Utility-Driven Graph Summarization

K. Ashwin Kumar
Symantec Research Labs
ashwin_kayyoor@symantec.com

Petros Efstathopoulos
Symantec Research Labs
petros_efstathopoulos@symantec.com

ABSTRACT

A lot of the large datasets analyzed today represent graphs. In many real-world applications, summarizing large graphs is beneficial (or necessary) so as to reduce a graph's size and, thus, achieve a number of benefits, including but not limited to 1) significant speed-up for graph algorithms, 2) graph storage space reduction, 3) faster network transmission, 4) improved data privacy, 5) more effective graph visualization, etc. During the summarization process, potentially useful information is removed from the graph (nodes and edges are removed or transformed). Consequently, one important problem with graph summarization is that, although it reduces the size of the input graph, it also adversely affects and reduces its utility. The key question that we pose in this paper is, *can we summarize and compress a graph while ensuring that its utility or usefulness does not drop below a certain user-specified utility threshold?*

We explore this question and propose a novel iterative utility-driven graph summarization approach. During iterative summarization, we incrementally keep track of the utility of the graph summary. This enables a user to query a graph summary that is conditioned on a user-specified utility value. We present both exhaustive and scalable approaches for implementing our proposed solution. Our experimental results on real-world graph datasets show the effectiveness of our proposed approach. Finally, through multiple real-world applications we demonstrate the practicality of our notion of utility of the computed graph summary.

PVLDB Reference Format:
K. Ashwin Kumar, Petros Efstathopoulos. Utility-Driven Graph Summarization. *PVLDB*, 12(4): 335-347, 2018.
DOI: https://doi.org/10.14778/3297753.3297755

1. INTRODUCTION

A lot of the vast amounts of information we are producing and analyzing today can be represented as graphs. This fact becomes clear if one consider all the real-life data networks that can be abstractly perceived as nodes connected by edges: social networks, financial transaction networks, communication networks, citation networks, parcel shipment data, protein-protein interaction networks, gene regulatory networks, disease transmission networks, ecological food networks, sensor networks, just to name a few. The size of such graphs is growing at an unprecedented rate, spanning millions and billions of nodes and edges. For instance, Google stores more than 1 trillion indexed pages that contain billions of incoming and outgoing links. Similarly, Facebook has 800 million active users and related network data. At the current rate of data volume increase, it is becoming highly impractical to store, process, analyze, and visualize these big graphs. Therefore, in order to make graph data management, processing and visualization tractable, summarization techniques are becoming increasingly important.

There is a plethora of benefits to employing graph summarization methods. First, given planetary scales of real-world graphs [18], graph summarization helps in reducing the size of the graph thereby reducing the on-disk storage footprint. The reduced graph can also be loaded directly into memory to improve the performance of analytics algorithms [25]. Second, many graph algorithms that are otherwise too complex or costly to run on larger graphs can be efficiently executed on summary graphs, with adequately accurate results [16]. Third, most of the real-world graphs suffer from a "small world" effect which makes them look too tangled to be effectively visualized and interpreted, resulting in the "hairball" graph phenomenon. Graph summarization essentially makes them simpler to visualize on a small screen in-turn helping with better analysis of these graphs [33, 5, 20, 14, 3]. Finally, when the original data is privacy sensitive, graph summarization may help conceal private information [12], thus enabling privacy-preserving analytics—especially among multiple mutually-distrustful parties.

A key challenge with graph summarization is that it can have a severe impact on the amount of "useful information" represented by the graph for the task at hand—i.e., the *utility* of the graph. Furthermore, it is difficult to predict the reduction in utility a graph will suffer when summarized. Ideally, we should be able to estimate the utility at each summarization step so that the obtained graph summary meets a user-specified utility threshold. To the best of our knowledge, state-of-the-art graph summarization approaches [23, 25] focus primarily on minimizing graph reconstruction error, and largely ignore the utility aspect—where the relative importance of nodes and edges should be considered during the summarization process. To address this gap, we pose the following key question:

Can we summarize a graph and compress it as much as possible, while ensuring that its utility does not drop below a user-defined utility threshold?

In other words, we desire a graph summarization system that permits a user to query a graph summary with given utility. To achieve this, our summarization algorithm must be able to keep track of the utility of the graph at each step of the summarization process. Moreover, we need utility estimation to be inexpensive and yet faithfully represent certain important properties of the underlying graph which we want to retain in the computed graph summary.

In our effort to achieve these goals, we evaluated various graph summarization techniques that have been proposed. In the *sparsification approach* edges are filtered based on certain criteria to simplify

Proceedings of the VLDB Endowment, Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297755

the underlying graph. On the other hand, the *sampling approach* performs sampling of a subset of nodes or edges so as to form a simpler representation of the original graph. The most popular approaches, however, are different variants of the *grouping approach*, that employ meaningful grouping of nodes into supernodes and edges into superedges to compute a graph summary. Grouping approaches owe their popularity to the fact that they are expressive enough to allow a user to logically explain the computed graph summary with respect to the original underlying graph. Moreover, iterative grouping approaches allow us to record the list of corrections made across the iterations, which can help us to reconstruct the exact original graph, or an approximate version of it, from the summary if needed. Subsequently, helps with provenance and explainability, where one can explain the steps taken to reach a particular summary for a given graph (useful for forensics, anomaly detection etc). Also, the iterative nature of the algorithm (grouping of nodes into supernodes and edges into superedges and vice versa) enables meaningful visualization and complex analysis during the summarization process. Therefore, for all the benefits it provides, we specifically focus on iterative grouping-based graph summarization approaches. However, since grouping-based graph summarization with minimum reconstruction error is shown to be NP-Hard [35], it is common to use heuristics and approximations to implement such algorithms.

In this paper, we propose a novel *utility-driven graph summarization* (UDS) technique, where graph utility is incrementally computed while iteratively performing the summarization. This allows us to obtain a summary with a user-specified utility threshold, thus offering the benefits of summarization while providing utility guarantees.

Our contributions in this work are as follows:

1. We introduce a new framework to measure the utility of a graph while it is being perturbed by the deletion of existing edges or the addition of spurious edges. Furthermore, we judiciously extend it to compute utility for graph summaries.
2. We present theoretical result showing computational intractability of UDS problem for obtaining a near optimal solution.
3. We introduce a novel UDS algorithm that iteratively summarizes a given graph by employing an objective function that maximizes the utility at each step of the transformation. Also, during iterative summarization of the graph, UDS incrementally computes and keeps track of the running utility value.
4. We improve scalability by orders of magnitude by proposing a memoization-based approach for UDS.
5. We conduct a comprehensive experimental study using several real datasets and applications, and the results demonstrate that UDS is capable of generating high-utility graph summaries.

The rest of the paper is organized as follows: In Section 2, we present the relevant background and the different concepts discussed in this paper. In Section 3, we present the formal definition of utility, describe the set of properties and conditions that a desirable utility metric should satisfy and introduce a generic framework to estimate utility of a perturbed graph given its base graph. We present our UDS approach in Section 4.1. We describe how we use memoization to improve the scalability of our technique for UDS in Section 4.2. In Section 5 we present experimental results evaluating the efficiency and effectiveness of UDS. Finally, we present related work in Section 6 and conclude in Section 7.

2. PRELIMINARIES

In this section, we present the background for graph summarization and the different concepts discussed in this paper.

Graph Summary. Given a graph $G = (V, E)$, its graph summary $\mathscr{G}_S = (\mathscr{V}_S, \mathscr{E}_S)$ where $\mathscr{V}_S = \{S^1, S^2, \ldots, S^k\}$ is a set of supernodes such that $k < |V|$. If $u \in V, v \in V$, then S_u represents the supernode containing node u and S_{uv} represents the supernode containing both the nodes u and v. Essentially, $\mathscr{V}_S$ consists of disjoint sets (*supernodes*) of nodes in V such that $V = \cup_{i=1}^{k} S^i$ and $S^i \cap S^j = 0$ $(\forall i \neq j)$. In $\mathscr{G}_S$, the edges $E_{N_{S^i}} \subset E$ connecting the set of nodes N_{S^i} belonging to a particular supernode S^i are not maintained. Whereas, only edges connecting individual supernodes are maintained. Also, if supernodes S^i and S^j are connected with a superedge, then $A_{i,j}$ represents the actual cross edges connecting the nodes in S^i and S^j. On the other hand, $\prod_{i,j}$ denotes the bipartite graph connecting the nodes in supernodes S^i and S^j where $(S^i, S^j) \in \mathscr{V}_S$. Alternative notations for $A_{i,j}$ and $\prod_{i,j}$ that we use in this paper are A_{S_u,S_v} and $\prod_{S_u,S_v}$ where $u \in V, v \in V$. Also, in this work, we assume un-directed, un-weighted and edge un-labeled graphs.

Reduction in Nodes (*RN*). We understand the effectiveness of our proposed techniques on varying *RN*. Formally, $RN = \frac{|V|-|\mathscr{V}_S|}{|V|}$, where a value of 0.2 means 20% of original nodes are collapsed into supernodes and summary retains 80% of the graph unmodified.

Zero Loss Encoding Transformations. We define certain encoding transformations (as shown in Figure 1) used to represent a group of nodes and edges in graph G with supernodes and superedges in a summarized graph $\mathscr{G}_S$ without loss of information. Rule 1: a group of nodes that are not connected to each other in the graph G is simply represented by a supernode without a self-loop. Rule 2: a group of nodes that form a clique in graph G is represented by a supernode (with a self-loop). Rule 3: if there is an all-to-all connection between two sets of nodes, then they are represented by two supernodes connected with a single superedge. "Zero loss" in this context means that if we apply these transformations in reverse order on a graph summary, then we should be able to obtain the original graph without needing any additional information or corrections. Note that in this context, *zero loss* also implies 100% utility because the transformations are able to preserve all the salient regions of G. We make use of these transformations during summarization and calculation of utility (Sections 3 and 4.1).

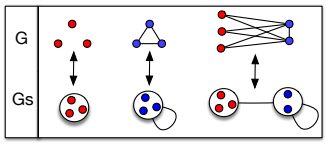
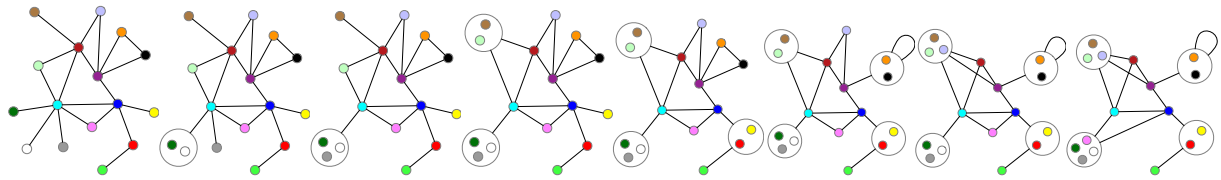


Figure 1: Examples of three encoding rules for zero-loss summarization

Utility (*EU*). The utility $0 \leq EU \leq 1$ of any graph $\mathscr{G}_S$ that is obtained by transforming an graph G indicates the usefulness of $\mathscr{G}_S$ with respect to G. The higher the extent to which important regions in G are preserved in the transformed graph, the greater the utility.

Example of Utility-Driven Graph Summarization (UDS). Let us consider an example. Figure 2 presents iterations of a desirable UDS system. We envision a summarization system that reports at each iteration the current *EU* and *RN* values of graph summary $\mathscr{G}_S$. Figure 2 offers the values for *EU* and *RN*, whose calcualtion is discussed in-detail in the coming sections. The input graph is shown in Figure (2a). The user provides a utility threshold Γ_U as a predicate to the UDS system, indicating that the summary $\mathscr{G}_S$ should have utility *no less than* Γ_U. In this example, let's say Γ_U equals 0.9. Figures (2b) – (2h) show the first eight iterations of graph summarization with varying *EU* and *RN* values along the way. At every iteration, a pair of nodes is selected and collapsed to form supernodes, and neighboring edges are adjusted accordingly. The summarization system analyzes the important parts and regions of the input graph (i.e., the output of the previous iteration) and prioritizes the order in which nodes are collapsed accordingly. In every iteration, the objective is to preserve important regions of the G as much as possible in $\mathscr{G}_S$. In the first iteration, two nodes are collapsed into a supernode, and edges are adjusted accordingly.



(a) Input graph **(b)** EU:1.0, RN:0.06 **(c)** EU:1.0, RN:0.13 **(d)** EU:0.99, RN:0.19 **(e)** EU:0.98, RN:0.25 **(f)** EU:0.98, RN:0.3 **(g)** EU:0.95, RN:0.38 **(h)** EU:0.92, RN:0.44

Figure 2: Example output of utility-driven graph summarization

Note that, in this iteration, the EU value remains 1.0 because we can still reconstruct G from $\mathcal{G}_S$ by simply applying the decoding rules of Figure 1. Also, $RN = 0.06$ as the number of nodes in $\mathcal{G}_S$ is reduced by 1. By the end of the second iteration, EU remains 1. In the third iteration, however, EU is reduced to 0.99, since reconstructing G from $\mathcal{G}_S$ produced in this step will introduce spurious edges. The reduction in EU is 0.01, based on the extent to which important regions are affected in G. Similarly, all iterations from 4 to 7 cause a drop in EU. Note that the quantum of reduction in EU from (2d) to (2e) is less than (2f) to (2g). This is because the merge step at (2e) preserves important regions better than the merge step at (2g). This will be explained in detail in coming Sections. Overall, the algorithm terminates at the seventh iteration (2h) as any attempt to further summarize the graph would cause the EU to drop below the user-specified threshold $\Gamma_U = 0.9$. Finally, the computed graph summary $\mathcal{G}_S$ (in Figure (2h)) is presented to the user as the output.

3. UTILITY OF A GRAPH SUMMARY

The fulcrum of this work is our proposed method for calculating the utility of a graph summary with respect to an underlying graph. We approach this problem by attempting to reconstruct the original graph G from a summary $\mathcal{G}_S$ with no extra information. For reconstruction, we apply the reverse of the transformations discussed in Section 2. This can result in the loss of original edges as well as introduction of spurious edges. Supernodes with self-loops are expanded into a clique of their contained nodes, otherwise they are expanded into disconnected nodes. A pair of sets of base nodes form a bipartite graph if the corresponding supernodes are connected by a superedge, otherwise they are completely disconnected. More formally, given $\mathcal{G}_S$ of graph G, we reconstruct the graph $G' = (V', E')$ from $\mathcal{G}_S$ such that $V = V'$. The number of nodes and the node set in the G' are equivalent to that of G, although the number of edges might vary—primarily due to the error introduced by graph summarization. Figure 3 presents an example of a graph G, its summarization $\mathcal{G}_S$, and graph G' which is reconstructed from $\mathcal{G}_S$ by applying the rules shown in Figure 1 in the reverse order.

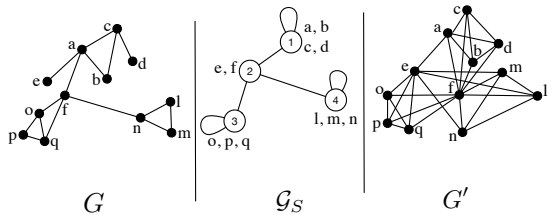


Figure 3: Example of a graph G, its summary $\mathcal{G}_S$, and reconstruction G'

Once $\mathcal{G}_S$ is transformed into a reconstructed graph G', the problem of calculating the utility of a graph summary is reduced to the problem of calculating the utility of G' with respect to G, using a utility function denoted by as $EU(G')_G$. In essence, *when there is greater structural similarity between G and the reconstructed graph G′ (i.e., the extent to which important edges and regions in G are preserved in G′) then the utility of the $\mathcal{G}_S$ is higher.* The reconstructed graph G' obtained from $\mathcal{G}_S$ is equivalent to a graph G' obtained by perturbing G (by adding certain spurious edges, or removing original edges, or both). Therefore, from now on we will call the reconstructed graph as the perturbed graph. Next, we present a generic framework to calculate the utility of G' with respect to G.

Generic Framework for a Graph Utility Function. Our key intuition is to penalize the utility of graph G' in accordance with the introduced perturbations. The amount of cost or penalty should be based on the importance of edges that are missing, or the number of spurious edges introduced, or both. An intuitive way to assess the relative importance of edges in the orginal graph G is by computing normalized edge centrality scores $edgeIS$. If $\{E - E'\}$ is the set of edges missing from G' compared to G's original edges, then the utility of G' is penalized by the sum of relative importance scores of missing edges. Next, we should penalize G''s utility according to any spurious edges it contains, that did not exist in G. We do this by calculating the proportion of spurious edges introduced in G' to the total number of spurious edges possible in the base graph G. More formally, the maximum number of spurious edges that can be introduced in G is $\binom{|V|}{2} - |E|$. If $\{E' - E\}$ is the set of spurious edges introduced in G', and assuming homogeneity, then for each spurious edge the utility $EU(G')_G$ is penalized by the amount $\frac{1}{\binom{|V|}{2}-|E|}$.

Algorithm 1 Generic Graph Utility Function (GGUF)

```
1:  procedure GGUF(G = (V,E), G' = (V,E'))
2:      utility = 1.0
3:      edgeIS = normalize(edge_centrality_scores(G))
4:      if G ≠ ∅ and G' ≠ ∅ then
5:          for e ∈ {E − E'} do
6:              utility = utility − edgeIS[e]
7:          end for
8:          for e ∈ {E' − E} do
9:              penalty = 1 / (C(|V|,2) − |E|)
10:             if penalty < utility then
11:                 utility = utility − penalty
12:             else
13:                 utility = 0
14:             end if
15:         end for
16:     end if
17:     return utility
18: end procedure
```

The value of utility is in the range [0,1]. Given a non-empty and non-clique graph G, there are four notable conditions under which the utility of G' is zero: 1) if G' is an empty graph, 2) if G' is a clique, 3) if G' is missing all the original edges, and 4) if G' contains all the possible spurious edges. Pseudocode for the generic graph utility function $GGUF$ is shown in Algorithm 1. Without loss of generality, it can be easily extended to weighted graphs where penalties will be weight adjusted. Moreover, the generic nature of $GGUF$ allows us to plug-in a variety of centrality metrics to form different types of utility functions each exhibiting different properties. Next, we

identify certain intuitive properties a utility function should exhibit and discuss how to assess its desirability.

Assessing the Desirability of a Utility Function To make a utility function aware of the important regions of G that are preserved in G', we use a set of fairly intuitive properties described in Table 1 that a desirable graph utility metric should exhibit. The key motivation in defining these properties and imposing necessary conditions for a desirable utility metric is that the maximization of such a utility metric during summarization should help maintain the results of important graph algorithms, such as ranking and community detection. To further explain these properties and test the desirability of a utility function, we use example model graphs shown in Figure 4 with various shapes and varying number of missing edges, such as: clique, path, cycle, barbell, wheel barbell, etc. Note that these examples are not exhaustive and are only meant to explain the key concepts. Also, it is not necessary that a desirable utility function exhibits all the listed properties in conjunction; it is only required to exhibit each property independently. We present an example test criterion 3.1 that uses the shown model graphs to test if a utility function exhibits the desired property—in this case criterion **C1**.

Table 1: Properties of a desirable Graph Utility Function

Criteria	Properties	Description
C1	*Edge Importance*	Changes that create disconnected components or weaken the connectivity should be penalized more than the changes that maintain the connectivity properties of the graphs.
C2	*Spurious Edge Awareness*	More spurious edges must lead to lower utility.
C3	*Weight Awareness*	In weighted graphs, higher the weight of the removed edge or added spurious edge is, the greater the impact on the similarity measure should be.
C4	*Edge Submodularity*	A specific change is more important in a graph with fewer edges than in a much denser graph.

Example Test Criteria 3.1 Consider barbell graphs B_n, mB_n and mmB_n to explain **C1**: *edge importance* criterion. Graph B_n has two cliques of size n_1 and n_2, such that $n = n_1 + n_2$. Graph mB_n has an edge removed from one of the cliques in B_n, where graph mmB_n has a missing bridge edge from B_n. In this case, according to *edge importance* criterion **C1**, following should satisfy:

$$(EU(mB_n)_{B_n} - EU(mmB_n)_{B_n}) > 0 \tag{1}$$

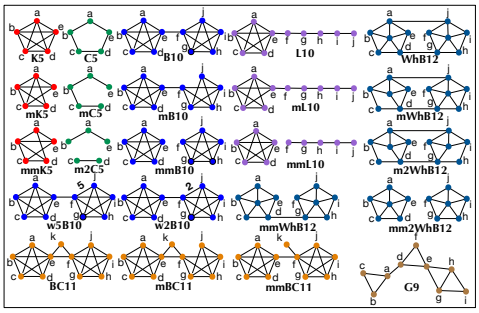


Figure 4: Model synthetic graphs used to validate utility function – K_n: *clique of size n*, P_n: *path of size n*, C_n: *cycle of size n*, L_n: *lollipop of size n*, B_n: *barbell of size n*, WhB_n: *wheel barbell of size n*, m_X: *missing X edges*, and mm_X: *missing X "bridge" edges*.

Similarly, additional example test criteria are presented in Section 5.1 to test if a utility function exhibits the remaining desired properties. Also, in Section 5.2 we present experimental results where we try various centrality metrics in *GGUF* and provide guidelines for the right set of centrality metrics to be plugged-in, so as to create a utility function that exhibits the properties of Table 1.

Discussion. Calculation of *utility* $EU(G')_G$ and structural similarity through simple graph edit distance (GED) between G' and G although seem similar, they differ in significant ways. GED essentially counts the number of different edges between the original graph and the restructured graph based on the graph summary. It can be noted that GED does not differentiate between non-important regions from important regions in the graph as *GGUF* does. Moreover, in simple GED, cost of edit operations is fixed, whereas in our case cost of edits is dynamic and depends on the structure of the original graph. Also, simple GED violates certain key properties that our utility function satisfies. For example, consider a graph $G = (V,E)$ with $|E| = \binom{|V|}{2} - 1$ edges and lets say $G' = (V,E')$ be its perturbed graph that is a clique with $|E'| = \binom{|V|}{2}$ edges. Then utility of G' with respect to G is zero (lowest) according to *GGUF* (Algorithm 1), but a simple GED would calculate the *utility* value > 0, where *utility* is calculated as $1 - \frac{d}{|E|}$, where d is the number of edits or distance. Intuitively, a *utility* value of zero is desirable in this case, because if G is a non-clique and non-empty, then no matter how dense G is, if G' is a clique, then essentially G' does not reveal any information with respect to G, thus rendering its utility equal to zero. We note that simple GED violates all the desired properties of an ideal utility function (Table 1) except C2 whereas *GGUF* when plugged with appropriate centrality metric satisfies all the properties. We have included an experiment in Table 5 to demonstrate this. We also note that, the simplicity of *GGUF* permits us to easily extend it so as to incrementally calculate the utility of G' while it is being perturbed. In this case, we start with a utility of 1.0 that represents $G' = G$. As we perturb G' by deleting (or adding spurious) edges, or both, we penalize the utility accordingly by subtracting the appropriate cost. Similarly, we incrementally calculate the utility of $\mathscr{G}_S$ at each iteration, by analyzing the possible perturbations without actually generating the reconstructed graph at each summary step.

4. UTILITY-DRIVEN SUMMARIZATION

We begin our discussion by presenting the mathematical formulation of our problem. Given a graph $G = (V,E)$ and utility threshold Γ_U, we want to summarize the graph G as much as possible by grouping nodes into minimum number of supernodes $\mathscr{V}_S$ and form superedges $\mathscr{E}_S$ between supernodes such that the difference between total utility of retained actual edges and total penalty of introduced spurious edges is very close to the given Γ_U. Initially, each node in the original graph is its own supernode in the summary graph.

$$minimize\ (|\mathscr{V}_S|) \tag{2}$$

Subject to

$$\sum_{S^i}\sum_{S^j}\left(\left(\sum_{\substack{e\in A_{i,j}\\(S^i,S^j)\in\mathscr{E}_S}} edgeIS[e]\right) - \frac{|\prod_{i,j} - A_{i,j}|}{\binom{|V|}{2} - |E|}\right) \geq \Gamma_U \tag{3}$$

Since problem of graph summarization is shown to be NP-Hard [35], one may be interested in obtaining a partition that is a p-approximation for some $p > 1$. However, a computational intractability result for obtaining a near optimal partition can be established as follows.

Theorem 4.1 [No Efficient Approximation Theorem] For any $\varepsilon > 0$, there is no $O(n^{1-\varepsilon})$-approximation for the problem of obtaining a feasible graph summarization with a minimum number of supernodes for a given utility threshold, unless NP = ZPP [1].

[1]This intractability result is based on the widely believed assumption that complexity classes NP and ZPP are different [29].

PROOF. Davidson et al., [4] have proved that for the problem of obtaining a feasible clustering with a minimum number of clusters under cannot-link (CL) constraints if, for some $\varepsilon > 0$, there exists $O(n^{1-\varepsilon})$-approximation for the feasibility problem then that would imply NP = ZPP. Here, CL constraints involve data points (that are required to be) in different clusters. Following this result, we directly reduce the problem of obtaining a feasible clustering with a minimum number of clusters to the our problem to prove the result.

Given a set of data points $D = \{d_1, d_2, ..., |V|\}$. Let $E_{i,j}$ be the measure of distance between data points where $0 < E_{i,j} \leq 1$ represents the points that are relatively closer to each other and $E_{i,j} = 0$ otherwise. Let $\mathscr{V}_S$ be the set of clusters of data points. Initially each data point d_i is its own cluster S^i. It is straightforward to see that data points D with prior distance values represent a graph $G = (V, E)$ where values $0 < E_{i,j} \leq 1$ represent edge weighted graphs and they represent edge unweighted graphs if these $E_{i,j}$ takes value of 0 or 1. Values of $A_{i,j}$, $\prod_{i,j}$ and $edgeIS$ can be calculated based on $E_{i,j}$ values. Since these values are defined over the data points that are in different clusters, constraint (Equation 3) using these values is essentially formed by set of CL constraints. Objective of minimizing the number of clusters $|\mathscr{V}_S|$ for a given set of CL constraints on pair of data points can be directly mapped to the objective of grouping the nodes from the original graph into minimum number of supernodes subject to the set of constraints involving pairs of nodes in different supernodes as shown in Equations 2 and 3. Proof completes. ■

Because it is not possible to devise a feasible or efficient approximation algorithm for the problem at hand. Instead, we rely on greedy heuristics that does best effort at each step taken.

4.1 Iterative Greedy UDS

We present a novel iterative greedy UDS algorithm with an incremental utility update. Our primary goal is to summarize the given graph G so as to compress it to an extent such that the utility of the summary graph $\mathscr{G}_S$ does not drop below a user-specified threshold Γ_U. To compose our algorithm we need to determine the following steps, based on principles presented in the previous Section: 1) introduce a strategy for grouping nodes, 2) find an iterative, utility-driven summarization recipe, 3) come up with appropriate superedge connectivity criteria, 4) present techniques to incrementally keep track of utility, 5) optimize the algorithm's performance and scalability.

Prioritizing Candidates to Merge. One way to prioritize the merging of nodes is by considering edge importance. The goal is to pick an edge e with the lowest importance and merge the nodes u and v at e's end-points so as to form a supernode w. However, this approach completely forgoes the benefit of merging nodes that are indirectly connected to each other. Many a times, collapsing nodes that are not directly connected and forming appropriate superedges might result in higher utility. For example, it is often beneficial to collapse nodes that have many common neighbors [25]—even not directly connected. Therefore, at each step, we consider pairs of nodes that are both 1) directly connected by an edge, or 2) indirectly (2-*hop*) connected via common neighbors, as candidates to form supernodes. Given a list of both 1-hop and 2-hop connected node pairs, we seek to prioritize or sort this list in ascending order of importance (denoted by $\uparrow$). We calculate the normalized node centrality scores $nodeIS$ for the nodes in the base graph and then calculate the combined importance score for a node pair $p =< u, v >$ as the sum of function of normalized centrality scores of the nodes—given by $(f(nodeIS[u]) + f(nodeIS[v]))$. In our implementation we use a square function as $f()$ as it helps in further delaying the merging of important nodes with relatively lesser ones. Let H be the list of node pairs sorted by their combined importance scores. Also, let $edgeIS$ be the map that maps each edge to its importance score. An edge importance score is calculated as the normalized edge centrality.

Iterative Greedy Summarization. As shown in Algorithm 2, we initially map each node in the base graph G to a unique supernode in the summarized graph $\mathscr{G}_S$. All edge connections between nodes in G are maintained between corresponding supernodes in $\mathscr{G}_S$. At each algorithm step, we pick from H the node pair (u, v) with the lowest importance score. Unless nodes u and v belong to the same supernode $S_u = S_v$, their corresponding supernodes S_u and S_v are collapsed into supernode S_{uv}. Let $V_{S_{uv}}$ indicate the set of nodes in G belonging to a particular supernode S_{uv}. We calculate the set of potential neighbors $\eta_{S_{uv}}$ of S_{uv} in $\mathscr{G}_S$ by finding the set of 1-*hop* neighbors of all nodes belonging to $V_{S_{uv}}$ in G and by calculating their corresponding supernodes in $\mathscr{G}_S$. For every unique potential neighbor $S_n \in \eta_{S_{uv}}$, where $n \in G$, we need to decide whether connecting S_{uv} and S_n with a superedge is beneficial for utility. We commit the decisions for all the potential neighbors in ascending order of the calculated penalties. Procedure $connectSuperEdge(\ldots)$ (pseudocode in Algorithm 3, discussed later) returns true if the given pair of supernodes should be connected by a superedge, or false otherwise. For a pair of supernodes this procedure calculates 1) $seCost$:

Algorithm 2 Utility-Driven Graph Summarization

```
1: procedure UDSUMMARIZER(G = (V,E), Γ_U)
2:     Initialize: utility = 1; V_S = {u : {u} | u ∈ V}; E_S = {({u},{v}) |
       (u,v) ∈ E}; S = {u : u | u ∈ V}
3:     nodeIS, edgeIS = normalize(centrality_scores(G))
4:     P_2hop = {(a,c) | (a,b) ∈ E, (b,c) ∈ E}
5:     H = sort(P_2hop | ↑ (f(nodeIS[a]) + f(nodeIS[c])), ∀(a,c) ∈ P_2hop)
6:     while utility ≥ Γ_U and H ≠ ∅ do
7:         (u,v) = H.pop()
8:         if S_u ≠ S_v then
9:             S_uv = {S_u ∪ S_v}
10:            V_S = {V_S ∪ S_uv} − {S_u, S_v}
11:            η_Suv = {S_b ∈ V_S | b ∈ η_a, ∀a ∈ S_uv} − {S_u ∪ S_v}
12:            for S_n ∈ η_Suv do
13:                bool, penalty = connectSuperEdge(S_uv, S_n, G, edgeIS)
14:                η_Suv[S_n].connect = bool
15:                η_Suv[S_n].penalty = penalty
16:            end for
17:            η_Suv = sort(η_Suv | ↑ (penalty))
18:            for S_n ∈ η_Suv do
19:                if η_Suv[S_n].connect is true then
20:                    E_S = {E_S ∪ (S_uv, S_n)} − {(S_n, S_u), (S_n, S_v)}
21:                end if
22:                utility = utility − η_Suv[S_n].penalty
23:                return G_S if utility < Γ_U
24:            end for
25:            connect, penalty = connectSuperEdge(S_uv, S_uv, G, edgeIS)
26:            if connect is true then
27:                E_S = E_S ∪ (S_uv, S_uv)
28:            end if
29:            utility = utility − penalty
30:        end if
31:    end while
32:    return G_S = (V_S, E_S)
33: end procedure
```

the penalty to connect them with a superedge, and 2) $nseCost$: the penalty to not connect them. If connectivity is deemed beneficial—i.e., $seCost < nseCost$—then S_{uv} and S_n are connected through a superedge. Subsequently, the utility is updated by subtracting the corresponding penalty values and all the previous connections between (S_n, S_u) and (S_n, S_v) are removed from $\mathscr{G}_S$. This particular way of connectivity decision making guides the summarization algorithm so as to maximize the utility of $\mathscr{G}_S$ at each summarization step. Similarly, the decision to self-connect a supernode S_{uv} or not is made based on utility maximization: if a self-loop is deemed beneficial, then a self-connection (S_{uv}, S_{uv}) is added to the set of superedges $\mathscr{E}_S$. In the next iteration, the node pair from H with the next lowest importance score is evaluated. The algorithm terminates and returns the final $\mathscr{G}_S$ when the current utility of $\mathscr{G}_S$ satisfies the utility threshold Γ_U or when all node pairs have been evaluated.

Superedge Connectivity Decision Making. Let us discuss the details of the procedure *connectSuperEdge*(...), as shown in Algorithm 3. As mentioned before, this procedure returns true if connecting two given supernodes by a superedge is beneficial in terms of utility, or false otherwise. The benefit is defined as the minimum penalty that paid (lost utility) when a particular action is performed. In our case, there are two possible cases to evaluate, 1) connecting two supernodes S_u and S_v by a superedge $(S_u, S_v) \in \mathscr{E}_S$, and 2) not connecting the supernodes $(S_u, S_v) \notin \mathscr{E}_S$. Note that, the two supernodes in question can be the same (see line 26, Algorithm 2), in this case, we evaluate an action of self-connecting the given supernode with a superedge (self-loop). Let's understand the implications of each of the actions below:

• **Case 1**: $(S_u, S_v) \in \mathscr{E}_S$ When two given supernodes S_u and S_v are connected by a superedge, it induces all-to-all connection $\prod_{S_u,S_v}$ between the set of base nodes contained in S_u and S_v (per the encoding rules of Figure 1). Consequently, apart from original cross edges $A_{u,v} \subset E$ in the G, we are introducing an additional set of spurious edges $\{\prod_{S_u,S_v} - A_{S_u,S_v}\}$ between the set of nodes contained in S_u and S_v. Essentially, at this step, reconstruction of G from the current $\mathscr{G}_S$ (as discussed in Section 3) would introduce $|\prod_{S_u,S_v} - A_{S_u,S_v}|$ number of spurious edges as a result of the current action. Additionally, we know that for each introduced spurious edge the utility is penalized by an amount $\frac{1}{(\binom{|V|}{2})-|E|)}$. Let *seCost* be the total penalty or cost associated with the action of connecting supernodes S_u and S_v.

• **Case 2**: $(S_u, S_v) \notin \mathscr{E}_S$ We know that if supernodes S_u and S_v are not connected, then we are missing the set of A_{S_u,S_v} original edges. In other words, reconstruction of G from the current $\mathscr{G}_S$ would have deleted $|A_{S_u,S_v}|$ number of edges that existed in G. Since the importance score of each edge e in G is given by $edgeIS[e]$, for each missing edge e the utility has to be penalized by an amount of $edgeIS[e]$. Let *nseCost* be the total penalty associated when an action of not connecting supernodes S_u and S_v is performed.

Finally, if $seCost > nseCost$, then the benefit of not connecting the given supernodes S_u and S_v is higher and vice versa[2].

Incremental Utility Calculation. To accurately calculate the utility at each iteration in an incremental fashion, we need to keep track of all actions and related penalties that have been imposed in previous iterations. This bookkeeping is explicit, to avoid redundant penalization of the utility at each iteration. For example, let's say we are evaluating the action of connecting two supernodes S_u and S_v by a superedge. Performing this action equates to the introduction of one or more spurious edges in the underlying graph between the sets of base nodes contained in S_u and S_v. In principle, we must penalize the utility for the introduced spurious edges. However, it may be the case that in previous summarization steps, the utility has already been penalized for spurious edges that we are considering in the current step. Thus, we need to keep track of spurious edges that we have penalized the utility for at each iteration. On the other hand, when we are evaluating $(S_u, S_v) \in \mathscr{E}_S$, we need not penalize for the original cross edges A_{S_u,S_v} between S_u and S_v. However, in previous iterations, some finalized action might have penalized the utility for some or all of these original edges $\subset E$. Thus, we need to rollback the penalty of these edges in the current action. This indicates that we need to keep track of original edges as well as spurious edges that we might have penalized the utility for in previous iterations. Accordingly, the amount of bookkeeping needed is in the order of $O(\binom{|V|}{2})$. This large space requirement makes it impractical to use any kind of deterministic data structure (list, hash table, hash set, etc.) for the purposes of bookkeeping. Instead, we need a more space-efficient data structure to keep track of processed edges.

Algorithm 3 Utility-Driven Superedge Connectivity Decision Maker and Incremental Utility Calculator

```
1: procedure CONNECTSUPEREDGE(V_{S_w}, V_{S_n}, G, edgeIS)
2:     Initialize: penalty = 0; seCost = 0; nseCost = 0; decision =
       false; CF = (cap, bSize, fSize); cf_se^+ = ∅; cf_se^- = ∅; cf_nse^+ = ∅; cf_nse^- = ∅
3:     totalSE = (|V| choose 2) − |E|
4:     for u ∈ V_{S_w} do
5:         for v ∈ V_{S_n} do
6:             if u ≠ v and (u,v) not seen before then
7:                 e = (u,v)
8:                 if e ∈ E and e ∈ CF then
9:                     seCost = seCost − edgeIS[e]
10:                    cf_se^- = cf_se^- ∪ e
11:                else if e ∉ E and e ∈ CF then
12:                    nseCost = nseCost − 1/totalSE
13:                    cf_nse^- = cf_nse^- ∪ e
14:                else if e ∈ E and e ∉ CF then
15:                    nseCost = nseCost + edgeIS[e]
16:                    cf_nse^+ = cf_nse^+ ∪ e
17:                else if e ∉ E and e ∉ CF then
18:                    seCost = seCost + 1/totalSE
19:                    cf_se^+ = cf_se^+ ∪ e
20:                end if
21:            end if
22:        end for
23:    end for
24:    if seCost < nseCost then
25:        penalty = seCost
26:        CF.insert((u,v)), for all (u,v) ∈ cf_+^se
27:        CF.delete((u,v)), for all (u,v) ∈ cf_-^se
28:        decision = true
29:    else
30:        penalty = nseCost
31:        CF.insert((u,v)), for all (u,v) ∈ cf_+^nse
32:        CF.delete((u,v)), for all (u,v) ∈ cf_-^nse
33:        decision = false
34:    end if
35:    return (decision, penalty)
36: end procedure
```

Probabilistic Data Structures to the Rescue. A Bloom filter is a potential option as it is a space-efficient data structure that can be used to keep track of already processed edges. Processed edges marked in the Bloom filter indicate that the utility has been (potentially) penalized for these edges. As discussed before, often certain penalties for already processed edges need to be rolled back. This implies that these edges should be deleted from the Bloom filter in such situations. Unfortunately, the standard Bloom filters do not support deletion of items. However, certain variants of Bloom filter such as counting Bloom filter allow both addition and deletion of items, but with significant space overhead. In fact, counting Bloom filters [9, 8] are known to use 3–4× space to retain the same false positive rate as a space-optimized Bloom filter. Fan et al., [8] introduced Cuckoo filters (*CF*). *CF* possess the dual advantage of space efficiency as well as the ability to handle deletion of items. Given their advantages, we make use of *CF* to manage the bookkeeping of processed edges and corresponding rollbacks.

Over-Optimism in Utility. We know that probabilistic data structures suffer from the problem of false positives—i.e., they may identify an item as a set member even though it is not. Cuckoo filters allow the false positive rate to be controlled by varying the capacity and fingerprint size [8]. Because of false positives introduced by *CF*, there is a possibility of unwarranted optimism in the calculation of utility. From Algorithm 3, we know that cf_{se}^- is the set of original edges already processed in previous steps as confirmed by *CF*, and cf_{nse}^+ is the set of original edges that are yet to be evalu-

[2]Note that our algorithm can be modified slightly to provide k-anonymity guarantees [12] under favorable conditions. A supernode comprising of k nodes will be k-anonymous—and the supernode comprising of the minimum number original nodes can be considered an anonymity lower bound.

ated. Whereas, cf^-_{nse} is the set of spurious edges already evaluated in previous steps as confirmed by CF, and cf^+_{se} is the set of spurious edges yet to be processed. We analyze two specific cases.

In the case where $(S_u, S_v) \in \mathscr{E}_S$, we connect the given supernodes S_u and S_v with a superedge. This action introduces spurious edges between the nodes in the given supernodes. We denote this set of spurious edges as $\{\prod_{S_u,S_v} - A_{S_u,S_v}\}$. The set of spurious edges cf^+_{se} that are yet to be evaluated is calculated as $\{\prod_{S_u,S_v} - A_{S_u,S_v} - cf^-_{nse}\}$.

We want to penalize the utility for extra spurious edges that have been unprocessed in previous iterations. In addition, we need to rollback penalties for the original cross edges that were processed in previous iterations for which the utility has already been penalized. The total penalty *seCost* is calculated by subtracting the total cost of edges in cf^-_{se} from the total cost of edges cf^+_{se}:

$$seCost = \left(\frac{|cf^+_{se}|}{\binom{|V|}{2} - |E|}\right) - \left(\sum_{e \in cf^-_{se}} edgeIS[e]\right) \tag{4}$$

The current utility at this step is calculated as $utility = utility - seCost$.

Theorem 4.2 If fpr is the false positive rate of CF and if $(S_u, S_v) \in \mathscr{E}_S$, then we have upper bound on utility over estimation δ_{se} where

$$\delta_{se} \leq \frac{|\prod_{S_u,S_v} - A_{S_u,S_v} - cf^-_{nse}|}{\binom{|V|}{2} - |E|} \times \frac{fpr}{1 - fpr} \tag{5}$$

PROOF. By the definition of the false positive rate, we know that

$$fpr = \frac{|false\ positives|}{|false\ positives| + |true\ negatives|}$$

From this we can derive an expression for $false\ positives$ in-terms of fpr and $true\ negatives$. Also, let $|det^+_{se}|$ be the set of spurious edges that are yet to be evaluated, and $|det^-_{se}|$ be the set of original edges already processed in previous steps as confirmed by a deterministic data structure (e.g., Hash Table). We know that cf^+_{se} and cf^-_{se} are calculated based on a probabilistic data structure, in our case, a Cuckoo Filter. Therefore, the utility over-estimation is the difference between *seCost* calculated based on the deterministic and probabilistic data structures.

$$\begin{aligned}\delta_{se} &= seCost^{\triangle} \\ &= \left(\frac{|det^+_{se} - cf^+_{se}|}{\binom{|V|}{2} - |E|}\right) - \left(\sum_{e \in \{det^-_{se} - cf^-_{se}\}} edgeIS[e]\right)\end{aligned}$$

To find the upper bound, we need to find the maximum value of $seCost^{\triangle}$ or minimize $\left(\sum_{e \in \{det^-_{se} - cf^-_{se}\}} edgeIS[e]\right)$. We know that we need at least one edge between the nodes in S_u and S_v to connect these supernodes with a superedge. Let's consider a single edge connecting S_u and S_v and let ε be the importance score of this edge. For a given original graph of large size, the value of ε can be close to zero and we can safely ignore it. So we have:

$$\begin{aligned}\delta_{se} &\leq \frac{|det^+_{se} - cf^+_{se}|}{\binom{|V|}{2} - |E|} - \varepsilon = \frac{|det^+_{se} - cf^+_{se}|}{\binom{|V|}{2} - |E|} \\ &\leq \frac{|false\ positives|}{\binom{|V|}{2} - |E|} = \frac{|true\ negatives|}{\binom{|V|}{2} - |E|} \times \frac{fpr}{1 - fpr}\end{aligned}$$

Here $true\ negatives$ is nothing but the set of spurious edges yet to be evaluated (i.e., cf^+_{se}) and we know that $cf^+_{se} = \{\prod_{S_u,S_v} - A_{S_u,S_v} - cf^-_{nse}\}$. Thus, we have an upper bound for the utility overestimation. ■

Similarly, in the case of $(S_u, S_v) \notin \mathscr{E}_S$, we can calculate the utility over estimation by analyzing the cost of not connecting any S_u, S_v.

In summary, use of CF for the purpose of incremental utility calculation can result in over-optimism because of false positives. However, with the careful selection of capacity and fingerprint size of the CF, fp can be made sufficiently small. Subsequently, utility over-estimation becomes almost negligible. Essentially, increasing capacity improves the occupancy of a cuckoo hash table whereas increasing fingerprint (hashes) size rejects more false queries, thereby reducing fp but with the caveat of increased space overhead.

Time Complexity Analysis. Since the calculation of importance scores (Algorithm 2, line 3) depends on the choice of underlying centrality algorithm, we will focus on the time complexity of the iterative node merging algorithm (lines 6–32). In each merge step, for each potential neighbor of merged supernode $\mathscr{O}(d_{av})$, we evaluate connectivity between merged supernode and its potential neighbor $\mathscr{O}(|V|^2)$. Therefore, the overall complexity of each merge step comes out to $\mathscr{O}(|V|^2\ d_{av})$, where d_{av} is the average degree.

Limitations. The key limitation of Algorithm 2 is that it does not scale well for large graphs. This is because node merging and superedge decision making (lines 6–32) are exhaustive in nature and perform redundant computations. For example, consider Figure 5(a) that shows a portion of the base graph where nodes a, b, and c are more densely connected to nodes $1, 2, 3$, and 4 in comparison to node set e, f, g. Figure 5(b) shows an iteration of graph summarization where three supernodes $S_1 = \{a, b, c\}, S_2 = \{e, f, g\}$ and $S_3 = \{1, 2, 3, 4\}$ are formed. In this iteration, supernodes S_1 and S_2 are evaluated against S_3 for connectivity. Total 12 comparisons (denoted by com(S_1, S_3)) are made to decide connectivity between S_1 and S_3 and 12 comparisons are performed for S_2 and S_3. Also, 3 comparisons each are made to decide self-connectivity for supernodes S_1 and S_2. So in total 30 comparisons are made for the case shown in Figure 5(b). However, in the next iteration (Figure 5(c)), we are merging S_1 and S_2 to form supernode w. In order to evaluate connectivity between w and S_3 we perform 24 redundant comparisons between the nodes contained in supernode w and nodes in S_3, that have already been performed in the previous iteration. Even to decide the self-connectivity of w, many (9) redundant computations are performed. In total, we count 33 redundant comparisons that could have been avoided if we were to reuse previous computations. This insight leads us to a more efficient approach, discussed next.

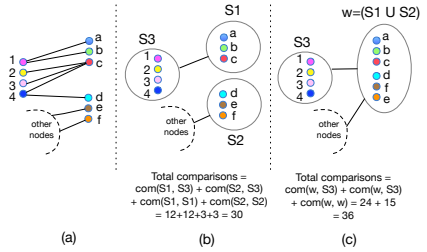


Figure 5: Example illustrating redundant computations (a) Portion of original graph, (b) Portion of graph summary showing superedge decision making between supernodes (S_1, S_3), (S_2, S_3) and self-connections, (c) Portion of graph summary showing superedge decision making between supernodes (w, S_3), (w, S_3) and self-connections

4.2 Memoization based Approach

To overcome scalability challenges, we introduce a memoization technique as a scalable approach to UDS. The key goal is to compute graph summaries and perform incremental utility calculation by reusing previous computations. Initially, each node and edge in the base graph G is its own supernode and superedge in the summary graph $\mathscr{G}_S$. We start by defining three variables for each superedge $(S_a, S_b) \in \mathscr{E}_S$ in $\mathscr{G}_S$: $seCost(S_a, S_b)$, $nseCost(S_a, S_b)$ and $(S_a, S_b)_{exist}$. Because S_a and S_b are already connected, the value of $seCost(S_a, S_b)$ is initialed to 0 (for all superedges). Also, initially

when $S_a = \{a\}$ and $S_b = \{b\}$, not deciding to connect a superedge between supernodes S_a and S_b incurs a cost of $edgeIS[(S_a, S_b)]$. Therefore, *nseCost* for all superedges is initialized to the corresponding $edgeIS[e]$ values. Whereas, $(S_a, S_b)_{exist}$ indicates if a given superedge is permanent (with value of 1) or ephemeral (value of 0). An ephemeral superedge indicates that we have not decided to connect the two given supernodes based on the result of a superedge decision-making process, while a permanent superedge indicates the opposite. The key advantage of an ephemeral superedge is that it provides a low-cost way to store calculated penalty costs for both connecting and not connecting a particular superedge. Although an ephemeral superedge is not considered a real edge, it helps us judiciously re-use the pre-computed penalty costs stored in it for upcoming cost computations. Initially, all the superedges are permanent, therefore, the value of $(S_a, S_b)_{exist}$ for all edges $(S_a, S_b) \in \mathscr{E}_S$ is set to 1. Initialization of all the superedge variables with required conditions is shown in Equation 6.

$$\left.\begin{aligned} seCost(S_a, S_b) &= 0 \\ nseCost(S_a, S_b) &= edgeIS[(S_a, S_b)] \\ (S_a, S_b)_{exist} &= 1 \end{aligned}\right\} \textbf{if} \begin{cases} (a,b) \in E, \\ S_a = \{a\}, S_b = \{b\}, \\ (S_a, S_b) \in \mathscr{E}_S \end{cases} \tag{6}$$

After initialization, in the upcoming iterations, connectivity costs *seCost* and *nseCost* can be calculated by reusing costs calculated from previous iterations as shown in Equations 7 and 8. For instance, let's say at iteration t we are evaluating connectivity between supernodes S_u and S_w and calculate utility penalty costs $seCost(S_u, S_w)$ and $nseCost(S_u, S_w)$. Let's say we decided not to connect S_u and S_w because *nseCost* is less than *seCost*. At this point, the current utility is calculated as $utility = utility - nseCost(S_u, S_w)$. So in the summary graph we connect an ephemeral edge between the given supernodes and set $(S_u, S_w)_{exist} = 0$. In a particular future iteration $t + k$ where $k \geq 1$, if we want to calculate the cost to connect supernodes S_u and S_w, then we need to nullify the previously subtracted penalty for disconnecting the given supernodes in the iteration t. More formally, $seCost(S_u, S_w)$ at iteration $t + k$ is calculated by reusing previous computations as $seCost(S_u, S_w)^{t+k} = seCost(S_u, S_w)^t - nseCost(S_u, S_w)^t$. However, if supernodes S_u and S_w were never evaluated before for connectivity, then *seCost* is calculated by estimating the penalty for introducing spurious edges across the nodes contained in the given supernodes. In similar essence, *nseCost* is calculated by reusing previously computed values as shown in Equation 8.

$$seCost(S_u, S_w) = \left.\begin{cases} seCost(S_u, S_w) - nseCost(S_u, S_w)\} \textbf{ if } \begin{cases} (S_u, S_w) \in \mathscr{E}_S, \\ (S_u, S_w)_{exist} = 0 \end{cases} \\ \dfrac{|S_u| \times |S_w|}{\binom{|V|}{2} - |E|} \end{cases}\right\} \textbf{if} (S_u, S_w) \notin \mathscr{E}_S \tag{7}$$

$$nseCost(S_u, S_w) = nseCost(S_u, S_w) - seCost(S_u, S_w)\} \textbf{ if } \begin{cases} (S_u, S_w) \in \mathscr{E}_S, \\ (S_u, S_w)_{exist} = 1 \end{cases} \tag{8}$$

Given the values of penalties calculated in the previous iterations for supernode pairs (S_u, S_w) and (S_v, S_w), we calculate utility penalties for supernode pair (S_{uv}, S_w) using Equations 9 and 10. Here, S_{uv} is the supernode obtained by merging supernodes S_u and S_v. For example, the *seCost* of connecting a merged supernode S_{uv} with an existing supernode S_w is calculated by adding the individual costs of (S_u, S_w) and (S_v, S_w). Similarly, we compute *nseCost* of (S_{uv}, S_w) by easily reusing individual costs of (S_u, S_w) and (S_v, S_w)

$$seCost(S_{uv}, S_w) = seCost(S_u, S_w) + seCost(S_v, S_w) \tag{9}$$

$$nseCost(S_{uv}, S_w) = nseCost(S_u, S_w) + nseCost(S_v, S_w) \tag{10}$$

Once we have the individual penalty costs of evaluating connectivity between merged supernode S_{uv} and its potential neighbors, then the total penalty cost of merging any two supernodes S_u and S_v is calculated by summing the corresponding individual costs of evaluating connectivity of S_{uv} with its potential neighbors. Equations 11 and 12 show the calculations.

$$seCost(S_{uv}) = \sum_{\substack{w \in \eta_m \\ \forall m \in \{S_u \cup S_v\}}} seCost(S_{uv}, S_w) \tag{11}$$

$$nseCost(S_{uv}) = \sum_{\substack{w \in \eta_m \\ \forall m \in \{S_u \cup S_v\}}} nseCost(S_{uv}, S_w) \tag{12}$$

Finally, the utility penalty or costs associated with merged supernode S_{uv}'s self-connectivity decision making can also be calculated by adding pre-computed costs of S_u's self-connectivity (S_u, S_u), S_v's self-connectivity, and the cost associated in evaluating supernode pair (S_u, S_v). Calculations are shown in Equations 13 and 14.

$$seCost(S_{uv}, S_{uv}) = seCost(S_u, S_v) + seCost(S_u, S_u) + seCost(S_v, S_v) \tag{13}$$

$$nseCost(S_{uv}, S_{uv}) = nseCost(S_u, S_v) + nseCost(S_u, S_u) + nseCost(S_v, S_v) \tag{14}$$

In summary, given a newly merged supernode S_{uv} and its potential neighbor S_w, to evaluate connectivity between them, we reuse previous computations between S_u, S_w and S_v, S_w as opposed to redundantly performing comparisons between base nodes contained in S_{uv} and S_w, as done in the previous approach (Section 4.1). As shown in Algorithm 2 (lines 12–16), we do this for all potential neighbors. As a result, by avoiding redundant computations, we have effectively reduced the complexity of each merge step from $O(|V|^2 d_{av})$ in the previous approach, to $O(d_{av})$ in the current approach. Also, by storing penalty costs (*seCost* and *nseCost*) for each superedge and using the concept of ephemeral edges we have introduced an extremely low-overhead way to keep track of the penalty costs for all the pairs of supernodes—whether decided to connect them or not. These stored penalty costs are used to efficiently calculate costs for upcoming computations.

Discussion: While memoization reduces the time complexity of each merging step, the time complexity of computing the importance scores can be still high. We improve the performance of this step by making use of the fast approximation algorithms for centrality calculation. For example, considering betweenness centrality based utility function, we make use of an approach that uses random sampling of shortest paths to estimate centrality values for all the nodes/edges [32]. Algorithm runs in the order of $O(|E|)$ per sample and interestingly, the number of samples needed to compute a good approximation to all vertices is a constant and independent from G.

Finally, we note that the techniques proposed in this paper are not just limited to un-directed, and un-weighted graphs. For instance, calculation of importance scores can be easily adapted to directed/weighted graphs as centrality computing algorithms exist for directed, weighted graphs as well. On the other hand, in the grouping step, node pair candidates to merge at each step can also be picked based on directions. For instance, if in a directed graph we have directed edges (a $\longrightarrow$ b) and (a $\longrightarrow$ c) then b and c can be one such candidate pair to merge. Also, since our utility function depends on calculation of importance scores for nodes and edges, it naturally adapts to weighted graphs.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Settings

Setup. We perform all our experiments on single Amazon EC2 m4.4xlarge instance with 16 vCPU, 64 GB memory, and 300 GB SSD storage. We use Python and create graphs using the Networkx [26] library. For certain scalable centrality implementations we rely on the networkit [34] library. To scale for the large datasets that barely fit in the memory, we made several programatic improvements to our code. For example, we carefully parallelized the loop

(lines 12–16, Algorithm 2). Also, we modified Networkx library to support external memory graph access (read and write). Specifically, we extend Networkx by subclassing the Graph class and providing user-defined factory functions. These functions query a database and cache the results in the dictionaries used by Networkx.

Datasets. In our experiments we make use of seven real-world undirected and un-labeled graph datasets. Among them, ca-GrQc, ca-AstroPh, ca-HepTh, and ca-HepPh are author collaboration networks from the e-print arXiv for Astrophysics, High Energy Physics, High Energy Physics Theory, and General Relativity categories. The dataset com-Amazon has connection between any two products if they co-purchased. Whereas LiveJournal and Friendster are online blogging and gaming networks. All the datasets can be downloaded from [19]. Table 2 presents the datasets and their properties such as size, average degree (Avg. Deg.), density, average clustering coefficient (Cl. Co.), number of connected components (CCs), and size of largest component (LC).

Table 2: Real world graph datasets

Dataset	Nodes	Edges	Avg. Deg.	Density	Cl. Co.	CCs	LC
ca-GrQc	5242	14496	5.526	0.1054%	0.5296	355	79.32%
ca-HepTh	9877	25998	5.259	0.0533%	0.4714	429	87.46%
ca-HepPh	12008	118521	19.73	0.1644%	0.6114	278	93.30%
ca-AstroPh	18,772	198110	21.10	0.1124%	0.6306	290	95.37%
com-amazon	334863	925872	5.529	0.0017%	0.3967	1	100.00%
com-LiveJournal	4036538	34681189	17.18	0.0004%	0.2815	38577	99.04%
com-Friendster	65608366	1806067135	55	8.4e-05%	0.1623	1	100%

Baselines. We closely study two key works in literature that provide iterative solutions for grouping-based greedy graph summarization. First is the work by Navlakha et al. [25] and second is by Tian et al. [23]. Because [23] builds on [25] and provides the distributed solution for it, we implement algorithm discussed in [25] as a baseline. We have added experimental results in Section 5.2 (Figure 7) comparing our results with state-of-the-art grouping-based summarization technique by Navlakha et al. [25]. According to this technique, the best pair of nodes is selected at each step on the basis of maximum gain. Gain is defined as the extent of compression achieved when the selected pair of nodes are merged. To scale this technique, authors select a node u at random and a neighbor v within 2-hops is selected that achieves maximum gain when merged with u. This is repeated until the required compression is achieved. Since this technique is based on the theory of Minimum Descriptive Length, we refer to this technique as MDL in our experiments. Next, we highlight the key design decisions that we made in our technique and replace each design decision with its random counterpart to create our other set of baselines. We make two key design decisions in our technique; first, we compute relative importance scores for nodes and edges using the shortest path betweenness centrality metric. Second, we select a pair of 2-hop neighbors in the ascending order of the sum of their importance scores. We randomize these key steps by 1) randomly assigning importance scores to nodes and edges (RNEI), 2) selecting the pair of 2-hop nodes in random order (RNPO) while assigning importance scores using betweenness centrality, and 3) performing both steps randomly (RNEI-RNPO). For random baselines we report the average of ten runs.

Evaluation Metrics. We evaluate our techniques using two popular real-world applications, measure the application-specific utility, and compare with our baselines. For each application, we define a utility metric that will indicate the usefulness of a graph summary with respect to the corresponding application.

• **Application 1: Top-k Query.** One of the widely used real-world applications is the selection of top-k or top t% of nodes, where the goal is to rank nodes using the Pagerank algorithm and select the top k nodes according to their ranks, in descending order. Given the value of t, k is derived as $k = |V| * t\%$ for G. Whereas, for $\mathscr{G}_S$, $k = |\mathscr{V}_S| * t\%$. If we run Pagerank on both graph G and its summary $\mathscr{G}_S$ and $V_{t\%}$ be the set of top-k nodes in G based on Pagerank values, then the utility of $\mathscr{G}_S$ is defined as:

$$\text{Top-}k\text{ Query App Utility} = \frac{\sum_{v \in V_k} \frac{1}{|S_v|}}{k} \tag{18}$$

In other words, if all the top k or t% nodes from G match exactly with top-k nodes in $\mathscr{G}_S$ then the utility score in this case equals 1 where each node contributes 1 to the summation in the numerator for Equation 18 as for that node $|S_v| = 1$. On the other hand, in the case where some of the top-k nodes are contained within a supernode containing more than one nodes, then each such node u contributes a value of $\frac{1}{|S_u|}$. This fraction (that is < 1) represents the information loss caused by the summarization process.

• **Application 2: Link Prediction.** Another real-world application is knowing if a given pair of nodes belongs to the same community, or not. In other words, based on the current community structure, predicting if there will be a link between the given pair of nodes, or not. To measure the utility of $\mathscr{G}_S$, we consider a list of all pairs of 2-hop nodes in graph G. For each pair, we predict a link if the pair belongs to the same community in $\mathscr{G}_S$, and we compare the result with the link prediction on G. More formally, if L_S is the binary link prediction result vector for $\mathscr{G}_S$, where each element corresponds to a link prediction result for a pair belonging to all 2-hop pairs, and if L be the result vector, then utility of $\mathscr{G}_S$ is defined as:

$$\text{Link Prediction App Utility} = \frac{|L_S \cap L|}{|L|} \tag{19}$$

Table 3: Example test criteria

Desirable Property Example Test Criteria
C2: Let's consider graphs $m_X C_n, m_Y C_n$, and $m_Z C_n$ ($X < Y < Z$) from Figure 4. Let $m_Z C_n$ be the base graph and $m_X C_n$ and $m_Y C_n$ be perturbed graphs obtained by introducing X and Y number of spurious edges to $m_Z C_n$. Then according to **C2**: *spurious edge awareness* criterion, the following condition should satisfy: $$\left(EU(m_X C_n)_{m_Z C_n} - EU(m_Y C_n)_{m_Z C_n}\right) > 0 \tag{15}$$
C3: Consider weighted barbell graphs $w_s B_n$, $w_t B_n$ and mB_n. Here $w_s B_n$ is a barbell graph of size n with a weight of exactly one of the edges being s, and the weights on the rest of the edges being r, where $s > r$. In this case, let mB_n be a barbell graph with a removed heavy-weighted edge. If $s > t$, then according to **C3**: *weight awareness* criteria, the following should satisfy: $$(EU(w_t B_n)_{mB_n} - EU(w_s B_n)_{mB_n}) > 0 \tag{16}$$
C4: Consider graphs K_n, mK_n and C_n, mC_n from Figure 4. These four graphs are equally sized in terms of number of nodes, where C_n has relatively fewer edges when compared to K_n. Graph mK_n is obtained by removing a single edge from K_n, similarly mC_n is obtained by removing a single edge from C_n. Then, according to **C4**: *edge submodularity* criteria: $$(EU(mK_n)_{K_n} - EU(mC_n)_{C_n}) > 0 \tag{17}$$

Example Criteria for a Desirable Utility Function to Satisfy. In addition to the example test criteria described in Section 3, here we provide a list of more example criteria shown in Table 3. These example criteria based on model graphs in Figure 4 help us understand the properties defined in Table 1, and evaluate the desirability of a utility function. Note that these criteria are not exhaustive and other criteria can be devised using the model graphs in Figure 4.

5.2 Experimental Results

Current-flow and Shortest Path Betweenness Centrality-based Utility Function Satisfies All Desired Properties. We start by evaluating the suitability of various centrality metrics that can be used during the calculation of edge importance scores, and form a utility function that exhibits the desired properties described in Section 3. Generally, the relative importance of each edge in the graph G is

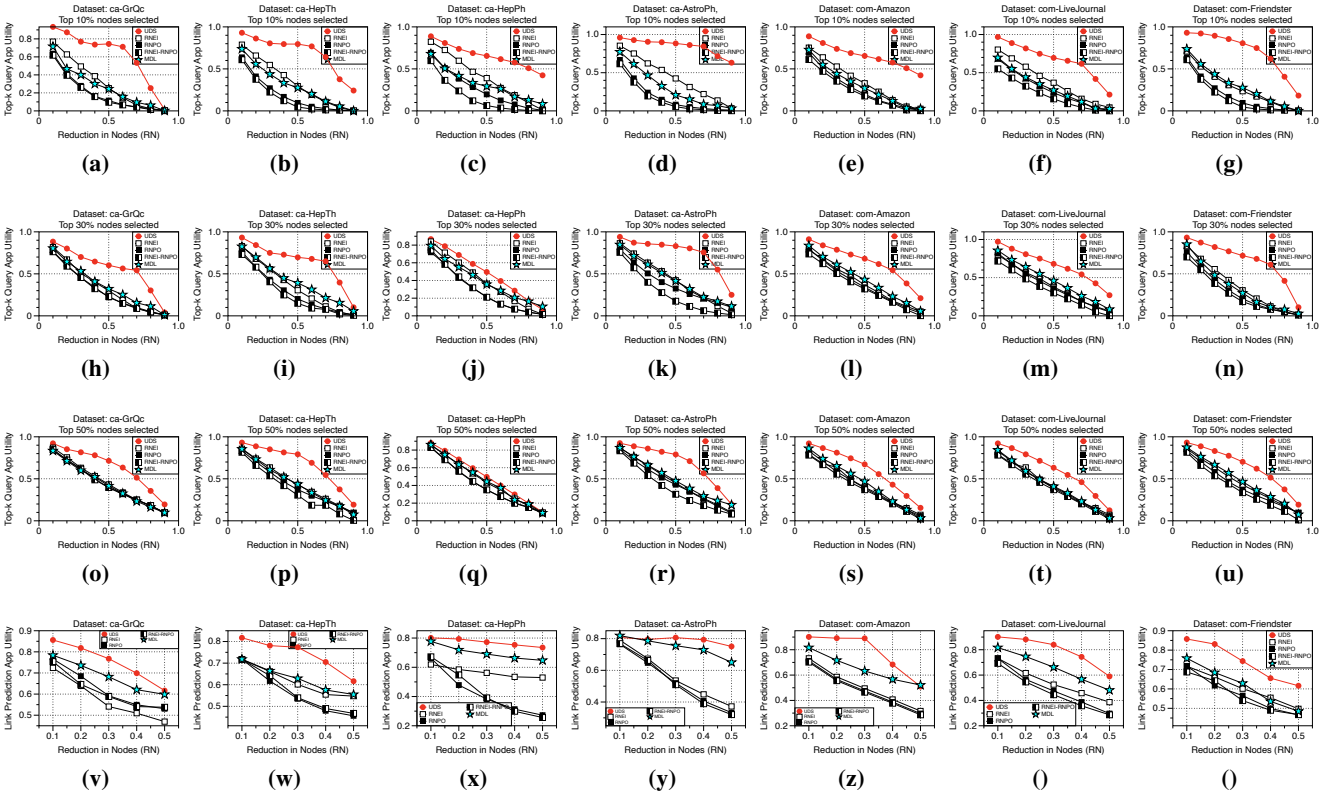


Figure 6: Experimental results demonstrating effectiveness of UDS design decisions

assessed by measuring the degree of participation of edges in communication between distinct parts of the network. This leads us to the notion of betweenness centrality. The most common betweenness centrality metric is based on shortest paths, where the centrality of an edge e is essentially an average number of shortest paths connecting all pairs of nodes in the graph that pass through edge e. There are some drawbacks with this approach. First, it takes into account only the shortest paths and ignores the slightly longer paths. Edges of such relatively longer paths are critical for communication in the network. Second, the actual number of shortest paths that lie between the source and destination is irrelevant. In our case, it is reasonable to consider the abundance and the length of all paths. Knowledge about the importance of each edge to the graph structure is enhanced when more routes are possible.

In order to take such paths into account, Current-Flow Betweenness centrality can be considered [2]. Here, the graph is imagined as a resistor network in which the edges are resistors and the nodes are junctions between resistors. Accordingly, the current-flow betweenness of an edge is the amount of current that flows through it, averaged over all source-destination pairs, when one unit of current is induced at the source and the destination (sink) is connected to the ground. Let's denote shortest-path and current-flow betweenness centrality-based graph utility functions as *SP-BCU* and *CF-BCU*.

Moreover, certain centrality metrics calculate centrality scores for nodes and cannot be directly used to calculate edge centrality—e.g., centrality metrics that are based on Pagerank, Eigenvector [27], Communicability [6], Communicability Betweenness [7], etc. Here, we treat the node centrality scores as node importance. Also, by intuition, we assign importance scores to edges based on the importance of the nodes they are connecting to—i.e., we assign an edge a high importance if it connects any two highly important nodes. Using these node-based centrality measures, we estimate an edge importance by summing up the normalized centrality scores of the pair of nodes it connects, and normalizing it. Let's denote utility functions based on these centrality metrics as *PRU*, *EVU*, *COU*, and *CO-BCU*. We compare these utility functions and evaluate their effectiveness using the model graphs (shown in Figure 4) and our ideal utility function properties. Table 6 demonstrates our evaluation results. Red cells or non-positive values indicate a violation of a corresponding property or criterion. Results show that *CF-BCU* and *BCU* obey all the formal required properties (C1-C4). Bold values represent max values that are highly discriminatory for each test criterion. We find *CF-BCU* to be most effective and highly discriminatory. Each row of the tables corresponds to a comparison between the similarities (or distances) of two pairs of graphs; pairs (A,B) and (A,C) for property (C1-C3); and pairs (A,B) and (C,D)

Table 4: Practicality of utility *EU* with respect to an application of top-*k* query

Application 1	Datasets													
	ca-GrQc		ca-HepTh		ca-HepPh		ca-AstroPh		com-Amazon		com-LiveJournal		com-Friendster	
Top % Nodes	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.	Pearson's r	Cos. Sim.
10	0.9475	0.9822	0.9569	0.9939	0.9453	0.9943	0.9835	0.9976	0.9329	0.9969	0.9289	0.9743	0.9448	0.9738
20	0.9232	0.9828	0.9709	0.9947	0.9438	0.9330	0.9398	0.9965	0.9628	0.9964	0.9519	0.9127	0.9474	0.9528
30	0.9403	0.9855	0.9561	0.9936	0.9488	0.9249	0.9654	0.9930	0.9794	0.9864	0.9832	0.9287	0.9527	0.9803
40	0.9505	0.9942	0.9565	0.9969	0.9428	0.9308	0.9925	0.9921	0.9877	0.9970	0.9328	0.9267	0.9378	0.9747
50	0.9280	0.9912	0.9864	0.9925	0.9426	0.9322	0.9987	0.9869	0.9893	0.9734	0.9737	0.9725	0.9735	0.9826

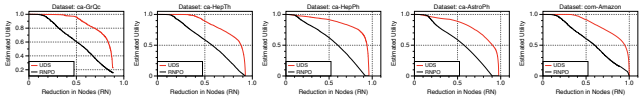
Table 5: Practicality of EU with respect to a link prediction application

Application 2	Datasets						
	ca-GrQc	ca-HepTh	ca-HepPh	ca-AstroPh	com-Amazon	com-LiveJournal	com-Friendster
Pearson's r	0.9424	0.9306	0.9950	0.9910	0.9259	0.9264	0.9371
Cos. Sim.	0.9657	0.9940	0.9999	0.9927	0.9863	0.9957	0.9873

for (C4). However, the calculation of current-flow betweenness centrality *CF-BC* is computationally intensive and does not scale even for graphs of a few thousand nodes. Hence in our experiments we use of shortest path betweenness centrality *SP-BC* that scales well for larger graphs. Also, from our results we note that, similar to *CF-BCU*, the utility function *SP-BCU* also exhibits all the desired properties. Moreover, it has been shown in [28] that compared to other centrality metrics, *SP-BC* is strongly correlated with *CF-BC*. Hence, it is beneficial to trade slight loss in quality to significant improvement in performance. Finally, we compare simple graph edit distance (GED) with our utility functions. Result shown in last column of Table 6 show that GED violates all the desired properties except C2. Hence GED is not fit as an utility function.

UDS Judiciously Exploits High Compressibility of the Graphs. In the first experiment, for each dataset we vary *RN* from 0.1 to 1 and apply UDS to analyze the incrementally calculated utility EU. Figure 7 shows the result. We compare the UDS approach with the random 2-hop pair selection for merging at each iteration (RNPO). There are three key takeaways from this experiment. First, the non-linear relationship between EU and *RN* (shown in red) for all the graphs indicates relatively high compressibility of corresponding graphs. Second, we observe that denser graphs have higher compressibility when compared to sparse graphs. Third, UDS smartly exploits compressibility of graphs by preserving important regions of the graphs at relatively higher *RN* when compared to the UDS approach with random 2-hop pair selection. We omit results for LiveJournal and Friendster datasets as we do not see different result.

UDS Design Decisions are Effective. Next, we evaluate the effectiveness of the two key design decisions that we make: 1) calculating relative importance score for nodes and edges, and 2) the order in which a pair of nodes is selected to merge in each summarization step. We compare UDS with the baselines using two real-world applications discussed in Section 5.1. In this experiment, we vary *RN* and calculate the application-based utility metric with UDS and related baselines where we randomize selected key design decisions. For the top-k or t% query application, we vary top % of nodes selected and measure application based utility for each dataset's summary. Figures (6a)–(6q) show that across various datasets, parameter values, and applications, UDS consistently results in graph summaries with significantly higher utility compared to baseline techniques—thus demonstrating the effectiveness of our design decisions. Figures (6a)–(6s) show results for the top-k query application. Figures (6v)–(6z) show results for the application of link prediction.



(a) **(b)** **(c)** **(d)** **(e)**

Figure 7: UDS judiciously exploits high compressibility of Graphs

Table 6: Evaluating various centrality metric-based utility functions and comparison with simple graph edit distance based utility (GED) metric

Test	Graphs				PRU	SP-BCU	EVU	COU	CO-BCU	CF-BCU	GED
Criteria	A	B	C		$EU(B)_A - EU(C)_A$						
C1	B10	mB10	mmB10		0.005	0.019	0.005	0.004	0.03	**0.13**	0
C1	L10	mL10	mmL10		-0.009	0.04	-0.03	-0.03	0.03	**0.3**	0
C1	BC11	mBC11	mmBC11		-0.009	0.00002	-0.007	-0.013	-0.005	**0.032**	0
C1, C4	WhB12	mWhB12	mmWhB12		-0.0002	0.04	0.001	-0.006	0.02	**0.063**	0
C1	WhB12	m2WhB12	mm2WhB12		-0.0003	0.08	0.003	-0.013	0.041	**0.127**	0
C2	m2C5	mC5	C5		0.023	0.023	0.023	0.023	0.023	0.023	**0.33**
C2	mmK5	mK5	K5		**0.5**	**0.5**	**0.5**	**0.5**	**0.5**	**0.5**	0.125
C2, C3	mB10	w2B10	w5B10		**0.095**	**0.095**	**0.095**	**0.095**	**0.095**	**0.095**	0
C2, C3	mB10	B10	w5B10		**0.132**	**0.132**	**0.132**	**0.132**	**0.132**	**0.132**	0

Test	Graphs				PRU	BCU	EVU	COU	CO-BCU	CF-BCU	GED
Criteria	A	B	C	D	$\delta = EU(B)_A - EU(D)_C$						
C4	K5	mK5	C5	mC5	0.1	0.099	0.1	0.1	0.1	**0.2**	0.1
C4	C5	mC5	mC5	m2C5	0.08	0.095	0.088	0.07	0.03	**0.142**	0.05

UDS Performs Well Compared to State-of-the-Art. Figure 6 shows the result of our experiments where we compare MDL approach with UDS with respect to top-k query and link prediction applications. Our approach consistently performs well when compared to MDL. We attribute this result to the fact that MDL approach does not optimize for the preservation the important regions of the graph as UDS does, hence tends to lose on summary quality.

UDS Provides Attractive Trade-Off Compared to LOPT. Theorem 1 implies that it is hard to find approximation factor or compare our solution empirically to global optimum. Nonetheless, we compare UDS where we pick best node pair to merge in each iteration in $O(1)$ time with local optimum LOPT where $O(N^2)$ comparisons are performed to pick best node pair to merge at each step. We present results in Tables 7 and 8. We perform two experiments. First, we compare UDS to LOPT with respect to the reduction in summary size relative to original size returned for a particular utility threshold. For this experiment, we generate Barabasi-Albert random graphs with parameters n (graph nodes) and p (preferential attachment). As we increase p from 1 to 5, density of graph increases. Table 7 shows that UDS performs very close to LOPT for sparser graphs and for denser graphs quality of solutions reduce by atmost 25% compared to LOPT. But for the given performance ($O(1)$ compared to $O(N^2)$ per iteration) UDS provides attractive trade-off compared to LOPT. Second, for a particular iteration, we compare UDS's choice of best pair to merge compared to LOPT's $O(N^2)$ choices (sorted in ascending order of cost). For example, Table 8 reports 0.1 if UDS's choice is within top 10% of LOPT's choices. Table 8 shows that across the iterations and for the random graphs (p=5, $utility$=0.5) of various sizes UDS's choice is within 10% of the LOPT's top choices.

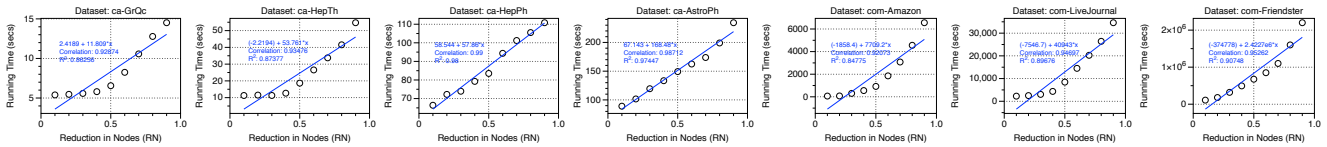
UDS's Estimated Utility (EU) is Practical. We compare EU with various application-specific utility values for varying *RN* to assess the practicality of EU to be used as an approximation for various application-specific utility metrics. For each real-world application, we calculate the Pearson's correlation coefficient r in order to measure the strength and direction of a linear relationship between EU and the application-specific utility, for varying *RN* from 0.1 to 1. Since EU values are in the range [0,1], we use Cosine similarity between EU and the application-specific utility in order to measure how closely related they are in magnitude. Table 4 shows the results for the application of top-k query where we can observe that correlation between Pearson's correlation coefficient r and EU is significant, with the value of r very close to 1 in almost all cases, and p-value $< 10^{-4}$. We observe similar result also for an application of link prediction as shown in Table 5. Hence, EU is practical.

UDS Scales Near Linearly with Varying *RN*. Figures (8a)–(8e) demonstrate that UDS performs well across all datasets, for uniformly increasing *RN*. Specifically, UDS exhibits near perfect linear scalability in the case of datasets with relatively higher density and average clustering coefficient (ca-HepPh and ca-AstroPh). On the other hand, in the case of relatively sparser datasets, *RN* values in the range 0.1 to 0.6 the cost of iteratively merging nodes remains relatively negligible when compared to the fixed cost of calculating node and edge importance. Overall, high correlation with linear best fit and R^2 values confirm our scalability conclusion.

UDS Visually Simplifies Complex Graphs with Guided EU. We also conduct visual validation of our UDS approach. For this experiment, we query graph summaries with a specified EU, rather than *RN*. Here, we visualize the ca-HepTh graph and its summaries for varying EU values. Figure (9a) shows the input graph. We can observe that the input is a disconnected graph with many small components and one large connected component. Figures (9b)–(9d)

Table 7: Comparing UDS to LOPT based on summary sizes for a given utility threshold.

Pref Attachment →	1						3						5					
Utility Threshold →	0.9		0.7		0.5		0.9		0.7		0.5		0.9		0.7		0.5	
Number of Nodes ↓	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT	UDS	LOPT
1000	0.805	0.975	0.90	0.97	0.95	0.97	0.30	0.40	0.50	0.75	0.72	0.92	0.12	0.27	0.36	0.55	0.67	0.80
5000	0.78	0.97	0.96	0.97	0.98	0.98	0.43	0.52	0.58	0.78	0.79	0.93	0.22	0.38	0.47	0.69	0.68	0.86
10000	0.91	0.99	0.94	0.99	0.98	0.99	0.48	0.57	0.66	0.80	0.77	0.92	0.31	0.40	0.51	0.67	0.66	0.84

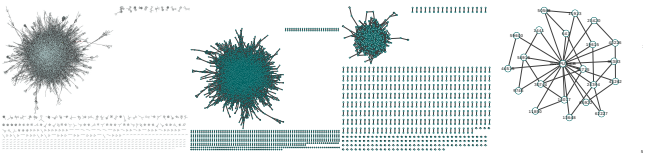


(a) **(b)** **(c)** **(d)** **(e)** **(f)** **(g)**

Figure 8: Scalability of UDS

Table 8: For a particular kth iteration, comparing UDS choice of best node pair to merge compared to LOPT top choices.

	k-th Iteration					
	1	10	20	30	40	50
Number of Nodes	UDS choice w.r.t LOPT Top % Choices					
1000	0.56	0.54	0.11	0.10	0.01	0.26
5000	0.84	0.71	0.54	1.31	0.86	0.36
10000	0.95	0.93	0.84	0.81	0.67	0.61



(a) Input **(b)** EU: 0.9 **(c)** EU: 0.5 **(d)** EU: 0.1

Figure 9: Visualization of graph ca-HepTh and its summaries for varying *utility EU*. For clarity, we ignore smaller CCs for (9d) .

show summaries with decreasing EU values from 0.9 to 0.1. We can observe from Figures (9b)–(9c) that, as the EU decreases the number of CCs increase. This happens because as RN increases by virtue of the decreasing EU, relatively lesser important parts of the graph are collapsed—and at some point they tend to get disconnected from the important regions because the UDS algorithm makes a decision that maintaining connectivity in that particular step is no more beneficial in terms of the graph's EU. For the sake of clarity, we ignore smaller CCs for Figure (9d) and only show largest CCs. Overall, we notice that as EU decreases, the largest component of the graph gets disentangled and becomes simpler.

6. RELATED WORK

Sparsification-based methods. Shen et al. [33] developed a tool called OntoVis that simplifies the underlying graph by relying on node filtering to understanding large social networks. Lin et al. [21] propose an unsupervised technique for egocentric information abstraction in heterogenous social networks where the key idea is to filter edges as opposed to nodes. They design criteria to distill important information to construct the abstracted graphs for visualization.

Sampling-based methods. Plethora of works [13, 17, 24, 11, 1, 37] focus on simplifying the underlying complex graph through sampling of nodes or edges from it. In summary, these techniques estimate the properties of the original graph, estimate relative frequencies of its substructures and then create a small sample subgraph that resembles the original graph. Also, there are techniques [10, 22] that use linear dimensionality reduction on the complex graph to generate simplified graph sketches or data synopses.

Grouping-based methods. The approaches [16, 36, 31] focus on the compression problem as a selection of supernodes, superedges to minimize the reconstruction error, while completely ignoring the preservation of important parts and regions of the graph. Purohit et al. [30] study the diffusion and propagation processes where they propose to merge two adjacent nodes such that the coarsened graph retains its diffusive properties. They do not consider the possibility of merging nodes that are not directly connected but share majority of the neighbors. Their limitation of not considering indirectly connected nodes prevents them from exploiting various opportunities to achieve high compression. Navlakha et al. [25] propose a highly compact two-part representation of a given graph consisting of a graph summary and a set of corrections. The corrections portion specifies the list of edge-corrections that helps to recreate the original graph. Using the concept of MDL (minimum description length) they try to create a graph summary with a minimal set of corrections. In other words, they are trying to minimize the reconstruction error of some form. Tian et al. [23] provide distributed systems solution for [25]. The solution that we propose in this paper can be classified into grouping-based methods.

Most of the techniques discussed above deal with minimizing reconstruction error without considering utility maximization. Only a handful of approaches consider preservation of utility that too in very specific scenarios. For example, Yan et al., [37] specifically focus on entity graphs with meaningful node, edge labels and sample important nodes (entities), relations to create a concise preview and utility is evaluated through human reviewers. We also note that none of the above discussed techniques deal with generating graph summaries with a user-specified utility threshold. Moreover, the majority of prior work evaluate their techniques with respect to a single application and do not demonstrate the effectiveness of their summaries to more than one real world application. Finally, we acknowledge Koutra et al's work [15] that partly inspired our work.

7. CONCLUSION

We strongly believe that given any complex, large graph, the ability to query a graph summary with a user-specified utility threshold has tremendous potential, and can find applications in a variety of use-cases. In this work, we present a novel approach to summarize a complex graph driven by the objective of maximizing the utility of the calculated graph summary. In doing so, we establish the theoretical foundations of governing the properties of an ideal utility function. We make theoretical connections to well known problems to prove inapproximability of the problem at hand. Subsequently, we propose a utility-driven summarization algorithm, and supplement it with scalable heuristics. Our iterative summarization technique allows a user to query a graph summary with a specified utility value. Finally, our experiments and evaluation results on multiple real-world datasets demonstrate the effectiveness of UDS both in terms of quality, performance, and overall practicality.

Acknowledgement. We thank reviewers and our colleagues Sandeep Bhatkar, and Matteo Dell'Amico for helpful reviews and comments.

8. REFERENCES

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *International Conference on World Wide Web*, *WWW*, pages 37–48. ACM, 2013.

[2] U. Brandes and D. Fleischer. Centrality measures based on current flow. In *Conference on Theoretical Aspects of Computer Science*, *STACS*, pages 533–544. Springer-Verlag, 2005.

[3] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Int. Res.*, 1(1):231–255, Feb. 1994.

[4] I. Davidson and S. S. Ravi. Intractability and clustering with constraints. In *International Conference on Machine Learning*, *ICML*, pages 201–208, New York, NY, USA, 2007. ACM.

[5] C. Dunne and B. Shneiderman. Motif simplification: Improving network visualization readability with fan, connector, and clique glyphs. In *Conference on Human Factors in Computing Systems*, *CHI*, pages 3247–3256. ACM, 2013.

[6] E. Estrada and N. Hatano. Communicability in complex networks. *Phys. Rev. E*, 77:036111, Mar 2008.

[7] E. Estrada, D. J Higham, and N. Hatano. Communicability betweenness in complex networks. 388, 05 2009.

[8] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *International Conference on Emerging Networking Experiments and Technologies*, *CoNEXT*, pages 75–88. ACM, 2014.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[10] M. Ghashami, E. Liberty, and J. M. Phillips. Efficient frequent directions algorithm for sparse matrices. In *International Conference on Knowledge Discovery and Data Mining*, *KDD*, pages 845–854. ACM, 2016.

[11] M. A. Hasan. Methods and applications of network sampling. In *Optimization Challenges in Complex, Networked and Risky Systems*, chapter 5, pages 115–139. 2016.

[12] M. Hay, G. Miklau, D. Jensen, D. Towsley, and C. Li. Resisting structural re-identification in anonymized social networks. *The VLDB Journal*, 19(6):797–823, Dec. 2010.

[13] C. Hübler, H. P. Kriegel, K. Borgwardt, and Z. Ghahramani. Metropolis algorithms for representative subgraph sampling. In *International Conference on Data Mining*, *ICDM*, pages 283–292. IEEE, 2008.

[14] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. Summarizing and understanding large graphs. *Stat. Anal. Data Min.*, 8(3):183–202, June 2015.

[15] D. Koutra, N. Shah, J. T. Vogelstein, B. Gallagher, and C. Faloutsos. Deltacon: Principled massive-graph similarity function with attribution. *TKDD*, 10(3):28:1–28:43, 2016.

[16] K. LeFevre and E. Terzi. Grass: Graph structure summarization. In *SDM*, pages 454–465. SIAM, 2010.

[17] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *International Conference on Knowledge Discovery and Data Mining*, *KDD*, pages 631–636. ACM, 2006.

[18] J. Leskovec and E. Horvitz. Planetary-scale views on a large instant-messaging network. In *International Conference on World Wide Web*, *WWW*, pages 915–924. ACM, 2008.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[20] C. Li, G. Baciu, and Y. Wang. Modulgraph: Modularity-based visualization of massive graphs. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, SA, pages 11:1–11:4. ACM, 2015.

[21] C. T. Li and S. D. Lin. Egocentric information abstraction for heterogeneous social networks. In *International Conference on Advances in Social Network Analysis and Mining*, *ASONAM*, pages 255–260, 2009.

[22] E. Liberty. Simple and deterministic matrix sketching. In *International Conference on Knowledge Discovery and Data Mining*, *KDD*, pages 581–588. ACM, 2013.

[23] X. Liu, Y. Tian, Q. He, W.-C. Lee, and J. McPherson. Distributed graph summarization. In *International Conference on Conference on Information and Knowledge Management*, *CIKM*, pages 799–808. ACM, 2014.

[24] A. S. Maiya and T. Y. Berger-Wolf. Sampling community structure. In *International Conference on World Wide Web*, *WWW*, pages 701–710. ACM, 2010.

[25] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *International Conference on Management of Data*, *SIGMOD*, pages 419–432. ACM, 2008.

[26] NetworkX developer team. Networkx. `https://networkx.github.io/`, 2014.

[27] M. Newman. *Networks: An Introduction*. Oxford University Press, Inc., 2010.

[28] M. J. Newman. A measure of betweenness centrality based on random walks. *Social Networks*, 27(1):39 – 54, 2005.

[29] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[30] M. Purohit, B. A. Prakash, C. Kang, Y. Zhang, and V. Subrahmanian. Fast influence-based coarsening for large networks. In *International Conference on Knowledge Discovery and Data Mining*, *KDD*, pages 1296–1305. ACM, 2014.

[31] M. Riondato, D. García-Soriano, and F. Bonchi. Graph summarization with quality guarantees. *Data Min. Knowl. Discov.*, 31(2):314–349, Mar. 2017.

[32] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *International Conference on Web Search and Data Mining*, *WSDM*, pages 413–422, New York, NY, USA, 2014. ACM.

[33] Z. Shen, K.-L. 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, Nov. 2006.

[34] C. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *CoRR*, abs/1403.3005, 2014.

[35] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *International Conference on Management of Data*, *SIGMOD*, pages 567–580. ACM, 2008.

[36] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *International Conference on Knowledge Discovery and Data Mining*, *KDD*, pages 965–973. ACM, 2011.

[37] N. Yan, S. Hasani, A. Asudeh, and C. Li. Generating preview tables for entity graphs. In *International Conference on Management of Data*, *SIGMOD*, pages 1797–1811. ACM, 2016.