Graph Summarization with Controlled Utility Loss

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

Jasbir Singh
University of Victoria
Victoria, Canada
jasbirsingh@uvic.ca

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

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

ABSTRACT

We present new algorithms for graph summarization where the loss in utility is fully controllable by the user. Specifically, we make three key contributions. First, we present a utility-driven graph summarization method G-SCIS, based on a clique and independent set decomposition, that produces optimal compression with zero loss of utility. The compression provided is significantly better than state-of-the-art in lossless graph summarization, while the runtime is two orders of magnitude lower. Second, we propose a highly scalable, utility-driven algorithm, T-BUDS, for fully controlled lossy summarization. It achieves high scalability by combining memory reduction using Maximum Spanning Tree with a novel binary search procedure. T-BUDS outperforms state-of-the-art drastically in terms of the quality of summarization and is about two orders of magnitude better in terms of speed. In contrast to the competition, we are able to handle web-scale graphs in a single machine without performance impediment as the utility threshold (and size of summary) decreases. Third, we show that our graph summaries can be used as-is to answer several important classes of queries, such as triangle enumeration, Pagerank and shortest paths.

KEYWORDS

social networks, graph summarization, utility loss

1 INTRODUCTION

Graphs are ubiquitous and are the most natural representation for many real-world data such as web graphs, social networks, communication networks, transaction networks, and epidemiological networks. Such graphs are growing at an unprecedented rate. For instance, the web graph consists of more than a trillion websites [2] and the social graphs of Facebook, Twitter, and Weibo, have billions of users with many friend/follow connections per user [1, 3, 4]. Consequently, storing such graphs and answering queries, mining patterns, and visualizing them are becoming highly impractical [11, 22]. Graph summarization is a fundamental task of finding a compact representation of the original graph called the summary. It allows us to decrease the footprint of the graph and derive analytical insights more efficiently [17, 22].

The problem has been approached from different directions, such as compression to reduce the number of required bits for describing graphs [5, 21], sparsification to remove less important nodes/edges in order to make the graph simpler [16, 25] and grouping to merge nodes into supernodes based on some interestingness measure [9, 11, 12, 14, 17, 20, 22]. Grouping methods constitute the most popular approach as they allow the user to logically relate the graph summary to the original graph. Our work belongs in this category.

The flip side of summarization is loss of utility, or loss of "useful information" contained in the original graph. At a conceptual level, utility is defined by attempting to reconstruct the original graph G from a summary $\mathcal{G}$ thus obtaining reconstructed graph G'. Compared to G, graph G' can miss original edges that have been lost or can have spurious edges that have been added. We define the utility of summary $\mathcal{G}$ to be the utility of the reconstructed graph G'. The utility of G' is penalized in terms of the number of lost edges and spurious edges. As a consequence, the more structural similarity G' has with G, the higher its utility.

A problem with most previous works is that it is hard to predict the utility of their produced summaries. They do not incorporate measuring utility at each step of the algorithm. To the best of our knowledge the only works that present utility-aware algorithms are [22], [9], and [11]. In [22], Shin et al. present SWeG, an algorithm that preserves a neighborhood similarity measure for each pair of corresponding nodes in G and G'. This however is a local measure, not easily generalizable to global utility for the whole graph. Furthermore, the edges are considered of equal importance, thus further hampering the measuring of utility. In [9], Ko et al. present MoSSo, an incremental algorithm for maintaining lossless summaries of dynamic graphs. Similar to [22], MoSSo also is not conducive to considering a global notion of utility.

In [11], Kumar and Efstathopoulos are the first to address global utility. However, their approach, UDS, requires time $O(V^2)$; based on our experiments, it can only handle small to moderate datasets, often requiring more than 100 hours. Furthermore UDS only scratches the surface of what is possible to achieve in utility-based graph summarization. For example, for the case when we desire zero loss of utility, UDS performs rather poorly producing a summary that is not very different from the original graph it started from. On the other hand, for the case when utility loss is allowed, UDS uses simple criteria for merging nodes of the original graph thus producing summaries that can be vastly improved.

To address these challenges, we propose two utility-driven algorithms, G-SCIS and T-BUDS, for the lossless and lossy cases, respectively, which can handle large graphs efficiently on a consumer-grade machine. G-SCIS is based on a clique and independent set decomposition that produces significant compression with zero loss of utility. Compared to SWeG, MoSSo, and UDS, G-SCIS produces better summaries with respect to reduction in number of nodes, while having a running time lower by two-orders of magnitude.

We also show that G-SCIS summaries possess an attractive characteristic not present in SWeG, MoSSo or UDS summaries. Due to our clique and independent set decomposition, we are able to compute important classes of queries, such as Pagerank, triangle enumeration, and shortest paths using the G-SCIS summary "as-is" without the need to perform postprocessing or execute neighborhood queries as SWeG and MoSSo do.

Our second algorithm, T-BUDS, is a highly scalable iterative algorithm for the lossy case, which incorporates measuring utility at each iteration and allows the user to fully control the loss of utility according to their needs. T-BUDS significantly outperforms SWeG and UDS in terms of node reduction while requiring significantly less time and space. We achieve this by combining the use of weighted Jaccard similarity, a memory reduction technique based on Maximum Spanning Tree and a novel binary-search approach for merging nodes. In summary, our contributions are as follows.

- We propose an optimal algorithm, G-SCIS, for lossless graph summarization and show that it outperforms state of art by two orders of magnitude in runtime while achieving better reduction in number of nodes.
- We show interesting applications of the summary produced by G-SCIS to triangle enumeration, Pagerank, and shortest path queries. For instance, we show that we can enumerate triangles and compute Pagerank on the G-SCIS summaries much faster than on the original graph.
- We propose a utility-driven algorithm, T-BUDS, for lossy summarization. T-BUDS achieves high scalability and outperforms state-of-the-art by two orders of magnitude.
- We also show that T-BUDS significantly outperforms state-of-the-art in terms of utility achieved for a given level of node reduction. Conversely, for a given utility threshold, T-BUDS offers much better node reduction than state-of-the-art.

2 PRELIMINARIES

Let $G = (V, E)$ be an undirected graph, where V is the set of nodes and E the set of edges. A summary graph is also undirected and denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of supernodes, and $\mathcal{E}$ a set of superedges. More precisely we have $\mathcal{V} = \{S_1, S_2, \dots, S_k\}$ such that $k \leq |V|$, $V = \bigcup_{i=1}^{k} S_i$ and $\forall i \neq j, S_i \cap S_j = \emptyset$. The supernode which a node $u \in V$ belongs to is denoted by $S(u)$.

Reconstruction. Given a summary graph, we can (lossily) reconstruct the original graph as follows. For each superedge (S_i, S_j) we construct edges (u, v), for each $u \in S_i$ and $v \in S_j$. For $i \neq j$, this amounts to building a complete bipartite graph with S_i and S_j as its *parts*. For $i = j$ (a self-loop superedge), the reconstruction amounts to building a clique among the vertices of S_i.

Utility. In order to reason about the utility of a graph summarization we need to define the notion of edge importance. We denote the importance of an edge (u, v) in G by $C(u, v)$. For example, the edge importance could measure its centrality. Obviously, the more important edges we recover during reconstruction, the better it is. However, this should not come at the cost of introducing spurious edges. In order to measure the amount of spuriousness, we also introduce the notion of importance for spurious edges and denote that by $C_s(u, v)$. Now in a similar way to [11] we define the utility of a summary graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as follows.

$$u(\mathcal{G}) = \sum_{(S_i,S_j)\in\mathcal{E}} \left(\sum_{\substack{(u,v)\in E \\ u\in S_i, v\in S_j}} C(u,v) - \sum_{\substack{(u,v)\notin E \\ u\in S_i, v\in S_j}} C_s(u,v) \right) \tag{1}$$

In words, the utility of a summary is measured by summing the importance scores of edges of the original graph that can be reconstructed from it and subtracting the sum of importance scores of the spurious edges that are introduced by the reconstruction. $C(u, v)$ and $C_s(u, v)$ are normalized so that their respective sums equal one. We give a numeric example in Appendix.

In order to have a good summary, the user defines a threshold τ, $0 \leq \tau \leq 1$, and requests that $u(\mathcal{G}) \geq \tau$. Now we define the optimization problem we study as follows. Given graph $G = (V, E)$ and user-specified utility threshold τ, our objective is to

$$\text{minimize } |\mathcal{V}| \text{ subject to } u(\mathcal{G}) \geq \tau. \tag{2}$$

3 OPTIMAL LOSSLESS ALGORITHM

Kumar et al. [11] showed that given a utility threshold τ, computing the optimal graph summary with utility loss of at most τ is *NP-Hard*. In this section, we analyze the problem for the special case of $\tau = 1$, that is, lossless graph summarization. When we reconstruct the graph from such a summary, no actual edge will be lost and no spurious edge will be introduced. We show that we can obtain in *polynomial time* the *optimal* summary in terms of the objective function, i.e. the one with the smallest number of supernodes.

We denote by $N(u)$ the set of neighbors of vertex u. In the following whenever we refer to a clique we mean a clique C, such that if $u, v \in C$, then $N(u) \cup \{u\} = N(v) \cup \{v\}$. This implies the vertices of C share the same neighbours outside the clique. Similarly, whenever we refer to an independent set (a set of nodes where no two nodes are connected) we mean an independent set I, such that if $u, v \in I$, then $N(u) = N(v)$. We have the following lemmas.

Lemma 3.1. *In a summary for $\tau = 1$, each node can be (1) in a supernode of size one, or (2) inside a supernode representing a clique in G, or (3) inside a supernode representing an independent set in G.*

Proof. During reconstruction, a supernode either generates just one node, a clique (when a self-loop exists), or an independent set (when a self-loop does not exist). Now if the original graph does not precisely correspond to what is reconstructed, then there will be at least either one spurious edge added (in the case of a clique supernode), or one actual edge lost (in the case of an independent set supernode). Thus the summary would be lossy and the reconstructed graph would be different from the original graph. □

Lemma 3.2. *A node v cannot be in a clique supernode in one lossless summary and in an independent set supernode in another.*

Proof. Let us assume, if possible, that nodes u, v are inside an independent set supernode in one lossless summary and u, v are inside a clique supernode in another. This implies that $N(u) = N(v)$ and $N(u) \cup \{u\} = N(v) \cup \{v\}$, a contradiction. □

We now show that there is a polynomial-time algorithm that computes an optimal lossless summary. Algorithm 1 gives a global greedy strategy for finding such a summary. For each node u, the goal of the algorithm is to find the largest supernode that u can be a part of. For the summary to be lossless, such a supernode has to be either an independent set or a clique.

Condition in line 6 of Algorithm 1 checks whether vertices u and v can be in the same clique or independent set with neighborhood properties as described above. If so, u and v are greedily merged. Further, Lemma 3.2 proved that the two conditions in line 6 are mutually exclusive. That is, all the vertices in $S(u)$ will satisfy

Algorithm 1 Finding the best summary for $\tau = 1$

```
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:     $S(u) \leftarrow \{u\}$, $Status[u] \leftarrow True$
5:     for $v \in V \wedge Status[v] = False$ do
6:         if $(N(u) = N(v)) \vee (N(u) \cup \{u\} = N(v) \cup \{v\})$ then
7:             $S(u) \leftarrow S(u) \cup \{v\}$, $Status[v] \leftarrow True$
8:     $\mathcal{S}.add(S(u))$
9: BuildSuperEdges($\mathcal{S}$)
```

exactly one of these two conditions. If neither of these conditions holds true, then node u should be in a supernode of size one.

Building Superedges. Once the appropriate supernodes have been identified we build superedges as follows. For each supernode S, an edge is added to another supernode S' iff $u \in S$ and $v \in S'$ and $(u, v) \in E$. We refer to this process as BuildSuperEdges (last line of Algorithm 1).

Theorem 3.3 (**Tractability of lossless graph summarization**). *Lossless graph summarization is in P. That is, Algorithm 1 computes the optimal solution in polynomial time for $\tau = 1$.*

Proof. We claim that the supernode S_u containing a vertex $u \in V$ in the summary output by Algorithm 1 is the largest possible supernode for u in any lossless summary. Suppose S_u is an independent set supernode. All the nodes in S_u must have the same neighbor set as u. As Algorithm 1 greedily finds and adds *all* vertices $v \in V$ such that $N(v) = N(u)$ to S_u, this must be the largest set possible; the only way to make such a supernode larger is to include a node with different neighbor set. Doing so makes the summary be no longer lossless. An analogous argument applies for the case when S_u is a clique supernode.

We now show that Algorithm 1 produces an optimal lossless summary. For contradiction, let us assume that there exists an optimal lossless summary in which the number of supernodes is less than the summary provided by Algorithm 1. If so, there should exist at least one node $u \in V$ such that its supernode size in the optimal summary is larger than the its supernode size in the summary provided by Algorithm 1. However, we proved in the previous paragraph that this can never happen and hence is a contradiction. Finally, it can be verified that the time complexity of Algorithm 1 is $O(V^2\Delta_{max})$, where Δ_{max} is the maximum degree of a node in G and hence lossless summarization is in P. □

3.1 Scalable Lossless Algorithm, G-SCIS

Algorithm 1 is of $O(V^2\Delta_{max})$ time complexity, which makes it impractical for large datasets. Here we propose an improved algorithm of $O(E)$ complexity, which uses hashing to speed up the process. We can break down the process into three parts: (a) finding candidate supernodes, (b) filtering supernodes, and (c) connecting superedges.

A hash function is used to hash $N(u)$ and $N(v)$ in the case of independent sets, or $N(u) \cup \{u\}$ and $N(v) \cup \{v\}$ in the case of cliques, respectively. If u and v have the same hash value, then they are candidates to be merged into an independent set or clique supernode.

Note that the use of a hash function could result in candidate supernodes with false positives (i.e. two nodes that should not belong to same supernode might be present into one candidate supernode) but there cannot be false negatives (i.e. two nodes that must belong to same supernode cannot be in two different candidate supernodes). Of course, the probability of a false positive depends on the quality of the hash function used. In order to completely remove false positives, we further examine each candidate supernode for false positives, which are then filtered out into separate supernodes. After this step all the supernodes are as they should be in an optimal summary and finally the superedges are added between them.

In Algorithm 2, two different hash values (h_c and h_i) are generated for the neighbor sets of each node. The nodes that have the same h_c value (line 4) are grouped together to form candidate clique supernodes. Similarly, the nodes that have the same h_i value (line 5) are grouped together to form candidate independent set supernodes. Note that due to possible false positives, there can exist a node that is present in both a candidate independent set and a candidate clique at the same time. Finally, Algorithm 2 returns two hashmaps, *mapC* and *mapI*, where keys are hash values and buckets contain the set of nodes falling in the same candidate clique or independent set supernode.

Algorithm 3 filters the candidate supernodes to become correct supernodes. For any candidate supernode, it selects a random node u, and, using its neighbourhood list, removes all the other nodes v in that supernode for which the appropriate condition is not satisfied. Namely, we have $N(u) \cup \{u\} = N(v) \cup \{v\}$ for the case of clique and $N(u) = N(v)$ for the case of independent set. If the quality of the hash function is perfect, i.e. no false positives occur, then the loop in line 4 executes only once and Algorithm 3 is very efficient. On the other hand, if there are false positives, the loop will execute several times. While we can devise perfect hashing (see [6]), in practice we observed that we have few false positives even for simple default hash functions and the number of iterations was always small.

Algorithm 4 is the main algorithm that drives the whole process and produces the summary. It obtains the two hashmaps *mapC* and *mapI* using Algorithm 2 (line 3). It then removes the false positives using Algorithm 3 (lines 4 and 5). Lines 7 to 10 handle the supernodes of size one. Finally, the superedges are built in line 11.

Time and space complexity: The work space requirement[1] of Algorithm 2 is $O(V)$ due to the fact that two hashmaps *mapC* and *mapI* as well as list of supernodes $\mathcal{S}$ only use $O(V)$ space. The runtime is $O(E)$ as the hash function has to traverse each neighbor set of each node. Similarly, building superedges takes $O(V)$ space and $O(E)$ runtime. Algorithm 3 takes $O(V)$ space. Its runtime, as mentioned above, depends on the quality of the hash function. For a perfect hash function (no false positives) this is $O(E)$. We observe very close to this order in practice even for simple hash functions, e.g. $x\%p$, where p is a large prime. The better the hash function the more (non-empty) buckets we have, but their number cannot be more than $O(V)$. To summarize, the runtime of Algorithm 4 is $O(E)$ and its work space requirement is $O(V)$.

[1]Not considering the read-only input graph and the write-only summary graph.

Algorithm 2 Candidate Supernodes

```
1: Input: $G = (V, E)$, $h$    ▷ hash function to map list to number
2: $mapC \leftarrow \{\}$ , $mapI \leftarrow \{\}$    ▷ hash maps
3: for $v \in V$ do
4:     $h_c \leftarrow h((N(v) \cup \{v\})_{sorted})$
5:     $h_i \leftarrow h(N(v)_{sorted})$
6:     $mapC[h_c] \leftarrow mapC[h_c] \cup \{v\}$
7:     $mapI[h_i] \leftarrow mapI[h_i] \cup \{v\}$
8: return $mapC, mapI$
```

Algorithm 3 Filter Supernodes

```
1: Input: $map, type$    ▷ map containing candidate supernodes
2: $\mathcal{S} \leftarrow []$    ▷ list of filtered supernodes
3: for $X \in values(map)$ do    ▷ for each candidate supernode
4:     while $X \neq \phi$ do
5:         $u \leftarrow$ remove-random-node($X$)
6:         if type = clique then
7:             $\mathcal{S}(u) \leftarrow \{v \in X \mid N(u) \cup \{u\} = N(v) \cup \{v\}\}$
8:         else $\mathcal{S}(u) \leftarrow \{v \in X \mid N(u) = N(v)\}$
9:         if $\mathcal{S}(u) \neq \{u\}$ then
10:            $X \leftarrow X \setminus \mathcal{S}(u)$
11:            $\mathcal{S}.append(\mathcal{S}(u))$
12: return $\mathcal{S}$
```

Algorithm 4 Scalable algorithm for $\tau = 1$

```
1: Input: $G = (V, E)$
2: $Status[\forall v \in V] \leftarrow FALSE$ , $\mathcal{S} \leftarrow []$    ▷ list of supernodes
3: $mapC, mapI \leftarrow$ CandidateSuperNodes($G$)
4: $C \leftarrow$ FilterSuperNodes($mapC, type = clique$)
5: $I \leftarrow$ FilterSuperNodes($mapI, type = independentset$)
6: $\mathcal{S}.append(C)$, $\mathcal{S}.append(I)$
7: for $\mathcal{S}_i \in \mathcal{S}$ do
8:     for $u \in \mathcal{S}_i$ do $Status[u] \leftarrow True$
9: for $u \in V$ AND $Status[u] = False$ do
10:    $\mathcal{S}.append(\{u\})$
11: BuildSuperEdges($\mathcal{S}$)
```

4 SCALABLE LOSSY ALGORITHM, T-BUDS

Although G-SCIS achieves significant compression without any loss in utility, in this section we discuss a scalable lossy algorithm, T-BUDS, to further compress the summary graph $\mathcal{G}$ produced by G-SCIS while minimizing the loss in utility. Our algorithm can work with a G-SCIS summary as well as with the input graph; hence we continue to denote the input by $G = (V, E)$. The output is a more compressed, lossy summary. T-BUDS iteratively merges pairs of (super)nodes until the utility of the graph drops below a user-specified threshold $\tau < 1$. Intuitively, it is desirable that any two nodes in the same supernode have similar neighborhoods. A starting point is to look at the two-hop away nodes, as they have at least one neighbor in common. However, we do not stop here; we use a special version of weighted Jaccard similarity that incorporates the weight of edges in order to come up with a proper score for similarity of nodes' neighborhoods. Based on this score we decide the merge sequence of nodes.

We denote the set of two-hop away nodes by F. To decide the order of merge operations, we consider F as the candidate pairs set. T-BUDS starts merging from the less desirable pairs (based on weighted Jaccard as described in Section 4.1) of F because they result in less damage to the utility. It iterates over a sorted version of F and in each iteration performs the following steps.

(1) Pick the next pair of candidate nodes $\langle u, v\rangle$ from F, find their corresponding supernodes $S(u), S(v)$, and merge them into a new supernode S, if $S(u) \neq S(v)$.
(2) Update the neighbors of S based on the neighbors of $S(u), S(v)$. In particular, add an edge from S to another supernode if the loss in utility is less than the loss if not added.
(3) (Re)compute the utility of the summary built so far and stop if the threshold is reached.

We reiterate that while this description provides the intuition behind T-BUDS, additional techniques described below are needed to ensure its time and space efficiency.

4.1 Ordering Candidate Pairs

We order the node pairs in F using weighted Jaccard similarity and make sure that only the highly similar nodes are merged together. Using weighted Jaccard similarity enables us to capture both the importance score of each edge as well as the number of the common neighbors between any pair of two-hop nodes $\langle u, v\rangle$.

Weighted Jaccard (*WJ*) Similarity: For a pair of nodes $\langle u, v\rangle$

$$WJ(u,v) = \frac{\sum_{x \in N(u) \cap N(v)} \min(w(u,x), w(v,x))}{\sum_{x \in N(u) \cup N(v)} \max(w(u,x), w(v,x))} \tag{3}$$

where we define the weights $w(u, x)$ as follows. If u is connected to x we define $w(u, x) = 2 \cdot \max_C - (C_u + C_x)$, where $\max_C = \max_{u \in V} C_u$, otherwise $w(u, x) = 0$. Observe that the higher the centrality of x, the lower $w(u, x)$ is. The intuition for the above is as follows. In the extreme when the neighborhoods of two nodes u and v are the same (i.e. mergable according to G-SCIS), we have that their degrees and centrality scores are the same, thus we have $WJ = 1$. Relaxing this, in order to merge u and v, we still want a high amount of neighborhood commonness, hence we use a form of Jaccard similarity between sets of neighbors. In this process, we would like to weigh the high centrality neighbors below low centrality ones. This stems from the fact that in general, we try to avoid merging high centrality nodes because this can cause a high loss in utility. Now, the greater the number of high-centrality neighbors, the higher the centrality of u and v usually is. Therefore, we give low priority to these u, v pairs by applying the proposed weighted Jaccard similarity. To summarize, we order the merge operations by sorting the two-hop away pairs by their WJ score in descending order.

Using Maximum Spanning Tree (MST). We observe that not every candidate pair will cause a merge. This is because the nodes in the pair can be already in a supernode together due to previous merges. Therefore, there are many useless pairs, which we can eliminate with our proposed MST technique below.

Let us denote the two-hop graph by $G_{2-hop} = (V, F)$. That is, $F = \{(a, c) | (a, b) \in E \text{ and } (b, c) \in E\}$. We propose a method to reduce the number of candidate pairs from $O(|F|)$ to $O(|V|)$ by creating an MST of G_{2-hop}. In Theorem 4.1, we prove that using the sorted edge list of MST of G_{2-hop} will produce exactly the same summary as using the sorted edge list of G_{2-hop}.

Let us denote by L the weight-based sorted version of F. Also, we denote by H the sorted list of edges of an MST for G_{2-hop}. We now present a sufficiency theorem, which says that using H instead of L as the list of candidates is sufficient. The idea of the proof is that the candidate pairs leading to a merge when L is used, in fact, exactly correspond to the edges of an MST.

THEOREM 4.1 (**MST SUFFICIENCY THEOREM**). *For utility threshold τ, using H as the list of candidate pairs will produce the same graph summary as using L.*[2]

PROOF. Initially $\mathcal{G}$ is same as G and let us assume that at iteration i a new pair $\langle u,v\rangle \leftarrow L[i]$ is chosen and $S(u)^{i-1}$ and $S(v)^{i-1}$ are their corresponding supernodes. If $S(u)^{i-1} \neq S(v)^{i-1}$ then they should be merged together into a new supernode. The following two claims ensure the sufficiency of H as a candidate set.
(1) If $\langle u_1,v_1\rangle$ and $\langle u_2,v_2\rangle$ are in H s.t. $\langle u_1,v_1\rangle$ appears before $\langle u_2,v_2\rangle$ in H then $\langle u_1,v_1\rangle$ appears before $\langle u_2,v_2\rangle$ in L.
(2) If u and v are not inside a same supernode, that is $S(u)^{i-1} \neq S(v)^{i-1}$, then $\langle u,v\rangle$ must be in H.
Proof of (1): As both H and L are sorted based on the weighted Jaccard similarity of the pairs, the order in which $\langle u_1,v_1\rangle$ and $\langle u_2,v_2\rangle$ appear in H will be the same as their order in L.
Proof of (2): $S(u)^{i-1} \neq S(v)^{i-1}$ implies that there does not exist any other pair $\langle u',v'\rangle \leftarrow L[j]$ for any $j < i$ such that $u' \in S(u)^{i-1}$ and $v' \in S(v)^{i-1}$. Otherwise, $S(u')^j$ would have been merged with $S(v')^j$ in the j-th iteration. Thus, u' and v' would belong to the same supernode and $S(u)^{i-1}$ should be same as $S(v)^{i-1}$. Hence, $\langle u,v\rangle$ is the largest weighted edge in G_{2-hop} connecting $S(u)^{i-1}$ and $S(v)^{i-1}$. We want to show now that $\langle u,v\rangle \in H$ i.e. part of the MST. To show this, we claim that, in fact, $\langle u,v\rangle$ is the largest weight edge in G_{2-hop} connecting $S(u)^{i-1}$ and $V\backslash S(u)^{i-1}$. Suppose not. Let us consider the edges between $S(u)^{i-1}$ and $V \setminus S(u)^{i-1}$. Recall that a cut in a connected graph is a minimal set of edges whose removal disconnects the graph. Therefore, the edges between $S(u)^{i-1}$ and $V \setminus S(u)^{i-1}$ form a cut in G_{2-hop}. A well known property called *cut property* of MST (maximum spanning tree) states that the maximum weight edge of any cut belongs to the MST [10]. Now let, if possible, a different edge, $\langle u'',v''\rangle$ in G_{2-hop} be the edge with the largest weight connecting $S(u)^{i-1}$ and $V \setminus S(u)^{i-1}$. Then by the cut property, $\langle u'',v''\rangle$ belongs to H and would have been considered as a candidate pair for merge in an earlier iteration. In that case, u'' and v'' will belong to the same supernode which is a contradiction. □

Using Locality Sensitive Hashing (LSH). Since computing the weighted Jaccard similarity for all the possible two-hop away nodes is an overhead, we deploy a locality sensitive hashing scheme presented in [24] to partition the two-hop graph into multiple buckets. We only compute weighted Jaccard for nodes u, v of edges $(u,v) \in F$ that fall in the same bucket. On the other hand, if u and v fall in different buckets, we consider the score of edge (u,v) to be zero.

[2]There can be different sorted versions of L due to possible ties (albeit unlikely as weights are real numbers). What this theorem shows is that the summary constructed based on MST is the same as the summary constructed using *some* sorted version of L.

Theorem 4.1 holds the same. For ease of notation we continue to call by WJ this LSH-based version of weighted Jaccard similarity.

4.2 Merging Candidate Pairs

Based on Theorem 4.1, we can use H instead of L for the list of candidate pairs. Furthermore, we show in following theorem that the utility is non-increasing as we merge candidate pairs of H in order. It can be verified that

THEOREM 4.2 (**NON-INCREASING UTILITY THEOREM**). *Let $\mathcal{G}^0 = G$ and $\mathcal{G}^t$ be the summary graph obtained by processing H in order from index 1 to t where $1 \le t \le |H|$. Then $u(\mathcal{G}^{t-1}) \ge u(\mathcal{G}^t)$.*

Theorems 4.1 and 4.2 form the basis of our new approach T-BUDS that uses binary search over the sorted list of MST edges, H, in order to find the largest index t for which $u(\mathcal{G}^t) \ge \tau$ (see Algorithm 5). This requires computing H (Algorithm 6) followed by $\lg(|H|)$ computations of utility. The latter is done using Algorithm 7.

Given graph $G = (V,E)$ and centrality scores for each node $C[u \in V]$, T-BUDS first creates the sorted candidate pairs H by calling the Two-hop MST function (Algorithm 6). This function follows the structure of Prim's algorithm [19] for computing MST. However, we do not want to build the G_{2-hop} graph explicitly. As such, we start with an arbitrary node s and insert it into a priority queue Q with a key value of ∞. All other nodes are initialized with a key value of 0. For any given node v with maximum key value deleted from Q, v is included in the MST, and the key values of its two-hop away neighbours are updated, when needed.

After creating the two-hop MST and sorting its edges, T-BUDS uses a binary search approach and iteratively performs merge operations from the first pair until the middle pair in H (Algorithm 5). In each iteration, we pick a pair of nodes u, v from H, find their supernodes $S(u)$ and $S(v)$ and merge them into a new supernode S. This process continues until the algorithm reaches the middle point. $\mathcal{G}$ is the resulting summary after these operations and we compute its utility in line 11. If this utility $\ge \tau$, then we search for the index t in the second half, otherwise, we search for the index t in the first half. The algorithm finds the best summary in $\lg|H|$ iterations and $|H|$, being the number of edges in the MST of G_{2-hop}, is just $O(V)$.

Algorithm 5 T-BUDS

```
1:  **Input:** $G = (V,E), C, \tau$
2:  $H \leftarrow$ TwoHopMST(G,C)
3:  $low \leftarrow 0, high \leftarrow |H| - 1$
4:  **while** $low \le high$ **do**
5:      $mid \leftarrow \frac{low+high}{2}, \mathcal{V} \leftarrow V, i \leftarrow 0$
6:      **for** $i \le mid$ **do**
7:          $\langle u,v\rangle \leftarrow H[i], i \leftarrow i+1$
8:          $S \leftarrow$ Merge($S(u), S(v)$)
9:          $\mathcal{V} \leftarrow (\mathcal{V} \setminus \{S(u), S(v)\}) \cup S$
10:     $u(\mathcal{G}) \leftarrow$ ComputeUtility($\mathcal{V}$)
11:     **if** $u(\mathcal{G}) \ge \tau$ **then** $high = mid - 1$
12:     **else** $low = mid + 1$
13: BuildSuperEdges($\mathcal{V}$)
```

Algorithm 7 computes the utility for a specific summary $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In following discussion we will assume that the centrality of any edge (u, v) is defined as $C(u, v) = \frac{C_u}{d_u} + \frac{C_v}{d_v}$. Also, we will assume all the non-existent edges are assigned equal weights and each such edge gets a weight $C_S(u, v) = \frac{1}{\binom{|V|}{2} - |E|}$. Our analysis can be easily adapted to other definitions of $C(u, v)$ and $C_S(u, v)$.

The algorithm iterates over all supernodes one at a time and for a given supernode S_i, it creates two maps (*count* and *sum*) to hold the details for the superedges connected to S_i. $count[S_j]$ stores the number of actual edges between supernodes S_i and S_j. Similarly, $sum[S_j]$ contains the sum of the weights for all the edges between S_i and S_j. Lines 4 to 12 initialize these two structures. Sedge(S_i, S_j) (the cost of drawing a super edge between S_i and S_j) and nSedge(S_i, S_j) (the cost of not drawing a super edge between S_i and S_j) can be estimated using *count* and *sum*. As nSedge(S_i, S_j) is the sum of weights of edges in G between nodes in S_i and S_j, it is exactly equal to $sum[S_j]$ (line 14). If $S_i \neq S_j$, the number of spurious edges is equal to $|S_i||S_j| - count[S_j]$ and since each spurious edge has cost $\frac{1}{\binom{|V|}{2} - |E|}$, Sedge$(S_i, S_j) = \frac{|S_i||S_j| - count[S_j]}{\binom{|V|}{2} - |E|}$ (line 15). Similarly, if $S_i = S_j$, the number of spurious edges is $\binom{|S_i|}{2} - count[S_j]$ and Sedge$(S_i, S_j) = \frac{\binom{|S_i|}{2} - count[S_j]}{\binom{|V|}{2} - |E|}$ (line 16). Finally the utility loss can be estimated as min(Sedge(S_i, S_j) , nSedge(S_i, S_j)) and the utility is decremented by the loss. Algorithm 7 returns the final utility for $\mathcal{G}$ which is used by Algorithm 5 to make decisions.

Building Superedges. Once the appropriate supernodes have been identified, a superedge is added between two supernodes S_i and S_j if and only if Sedge$(S_i, S_j) \leq$ nSedge(S_i, S_j). This task can be completed in $O(|E|)$ time: Line 18 of Algorithm 7 can be replaced by the task of adding superedge between S_i and S_j.

Algorithm 6 Two-hop MST

```
1:  Input: $G = (V, E), C$    ▷ $C$ is centrality scores array for nodes
2:  $key[s] \leftarrow \infty, parent[s] \leftarrow Null, Q.insert(s, key[s])$
3:  for ($v \in V \setminus \{s\}$) do
4:      $key[v] \leftarrow 0, parent[v] \leftarrow Null, Q.insert(v, key[v])$
5:  while $!isEmpty(Q)$ do
6:      $(v, \_) = Q.delMax()$
7:      for ($w \in N(N(v)) \mid w \in Q \ \& \ w \neq v$) do
8:          if $key[w] < WJ(v, w)$ then
9:              $Q.setKey(w, WJ(v, w))$
10:             $parent[w] \leftarrow v$
11: $H \leftarrow \{(v, parent[v]) : v \in V \setminus \{s\}\}$
12: return sorted $H$ based on $C$
```

Data structures. We used the union-find algorithm [8] for representing our supernodes. The union operation was used to implement the merge operation in line 8 of Algorithm 5 and the find operation was used to find the corresponding supernode for a specific node in line 8 of Algorithm 5 and line 6 of Algorithm 7. Using path compression with the union-find algorithm allows reducing the complexity of the union and find operations to $O(\lg^\star |V|)$ (iterated logarithm of $|V|$). As $\lg^\star |V|$ is about 5 when $|V|$ is even more than a billion, we treat it as a constant in our calculations. The union-find algorithm only needs two arrays of size $|V|$ and thus the working memory requirement is $O(|V|)$.

Algorithm 7 Compute Utility

```
1:  Input: $G = (V, E), utility \leftarrow 1, \mathcal{V}$        ▷ set of supernodes
2:  for $S_i \in \mathcal{V}$ do                              ▷ for each supernode
3:      $count \leftarrow \{\}, sum \leftarrow \{\}$
4:      for $u \in S_i$ do
5:          for $v \in N(u)$ do
6:              $S_j \leftarrow S(v)$
7:              if $(S_i \neq S_j) \vee (S_i = S_j \wedge i < j)$ then
8:                  if $count[S_j] \geq 1$ then
9:                      $count[S_j] \leftarrow count[S_j] + 1$
10:                     $sum[S_j] \leftarrow sum[S_j] + C(u, v)$
11:                 else
12:                     $count[S_j] \leftarrow 1, sum[S_j] \leftarrow C(u, v)$
13:     for $S_j \in count.keys \wedge i \leq j$ do
14:         nSedge$(S_i, S_j) \leftarrow sum[S_j]$
15:         if $S_i \neq S_j$ then Sedge$(S_i, S_j) \leftarrow \frac{|S_i||S_j| - count[S_j]}{\binom{|V|}{2} - |E|}$
16:         else Sedge$(S_i, S_j) \leftarrow \frac{\binom{|S_i|}{2} - count[S_j]}{\binom{|V|}{2} - |E|}$
17:         if Sedge$(S_i, S_j) \leq$ nSedge$(S_i, S_j)$ then
18:             $utility \leftarrow utility -$ Sedge$(S_i, S_j)$
19:         else $utility \leftarrow utility -$ nSedge$(S_i, S_j)$
20: return $utility$
```

Complexity analysis. We can show that the time complexity of T-BUDS is $O(((|F| \cdot \Delta_{max} + |V|) \cdot \lg |V|))$ and the space complexity is $O(|V|)$ (Appendix). Of course in practice by using LSH we reduce the $O(F \cdot \Delta_{max})$ complexity to only $O(F' \cdot \Delta_{max})$, where F' is the subset of u, v pairs in F such that u and v fall in the same LSH bucket.

5 EXPERIMENTS

The experimental evaluation is divided into the following four parts:

(1) Performance analysis of G-SCIS versus lossless versions of SWeG, UDS and MoSSo [9, 11, 22] in terms of running time and node reduction.[3]
(2) Performance analysis of G-SCIS on Pagerank computation and triangle enumeration (see Appendix).
(3) Performance analysis of T-BUDS versus lossy versions of SWeG and UDS in terms of running time, reduction in nodes, and scalability.
(4) Accuracy analysis of T-BUDS versus SWeG and UDS in terms of top-k queries on the reconstructed graph.

Except for MoSSo for which the source code is publicly available, we implemented all other algorithms in Java 14 (https://anonymous.4open.science/r/64699cbc-f4a7-4d16-b30b-3c9f2072a169/) on a single machine with dual 6 core 2.10 GHz Intel Xeon CPUs, 64 GB RAM and running Ubuntu 18.04.2 LTS. In [11] UDS was run on higher-end AWS hardware for more than 2 weeks, whereas we have a time cutoff of 100 hours (about 4 days) on commodity hardware. We used seven web and social graphs from (http://law.di.unimi.it/datasets.php) varying from moderate size to very large (Table 1).

[3]MoSSo only performs lossless summarization.

Graph	Abbr	Nodes	Edges
cnr-2000	CN	325,557	5,565,380
hollywood-2009	H1	1,139,905	113,891,327
hollywood-2011	H2	2,180,759	228,985,632
indochina-2004	IC	7,414,866	304,472,122
uk-2002	U1	18,520,486	529,444,615
arabic-2005	AR	22,744,080	1,116,651,935
uk-2005	U2	39,459,925	1,581,073,454

Table 1: Summary of datasets

5.1 Lossless Case: G-SCIS

In this section, we evaluate the performance of G-SCIS in terms of (1) reduction in nodes, (2) running time, and (3) efficiency of Pagerank computation and triangle enumeration. We observed that the reduction in nodes of UDS [11] for the lossless case is not competitive with the other algorithms; in all our datasets the UDS reduction in nodes was less than 0.01 (1%). Therefore we decided to not show the UDS numbers alongside G-SCIS, SWeG and MoSSo.

5.1.1 Comparison of G-SCIS to SWeG and MoSSo. The reduction in nodes (RN) is defined as $RN = (|V| - |\mathcal{V}|)/|V|$ (c.f. [11]). Since SWeG and MoSSo produce also correction graphs to add/delete (C^+, C^-), RN for them is more precisely computed as $RN = (|V| - (|\mathcal{V}| \cup |V_{C^+}| \cup |V_{C^-}|)) / |V|$. We ran SWeG for different choices of the number of iterations up to 80 and chose the best RN value obtained. We use the same configuration for MoSSo as in [9], 0.3 escape probability and 120 sample size for each trial. Since MoSSo is an incremental algorithm, we started from an empty graph and inserted one edge at a time and updated the summary after each step until all edges are inserted.

Figure 1 shows the comparison between G-SCIS, SWeG, and MoSSo in terms of RN and running time.[4] G-SCIS outperforms both SWeG and MoSSo in terms of RN and is also orders of magnitude faster. On large graphs like AR and U2, SWeG is not runnable within 100 hours while G-SCIS finishes in about 15 and 23 minutes respectively. MoSSo, on the other hand, is much faster than SWeG and is able to finish on large graphs but it is still not competitive with G-SCIS on both running time and RN. For example, on H2, G-SCIS is 131x faster and produces 6x more compression than MoSSo.

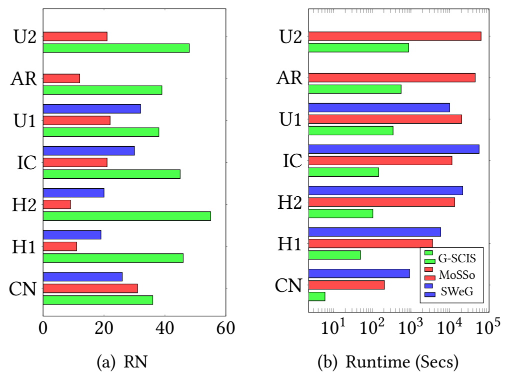


Figure 1: G-SCIS vs SWeG vs MoSSo in terms of node reduction and running time. G-SCIS achieves better reduction than both SWeG and MoSSo. Runtime of G-SCIS is orders of magnitude better than them. SWeG could not run within 100h for AR and U2.

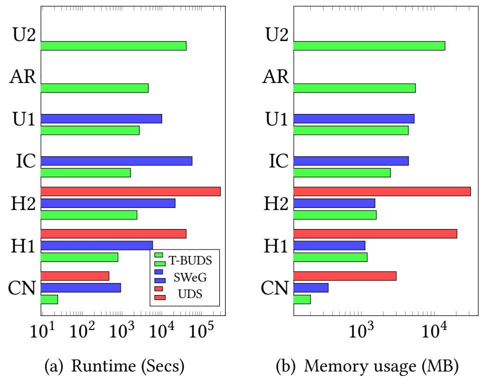


Figure 2: T-BUDS vs UDS and SWeG in terms of runtime in (sec). τ is set to 0.8. T-BUDS is faster than both UDS and SWeG. We provide our MST edge pairs as input to UDS because the original version of UDS could not complete within 100h for all the datasets but CN. Still, even with MST as input, UDS could not complete for IC, U1, AR, and U2.

5.2 Lossy Case: T-BUDS

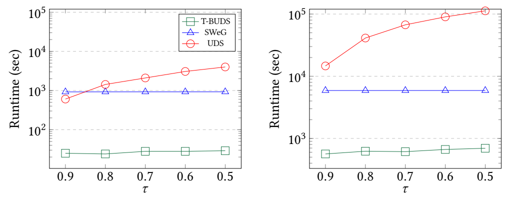
5.2.1 Performance of T-BUDS. In this section, the performance of T-BUDS is compared to the performance of UDS and SWeG in terms of running time and memory usage (Figure 2). For our comparison, we set the utility threshold at 0.8. In order to extend SWeG to support edge weights, we subtract the weight of each removed edge from the current utility value in the dropping step of [22]. Dropping continues until the value of utility drops below the threshold.

We observe that UDS ([11]) is quite slow on moderate and large datasets. Namely, it was not able to complete in reasonable time (100h) for none of the datasets. As such, we provide as input to UDS not the full list of 2-hop pairs as in [11], but the reduced list from the MST of G_{2-hop}. This way, we were able to handle with UDS the datasets CN, H1, and H2. However, we still could not have UDS complete for the rest of the datasets. SWeG on the other hand, is much faster than UDS but still significantly slower than T-BUDS.

Specifically, in Figure 2, we observe that T-BUDS outperforms UDS in running time by orders of magnitude. T-BUDS can easily deal with the largest graph, U2, in less than 7 hours. In contrast UDS takes more than 90 hours on a moderate graph, such as H2, to produce results. T-BUDS also outperforms SWeG significantly (see for example CN and IC). Regarding memory consumption both T-BUDS and SWeG need orders of magnitude less memory than UDS.

In another experiment we compare the performance of T-BUDS, UDS and SWeG for varying utility thresholds. Figure 3 shows the runtime of the three algorithms on two different graphs CN and H1 in terms of varying utility threshold. Having an algorithm that is computationally insensitive to changing the threshold is desirable

[4]When we compare our algorithms to SWeG, MoSSo and UDS, we use green for G-SCIS and T-BUDS, blue for SWeG and red for either UDS or MoSSo.



(a) CN (b) H1

Figure 3: Runtime of T-BUDS, UDS and SWeG for different utility thresholds on CN and H1. T-BUDS and SWeG are independent of threshold while UDS heavily depends on it. T-BUDS is order of magnitude faster than both SWeG and UDS.

because it allows the user to conveniently experiment with different values of the threshold. As shown in the figure, the runtimes of T-BUDS and SWeG remain almost unchanged across different utility thresholds. In contrast, UDS strongly depends on the utility threshold and its runtime grows as the threshold decreases.

In Figure 4, we compare the summarization performance (RN) values for each algorithm subject to the utility threshold. As results show, T-BUDS significantly outperforms SWeG and UDS for all datasets and all threshold values considered (0.5 - 1.0). For instance, for threshold value 0.7 on CN, T-BUDS offers a node reduction of 0.83, much higher than UDS (RN=0.59) and SWeG (RN=0.38). UDS starts off worse than SWeG for higher values of τ but improves for lower values; it is still worse than T-BUDS for any value of τ. For medium and big datasets, UDS cannot complete within 100 hours. Hence there are no results for UDS for IC and U1.

5.2.2 Accuracy analysis of top-k query on reconstructed graph. We study the performance of T-BUDS towards top-k query answering. To do so, we compute the PageRank centrality (P) for the nodes, and assign (normalized) importance score to each edge (u, v), $C(u, v)$, based on the importance scores of its two endpoints. We then compute the summary using T-BUDS. Similarly, we also summarize graphs using UDS [11] and SWeG [22]. In the end we reconstruct the graph from the summary of each method (T-BUDS, UDS, and SWeG) and run PageRank centrality on them. We obtain the top $k\%$ of central nodes in the reconstructed graph and match against the actual top $k\%$ central nodes in the original graph. A higher number of matches indicates that the lossy summary obtained is better at preserving the graph structure.

Table 2 shows the top-$k\%$ match performance of T-BUDS with varying RN on two graphs, CN and H1. The five columns after RN show results for different levels of k in top-$k\%$ queries. The match performance of T-BUDS is impressive. For instance, on H1 for RN of 0.5 we get a 97% match of the top-10% nodes in the original graph.

Now we compare the performance of T-BUDS vs. the lossy versions of SWeG and UDS in terms of RN. In order to find the summary of SWeG given the RN value, we first obtain the lossless summary of SWeG and then in the dropping step find the appropriate value for error bound ϵ (see [22]) which results in the same value for RN.

Figure 5 shows the relative improvement of T-BUDS over UDS and SWeG. Each subfigure shows the relative improvement of T-BUDS over SWeG and UDS for a specific value $k\%$ of top-$k\%$. For

Graph	RN	10%	20%	30%	40%	50%
CN	0.50	0.85	0.92	0.94	0.95	0.96
	0.55	0.79	0.87	0.91	0.93	0.94
	0.60	0.75	0.84	0.88	0.90	0.91
	0.65	0.69	0.78	0.84	0.88	0.89
	0.70	0.61	0.71	0.78	0.83	0.85
H1	0.50	0.97	0.98	0.98	0.99	0.99
	0.55	0.87	0.90	0.93	0.94	0.95
	0.60	0.82	0.87	0.90	0.92	0.93
	0.65	0.74	0.83	0.85	0.87	0.89
	0.70	0.65	0.76	0.80	0.81	0.83

Table 2: Match of top-$k\%$ PageRank query on reconstructed graph from T-BUDS summary. RN is the node reduction level we consider. Each of the following columns represent a level of k in the top-$k\%$ query. The numbers in these columns shows the ratio of the top-$k\%$ nodes that appear in both the original and reconstructed graphs.

each choice of top $k\%$ query, we run PageRank on the original graph and on the reconstructed graph obtained from the summaries of the three algorithms. We match the top $k\%$ central nodes of each reconstructed graph with the top $k\%$ central nodes of the original graph. We observe T-BUDS to be significantly better than both SWeG and UDS as seen in Figure 5. Its relative improvement over UDS and SWeG is impressive; mostly above 60% and 20%, respectively.

6 RELATED WORK

We can classify the proposed methodologies in graph summarization into two general categories, grouping and non-grouping. The non-grouping category includes sparsification-based methods [16] and sampling-based methods [13]. For a more detailed analysis of non-grouping methods, see the survey by Liu et al. [15].

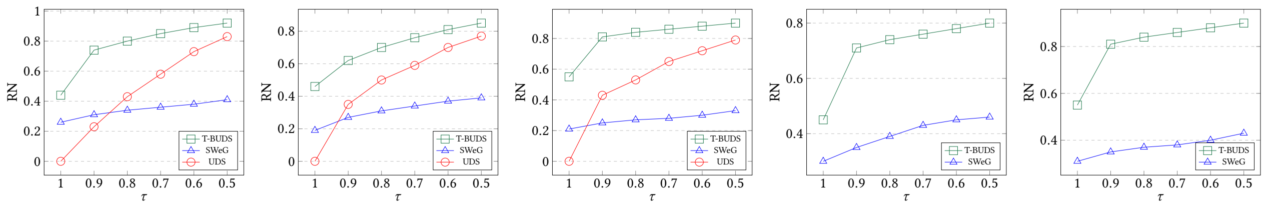
The grouping category of methods is more commonly used for graph summarization and as such has received a lot of attention [7, 9, 11, 12, 14, 17, 20, 22, 23]. In this category, works such as [12, 20] can only produce lossy summarizations optimizing different objectives. On the other hand, [17, 22] are able to generate both lossy and lossless summarizations. Among works of the grouping category, we discuss the following works [9, 11, 14, 17, 22] that aim to preserve utility and as such are more closely related to our work.

Navlakha et al. [17] introduced the technique of summarizing the graph by a compact representation containing the summary along with correction sets. Liu et al. [14] proposed a distributed solution to improve its scalability. Recently, Shin et al. [22], proposed SWeG, that builds on the work of [17]. They used a shingling and minhash based approach to prune the search space for discovering promising candidate pairs. MoSSo is a recent incremental algorithm for summarizing dynamic graphs using correction sets [9].

Kumar and Efstathopoulos [11] presented the UDS algorithm that preserves the utility above a user specified threshold. However, UDS cannot be effectively used for lossless summarization as its summary is very close to the original graph. Furthermore, for the lossy case, UDS is not scalable to moderate or large graphs.

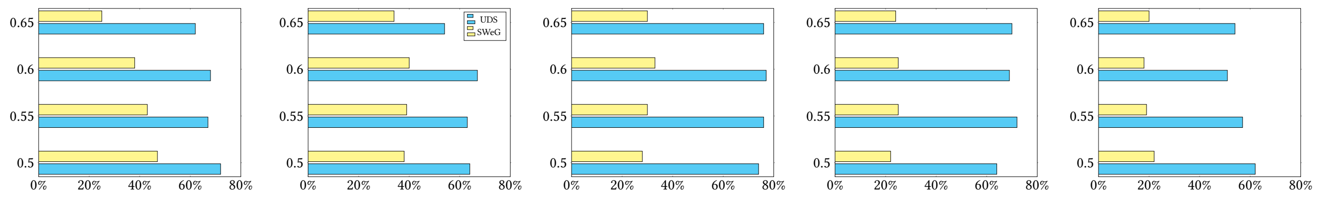
7 CONCLUSIONS

In this work, we study utility-driven graph summarization in-depth and made several novel contributions. We present a new, lossless



(a) CN (b) H1 (c) H2 (d) IC (e) U1

Figure 4: RN values for different utility thresholds. For each threshold, T-BUDS gives more compression than SWeG and UDS.



(a) top 10% (b) top 20% (c) top 30% (d) top 40% (e) top 50%

Figure 5: Relative improvement for top-k% query answering of T-BUDS over SWeG and UDS. The cyan bar shows the improvement of T-BUDS over UDS and the yellow bar shows the improvement of T-BUDS over SWeG. The y axis shows different RN values (i.e. 0.5, 0.55, 0.6, 0,65) and the x axis shows the percentage of relative improvement of T-BUDS over each algorithm.

graph summarizer, G-SCIS, that can output the optimal summary, with the smallest number of supernodes. We design a scalable, lossy summarization algorithm, T-BUDS, that uses weighted Jaccard similarity for measuring neighbourhood similarity. Two key insights leading to the scalability of T-BUDS are the use of MST of the two-hop graph combined with binary search over the MST edges.

REFERENCES

[1] Facebook by the numbers: Stats, demographics & fun facts. https://www.omnicoreagency.com/facebook-statistics. Accessed: 2020-05-23.

[2] How many websites are there around the world? [2020]. https://www.millforbusiness.com/how-many-websites-are-there. Accessed: 2020-05-23.

[3] Number of sina weibo users in china from 2017 to 2021. https://www.statista.com/statistics/941456/china-number-of-sina-weibo-users. Accessed: 2020-05-23.

[4] Twitter by the numbers: Stats, demographics & fun facts. https://www.omnicoreagency.com/twitter-statistics. Accessed: 2020-05-23.

[5] Boldi, P., and Vigna, S. The webgraph framework i: compression techniques. In *WWW* (2004), pp. 595–602.

[6] Fredman, M. L., Komlós, J., and Szemerédi, E. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM 31*, 3 (1984), 538–544.

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

[8] Hopcroft, J. E., and Ullman, J. D. Set merging algorithms. *SIAM Journal on Computing 2*, 4 (1973), 294–303.

[9] Ko, J., Kook, Y., and Shin, K. Incremental lossless graph summarization. In *KDD* (2020), pp. 317–327.

[10] Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Math. Soc. 7*, 1 (1956), 48–50.

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

[12] LeFevre, K., and Terzi, E. Grass: Graph structure summarization. In *SDM* (2010), pp. 454–465.

[13] Liberty, E. Simple and deterministic matrix sketching. In *KDD* (2013).

[14] Liu, X., Tian, Y., He, Q., Lee, W.-C., and McPherson, J. Distributed graph summarization. In *CIKM* (2014), pp. 799–808.

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

[16] Maccioni, A., and Abadi, D. J. Scalable pattern matching over compressed graphs via dedensification. In *KDD* (2016), pp. 1755–1764.

[17] Navlakha, S., Rastogi, R., and Shrivastava, N. Graph summarization with bounded error. In *SIGMOD* (2008), pp. 419–432.

[18] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[19] Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal 36*, 6 (1957), 1389–1401.

[20] Riondato, M., García-Soriano, D., and Bonchi, F. Graph summarization with quality guarantees. In *ICDM* (2014), pp. 947–952.

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

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

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

[24] Shrivastava, A., and Li, P. Improved densification of one permutation hashing. In *UAI* (2014), pp. 732–741.

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

8 APPENDIX

8.1 Example of the Utility Framework

Figure 6 shows an example for this framework. There are 14 edges and 11 nodes. We assume that the weight of each actual edge is equal to $\frac{1}{|E|} = \frac{1}{14}$ and the weight of each spurious edge is equal to $\frac{1}{\binom{11}{2}-14} = \frac{1}{41}$. That is, there are 41 spurious edges in total and the weight of each is set in this example to be equal to 1/41. In part (a) the set of nodes inside the circles merge together into new supernodes and the utility still remains one because no information has been lost. In part (b) the circles show two merge cases. In the first case, the blue supernode merges with the red node and in the second case, the green supernode merges with the blue node. In the first case, there is a utility loss of $\frac{1}{14}$ for missing one actual edge (see part (d) for the reconstructed graph). We chose not to add an edge from the new blue supernode to one of the neighbours of the red node because doing so would introduce three spurious edges for a cost of $\frac{3}{41}$ that is greater than $\frac{1}{14}$ (cost of missing one actual edge). Similarly, in the second case, there is a utility loss of $\frac{2}{41}$ for introducing two spurious edges. Therefore, the utility after this step is $1 - \frac{1}{14} - \frac{2}{41} = \frac{505}{574}$. Part (c) shows the summary after all the four merges and part (d) shows the reconstructed graph of summary in part (c).

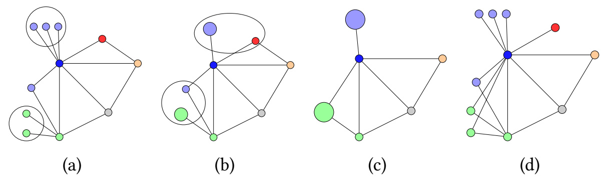


Figure 6: Example of the utility-based framework. (a) Shows the original graph with two candidate merges with no loss of utility. The result is shown in (b) along with two more candidate merges. The merge of the green supernode with the blue node introduces two spurious edges (see the relevant part in the reconstructed graph in (d)). The merge of the blue supernode with the red node loses an actual edge as shown by the result in (d). (d) shows the reconstructed graph starting from the summary graph in (c).

8.2 How to query G-SCIS graph summaries?

In general there are two ways to query graph summaries. The first is to reconstruct the original graph, incrementally and on-the-fly, then answer queries. For example, using neighborhood-queries as a primitive illustrates such a reconstruction (c.f. [22]). The run time of this approach is at least as much as querying the original graph.

The second approach is to devise query answering algorithms that work directly on the summary graph and never reconstruct the original graph. This class of algorithms has the potential to produce significant gains in running time compared to executing the query on the original graph. Here we propose three such algorithms for summaries produced by G-SCIS. They are for computing Pagerank, enumerating triangles, and answering shortest path queries, which form the basis for many graph-analytic tasks.

Computing Pagerank. We show now how to find the Pagerank scores of all nodes in G without reconstructing G.

Let $P^i(u)$ denote the Pagerank value of any node u after i-th iteration of the Pagerank algorithm [18]. For any undirected graph $G = (V, E)$, all the nodes are initialized with the same Pagerank value i.e. $\forall_{u \in V} P^0(u) = 1$. In iteration i, it is updated as follows: $P^i(u) \leftarrow \sum_{w \in N(u)} \frac{P^{i-1}(w)}{|N(w)|}$. In this equation we ignore damping factor for simplicity but it can be easily incorporated without impacting our results. We can show the following result.

Theorem 8.1. *For any supernode $S \in \mathcal{V}$, all the nodes inside S must have the same Pagerank value.*

To calculate the exact Pagerank scores of the nodes in G using its summary $\mathcal{G}$, we propose Algorithm 8, an adaptation of the Pagerank algorithm, that runs directly on $\mathcal{G}$. Algorithm 8 maintains the invariant that the Pagerank of a supernode after iteration i is the sum of the Pagerank of its nodes after iteration i of the Pagerank algorithm. It initializes the Pagerank of a supernode to be its size (line 2). It computes the number of neighbours of a node inside a supernode X (lines 5 and 7). Using this, it updates the Pagerank of supernode X in iteration i (line 10 to 13). Finally, it computes the Pagerank of each node of G from the Pagerank of its supernode in $\mathcal{G}$ (line 16). We can show the following theorem.

Theorem 8.2. *Algorithm 8 outputs exactly the same Pagerank score for each node v in G as the Pagerank algorithm.*

Enumerating Triangles. Triangle enumeration using G-SCIS summary is described in Algorithm 9. It can be extended to enumerate other types of graphlets, such as squares, 4-cliques, etc.

There are three types of triangles in the summary: (a) those having all three vertices in the same supernode, (b) those having two vertices in one supernode and one in another, and (c) those having all vertices in different supernodes. The idea underlying Algorithm 9 is to enumerate type-(a) and type-(b) triangles by iterating over the clique supernodes in $\mathcal{G}$ and to generate type-(c) triangles by considering all the supernodes (cliques and independent sets).

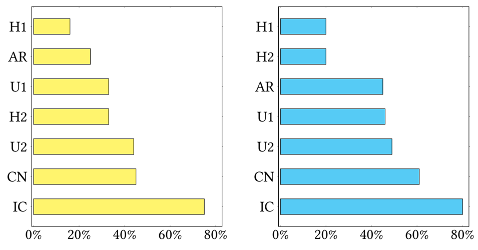
Let X be a clique supernode. Type-(a) triangles from X can be found by listing every subset of three vertices in X (see lines 3 and 4). Type-(b) triangles with two vertices in X can be computed by listing every subset of two vertices in X combined with every subset of one vertex from a neighbor supernode Y (lines 5 and 6).

Finally, any triangle enumeration algorithm can be used on the summary graph to find all the super triangles (triangles formed by three supernodes). Type-(c) triangles can now be listed as follows. If (X, Y, Z) is a super triangle, then all the corresponding type-(c) triangles can be listed by combining every choice of the first node from X, second node from Y, and third node from Z (lines 7 to 9).

Answering Shortest Path Queries. We observe that $\mathcal{G}$ can be used to compute lengths of shortest paths between any two nodes $u, v \in G$ in time $O(|\mathcal{E}| + |\mathcal{V}|)$. We can show the following.

Theorem 8.3. *Given nodes u, v such that $S(u) = S(v)$, we have*

1. *If $S(u)$ is a clique, the shortest path length between u and v in G, $d(u, v)$, is 1.*
2. *If $S(u)$ is an independent set and $|N(S(u))| > 0$, then $d(u, v) = 2$. Otherwise, $d(u, v) = \infty$.*



(a) PageRank (b) Triangle

Figure 7: Runtime improvement of Pagerank and triangle enumeration when running directly on G-SCIS summary vs running on SWeG summary using neighbor queries.

Algorithm 8 Pagerank using G-SCIS summary

```
1: Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
2: Initialization: $\forall X \in \mathcal{V}, P^0(X) = |X|, i \leftarrow 1$
3: for $X \in \mathcal{V}$ do
4:     if $X \notin N(X)$ then
5:         $W(X) \leftarrow \sum_{Y \in N^-(X)} |Y|$                ▷ X is IS
6:     else
7:         $W(X) \leftarrow \sum_{Y \in N^-(X)} |Y| + (|X| - 1)$    ▷ X is clique
8: while $P^i \neq P^{i-1}$ do                                  ▷ until convergence
9:     for $X \in \mathcal{V}$ do
10:        if $X \notin N(X)$ then                                ▷ X is IS
11:            $P^i(X) \leftarrow \sum_{Y \in N^-(X)} \frac{|X| \cdot P^{i-1}(Y)}{W(Y)}$
12:        else                                                    ▷ X is clique
13:            $P^i(X) \leftarrow \sum_{Y \in N^-(X)} \frac{|X| \cdot P^{i-1}(Y)}{W(Y)} + \frac{(|X|-1) \cdot P^{i-1}(X)}{W(X)}$
14: for $X \in \mathcal{V}$ do
15:     for $u \in nodes(X)$ do
16:         $P(u) \leftarrow \frac{P^i(X)}{|X|}$
17: return $P$
```

Algorithm 9 Enumerating Triangles

```
1: Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ , triangle-enum  ▷ State of the art triangle
   enumeration algorithm
2: for $X \in \mathcal{V}$ do
3:     if $X \in N(X)$ then                              ▷ $X$ has a superloop
4:         Output all type a triangles in $X$            ▷ $\binom{|X|}{3}$ triangles
5:         for $Y \in \{N(X) \setminus X\}$ do
6:             Output all type b triangles having 2 vertices
                in X and 1 vertex in Y                   ▷ $\binom{|X|}{2}|Y|$ triang.
7:     super-triangles ← triangle-enum($\mathcal{G}$)
8:     for $(X, Y, Z) \in$ super-triangles do
9:         Output all type c triangles in (X,Y,Z)  ▷ |X||Y||Z| triang.
```

Theorem 8.4. *Given nodes u, v such that $S(u) \neq S(v)$, $d(u, v)$ is equal to the length of shortest path between $S(u)$ and $S(v)$ in $\mathcal{G}$.*

Based on the above theorems we present Algorithm 10 for computing $d(u, v)$ given two nodes $u, v \in V$ using a G-SCIS summary.

Algorithm 10 Shortest Paths using G-SCIS summary

```
1: Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E}), u, v \in V$
2: if $S(u) = S(v)$ then
3:     if $S(u)$ is a clique then
4:         $d(u, v) = 1$
5:     else
6:         if $N(S(u)) > 0$ then
7:             $d(u, v) = 2$
8:         else
9:             $d(u, v) = \infty$
10: else
11:     $d(u, v) = d(S(u), S(v))$
12: return $d(u, v)$
```

8.2.1 *Pagerank computation and triangle enumeration.* In Figure 7, we show the reduction in runtime for Pagerank computation and triangle enumeration using G-SCIS summaries as described in Section 8.2 versus the runtime using SWeG summaries with neighbor (adjacency list) queries ([22]). We see a significant reduction in time for both queries for all datasets, reaching up to 80% for IC. We omit results for shortest paths due to space constraints. We observe in Figure 7 (left) and (right) a similar order of datasets with some exceptions, such as H2 or U1, for which the order is reversed. We attribute this to the size of the output in triangle enumeration.

8.3 Complexity analysis of T-BUDS

Let us begin by analysing the time complexity of Algorithm 6. As its structure follows that of Prim's algorithm [19], and computing weighted jaccard similarity adds an overhead of $O(\Delta_{max})$ for each pair (assuming the neighbor lists are sorted), As the number of edges in H is $O(|V|)$, sorting it takes $O(|V| \lg |V|)$ time. it requires $O(|F| \cdot \lg |V| \cdot \Delta_{max})$ steps to compute MST. Thus, the total time complexity of Algorithm 6 is $O((|F| \cdot \Delta_{max} + |V|) \cdot \lg |V|)$.

The total space required by Algorithm 6 is $O(|V|)$ as it stores the priority queue Q and arrays *key*, *parents*, and H all of size $O(|V|)$.

Now let us analyse the time complexity of Algorithm 7. To compute the utility of $\mathcal{G}$, the algorithm iterates over all the edges in G, each edge exactly once, to identify pairs of supernodes (S_i, S_j) that have at least one edge of G between them. This step, that includes the computation of *count* and *sum* for each supernode, takes $O(E)$ time. Once this step is completed, it takes $O(1)$ time to compute the Sedge and nSedge cost for a pair (S_i, S_j). Therefore, the time complexity of Algorithm 7 is $O(|E|)$. It requires $O(|V|)$ space to store the count and sum arrays.

Finally, let us analyse the time and space complexity of Algorithm 5. As discussed in Section 5.2, Algorithm 5 will perform $\lg |H|$ iterations. For each iteration, merging supernodes in Algorithm 5 requires $O(|H|)$ operations and the utility estimation using Algorithm 7 requires $O(|E|)$ time. Thus the time complexity for each iteration is $O((|E| + |V|)$ and time for a total of $\lg |H|$ iterations is $O((|E| + |V|) \cdot \lg(|V|))$. The space requirement inside Algorithm 5 is storing H and $\mathcal{V}$, which is $O(|V|)$. Thus, the space requirement of Algorithm 5 is $O(|V|)$. Summarizing all the above, we have

Theorem 8.5. *The time complexity of T-BUDS is $O(((|F| \cdot \Delta_{max} + |V|) \cdot \lg |V|))$. The space complexity of T-BUDS is $O(|V|)$.*