Efficient Graph Summarization of Large Networks

by

Mahdi Hajiabadi
B.Sc., Amirkabir University of Technology, 2012
M.Sc., University of Tehran, 2016

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Efficient Graph Summarization of Large Networks

by

Mahdi Hajiabadi
B.Sc., Amirkabir University of Technology, 2012
M.Sc., University of Tehran, 2016

Supervisory Committee

Dr. Venkatesh Srinivasan, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Fayez Gebali, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

In this thesis, we study the notion of graph summarization, which is a fundamental task of finding a compact representation of the original graph called the summary. Graph summarization can be used for reducing the footprint of the input graph, better visualization, anonymizing the identity of users, and query answering.

There are two different frameworks of graph summarization we consider in this thesis, the utility-based framework and the correction set-based framework. In the utility-based framework, the input graph is summarized until a utility threshold is not violated. In the correction set-based framework a set of correction edges is produced along with the summary graph. In this thesis we propose two algorithms for the utility-based framework and one for the correction set-based framework. All these three algorithms are for static graphs (i.e. graphs that do not change over time). Then, we propose two more utility-based algorithms for fully dynamic graphs (i.e. graphs with edge insertions and deletions).

Algorithms for graph summarization can be lossless (summarizing the input graph without losing any information) or lossy (losing some information about the input graph in order to summarize it more). Some of our algorithms are lossless and some lossy, but with controlled utility loss.

Our first utility-driven graph summarization algorithm, G-SCIS, is 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.

Our second algorithm is T-BUDS, a highly scalable, utility-driven algorithm 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. Also, 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.

We then propose algorithm LDME, a correction set-based graph summarization algorithm that produces compact output representations in a fast and scalable man-

ner. To achieve this, we introduce (1) weighted locality sensitive hashing to drastically reduce the number of comparisons required to find good node merges, (2) an efficient way to compute the best quality merges that produces more compact outputs, and (3) a new sort-based encoding algorithm that is faster and more robust. More interestingly, our algorithm provides performance tuning settings to allow the option of trading compression for running time. On high compression settings, LDME achieves compression equal to or better than the state of the art with up to 53x speedup in running time. On high speed settings, LDME achieves up to two orders of magnitude speedup with only slightly lower compression.

We also present two lossless summarization algorithms, Optimal and Scalable, for summarizing fully dynamic graphs. More concretely, we follow the framework of G-SCIS, which produces summaries that can be used as-is in several graph analytics tasks. Different from G-SCIS, which is a batch algorithm, Optimal and Scalable are fully dynamic and can respond rapidly to each change in the graph. Not only are Optimal and Scalable able to outperform G-SCIS and other batch algorithms by several orders of magnitude, but they also significantly outperform MoSSo, the state-of-the-art in lossless dynamic graph summarization. While Optimal produces always the most optimal summary, 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. An interesting fact that we observed experimentally is that even if we were to run a batch algorithm, such as G-SCIS, once for every big batch of changes, still they would be much slower than Scalable. For instance, if 1 million changes occur in a graph, Scalable is two orders of magnitude faster than running G-SCIS just once at the end of the 1 million-edge sequence.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my supervisors, Prof. Venkatesh Srinivasan, and Prof. Alex Thomo, for their generous support and great encouragement to conduct this research as well as their valuable comments to enhance the quality of the dissertation. This dissertation would not be possible without them. I want to thank Prof. Fayez Gebali for his guidance as a member of my supervisory committee. A special thanks to Prof. Kijung Shin, who graciously accepted to act as the external examiner for my oral examination. I am grateful for my family, especially my brother Mohammad, who supported me full-heartedly throughout my degree. Next, thanks to my colleagues and lab mates, Jasbir Singh and Quinton Yong for their contributions and vital discussions we had before.

I would like to thank the staff in the computer science department, namely Wendy Beggs, Kath Milinazzo, Nancy Chan and Erin Robinson for their help with every administrative during my time here. A special thanks to Glen McCloskey for fixing my computer several times and lending chargers and other stuff to me. A special thank to all my close friends, my bike, my Xbox and my consoles for making me feel better whenever I got stuck in life.

Chapter 1

Introduction

Graphs are ubiquitous and are the most natural representation for many real-world data such as web graphs, social networks, communication networks, citation networks, transaction networks, ecological 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, 4, 3].

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 query more efficiently [15, 44, 51]. Graph summarization also makes possible effective visualization thus facilitating better insights on large-scale graphs [60, 14, 37, 30, 13]. Also crucial is the privacy that a graph summary can provide for privacy-aware graph analytic [34, 21].

The problem has been approached from different directions, such as compression techniques to reduce the number of required bits for describing graphs [48, 7, 11, 49], sparsification techniques to remove less important nodes/edges in order to make the graph more informative [54, 42] and grouping methods that merge nodes into supernodes based on some interestingness measure [34, 44, 51, 47, 39]. Grouping methods constitute the most popular summarization approach because they allow the user to logically relate the graph summary to the original graph. The grouping based methods are mainly categorized into three main categories: utility based approaches [32, 47], correction-set based approaches [51, 44, 28] and approaches for inferring the expected adjacency matrices of summaries [34, 47, 33, 8].

The first and second categories are the focus of this thesis. On the other hand, algorithms in the third category aim at inferring the *expected* adjacency matrix by

minimizing the error between the reconstructed graph from the summary and the original graph [33, 34, 47, 8]. However there are some limitations with these algorithms, namely, they are not scalable to large graphs with more than a billion edges, they produce a lossy summary and the losslessness is not guaranteed. In addition, they need to store weights which may cause the footprint of the summary graph to exceed the original graph.

In the following we explore the utility-based framework and correction set based framework in more detail and discuss our contributions to these frameworks. Finally, we explore the lossless graph summarization problem in the fully dynamic scenario in which edges are inserted or removed from the original graph and discuss our proposed lossless algorithms that are able to incrementally update the summary graph in constant time.

1.1 Utility-based summarization

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 [51], [28], and [32]. In [51], 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 [28], Ko et al. present MoSSo, an incremental algorithm for maintaining lossless summaries of dynamic graphs. Similar to [51], MoSSo also is not conducive to considering a global notion of utility.

In [32], 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.

1.1.1 Contributions

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.

1.2 Summarization in fully dynamic scenario

We also tackle the problem of summarizing massive dynamic graphs that come as a fast stream of edge insertions and deletions [57, 28, 20]. 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 [25, 40, 12]. There are many summarization methods, such as graph compression to reduce the volume of input graph [11], graph sparsification to remove less important nodes or edges [42], and group-based graph summarization (GGS) to group similar nodes and edges into supernodes and superedges [51, 44, 32, 33, 47]. 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 [61, 57, 18, 50], they produce lossy summaries, i.e., salient information in

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

the original graph can be irrecoverably lost.

Recently, Ko et al. in [28] proposed MoSSo, the first lossless dynamic GGS algorithm, which builds on the framework of SWeG [51] 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, these summaries can mainly be used as compression devices, rather than data structures to speed up graph analytics.

In contrast, the G-SCIS framework, introduced by Hajiabadi et al. in [19], produces summaries that can be used as-is to speed up important classes of graph analytics, such as graphlet enumeration, centrality computation, and shortest paths. However, the G-SCIS algorithm given in [19] 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 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.

1.2.1 Contributions

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 al-

gorithm, 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 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 that provides the smallest-anytime 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.

1.3 Correction-set based framework

Correction-set based framework, is another popular framework in grouping based approach. In this approach, as defined in [44, 51], the output representation consists of a summary graph and correction set. The correction set is used to reconstruct the original graph from the summary graph either perfectly (lossless) or with some loss in information (lossy). Correction set based graph summarization algorithms consist of three broad steps: merge nodes of the original graph into supernodes of the summary graph, encode the original edges into superedges of the summary graph and correction set, and drop some edges from the summary graph and correction set to yield a more compact output (for lossy case).

The current state of the art correction set based graph summarization algorithm is SWeG of [51]. SWeG is faster than all of its competitors, yields better compression than other methods, and can also run in a distributed setting. SWeG improves upon the original framework of [44] by adding a dividing step that divides the nodes into smaller groups prior to merging (for parallelizability and efficiency) and introducing an approximation metric for finding nodes to merge. Despite the impressive performance of SWeG compared to other algorithms, there are several steps in the algorithm which bottleneck its performance. In particular, the merging algorithm is quadratic in the size of groups, so its running time suffers due to the dividing step not creating small enough groups of nodes. The merging step also uses an approximation metric to find good merges since [51] did not present an efficient way to directly compute the true

best merges in a group. Finally, the encoding algorithm also becomes a bottleneck since it scales quadratically based on the number of supernodes, making it perform poorly for larger graphs.

1.3.1 Contributions

We propose LDME (**L**ocality Sensitive Hashing **D**ivide **M**erge **E**ncode), an efficient and scalable correction set based graph summarization method which makes optimizations in each step of SWeG. In particular, LDME introduces weighted locality sensitive hashing to reduce the amount of computation during the merge phase, uses an efficient method of computing the best merges, and implements a faster and more scalable encoding algorithm. Our optimizations are such that we can now handle large datasets requiring only a single machine without the need for expensive clusters of machines. Additionally, LDME includes the benefit of being able to tune its performance to trade off compression for running time. In particular, LDME with high compression settings achieves up to 53x speedup with equal or better compression than SWeG and with high speed settings achieves up to two orders of magnitude speedup with only a small loss in compression. In summary, our contributions are as follows.

- New node dividing algorithm which introduces weighted locality sensitive hashing to significantly improve the running time of the bottleneck merging step
- Efficient method to directly compute the best nodes to merge that gives better overall compression and is faster than the approximation metric used in SWeG
- New edge encoding algorithm that scales better based on only the number of edges in the original graph and is up to 26x faster than SWeG (especially on very large graphs)
- A performance tuning technique to allow the choice of more compressed output representation or faster running time
- Extensive experimental results showing the speedup and scalability of our approach that is able to handle billion-scale datasets on a single machine.

1.4 Selected Publications

In the following, I have listed my publications which show the outcome of my research and collaboration, as the main author, with my supervisors and lab mates.

- Yong Q, Hajiabadi M, Srinivasan V, Thomo A. Efficient graph summarization using weighted lsh at billion-scale. In Proceedings of the 2021 International Conference on Management of Data 2021 Jun 9 (pp. 2357-2365).
- Hajiabadi M, Singh J, Srinivasan V, Thomo A. Graph Summarization with Controlled Utility Loss. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining 2021 Aug 14 (pp. 536-546).
- Hajiabadi M, Srinivasan V, Thomo A. Dynamic Graph Summarization: Optimal and Scalable, conference version in submission, 2022.

1.5 Dissertation Outline

The outline of this dissertation is as follows.

- **Chapter 2** explores the utility based graph summarization algorithms in more detail and provides an overview of existing works in literature and proposes two utility-based graph algorithms, G-SCIS which is a lossless summarization algorithm, and T-BUDS which is a lossy summarization algorithm. This chapter reflects our research published in [19].
- **Chapter 3** focuses on graph summarization in a fully dynamic scenario, and proposes two different algorithms for lossless dynamic graph summarization. They are called *Optimal* and *Scalable*. Optimal produces the smallest-any-time lossless summary. Scalable does not have this guarantee, however, it is still lossless, and a magnitude faster than Optimal. This chapter reflects our recent work which is under submission.
- **Chapter 4** focuses on the correction-set based framework for graph summarization. It investigates the current state-of-the-art algorithms in this framework and proposes LDME (**L**ocality Sensitive Hashing **D**ivide **M**erge **E**ncode), an efficient and scalable correction set-based graph summarization method. This chapter reflects our research published in [59].

- **Chapter 5** concludes the dissertation and discusses potential future work.

Chapter 2

Graph Summarization with Controlled Utility Loss

This chapter starts with an overview of literature on utility-based graph summarization. Next, we introduce the problem statement and the proposed methods, namely G-SCIS for lossless summarization and T-BUDS for lossy summarization. We then propose how we are able to use G-SCIS summary as-is to answer queries with more effective time and finally show experimental results and conclusions.

2.1 Related Work

Graph summarization has been studied in different contexts and we can classify the proposed methodologies into two general categories, grouping and non-grouping. The non-grouping category includes sparsification-based methods [36, 42] and sampling-based methods [5, 38]. For a more detailed analysis of non-grouping methods, see the survey by Liu et al. [40].

The grouping category of methods is more commonly used for graph summarization and as such has received a lot of attention [32, 34, 44, 47, 51, 28, 39]. In this category, works such as [34, 47] can only produce lossy summarizations optimizing different objectives. On the other hand, [44, 51] are able to generate both lossy and lossless summarizations. Among works of the grouping category, we discuss the following works [32, 39, 44, 51, 28] that aim to preserve utility and as such are more closely related to our work.

Navlakha et al. [44] introduced the technique of summarizing the graph by a com-

pact representation containing the summary along with correction sets. Two different algorithms greedy and randomized were proposed in [44] to produce a compact summary along with the correction sets. Liu et al. [39] proposed a distributed solution to improve its scalability. Recently, Shin et al. [51], proposed SWeG, that builds on the work of [44]. 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 [28]. While SWeG [51] summaries are queryable, the use of correction sets makes SWeG inefficient for queries such as Pagerank or triangle enumeration.

Finally, we note that there are other methods in literature which also aim to create summary graphs with the goal of minimizing reconstruction error, albeit differently [34, 47]. They could find some interesting bounds in the quality of summary or the accuracy of query answering such as number of triangles, but in fact storing the expected adjacency matrix of the summary graph required much enough memory to unable these methods to create summary for large graphs. GraSS [34] leveraged the idea to generate a probabilistic approximate adjacency matrix, $\bar{A}$,upon which incoming queries are computed. Due to maintaining $|V| \times |V|$ matrix and $|V|$ (number of nodes in graph) being some billions in social network graphs like Facebook, the approach has high memory requirements and is thus not scalable.

The objective function is to minimize normalized reconstruction error so that the summary can behave like the original graph. If A is the original adjacency matrix for $G = (V, E)$ and $\bar{A}$ is the real valued approximate adjacency matrix, then the normalized reconstruction error is evaluated as

$$error = \frac{1}{|V|^2} \sum_{i \in V} \sum_{j \in V} |\bar{A}(i, j) - A(i, j)| \tag{2.1}$$

While GraSS [34] did not specify any guarantees on the queries, Riondato et al. [47] provided an approach with theoretical bounds for the query efficiency. The proposed algorithm was the first polynomial-time approximation algorithm of it's kind. Given k number of supernodes, the objective function is to minimize p-norm of $A - \bar{A}$ as compared to normalized reconstruction error in GraSS [34].

Kumar and Efstathopoulos [32] presented UDS, an algorithm that preserves the utility above a user specified threshold. The framework assigned an importance score to each edge based on their centrality and the loss of utility is measured using these centrality scores. A lossy algorithm, UDS, was proposed using a memoization-based

approach and its effectiveness was established in terms of quality and overall practicality. However, UDS cannot be effectively used for lossless summarization as the summary generated for such a case is very close to the original graph. Furthermore, for the lossy case, UDS is not scalable to moderate or large graphs.

2.2 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 $\mathcal{V} = \{S_1, S_2, \ldots, S_k\}$ 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 [32] we define the utility of a summary graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as follows.

$$u(\mathcal{G}) = \sum_{(S_i, S_j) \in \mathcal{E}} \left(\sum_{\substack{(u,v) \in E \\ u \in S_i, v \in S_j}} C(u, v) - \sum_{\substack{(u,v) \notin E \\ u \in S_i, v \in S_j}} C_s(u, v) \right) \tag{2.2}$$

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.

Figure 2.1 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).

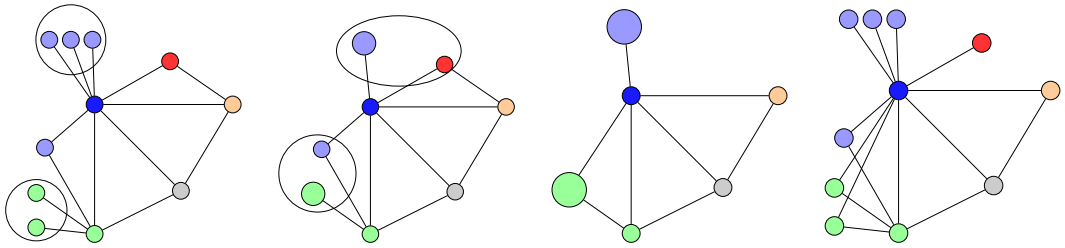


Figure 2.1: Example of the utility-based framework. (a) Shows the original graph with two candidate merges with no loss of utility. The result is shown in (b) along with two more candidate merges. The merge of the green supernode with the blue node introduces two spurious edges (see the relevant part 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).

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

$$\text{minimize } |\mathcal{V}| \text{ subject to } u(\mathcal{G}) \geq \tau. \tag{2.3}$$

2.3 Optimal lossless algorithm

Kumar et al. [32] 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 2.3.1. *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 2.3.2. *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.

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: $\quad S(u) \leftarrow \{u\}$, $Status[u] \leftarrow True$
5: $\quad$ **for** $v \in V \wedge Status[v] = False$ **do**
6: $\quad\quad$ **if** $(N(u) = N(v)) \vee (N(u) \cup \{u\} = N(v) \cup \{v\})$ **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($\mathcal{S}$)

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 2.3.2 proved that the two conditions in line 6 are mutually exclusive. That is, all the vertices in $S(u)$ will satisfy 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 2.3.3 (**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. □

2.3.1 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 [17]), 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 $O(V)$.

2.3.2 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. [51]).

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

Algorithm 2 Candidate Supernodes

Input: $G = (V, E)$, h ▷ hash function to map list to number
$mapC \leftarrow \{\}$, $mapI \leftarrow \{\}$ ▷ hash maps
for $v \in V$ **do**
    $h_c \leftarrow h((N(v) \cup \{v\})_{sorted})$
    $h_i \leftarrow h(N(v)_{sorted})$
    $mapC[h_c] \leftarrow mapC[h_c] \cup \{v\}$
    $mapI[h_i] \leftarrow mapI[h_i] \cup \{v\}$
return $mapC, mapI$

Algorithm 3 Filter Supernodes

Input: $map, type$ ▷ map containing candidate supernodes
$S \leftarrow []$ ▷ list of filtered supernodes
for $X \in values(map)$ **do** ▷ for each candidate supernode
    while $X \neq \phi$ **do**
        $u \leftarrow$ remove-random-node(X)
        if type = clique **then**
            $S(u) \leftarrow \{v \in X \mid N(u) \cup \{u\} = N(v) \cup \{v\}\}$
        else $S(u) \leftarrow \{v \in X \mid N(u) = N(v)\}$
        if $\mathcal{S}(u) \neq \{u\}$ **then**
            $X \leftarrow X \setminus \mathcal{S}(u)$
            $\mathcal{S}.append(\mathcal{S}(u))$
return $\mathcal{S}$

Algorithm 4 Scalable algorithm for $\tau = 1$

Input: $G = (V, E)$
$Status[\forall v \in V] \leftarrow FALSE$, $S \leftarrow []$ ▷ list of supernodes
$mapC, mapI \leftarrow$ CandidateSuperNodes(G)
$C \leftarrow$ FilterSuperNodes($mapC, type = clique$)
$I \leftarrow$ FilterSuperNodes($mapI, type = independentset$)
$\mathcal{S}.append(C)$, $\mathcal{S}.append(I)$
for $\mathcal{S}_i \in \mathcal{S}$ **do**
    for $u \in \mathcal{S}_i$ **do** $Status[u] \leftarrow True$
for $u \in V$ **AND** $Status[u] = False$ **do**
    $\mathcal{S}.append(\{u\})$
BuildSuperEdges($\mathcal{S}$)

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 [45]. 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 2.3.4. *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 5, an adaptation of the Pagerank algorithm, that runs directly on $\mathcal{G}$. Algorithm 5 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 2.3.5. *Algorithm 5 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 6. 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

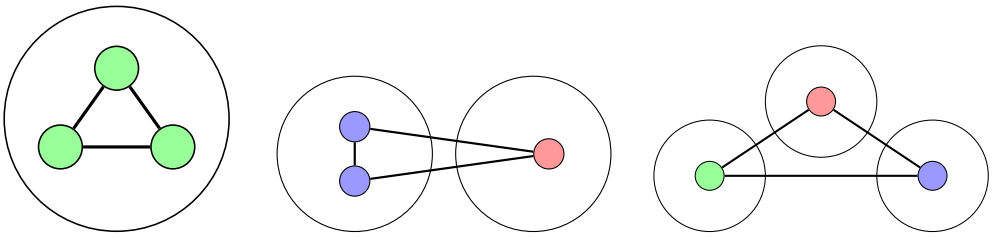


Figure 2.2: Three different type of triangles

one in another, and (c) those having all vertices in different supernodes. The idea underlying Algorithm 6 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 2.3.6. *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, $d(u, v) = \infty$.*

Theorem 2.3.7. *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 7 for computing $d(u, v)$ given two nodes $u, v \in V$ using a G-SCIS summary.

Algorithm 5 Pagerank using G-SCIS summary

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Initialization: $\forall X \in \mathcal{V}, P^0(X) = |X|, i \leftarrow 1$
for $X \in \mathcal{V}$ **do**
    if $X \notin N(X)$ **then**
        $W(X) \leftarrow \sum_{Y \in N^-(X)} |Y|$ ▷ X is IS
    else
        $W(X) \leftarrow \sum_{Y \in N^-(X)} |Y| + (|X| - 1)$ ▷ X is clique
while $P^i \neq P^{i-1}$ **do** ▷ until convergence
    for $X \in \mathcal{V}$ **do**
        if $X \notin N(X)$ **then** ▷ X is IS
            $P^i(X) \leftarrow \sum_{Y \in N^-(X)} \frac{|X| \cdot P^{i-1}(Y)}{W(Y)}$
        else ▷ X is clique
            $P^i(X) \leftarrow \sum_{Y \in N^-(X)} \frac{|X| \cdot P^{i-1}(Y)}{W(Y)} + \frac{(|X|-1) \cdot P^{i-1}(X)}{W(X)}$
for $X \in \mathcal{V}$ **do**
    for $u \in nodes(X)$ **do**
        $P(u) \leftarrow \frac{P^i(X)}{|X|}$
return P

Algorithm 6 Enumerating Triangles

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

Algorithm 7 Shortest Paths using G-SCIS summary

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E}), u, v \in V$
if $S(u) = S(v)$ **then**
    if $S(u)$ is a clique **then**
        $d(u, v) = 1$
    else
        if $N(S(u) > 0$ **then**
            $d(u, v) = 2$
        else
            $d(u, v) = \infty$
else
    $d(u, v) = d(S(u), S(v))$
return $d(u, v)$

2.4 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 2.4.1) 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.

2.4.1 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 $\langle u, v \rangle$

$$WJ(u,v) = \frac{\sum_{x \in N(u) \cap N(v)} \min(w(u,x), w(v,x))}{\sum_{x \in N(u) \cup N(v)} \max(w(u,x), w(v,x))} \tag{2.4}$$

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 2.4.1, 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 2.4.1 (**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 $V \setminus S(u)^{i-1}$. 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 [31]. 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 Jac-

[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.

card similarity for all the possible two-hop away nodes is an overhead, we deploy a locality sensitive hashing scheme presented in [53] 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. Theorem 2.4.1 holds the same. For ease of notation we continue to call by *WJ* this LSH-based version of weighted Jaccard similarity.

2.4.2 Merging Candidate Pairs

Based on Theorem 2.4.1, 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 2.4.2 (**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 $1 \leq t \leq |H|$. Then $u(\mathcal{G}^{t-1}) \geq u(\mathcal{G}^t)$.*

Theorems 2.4.1 and 2.4.2 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 $u(\mathcal{G}^t) \geq \tau$ (see Algorithm 8). This requires computing H (Algorithm 9) followed by $\lg(|H|)$ computations of utility. The latter is done using Algorithm 10.

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 9). This function follows the structure of Prim's algorithm [46] 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 8). 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

$\geq \tau$, 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 8 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** $low \leq high$ **do**
5: $\quad mid \leftarrow \frac{low+high}{2}$, $\mathcal{V} \leftarrow V$, $i \leftarrow 0$
6: $\quad$ **for** $i \leq mid$ **do**
7: $\quad\quad \langle u, v \rangle \leftarrow H[i]$, $i \leftarrow i + 1$
8: $\quad\quad S \leftarrow$ Merge$(S(u), S(v))$
9: $\quad\quad \mathcal{V} \leftarrow (\mathcal{V} \setminus \{S(u), S(v)\}) \cup S$
10: $\quad u(\mathcal{G}) \leftarrow$ ComputeUtility$(\mathcal{V})$
11: $\quad$ **if** $u(\mathcal{G}) \geq \tau$ **then** $high = mid - 1$
12: $\quad$ **else** $low = mid + 1$
13: BuildSuperEdges$(\mathcal{V})$

Algorithm 10 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 10 returns the final utility for $\mathcal{G}$ which is used by Algorithm 8 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 $\text{Sedge}(S_i, S_j) \leq \text{nSedge}(S_i, S_j)$. This task can be completed in $O(|E|)$ time: Line 18 of Algorithm 10 can be replaced by the task of adding superedge between S_i and S_j.

Algorithm 9 Two-hop MST

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

Data structures. We used the union-find algorithm [22] for representing our supernodes. The union operation was used to implement the merge operation in line 8 of Algorithm 8 and the find operation was used to find the corresponding supernode for a specific node in line 8 of Algorithm 8 and line 6 of Algorithm 10. 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 $O(|V|)$.

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 $O(|V|)$. 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.

Algorithm 10 Compute Utility

Input: $G = (V, E), utility \leftarrow 1, \mathcal{V}$ ▷ set of supernodes
for $S_i \in \mathcal{V}$ **do** ▷ for each supernode
  $count \leftarrow \{\}$, $sum \leftarrow \{\}$
  for $u \in S_i$ **do**
    for $v \in N(u)$ **do**
      $S_j \leftarrow S(v)$
      if $(S_i \neq S_j) \vee (S_i = S_j \wedge i < j)$ **then**
        if $count[S_j] \geq 1$ **then**
          $count[S_j] \leftarrow count[S_j] + 1$
          $sum[S_j] \leftarrow sum[S_j] + C(u, v)$
        else
          $count[S_j] \leftarrow 1$, $sum[S_j] \leftarrow C(u, v)$
  for $S_j \in count.keys \wedge i \leq j$ **do**
    $\text{nSedge}(S_i, S_j) \leftarrow sum[S_j]$
    if $S_i \neq S_j$ **then** $\text{Sedge}(S_i, S_j) \leftarrow \frac{|S_i||S_j| - count[S_j]}{\binom{|V|}{2} - |E|}$
    else $\text{Sedge}(S_i, S_j) \leftarrow \frac{\binom{|S_i|}{2} - count[S_j]}{\binom{|V|}{2} - |E|}$
    if $\text{Sedge}(S_i, S_j) \leq \text{nSedge}(S_i, S_j)$ **then**
      $utility \leftarrow utility - \text{Sedge}(S_i, S_j)$
    else $utility \leftarrow utility - \text{nSedge}(S_i, S_j)$
return $utility$

2.4.3 Complexity analysis of T-BUDS

Let us begin by analysing the time complexity of Algorithm 9. As its structure follows that of Prim's algorithm [46], 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 9 is $O((|F| \cdot \Delta_{max} + |V|) \cdot \lg |V|)$.

The total space required by Algorithm 9 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 10. 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 10 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 8. As discussed in Section 5.2, Algorithm 8 will perform $\lg |H|$ iterations. For each iteration, merging supernodes in Algorithm 8 requires $O(|H|)$ operations and the utility estimation using Algorithm 10 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 8 is storing H and $\mathcal{V}$, which is $O(|V|)$. Thus, the space requirement of Algorithm 8 is $O(|V|)$. Summarizing all the above, we have

Theorem 2.4.3. *The time complexity of T-BUDS is* $O(((|F| \cdot \Delta_{max} + |V|) \cdot \lg |V|))$. *The space complexity of T-BUDS is* $O(|V|)$.

2.5 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 [32, 51, 28] in terms of running time and node reduction.[3]

[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 2.1: Summary of datasets

2. Performance analysis of G-SCIS on Pagerank computation and triangle enumeration.

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://github.com/MahdiHajiabadi/GSCIS_TBUDS`) 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 [32] 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 (`https://law.di.unimi.it/datasets.php`) varying from moderate size to very large (Table 2.1).

2.5.1 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 [32] 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.

Comparison of G-SCIS to SWeG and MoSSo

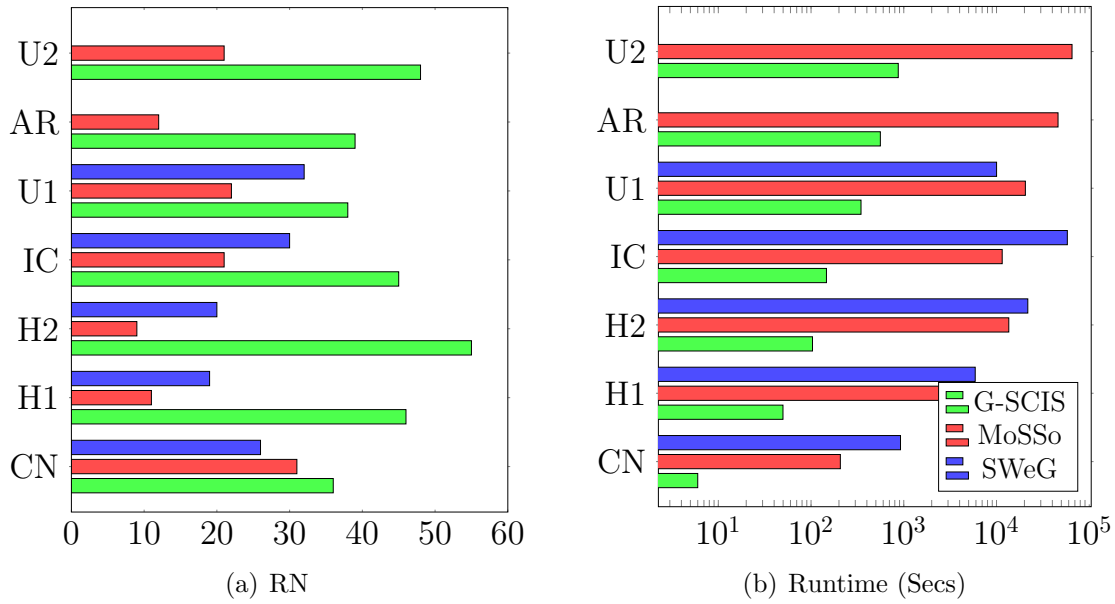
The reduction in nodes (RN) is defined as $RN = (|V| - |\mathcal{V}|)/|V|$ (c.f. [32]). Since SWeG and MoSSo produce also correction graphs to add/delete (C^+, C^-), RN for them is more precisely computed as
$RN = (|V| - (|\mathcal{V}| \cup |V_{C^+}| \cup |V_{C^-}|))/|V|$. We ran SWeG for different choices of the number of iterations up to 80 and chose the best RN value obtained. We use the same configuration for MoSSo as in [28], 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 2.3 shows the comparison between G-SCIS, SWeG, and MoSSo in terms of RN and running time.[4] As the figure shows, 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.

Pagerank computation and triangle enumeration

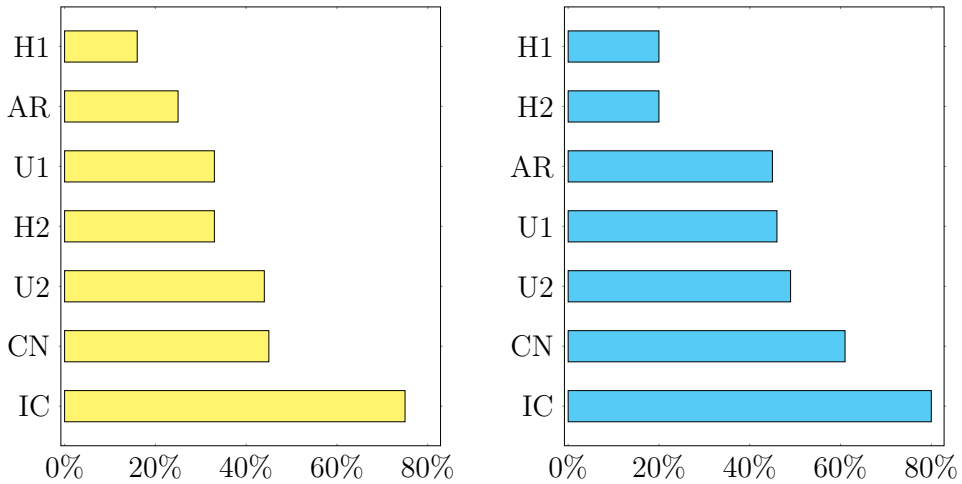
In Figure 2.4, we show the reduction in runtime for Pagerank computation and triangle enumeration using G-SCIS summaries as described in Section 2.3.2 versus the runtime using SWeG summaries with neighbor (adjacency list) queries ([51]). We see a significant reduction in time for both queries for all datasets, reaching up to 80% for IC. We omit results for shortest paths due to space constraints. We observe in Figure 2.4 (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.

[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) RN (b) Runtime (Secs)

Figure 2.3: 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.



(a) PageRank (b) Triangle

Figure 2.4: Runtime improvement of Pagerank and triangle enumeration when running directly on G-SCIS summary vs running on SWeG summary using neighbor queries.

2.5.2 Lossy Case: T-BUDS

Performance of T-BUDS

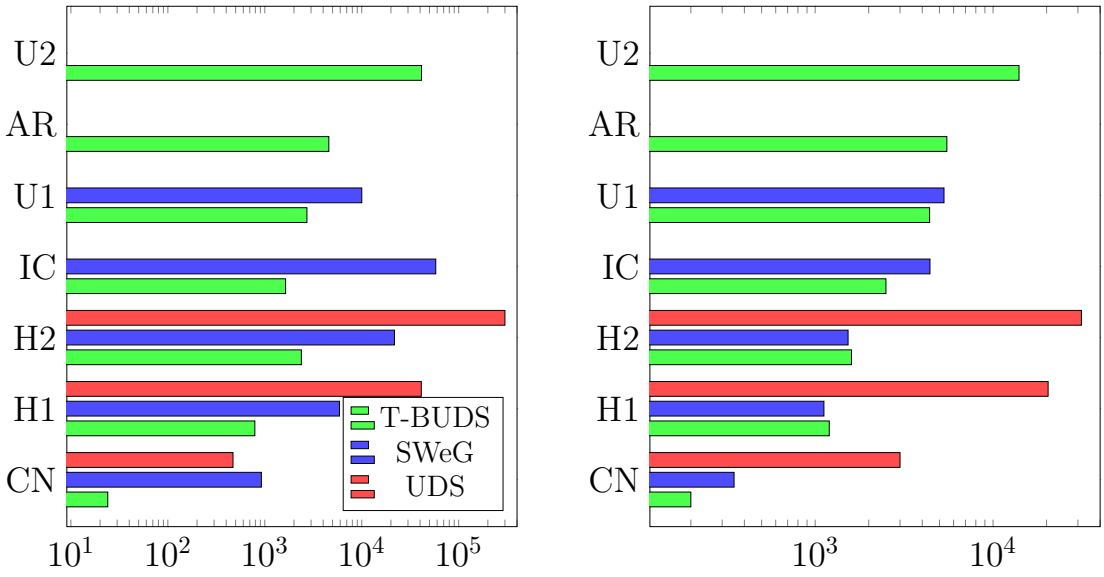
In this section, the performance of T-BUDS is compared to the performance of UDS and SWeG in terms of running time and memory usage (Figure 2.5). For our comparison, we set the utility threshold at 0.8. In order to extend SWeG to support edge weights, we subtract the weight of each removed edge from the current utility value in the dropping step of [51]. Dropping continues until the value of utility drops below the threshold.

We observe that UDS ([32]) 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 [32], 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 2.5, 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.

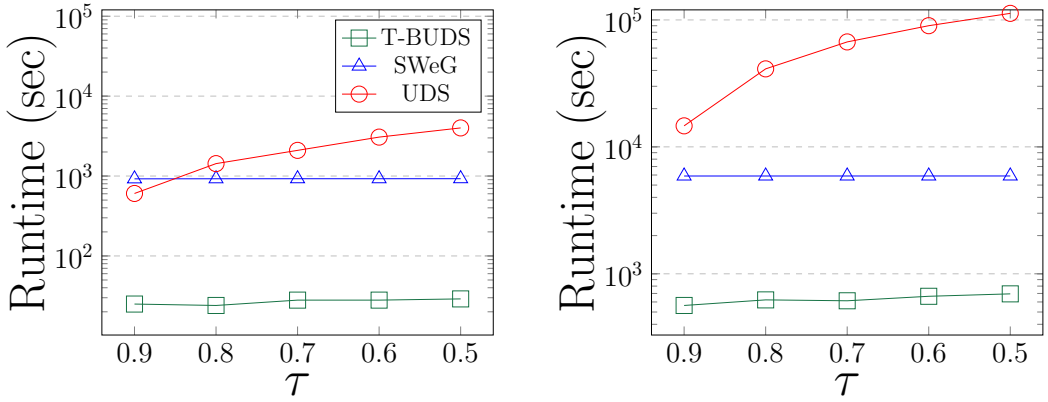
In another experiment we compare the performance of T-BUDS, UDS and SWeG for varying utility thresholds. Figure 2.6 shows the runtime of the three algorithms on two different graphs CN and H1 in terms of varying utility threshold. Having an algorithm that is computationally insensitive to changing the threshold is desirable 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 2.7, 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). Similar behaviour



(a) Runtime (Secs) (b) Memory usage (MB)

Figure 2.5: 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.



(a) CN (b) H1

Figure 2.6: 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.

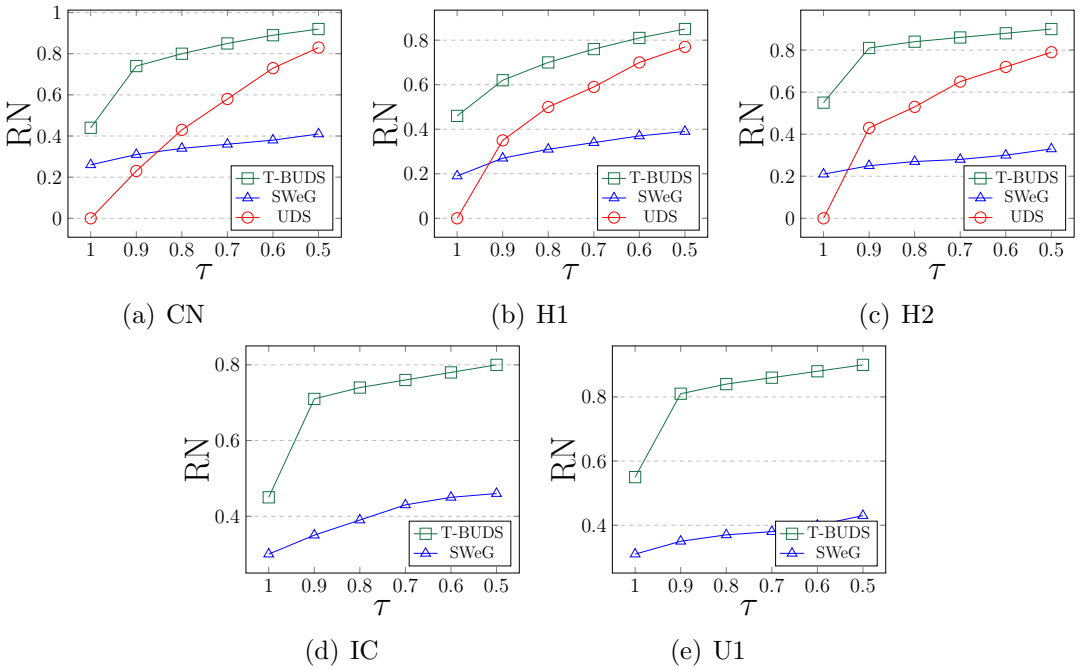


Figure 2.7: RN values for different utility thresholds. For each threshold, T-BUDS gives more compression than SWeG and UDS.

is seen for other datasets. 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. As already mentioned, for medium and big datasets UDS cannot complete within 100 hours. Hence there are no results for UDS for IC and U1.

Accuracy analysis of top-k query on reconstructed graph

In this section, 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 [32] and SWeG [51]. 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 2.2 shows the top-k% match performance of T-BUDS with varying RN on

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

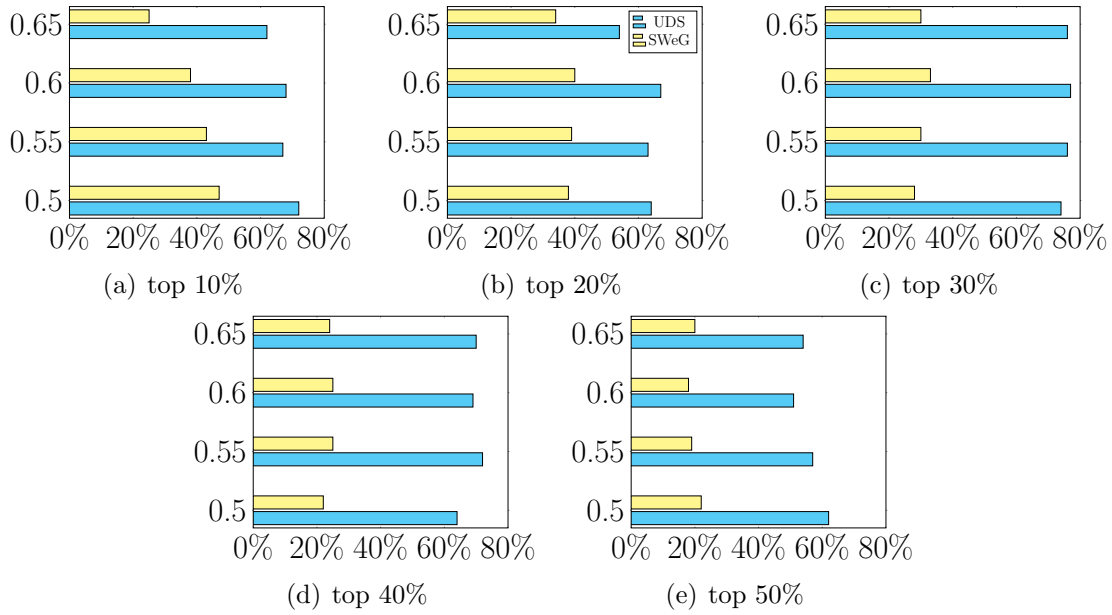
Table 2.2: Match of top-k% PageRank query on reconstructed graph from T-BUDS summary. RN is the node reduction level we consider. Each of the following columns represent a level of k in the top-k% query. The numbers in these columns shows the ratio of the top-k% nodes that appear in both the original and reconstructed graphs. 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 [51]) which results in the same value for RN.

Figure 2.8 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 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 2.8. Its relative improvement over UDS and SWeG is impressive; mostly above 60% and 20%, respectively.

2.6 Conclusions

In this chapter, we studied utility-driven graph summarization in-depth and made several novel contributions. We presented a new, lossless graph summarizer, G-SCIS, that can output the optimal summary, with the smallest number of supernodes, with-



(a) top 10% (b) top 20% (c) top 30%

(d) top 40% (e) top 50%

Figure 2.8: 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.

out using correction graphs as in previous approaches. We showed the versatility of the G-SCIS summary using popular queries such as enumerating triangles, estimating Pagerank and computing shortest paths.

We designed 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. We demonstrated the effectiveness of T-BUDS for answering top-k% Pagerank queries and showed significant improvement on the quality of results compared to state-of-the-art.

Chapter 3

Dynamic Graph Summarization: Optimal and Scalable

This chapter starts with exploring state-of-the-art algorithms in summarizing fully dynamic graphs. While many state-of-the-art algorithms in the dynamic setting are for lossy summarization, we propose two different algorithms for losslessly summarizing a dynamic graph. Our first algorithm, Optimal, produces the smallest-possible-anytime summary and the second algorithm, Scalable, 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. Finally, we conduct extensive experiments to show that both algorithms outperform competitors [28] with an order of magnitude running time improvement and 6x improvement in reduction in nodes.

3.1 Related Work

Graph summarization has been studied for more than a decade [40, 25]. The algorithms are generally classified as grouping and non-grouping methods and our work belongs in the former category.

Lossless graph summarization has been studied in the utility-based framework [32, 19] in which the objective is to minimize the number of supernodes in the summary while preserving the utility of the summary. In the lossless setting, all the actual edges should be possible to be recovered and no spurious edge should be added [32, 19]. In the correction set based framework for lossless graph summarization [59, 51, 44, 27], two sets of side information $(C+, C-)$ are stored along with the summary graph and

they are used for correction during reconstruction.

Recently, incremental graph summarization algorithms have received attention [18, 28, 50, 57, 16, 6, 55, 24, 61, 10]. Shah et al. [50] used the minimum description length (MDL) technique to minimize the number of required bits for describing the graph. Fernandes et al. [16] proposed a memory-less dynamic lossy graph summarization algorithm where in each step the summary is recomputed from the scratch. All of these algorithms, with the exception of [28], are lossy. Ko et al. in [28] proposed the first incremental lossless algorithm, MoSSo, in the correction set framework. MoSSo is able to update the summary in near-constant time. [41] 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. Our work is the first to present dynamic lossless summarization in the utility based framework.

3.2 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 [19] 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

$$\text{minimize } |\mathcal{V}_t| \quad \text{subject to } \hat{G}_t = G_t. \tag{3.1}$$

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 [19].

Proposition 3.2.1. *(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 11 below from [19] 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 11 Finding the best summary

```
1: Input: G = (V, E)
2: Initialization: Status[∀v ∈ V] ← False, S ← []
3: for u ∈ V ∧ Status[u] = False do
4:     S(u) ← {u}, Status[u] ← True
5:     for v ∈ V ∧ Status[v] = False do
6:         if (N(u) = N(v)) ∨ (N+(u) = N+(v)) then
7:             S(u) ← S(u) ∪ {v}, Status[v] ← True
8:     S.add(S(u))
9: BuildSuperEdges(S)
```

Algorithm 11 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 3.2.1, the two 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 $(u, v) \in E$.

3.3 Optimal Lossless Algorithm

Algorithm 11 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* when there is 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 3.2, 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.

$$\begin{aligned}
I_k[u] &= I_k[u] + v^k \mod P && \langle u, v \rangle \text{ insertion} \\
C_k[u] &= C_k[u] + v^k \mod P && \langle u, v \rangle \text{ insertion} \\
I_k[u] &= I_k[u] - v^k \mod P && \langle u, v \rangle \text{ deletion} \\
C_k[u] &= C_k[u] - v^k \mod P && \langle u, v \rangle \text{ deletion}
\end{aligned} \tag{3.2}$$

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 12 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 (3.2).

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 the type of supernode) which we call it the representative of that supernode. We remark here that the structures HI and HC are similar to *mapI* and *mapC* used in Chapter 2 because both of them store the candidate independent set and clique supernodes, however they are different because they use different hashing schemes.

After each change ($\langle u, v\rangle$ insertion or deletion) the algorithm calls function *FindNodes* (Algorithm 13) 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 3.2.1). 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 12) we update the representation of its supernode S_a in HI and HC as described in Algorithm 14.

In line 6 of Algorithm 12, for each $a \in \{u, v\}$, we update $I[a]$ and $C[a]$ based on their current values and the recent change using Equation 3.2. 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 12). Otherwise, a similar process is performed for $C[a]$ and HC (lines 12-16). If neither of the lookups succeed, we call Algorithm 15 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 12 Optimal Algorithm (Optimal)

```
1:  Input: e = ⟨u, v⟩, +/−
2:  nodes = findNodes(u, v)
3:  for a ∈ nodes do
4:      updateH(a)
5:      if a == u ∨ a == v then
6:          I[a], C[a] ← incremental(a, I[a], C[a])          ▷ Eq. (3.2)
7:      if (I[a] ∈ HI) then
8:          for a′ ∈ HI(I[a]) do
9:              if N(a′) == N(a) then
10:                 merge(a, a′)
11:                 continue
12:     if C[a] ∈ HC then
13:         for a′ ∈ HC(C[a]) do
14:             if N⁺(a′) == N⁺(a) then
15:                 merge(a, a′))
16:                 continue
17:     singleton(I[a], C[a], a)
```

Algorithm 13 findNodes(u, v)

```
1: r ← {u, v}
2: if S_u exists and |S_u| == 2 then
3:     Let _u be the other node in S_u
4:     Add _u to r
5: if S_v exists and |S_v| == 2 then
6:     Let _v be the other node in S_v
7:     Add _v to r
8: Return r
```

Figure 3.1 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 3.1 shows detailed values of our data structures. Equation 3.2 with $k = 1$ is used for incrementally

Algorithm 14 updateH(a)

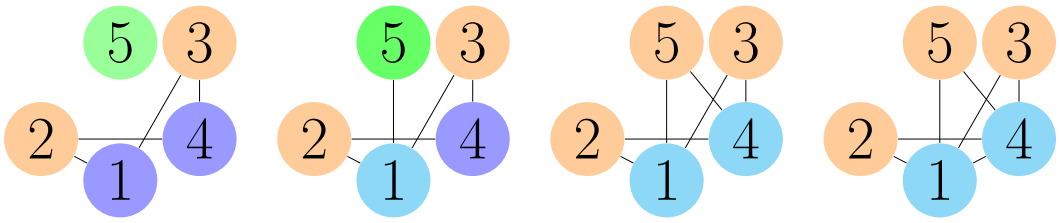
1: **if** a serves as a representative of S_a in HI **then**
2: replace a by some other node $b \in S_a$ in HI.
3: **if** a serves as a representative S_a in HC **then**
4: replace a by some other node $b \in S_u$ in HC.

Algorithm 15 singleton($hI[u], hC[u], u$)

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

Algorithm 16 merge(u, _u)

1: $n[u] \leftarrow \_u, p[u] \leftarrow p[\_u]$
2: $n[p[\_u]] \leftarrow u, p[\_u] \leftarrow u$



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

Figure 3.1: 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 3.1: Values of $I[u]$, $C[u]$, HI and HC after each change.

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 12 computes the lossless summary of G_t with minimum number of supernodes. We begin with the following observation.

Proposition 3.3.1. *The hash function in Eq. 3.2 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 3.2.

In Algorithm 12, each supernode has a representative in the data structures HI or HC depending on its type.

Proposition 3.3.2. *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 12 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 the following property

Proposition 3.3.3. *At the end of step t, Algorithm 12 always computes a lossless summary of G_t.*

We now state and prove our main result.

Theorem 3.3.4. *At the end of step t, Algorithm 12 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 12 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 12 after processing this step t is more than the number of supernodes in the optimal summary. According to Proposition 3.3.3, Algorithm 12 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 12. 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 12 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 3.3.2, supernode s has one representative node u in HI or HC (in both if it is of size 1) which enables Algorithm 12 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 12.

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 12 will add both u' and v' to s as they will hash to the same value by Proposition 3.3.1. 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 12 and hence will be in the same supernode.

Therefore, in all cases the summary output by Algorithm 12 is the same as the optimal summary. □

Complexity analysis. Algorithms 13, 14, 15, and 16 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 12 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).

3.4 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 12.

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, $t \geq 0$,

$$\text{minimize } |\mathcal{V}_K^t| \quad \text{subject to } \hat{G}_t = G_t. \tag{3.3}$$

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 3.2. $I_k(u)$ 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 3.2. 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 17 shows the main steps. As in Algorithm 12, 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 17 is perfect (proven by Theorem 3.4.1), each entry in HI or HC has exactly one value.

For each change $\langle u, v\rangle$, as in Algorithm 12, it first calls *findNodes* function (Algorithm 13) 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 17). Otherwise, it calls *updateH* function (Algorithm 14) to update HI and HC. Algorithm 14 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 18 is called to update the tuple of hash values for node a. *Incremental* function updates $I(u)$ and $C(u)$ based on the Equation 3.2. The last element in $I(u)$ or $C(u)$ is incremented or decremented depending on insertion or deletion.

Line 10 of Algorithm 17 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 16 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 17 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 ($n[a] = a, p[a] = a$).

Algorithm 17 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. 3.2
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 18 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 3.2
4: return $C(u), I(u)$

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 3.4 describes the false positive error of the proposed hashing scheme while checking if $N(u) = N(v)$ or not. We will show that if $K \geq d$, (3.4) cannot be satisfied.

$$\begin{aligned}
\sum_{u_i \in N(u)} u_i \mod P &= \sum_{v_i \in N(v)} v_i \mod P \\
\sum_{u_i \in N(u)} u_i^2 \mod P &= \sum_{v_i \in N(v)} v_i^2 \mod P \\
&\vdots \\
\sum_{u_i \in N(u)} u_i^K \mod P &= \sum_{v_i \in N(v)} v_i^K \mod P
\end{aligned} \tag{3.4}$$

Theorem 3.4.1. *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 (3.4) 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 3.4 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.

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. (3.4) holds, then the d elementary symmetric polynomials are all equal modulo P.

Lemma 3.4.2. *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 (3.4). 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 3.4.3. *For all $d \geq 1$ and $d \geq k \geq 1$,*

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

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 3.4.4. *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:

$$\prod_{i=1}^{d}(x + u_i) = x^d + e_1(u_1, u_2, \ldots, u_d)x^{d-1} + \cdots + e_d(u_1, u_2, \ldots, u_d)$$

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 3.4.1. □

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 3.4.5. *At the end of step* t, *Algorithm 17 computes a lossless summary that is* optimal *for nodes of degree at most* K.

Complexity analysis. As in Algorithm 12, *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.

3.4.1 Summarizing Directed Graphs

In this section, we extend the proposed scalable algorithm to summarize dynamic directed graphs. In a directed graph, each node has a set of in-neighbors and out-neighbors. So, we begin with the definition of independent sets and cliques in this setting.

Definition 3.4.6. *(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 3.2 for updating hash values. Algorithm 19 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 is any match or not. Finally it puts that node into a singleton supernode and adds both I and C to HI and HC.

3.5 Experiments

Implementation: We implemented the proposed algorithms in Java 14 on a machine

Algorithm 19 Directed-Scalable Algorithm

Input: $\langle \overrightarrow{u,v} \rangle, +/-, K$
▷ Updating out neighbors hashes for u and in-neighbors for v
for $a \in \{u, v\}$ **do**
  if $I(a)(K+2) \vee I(a)(K/2+1) \geq K/2$ **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 $C(a)(K/2+2)$
  $I(a), C(a) \leftarrow Incremental(I(a), C(a), u, v)$ ▷ Eq. 3.2
  if $I(a) \in HI$ **then**
    $a' \leftarrow HI(I(a))$
    merge(a, a')
    continue
  if $C(a) \in HC$ **then**
    $a' \leftarrow HC(C(a))$
    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 3.2: Summary of datasets

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 3.2 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 [19, 59] and dynamic [28]. Source codes of all state-of-the-art lossless algorithms are publicly available and they were all implemented in Java. G-SCIS [19] does not require any input parameters. For LDME [59] 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 [28]. We use MoSSo-MoSSo (MoSSo) as it is orders of magnitude faster than the other variants and we use the same configuration as in [28], $e = 0.3$ and $c = 120$, where e is the escape probability and c is the sample size of each trial (see [28]). SWeG [**?**] 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 [59], which we use in our experiments.

Evaluation: Reduction in nodes (RN) [19, 32] 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 $RN = (|V| - (|\mathcal{V} \cup V(C^+) \cup V(C^-)|))/|V|$.

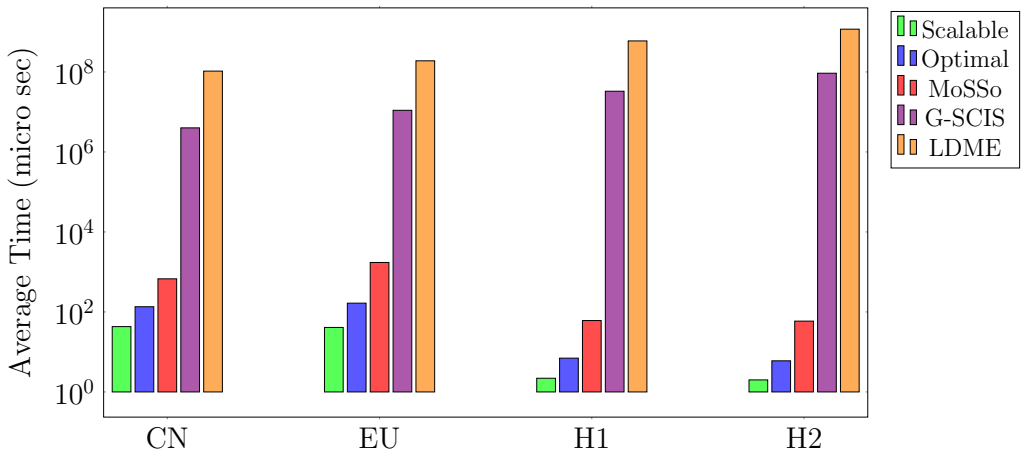


Figure 3.2: Average processing time per edge for Optimal and Scalable vs. MoSSo, G-SCIS, LDME. Scalable is up to 8 and 7 orders of magnitude faster than LDME and G-SCIS, respectively, and around 30 times faster than MoSSo.

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 3.2 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 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 3.3 and 3.5. Optimal is faster than MoSSo and provides far better reduction in nodes. Scalable is up to

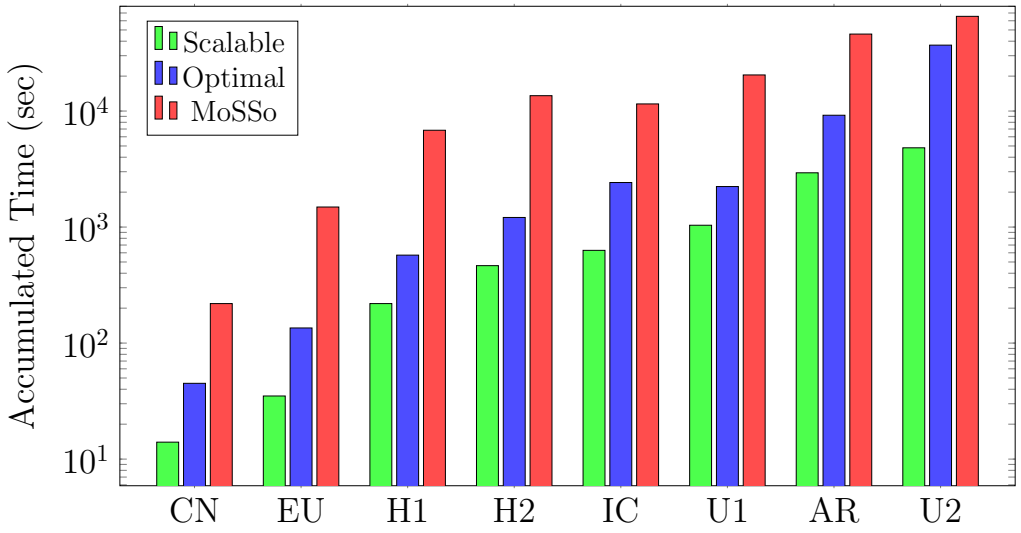
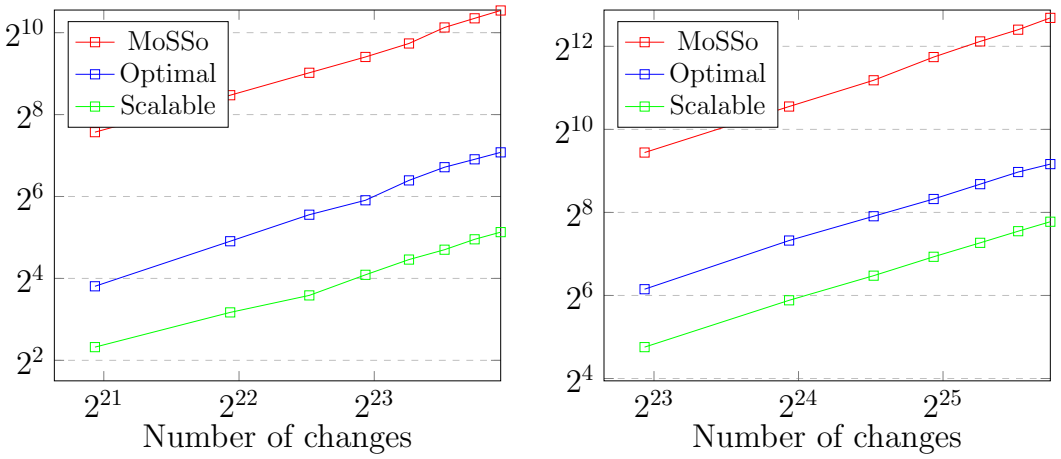


Figure 3.3: Accumulated running time for Scalable, Optimal, and MoSSo after E changes to each graph. Scalable is up to 40x faster than MoSSo and also up to 10x faster than the optimal algorithm (see for instance U2 and CN).

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 3.4. 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 3.5, 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 3.6 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 3.7. The details of Directed-Scalable are given in the Supplemental Material. We can see in Figure 3.7, for instance, that Directed-Scalable is able to 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.



(a) EU (b) H1

Figure 3.4: Accumulated running time in seconds (vertical axis) for Scalable vs Optimal vs MoSSo with respect to number of total changes (horizontal axis). (a) Time measured for EU every 2M changes, (b) Time measured H1 every 8M changes. The charts show that the scalability of all three algorithms is quite linear in $|E|$, with Optimal and Scalable being significantly better than MoSSo.

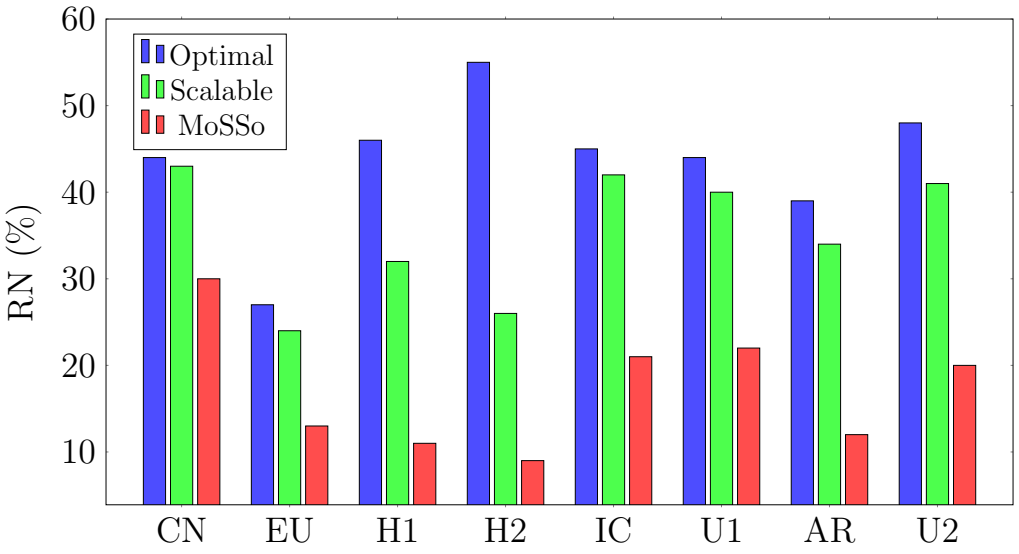
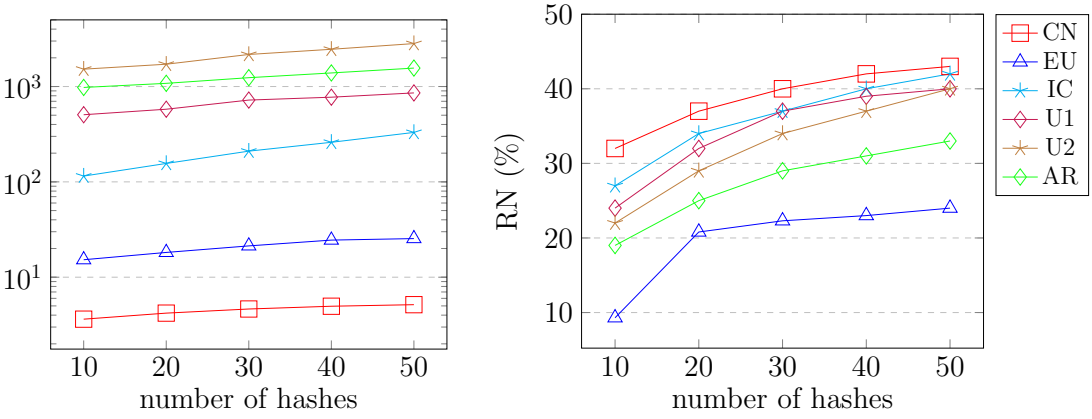
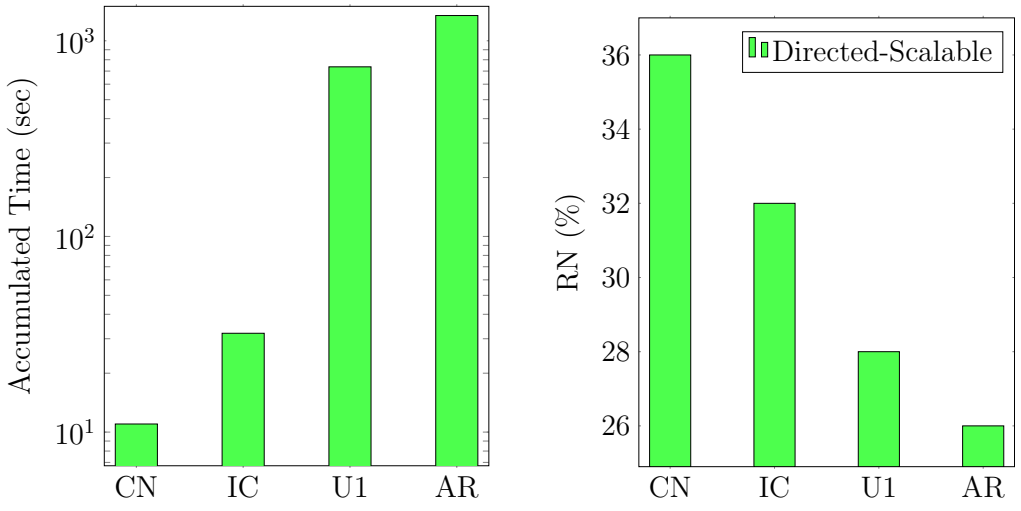


Figure 3.5: Reduction in nodes for Optimal, Scalable, and MoSSo. Both Optimal and Scalable outperform MoSSo. For CN, EU, IC, U1, AR, U2 Scalable is quite close to Optimal.



(a) running time (b) RN

Figure 3.6: Effects of number of hashes on the performance of Scalable. Running time in (a) is in seconds.



(a) running time (b) RN

Figure 3.7: Performance of Directed-Scalable on different directed graphs in a fully dynamic scenario.

3.6 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. More concretely, we follow the framework of G-SCIS [19] which produces summaries that can used as-is in several graph analytics tasks. Different from G-SCIS, which 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, but 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. An interesting fact that we observed experimentally is that even if we were to run a batch algorithm, such as G-SCIS, once for every big batch of changes, still they would be much slower than Scalable. For instance, if 1 million changes occur in a graph, Scalable is two orders of magnitude faster than running G-SCIS just once at the end of the 1 million-edge sequence.

Chapter 4

Efficient Graph Summarization using Weighted LSH at Billion-Scale

In this chapter we explore the correction-set based framework for graph summarization. We start by reviewing related work and exploring a recent state-of-the-art algorithm, SWeG [51], based on this framework. We then propose our algorithm LDME and show that LDME outperforms SWeG by conducting thorough experiments.

4.1 Related Work

Graph summarization is an active area of research and studied in a variety of settings (see [40, 25] for detailed surveys). Previous work on this topic can be classified into two categories, grouping [32, 34, 15, 44, 51, 47, 56, 27, 33] and non-grouping [60, 36, 42, 23, 35, 43, 5, 58, 38, 18, 55]. Grouping based methods received more attention in the last few years and they are divided into non-correction set based methods such as [33, 34, 47, 32, 29, 15, 18, 56] and correction set based methods [44, 51, 28, 27].

Navlakha et al. [44] introduced a novel framework in which a graph is represented compactly as a summary graph along with correction sets. Their algorithm, RANDOMIZED, picks a random supernode and identifies the supernode that gives the best savings with it among possible candidate merges from supernodes that are 2-hops away. SAGS [26] uses simple locality sensitive hashing instead of Saving or

SuperJaccard in the merge phase to choose the best pair among pairs that are 2-hop away. VoG [29] uses existing clustering algorithms for finding important candidate subgraphs to summarize. These works, however, are unable to achieve strong compression while maintaining scalability.

The state of the art algorithm for correction set based graph summarization is SWeG studied by Shin et al. [51] which achieves strong compression and scales an order of magnitude better than RANDOMIZED and SAGS. MoSSo is a recent incremental algorithm for summarizing dynamic graphs using correction set [28] which we also include in our comparisons. SlimGraph [9] is a programming framework, where various summarization algorithms can be plugged-in (such as SWeG), rather than an algorithmic contribution, so there is no direct avenue to compare it with our work.

4.2 Correction Set Based Graph Summarization

Here we describe the framework of correction set based graph summarization (CGS). In this framework, we are given a simple undirected input graph $G = (V, E)$, and the output consists of a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and corrections sets C^+ and C^- which contain edges to be inserted and deleted respectively. The goal is to reconstruct G using the summary graph and correction sets. We denote the reconstructed graph by $\hat{G} = (V, \hat{E})$. If $\hat{G} = G$, we call the summarization lossless; otherwise, the summarization is lossy. We denote the set of neighbours of node v in G ($\hat{G}$) by N_v ($\hat{N}_v$).

Problem Definition. We begin by formally describing the method used to obtain the reconstructed graph $\hat{G}$ from the output of CGS, namely the summary graph and the correction set. Given a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-, we build $\hat{G} = (V, \hat{E})$ as follows:

1. For each superedge $(A, B) \in \mathcal{P}$, add all pairs of nodes (a, b) as edges to $\hat{E}$ where $a \in A$ and $b \in B$.
2. Add each edge in C^+ to $\hat{E}$
3. Remove each edge in C^- from $\hat{E}$

The *graph summarization problem* (an optimization problem) that CGS algorithms aim to solve is defined below.

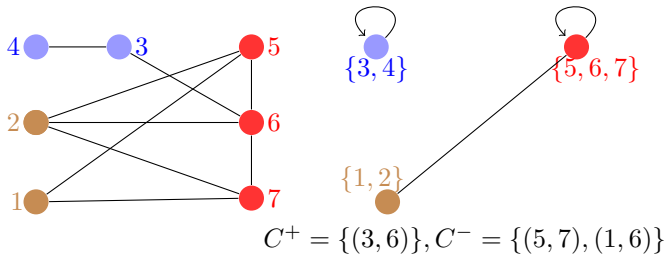


Figure 4.1: The scheme of correction set based graph summarization

Graph Summarization Problem

Input: Graph $G = (V, E)$

Output: Summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-

Minimizing

$$|\mathcal{P}| + |C^+| + |C^-| \tag{4.1}$$

Such that the restored graph $\hat{G} = (V, \hat{E})$ satisfies the constraint

$$|N_v \setminus \hat{N}_v| + |\hat{N}_v \setminus N_v| \leq \epsilon |N_v|, \ \forall v \in V \tag{4.2}$$

The objective function (Eq. (4.1)), which we want to minimize, is the sum of the number of superedges in the summary graph and the number of edges in the correction sets. In our implementation, we only count the non-loop superedges since self loops can be encoded using a single bit, so their total size is negligible. Note that the constraint in Eq. (4.2) applies only to lossy summarization which is orthogonal to the contributions in this work. Figure 4.1 shows graph summarization using correction sets. It losslessly summarizes the input graph with 7 nodes and 9 edges to the summary with 3 supernodes, 3 superedges and 3 correction edges (1 insertion and 2 deletions). SWeG [51] is the current state of the art of CGS for large-scale graphs that builds on the algorithm by Navlahka et al. [44] and provides approximate solutions to the graph summarization problem. SWeG contains several optimization techniques over [44], namely an efficient approximation metric for determining merge pairs and a dividing step to speed up the algorithm and parallelize the code for a distributed computing environment. The goal of our work is to introduce new techniques to summarize large scale graphs faster than SWeG by significantly reducing the amount of computation, while maintaining comparable output compactness.

Algorithmic Framework. To better describe our algorithm, we will first outline the CGS approach of [51, 44].

At a high level, the input to the algorithm is a graph $G = (V, E)$, number of

iterations T, and an error bound ϵ; the output is a summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and correction sets C^+, C^-. Firstly, the set of supernodes $\mathcal{S}$ is initialized such that every supernode contains exactly one vertex of V and every vertex of V is in exactly one supernode. Then, $\mathcal{S}$ is repeatedly updated over T iterations by performing sequences of supernode merges per iteration. In the case of SWeG, in each iteration, $\mathcal{S}$ is divided into disjoint groups prior to the merging step and merges are performed within each group.

Once the supernodes are identified, the original edges E are encoded into superedges $\mathcal{P}$ and correction sets C^+ and C^-. Finally, in the case of lossy summarization (if $\epsilon > 0$), some superedges/edges are dropped from $\mathcal{P}$, C^+, and C^- while maintaining the constraint in Eq. (4.2). The individual steps in the algorithm will now be described in further detail.

Dividing step. The dividing step divides $\mathcal{S}$ into disjoint groups of supernodes such that supernodes which are similarly connected are placed into the same group. This step is an optimization technique introduced by [51] for the purpose of speed, memory efficiency, and parallelizability. The division is performed by grouping supernodes based on their *shingle*. The *shingle* $f(v)$ of a regular node $v \in V$ is defined as $f(v) := min_{u \in N_v \text{ or } u=v}\ h(u)$ where h is a random bijective function $h : V \to \{1, \ldots, |V|\}$. The shingle $F(A)$ of a supernode $A \in \mathcal{S}$ is defined as as $F(A) := min_{v \in A} f(v)$. $\mathcal{S}$ is then divided into disjoint groups $\{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(m)}\}$ where supernodes in each group have the same shingle value.

Merging supernodes. Let $\mathcal{S}$ be the entire set of supernodes. In the case of [51], set $\mathcal{S}$ to be each $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(m)}\}$ from the dividing step and perform the following for each group. (In a parallel implementation, each group is processed in parallel).

The merging step merges supernodes in $\mathcal{S}$ by selecting a random node A from $\mathcal{S}$, determining A's best merge candidate supernode B (in terms of minimizing Eq. (4.1)) from $\mathcal{S}$, then merging A and B if the result of the merge reduces Eq. (4.1) by a sufficient amount. Formally, this is done as follows:

- Initialize temporary set *temp* to $\mathcal{S}$
- While *temp* is not empty:
  - Randomly remove a supernode A from *temp*
  - Find the best merge candidate B for A from *temp*

– If the *savings* (defined below) obtained from merging A and B is above some threshold, merge A and B then replace B in *temp* with the merged result

To define *savings*, we need the notion of *cost* described as follows. The cost of each supernode $A \in \mathcal{S}$ is denoted by $Cost(A, \mathcal{S})$ and is defined to be the number of superedges in $\mathcal{P}$ and edges in C^+ and C^- that A contributes to Eq. (4.1). This is computed by performing a temporary edge encoding step (described later) relative to A on the current state of $\mathcal{S}$. In particular, to compute $Cost(A, \mathcal{S})$ for some supernode A, we would calculate the number of edges between A and every adjacent supernode, then use this to calculate how many superedges and correction set edges would be encoded. Similarly, $Cost(A \cup B)$ is computed by looking at edges between merged supernode $A \cup B$ and all adjacent supernodes.

The *savings* obtained by merging two supernodes $A \neq B \in \mathcal{S}$, denoted $Saving(A, B, \mathcal{S})$, describes the "benefit" of merging A and B by calculating the inverse of the ratio between the cost of merged supernode $A \cup B$ and the sum of A and B's separate cost. Savings is formally defined as

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

[51] claims that computing *Saving* is computationally expensive, and uses an approximation metric known as *SuperJaccard* similarity to approximate *Saving*. The *SuperJaccard* similarity between two supernodes is defined as

$$SuperJaccard(A, B) = \frac{\sum_{v \in N_A \cup N_B} min(w(A, v), w(B, v))}{\sum_{v \in N_A \cup N_B} max(w(A, v), w(B, v))} \tag{4.3}$$

where N_A is the set of nodes adjacent to the nodes inside supernode $A \in \mathcal{S}$ and $w(A, v)$ is the number of nodes in supernode $A \in \mathcal{S}$ adjacent to node $v \in V$, formally defined as $w(A, v) := |\{u \in A : \{u, v\} \in E\}|$. *SuperJaccard* aims to measure the similarity between two supernodes based on the similarity of their connectivity and is used to approximate the best merge candidate. After the best merge candidate B is identified using *SuperJaccard*, then $Saving(A, B, \mathcal{S})$ is computed once to decide whether or not to merge. Formally, if $Saving(A, B, \mathcal{S}) \geq$ *merging threshold* $\theta(t)$, for iteration $1 \leq t < T$, then A and B are merged. Here, $\theta(t)$ is defined as $1/(1 + t)$ so that more merge opportunity is allowed in the later iterations.

Encoding edges. The encoding step takes the supernodes $\mathcal{S}$ from the merging step and encodes the edges E of the original graph into superedges $\mathcal{P}$ and corrections C^+, C^-. This is done by iterating over all pairs of supernodes (A, B) where the set of edges in E between the nodes in A and nodes in B is not empty. Then we either (1) choose to not encode a superedge between A and B or (2) choose to encode a superedge between A and B. In case (1), since we do not introduce a superedge, we would lose all the edges between A and B in the reconstruction step, so we must add all these edges to C^+. In case (2), we add a superedge (A, B) to $\mathcal{P}$ and as a result we could potentially introduce extraneous edges that were not in the original graph; we add those edges to C^-. For each supernode pair $A, B \in \mathcal{S}$, E_{AB} is the set of edges in E that are between the nodes inside supernodes A and B, and F_{AB} is the set edges between all pairs of nodes in A and B. Formally,

$$E_{AB} := \{\{u, v\} \subset V : u \in A, v \in B, u, v \in E\}$$
$$F_{AB} := \{\{u, v\} \subset V : u \in A, v \in B\}$$

For each pair of supernodes $A, B \in \mathcal{S}$ where $E_{AB} \neq \emptyset$, if $|E_{AB}| \leq \frac{|F_{AB}|}{2} = \frac{|A| \cdot |B|}{2}$ then we do not encode E_{AB} as a superedge, so E_{AB} is merged into C^+. Otherwise, E_{AB} is encoded as a superedge, so edge (A, B) is added to superedges $\mathcal{P}$ and $(F_{AB} \setminus E_{AB})$ is merged into C^-. In the special case where $A = B$, the condition is such that we do not encode a superedge (superloop) if $|E_{AA}| \leq \frac{|F_{AA}|}{2} = \frac{|A| \cdot (|A|-1)}{4}$. Otherwise, we do encode a superloop.

Dropping edges. In [51, 44] there is also an optional post processing step (when $\epsilon > 0$), called the *dropping step*, where some edges are removed from the summary graph and correction sets so that the summary is more compact. The dropping step ensures that Eq. (4.2) is satisfied by verifying the constraint is still met as each edge is removed. We do not discuss this step further as its running time is negligible and it is orthogonal to the main approach.

4.3 Proposed Method

Since the correction set based summarization method of [51] is the current state of the art, we will discuss our proposed methods of improvement with respect to SWeG. Our goal is to optimize the steps of SWeG by reducing the amount of computation performed.

The merging step is, computationally, the most challenging stage of the algorithm and takes a significant fraction of the total running time, making it the overall bottleneck step for most graphs. This is because the process is quadratic in the size of groups. Here we propose an improved dividing step that reduces the size of groups, thus significantly speeding up the merge phase. Additionally, we propose an efficient algorithm to compute Saving directly during merge eliminating the need to use an approximation metric. For some graphs, the encoding step becomes a problematic step since it iterates through all pairs of supernodes, inducing a large amount of overhead required for computing/remembering the edges between pairs of supernodes. For graphs with a high number of supernodes, the overhead causes the encoding step to run much slower and in some cases not complete within reasonable time. We present a new sort-based encoding step which has an improved practical running time and is more robust than the one in [51]. We show the overall structure of our approach in Algorithm 20. It consists of the divide, merge and encode steps as outlined in Section 4.2.

Algorithm 20 Algorithm Overview

1: **Input:** input graph $G = (V, E)$, number of iterations T
2: **Output:** summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$, corrections C^+ and C^-
3: initialize supernodes $\mathcal{S}$ to each vertex in V
4: **for** $t = 1...T$ **do**
5: compute weighted LSH signature of each supernode in $\mathcal{S}$
6: divide $\mathcal{S}$ into disjoint groups based on their signature
7: perform merges in each disjoint groups
8: encode edges E into superedges $\mathcal{P}$ and correction edges C^+ and C^-
9: **return** summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$ and corrections C^+, C^-

Speeding up the merging step. In the merging step of [51], after picking a random supernode A from the group, the merge partner B for A is determined by checking every other supernode in the group and selecting the best candidate. Overall, the merging step takes time quadratic in the number of supernodes in a group. Conceptually, we can improve the running time of the merging step by reducing the size of each group $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$. We achieve this by introducing a refined divide step which uses *SuperJaccard* similarity for grouping supernodes. Note that SWeG uses *SuperJaccard* as an approximation of Saving in the merge phase. We instead propose to use *SuperJaccard* in the divide step. Namely, we want to create groups such that nodes with high *SuperJaccard* similarity end up in the same group. How-

ever, we will aim to avoid computing *SuperJaccard* for every pair of supernodes as this would be prohibitive. Instead we want to devise a locality sensitive hashing scheme for *SuperJaccard* and then hash the supernodes to the proper groups.

Locality sensitive hashing (LSH) is a technique used to group similar items. A hash function (or hash function family) is used to assign items to buckets, and the items in each bucket are "similar" to each other with high probability. Typically, items are considered to be sets and LSH schemes are designed for well-known set similarity measures, such as simple Jaccard, Hamming, and Cosine similarity. In the following, we show that *SuperJaccard* can be casted as weighted Jaccard similarity, for which there exist locality sensitive hash functions.

The weighted Jaccard similarity $WJ(X, Y)$ between two vectors X and Y of equal length and with integers weights is defined is

$$WJ(X, Y) := \frac{\sum_v min(X_v, Y_v)}{\sum_v max(X_v, Y_v)}$$

Note that when X and Y are Boolean vectors, the above equation gives the simple Jaccard similarity between two sets represented by vectors X and Y.

Weighted LSH based dividing step. We assign a "supervector" V_S of size n to each supernode $S \in \mathcal{S}$, where n is the number of nodes in V. Each index u in V_S (where $1 \leq u \leq n$) represents a node in V, and the weight of V_S at index u for each node u is $w(S, u)$. Recall, from Section 4.2, that $w(S, u)$ is the number of nodes in S adjacent to u.

We claim that for two supernodes A and B, the weighted Jaccard similarity between their supervectors V_A and V_B is equal to the SuperJaccard similarity between A and B. Namely, we note that

$$WJ(V_A, V_B) = \frac{\sum_v min((V_A)_v, (V_B)_v)}{\sum_v max((V_A)_v, (V_B)_v)} = \frac{\sum_v min(w(A, v), w(B, v))}{\sum_v max(w(A, v), w(B, v))}$$

is equal to $SuperJaccard(A, B)$ as defined in Eq. (4.3). For any supernode S, the non-zero values in its assigned supervector V_S correspond exactly to the nodes in N_S. So, in $WJ(V_A, V_B)$, only the indices $v \in N_A \cup N_B$ contribute to the overall value. Therefore, $WJ(V_A, V_B) = SuperJaccard(A, B)$.

Now, we can use a weighted LSH scheme (to be described shortly) in order to split $\mathcal{S}$ in the dividing step and have the property that supernodes with high SuperJaccard similarity have a high probability of being in the same bucket. Compared to the simple

single-shingle based approach of [51], weighted LSH is a more precise metric that divides S into more groups of smaller size while ensuring that supernodes with similar connectivity are in the same group. Furthermore, we can control the performance of our algorithm by tuning the precision level of the hash function in order to trade compression for running time.

Updated Dividing Step. As a weighted LSH scheme, we use Densified One Permutation Hashing (DOPH) [53], which takes as input a binary vector I of length $|V|$ and uses a single permutation $h : \{1, \ldots, |V|\} \rightarrow \{1, \ldots, |V|\}$ to produce a hash signature H_I for I of length k. We use the fact from [52] that for any sparse weighted vectors V_A and V_B, the probability that the binarized forms of V_A and V_B have the same DOPH signature is approximately the weighted Jaccard similarity between V_A and V_B (non binarized).

Given a binary vector I of length $|V|$, a random permutation $h : \{1, \ldots, |V|\} \rightarrow \{1, \ldots, |V|\}$, a hash signature length k, and a random binary vector D of length k (where D_i, $1 \leq i \leq k$, is set to 0 or 1 independently and uniformly at random), DOPH signature H_I is computed as follows (see Algorithm 21):

- Permute the bits in I by re-indexing based on h. (Line 1)
- Separate I into k equal length bins. We denote bin i as b_i, $1 \leq i \leq k$. (Line 2)
- Define H_{b_i} as the first index within the bin b_i that contains a non-zero value. If the bin contains no non-zero values, H_{b_i} is "empty". Set H_{b_i} to be the value of the signature H_I at position i. (Lines 3 - 7)
- For each empty H_{b_i}, we define the value to be the first non-empty signature index either to the left or right (with wraparound at the endpoints). The choice of left or right is determined by the bit D_i. (Lines 8 - 12)
- Return $H_I = \{H_{b_1}, ..., H_{b_k}\}$. (Line 13)

We use DOPH as a new metric for dividing supernodes $\mathcal{S}$ into disjoint groups, where each group of supernodes has the same DOPH signature. For each supernode $A \in \mathcal{S}$, we binarize the supervector V_A (supervector defined in the previous section) by converting each non-zero entry to 1 and compute A's hash signature H_A. We then divide $\mathcal{S}$ into groups by hash signature value. The algorithm is illustrated in Algorithm 22.

Tuning the performance. Using DOPH to divide $\mathcal{S}$ allows us to tune the performance of the overall algorithm. In particular, we can obtain a more compact output

Algorithm 21 Densified One Permutation Hashing (DOPH)

Input: Binary vector I, random permutation $h : \{1, \ldots, |V|\} \rightarrow \{1, ..., |V|\}$, hash signature length k, random binary vector D of length k
Output: Hash signature H
permute the values of I using h
divide V into k sequential bins of equal size (right pad V with zeroes if k does not divide $|V|$)
for each bin b_i, $1 \leq i \leq k$ **do**
    if b_i has a non-zero entry (i.e. b_i is non-empty) **then**
        $H_{b_i} \leftarrow$ index of first non-zero entry in b_i, $1 \leq H_{b_i} \leq |b_i|$
    else
        $H_{b_i} \leftarrow \emptyset$
for each H_{b_i} where $H_{b_i} = \emptyset$, $1 \leq i \leq k$
    if $D_i = 1$ **then**
        $H_{b_i} \leftarrow H_{b_j}$, j is index of first non-empty bin to the right
    else
        $H_{b_i} \leftarrow H_{b_j}$, j is index of first non-empty bin to the left
return $H = \{H_{b_1}, ..., H_{b_k}\}$

Algorithm 22 Weighted LSH Divide

Input: Graph $G = (V, E)$, current supernodes $\mathcal{S}$, signature length k
Output: Disjoint groups of supernodes: $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$
generate a random permutation $h : \{1, ..., |V|\} \rightarrow \{1, ..., |V|\}$
generate random binary vector D of length k
for each supernode $A \in \mathcal{S}$ **do**
    compute supervector V_A (as previously defined)
    $I_A \leftarrow binarize(V_A)$
    $DOPH(I_A, h, k, D)$ ▷ compute hash signature H_A (Algorithm 21)
divide supernodes in $\mathcal{S}$ into $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ by their hash signature
return $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$

with more running time or obtain a less compact output with less running time. Tuning is done through modifying the k value in DOPH which specifies the number of bins that we divide the input vector V into and the length of the hash signature. Increasing k means that vectors must be more similar to have the same signature, which results in $\mathcal{S}$ being divided into more groups of smaller size with higher weighted Jaccard similarity. Conversely, reducing k results in less groups but of larger size. As k increases, the running time of the merge algorithm significantly decreases since the merge algorithm receives groups of smaller size. Amount of compression also decreases due to the probabilistic nature of LSH since groups of smaller size result in a higher likelihood of good potential merge pairs being placed in different groups. Being able to tune k allows the option of trading compression for running time (depending on which is more important for a particular application). The number of groups increasing with k can also be explained combinatorially by analyzing the number of possible signatures for any given k, which we observe is $(\frac{n}{k}+1)^k$. This is because there are $\frac{n}{k}+1$ possible values per bin (namely $\frac{n}{k}$ indices and the *empty* value) and k total bins in a DOPH signature. Hence, the number of possible groups grows exponentially with k, compared to the shingle method in which the number of possible groups is fixed at n.

Efficiently computing Saving. As previously discussed, the merging step in [51] computes the SuperJaccard between supernodes to find the best merge partner for a particular supernode. SuperJaccard is an approximation for Saving, where Saving is the true amount of decrease in Eq. (4.1) when merging two supernodes. Here we present a method to directly compute the Saving between supernodes in running time no more than computing SuperJaccard.

For each group $\mathcal{S}^{(i)}$ of supernodes, we create a hashtable-of-hashtables data structure W. The first level hashtable of W is keyed by the supernodes in the group $\mathcal{S}^{(i)}$. A second level hashtable, denoted by W_A for some supernode A in $\mathcal{S}^{(i)}$, contains key-value pairs (B, val) where the key B is a supernode in $\mathcal{S}$ such that there exist edges in E between the nodes within A and B and the value val is the number of edges in E between A and B. This data structure enables us to find out the number of actual edges between any pair of supernodes in expected constant time and consequently compute its Cost and Saving with other supernodes. Algorithm 23 shows how to calculate Saving for a pair of supernodes A and B. The algorithm works along the lines of the decision process inside the encoding step described in Section 4.2. For this, it needs the numbers of edges between supernode pairs, which are conveniently

retrieved from the hashtable structure we described (see lines 4,5,8,9,11).

Algorithm 23 Compute Saving

1: **Input:** Hashtables W_A and W_B
2: **Output:** $Saving(A, B, \mathcal{S})$
3: initialize $Cost(A) = 0, Cost(B) = 0, Cost(A \cup B) = 0$
4: **for each** supernode $C \in$ keyset of W_A **do**
5: **if** keyset of W_B does not contain C **then**
6: $Cost(A)$+=min $\left(\frac{|A|\cdot(|C|-1)}{2}, W_A[C]\right)$
7: $Cost(A \cup B)$+=min $\left(\frac{(|A|+|B|)\cdot(|C|-1)}{2}, W_A[C]\right)$
8: **for each** supernode $C \in$ keyset of W_B **do**
9: **if** keyset of W_A does not contain C **then**
10: $Cost(B)$+=min $\left(\frac{|B|\cdot(|C|-1)}{2}, W_{BC}\right)$
11: $Cost(A \cup B)$+=min $\left(\frac{(|A|+|B|)\cdot(|C|-1)}{2}, W_B[C]\right)$
12: **else**
13: $Cost(A \cup B)$+=min $\left(\frac{(|A|+|B|)\cdot(|C|-1)}{2}, W_B[C] + W_A[C]\right)$
14: **return** $1 - \frac{Cost(A \cup B)}{Cost(A)+Cost(B)}$

After merging any two supernodes A and B, we update W by iterating over the keyset of the hashtable corresponding to the supernode with smaller size (say B). For each key-value pair $(C, W_B[C])$ in W_B (i.e. C shares an edge in E with B) we do the following steps:

1. If key C exists in W_A (i.e. C shares an edge in E with A), we set $W_A[C] = W_A[C] + W_B[C]$ (i.e we add the number of edges between B and C to the count of edges between A and C). Otherwise, we add a new pair $(C, W_B[C])$ to W_A.

2. If W_C is in W (i.e. C belongs to the same group as A and B) and key A exists in W_C, we set $W_C[A] = W_C[A] + W_C[B]$ and remove B from W_C. If key A does not exist in W_C, we create a new entry for A, set $W_C[A] = W_C[B]$, and remove B from W_C.

Finally, we remove B from W since the supernode B no longer exists after the merge. We note that computing Saving using Algorithm 23 requires only iterating over supernodes in $\mathcal{S}$ (in contrast to computing SuperJaccard which requires iterating over nodes in V). This helps us to speed up the merge step while eliminating the need of approximating the Saving computation.

Improving the encoding step. The encoding algorithm of [51] requires iterating over all supernodes and for each supernode A, identifying all supernodes $B \in \mathcal{S}$ for

which there is at least one edge in E between the nodes within A and B (i.e. $E_{AB} \neq \emptyset$). Implementing this step requires a significant amount of computational overhead especially for summary graphs with many supernodes. In a simple implementation, all pairs of supernodes (A, B) would be checked and if $E_{AB} \neq \emptyset$, we would perform the rest of the encoding algorithm. However, this implementation would take time that is quadratic in the number of supernodes leading to poor scalability. In a more careful implementation, for each supernode A, we only iterate over the supernodes B where $E_{AB} \neq \emptyset$. To do this, we require a preprocessing step where we iterate over the nodes in V within A, compute the edges incident to these nodes to obtain the supernodes B that shares an edge in E with A, and save these edges in a lookup table for the subsequent steps. However, there is a significant overhead due to computing, storing, and looking up incident edges for each supernode.

Nevertheless, even the more careful implementation above performs poorly when the number of supernodes is large and in some cases caused the algorithm to not complete within reasonable time. Here we will introduce a restructured encoding step algorithm that has faster practical performance, is more consistent, and requires little computational overhead aside from reading the edges in E.

Updated Encoding Step. In our encoding algorithm (Algorithm 24), for each edge $e \in E$, we add a 2-tuple (s, e) to a list L. In each 2-tuple, $e = (u, v)$ corresponds to the original edge and $s = (A, B)$ is a "candidate superedge" where A and B are the supernodes containing nodes u and v respectively. The candidate superedge identifies which pair of supernodes an edge is between. We then group L into $\{L^{(1)}, ..., L^{(k)}\}$ by their candidate superedge value s so that any supernode pair (A, B) where $E_{AB} \neq \emptyset$, which have edges to be encoded, have the edges between them grouped together (line 5 in Algorithm 24). Each group $L^{(i)} \in \{L^{(1)}, ..., L^{(k)}\}$ is associated with a pair of supernodes (A, B) and contains exactly the edges between A and B (namely E_{AB}) that are needed for the encoding step. Thus, for each group, we can look at the respective E_{AB} to decide whether or not to encode a superedge using the same conditions as in [44, 51]. Line 5 in Algorithm 24 can be done efficiently by lexicographically sorting the 2-tuples in L by their candidate superedge value s. Effectively, this groups all the edges in E by their supernode endpoints. Iterating through each group (line 6) can be done by linear scanning L in sorted order and a group change is detected by a change in the candidate superedge s of a 2-tuple during the scan. The edges between the pair of supernodes of a group are obtained by temporarily saving each edge e in the 2-tuple scan until the group changes, at which point the remainder of the encoding

step can be performed on the saved edges.

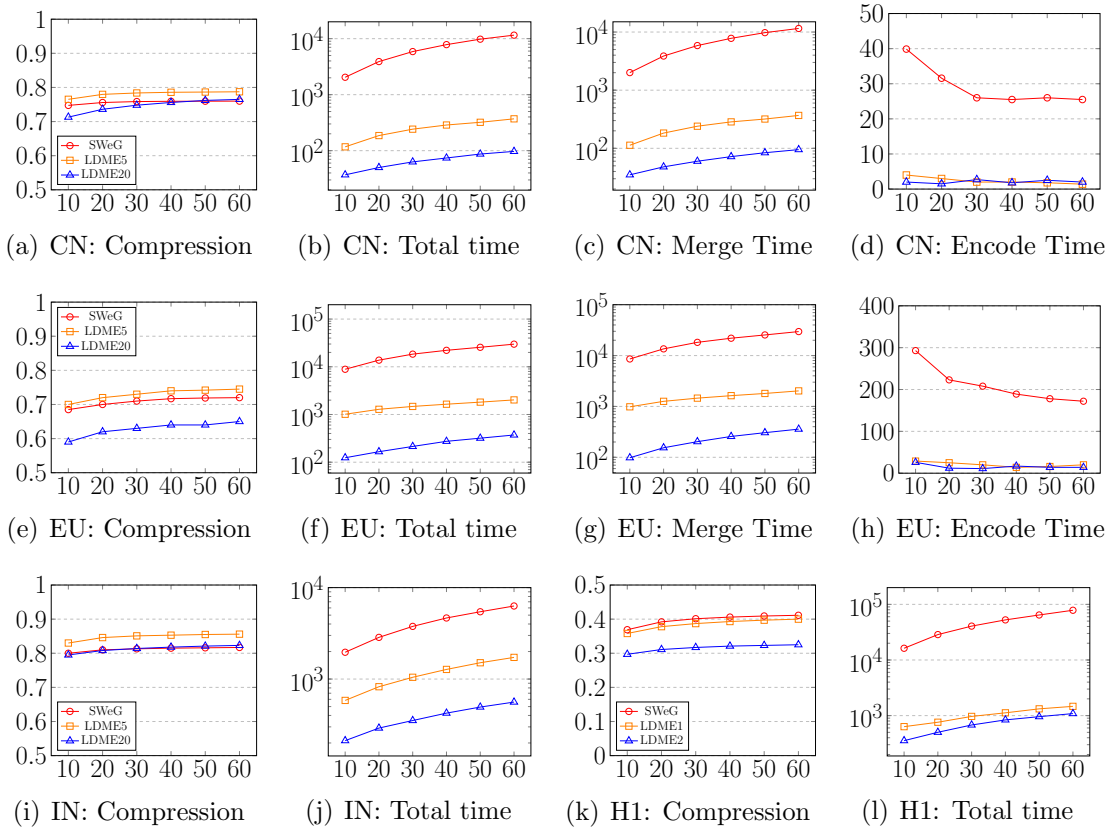
Algorithm 24 Updated Encoding Step

Input: Input graph $G = (V, E)$, supernodes $\mathcal{S}$
Output: Summary graph $\overline{G} = (\mathcal{S}, \mathcal{P})$, corrections C^+ and C^-
$L = \{\}$
for each edge $e = (u, v)$ in E **do**
    $A \leftarrow$ supernode containing u, $B \leftarrow$ supernode containing v
    $s \leftarrow$ candidate superedge (A, B), $L \leftarrow L \cup \{(s, e)\}$
group L into $\{L^{(1)}, ..., L^{(k)}\}$ by candidate superedge $s = (A, B)$
for each $L^{(i)} \in \{L^{(1)}, ..., L^{(k)}\}$ **do**
    $(A, B) \leftarrow$ candidate superedge s of $L^{(i)}$
    $E_{AB} \leftarrow$ set of edges e in $L^{(i)}$ (denoted E_{AA} if $A = B$)
    if $A \neq B$ **then**
        if $|E_{AB}| \leq \frac{|A| \cdot |B|}{2}$ **then** $C^+ \leftarrow C^+ \cup E_{AB}$
        else $\mathcal{P} \leftarrow \mathcal{P} \cup \{(A, B)\}$, $C^- \leftarrow C^- \cup (F_{AB} \setminus E_{AB})$
    else
        if $E_{AA} \leq \frac{|A| \cdot (|A|-1)}{4}$ **then** $C^+ \leftarrow C^+ \cup E_{AA}$
        else $\mathcal{P} \leftarrow \mathcal{P} \cup \{(A, A)\}$; $C^- \leftarrow C^- \cup (F_{AA} \setminus E_{AA})$
return $\overline{G} = (\mathcal{S}, \mathcal{P})$ and C^+, C^-

In summary, we significantly reduce the running time of the encoding algorithm of [44, 51] in practice by directly working with the edges in E. This reduces the overhead that comes from iterating over pairs of supernodes and computing the associated edges.

Time Complexity. The time complexity of LDME is dominated by the merge phase, which is $O((n/|\mathcal{S}^*|) \cdot |\mathcal{S}^*|^2) = O(n \cdot |\mathcal{S}^*—)$, where $\mathcal{S}^*$ denotes the largest group in $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ from the divide phase. Note that this is similar to the time complexity of the merge phase in SWeG, however the largest group size in LDME is much smaller than in SWeG, thus resulting in a significant improvement in running time in practice.

Space Complexity. The space requirement for storing $G, \overline{G}, C^+$, and C^- is $O(|V| + |E|) = O(|E|)$. The hashtable-of-hashtables W, created for each group, stores the number of edges between supernodes in the group and their adjacent supernodes, which in the worst case is $O(|E|)$. However, LDME's divide phase creates small groups, especially as k increases, thus the space requirement will be much less in practice. The other data structures used, such as the signatures and the encoding edge list, are all $O(|E|)$.



(a) CN: Compression (b) CN: Total time (c) CN: Merge Time (d) CN: Encode Time

(e) EU: Compression (f) EU: Total time (g) EU: Merge Time (h) EU: Encode Time

(i) IN: Compression (j) IN: Total time (k) H1: Compression (l) H1: Total time

Figure 4.2: The comparison between original SWeG, LDME5 and LDME20 in terms of compression, total time (seconds), divide/merge time (seconds), encode time (seconds) over 60 iterations.

Parallel Implementation Description. Similar to SWeG, LDME is highly parallelizeable and can run in a distributed environment for higher speed and scalability. In the dividing step, the DOPH signature of each supernode can be computed in parallel (lines 5 and 6 of Algorithm 22). Then, the merging step can be performed on each group in $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(m)}\}$ in parallel (line 5 in Algorithm 20). Finally, each supernode A in the encoding step can be processed in parallel, so line 2 of Algorithm 24 only reads in the edges incident to each A. Processing each supernode in parallel also removes the need for grouping candidate superedges via sorting in line 5.

4.4 Experiments

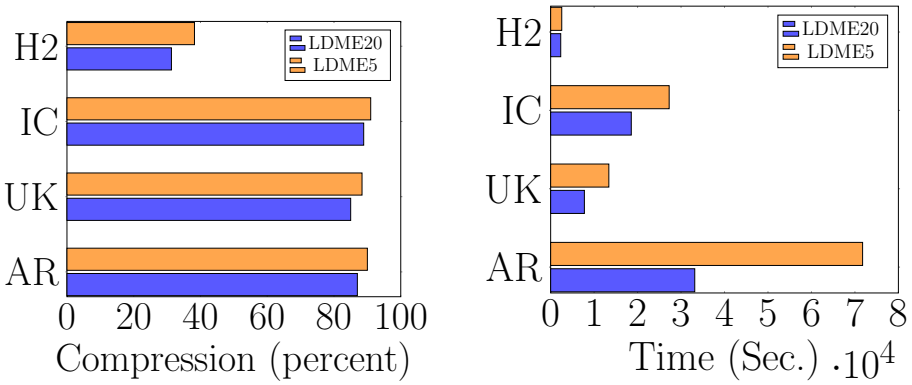


Figure 4.3: Compression and total time at final iteration 60 of LDME5/20 on graphs that SWeG could not complete within 1 day.

In our experiments, we compare our approach to SWeG [51], Mosso [28], and VoG [29] (we do not compare versus [44] because it is superseded by SWeG). In our experiments, we wish to test how our algorithmic improvements translate to gains in running time compared to the mentioned algorithms. We evaluate the implementations of all above algorithms on a single-threaded machine where we can better observe our main goal of reducing the amount of computation. We also compare the performance of the parallel implementation of LDME and SWeG in a distributed setting.

We experiment with two versions of our approach, one where we use DOPH signature length $k = 5$ in our dividing step and the other where $k = 20$. We call our general approach LSH-based Divide-Merge-Encode (LDME), and the two versions LDME5 and LDME20 for $k = 5$ and $k = 20$ respectively. Both versions demonstrate a significant speedup over the compared algorithms (often an order of magnitude), with LDME20 faster than LDME5. With respect to compression, LDME5 is very similar to SWeG and Mosso, whereas LDME20 shows some moderate reduction in compression as trade-off for its speedier execution over LDME5.

We use the datasets in Table 4.1 from Laboratory of Web Algorithmics (`http://law.di.unimi.it/datasets.php`). The sizes of the graphs we use are as follows: cnr-2000 is a (relatively) small graph, in-2004, eu-2005, and hollywood-2009 are medium graphs, and hollywood-2011, indochina-2004, uk-2002, and arabic-2005 are large graphs. Table 4.1 shows the characteristics of each graph. We note that the number of edges shown in Table 4.1 is after symmetrization, where we add the

Graph	Abbr	Nodes	Edges
cnr-2000	CN	325,557	5,565,380
in-2004	IN	1,382,908	27,560,356
eu-2005	EU	862,664	32,778,363
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	UK	18,520,486	529,444,615
arabic-2005	AR	22,744,080	1,116,651,935

Table 4.1: Summary of datasets

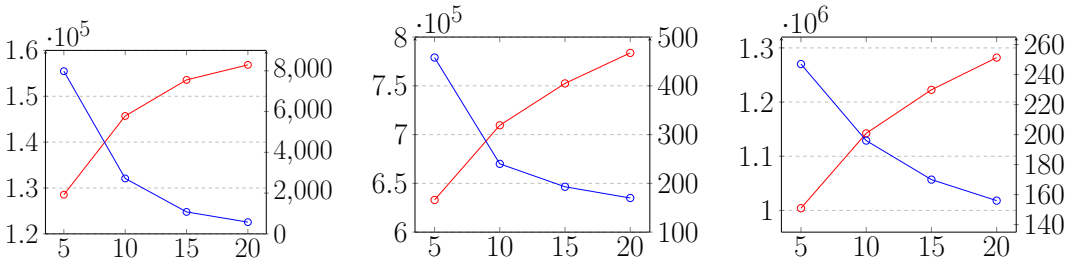
reverse of directed edges if they do not already exist.

LDME vs. SWeG. We evaluate the difference between SWeG and LDME5/LDME20 using four different metrics: (1) compression (2) total running time, (3) dividing and merging time, and (4) encoding time. For each metric, we ran SWeG and LDME5/LDME20 for $T = 10, 20, 30, 40, 50, 60$ iterations. The amount of compression was computed using the complement of the ratio between the number of superedges + correction set edges and number of original edges. Specifically:

$$Compression = 1 - \frac{|\mathcal{P}| + |C^+| + |C^-|}{|E|}$$

Figure 4.2 illustrates the compression and total running time of SWeG, LDME5 and LDME20 on graphs for which all algorithms could complete within a reasonable time of 1 day (CN, IN, EU, and H1). In terms of compression, LDME5 demonstrates a similar final compression as SWeG; CN, IN, and EU achieved 2% to 4% increase at iteration 60 and H1 had a 1% decrease. LDME20 demonstrates a final compression that matches or is only slightly lower than SWeG; CN and IN achieved 0.5% increase at iteration 60 while EU and H1 had a 7% and 8% decrease respectively. In terms of total running time, LDME5 demonstrates a 3.6x to 31x speedup over SWeG on CN, IN and EU, and is 53x faster on H1. LDME20 shows an even more significant speedup ranging from 11x (IN) to 96x (CN) faster than SWeG. Thus, the expected effect of increasing k from 5 to 20 in LDME is clearly demonstrated in both compression and running time.

Figure 4.2 also shows the comparison of SWeG and LDME's divide/merge and encode times on CN and EU (we do not show this for IN and H1 since the behavior is similar). Since the merging step dominates the total running time of the algorithms, the divide/merge time and total running time are similar. The breakdown of encoding



(a) CN (b) H1 (c) H2

Figure 4.4: The number of groups (red, left y-axis) and the largest group size (blue, right y-axis) for increasing values of k.

time illustrates the difference between our encoding algorithm and SWeG's encoding algorithm. LDME's encoding time is rather uniform through all iterations, while SWeG's starts high and decreases over iterations as the number of supernodes is compressed (since it scales based on —$\mathcal{S}$—). On CN, IN, EU, and H1, LDME's encoding time is 7x to 26x faster than SWeG, and for larger datasets (eg. UK, AR), SWeG's encoding could not complete within reasonable time while our encoding algorithm stayed consistently small.

To illustrate the superior scalability of LDME over SWeG, Figure 4.3 shows the final compression and running time of LDME5/20 on larger graphs H2, IC, UK, and AR, on all of which SWeG could not complete within reasonable time (1 day). SWeG ran overtime on these graphs due to its slow merging step and in some cases also due to its inefficient encoding step. Similar to the results in Figure 4.2, LDME20's compression is slightly lower than that of LDME5, but achieves a faster running time. AR, having over 1 billion edges, also shows that our algorithm can successfully summarize billion edge scale graphs using only a single machine.

Results of tuning k. Figure 4.4 illustrates the number of groups created in the dividing step and the size of the largest group for $k = \{5, 10, 15, 20\}$ on graphs CN, H1, and H2. As k increases, there is a clear significant increase and decrease in number of groups and max group size respectively.

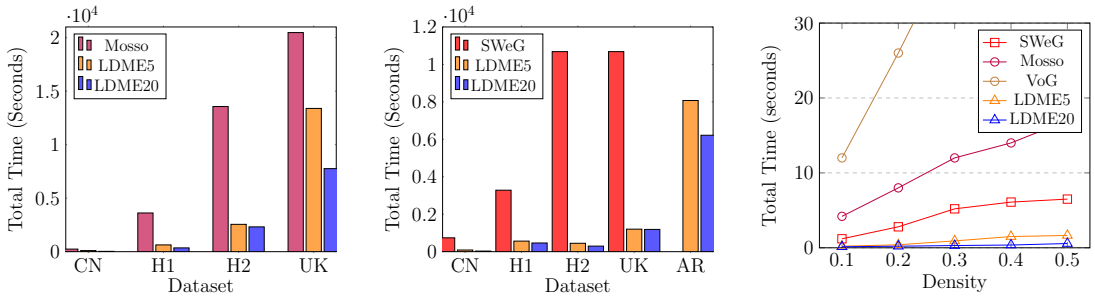


Figure 4.5: Running time of (a) LDME vs. Mosso on a single machine, (b) LDME vs. SWeG in a distributed environment, and (c) SBM experiments for LDME, SWeG, Mosso, and VoG on a single machine

LDME vs. Mosso and VoG. Figure 4.5 (a) shows the running time for Mosso and LDME5/20 (for 10 iterations) on CN, H1, H2, and UK. VoG was also run, but it was over 40x slower than LDME on all datasets, hence its running time is not displayed. We use the configuration ($e = 0.3$, $c = 120$) for MoSSo as used in [28], where e is the escape probability and c is the sample size of each trial. LDME5 and LDME20 achieved a 1.5x to 5.7x and 2.6x to 10.2x speedup over Mosso respectively. Mosso could also not complete AR within reasonable time (1 day) while both LDME5/20 could (see Figure 4.3).

Distributed Environment Experiments: Figure 4.5 (b) illustrates the running time of the parallel implementations of SWeG and LDME5/20, both run for 10 iterations, in a distributed environment. The implementation of both algorithms were done using Apache Spark and they were run on Amazon EMR clusters with the following specifications: 8 m5xlarge (4 vCore, 16GB memory, 64GB EBS storage) instances for CN/H1/H2 and 8 m5.2xlarge (8 vCore, 32GB memory, 128GB EBS Storage) instances for UK/AR. LDME's significant improvement in running time also translates to a distributed setting, where LDME5 achieved 3.0x to 23.8x speedup and LDME20 achieved 3.1x to 36.0x speedup on the experimented datasets. LDME also achieves higher scalability, as illustrated by SWeG being unable to complete AR within reasonable time (12 hours).

Stochastic Block Model Experiments: Stochastic block model requires two parameters for generating random graphs, the number of nodes in each community and the block matrix which shows the level of interaction between communities. We generate different random graphs with 3 communities, 300 nodes in each community and 900 nodes in total. We gradually increase the level of interactions between/within

communities to generate more dense random graphs. We compare the running time of LDME5/20 with MoSSo, SWeG and VoG. Figure 4.5 (c) shows the performance of each algorithm. VoG goes off the figure and MoSSo running time increases substantially as the density of graph increases. SWeG and LDME5/20 are quite resilient with respect to density and LDME5 is up to 8x faster than SWeG in some cases.

4.5 Conclusions

We proposed LDME, a correction set based graph summarization method that highlights the usefulness of weighted LSH to graph compression. LDME is able to handle large datasets using only a single machine and improves each step of SWeG, namely, the dividing, merging, and encoding steps. Furthermore, using weighted locality sensitive hashing in the dividing step allows for performance tuning of LDME where compression can be traded off for running time. With high compression settings, LDME achieves up to 53x times faster running time while maintaining compression rates as good as SWeG. With high speed settings, LDME achieves up to two orders of magnitude speedup while allowing a small loss in compression.

Chapter 5

Conclusion and Future Work

Many real world networks are huge and contain more than million nodes and billion edges. Graph summarization is an important task of finding a compact representation of the original graph called summary. The summary graph can be used for privacy, better visualization, reducing the foot print of graph on memory and answering queries more effectively in terms of running time. In this thesis we analyzed the graph summarization problem and investigated the state-of-the-art algorithms and showed their limitations. We then proposed five different graph summarization algorithms in order to overcome the challenges of existing algorithms. In particular, the contributions of this thesis were as follows.

- We first presented G-SCIS, a lossless algorithm based on clique and independent set decomposition. We showed that the G-SCIS summary is the smallest graph in terms of number of supernodes and finally we showed that the G-SCIS graph summary can be used as-is to answer several important classes of queries, such as triangle enumeration, Pagerank and shortest paths.

- We proposed T-BUDS, a utility-based lossy algorithm for fully controlled utility loss. T-BUDS achieved high scalability by combining memory reduction using Maximum Spanning Tree with a novel binary search procedure. T-BUDS outperformed state-of-the-art drastically in terms of the quality of summarization and is about two orders of magnitude better in terms of speed.

- We proposed Optimal, an incremental lossless graph summarization algorithm in the fully dynamic scenario in which edges are inserted into or removed from the graph. We showed that the lossless summary that Optimal maintains is the

smallest-possible-anytime in terms of the number of supernodes and showed that this algorithm is 8 orders of magnitude faster than batch algorithms and 12x faster than dynamic graph summarization competitors while at the same time achieving a 6x improvement in node reduction. We then proposed Scalable, another dynamic lossless algorithm, that achieves an additional order of magnitude over Optimal at the cost of having slightly less node reduction.

- Finally, we proposed a correction set-based graph summarization algorithm LDME, which is based on weighted locality sensitive hashing (LSH) that enables us to have a trade-off between running time and compression rate. We showed that LDME is about two orders of magnitude faster than competitors while achieving a compression ratio as good as them.

5.1 Future Work

In the following we discuss some ideas and research directions that can be beneficial to investigate in the future.

- While in literature, most algorithms work on deterministic graphs, many real world graphs such as influence, trust and biological graphs are modeled as probabilistic graphs in which the existence of an edge is associated with a probability. Making existing algorithms work on probabilistic graphs could be one of the future directions of this area.

- In Chapter 2 we showed how G-SCIS summary can be used as-is for counting the number of triangles more efficiently. There are more than 20 different 3,4 and 5-nodes graphlets and extending the use of G-SCIS summary to count all graphlets with more than 3 nodes could be another direction in future.

Bibliography

[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] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48, 2013.

[6] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.

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

[8] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. Scalable approximation algorithm for graph summarization. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 502–514. Springer, 2018.

[9] Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger, Andrey Ivanov, Yishai Oltchik, and Torsten Hoefler. Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Till Blume, David Richerby, and Ansgar Scherp. Incremental and parallel computation of structural graph summaries for evolving graphs. In *CIKM*, pages 75–84, 2020.

[11] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[12] Angela Bonifati, Stefania Dumbrava, and Haridimos Kodylakis. Graph summarization. *arXiv preprint arXiv:2004.14794*, 2020.

[13] Diane J Cook and Lawrence B Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1993.

[14] Cody Dunne and Ben Shneiderman. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3247–3256, 2013.

[15] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 38th ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2012.

[16] Sofia Fernandes, Hadi Fanaee-T, and João Gama. Dynamic graph summarization: a tensor decomposition approach. *DMKD*, pages 1397–1420, 2018.

[17] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[18] Xiangyang Gou, Lei Zou, Chenxingyu Zhao, and Tong Yang. Fast and accurate graph stream summarization. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)*, pages 1118–1129, 2019.

[19] Mahdi Hajiabadi, Jasbir Singh, Venkatesh Srinivasan, and Alex Thomo. Graph summarization with controlled utility loss. In *SIGKDD*, pages 536–546, 2021.

[20] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms. *arXiv preprint arXiv:2102.11169*, 2021.

[21] Michael Hay, Gerome Miklau, David Jensen, Don Towsley, and Philipp Weis. Resisting structural re-identification in anonymized social networks. *Proceedings of the VLDB Endowment*, 1(1):102–114, 2008.

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

[23] Christian Hübler, Hans-Peter Kriegel, Karsten Borgwardt, and Zoubin Ghahramani. Metropolis algorithms for representative subgraph sampling. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 283–292, 2008.

[24] Arijit Khan and Charu Aggarwal. Toward query-friendly compression of rapid graph streams. *SNAM*, pages 1–19, 2017.

[25] Arijit Khan, Sourav S Bhowmick, and Francesco Bonchi. Summarizing static and dynamic big graphs. *PVLDB*, 10(12):1981–1984, 2017.

[26] K. U. Khan. Set-based approach for lossless graph summarization using locality sensitive hashing. In *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops*, pages 255–259, 2015.

[27] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Set-based approximate approach for lossless graph summarization. *Computing*, 97(12):1185–1207, 2015.

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

[29] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Vog: Summarizing and understanding large graphs. *CoRR*, abs/1406.3411, 2014.

[30] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 8(3):183–202, 2015.

[31] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[32] K Ashwin Kumar and Petros Efstathopoulos. Utility-driven graph summarization. *Proceedings of the VLDB Endowment*, 12(4):335–347, 2018.

[33] Kyuhan Lee, Hyeonsoo Jo, Jihoon Ko, Sungsu Lim, and Kijung Shin. Ssumm: Sparse summarization of massive graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 144–154, 2020.

[34] Kristen LeFevre and Evimaria Terzi. Grass: Graph structure summarization. In *Proceedings of the 10th SIAM International Conference on Data Mining (SDM)*, pages 454–465, 2010.

[35] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[36] Cheng-Te Li and Shou-De Lin. Egocentric information abstraction for heterogeneous social networks. In *Proceedings of the 1st International Conference on Advances in Social Network Analysis and Mining*, pages 255–260, 2009.

[37] Chenhui Li, George Baciu, and Yunzhe Wang. Modulgraph: modularity-based visualization of massive graphs. In *Proceedings of the SIGGRAPH Asia 2015 Visualization in High Performance Computing*, pages 1–4, 2015.

[38] Edo Liberty. Simple and deterministic matrix sketching. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–588, 2013.

[39] Xingjie Liu, Yuanyuan Tian, Qi He, Wang-Chien Lee, and John McPherson. Distributed graph summarization. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 799–808, 2014.

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

[41] Ziyi Ma, Jianye Yang, Kenli Li, Yuling Liu, Xu Zhou, and Yikun Hu. A parameter-free approach for lossless streaming graph summarization. In *DASFAA*, pages 385–393. Springer, 2021.

[42] Antonio Maccioni and Daniel J Abadi. Scalable pattern matching over compressed graphs via dedensification. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1755–1764, 2016.

[43] Arun S Maiya and Tanya Y Berger-Wolf. Sampling community structure. In *Proceedings of the 19th international conference on World wide web*, pages 701–710, 2010.

[44] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the 34th ACM SIGMOD international conference on Management of data*, pages 419–432, 2008.

[45] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[46] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[47] Matteo Riondato, David García-Soriano, and Francesco Bonchi. Graph summarization with quality guarantees. In *Proceedings of the 14th IEEE International Conference on Data Mining (ICDM)*, pages 947–952, 2014.

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

[49] Neil Shah, Danai Koutra, Lisa Jin, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. On summarizing large-scale dynamic graphs. *IEEE Data Eng. Bull.*, 40(3):75–88, 2017.

[50] Neil Shah, Danai Koutra, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *SIGKDD*, pages 1055–1064, 2015.

[51] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. Sweg: Lossless and lossy summarization of web-scale graphs. In *Proceedings of the 28th international conference on World Wide Web*, pages 1679–1690, 2019.

[52] Anshumali Shrivastava. Simple and efficient weighted minwise hashing. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 1498–1506, 2016.

[53] Anshumali Shrivastava and Ping Li. Improved densification of one permutation hashing. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 732–741, 2014.

[54] Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[55] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1481–1496, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 34th ACM SIGMOD international conference on Management of data*, pages 567–580, 2008.

[57] Ioanna Tsalouchidou, Francesco Bonchi, Gianmarco De Francisci Morales, and Ricardo Baeza-Yates. Scalable dynamic graph summarization. *IEEE TKDE*, 32(2):360–373, 2018.

[58] Ning Yan, Sona Hasani, Abolfazl Asudeh, and Chengkai Li. Generating preview tables for entity graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1797–1811, 2016.

[59] Quinton Yong, Mahdi Hajiabadi, Venkatesh Srinivasan, and Alex Thomo. Efficient graph summarization using weighted lsh at billion-scale. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2357–2365, 2021.

[60] Zeqian Shen, Kwan-Liu Ma, and T. Eliassi-Rad. Visual analysis of large heterogeneous social networks by semantic and structural abstraction. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1427–1439, 2006.

[61] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: On query estimation in graph streams. *PVLDB*, 5(3), 2011.