Dynamic Graph Summarization: Optimal and Scalable

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

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

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

***Abstract*—Dynamic graph summarization is the task of obtaining and updating a summary of the current snapshot of a dynamic graph when changes (edge insertions/deletions) occur in the graph. As real graphs are massive and undergoing lots of changes, we need dynamic summarization algorithms that scale and are able to respond rapidly to changes in the graph. In this paper, we present two algorithms for lossless summarization of dynamic graphs. We first give an algorithm (Optimal) that is able to obtain and dynamically update the smallest-possible-anytime lossless summary in terms of node reduction. We achieve up to 8 orders of magnitude running time improvement over batch counterparts, and up to 12x improvement over the state-of-art in dynamic graph summarization, while at the same time offering up to 6x improvement in node reduction. We then present an even faster lossless summarization algorithm (Scalable), which goes further into speeding up dynamic updates by offering an additional order of magnitude improvement over Optimal at the cost of having lesser node reduction. Extensive experiments show that Scalable offers node reduction rates that are close to those of Optimal for many datasets. As such, Scalable is a preferred choice when speed of change is very high.**

I. Introduction

We tackle the problem of summarizing massive dynamic graphs that come as a fast stream of edge insertions and deletions [1], [2], [3]. The problem is of paramount importance. Real graphs are massive (e.g. web and social networks) spanning billions of nodes and edges, thus, summarizing them is imperative in order to make graph processing feasible in practice. Real graphs are also highly dynamic, for example, more than 250,000 new web pages and 500,000 new Facebook users are added every day[1] and many millions of links and connections are created every minute. Therefore, we need dynamic graph summarization algorithms that can scale and rapidly respond to changes in the graph.

Graph summarization takes as input a graph and it produces a more compact graph as output [4], [5], [6]. There are many summarization methods, such as graph compression to reduce the volume of input graph [7], graph sparsification to remove less important nodes or edges [8], and group-based graph summarization (GGS) to group similar nodes and edges into supernodes and superedges [9], [10], [11], [12], [13]. GGS is by far the most popular family of methods and our work also belongs in GGS. However, most of the works in GGS consider static graphs, thus ignoring the highly dynamic nature of real graphs. While there are some works on dynamic graph summarization [14], [1], [15], [16], they produce lossy summaries, i.e., salient information in the original graph can be irrecoverably lost.

Recently, Ko et al. in [2] proposed MoSSo, the first lossless dynamic GGS algorithm, which builds on the framework of SWeG [9] for static graphs. However, the summaries of MoSSo and SWeG can only be used via neighborhood queries, i.e., given a node, return its neighbors. This amounts to slowly and incrementally reconstructing the original graph, one node at a time, often multiple times for the same node if the node is requested repeatedly. As such, while these summaries achieve high compression, their utility as data structures to speed up graph analytics is quite limited.

In contrast, the G-SCIS framework, introduced by Hajiabadi et al. in [17], has the advantage of producing summaries that can be used as-is to speed up important classes of graph analytics, such as graphlet enumeration, centrality computation, and shortest paths. Hence, this framework achieves the right tradeoff between compression and utility of the summary. However, the G-SCIS algorithm given in [17] works only for static graphs leading us to ask the following questions: For a dynamic graph stream, is it possible to maintain and update a summary efficiently in the G-SCIS framework? Furthermore, can such summaries be lossless and optimal size or close to optimal size in the G-SCIS framework?

We propose an optimal dynamic lossless summarization algorithm (Optimal) that works in near constant time for each change. Optimal obtains and dynamically updates the smallest-possible-anytime lossless summary in terms of node reduction. We achieve up to 8 orders of magnitude running time improvement over batch counterparts, and up to 12x improvement over MoSSo, while at the same time offering up to 6x improvement in node reduction compared to MoSSo. We then present a second algorithm, Scalable, which offers an additional order of magnitude speed improvement at the cost of having less node reduction than Optimal. Nevertheless, our extensive experiments show that node reduction rates of Scalable are close to those of Optimal and still better than those of MoSSo. As such, Scalable is a good choice when the speed of change is very high. In contrast to MoSSo, which is a randomized algorithm, our Optimal and Scalable

[1] https://siteefy.com/how-many-websites-are-there
https://backlinko.com/facebook-users

algorithms are deterministic and rooted in number theory, i.e., they produce always the same output for a given input.

More specifically, Optimal uses a sort-insensitive hashing scheme to bucketize nodes using their neighbor sets; nodes in same bucket are candidates for merge. Sort-insensitivity allows quick update of the hash value of a node upon an edge change that changes its neighbor set. Typical hash schemes for sets or strings assume a sort order before applying hashing. However, resorting a neighbor set each time a change occurs is impractical for a high rate of changes. We also pay special care to properly identify the set of nodes that need a new supernode home. This is important because the relocation of a node can cause other nodes to relocate too. Nevertheless, we show that the number of affected nodes is never more than four, thus keeping several computations at constant complexity.

The main cost that Optimal incurs is neighbor set equality checks it performs between nodes in the same bucket. Our next algorithm, Scalable, addresses this problem by introducing a hash signature of K sort-insensitive functions, where K is a user-specified integer. Using elementary symmetric polynomials and Newton's identity, we show that our hash signature is such that, for every node of degree less or equal to K, we obtain exactly the same grouping result as Optimal. Nodes with degree greater than K are left alone in singleton supernodes, not merged with other nodes. The bigger the value of K, the more node reduction we obtain, but the slower the algorithm becomes. For K equal to maximum degree in the graph, Scalable produces the same summary as Optimal. However, we do not need to increase K too much to see good summaries. A small value of 20, is sufficient for most datasets to see node reduction rates very close to Optimal. This is because most of group merges happen among nodes of small degree. Finding similar nodes among nodes of higher degree is quite rare. As such, Scalable performs excellently both in term of speed and quality for small values of K.

In summary, we make the following contributions.

- We present Optimal, a fully dynamic algorithm in the G-SCIS framework that provides the smallest-any-time lossless summary of a graph that arrives as a stream of edges. It comes with a complete guarantee of always producing the same summary as the batch counterpart.
- We present Scalable, another fully dynamic algorithm, which is an order of magnitude faster than Optimal at the cost of less node reduction. Scalable still produces lossless summaries, and its node compression rate is close to that of Optimal.
- We conduct an extensive experimental analysis that shows the superiority of our algorithms compared to the batch and dynamic state-of-the-art.
- We present Directed-Scalable, which is an adaption of Scalable for directed graphs. We show that Directed-Scalable exhibits similar characteristics as Scalable with excellent scalability and node reduction ratio.

The rest of the paper is organised as follows. In Section II, we describe the G-SCIS framework of graph summarization. In Section III, we present our Optimal algorithm. Next, in Section IV, we give our Scalable algorithm and formally characterize it in terms of guarantees it offers. Then, in Section V we present our experimental analysis. Finally, Section VI discusses related work and Section VII concludes the paper.

II. Preliminaries: G-SCIS Framework

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. A G-SCIS summary [17] is also an undirected graph and is denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{S_1, S_2, \ldots, S_k\}$ for $k \leq |V|$, $V = \bigcup_{i=1}^{k} S_i$ and $\forall i \neq j, S_i \cap S_j = \emptyset$. We refer to $\mathcal{V}$ as the set of supernodes and $\mathcal{E}$ as the set of superedges.

Reconstruction. Given a summary, we reconstruct (as a thought process) the original graph as follows. For each superedge $(S_i, S_j) \in \mathcal{E}$, we construct edges (u, v), for each $u \in S_i$ and $v \in S_j$. That is, for $i \neq j$, we build a complete bipartite graph with S_i and S_j and for $i = j$ (a self-loop superedge), we build a clique among the vertices of S_i. If the reconstructed graph ($\hat{G}$) is exactly the same as the original graph $\hat{G} = G$, the summary is called *lossless*. Otherwise, it is *lossy*.

Fully Dynamic Graphs They can be viewed as a sequence of modifications, where at each time step $t, t \geq 0$, a new edge $e_t = (u, v)$ is either added to or removed from the graph. We denote the graph at time t by $G_t = (V_t, E_t)$ and assume that G_0 is empty.

Problem Formulation. Our objective is to maintain a lossless summary $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ of a fully dynamic graph $G_t = (V_t, E_t)$ such that the number of supernodes in the summary is the minimum possible after each time step t. Formally, for all t, $t \geq 0$, we seek to

$$\text{minimize } |\mathcal{V}_t| \quad \text{subject to } \hat{G}_t = G_t. \tag{1}$$

Optimal Lossless Summary (OLS) in Static Graphs: Let $N(u)$ denote the set of neighbors of a vertex u and let $N^+(u) = N(u) \cup \{u\}$. A set of nodes I is an independent set in G if for all $u, v \in I$, $N(u) = N(v)$. A set of nodes C is a clique in G if for all $u, v \in C$, $N^+(u) = N^+(v)$. In other words, our definition of an independent set I must satisfy two conditions: (1) no two vertices in I are connected by an edge, and (2) every vertex in I must be connected to the same set of vertices outside I. Similar conditions hold for a clique C.

The following result is shown in [17].

Proposition II.1. *(1) In an OLS, each node is in a supernode of size one, or in a supernode that is a clique in G, or in a supernode that is an independent set in G. (2) Furthermore, a node v cannot be in a clique supernode C in one OLS and in an independent set supernode I in another.*

Using this proposition, Algorithm 1 below from [17] computes an OLS as follows. For each node u, it greedily finds the largest independent set or clique supernode that u can be a part of.

Algorithm 1 checks whether vertices u and v can be a part of a clique or an independent set (Line 6) supernode. If so, u and v are merged. Note that by Proposition II.1, the two

Algorithm 1 Finding the best summary

1: **Input:** $G = (V, E)$
2: **Initialization:** $Status[\forall v \in V] \leftarrow$ *False*, $\mathcal{S} \leftarrow []$
3: **for** $u \in V \wedge Status[u] =$ *False* **do**
4: $\quad S(u) \leftarrow \{u\}$, $Status[u] \leftarrow$ *True*
5: $\quad$ **for** $v \in V \wedge Status[v] =$ *False* **do**
6: $\quad\quad$ **if** $(N(u) = N(v)) \vee (N^+(u) = N^+(v))$ **then**
7: $\quad\quad\quad S(u) \leftarrow S(u) \cup \{v\}$, $Status[v] \leftarrow$ *True*
8: $\quad \mathcal{S}.add(S(u))$
9: BUILDSUPEREDGES($\mathcal{S}$)

conditions in line 6 are mutually exclusive. If such a set is not found, node u becomes a supernode of size one. After constructing the supernodes, the algorithm calls the function BUILDSUPEREDGES($\mathcal{S}$), which builds superedges as follows. An edge is added between supernodes S and S' iff $u \in S$ and $v \in S'$ and $(u, v) \in E$.

III. Optimal Lossless Algorithm

Algorithm 1 has $O(|V|^2 d_{max})$ time complexity, making it impractical for large datasets. An idea is to hash nodes based on their neighborhood set. Nodes that hash to the same bucket are candidates to be grouped together. However, hashing sets in a dynamic setting needs special care. This is because as new neighbors are gained or current neighbors are lost, we want to recompute the hash value of the set quickly in order to rapidly respond to each change.

In this section, we propose a hashing framework which is able to (1) update the hash value of a node u in *constant time* after some change in the u's neighborhood, (2) only compare u's neighborhood with the neighborhoods of a small set of nodes in order to determine any possible merge (3) guarantee the optimality of the summary graph in each step.

One of the important properties desirable for hashing nodes in dynamic setting is *sort insensitivity* which avoids sorting the set of neighbors of a node before applying hashing on the list. The hash function we propose is a sum of the k'th powers of neighbors of a node modulo P, where k is a small integer, e.g. 1, 2 or 3, and P is a large prime number. Using Equation 2, we can update the hash value of a node u using its previous value and the recent neighbor (v) it was connected to or disconnected from. In practice, we found that $k = 2$ gives fewer collisions.

$$\begin{aligned} I_k[u] &= I_k[u] + v^k \mod P && \langle u, v\rangle \text{ insertion} \\ C_k[u] &= C_k[u] + v^k \mod P && \langle u, v\rangle \text{ insertion} \\ I_k[u] &= I_k[u] - v^k \mod P && \langle u, v\rangle \text{ deletion} \\ C_k[u] &= C_k[u] - v^k \mod P && \langle u, v\rangle \text{ deletion} \end{aligned} \quad (2)$$

After updating the hash value of node u, we lookup for supernodes with the same hash value as u. We call these supernodes, candidate supernodes. Now, we need to evaluate each candidate supernode S whether it is a correct supernode for u or a false positive (i.e. hash collision). In order to deal with this, we perform a neighborhood equality check for u and a representative node (u') of S. If there is a match, i.e., $N(u) = N(u')$ when S is an independent set supernode, or $N^+(u) = N^+(u')$ when S is a clique supernode, we add u to S, otherwise we make u a singleton supernode.

Algorithm 2 shows the steps of our optimal algorithm. It uses two arrays I and C for storing hash values of each node. $I[u]$ is the independent set hash value of node u based on $N(u)$ and $C[u]$ is the clique hash value of node u based on $N^+(u)$. After an edge change $\langle u, v\rangle$, $I[u]$ and $C[u]$ are incrementally updated based on (2).

Supernodes are represented by two static arrays n and p where $n[u]$ points to the next node in the supernode of node u and $p[u]$ points to the previous node in the supernode of node u. If node u is in a singleton supernode then $n[u] = u, p[u] = u$.

We also use two hash data structures HI and HC, where each entry in HI and HC is keyed by an independent set or a clique hash value respectively and the value for each key is a set of single nodes with the same hash value but different neighborhoods. In other words, each pair of key value $\langle h, \{u_1, \ldots, u_k\}\rangle$ in HI has the following characteristics. For any pair of nodes $(u_i, u_j) \in \{u_1, \ldots, u_k\}, i \neq j$, $I[u_i] = I[u_j] = h$ and $N(u_i) \neq N(u_j)$, and for each pair of key value $\langle h', \{u'_1, \ldots, u'_k\}\rangle$ in HC, $C[u'_i] = C[u'_j] = h'$ and $i \neq j, N^+(u'_i) \neq N^+(u'_j)$. For each supernode there is just one node from that supernode in HI or HC (depending on its type) which we call it the representative of that supernode.

After each change ($\langle u, v\rangle$ insertion or deletion) the algorithm calls function *FindNodes* (Algorithm 3) to find all nodes whose supernodes need to be updated. *FindNodes* returns a set of 2, 3, or 4 nodes. Nodes u and v are always in this set. If $\_u$ is in the same supernode of size 2 as u, then $\_u$ is added in the returned set. Same logic is applied for $\_v$. The reason for this choice is that if u is removed from a supernode of size 2, then the other node in that supernode, $\_u$, may be able to join another supernode of size greater than one (recall Proposition II.1). So in order to keep the number of supernodes minimum, we also need to update the supernodes of $\_u$ and $\_v$. Let *nodes* be the set of nodes whose supernodes need to be updated. For each a in *nodes* (line 3 of Algorithm 2) we update the representation of its supernode S_a in HI and HC as described in Algorithm 4.

In line 6 of Algorithm 2, for each $a \in \{u, v\}$, we update $I[a]$ and $C[a]$ based on their current values and the recent change using Equation 2. Next, we lookup $I[a]$ in HI. If $I[a]$ exists in HI, we iterate over all nodes a' in value set $HI(I[a])$ and perform a neighborhood equality check for nodes a and a'. If there is a match of a and a', a is inserted to the supernode of a' by calling function *merge* and the process for node a is terminated (lines 7-11 of Algorithm 2). Otherwise, a similar process is performed for $C[a]$ and HC (lines 12-16). If neither of the lookups succeed, we call Algorithm 5 in line 17 to put node a into a singleton supernode and add both $(I[a], \{a\})$ and $(C[a], \{a\})$ to HI and HC respectively.

Algorithm 2 Optimal Algorithm (Optimal)

Input: $e = \langle u, v \rangle, +/-$
$nodes = findNodes(u, v)$
for $a \in nodes$ **do**
  updateH(a)
  if $a == u \vee a == v$ **then**
    $I[a], C[a] \leftarrow incremental(a, I[a], C[a])$ ▷ Eq. (2)
  if $(I[a] \in HI)$ **then**
    for $a' \in HI(I[a])$ **do**
      if $N(a') == N(a)$ **then**
        merge(a, a')
        continue
  if $C[a] \in HC$ **then**
    for $a' \in HC(C[a])$ **do**
      if $N^+(a') == N^+(a)$ **then**
        merge(a, a'))
        continue
  singleton$(I[a], C[a], a)$

Algorithm 3 findNodes(u, v)

$r \leftarrow \{u, v\}$
if S_u exists and $|S_u| == 2$ **then**
  Let $\_u$ be the other node in S_u
  Add $\_u$ to r
if S_v exists and $|S_v| == 2$ **then**
  Let $\_v$ be the other node in S_v
  Add $\_v$ to r
Return r

Algorithm 4 updateH(a)

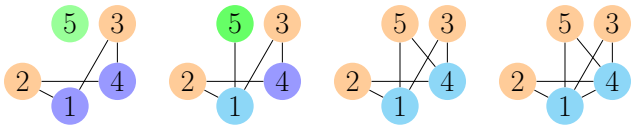
if a serves as a representative of S_a in HI **then**
  replace a by some other node $b \in S_a$ in HI.
if a serves as a representative S_a in HC **then**
  replace a by some other node $b \in S_a$ in HC.

Algorithm 5 singleton$(hI[u], hC[u], u)$

Add $I[u]$ to HI and u to $HI(I[u])$
Add $C[u]$ to HC and u to $HC(C[u])$
Create a singleton supernode ▷ $n[u] = u, p[u] = u$

Algorithm 6 merge$(u, _u)$

$n[u] \leftarrow \_u, p[u] \leftarrow p[\_u]$
$n[p[\_u]] \leftarrow u, p[\_u] \leftarrow u$



(a) initial graph (b) $\langle 1, 5 \rangle$ added (c) $\langle 4, 5 \rangle$ added (d) $\langle 1, 4 \rangle$ added

Figure 1. Evolution of supernodes as new edges come in. Nodes with the same color are in the same supernode.

step	edge	I	C	HI	HC
a		–	–	$\langle 5, \{1, 2\} \rangle$	$\langle 6, \{1\} \rangle$
b	$\langle 1, 5 \rangle$	$hI[1] = 10$ $hI[5] = 1$ $hI[4] = 5$	$hC[1] = 11$ $hC[5] = 6$ $hC[4] = 9$	$\langle 5, \{4, 2\} \rangle, \langle 10, \{1\} \rangle$ $\langle 5, \{4, 2\} \rangle, \langle 10, \{1\} \rangle, \langle 1, \{5\} \rangle$ $\langle 5, \{4, 2\} \rangle, \langle 10, \{1\} \rangle, \langle 1, \{5\} \rangle$	$\langle 11, \{1\} \rangle$ $\langle 11, \{1\} \rangle, \langle 6, \{5\} \rangle$ $\langle 11, \{1\} \rangle, \langle 6, \{5\} \rangle, \langle 9, \{4\} \rangle$
c	$\langle 4, 5 \rangle$	$hI[4] = 10$ $hI[5] = 5$	$hC[4] = 14$ $hC[5] = 10$	$\langle 5, \{2\} \rangle, \langle 10, \{1\} \rangle, \langle 1, \{5\} \rangle$ $\langle 5, \{2\} \rangle, \langle 10, \{1\} \rangle$	$\langle 11, \{1\} \rangle, \langle 6, \{5\} \rangle$ $\langle 11, \{1\} \rangle$
d	$\langle 1, 4 \rangle$	$hI[1] = 14$ $hI[4] = 11$	$hC[1] = 15$ $hC[4] = 15$	$\langle 5, \{2\} \rangle, \langle 10, \{4\} \rangle, \langle 14, \{1\} \rangle$ $\langle 5, \{2\} \rangle, \langle 14, \{1\} \rangle$	$\langle 15, \{1\} \rangle$ $\langle 15, \{1\} \rangle$

Table I
VALUES OF $I[u]$, $C[u]$, HI AND HC AFTER EACH CHANGE.

Figure 1 shows an example of a graph going through changes from step a to step d. Nodes with the same color are in the same supernode. Table I shows detailed values of our data structures. Equation 2 with $k = 1$ is used for incrementally updating hash values. In step (a), $HI(5)$ has two values, 1 and 2, because $I[1] = I[2] = 5$ and $N(1) \neq N(2)$. Other entries are computed similarly.

Correctness of Optimal. We show that after each step t, Algorithm 2 computes the lossless summary of G_t with minimum number of supernodes. We begin with an observation.

Proposition III.1. *The hash function in Eq. 2 is order independent.*

That is, two nodes with the same neighbours which arrived in a different order (e.g., $\langle u_1, u_2, u_3 \rangle$ and $\langle u_2, u_1, u_3 \rangle$), are hashed to the same bucket for any choice of k in Equation 2.

In Algorithm 2, each supernode has a representative in the data structures HI or HC depending on its type.

Proposition III.2. *An independent set (clique) supernode has one of its nodes u as its representative in HI (HC). That is, the entry $\langle I[u], u \rangle$ in HI ($\langle C[u], u \rangle$ in HC) is non-empty,.*

Recall that Algorithm 2 after updating a hash value for a node u does a neighborhood equality check to ensure that the neighborhood set of node u is the same as the neighborhood set of the representative node of that supernode under either independent set or clique requirements. Hence, we have

Proposition III.3. *At the end of step t, Algorithm 2 always computes a lossless summary of G_t.*

We now state and prove our main result.

Theorem III.4. *At the end of step t, Algorithm 2 computes the lossless summary of G_t with the minimum number of supernodes.*

Proof. We use induction on time t. Consider the case $t = 1$ when an edge $\langle u, v \rangle$ is added to an empty graph. Algorithm 2 places both nodes u and v inside a clique supernode because $C[u] = C[v] = u^k + v^k$ for any k. Hence, the number of supernodes is 1 and minimum. Let us suppose that the statement is true after $t - 1$ steps. Suppose that a new edge

$\langle u', v' \rangle$ arrives and the number of supernodes created by Algorithm 2 after processing this step t is more than the number of supernodes in the optimal summary. According to Proposition III.3, Algorithm 2 always generates a lossless summary. So, there must be one supernode s, at the end of step t, with size greater than one in optimal solution that is split into two or more supernodes in the summary constructed by Algorithm 2. We consider three possible cases: (1) Neither u' nor v' are in s; (2) Either node $u' \in s$ or $v' \in s$; (3) Both nodes u' and v' are in s.

Case 1: To show that this is not possible, we note that no change occurred to the neighborhood of nodes in s, $|s| \geq 2$. So, all the nodes in s must have been in the same supernode of the optimal solution before step t and hence in the same supernode of the summary produced by Algorithm 2 before step t. Therefore, these nodes would stay together in the same supernode of the summary at the end of step t also and will not be split by the algorithm.

Case 2: Now let us assume either node u' or v' is in s. There are two possible scenarios: 1) supernode s existed before $\langle u', v' \rangle$ was inserted or deleted in step t and one of u' or v' joined s after the change or 2) s was created in step t. In the former case, the only possibility is that the node u' and the rest of the nodes of s are in different supernodes of the summary constructed by the algorithm (see the discussion in Case 1). However, based on Proposition III.2, supernode s has one representative node u in HI or HC (in both if it is of size 1) which enables Algorithm 2 to add u' to supernode s. In the latter case, the size of s cannot be greater than one in the optimal solution because if it was greater than one the supernode s existed before the change. On the other hand, if it is of size one, it cannot be split further in the summary output by Algorithm 2.

Case 3: This is similar to case 2. Nodes u' and v' are in the same supernode s in the optimal solution. If s existed before step t, it has one representative node in HI or HC and Algorithm 2 will add both u' and v' to s as they will hash to the same value by Proposition III.1. If s is a new supernode created in step t, then the nodes u' and v' will be added to that supernode in summary constructed by Algorithm 2 and hence will be in the same supernode.

Therefore, in all cases the summary output by Algorithm 2 is the same as the optimal summary. □

Complexity analysis. Algorithms 3, 4, 5, and 6 take constant time to find nodes whose supernodes need to be updated, to update HI and HC, to place a node into a singleton supernode, and to add a node to a supernode.

The quality of the hash function affects the number of neighborhood equality checks between node u and other nodes in $HI(I[u])$ or $HC(C[u])$ in Algorithm 2 and consequently affects the total update time. If the hash function is perfect, each entry of HI and HC has at most one value and thus each step takes either constant time if node u ends up with a singleton supernode or $O(|N(u)|)$ time if u ends up with a supernode of size greater than one (it needs to perform one neighborhood equality check between node u and the representative node of supernode of node u).

IV. Scalable lossless algorithm

Intuitively, as the degree of a node increases, the node becomes less likely to find a set of nodes in the graph that share the same neighborhood under either clique or independent set conditions. In other words, it is more likely that higher degree nodes reside in singleton supernodes and lower degree nodes reside in crowded supernodes. Therefore, if we summarize the nodes with low degree (less than a threshold K) we are able to 1) achieve high reduction in nodes that remains close to the optimal point, 2) generate a *perfect hashing scheme* to eliminate the neighborhood equality check which was a computational bottleneck in Algorithm 2.

If $\mathcal{V}_K^t$ represents the set of supernodes in $\mathcal{G}_t$ containing nodes of degree less or equal to K, our objective in this section is as follows. For all t, $t \geq 0$,

$$\text{minimize } |\mathcal{V}_K^t| \quad \text{subject to } \hat{G}_t = G_t. \tag{3}$$

We propose a new hashing scheme which 1) updates K different hash values of a node u in *constant time* 2) guarantees a *perfect hashing* which is significant as it enables us to avoid the expensive process of the neighborhood equality check while maintaining losslessness, 3) guarantees the optimality of summary for all nodes with degrees less or equal to K.

We assume each independent set supernode is represented by a tuple of size $K+1$ ($\langle I_1, I_2, \ldots, I_K, d \rangle$) and each clique supernode is represented by a tuple of size $K+2$ ($\langle C_1, C_2, \ldots, C_k, , C_{k+1}, d \rangle$) where the last element d represents the degree of nodes in graph and others represent the hash values of using different (K for independent set and $K+1$ for clique) hash functions. For example, $I_1(u)$ is a hash function of linear sum of neighbors of node u, $I_2(u)$ is sum of squares of neighbors of node u and etc. $C_1(u)$ is similar to $I_1(u)$ but it also includes node u in the computation. The hashing scheme we propose is shown by Equation 2. $I_k(u)$ for $1 \leq k \leq K$ and $C_k(u)$ for $1 \leq k \leq K+1$ are updated based on their previous values and the recent change to the neighborhood of node u (i.e $\langle u, v \rangle$ insertion or $\langle u, v \rangle$ deletion). The last element d capturing the degree of node u is incremented or decremented depending on the edge insertion or deletion. Each $I_k(u)$ is initialized by 0 and each $C_k(u)$ is initialized by $u^k \mod P$.

We note that K is a user specified threshold that allows the algorithm to summarize only the nodes with degrees less or equal to K, while others with degree greater than K are placed into singleton supernodes. When there is a change in the neighborhood of node u and its total degree so far is less and equal than K, the algorithm incrementally updates $1 \leq k \leq K, I_k(u)$ and $1 \leq k \leq K+1, C_k(u)$ based on Equation 2. It then searches updated tuple of node u in supernodes set. If there exists a supernode with the same tuple, node u is added to that supernode without requiring the neighborhood equality check, otherwise node u will be in a singleton supernode as there is no match for the tuple of node u.

Algorithm 7 shows the main steps. As in Algorithm 2, supernodes are represented by two static arrays n and p. It uses two data structures HI and HC which each entry in HI is keyed by a tuple of size $K+1$ and each entry in HC is keyed by a tuple of size $K+2$. As the hashing scheme used in Algorithm 7 is perfect (proven by Theorem IV.1), each entry in HI or HC has exactly one value.

For each change $\langle u, v\rangle$, as in Algorithm 2, it first calls *findNodes* function (Algorithm 3) to find all nodes whose supernodes may need to be updated. For each node $a \in nodes$ it checks the degree of node a. If it is greater than K then node a will be in a singleton supernode and the process is terminated for node a (line 5-7 of Algorithm 7). Otherwise, it calls *updateH* function (Algorithm 4) to update HI and HC. Algorithm 4 checks whether node a is one of the representatives in HI or HC (i.e. either $I(a)$ or $C(a)$ is non-empty and node a is the value). If so, it is replaced by another node in that supernode. Then Algorithm 8 is called to update the tuple of hash values for node a. *Incremental* function updates $I(u)$ and $C(u)$ based on the Equation 2. The last element in $I(u)$ or $C(u)$ is incremented or decremented depending on insertion or deletion.

Line 10 of Algorithm 7 searches $I(a)$ through HI, if there is an entry in HI ($HI(I(a)$ has a value a') it adds a to the supernode of a' by calling Algorithm 6 and terminates the process for node a, otherwise it goes ahead and searches $C(a)$ through HC to see if there is a non-empty entry of $HC(C(a))$, if there is, a is added to the supernode of a' and the process is terminated. Finally, if the Algorithm 7 found no match in HI or HC, it adds $I(a)$ to HI and node a to $HI(I(a))$ and $C(a)$ to HC and adds node a to $HC(C(a))$ and places node a into a singleton supernode ($n[a] = a, p[a] = a$).

Correctness of scalable algorithm. Suppose that for two nodes u and v, $d(u) = d(v) = d$ and $N(u) \neq N(v)$. The system of Equations 4 describes the false positive error of the proposed hashing scheme while checking if $N(u) = N(v)$ or not. We will show that if $K \geq d$, (4) cannot be satisfied.

$$\begin{aligned}
\sum_{u_i \in N(u)} u_i \mod P &= \sum_{v_i \in N(v)} v_i \mod P \\
\sum_{u_i \in N(u)} u_i^2 \mod P &= \sum_{v_i \in N(v)} v_i^2 \mod P \\
&\vdots \\
\sum_{u_i \in N(u)} u_i^K \mod P &= \sum_{v_i \in N(v)} v_i^K \mod P
\end{aligned} \tag{4}$$

Theorem IV.1. *Suppose that u and v are two vertices in G such that $d(u) = d(v) = d$ and let $K \geq d$. Then the system of equations (4) holds if and only if $N(u) = N(v)$.*

Proof. One direction is clear: if $N(u) = N(v)$, all the equalities in the system of equations 4 hold. We now show the other direction using two key lemmas. Let $N(u) = \{u_1, \ldots, u_d\}$ and $N(v) = \{v_1, \ldots, v_d\}$. We begin with a definition.

Algorithm 7 Scalable Algorithm (Scalable)

1: **Input:** $e = \langle u, v\rangle, +/-, K$
2: $nodes = findNodes(u, v)$
3: **for** $a \in nodes$ **do**
4: **if** $a == u \vee a == v$ **then**
5: **if** $I(a)(K+1) \geq K$ **then** ▷ Degree greater than K
6: Create a singleton supernode ▷ $n[a] = a, p[a] = a$
7: **continue**
8: updateH(a)
9: $I(a), C(a) \leftarrow Incremental(I(a), C(a), u, v)$ ▷ Eq. 2
10: **if** $I(a) \in HI$ **then**
11: $a' \leftarrow HI(I(a))$
12: **merge**(a, a')
13: **continue**
14: **if** $C(a) \in HC$ **then**
15: $a' \leftarrow HC(C(a))$
16: **merge**(a, a')
17: **continue**
18: Add $I(a)$ to HI and a to $HI(I(u))$
19: Add $C(u($ to HC and u to $HC(C(u))$
20: Create a singleton supernode ▷ $n[a] = a, p[a] = a$

Algorithm 8 Incremental

1: **Input:** $I(u), C(u), u, v,$
2: Update degree, $I_{K+1}(u), C_{K+2}(u)$
3: Update $I_k(u)$ and $C_k(u)$, based on Eq 2
4: return $C(u), I(u)$

The elementary symmetric polynomials on d variables are defined as follows: $e_1(X_1, X_2, \ldots, X_d) = \sum_{1 \leq j \leq d} X_j$, $e_2(X_1, X_2, \ldots, X_d)$ $= \prod_{1 \leq i \leq j \leq d} X_i X_j$, $\ldots$, $e_d(X_1, X_2, \ldots, X_d) = X_1 X_2 \ldots X_d$.

The following lemma states that if Eqn. (4) holds, then the d elementary symmetric polynomials are all equal modulo P.

Lemma IV.2. *Suppose that $\sum_{u_i \in N(u)} u_i^j = \sum_{v_i \in N(v)} v_i^j \mod P$ for all $j = \{1, 2, \ldots, K\}$. Then $e_k(u_1, u_2, \ldots, u_d) = e_k(v_1, v_2, \ldots, v_d) \mod P$, for all $k = \{1, 2, \ldots, d\}$.*

Proof. We prove the lemma by induction on k. Clearly, the claim is true for $k = 1$ as $e_1(u_1, u_2, \ldots, u_d) = e_1(v_1, v_2, \ldots, v_d) \mod P$ is an equation in (4). Let us assume that the induction hypothesis holds for all $k \leq t$ and prove it for $k = t + 1$. For this, we make use of Newton's identity.

Proposition IV.3. *For all $d \geq 1$ and $d \geq k \geq 1$,*

$$ke_k(x_1, .., x_d) = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i}(x_1, .., x_d) P_i(x_1, .., x_d) \mod P$$

where $P_i(x_1, \ldots, x_d) = \sum_{j=1}^{d} x_j^i$ and $e_0(x_1, \ldots, x_d) = 1$.

Using the facts, $\sum u_i \mod P = \sum v_i \mod P, \sum u_i^2 \mod P = \sum v_i^2 \mod P, \ldots, \sum u_i^{t+1} \mod P = \sum v_i^{t+1} \mod P$, and $e_k(u_1, u_2, \ldots, u_d) \mod P = e_k(v_1, v_2, \ldots, v_d) \mod P$, for all $k \in \{1, 2, \ldots, t\}$. and combining them with Newton's identity, we get $e_{t+1}(u_1, u_2, \ldots, u_d) \mod P = e_{t+1}(v_1, v_2, \ldots, v_d) \mod P$. □

The following lemma shows that if the d elementary symmetric polynomials are equal modulo P for two vertices of degree d, their neighbourhoods are the same.

Lemma IV.4. *Suppose* $d(u) = d(v) = d$ *and* $e_k(u_1, u_2, \ldots, u_d) = e_k(v_1, v_2, \ldots, v_d) \mod P$, *for all* $k \in [1, d]$. *Then* $N(u) = N(v)$.

Proof. To show this, we use the following equation:

$$\prod_{i=1}^{d}(x+u_i) = x^d + e_1(u_1, \ldots, u_d)x^{d-1} + \cdots + e_d(u_1, \ldots, u_d)$$

From the above expression and the assumption that $e_k(u_1, u_2, \ldots, u_d) = e_k(v_1, v_2, \ldots, v_d) \mod P$, for all $k = \{1, 2, \ldots, d\}$, we can conclude that $\prod_{i=1}^{d}(x+u_i) = \prod_{i=1}^{d}(x+v_i) \mod P$. Using this identity and the well known fact that if a prime P divides $a_1 a_2 \ldots a_n$, then it must divide a_i for some i, we can show that $\{u_1, u_2, \ldots, u_d\} = \{v_1, v_2, \ldots, v_d\}$. To prove this, substitute $x = -u_1$ in the identity above. As the LHS of the equation is 0, it implies that $u_1 = v_i$ for some i using the fact above. Repeating this reasoning helps us conclude that $N(u) = N(v)$. □

The two lemmas together prove Theorem IV.1. □

A similar result holds for clique supernodes with $N^+(u)$ instead of $N(u)$ and $K \geq d+1$ equations instead of $K \geq d$. Thus, we obtain

Corollary 1. *At the end of step t, Algorithm 7 computes a lossless summary that is* optimal *for nodes of degree at most* K.

Complexity analysis. As in Algorithm 2, *findNodes, updateH* take constant time and we need $O(K)$ time to update hash values of nodes where K is user specified threshold (between 10-50). Updating supernodes takes constant time as we do not need to perform neighborhood equality check. Hence, the time for each update is $O(K)$ if the degree of node u is less than K; otherwise it takes constant time to place u in a singleton supernode.

A. Summarizing Directed Graphs

In this section, we extend the proposed scalable algorithm to summarize dynamic directed graphs. We begin with the definition of independent sets and cliques in this setting.

Definition IV.1. (1) A set of nodes I is an independent set in a directed graph $\overrightarrow{G}$ if and only if their in-neighbor and out-neighbor sets are the same. That is, for all $u, v \in I$, $N_{in}(u) = N_{in}(v)$ & $N_{out}(u) = N_{out}(v)$ (2) A set of nodes C is a clique in a directed graph $\overrightarrow{G}$ if and only if: $N_{in}(u) \cup \{u\} = N_{in}(v) \cup \{v\}$ & $N_{out}(u) \cup \{u\} = N_{out}(v) \cup \{v\}$

To adapt the scalable approach for directed graphs, we use the threshold K to bound the in-degree and the out-degree of the vertices to be processed. In particular, we use $K/2$ as an upper bound for the in-degree and the out-degree. Therefore, each I supernode a tuple $\langle I_1, \ldots, I_{K/2}, d_{in}, I_{K/2+1}, \ldots, I_K, d_{out}\rangle$ of size $K+2$ and each C supernode is a tuple $\langle C_1, \ldots, C_{K/2}, C_{K/2+1}, d_{in}, C_{K/2+2}, \ldots, C_{K+3}, d_{out}\rangle$ of size $K+4$ where K is an even number. The first $K/2$ (or $K/2+1$ in C) elements are hash values of in-neighbors of node u for I (for C), followed by the in-degree of node u; the next $K/2$ (or $K/2+1$ in C) elements are hash values of out-neighbors of node u, followed by the out-degree of node u. It also uses Equation 2 for updating hash values. Algorithm 9 shows the main steps of directed dynamic graph summarization. It receives an edge change $\langle \overrightarrow{u, v}\rangle$ and the number of hash functions K. It needs to update the hash values of the out-neighbors of node u and hash values of the in-neighbors of node v. If the degree of in-neighbors or out-neighbors of nodes u or v exceeds the threshold $K/2$, it puts that node into a singleton supernode. The other steps are similar to scalable algorithm; after updating hash values it searches through HI or HC to see is there a match or not. Finally if there is not a match in HI or HC, it puts that node into a singleton supernode and adds both I and C to HI and HC.

Algorithm 9 Directed-Scalable Algorithm

```
Input: ⟨u→, v⟩, +/−, K
▷ Updating out neighbors hashes for u and in-neighbors for v
for a ∈ {u, v} do
    if I(a)(K+2) ∨ I(a)(K/2+1) ≥ K/2 then
        Create a singleton supernode ▷ n[a] = a, p[a] = a
        continue
    updateH(a)
    if a == u then
        Update I(a)(K+2) and C(a)(K+4)
    if a == v then
        Update I(a)(K/2+1) and C(a)(K/2+2)
    I(a), C(a) ← Incremental(I(a), C(a), u, v) ▷ Eq. 2
    if I(a) ∈ HI then
        a' ← HI(I(a))
        merge(a, a')
        continue
    if C(a) ∈ HC then
        a' ← HC(C(a))
        merge(a, a')
        continue
    Add I(a) to HI and a to HI(I(u))
    Add C(u) to HC and u to HC(C(u))
    Create a singleton supernode      ▷ n[a] = a, p[a] = a
```

Graph	Abbr	Nodes	Edges
cnr-2000	CN	325,557	5,565,380
Eu-2005	EU	862644	19,235,140
hollywood-2009	H1	1,139,905	113,891,327
hollywood-2011	H2	2,180,759	228,985,632
indochina-2004	IC	7,414,866	150,984,819
uk-2002	U1	18,520,486	261,787,258
arabic-2005	AR	22,744,080	553,903,073
uk-2005	U2	39,459,925	1,581,073,454

Table II
SUMMARY OF DATASETS

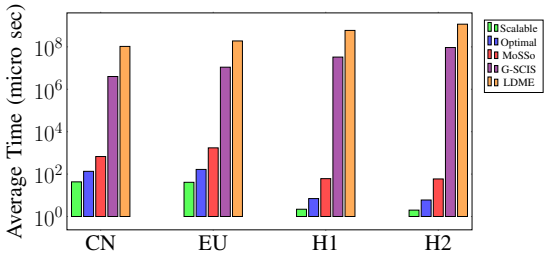


Figure 2. Average processing time per edge for Optimal and Scalable vs. MoSSo, G-SCIS, LDME. Scalable is up to 8 and 7 orders of magnitude faster than LDME and G-SCIS, resp., and around 30 times faster than MoSSo.

V. EXPERIMENTS

Implementation: We implemented the proposed algorithms in Java 14 on a machine with dual 6 core 2.10 GHz Intel Xeon CPUs, 64 GB RAM and running Ubuntu 18.04.2 LTS (https://anonymous.4open.science/r/GraphSumDynamic-359C/README.md). Other state-of-the-art algorithms were also implemented in Java and they are publicly available. All the dynamic algorithms are evaluated in a fully dynamic scenario; each time an edge is added to or removed from graph, the summary is updated dynamically.

Datasets: We choose different graphs varying from moderate to large (see Table II for details). All graphs have been symmetrized. We also perfomed experiments with the unsymmetrized versions of several graphs (directed case). Due to space constraints we mostly show results for symmetrized graphs (undirected case).

Baseline algorithms: We use state-of-the-art lossless algorithms including batch [17], [18] and dynamic [2]. Although the objective function of [2] is different, it is the only state-of-the-art dynamic lossless algorithm that scales to large graphs. Source codes of all state-of-the-art lossless algorithms are publicly available and they were all implemented in Java. G-SCIS [17] does not require any input parameters. For LDME [18] we use k=20 as the size of DOPH signature. There are 4 different versions of MoSSo (MoSSo-Greedy, MoSSo-MCMC, MoSSo-Simple and MoSSo-MoSSo) in [2]. We use MoSSo-MoSSo (MoSSo) as it is orders of magnitude faster than the other variants and we use the same configuration as in [2], $e = 0.3$ and $c = 120$, where e is the escape probability and c is the sample size of each trial (see [2]). SWeG [9] is another state-of-the-art algorithm but we decided to not include it in our experiments because first the source code is not publicly available and also because it was improved by LDME [18], which we use in our experiments.

Evaluation: Reduction in nodes (RN) [17], [11] is a metric used to evaluate the degree of summarization for each algorithm. It is defined as $RN = (|V| - |\mathcal{V}|)/|V|$. Regarding MoSSo, since it produces also correction graphs C^+, C^-, we need to consider them in order to reconstruct the original graph. Thus, RN for MoSSo is more precisely computed as $RN = (|V| - (|\mathcal{V} \cup V(C^+) \cup V(C^-)|))/|V|$.

Average processing time per change: We compare the average processing time per edge change of different algorithms. The averages are obtained by inserting or deleting 100 random edges each time and performing 10 experiments. Since batch algorithms are not able to incrementally update the summary, they need to obtain the summary from scratch for each change to the graph. Figure 2 shows the results of our comparison in microseconds. We can see for instance that Scalable is up to 8 and 7 orders of magnitude faster than LDME and G-SCIS, respectively, and also around 30 times faster than MoSSo. It is interesting to note that, if 1 million changes occur in a graph (take $H2$ for example), Scalable is 2 orders of magnitude faster than running just once a batch algorithm at the end of the 1 million-edge sequence. Also as the size of graph increases, both Optimal and Scalable have a higher chance for grouping nodes and consequently their average processing time for each change decreases. MoSSo also exhibits a similar behavior as the graph sizes become larger. We observed in our experiments that the average processing time per edge was constant for each dataset, namely that the running time as the graph gets more insertions grows linearly. However, this constant is different for different graphs, not related to their size.

Accumulative running time and reduction in nodes: We compute the accumulative running time and reduction in nodes of MoSSo and proposed methods after $|E|$ steps of edge insertion. The results are shown in Figures 3 and 5. Optimal is faster than MoSSo and provides far better reduction in nodes. Scalable is up to 40x faster than MoSSo and also up to 10x faster than the optimal algorithm (see for instance CN and U2). We also provide a more refined analysis of accumulated time in Figure 4. We see that all three dynamic algorithms, Optimal, Scalable, and MoSSo, scale linearly with $|E|$, with Optimal and Scalable being significantly better than MoSSo. In terms of RN, Figure 5, we can see that both Optimal and Scalable outperform MoSSo.

Sensitivity analysis to input parameters: Next we show the performance of Scalable with respect to the number of hash functions. Figure 6 shows these results and we can see that Scalable behaves similarly for different datasets as k varies. Running time does not increase much and the RN value only changes slightly after some point, k value of 20 or 30, so that is a sweet spot for k.

Performance on Directed Graphs: Since other state-of-the-art lossless algorithms are not able to summarize directed graphs, we can only show the performance of our Directed-Scalable algorithm in a fully dynamic scenario in Figure 7. Figure 7, for instance, shows that Directed-Scalable is able to

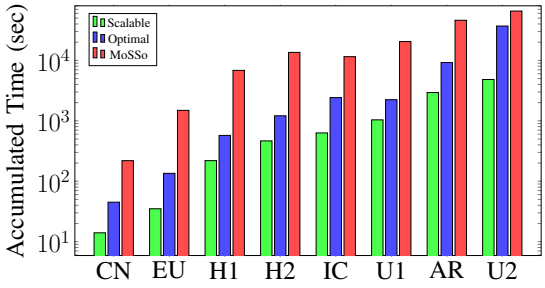
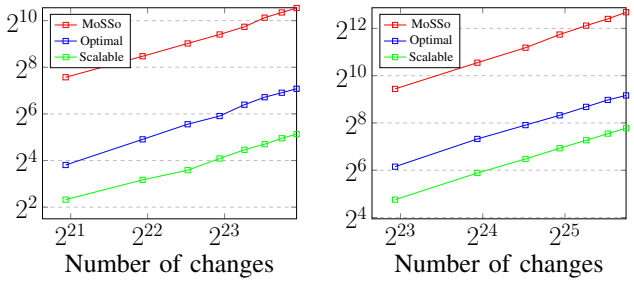


Figure 3. Accumulated running time for Scalable, Optimal, and MoSSo after E changes to each graph. Scalable is up to 40x faster than MoSSo and also up to 10x faster than the optimal algorithm (see for instance U2 and CN).



(a) EU (b) H1

Figure 4. Accumulated running time in seconds (vertical axis) for Scalable vs Optimal vs MoSSo with respect to number of changes (horizontal axis). (a) Time measured for EU every 2M changes, (b) Time measured for H1 every 8M changes. As seen, the scalability of all three algorithms is quite linear in $|E|$, with Optimal and Scalable being significantly better than MoSSo.

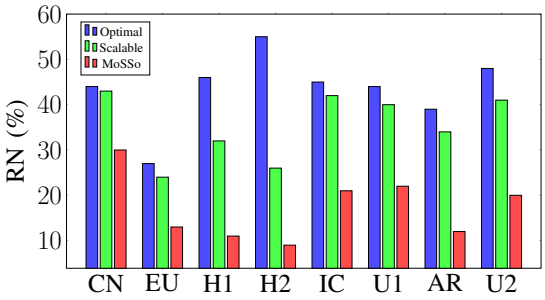
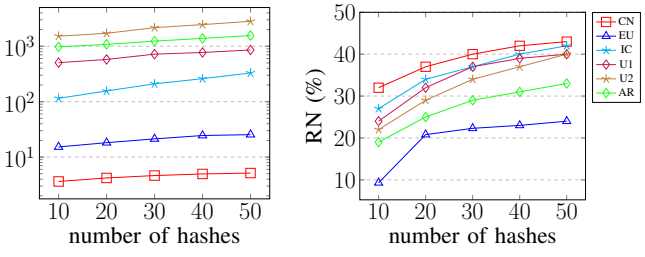
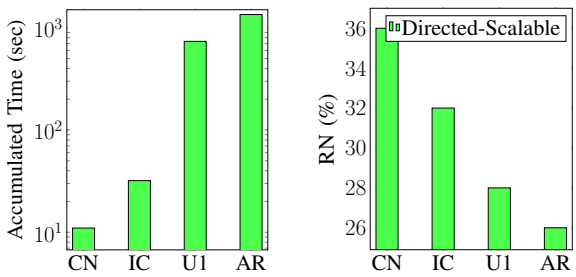


Figure 5. Reduction in nodes for Optimal, Scalable, and MoSSo. Both Optimal and Scalable outperform MoSSo. For CN, EU, IC, U1, AR, U2 Scalable is quite close to Optimal.



(a) running time (b) RN

Figure 6. Effects of number of hashes on the performance of Scalable. Running time in (a) is in seconds.



(a) running time (b) RN

Figure 7. Performance of Directed-Scalable on different directed graphs in a fully dynamic scenario.

achieve a 35% reduction in nodes for CN and in most cases the RN is more than 25%. The figure also shows a running time performance similar to that of Scalable on undirected graphs.

VI. Related Work

Graph summarization has been studied extensively in the recent years (see [5], [4] for an overview). The algorithms are generally classified as grouping and non-grouping methods and our work belongs to the former category. The grouping based methods can be either lossless or lossy and can address static or dynamic graphs. Table III categorizes the existing grouping-based approaches on static/dynamic graphs.

In the first row, we show the works that propose correction set-based approaches. In this framework [18], [9], [10], [19], two sets of side information $(C+, C-)$ are stored along with the summary graph and are used for correction during reconstruction. The framework was introduced by [10], then refined in [19] and made scalable by [9]. The latter addresses both the lossless and lossy cases for static graphs. Work [18] improves on the approach of [9] in terms of scalability and [20] offers interpretability of the supernodes in the summary. Ko et al. in [2] proposed the first dynamic lossless algorithm, MoSSo, in this framework. MoSSo is able to update the summary in near-constant time. [21] proposed a parameter-free incremental lossless algorithm in the correction set framework, based on exploring the subgraph influenced by the insertion of an edge. However, achieving parameter-freeness comes at the cost of being up to an order of magnitude slower than MoSSo.

In the second row, we show the works that propose utility-based approaches. The pioneering work in this category is [11] where the goal is minimize the number of supernodes while ensuring the utility of the summary does not drop below a threshold. This work produces lossy summaries for static graphs. Work [17] proposes a lossless summarization approach for static graphs as well as scales up the lossy summarization, again for static graphs. Our proposed approach in this present paper is the first to address dynamic lossless summarization in the utility based framework.

In the third row, we show the works that produce lossy summaries which contain superedges weighted by a fraction representing the number of edges between vertices in two supernodes over the maximum number of possible edges

Name \ Style	Lossless(Static)	Lossy(Static)	Lossless(Dynamic)	Lossy(Dynamic)
Correction-set	[9], [10], [18], [19], [20]	[9], [10]	[2], [21]	
Utility-Based	[17]	[17], [11]	Proposed	
Weighted Adjacency-Based		[12], [13], [22]		[23], [1]
Decomposition-Based	[24], [25]	[26], [25], [27]	[16]	[15], [28], [29], [30], [14], [31]

Table III
AN OVERVIEW OF THE GROUPING-BASED ALGORITHMS ON PLAIN (STATIC/DYNAMIC) GRAPHS

between them. Given such a summary, the goal is the minimize the error between the original and the reconstructed graph, referred to as the reconstruction error. All these works address the lossy case. [12], [13], [22] deal with the static case, whereas [23], [1] the dynamic case.

Finally, works listed in the fourth row fall under the decomposition-based approach which aims at producing summaries tailored to answering certain types of graph queries [24], [26] or compressing the graph based on certain types of graphlets such as triangles, higher order cliques and quasi-cliques, chains. [25], [27]. All these works address static graphs. Dynamic graph summarization has also received considerable attention in this category. [16] addresses the lossless case using a minimum description length (MDL) technique to minimize the number of required bits for describing the graph. However, the algorithm of [16] does not scale to large graphs. Works [15], [28], [29], [30], [14], [31] address the lossy case. All these algorithms are sketching-based approaches using approximation techniques not applicable to the lossless case.

VII. CONCLUSIONS

In this work, we focused on lossless summarization of dynamic graphs where the objective is to minimize the number of supernodes in the summary after each change. We presented two lossless summarization algorithms, Optimal and Scalable, for summarizing fully dynamic graphs. Our results are in the G-SCIS framework [17] which produces summaries that can used as-is in several graph analytics tasks while also achieving strong compression. While G-SCIS is a batch algorithm, our algorithms are fully dynamic and can respond rapidly to each change in the graph. Not only are our algorithms able to outperform G-SCIS and other batch algorithms by several orders of magnitude, they also significantly outperform MoSSo, the state-of-the-art in lossless dynamic graph summarization. While our first algorithm, Optimal, produces always the most optimal summary, our second algorithm, Scalable is able to trade the amount of node reduction for extra scalability. For reasonable values of the parameter K, Scalable is able to outperform Optimal by an order of magnitude in speed, while keeping the rate of node reduction close to that of Optimal.

REFERENCES

[1] I. Tsalouchidou, F. Bonchi, G. D. F. Morales, and R. Baeza-Yates, "Scalable dynamic graph summarization," *IEEE TKDE*, vol. 32, no. 2, pp. 360–373, 2018.

[2] J. Ko, Y. Kook, and K. Shin, "Incremental lossless graph summarization," in *SIGKDD*, 2020, pp. 317–327.

[3] K. Hanauer, M. Henzinger, and C. Schulz, "Recent advances in fully dynamic graph algorithms," *arXiv preprint arXiv:2102.11169*, 2021.

[4] A. Khan, S. S. Bhowmick, and F. Bonchi, "Summarizing static and dynamic big graphs," *PVLDB*, vol. 10, no. 12, pp. 1981–1984, 2017.

[5] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *CSUR*, vol. 51(3), pp. 1–34, 2018.

[6] A. Bonifati, S. Dumbrava, and H. Kodylakis, "Graph summarization," *arXiv preprint arXiv:2004.14794*, 2020.

[7] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *WWW*, 2004, pp. 595–602.

[8] A. Maccioni and D. J. Abadi, "Scalable pattern matching over compressed graphs via dedensification," in *SIGKDD*, 2016, pp. 1755–1764.

[9] K. Shin, A. Ghoting, M. Kim, and H. Raghavan, "Sweg: Lossless and lossy summarization of web-scale graphs," in *WWW*, 2019, pp. 1679–1690.

[10] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *SIGMOD*, 2008, pp. 419–432.

[11] K. A. Kumar and P. Efstathopoulos, "Utility-driven graph summarization," *PVLDB*, pp. 335–347, 2018.

[12] K. Lee, H. Jo, J. Ko, S. Lim, and K. Shin, "Ssumm: Sparse summarization of massive graphs," in *SIGKDD*, 2020, pp. 144–154.

[13] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," *DMKD*, pp. 314–349, 2017.

[14] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: On query estimation in graph streams," *PVLDB*, vol. 5, no. 3, 2011.

[15] X. Gou, L. Zou, C. Zhao, and T. Yang, "Fast and accurate graph stream summarization," in *ICDE*, 2019, pp. 1118–1129.

[16] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos, "Timecrunch: Interpretable dynamic graph summarization," in *SIGKDD*, 2015, pp. 1055–1064.

[17] M. Hajiabadi, J. Singh, V. Srinivasan, and A. Thomo, "Graph summarization with controlled utility loss," in *SIGKDD*, 2021, pp. 536–546.

[18] Q. Yong, M. Hajiabadi, V. Srinivasan, and A. Thomo, "Efficient graph summarization using weighted lsh at billion-scale," in *SIGMOD*, 2021, pp. 2357–2365.

[19] K. U. Khan, W. Nawaz, and Y.-K. Lee, "Set-based approximate approach for lossless graph summarization," *Computing*, pp. 1185–1207, 2015.

[20] K. Lee, J. Ko, and K. Shin, "Slugger: Lossless hierarchical summarization of massive graphs," *arXiv preprint arXiv:2112.05374*, 2021.

[21] Z. Ma, J. Yang, K. Li, Y. Liu, X. Zhou, and Y. Hu, "A parameter-free approach for lossless streaming graph summarization," in *DASFAA*. Springer, 2021, pp. 385–393.

[22] M. Purohit, B. A. Prakash, C. Kang, Y. Zhang, and V. Subrahmanian, "Fast influence-based coarsening for large networks," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1296–1305.

[23] S. Fernandes, H. Fanaee-T, and J. Gama, "Dynamic graph summarization: a tensor decomposition approach," *DMKD*, pp. 1397–1420, 2018.

[24] W. Fan, Y. Li, M. Liu, and C. Lu, "Making graphs compact by lossless contraction," *The VLDB Journal*, pp. 1–25, 2022.

[25] L. Wang, Y. Lu, B. Jiang, K. T. Gao, and T. H. Zhou, "Dense subgraphs summarization: An efficient way to summarize large scale graphs by super nodes," in *International Conference on Intelligent Computing*. Springer, 2020, pp. 520–530.

[26] S. Kang, K. Lee, and K. Shin, "Personalized graph summarization: formulation, scalable algorithms, and applications," *arXiv preprint arXiv:2203.14755*, 2022.

[27] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, "Vog: Summarizing and understanding large graphs," in *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 2014, pp. 91–99.

[28] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: sparsification, spanners, and subgraphs," in *PODS*, 2012, pp. 5–14.

[29] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *SIGMOD*, 2016, pp. 1481–1496.

[30] A. Khan and C. Aggarwal, "Toward query-friendly compression of rapid graph streams," *SNAM*, pp. 1–19, 2017.

[31] T. Blume, D. Richerby, and A. Scherp, "Incremental and parallel computation of structural graph summaries for evolving graphs," in *CIKM*, 2020, pp. 75–84.