcal{S}$ **do**
3: **for each** node $v \in A$ **do**
4: $f(v) \leftarrow \min(\{h(u) : u \in N_v \text{ or } u = v\})$ ▷ shingle of node v
5: $F(A) \leftarrow \min(\{f(v) : v \in A\})$ ▷ shingle of supernode A
6: divide the supernodes in $\mathcal{S}$ into $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(k)}\}$ by their $F(\cdot)$ value
7: **return** $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(k)}\}$

3.1 Description of the Algorithm

We first give an overview of SWeG, and then describe each step of it in detail. In this subsection, for simplicity, we assume that SWeG is executed serially and the input graph is small enough to fit in main memory of one machine. Parallelization and distributed processing are discussed in Sections 3.2 and 3.3.

3.1.1 **Overview (Algorithm 1).** SWeG requires an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the number of iterations T, and an error bound ϵ, as in Problem 1. SWeG first initializes the set $\mathcal{S}$ of supernodes so that (a) each supernode consists of a node in $\mathcal{V}$ and (b) each node in $\mathcal{V}$ belongs to one supernode (line 1 of Algorithm 1). Then, SWeG updates $\mathcal{S}$ by repeating the following steps T times (line 2):

- **Dividing step (line 3, Section 3.1.2)**: divides $\mathcal{S}$ into disjoint groups, each of which is composed of supernodes with similar connectivity. Different groups are obtained in each iteration.
- **Merging step (line 4, Section 3.1.3)**: merges some supernodes within each group in a greedy manner.

Dividing $\mathcal{S}$ into small groups makes SWeG faster and more memory efficient by allowing it to process different groups in parallel without having to load entire $\mathcal{G}$ in memory (see Sections 3.2 and 3.3). After updating $\mathcal{S}$, SWeG performs the following steps:

- **Encoding step (line 5, Section 3.1.4)**: encodes the edges $\mathcal{E}$ into superedges $\mathcal{P}$ and corrections C so that the count of non-loop (super) edges (i.e., Eq. (1)) is minimized given supernodes $\mathcal{S}$.
- **Dropping step (line 6, Section 3.1.5)**: makes $\mathcal{P}$ and C more compact by dropping some (super) edges from them within the error bounds (i.e., Eq. (2)) in cases of lossy summarization.

Lastly, SWeG returns the summary graph $\bar{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and the corrections C as its outputs (line 7).

3.1.2 **Dividing Step (Algorithm 2).** The dividing step aims to divide the supernodes $\mathcal{S}$ into disjoint groups of supernodes with similar connectivity. To this end, we extend the shingle of nodes, for which it is known that two nodes have the same shingle with

Algorithm 3 Merging Step of SWeG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, current supernodes $\mathcal{S}$, current iteration t, disjoint groups of supernodes $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(k)}\}$
Output: updated supernodes $\mathcal{S}$

1: **for each** group $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(k)}\}$ **do**
2: $\quad Q \leftarrow \mathcal{S}^{(i)}$
3: $\quad$ **while** $|Q| > 1$ **do**
4: $\quad\quad$ pick and remove a random supernode A from Q
5: $\quad\quad B \leftarrow \arg\max_{C \in Q} \textit{SuperJaccard}(A, C)$ ▷ Eq. (4)
6: $\quad\quad$ **if** $\textit{Saving}(A, B, \mathcal{S}) \geq \theta(t)$ **then** ▷ Eq. (3) and Eq. (5)
7: $\quad\quad\quad \mathcal{S} \leftarrow (\mathcal{S} - \{A, B\}) \cup \{A \cup B\}$ ▷ merge A and B
8: $\quad\quad\quad \mathcal{S}^{(i)} \leftarrow (\mathcal{S}^{(i)} - \{A, B\}) \cup \{A \cup B\}$
9: $\quad\quad\quad Q \leftarrow (Q - \{B\}) \cup \{A \cup B\}$ ▷ replace B with $A \cup B$
10: **return** $\mathcal{S}$

probability equal to the jaccard similarity of their neighbor sets [6], to supernodes. Specifically, we define the *shingle* $F(A)$ of each supernode $A \in \mathcal{S}$ as the smallest shingle of the nodes in A. Then, two supernodes $A \neq B \in \mathcal{S}$ are more likely to have the same shingle if the nodes in A and those in B have similar connectivity. Formally, given a random bijective hash function $h : \mathcal{V} \to \{1, ..., |\mathcal{V}|\}$,

$$F(A) := \min_{v \in A}(f(v)),$$

where $f(v) := \min_{u \in N_v \text{ or } u=v} h(u)$ is the shingle of node $v \in \mathcal{V}$, and N_v is the set of neighbors of node $v \in \mathcal{V}$ in the input graph $\mathcal{G}$. SWeG generates such a hash function h by shuffling the order of the nodes in $\mathcal{V}$ and mapping each i-th node to i (line 1 of Algorithm 2) and computes the shingle of every supernode in $\mathcal{S}$ (lines 2-5). Lastly, SWeG divides the supernodes $\mathcal{S}$ into disjoint groups $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$ by their shingle and returns the groups (lines 6-7). In cases where large groups exist, Algorithm 2 can be applied recursively to each of the large groups with a new hash function h.

3.1.3 **Merging Step (Algorithm 3).** To describe this step, we first define several concepts. We define the *cost* of each supernode $A \in \mathcal{S}$ given supernodes $\mathcal{S}$, denoted by $\textit{Cost}(A, \mathcal{S})$, as the amount of increase in Eq. (1) (i.e., the count of non-loop (super) edges in outputs) due to the edges adjacent to any node in A (see the encoding step in Section 3.1.4). Then, we define the *saving* due to the merger between supernodes $A \neq B \in \mathcal{S}$ given supernodes $\mathcal{S}$ as

$$\textit{Saving}(A, B, \mathcal{S}) := 1 - \frac{\textit{Cost}(A \cup B, (\mathcal{S} - \{A, B\}) \cup \{A \cup B\})}{\textit{Cost}(A, \mathcal{S}) + \textit{Cost}(B, \mathcal{S})}, \quad (3)$$

where $\textit{Cost}(A, \mathcal{S}) + \textit{Cost}(B, \mathcal{S})$ is the cost of A and B before their merger, and $\textit{Cost}(A \cup B, (\mathcal{S} - \{A, B\}) \cup \{A \cup B\})$ is their cost after their merger. That is, $\textit{Saving}(A, B, \mathcal{S})$ is the ratio of the cost reduction due to the merger and the cost before the merger. Lastly, we define the *supernode jaccard similarity* between supernodes $A \neq B \in \mathcal{S}$ as

$$\textit{SuperJaccard}(A, B) := \frac{\sum_{v \in N_A \cup N_B} \min(w(A, v), w(B, v))}{\sum_{v \in N_A \cup N_B} \max(w(A, v), w(B, v))}, \quad (4)$$

where $N_A := \bigcup_{v \in A} N_v$ is the set of nodes adjacent to any node in supernode $A \in \mathcal{S}$, and $w(A, v) := |\{u \in A : \{u, v\} \in \mathcal{E}\}|$ is the number of nodes in supernode $A \in \mathcal{S}$ adjacent to node $v \in \mathcal{V}$. $\textit{SuperJaccard}(A, B)$ measures the similarity of A and B in terms of their connectivity. Notice that it is 1 if A and B have the same connectivity (i.e., $w(A, v) = w(B, v)$ for every $v \in N_A \cup N_B$) and it is 0 if their nodes have no common neighbors (i.e., $N_A \cap N_B = \emptyset$).

Given disjoint groups of supernodes $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$, the merging step merges some pairs of supernodes within each group $\mathcal{S}^{(i)}$ in a greedy manner, as described in Algorithm 3. Notice that, in line 5, SWeG uses $\textit{SuperJaccard}(A, B)$ instead of $\textit{Saving}(A, B, \mathcal{S})$, which is a more straightforward choice, to find a candidate pair of supernodes $A \neq B \in \mathcal{S}^{(i)}$. This is because (a) $\textit{SuperJaccard}(A, B)$ is cheaper to compute than $\textit{Saving}(A, B, \mathcal{S})$ and (b) intuitively, $\textit{Saving}(A, B, \mathcal{S})$ tends to be high when A and B have similar connectivity. Computing $\textit{SuperJaccard}(A, C)$ instead of $\textit{Saving}(A, C, \mathcal{S})$ for every supernode $C \in Q$ in line 5 leads to a significant improvement in speed with small loss in the compactness of outputs, as shown empirically in Section 4.2. In line 6, the *merging threshold* $\theta(t)$ is set as in Eq. (5) so that SWeG gradually shifts from exploration (of supernodes in the other groups) to exploitation (of supernodes in the same group).

$$\theta(t) := \begin{cases} (1 + t)^{-1} & \text{if } t < T, \\ 0 & \text{if } t = T \end{cases} \quad (5)$$

This decreasing threshold is crucial for obtaining compact output representations, as shown empirically in Section 4.2.

3.1.4 **Encoding Step (Algorithm 4).** Given the supernodes $\mathcal{S}$ from the previous steps, the encoding step encodes the edges $\mathcal{E}$ of the input graph into superedges $\mathcal{P}$ and corrections $C =< C^+, C^- >$. We first describe how to encode edges that connect different supernodes (lines 3-5 of Algorithm 4). For each supernode pair $A \neq B \in \mathcal{S}$, we let $\mathcal{E}_{AB} \subset \mathcal{E}$ be the set of edges connecting A and B, and π_{AB} be the set of all pairs of nodes in A and B. That is,

$$\mathcal{E}_{AB} := \{\{u, v\} \subset \mathcal{V} : u \in A, v \in B, \{u, v\} \in \mathcal{E}\}, \quad (6)$$

$$\pi_{AB} := \{\{u, v\} \subset \mathcal{V} : u \in A, v \in B\}. \quad (7)$$

Recall how a graph is restored from $\mathcal{P}$ and C in Section 2. Then, the two options to encode the edges in $\mathcal{E}_{AB}$ are as follows:

(a) *without superedges*: merge $\mathcal{E}_{AB}$ into C^+,
(b) *with a superedge*: add $\{A, B\}$ to $\mathcal{P}$; merge $(\pi_{AB} - \mathcal{E}_{AB})$ into C^-.

Since (a) and (b) increase our objective (i.e., Eq. (1)) by $|\mathcal{E}_{AB}|$ and $(1 + |\pi_{AB}| - |\mathcal{E}_{AB}|)$, respectively, SWeG chooses (a) if $|\mathcal{E}_{AB}| \leq |\pi_{AB}|/2 = |A| \cdot |B|/2$ (line 4). Otherwise, it chooses (b) (line 5).

SWeG encodes edges between nodes within each supernode in a similar manner (lines 6-7). For each supernode $A \in \mathcal{S}$, we let $\mathcal{E}_{AA} \subset \mathcal{E}$ be the set of edges between nodes within A, and π_{AA} be the set of all pairs of distinct nodes within A. That is,

$$\mathcal{E}_{AA} := \{\{u, v\} \subset \mathcal{V} : u \neq v \in A, \{u, v\} \in \mathcal{E}\}, \quad (8)$$

$$\pi_{AA} := \{\{u, v\} \subset \mathcal{V} : u \neq v \in A\}. \quad (9)$$

Then, the two options to encode the edges in $\mathcal{E}_{AA}$ are as follows:

(c) *without superloops*: merge $\mathcal{E}_{AA}$ into C^+,
(d) *with a superloop*: add $\{A, A\}$ to $\mathcal{P}$; merge $(\pi_{AA} - \mathcal{E}_{AA})$ into C^-.

Since (c) and (d) increase our objective (i.e., Eq. (1)) by $|\mathcal{E}_{AA}|$ and $(|\pi_{AA}| - |\mathcal{E}_{AA}|)$, respectively, SWeG chooses (c) if $|\mathcal{E}_{AA}| \leq |\pi_{AA}|/2 = |A| \cdot (|A| - 1)/4$ (line 6). Otherwise, it chooses (d) (line 7).

3.1.5 **Dropping Step (Algorithm 5).** The dropping step is an optional step for lossy summarization (i.e., when $\epsilon > 0$). This step is skipped if lossless summarization is needed (i.e., when $\epsilon = 0$). Given the summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $C =< C^+, C^- >$ from the previous step, the dropping step drops some (super) edges from $\mathcal{P}$ and C to make the output representation

Algorithm 4 Encoding Step of SWeG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, supernodes $\mathcal{S}$
Output: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $C =< C^+, C^- >$

1: $\mathcal{P} \leftarrow \emptyset$; $C^+ \leftarrow \emptyset$; $C^- \leftarrow \emptyset$;
2: **for each** supernode $A \in \mathcal{S}$ **do**
3: **for each** supernode B ($\neq A$) where $\mathcal{E}_{AB} \neq \emptyset$ **do** ▷ Eq. (6)
4: **if** $\mathcal{E}_{AB} \leq \frac{|A|\cdot|B|}{2}$ **then** $C^+ \leftarrow C^+ \cup \mathcal{E}_{AB}$
5: **else** $\mathcal{P} \leftarrow \mathcal{P} \cup \{\{A, B\}\}$; $C^- \leftarrow C^- \cup (\pi_{AB} - \mathcal{E}_{AB})$ ▷ Eq. (7)
6: **if** $\mathcal{E}_{AA} \leq \frac{|A|\cdot(|A|-1)}{4}$ **then** $C^+ \leftarrow C^+ \cup \mathcal{E}_{AA}$ ▷ Eq. (8)
7: **else** $\mathcal{P} \leftarrow \mathcal{P} \cup \{\{A, A\}\}$; $C^- \leftarrow C^- \cup (\pi_{AA} - \mathcal{E}_{AA})$ ▷ Eq. (9)
8: **return** $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $C =< C^+, C^- >$

more compact (i.e., to further reduce our objective Eq. (1)) without changing more than ϵ of the neighbors of each node (i.e., within the error bounds given in Eq. (2)). SWeG first initializes the *change limit* c_v of each node $v \in \mathcal{V}$ to $\epsilon \cdot |N_v|$, which is the right-hand side of Eq. (2) (line 1 of Algorithm 5). Then, within the change limit of each node, SWeG drops some adjacent edges from C^+, C^-, and $\mathcal{P}$, as described in lines 2-4, lines 5-7, and lines 8-12, respectively. Notice that, when a superedge $\{A, B\}$ is dropped, the total decrement in the change limits is proportional to $|A| \cdot |B|$, which is used to sort the superedges in line 8. Lastly, SWeG returns the updated summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $C =< C^+, C^- >$ that satisfy the error bounds given in Eq. (2), as shown in Theorem 3.1 (line 13).

Theorem 3.1 (Error Bounds). *Given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, and corrections $C =< C^+, C^- >$ satisfying that the restored graph $\hat{\mathcal{G}}$ is equal to $\mathcal{G}$, Algorithm 5 returns $\overline{\mathcal{G}}$ and C that satisfy Eq. (2).*

Proof. From $\hat{\mathcal{G}} = \mathcal{G}$, $\hat{N}_v = N_v$ holds for every node $v \in \mathcal{V}$ at the beginning of the algorithm. Thus, after the change limit c_v of each node $v \in \mathcal{V}$ is initialized to $\epsilon \cdot |N_v|$ in line 1, Eq. (10) holds.

$$c_v \leq \epsilon \cdot |N_v| - |\hat{N}_v - N_v| - |N_v - \hat{N}_v|, \forall v \in \mathcal{V}. \tag{10}$$

Recall how $\hat{\mathcal{G}}$ is constructed from $\overline{\mathcal{G}}$ and C in Section 2. Eq. (10) still holds after C^+ is processed in lines 2-4 since dropping an edge from C^+ decreases c_v by 1, increases $|N_v - \hat{N}_v|$ by at most 1, and keeps $|\hat{N}_v - N_v|$ the same for each adjacent node $v \in \mathcal{V}$. Likewise, Eq. (10) still holds after C^- is processed in lines 5-7 since dropping an edge in C^- decreases c_v by 1, increases $|\hat{N}_v - N_v|$ by at most 1, and keeps $|N_v - \hat{N}_v|$ the same for each adjacent node $v \in \mathcal{V}$. Similarly, we can show that Eq. (10) still holds after $\mathcal{P}$ is processed in lines 8-12. Eq. (11) is enforced by lines 3, 6, and 9.

$$c_v \geq 0, \forall v \in \mathcal{V}. \tag{11}$$

Eq. (10) and Eq. (11) imply Eq. (2). ■

3.2 Parallelization in Shared Memory

We describe how each step of SWeG (Algorithm 1) is parallelized in shared-memory environments. In the dividing step (Algorithm 2), the supernodes in line 2 are processed independently in parallel. In the merging step (Algorithm 3), the groups of supernodes in line 1 are processed in parallel. In lines 6 and 7, the accesses and updates of $\mathcal{S}$ are synchronized. In the encoding step (Algorithm 4), the supernodes in line 2 are processed independently in parallel. Specifically, each thread has its own copies of $\mathcal{P}$, C^+, and C^-; and the copies are merged once, after all supernodes are processed. Lastly, in the dropping step (line 5), parallel merge sort [24] is used when sorting $\mathcal{P}$ in line 8. Although the other parts of the dropping step are executed serially, they take a negligible portion of the total execution time because they are executed only once, while the dividing and merging steps are repeated multiple times.

Algorithm 5 Dropping Step of SWeG (Optional)

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, error bound ϵ
summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $C =< C^+, C^- >$
Output: updated summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$,
updated corrections $C =< C^+, C^- >$

1: $c_v \leftarrow \epsilon \cdot |N_v|$ for each node $v \in \mathcal{V}$ ▷ change limits of nodes
2: **for each** edge $\{u, v\} \in C^+$ **do**
3: **if** $c_u \geq 1$ and $c_v \geq 1$ **then**
4: $C^+ \leftarrow C^+ - \{\{u, v\}\}$; $c_u \leftarrow c_u - 1$; $c_v \leftarrow c_v - 1$
5: **for each** edge $\{u, v\} \in C^-$ **do**
6: **if** $c_u \geq 1$ and $c_v \geq 1$ **then**
7: $C^- \leftarrow C^- - \{\{u, v\}\}$; $c_u \leftarrow c_u - 1$; $c_v \leftarrow c_v - 1$
8: **for each** superedge $\{A, B\} \in \mathcal{P}$ in the increasing order of $|A| \cdot |B|$ **do**
9: **if** $A \neq B$ and ($\forall v \in A$, $c_v \geq |B|$) and ($\forall v \in B$, $c_v \geq |A|$) **then**
10: $\mathcal{P} \leftarrow \mathcal{P} - \{\{A, B\}\}$;
11: **for each** $v \in A$ **do** $c_v \leftarrow c_v - |B|$
12: **for each** $v \in B$ **do** $c_v \leftarrow c_v - |A|$
13: **return** $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $C =< C^+, C^- >$

3.3 Distributed Processing with MapReduce

We describe how each step of SWeG is implemented in the MapReduce framework for large-scale graphs not fitting in main memory. We assume that the input graph $\mathcal{G}$ is stored in a file in a distributed file system where each record describes the nodes in a supernode and the neighbors of the nodes in $\mathcal{G}$. Specifically, the record $R(A)$ for a supernode $A \in \mathcal{S}$ is in the following format:

$$R(A) := (\text{id of } A, |A|, \underbrace{(u, |N_u|, (\overbrace{v, \ldots, w}^{|N_u|})), \ldots, (x, |N_x|, (\overbrace{y, \ldots, z}^{|N_x|}))}_{|A|}), \tag{12}$$

where $(u, |N_u|, (v, \ldots, w))$ describes the neighbors of node $u \in A$.

Dividing and Merging Steps: First, each iteration of the dividing and merging steps (Algorithms 2 and 3) is performed by the following MapReduce job:

- **Map-1**: The hash function h is broadcast to the mappers. Each mapper repeats taking a record $R(A)$, computing $F(A)$ (lines 3-5 of Algorithm 2), and emitting $< F(A), R(A) >$.
- **Reduce-1:** The supernodes $\mathcal{S}$ are broadcast to the reducers. Each reducer repeats taking $\{R(A) | A \in \mathcal{S}^{(i)}\}$ for a group $\mathcal{S}^{(i)}$, updating $\mathcal{S}^{(i)}$ (lines 2-9 of Algorithm 3), and emitting $R(A)$ for each supernode A in the updated $\mathcal{S}^{(i)}$. In the end, each reducer writes the updates in $\mathcal{S}$ to the distributed file system.

Note that updates in $\mathcal{S}$ are not shared among the reducers during the reduce stage. However, in our experiments, the effect of this lazy synchronization on the compactness of output representations was negligible.

Encoding Step: Next, the following map-only job performs the encoding step (Algorithm 4):

- **Map-2:** The supernodes $\mathcal{S}$ are broadcast to the mappers. Each mapper repeats taking a record $R(A)$, encoding the edges adjacent to any node in A (lines 3-7 of Algorithm 4), and emitting

the new (super) edges in $\mathcal{P}$, C^+, and C^-. Different output paths are used for $\mathcal{P}$, C^+, and C^-.

Dropping Step: Lastly, for the dropping step (Algorithm 5), Map-3 initializes the change limit of each node (line 1); and Map-4 and Reduce-4 sort the superedges in $\mathcal{P}$ (line 9).

- **Map-3**: Each mapper repeats taking a record $R(A)$ and emitting $< v, \epsilon \cdot |N_v| >$ for each node $v \in A$.
- **Map-4**: The input file, which is an output of Map-2, lists the superedges in $\mathcal{P}$. Each mapper repeats taking a superedge $\{A, B\}$ and emitting $< (|A| \cdot |B|, \{A, B\}), \emptyset >$.
- **Reduce-4**: A single reducer repeats taking a superedge $\{A, B\} \in \mathcal{P}$ and emitting it. The superedges are sorted in the shuffle stage.

The other parts of the dropping step are processed serially. Specifically, after loading the nodes' change limits (the output of Map-3) in memory, our implementation repeats reading a (super) edge in C^+ (an output of Map-2), C^- (an output of Map-2), and $\mathcal{P}$ (the output of Reduce-4) and writing the (super) edge to the output file if it is not dropped. Note that entire C^+, C^-, or $\mathcal{P}$ is not loaded in memory at once. These serial parts take a small portion of the total execution time because they are executed only once, while the dividing and merging steps are repeated multiple times.

3.4 Complexity Analysis

We analyze the time complexity and memory requirements of SWeG. To this end, we let $\mathcal{E}_A := \{\{u, v\} \in \mathcal{E} : u \in A\}$ be the set of edges adjacent to any node in supernode $A \in \mathcal{S}$ and $\mathcal{E}^{(i)} := \bigcup_{A \in \mathcal{S}^{(i)}} \mathcal{E}_A$ be the set of edges adjacent to any node in any supernode in group $\mathcal{S}^{(i)}$. We also let k be the number of groups of supernodes from the dividing step. Since the groups are disjoint,

$$\sum_{i=1}^{k} |\mathcal{E}^{(i)}| \leq \sum_{A \in \mathcal{S}} |\mathcal{E}_A| \leq 2|\mathcal{E}|. \tag{13}$$

Since the encoding step (Algorithm 4) yields outputs with no more (super) edges than ($\mathcal{P} \leftarrow \emptyset, C^+ \leftarrow \mathcal{E}, C^- \leftarrow \emptyset$),

$$|C^+| + |C^-| + |\mathcal{P}| \leq |\mathcal{E}|. \tag{14}$$

Lastly, we assume that $|\mathcal{V}| \leq |\mathcal{E}|$, for simplicity.

3.4.1 Time Complexity. The dividing step (Algorithm 2) takes $O(|\mathcal{E}|)$ time since its computational bottleneck is to compute the shingles of all supernodes (lines 2-5), which requires accessing every edge in $\mathcal{E}$ twice. The merging step (Algorithm 3) takes $O(\sum_{i=1}^{k} |\mathcal{S}^{(i)}| \cdot |\mathcal{E}^{(i)}|)$ time, since when each group $\mathcal{S}^{(i)}$ is processed (lines 2-9), the number of iterations is $|\mathcal{S}^{(i)}| - 1$ and each iteration takes $O(|\mathcal{E}^{(i)}|)$ time. The encoding step (Algorithm 4) takes $O(\sum_{A \in \mathcal{S}} |\mathcal{E}_A|) = O(|\mathcal{E}|)$ time since to process each supernode A (lines 3-7) takes $O(|\mathcal{E}_{AA}| + \sum_{B(\neq A):\mathcal{E}_{AB} \neq \emptyset} |\mathcal{E}_{AB}|) = O(|\mathcal{E}_A|)$ time. This follows from $|\pi_{AB}| < 2 \cdot |\mathcal{E}_{AB}|$ and $|\pi_{AA}| < 2 \cdot |\mathcal{E}_{AA}|$ in lines 5 and 7, which are due to the conditions in lines 4 and 6. Lastly, the dropping step (Algorithm 5) takes $O(|\mathcal{E}| + |C^+| + |C^-| + |\mathcal{P}|) = O(|\mathcal{E}|)$ time (see Eq. (14)). Note that $\mathcal{P}$ (line 8) can be sorted in $O(|\mathcal{P}| + |\mathcal{E}|)$ time using any linear-time integer sort (e.g., counting sort) since $|A| \cdot |B| \in \{1, ..., 2 \cdot |\mathcal{E}|\}$ for every $\{A, B\} \in \mathcal{P}$ due to the conditions in lines 4 and 6 of Algorithm 4. Thus, all the steps except for the merging step take $O(|\mathcal{E}|)$ time. The merging step, whose complexity is $O(\sum_{i=1}^{k} |\mathcal{S}^{(i)}| \cdot |\mathcal{E}^{(i)}|)$, also takes $O(|\mathcal{E}|)$ time if $\mathcal{S}$ is divided finely in the dividing step so that the size of each group is less than a constant (see Eq. (13)). In such cases, the overall time complexity of SWeG (Algorithm 1) is $O(T \cdot |\mathcal{E}|)$ since the dividing and merging steps are repeated T times. This linear scalability is shown experimentally in Section 4.4.

Algorithm 6 SWeG+: Algorithm for Further Compression

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of iterations T, error bound ϵ
graph-compression method ALG
Output: compressed $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, C^+)$, and $\mathcal{G}^- = (\mathcal{V}, C^-)$
1: run SWeG to get $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $C =< C^+, C^- >$ ▷ Algorithm 1
2: run ALG on each of $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, C^+)$, and $\mathcal{G}^- = (\mathcal{V}, C^-)$
3: **return** the compressed $\overline{\mathcal{G}}$, $\mathcal{G}^+$, and $\mathcal{G}^-$

Table 2: Summary of the real-world datasets that we used.

Name	# Nodes	# Edges	Summary
Caida (CA) [27]	26,475	53,381	Internet
Protein (PR) [18]	6,229	146,160	Protein Interaction
Email (EM) [21]	36,692	183,831	Email
DBLP (DB) [56]	317,080	1,049,866	Collaboration
Amazon (AM) [26]	403,394	2,443,408	Co-purchase
Youtube (YO) [37]	1,134,890	2,987,624	Social
Skitter (SK) [27]	1,696,415	11,095,298	Internet
Web-Small (WS) [5]	862,664	16,138,468	Hyperlinks
Patent (PA) [15]	3,774,768	16,518,947	Citations
LiveJournal (LJ) [56]	3,997,962	34,681,189	Social
Hollywood (HO) [5]	1,985,306	114,492,816	Collaboration
Web-Large (WL) [5]	39,454,463	783,027,125	Hyperlinks
LinkedIn (LI)	> 600 millions	> 20 billions	Social

3.4.2 Memory Requirements. The space required for storing $\mathcal{S}$, $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$, $\{h(v) : v \in \mathcal{V}\}$, and $\{c_v : v \in \mathcal{V}\}$ is $O(|\mathcal{V}|)$, and the space required for storing $\mathcal{G}$, $\overline{\mathcal{G}}$, and C is $O(|\mathcal{E}|)$ (see Eq. (14)). Thus, in shared-memory settings, where all of them are stored in memory, the memory requirements of SWeG (Algorithm 1) are $O(|\mathcal{V}| + |\mathcal{E}|) = O(|\mathcal{E}|)$. In MapReduce settings, as described in Section 3.3, each mapper or reducer requires $O(|\mathcal{V}|)$ memory, which is used to store $\mathcal{S}$, $\{h(v) : v \in \mathcal{V}\}$, or $\{c_v : v \in \mathcal{V}\}$, in all the stages except for Reduce-1. In Reduce-1, in addition to $O(|\mathcal{V}|)$ memory for $\mathcal{S}$, each reducer requires $O(\max_{1 \leq i \leq k} |\mathcal{E}^{(i)}|)$ memory to load $\{R(A) : A \in \mathcal{S}^{(i)}\}$ for each group $\mathcal{S}^{(i)}$ at once. Thus, the memory requirements are $O(|\mathcal{V}| + \max_{1 \leq i \leq k} |\mathcal{E}^{(i)}|)$ per reducer. In real-world graphs, $\max_{1 \leq i \leq k} |\mathcal{E}^{(i)}|$ is much smaller than $|\mathcal{E}|$, as shown experimentally in Appendix B.

3.5 Further Compression: SWeG+

We propose SWeG+, an algorithm for further compression. The outputs of SWeG (i.e., $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $C =< C^+, C^- >$) can be represented as three graphs: $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, C^+)$, and $\mathcal{G}^- = (\mathcal{V}, C^-)$. As described in Algorithm 6, SWeG+ further compresses each of the graphs using a given graph-compression method ALG. Any graph-compression method, such as [2, 5, 7, 9, 11, 43], can be used as ALG depending on the objectives of compression.[1] In Section 4.6, we empirically show that for many graph-compression methods, SWeG+ gives significantly more compact representations than directly compressing the input graph using the methods.

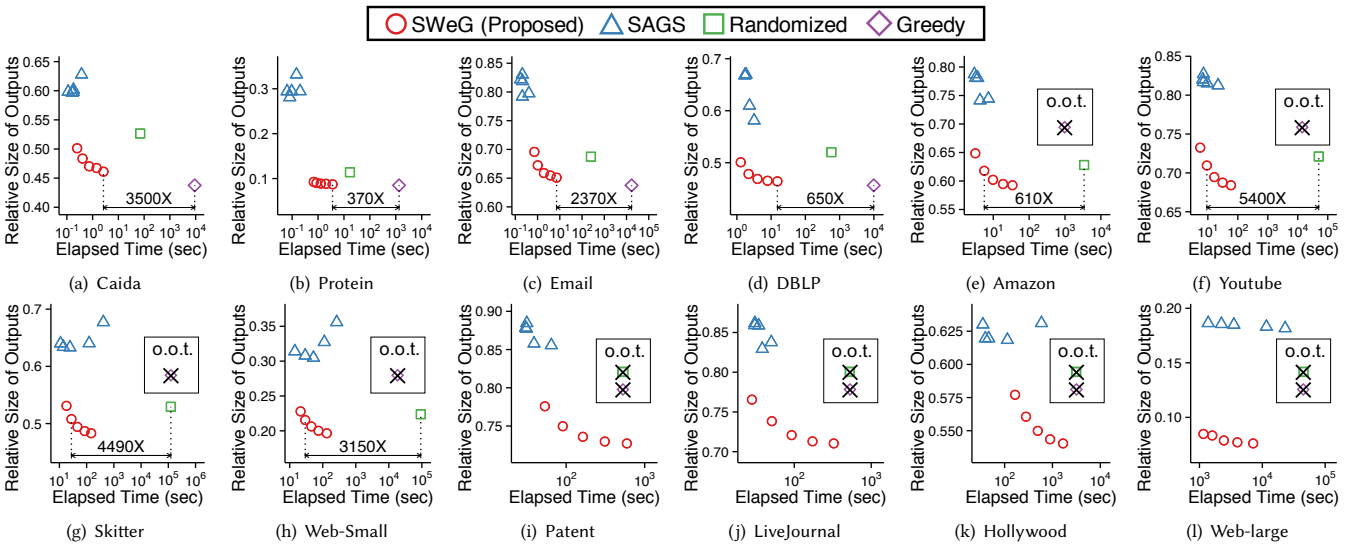


Figure 3: SWeG (lossless and shared-memory) significantly outperforms existing lossless summarization methods. o.o.t.: out of time (> 48 hours). Specifically, SWeG was up to $5,400\times$ faster than the others that give similarly compact outputs.

4 EXPERIMENTS

We review our experiments for answering the following questions:

- **Q1. Lossless Summarization**: Does the lossless version of SWeG yield more compact representations faster than its competitors?
- **Q2. Lossy Summarization**: Does the lossy version SWeG yield more compact and accurate representations than baselines?
- **Q3. Scalability**: How well does SWeG scale as the size of the input graph, the number of machines, and the number of cores grow?
- **Q4. Effects of Parameters**: How do the number of iterations T and the error bound ϵ affect the compactness of outputs?
- **Q5. Further Compression**: How much does SWeG+ improve the compression rates of combined compression methods?

4.1 Experimental Settings

Machines: We ran single instance experiments on a machine with 2.10GHz Intel Xeon E6-2620 CPUs (with 6 cores) and 64GB memory. We ran MapReduce experiments on a private Hadoop cluster.

Datasets: We used the graphs listed in Table 2. We ignored the direction of edges in all of them.

Implementations: We implemented all the considered algorithms in Java 1.8. We implemented the shared-memory version of SWeG using standard Java multithreading, and we set the number of threads to 8 unless otherwise stated. We implemented the MapReduce version of SWeG using Hadoop 2.6.1, and we set the numbers of mappers and reducers to 40 unless otherwise stated. In both implementations, we ran the dividing step recursively so that each group had at most 500 supernodes, as described in Section 3.1.2.

Evaluation Metric: Given an output representation $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $C = < C^+, C^- >$ of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we measured its compactness using the *relative size of outputs*, defined as

$$(|\mathcal{P}^*| + |C^+| + |C^-|)/|\mathcal{E}|, \tag{15}$$

where the numerator is our objective function (i.e., Eq. (1)) and the denominator is a constant for a given input graph.

[1] If ALG relabels nodes, to keep the labels of $\mathcal{V}$ in $\mathcal{G}^+$ and $\mathcal{G}^-$ the same, the labels obtained when compressing $\mathcal{G}^+$ are used for $\mathcal{G}^-$, which tends to be smaller than $\mathcal{G}^+$.

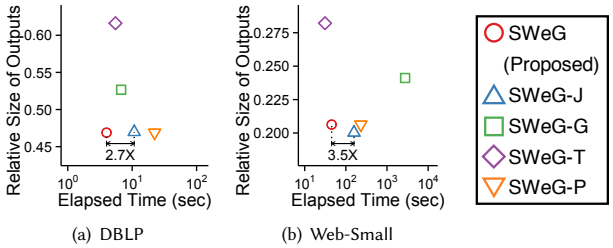


Figure 4: SWeG (lossless and shared-memory) significantly outperforms its variants. This justifies our design choices.

4.2 Q1. Lossless Summarization

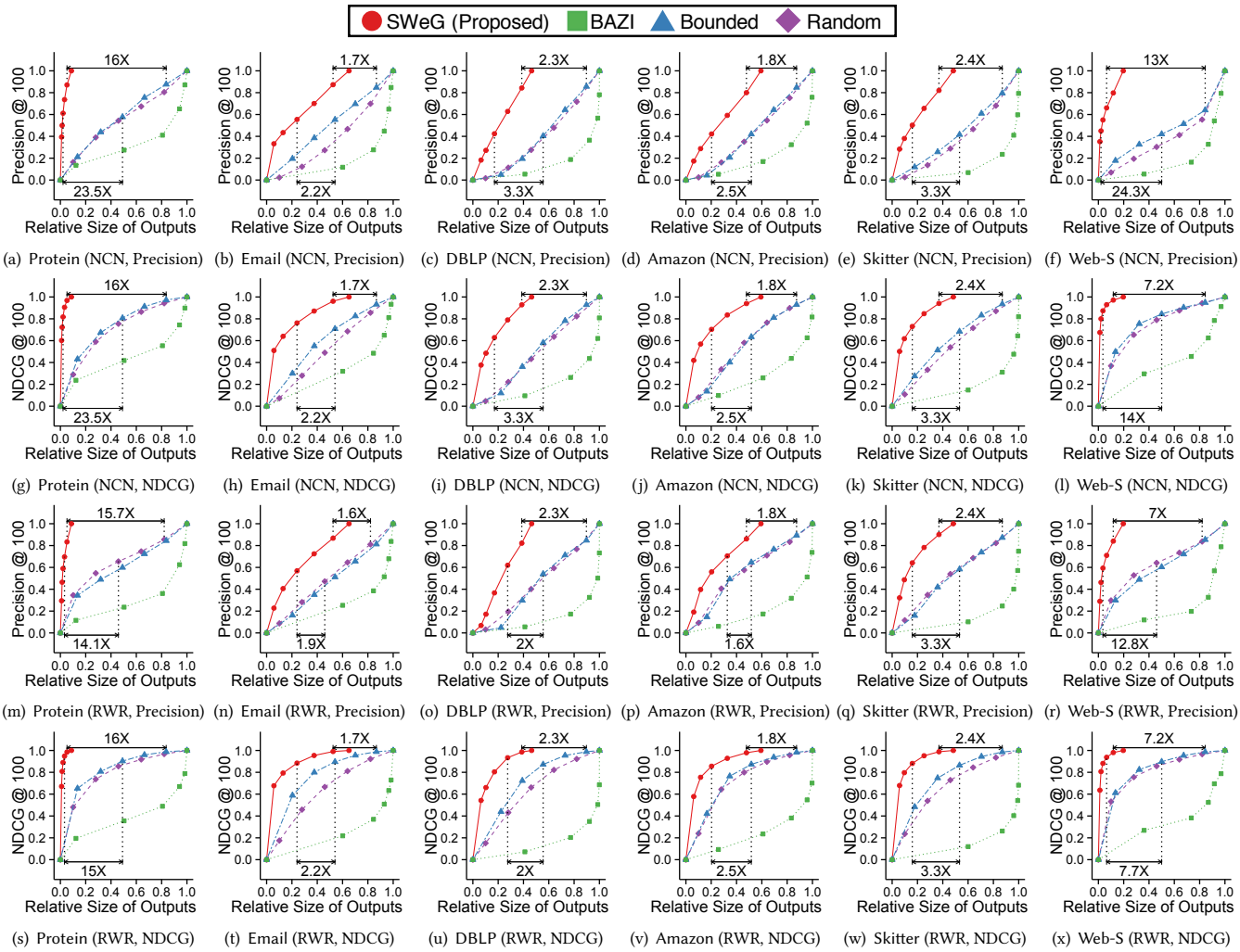
We compared the following lossless graph-summarization methods in terms of speed and compactness of representations: [2]

(a) SWeG (proposed): the shared-memory and lossless version of SWeG with $\epsilon = 0$ and $T = \{5, 10, 20, 40, 80\}$.
(b) SAGS [20]: SAGS with five different parameter settings. [3]
(c) Randomized [38].
(d) Greedy [38].
(e) SWeG-P: a variant of (a) without **p**arallelization.
(f) SWeG-G: a variant of (a) that does not **g**roup the supernodes.
(g) SWeG-J: a variant of (a) that chooses supernodes to merge based on the exact savings instead of **j**accard similarity.
(h) SWeG-T: a variant of (a) where the merging **t**hresohld $\theta(t)$ is 0.

To this end, we measured the elapsed time and the relative size of the outputs (i.e., Eq. (15)) of each algorithm.

SWeG gave the best trade-off between speed and compactness of outputs on all the datasets, as seen in Figure 3. For example, on the Youtube dataset, SWeG was **5,400× faster** with 2% smaller outputs than Randomized and 18% faster with 10% smaller outputs than SAGS. Greedy did not terminate within a reasonable

[2] We slightly modified (b)-(d) so that they aim to minimize the same objective of SWeG (i.e., Eq. (1)). We fixed T to 20 in (e)-(h). (b)-(e) are serial, and the others are parallel.
[3] $\{h = 28, b = 7, p = 0.3\}$, $\{h = 28, b = 7, p = 0.1\}$, $\{h = 28, b = 7, p = 0.2\}$, $\{h = 30, b = 10, p = 0.3\}$,$\{h = 30, b = 15, p = 0.3\}$. p denotes the overlap ratio.



(a) Protein (NCN, Precision) (b) Email (NCN, Precision) (c) DBLP (NCN, Precision) (d) Amazon (NCN, Precision) (e) Skitter (NCN, Precision) (f) Web-S (NCN, Precision)

(g) Protein (NCN, NDCG) (h) Email (NCN, NDCG) (i) DBLP (NCN, NDCG) (j) Amazon (NCN, NDCG) (k) Skitter (NCN, NDCG) (l) Web-S (NCN, NDCG)

(m) Protein (RWR, Precision) (n) Email (RWR, Precision) (o) DBLP (RWR, Precision) (p) Amazon (RWR, Precision) (q) Skitter (RWR, Precision) (r) Web-S (RWR, Precision)

(s) Protein (RWR, NDCG) (t) Email (RWR, NDCG) (u) DBLP (RWR, NDCG) (v) Amazon (RWR, NDCG) (w) Skitter (RWR, NDCG) (x) Web-S (RWR, NDCG)

Figure 5: SWeG (lossy) significantly outperforms baseline methods for lossy graph summarization. Specifically, SWeG yielded up to 24.3× more compact and similarly accurate representations than the other methods.

time (48 hours). In Figure 4, we show that SWeG (with $T = 20$) also outperformed its variants, justifying our design choices in Section 3.

4.3 Q2. Lossy Summarization

We compared the following lossy graph-summarization methods in terms of the compactness and accuracy of output representations:

(a) SWeG (proposed): the shared-memory and lossy version of SWeG with $T = 80$ and $\epsilon = \{0, 0.18, 0.36, 0.54, 0.72, 0.9\}$.
(b) BAZI [3]: BAZI with $s = \log^2 |\mathcal{S}|$, $w = 50$, and $k = \{0.1 \cdot |\mathcal{V}|, 0.28 \cdot |\mathcal{V}|, 0.46 \cdot |\mathcal{V}|, 0.64 \cdot |\mathcal{V}|, 0.82 \cdot |\mathcal{V}|, |\mathcal{V}|\}$.
(c) Bounded: a baseline that performs only the dropping step of SWeG, i.e., SWeG with $T = 0$ and $\epsilon = \{0.18, 0.36, 0.54, 0.72, 0.9\}$.
(d) Random: a baseline that randomly drops $\epsilon = \{0.18, 0.36, 0.54, 0.72, 0.9\}$ of the edges from the input graph.

To this end, for each method, we measured the relative size of outputs (e.g., Eq. (15)) and measured how accurately the outputs preserve the relevances between nodes as follows:

S1. randomly choose 100,000 seed nodes in the input graph.
S2. compute the **true relevances** between each seed node and the other nodes in the input graph.
S3. compute the **approximate relevances** between each seed node and the other nodes in the graph restored from the outputs.
S4. measure how accurate the approximate relevances from S3 are.

In **S2** and **S3**, we used one of the following relevance scores:

- *Random Walk with Restart (RWR)* [53]: each node's RWR score with respect to a seed node is defined as the stationary probability that a random surfer is at the node. The random surfer either moves to a neighboring node of the current node (with probability 0.8) or restarts at the seed node (with probability 0.2).
- *Number of Common Neighbors (NCN)*: each node's NCN score with respect to a seed node is defined as the number of common neighbors of the node and the seed node.

In **S4**, we computed one of the following accuracy measures for every seed node and then averaged them.

- *Precision@100*: Precision@100 is defined as the fraction of the 100 most relevant nodes in terms of the true relevances among those in terms of the approximate relevances.

- *NDCG@*100 [17]: Let $r(i)$ be the true relevance of the i-th most relevant node in terms of the approximate relevances. Then,

$$\text{NDCG@100} := \frac{1}{Z}\sum_{i=1}^{100}\frac{2^{r(i)}}{\log_2(1+i)},$$

where Z is a constant normalizing NDCG@100 to be within $[0, 1]$.

SWeG yielded the most compact and accurate representations on all the considered datasets, as seen in Figure 5. For example, on the Web-Small dataset, SWeG gave a **24.3× more compact** and similarly accurate representation than the other methods. BAZI tended to yield representations with most (super) edges since it aims to reduce the number of supernodes instead of minimizing the number of (super) edges (see Section 5).

4.4 Q3. Scalability

We evaluated the scalability of the shared-memory and MapReduce implementations of SWeG. Specifically, we measured how rapidly their running times change depending on the size of the input graph, the number of threads, and the number of machines (i.e., the numbers of mappers and reducers in the MapReduce framework). In the shared-memory setting, we used graphs with different sizes obtained by sampling different numbers of nodes from the Web-Large dataset. In the MapReduce setting, we used graphs obtained in the same manner using the LinkedIn dataset. When measuring the data scalability, we fixed the number of threads to 8 and the number of machines to 40. When measuring the machine and multi-core scalability, we fixed the size of the input graph.

SWeG scaled linearly with the size of the input graph, as seen in Figures 6(a) and 6(b). The lossless summarization by SWeG as well as the additional dropping step for lossy summarization (with $\epsilon = 0.1$) scaled near linearly with the number of edges in the input graph in both settings. Note that the largest graph used had more than **20 billion edges**.

SWeG achieved significant speedup in the shared-memory and MapReduce settings. As seen in Figure 6(c), the speedup, defined as T_1/T_N where T_N is the running time of SWeG with N threads, increased near linearly with the number of threads in the shared-memory setting. Specifically, SWeG provided a speedup of 3.3 with 4 threads and 5.7 with 8 threads. As seen in Figure 6(d), the speedup of SWeG, defined as T_1/T_N where T_N is the running time of SWeG with N machines, increased near linearly with the number of machines in the MapReduce setting. Specifically, SWeG provided a speedup of 8.3 with 10 machines and 26.4 with 40 machines.

4.5 Q4. Effects of Parameters

We measured how the number of iterations T and the error bound ϵ in SWeG affect the compactness of its output representation using the relative size of outputs (i.e., Eq. (15)). When measuring the effect of T, we fixed ϵ to 0 and changed T from 1 to 80. When measuring the effect of ϵ, we fixed T to 80 and changed ϵ from 0 to 0.5.

The larger the number of iterations, the more compact the output representation. As seen in Figure 7, the size of outputs decreased over iterations and eventually plateaued. **The larger the error bound, the more compact the output representation**. As seen in Figure 8, the size of outputs decreased near linearly as the error bound increased. The relative size of outputs was especially small in web graphs and protein-interaction graphs, where nodes tend to have similar connectivity [10, 23].

4.6 Q5. Further Compression

We measured how much SWeG+ improves the compression rates of the following advanced graph-compression methods:

(a) BP [11]: reordering nodes as suggested in [11] with 20 iterations and using the webgraph framework [5].
(b) Shingle [9]: reordering nodes as suggested in [9] and using the webgraph framework [5].
(c) BFS [2]: running BFS with the default parameter setting at https://github.com/drovandi/GraphCompressionByBFS.
(d) VNMiner [7]: running VNMiner with 80 iterations.

For the webgraph framework in (a)-(b), we used the default parameter setting (i.e., $r = 3$, $W = 7$, $L_{min} = 7$, and ζ_3). We measured the compression rates using the objective function of each compression method. That is, we used the number of bits per directed edge[4] for (a)-(c) and the relative size of outputs (i.e., Eq. (15)) for (d).

SWeG+ achieved significant further compression. As seen in Figures 1(d) and 9, the lossless version of SWeG+ with $T = 80$ and $\epsilon = 0$ yielded up to **3.4× more compact** representations than all the input compression methods on all the datasets. Especially, SWeG+ with ALG = BFS represented the Web-Large dataset using less than **0.7 bits per directed edge** without loss of information.

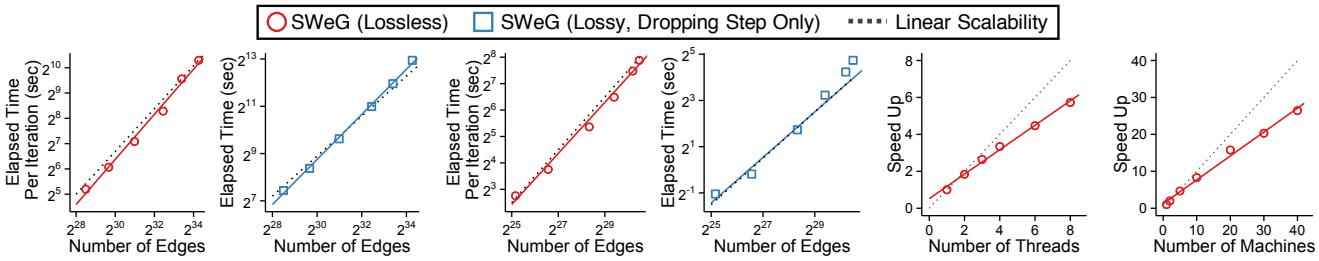
5 RELATED WORK

While this paper addresses Problem 1, the term "graph summarization" has been used for a wider range of problems related to concisely describing static plain graphs [12, 22, 28, 33–35, 38, 39, 42, 46, 52, 58], static attributed graphs [8, 13, 16, 19, 41, 47, 49, 51, 55, 57], and dynamic graphs [1, 29, 40, 45, 50]. We refer the reader to an excellent survey [32] for a review on these problems. In this section, we focus on previous work closely related to Problem 1.

Lossless Graph Summarization. Greedy [38] repeatedly finds and merges a pair of supernodes so that savings in space are maximized given the other supernodes. If there is no pair whose merger leads to non-negative savings, then Greedy creates a summary graph and corrections so that the number of (super) edges is minimized given the current supernodes. Greedy is computationally and memory expensive because it maintains and updates savings for $O(|\mathcal{V}|^2)$ pairs of supernodes. Without maintaining any precomputed savings, Randomized [38], whose time complexity is $O(|\mathcal{V}| \cdot |\mathcal{E}|)$, randomly chooses a supernode first and then chooses another supernode to be merged so that savings in space are maximized given the other supernodes. SAGS [20] chooses supernodes to be merged using locality sensitive hashing without computing savings in space, which are expensive to compute. We empirically show in Section 4.2 that all these serial algorithms are not satisfactory in terms of speed or compactness of outputs. They either have high computational complexity or significantly sacrifice compactness of outputs for lower complexity. More importantly, they cannot handle large-scale graphs that do not fit in main memory.

Lossy Graph Summarization. Apxmdl [38] reduces the problem of choosing edges to be dropped from outputs of Greedy to a maximum b-matching problem and uses GabowâĂŹs algorithm [14], whose time complexity is $O(\min(|\mathcal{E}|^2 \log|\mathcal{V}|, |\mathcal{E}| \cdot |\mathcal{V}|^2))$. Due to this high computational complexity, Apxmdl does not scale

[4]We regarded undirected graphs as symmetric directed graphs.



(a) Data Scalability (MapReduce) (b) Data Scalability (Shared Memory) (c) Multi-Core Scalability (Shared Memory) (d) Machine Scalability (MapReduce)

Figure 6: SWeG is scalable. (a-b) SWeG scaled linearly with the size of the input graph. (c-d) SWeG achieved significant speedup as more machines and CPU cores were used. Note that the largest graph used had more than 20 billion edges.

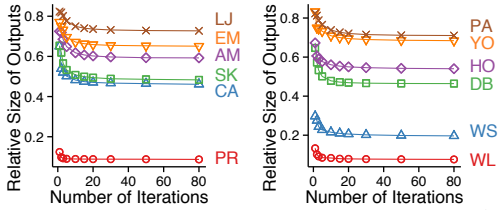


Figure 7: As the number of iterations in SWeG (lossless) increases, the output representations become compact.

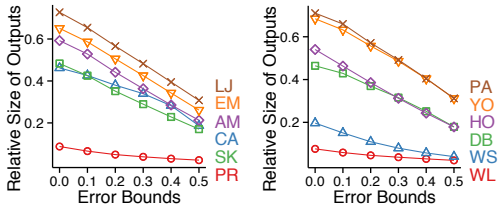


Figure 8: As the error bound in SWeG (lossy) increases, the output representations become compact.

even to moderately-sized graphs. For the same problem, the dropping step of SWeG takes $O(|\mathcal{E}|)$ time (see Section 3.4.1) without sacrificing the compactness of outputs (see Appendix C). In [38], combining Greedy and Apxmdl into a single step was also discussed. Several algorithms [3, 25, 31, 42] have been developed for a problem that is similar but not identical to Problem 1. They aim to find a weighted summary graph with a given number of supernodes so that the difference between the original and restored graphs is minimized without edge corrections. The way of restoring a graph is also different from that in Problem 1. Since these algorithms aim to reduce the number of supernodes, instead of (super) edges, they are not effective for Problem 1 (see Section 4.3).

Combination with Other Compression Techniques. In addition to graph summarization, numerous graph-compression techniques have been developed, including relabeling nodes [2, 5, 9, 11], utilizing encoding schemes for integer sequences (e.g., reference, gap, and interval encodings) [5], and encoding common structures (e.g., cliques, bipartite-cores, and stars) with fewer bits [7, 22, 43]. See [4] for a comprehensive survey on these techniques. As described in Section 3.5 and Section 4.6, SWeG is readily combinable with any compression technique for static plain graphs. In [44], tightly combining two specific summarization and compression algorithms [38, 54] into a single process was presented.

Algorithm 7 Neighbor Query Processing on $\overline{\mathcal{G}}$ and C

Input: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $C =< C^+, C^- >$, query node $v \in \mathcal{V}$
Output: the set $\hat{N}_v$ of v's neighbors in the restored graph $\hat{\mathcal{G}}$

```
N̂_v ← ∅; S_v ← the supernode in S where v ∈ S_v
if S_v has a self-loop in Ḡ then N̂_v ← N̂_v ∪ (S_v − {v})
for each neighbor A (≠ S_v) of S_v in Ḡ do N̂_v ← N̂_v ∪ A
N̂_v ← (N̂_v ∪ N_v^+) − N_v^−        ▷ N_v^+: v's neighbors in C^+
return N̂_v                          ▷ N_v^− : v's neighbors in C^−
```

6 CONCLUSIONS

We propose SWeG, a fast parallel algorithm for lossless and lossy summarization of large-scale graphs, which may not fit in main memory. We present efficient implementations of SWeG in shared-memory and MapReduce environments. We also propose SWeG+ where SWeG and other graph-compression methods are combined to achieve better compression than individual methods. We theoretically and empirically show the following strengths of SWeG:

- **Speed**: SWeG provides similarly compact representations up to *5, 400× faster* than existing summarization methods (Figure 3).
- **Scalability**: SWeG scales near linearly with the size of the input graph, the number of machines, and the number of CPU cores. It successfully scales to graphs with over *20 billion* edges (Figure 6).
- **Compression**: SWeG+ achieves up to *3.4× better* compression than individual state-of-the-art graph-compression methods that are combined with SWeG (Figures 1(d) and 9).

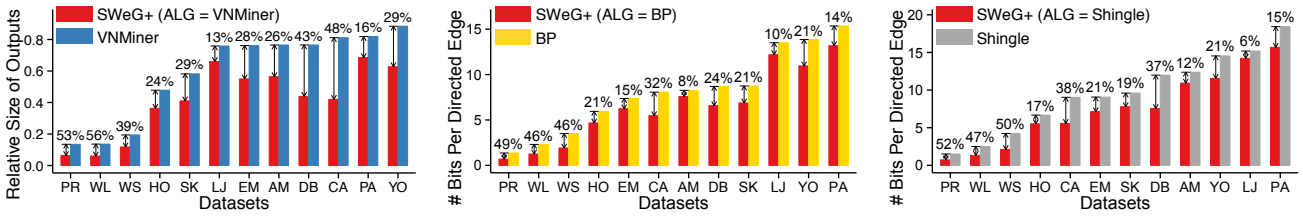
A APPENDIX: NEIGHBOR QUERIES

Algorithm 7 describes how to answer neighbor queries (i.e., finding the neighbors $\hat{N}_v$ of a given node $v \in \mathcal{V}$) efficiently on a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $C =< C^+, C^- >$ without restoring entire $\hat{\mathcal{G}}$. Let N_v^- be the neighbors of node $v \in \mathcal{V}$ in C^-. If there is no redundant edge[5] in C^+ and we use a hash table for $\hat{N}_v$ and adjacency lists for C^+, C^-, and $\mathcal{P}$, then the running time of Algorithm 7 is proportional to $|\hat{N}_v| + 2|N_v^-|$. In every dataset listed in Table 2 in Section 4, when SWeG ($T = 80$) was used, $|C^-|$ was at most 6% of the number of edges in the input graph, regardless of the error bound ϵ. Thus, since $|\hat{N}_v| \leq (1+\epsilon) \cdot |N_v|$ from Eq. (2), $|\hat{N}_v| + 2|N_v^-|$ was at most $(1.12+\epsilon) \cdot |N_v|$ on average.

B APPENDIX: SIZE OF SUBGRAPHS

We measured the size of the largest subgraphs that need to be loaded in main memory at once in our MapReduce implementation of SWeG (with $T = 80$). As seen in Figure 10, the largest subgraphs had

[5] As in the outputs of SWeG, if $\{A, B\} \in \mathcal{P}$, $u \in A$, and $v \in B$, then $\{u, v\} \notin C^+$.



(a) SWeG+ (lossless) vs. VNMiner (b) SWeG+ (lossless) vs. BP (c) SWeG+ (lossless) vs. Shingle

Figure 9: SWeG+ (lossless) significantly improves the compression rates of state-of-the-art graph-compression algorithms.

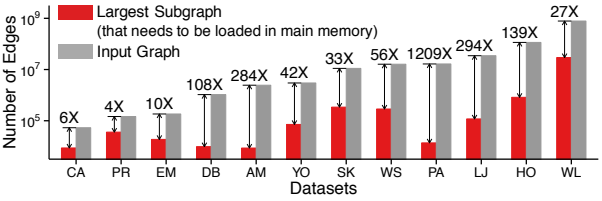


Figure 10: Subgraphs that SWeG (MapReduce) loads in memory at once are significantly smaller than input graphs.

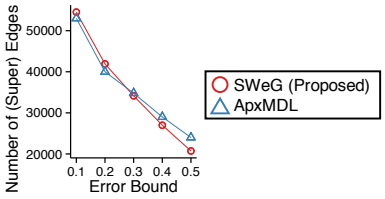


Figure 11: While SWeG has much lower time complexity than Apxmdl, they give similarly compact results.

up to **1,209× fewer** edges than the entire graphs. The experimental settings were the same with those in Section 4.1.

C APPENDIX: COMPARISON WITH APXMDL

To compare SWeG and Apxmdl [38] in terms of the compactness of their output representations, we measured how the number of (super) edges in their outputs, which Apxmdl aims to minimize, changes depending on the error bound ϵ. We used the same small-scale graph used in [38], which is a web graph with 40,000 nodes [5]. SWeG (with $T = 80$) and Apxmdl yielded similarly compact outputs, as seen in Figure 11, where we compare with the numbers reported in [38]. Specifically, SWeG gave more compact outputs than Apxmdl when ϵ was large, while the opposite happened when ϵ was small. SWeG has much lower time complexity than Apxmdl. Specifically, the dropping step of Apxmdl takes $O(\min(|\mathcal{E}|^2 \log |\mathcal{V}|, |\mathcal{E}| \cdot |\mathcal{V}|^2))$ time, while that of SWeG takes $O(|\mathcal{E}|)$ time (see Section 3.4.1).

REFERENCES

[1] Bijaya Adhikari, Yao Zhang, Aditya Bharadwaj, and B Aditya Prakash. 2017. Condensing temporal networks using propagation. In *SDM*.
[2] Alberto Apostolico and Guido Drovandi. 2009. Graph compression by BFS. *Algorithms* 2, 3 (2009), 1031–1044.
[3] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. 2018. Scalable Approximation Algorithm for Graph Summarization. In *PAKDD*.
[4] Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *arXiv preprint arXiv:1806.01799* (2018).
[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*.
[6] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 2000. Min-wise independent permutations. *J. Comput. System 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*.
[8] Chen Chen, Cindy X Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. 2009. Mining graph patterns efficiently via randomized summaries. *PVLDB* 2, 1 (2009), 742–753.
[9] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *KDD*.
[10] Fan Chung, Linyuan Lu, T Gregory Dewey, and David J Galas. 2003. Duplication models for biological networks. *Journal of Computational Biology* 10, 5 (2003), 677–687.
[11] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *KDD*.
[12] Cody Dunne and Ben Shneiderman. 2013. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *CHI*.
[13] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified top-k graph pattern matching. *PVLDB* 6, 13 (2013), 1510–1521.
[14] Harold N. Gabow. 1983. An Efficient Reduction Technique for Degree-constrained Subgraph and Bidirected Network Flow Problems. In *STOC*.
[15] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. 2001. *The NBER patent citation data file: Lessons, insights and methodological tools.* Technical Report. National Bureau of Economic Research.
[16] Nasrin Hassanlou, Maryam Shoaran, and Alex Thomo. 2013. Probabilistic graph summarization. In *WAIM*.
[17] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *TOIS* 20, 4 (2002), 422–446.
[18] G Joshi-Tope, Marc Gillespie, Imre Vastrik, Peter D'Eustachio, Esther Schmidt, Bernard de Bono, Bijay Jassal, GR Gopinath, GR Wu, Lisa Matthews, et al. 2005. Reactome: a knowledgebase of biological pathways. *Nucleic Acids Research* 33, suppl_1 (2005), D428–D432.
[19] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. 2014. Set-based unified approach for attributed graph summarization. In *BdCloud*.
[20] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. 2015. Set-based approximate approach for lossless graph summarization. *Computing* 97, 12 (2015), 1185–1207.
[21] Bryan Klimt and Yiming Yang. 2004. The enron corpus: A new dataset for email classification research. In *ECML*.
[22] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. 2014. VOG: Summarizing and Understanding Large Graphs. In *SDM*.
[23] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D Sivakumar, Andrew Tomkins, and Eli Upfal. 2000. Stochastic models for the web graph. In *FOCS*.
[24] Douglas Lea. 2000. *Concurrent programming in Java: design principles and patterns.* Addison-Wesley Professional.
[25] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *SDM*.
[26] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *TWEB* 1, 1 (2007), 5.
[27] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007), 2.
[28] Cheng-Te Li and Shou-De Lin. 2009. Egocentric information abstraction for heterogeneous social networks. In *ASONAM*.
[29] Yu-Ru Lin, Hari Sundaram, and Aisling Kelliher. 2008. Summarization of social activity over time: people, actions and concepts in dynamic networks. In *CIKM*.
[30] Chao Liu, Fan Guo, and Christos Faloutsos. 2009. Bbm: bayesian browsing model from petabyte-scale data. In *KDD*.
[31] Xingjie Liu, Yuanyuan Tian, Qi He, Wang-Chien Lee, and John McPherson. 2014. Distributed graph summarization. In *CIKM*.
[32] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *CSUR* 51, 3 (2018), 62.
[33] Antonio Maccioni and Daniel J Abadi. 2016. Scalable pattern matching over compressed graphs via dedensification. In *KDD*.
[34] Michael Mathioudakis, Francesco Bonchi, Carlos Castillo, Aristides Gionis, and Antti Ukkonen. 2011. Sparsification of influence networks. In *KDD*.

[35] Yasir Mehmood, Nicola Barbieri, Francesco Bonchi, and Antti Ukkonen. 2013. Csi: Community-level social influence analysis. In *ECML/PKDD*.

[36] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph structure in the web—revisited: a trick of the heavy tail. In *WWW*.

[37] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *IMC*.

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

[39] Manish Purohit, B Aditya Prakash, Chanhyun Kang, Yao Zhang, and VS Subrahmanian. 2014. Fast influence-based coarsening for large networks. In *KDD*.

[40] Qiang Qu, Siyuan Liu, Christian S Jensen, Feida Zhu, and Christos Faloutsos. 2014. Interestingness-driven diffusion process summarization in dynamic networks. In *ECML/PKDD*.

[41] Sriram Raghavan and Hector Garcia-Molina. 2003. Representing web graphs. In *ICDE*.

[42] Matteo Riondato, David García-Soriano, and Francesco Bonchi. 2017. Graph summarization with quality guarantees. *Data Mining and Knowledge Discovery* 31, 2 (2017), 314–349.

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

[44] Hojin Seo, Kisung Park, Yongkoo Han, Hyunwook Kim, Muhammad Umair, Kifayat Ullah Khan, and Young-Koo Lee. 2018. An effective graph summarization and compression technique for a large-scaled graph. *The Journal of Supercomputing* (2018), 1–15.

[45] Neil Shah, Danai Koutra, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. 2015. Timecrunch: Interpretable dynamic graph summarization. In *KDD*.

[46] Zeqian Shen, Kwan-Liu Ma, and Tina Eliassi-Rad. 2006. Visual analysis of large heterogeneous social networks by semantic and structural abstraction. *TVCG* 12, 6 (2006), 1427–1439.

[47] Maryam Shoaran, Alex Thomo, and Jens H Weber-Jahnke. 2013. Zero-knowledge private graph summarization.. In *Big Data*.

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

[49] 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.

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

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

[52] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. 2011. Compression of weighted graphs. In *KDD*.

[53] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *ICDM*.

[54] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *TODS* 31, 1 (2006), 1–38.

[55] Ye Wu, Zhinong Zhong, Wei Xiong, and Ning Jing. 2014. Graph summarization for attributed graphs. In *ISEEE*.

[56] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *KAIS* 42, 1 (2015), 181–213.

[57] Ning Zhang, Yuanyuan Tian, and Jignesh M Patel. 2010. Discovery-driven graph summarization. In *ICDE*.

[58] Linhong Zhu, Majid Ghasemi-Gol, Pedro Szekely, Aram Galstyan, and Craig A Knoblock. 2016. Unsupervised entity resolution on multi-type graphs. In *ISWC*.