Topological Graph Sketching for Incremental and Scalable Analytics

Bortik Bandyopadhyay, David Fuhry, Aniket Chakrabarti and Srinivasan Parthasarathy
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio, USA
{bandyopadhyay.14, fuhry.4, chakrabarti.14}@osu.edu, srini@cse.ohio-state.edu

ABSTRACT

We propose a novel, scalable, and principled graph sketching technique based on minwise hashing of local neighborhood. For an n-node graph with e-edges ($e \gg n$), we incrementally maintain in real-time a minwise neighbor sampled subgraph using k hash functions in $O(n \times k)$ memory, limit being user-configurable by the parameter k. Symmetrization and similarity based techniques can recover from these data structures a significant portion of the original graph. We present theoretical analysis of the minwise sampling strategy and also derive unbiased estimators for important graph properties such as triangle count and neighborhood overlap.

We perform an extensive empirical evaluation of our graph sketch and it's derivatives on a wide variety of real-world graph data sets drawn from different application domains using important large network analysis algorithms: local and global clustering coefficient, PageRank, and local graph sparsification. With bounded memory, the quality of results using the sketch representation is competitive against baselines which use the full graph, and the computational performance is often better. Our framework is flexible and configurable to be leveraged by numerous other graph analytics algorithms, potentially reducing the information mining time on large streamed graphs for a variety of applications.

1. INTRODUCTION

1.1 Motivation

The dramatic increase in the volume of graph data, and the pace of large graph data streams, poses significant challenges to traditional graph analysis methods when applied in real-time, besides making the analysis task compute and storage resource heavy. Modern approaches to handle this problem have seen wide use of sampling-based techniques [13, 17]. These techniques aim to extract a small subgraph, presenting a view of the original graph which can be used to efficiently approximate important graph properties with acceptable accuracy. In contrast to multi-pass or random-access models, for large graphs a single-pass edge streaming model is very desirable since significant I/O and network resources are needed to scan a large stored graph just once, and for large streaming graphs there may be no stored copy at all.

Pertinent questions that arise here are:
Q1: How can a large streaming graph be succinctly represented as a sketch, within user-controlled memory limit, using a one-pass efficient update strategy?
Q2: What theoretical properties of this sketch can be postulated to guide an intuitive understanding of this representation?
Q3: What properties of the original graph can be estimated with reasonable accuracy, using the information present in this compressed representation?
Q4: How can the same sketch be efficiently leveraged by existing graph analytic algorithms?

1.2 Our Contributions

To address these challenges and answer the above questions, we propose a novel framework built on the concepts of *Graph Sketching*. Based on scalable and principled *minwise hashing* [7] of local node neighborhoods, instead of maintaining a full adjacency list or sparse matrix graph representation, in Graph Sketching two data structures are maintained as part of the framework. For a (possibly streaming) input graph with n nodes, e edges and a user-chosen constant k, the first data-structure is an $(n \times k)$ *sketch matrix* M_{k}, and the second is an n-element neighbor count vector C. M_{k} stores an incrementally updatable minwise sampled neighborhood of each node of the entire graph using k ($\ll n$) independent linear minwise hash functions [6, 12], per-edge update time being $O(k)$. This sketch representation, capable of handling real-time edge streaming rate, lowers the memory requirement to $O(n \times k)$ instead of $O(e)$, making it particularly useful for streaming graphs commonly with $e \gg n$, with both n and e possibly unknown apriori.

Minwise independent permutation based hash functions have seen ubiquitous use in graph and network problems, in the context of dense subgraph (cluster) detection [8], community detection [23], graph sparsification [25] and computing various measures of interest like local triangle count [4] for large graphs. To the best of our knowledge, it's use has not been suggested as a fixed size sketch for streaming graph analytics, where different algorithms can be run on the derivatives of this generalized framework. In this work, we first show how to construct this sketch incrementally and efficiently via a minhash based smart buffering strategy and also provide some theoretical insights on the sampling in-

CIKM'16 , October 24 - 28, 2016, Indianapolis, IN, USA

ISBN 978-1-4503-4073-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2983323.2983735

duced subgraph. We then demonstrate how using such an in-core sketch one may compute various measures of interest (e.g. clustering coefficient) and even re-construct variants of the original graph using symmetrization and similarity based techniques, if needed, for downstream analysis by any state-of-the-art algorithms (eg. PageRank, clustering using METIS (v 5.0) [14]). Our results illustrate that minwise sampling preserves important topological properties of the graph with reasonable accuracy, while providing a compact yet flexible sketch representation within bounded memory.

1.3 Key Aspects of our Framework

Efficient Sampling: The proposed neighborhood sampling technique, while being principled and simple, can preserve various important properties of the graph, considerably reducing it's memory footprint. In section 7.4, we observe that for Twitter dataset with $d_{avg} = 1172.6$, about 85% of memory is saved with $k = 256$, while still approximately preserving the clustering of the dataset post-recovery.
Scalability: Empirically we show that, the best edge processing rate in our sequential framework for the streamed edge-list format is 205,980 edges/sec (using cit-Patents), more than 35-times faster than the real-time tweeting rate [22], with potential for further speed-up through parallelism.
Flexibility: A fixed-size data structure representing a node's approximate (sampled) neighborhood is maintained. This structure is efficiently updatable given only a single graph edge, enabling sketch construction from a single-pass edge streaming model. Our model can observe edges in an arbitrary order, an advantage over other streaming models which need to observe a node's entire neighborhood at once.
Configurability: Some data streams may contain arbitrary (non-integral, non-sequential) node names, which our framework can handle using a string to integer map for an additional O(n) space cost without affecting the asymptotics. This mapping separates the private information (like tweet text,user id etc.) from the structural information (like degree,clustering coefficient etc.) of the graph on the fly.
Information Richness: For any two nodes i and j, given that we can compute the degree of nodes i and j pre and post sampling, we can still do many probabilistic operations on their neighborhoods, as shown in section 5.2.2. Also, our framework has sufficient information to estimate many common *local similarity measures* as surveyed by Lü et al. [20].

2. BACKGROUND AND RELATED WORKS

2.1 Background

Minwise Hashing: Minwise Hashing, originally introduced by Broder et al. [7], is a well known technique for approximate estimation of Jaccard similarity between two sets and has seen wide usage in multiple different contexts. Our sampling strategy builds upon minwise hashing and hence we provide some background about it first. Let Ω be the universe and the sets X and $Y \subset \Omega$. Also, $\mathcal{F} : \Omega \to \Omega$ be a family of independent random permutations of Ω and let $\pi \in \mathcal{F}$ be one such random permutation. The exact Jaccard similarity between sets X and Y is defined as $sim_{XY} = \frac{|X \cap Y|}{|X \cup Y|}$. Minwise hashing provides a simple and scalable way to compute the approximate Jaccard similarity between any two sets, which are due to the below propositions:

PROPOSITION 1. *[7] When π is applied on X, then for any $x \in X$,*

$$P(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}$$

PROPOSITION 2. *[7] When π is applied on two sets X and Y, then Jaccard Similarity s_{XY} is given by*

$$s_{XY} = P(min\{\pi(X)\} = min\{\pi(Y)\})$$

PROPOSITION 3. *[7] An unbiased estimator $\widehat{s_{XY}}$ for s_{XY} under k permutations is*

$$\widehat{s_{XY}} = \frac{1}{k} \sum_{m=1}^{k} I[min\{\pi_m(X)\} == min\{\pi_m(Y)\}]$$

where $I(x) = 1$ if x is true and 0 otherwise.

To get a good estimate of the similarity, *a set of k independent permutations* are randomly chosen from $\mathcal{F}$. In practice, it is very expensive to generate a true independent permutation of the universal set, let alone doing it k times, on large real-world data sets. A scalable alternative is using linear hash functions, as proposed by Indyk [12] and Bohman et al. [6]. For sufficiently large prime p, such a linear *approximately minwise independent* family $\mathcal{F}_p$ of hash functions is defined as $\mathcal{F}_p = \{h_{a,b} : 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$, where $h_{a,b}(x) = (ax+b) \bmod p, \forall x \in X$ - the data set. Using k such independent hash functions $h_{a_m,b_m} = h_m \in \mathcal{F}_p$, $m \in [1,k]$, the *approximate* Jaccard similarity of any pair of sets X and Y can be computed as $(\frac{1}{k}\sum_{m=1}^{k} I[h_m(X) == h_m(Y)])$.
Proposed Minwise Neighborhood Sampling: Based on linear approximate minwise independent permutations as discussed above, we perform *incremental scalable neighborhood sampling of each node* such that *we retain all nodes of the graph and a subset of edges (neighbors) per node within a user-configurable fixed memory limit.* A node subset can also be obtained using minwise sampling, but it fails to succinctly capture neighborhood information of *all nodes* in the graph. We provide theoretical insights of our neighborhood sampling technique, which is bolstered by a variety of empirical evaluations to solve real world problems. Next, we discuss some related frameworks for representing streaming graphs.

2.2 Related Work

Graph Sketching in streaming context: Graph compression works [1, 9] focus on reducing the size of a graph, but assume the entire graph is available at compression time, or being restricted to a single application such as web search [26]. Other sampling and sketching works aim to preserve graph distances, cut properties and dense subgraphs, and to find small isomorphic subgraphs [5, 2, 3] by reducing dimensionality. While some sketching related works [30, 27] additionally incorporate the edge frequency information and allow queries in graph streams, we only focus on the topological property of streaming graphs. In context of our framework, few works using minwise hashing are relevant and discussed next.

Gibson et al. [8] has used linear minwise independent hash functions to construct shingle based fingerprint of large scale graphs, used the sketch to cluster (hierarchically) and detect link spams in web, but requires the adjacency list of each node to be presented compactly to the algorithm. Becchetti et al. [4] uses minwise hashing in semi-streaming context

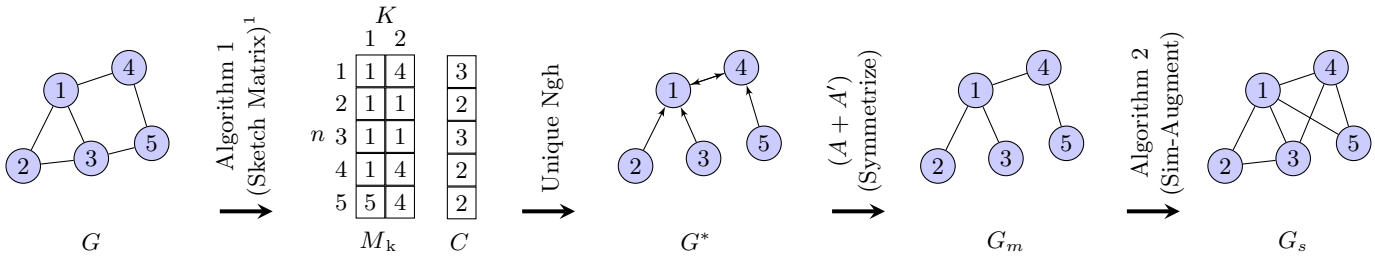


Figure 1: Toy example for Minwise Neighborhood Graph Sketching framework

to perform local triangle counting with provable theoretical bounds, but cannot generalize to other problems. Our framework incrementally constructs the minwise sketch matrix and degree of each node of the original graph, with no assumption on edge streaming order, consuming a fixed amount of memory controlled by the user. Using our proposed unbiased estimator in corollary 5.7 and the information present in our framework, local triangle counting can also be done as [4].

However, instead of solving a single problem using minwise hashing, we aim to provide a general framework. A direct derivative of our sampling strategy is the minwise sampling induced subgraph of the original graph, which is preprocessed (duplicate edges removed) and sparsified on-the-fly, with (presumably) better clustering property, while still retaining important topological measures with high quality. Using our framework, simple Horvitz-Thompson [11] based estimators can be constructed to estimate important properties of the original graph from this sampled subgraph, as shown in lemma 5.4, 5.5. The sampled subgraph can also be used by a wide variety of off-the-shelf (or slightly modified versions of) graph analytic algorithms.

Application algorithms: PageRank [21] and Clustering coefficients [29] are very important topological properties of graphs. Popular approaches to estimate triangle count or density of large streaming graphs include semi-streaming [4] and sampling [28] based techniques. We choose the *wedge sampling* based strategy introduced by Kolda et al. [16] to compute the Global Clustering Coefficient (GCC) and Local Clustering Coefficient (LCC) and show that such a scalable triangle counting algorithm is able to compute the clustering coefficients with acceptable accuracy. Satuluri et al. [25] introduce strategies for "edge sparsifying" a graph such that only edges with a high fraction of mutual neighbors are retained. We implement sparsification in our framework to cater to the user's need of a smaller subgraph from the "possibly larger" sampled graph, and can also sparsify incrementally while still lacking the complete graph. But differently, we show that we handle the opposite operation of recovering dropped edges using simple symmetrization and similarity based edge augmentation technique, which makes our framework more general and flexible.

In the rest of the paper, we provide an overview of our framework, formalize our methodology, provide theoretical insights about minwise sampling induced subgraph and then empirically evaluate the quality of off-the-shelf algorithms on the derivatives of the framework.

3. OVERVIEW

Notation	Description & Properties
$G = (V, E)$	Source graph of $n = \|V\|$ vertices and $e = \|E\|$ edges.
d_{avg}	Average degree of each node in $G = (V, E)$.
$\Gamma(v_i)$	Neighbors of vertex i in graph G.
k	User specified number of minwise hash functions.
$h_m(i)$	Hash value of i under minwise permutation m, $1 \leq m \leq k$, $1 \leq i \leq \|V\|$.
M_{k}	$(n \times k)$ Sketch Matrix minwise neighbor sampled from G.
C	Count vector containing the original degree of each node.
$G^* = (V, E^*)$	Directed Graph with unique neighbors from M_{k}.
$G_m = (V, E_m)$	Undirected Graph formed by symmetrizing G^*. $E^* \subseteq E_m \subseteq E$.
$G_s = (V, E_s)$	G_m augmented with similarity-induced edges. $\|E_s\| \geq \|E\|$.

Table 1: Notation

Table 1 summarizes notation used in the paper. Figure 1 shows a toy example of the minwise neighborhood sampling, graph construction, and edge recovery of our graph sketching framework. In the first step at the left, the undirected edges of source graph G are processed iteratively by Algorithm 1 to construct M_{k} and C. Each node i in G is represented by row i in M_{k}, which is a minwise sample of i's neighborhood. Next, unique neighbors of each node (row) in M_{k} form directed graph G^*, which is symmetrized using $(A + A')$-symmetrization [24] to generate G_m. In the last step, Algorithm 2 uses M_{k} and C to augment G_m with similarity induced edges thereby generating G_s. Not all steps are necessary, since as shown in our experiments, many algorithms can operate directly on the sketch and its derivatives.

We can see in Figure 1 that a small k will omit edges from the minwise sample, for example edge $(2, 3)$ present in G but not in G^* or G_m due to matrix row $M_{\text{k}}[2]$ lacking 3 and $M_{\text{k}}[3]$ lacking 2. The symmetrization strategy recovers some edges like $(1, 2)$ and $(1, 3)$ for node 1 and $(4, 5)$ for node 4 in G_m which are not present in directed G^*. It is possible to approximately recover some edges, shown as G_s in which edge $(2, 3)$ has been recovered due to the similarity of rows 2 and 3 in M_{k}. The recovery process can also introduce artifacts in G_s which were not in G, like edges $(3, 4)$ and $(1, 5)$, and it can fail to recover some edges like $(3, 5)$. Building on the basic toy example, in the next sections, we formally answer the four motivating questions to better explain our framework.

[1]In this example let the randomized h_1 permutation be $1, 5, 2, 4, 3$, that is, $h_1(1) < h_1(5) < \ldots < h_1(3)$, and h_2 permutation be $4, 1, 5, 2, 3$.

4. METHODOLOGY

4.1 Sketch Creation and Updating (Q1)

Both M_{k} and C are incrementally constructed from an edge stream of original graph G, where G's edges can arrive in arbitrary order.

Initialization: Initially M_{k} and C are empty data structures with $n = |C| = 0$, and a small constant number k of minwise hash functions are constructed. Whenever an input edge (i, j) contains an unseen node (w.l.o.g. let it be i with id $i = n + 1$), M_{k} and C are extended by one row and one cell, respectively. Each cell of new row $M_{\mathrm{k}}[i]$ is initialized to i representing i's neighborhood, and $C[i]$ is initialized to one.

Algorithm 1 Update Sketch Matrix

Require: Sketch Matrix M_{k}
Require: Count Vector C
Require: new edge (i,j)
1: **for** $m = 1$ to k **do**
2: **if** $h_m(j) < h_m(M_{\mathrm{k}}[i, m])$ **then**
3: $M_{\mathrm{k}}[i, m] = j$
4: **end if**
5: **if** $h_m(i) < h_m(M_{\mathrm{k}}[j, m])$ **then**
6: $M_{\mathrm{k}}[j, m] = i$
7: **end if**
8: **end for**
9: $C[i] + +$; $C[j] + +$;

Updating: Algorithm 1 takes each input edge (i, j) and updates M_{k} and C accordingly. Lines 2-4 update M_{k}'s row i, adding j to i's minwise sampled neighborhood; lines 5-7 likewise update j with i. G^* can be constructed by simply scanning each row i of M_{k} to gather the list of unique elements present in row i, ignoring self-loop. If an element occurs multiple times in $M_k[i]$, it appears only once in $G^*[i]$ and hence $E^* \subseteq E$. Also, G^* does not contain any edge which is absent in the original graph G (i.e, no false edges).

Discussions:

- The edge update process involves $O(k)$ arithmetic and logical operations for each undirected edge, making the time complexity $O(k)$.
- The overall sketch construction time for e edges of the entire graph in this framework is $O(e * k)$.
- G^* contains the minwise sampled neighborhood of each node, efficiently constructed in one-pass from M_k and by design, occupies lesser storage space than G.
- The bias introduced by using approximate minwise independent *linear* permutations [7, 6] is well amortized by the compute savings and manifold speed-up obtained thereby.
- The above implementation treats all input graphs as undirected. However, directionality can be easily incorporated by removing the processing of the edge (j, i) for the incoming edge (i, j).
- The minwise sampling strategy may produce disconnected components at low k, which can be (partially) ameliorated by using symmetrization and edge augmentations as discussed later.
- In our current work, we did not evaluate edge deletion scenario. However, one intuitive approach of handling the deletion of edge (i, j) could be, to replace any occurrence of j in $M_{\mathrm{k}}[i]$ by i (thereby re-initializing the slot with self-loop) and vice-versa.

4.2 Recovering G_m from M_{k}

When the sampling rate is high ($k < d_{avg}$), G^* will lose a substantial portion of G, potentially losing topological properties. Also, it will make G^* a directed graph, making it unsuitable for some applications which require undirected graph as input. Hence a simple and intuitive $(A + A')$-symmetrization algorithm based on [24] is used to build an undirected G_m (without self-loop) from G^*. The choice of this symmetrization strategy guarantees that there are no false edges (edges that do not belong to G) in G_m, which makes the per node neighborhood of G_m more similar to G. As shown earlier in Figure 1, G_m contains more edges of G than G^* (edges of G_m being undirected as opposed to directed edges of G^*). Hence per-node neighborhood of G_m is topologically more similar to G due to the edge recovery. In general, $E^* \subseteq E_m \subseteq E$, E_m being the set of edges in G_m.

4.3 Generating G_s from G_m

Algorithm 2 Generating G_s from G_m

1: Initialize G_s with G_m.
2: For every node i in G_m, let $extraEdge = C[i] - |G_m[i]|$ be the number of edges that need to be augmented for that node.
3: Compute pairwise Jaccard similarity between i and all other nodes of the graph, except the nodes already present in G_m.
4: Sort these list of nodes in decreasing order of similarity values, pick the top $extraEdge$ and augment them to $G_s[i]$
5: After processing all the nodes of the graph, symmetrize G_s using $(A + A')$-symmetrization.
6: **return** G_s

For very small k ($\ll d_{avg}$ of G), G_m will not be able to retain a lot of edges using just the basic symmetrization algorithm. To improve the neighborhood quality of each node by increasing the edge retention, using M_{k}, C and G_m, we try to regenerate as much of the original graph G as possible by adding *similarity-induced edges* to G_m using Algorithm 2. The regenerated graph G_s is undirected and has $|E_s| \geq |E|$, where the inequality is due to false edges not present in G. Of the variety of local similarity measures our framework can estimate [20], we used Jaccard similarity.

5. THEORETICAL INSIGHTS

5.1 Minwise Graph Sketch's Property (Q2)

Minhash sketching of G can be thought of as a neighborhood (edge) sampling strategy. Here we analyze the properties of the sampled sub-graph, which are used as the building blocks for graph property estimation in next section.

LEMMA 5.1. *For any node i with degree d_i, the probability of losing any edge (i, j) of G in G^* with k hashes is $(1 - \frac{1}{d_i})^k$.*

PROOF. For node i, there are k slots and each slot can be filled up by any of it's d_i neighbors. Thus, each slot can be filled up in d_i ways, resulting in total possibilities to be d_i^k. Each of these possibilities is equally likely due to Proposition 1 and the fact that the k slots are filled using independent permutations. Assuming the node j (of the edge (i,j)) has been sparsified (dropped) in G^*, these k slots can now be filled up in a total of $(d_i-1)^k$ ways. Probability of losing the edge (i,j) in i's minhash neighborhood is $\frac{(d_i-1)^k}{d_i^k}$ i.e, $(1-\frac{1}{d_i})^k$. □

Remark: For fixed k, when $d_i \ll k$ the probability of losing an edge is very low and vice-versa. Assuming the graph contains noisy edges, small neighborhood indicates less noisy edges and hence are not sparsified. On the other hand, hub nodes which connect with most of the other nodes in the network resulting in very high d_i, are sparsified more aggressively, which can potentially improve the community structure, while still preserving the connectivity information.

LEMMA 5.2. *The inclusion probability p_{ij} of any edge (i,j) of G in G_m is*

$$p_{ij} = 1 - [(1 - \frac{1}{d_i}) \times (1 - \frac{1}{d_j})]^k$$

PROOF. From Lemma 5.1, we can write that, the probability of losing the edge (i,j) in i's min-hash neighborhood is $(1-\frac{1}{d_i})^k$. Similarly, probability of losing the edge (i,j) in j's min-hash neighborhood is $(1-\frac{1}{d_j})^k$. Minwise neighborhood sampling and sketch generation of nodes i and j are independent of each other, given both are using the same set of hash functions. Hence, the probability of losing the edge (i,j) from both the neighborhood's of i and j is the product of the individual edge loss probabilities computed above. The proof follows from taking a complement of the above probability. □

COROLLARY 5.3. *The above lemma can be generalized to higher order motifs such as triangles, with inclusion probability being $(p_{ij} * p_{jk} * p_{ki})$, where the triple (i,j,k) is a triangle in G.*

5.2 Sketch Based Original Graph Property Estimation (Q3)

In this section, we describe the properties of the original graph G that can be theoretically inferred from the sketch representation.

5.2.1 Global Edge and Triangle Count estimation

LEMMA 5.4. *From G_m, an unbiased estimator of the total number of edges of G using edges E_m of G_m is*

$$\sum_{\{(i,j):E_m \in G_m\}} \frac{1}{(1-[(1-\frac{1}{d_i}) \times (1-\frac{1}{d_j})]^k)}$$

PROOF. From Lemma-5.2, we have the inclusion probability of any edge (i,j) from G to G_m as p_{ij}. Hence a Horvitz-Thompson unbiased estimator [11] will be :

$$\sum_{\{(i,j):E_m \in G_m\}} \frac{1}{p_{ij}} = \sum_{\{(i,j):E_m \in G_m\}} \frac{1}{(1-[(1-\frac{1}{d_i}) \times (1-\frac{1}{d_j})]^k)} \quad (1)$$

□

COROLLARY 5.5. *Using Lemma 5.4, an unbiased estimator of the total number of triangles of G using edges E_m of G_m is*

$$\sum_{\{(i,j),(j,k),(k,i):E_m \in G_m\}} \frac{1}{p_{ij} * p_{jk} * p_{ki}}$$

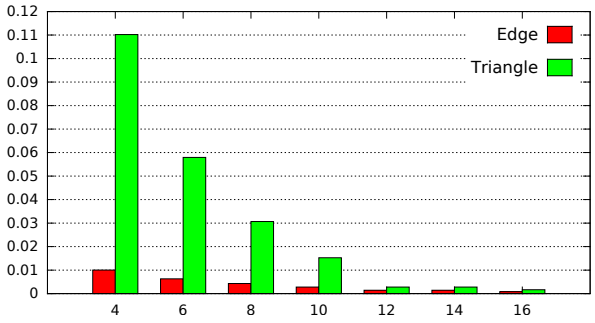


Figure 2: Performance of Horvitz-Thompson Estimators

Model validation: Figure 2 presents the quality of our estimators on *Amazon0302* dataset [18] having *nodes* $= 262,111$, *edges* $= 899,792$ and *triangles* $= 717,719$. We have implemented minwise hashing [7] using MATLAB for model validation and ran the experiments on *Ohio Supercomputing Center*. Relative error w.r.t ground truth has been used to evaluate the quality. We observe that edge estimator performs extremely well even for a very small k (=4). Performance of the triangle estimator significantly improves with increasing k, relative error dropping below 0.06 at $k = 6$. This improvement in estimation quality is in accordance with our theoretical intuition that, as we increase k, more edges and triangles are sampled.

5.2.2 Estimation of Local Neighborhood Properties:

We next focus on finding estimates for the size of the neighborhood intersection of G using the information present in our framework. Note that, neighborhood intersection of two nodes capture the information about the number of shared triangles between them and hence is a very important property to estimate. Let $\hat{s_{ij}}$ be the unbiased estimator of Jaccard similarity between a pair of nodes i and j. Using Proposition 3, we compute it from our framework as:

$$\hat{s_{ij}} = \frac{|\Gamma(i) \cap \Gamma(j)|}{|\Gamma(i) \cup \Gamma(j)|} = \frac{1}{k} \sum_{m=1}^{k} I[h_m(i) == h_m(j)]$$

LEMMA 5.6. *The size of neighborhood intersection $(|\Gamma(i) \cap \Gamma(j)|)$ of any pair of nodes i and j of the original graph G can be estimated as*

$$\widehat{|\Gamma(i) \cap \Gamma(j)|} = \frac{(|\Gamma(i)| + |\Gamma(j)|) * \hat{s_{ij}}}{(1 + \hat{s_{ij}})}$$

PROOF. Using s_{ij}, we compute the neighborhood inter-

section $|\Gamma(i) \cap \Gamma(j)|$ as follows:

$$s_{ij} = \frac{|\Gamma(i) \cap \Gamma(j)|}{|\Gamma(i) \cup \Gamma(j)|}$$

$$\Rightarrow s_{ij} = \frac{|\Gamma(i) \cap \Gamma(j)|}{|\Gamma(i)| + |\Gamma(j)| - |\Gamma(i) \cap \Gamma(j)|}$$

$$\Rightarrow |\Gamma(i) \cap \Gamma(j)| = \frac{(|\Gamma(i)| + |\Gamma(j)|) * s_{ij}}{(1 + s_{ij})} \quad (2)$$

Now, given a MLE $\hat{s_{ij}}$ for s_{ij}, a MLE $|\widehat{\Gamma(i) \cap \Gamma}(j)|$ of $|\Gamma(i) \cap \Gamma(j)|$ will be

$$|\widehat{\Gamma(i) \cap \Gamma}(j)| = \frac{(|\Gamma(i)| + |\Gamma(j)|) * \hat{s_{ij}}}{(1 + \hat{s_{ij}})}$$

. □

COROLLARY 5.7. *Using lemma 5.6 and the definition of local triangle count, an unbiased estimator of the local triangle count of each node i in G_m is*

$$\widehat{T(i)} = \frac{1}{2} \sum_{j \in \Gamma(i)} |\widehat{\Gamma(i) \cap \Gamma}(j)| = \frac{1}{2} \sum_{j \in \Gamma(i)} \frac{(|\Gamma(i)| + |\Gamma(j)|) * \hat{s_{ij}}}{(1 + \hat{s_{ij}})}$$

The Count vector has actual values of $|\Gamma(i)|$ and $|\Gamma(j)|$. $\hat{s_{ij}}$ being a MLE, the estimated intersection $(|\Gamma(i) \cap \Gamma(j)|)$ size is too a MLE due to the invariance property of MLE. Instead of the current single pass strategy, in semi-streaming scenario with two passes and additionally retaining the edge list of G, our estimator $\widehat{T(i)}$ can be used similarly as [4] to count local triangles of the original graph (G). We next focus on the flexibility of our framework to cater to different off-the-shelf graph analytic algorithms.

6. LEVERAGING SKETCH DERIVATIVES (Q4)

We empirically evaluate the quality and effectiveness of our sampling technique in a computational paradigm where, instead of full access to a graph, the only available inputs to each algorithm are the $(n \times k)$ *sketch matrix* M_{k}, *neighbor count vector* C and their derivatives.

6.1 Clustering Coefficient Estimation

The Wedge Sampling technique described by Kolda et al. [16] is modified to directly leverage our in-memory data structures. To compute the probability of the number of wedges centered at a vertex v (i.e,p_v), we use the original degree of the node as reported in C. Wedges (u, v) and (v, w) are generated by looking at the node v's sampled neighborhood in sketch matrix. To determine whether the triplet (u, v, w) forms a triangle, we check the presence of node u in w's unique neighborlist or node w in u's unique neighborlist to declare that the edge (u, w) exists. Also, some node samples resulting in a single node element instead of two node line or three node wedge, are ignored during estimation.

6.2 Local Graph Sparsification

Local Graph Sparsification, as described by Satuluri et al. [25], is modified to be done per node by using our in-memory data structure, G^*. Pairwise Jaccard similarity of two nodes' neighborhoods is directly estimated from M_{k} as discussed previously. While retaining the appropriate number of neighbors for each node using G^* and user specified sparsification exponent, inline symmetrization is done so that the final sparsified output is an undirected graph.

7. EXPERIMENTS

In this section we evaluate, for varying values of k, the run time and memory cost of our minwise neighborhood sketching framework, the amount of graph edges retained by M_{k} and its derivative G^*, G_m, and G_s graphs, and the values of graph measures obtained is compared with the complete, original graphs. Our algorithms can be executed incrementally at any point in time of the graph streaming. However, to consistently measure performance against our baselines, in all experiments we build the sketch matrix against the entire input graph, then run our algorithms to generate results.

7.1 Environment

We implemented Algorithms 1-2 for sketch matrix and graph (re)construction, as well as graph clustering coefficient and sparsification algorithms which use our data structures in C++. For comparison we use the sparsification implementation of Satuluri et al. [25] and the PageRank implementation from Stanford SNAP [19]. For clustering coefficient related experiment, we directly compare the results as reported by Kolda et al. [16]. We used METIS (v 5.0) [14] for clustering and Conductance (ϕ) to evaluate the quality of clusters throughout. Experiments are run on a 3.40GHz Intel(R) Core(TM) i7-2600 machine with 256 KB L1, 1 MB L2, and 8 MB L3 cache and 16 GB memory.

7.2 Datasets

Dataset	Nodes	Edges	$\mathbf{d_{avg}}$
amazon0312	401K	2,350K	11.73
amazon0505	410K	2,439K	11.89
amazon0601	403K	2,443K	12.11
as-skitter	1,696K	11,095K	13.02
cit-Patents	3,775K	16,519K	8.71
roadNet-CA	1,965K	2,767K	2.82
web-BerkStan	685K	6,649K	19.41
web-Google	876K	4,322K	9.87
web-Stanford	282K	1,993K	14.14
wiki-Talk	2,394K	4,660K	3.89
DIP	5K	15K	6.4
Human	12K	63K	10.8
Orkut	3,073K	117,185K	76.2
Twitter	142K	83,271K	1,172.6
Flickr	31K	520K	33.5

Table 2: Dataset properties

Table 2 shows the properties of the wide variety of real-world datasets [18, 16, 25] used in our experiments. We used 300, 15000 and 1500 as number of clusters for Flickr, Orkut and Twitter datasets respectively when using METIS [14].

7.3 Data streaming models

In discussion so far and in Algorithm 1, each input graph is processed as a stream of independent edges, an *edgelist* streaming model. We experiment with this and a more efficient *neighborlist* model as follows:

Edgelist streaming model: Edge (i, j) pairs are processed individually as in Algorithm 1.

Neighborlist streaming model: Neighbor list $(i, \Gamma(i))$ tuples are processed individually where $\Gamma(i)$ are all neighbors of i.

Both models construct the same sketch matrix M_{k} after processing all edges. In the neighborlist model, since complete neighbor information of node i is available at one time,

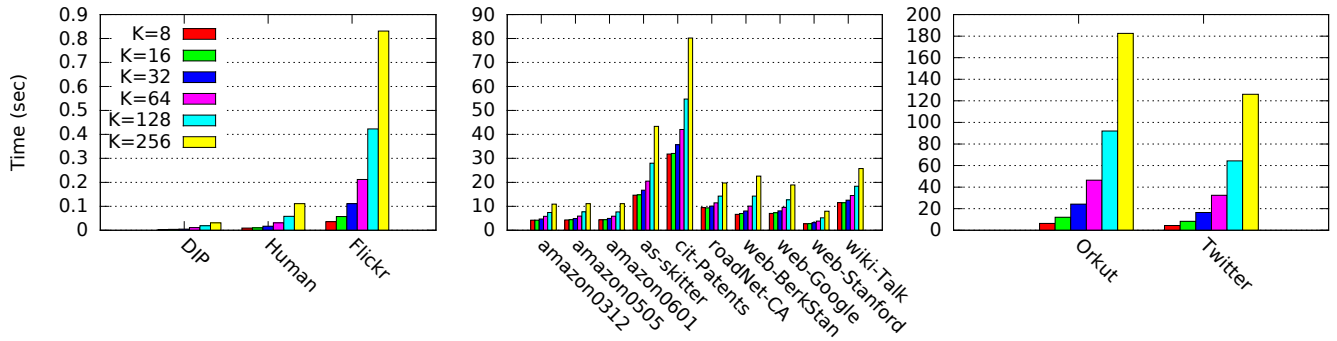


Figure 3: **Sketch matrix construction (Algorithm 1) time varying k.** Datasets grouped by size for scale. At $k = 256$, for cit-Patents (edgelist format) the processing speed is 205,980 edges/sec and for Orkut (neighborlist format) 642,145 edges/sec.

we jointly minwise hash all neighbors $\Gamma(i)$. Compared to the edgelist model, this optimization saves some hash comparisons and accesses (writes to) each M_{k} row only once, resulting in a significant speed-up. Our Clustering Coefficient and PageRank experiments are run using the edgelist streaming model on the SNAP datasets [18]. Edges are treated as undirected, self loops are ignored, and duplicate edges are removed with a bloom filter (false positive probability of 0.01 assuming 100,000,000 distinct elements, which requires 115MB of memory). The cluster quality assessment and sparsification experiments are run using the neighborlist streaming model on the other datasets [25].

7.4 Sketch Construction Costs

In this subsection we evaluate the time and memory cost of building a sketch matrix (M_k) from a source graph (G). We vary k from $8, 16, \ldots, 256$ to show the time and space trade-off of using different numbers of hash functions.

***M_{k}* (sketch) memory cost:** The memory used by the sketch matrix is $(2 * k + n * k + n) * 4$ bytes, where the first term is for hash function parameters, the second term is for M_{k}, and the third term is for C. Since $n \gg k$, we can ignore the negligible space to store $2 * k$ integers when analytically computing space consumption. Thus the amount of memory used by M_{k} is $O(n * k)$, significantly smaller than the $O(e)$ size of G when $d_{\mathrm{avg}} \gg k$.

***M_{k}* (sketch) construction time:** Figure 3 shows sketch construction time for each of our datasets using the sequential Algorithm 1 while varying k. Sketch construction is very fast; even for the largest value of $k = 256$, on cit-Patents (using edgelist format) processing speed is 205,980 edges/sec, and on Orkut (using neighborlist format) 642,145 edges/sec. For context, about 500 million tweets are generated per day on the Twitter social network, for an average of 5800 tweets/sec, well within our processing capacity (using edgelist format), especially considering that only 30% of tweets involve a user interaction edge (retweet or response) [22]. These high and consistent processing speeds validate the suitability of our approach for handling massive real-time data streams.

***G^** memory cost and construction time:** G^*, being a per-node unique element's list, by construction has a smaller memory footprint than G or M_{k}, and for most of our datasets with $d_{avg} < 20$ (all but three), about half of their neighborhood is retained at $k = 32$, allowing a lot of space saving (Figure 4). For Orkut dataset with $k = 256$, the space

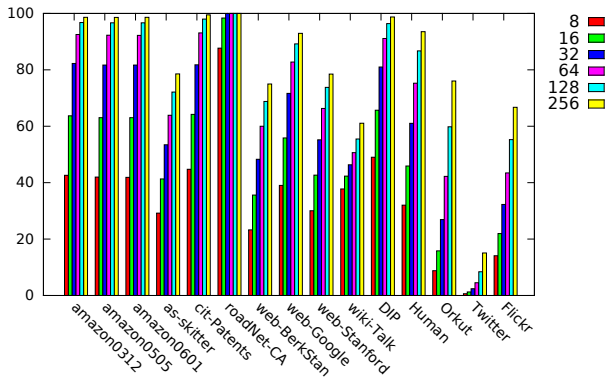


Figure 4: Percent Neighborhood Retention of G^* varying k.

savings using G^* is approximately 24% as compared to full graph representation cost, while retaining about 76.052% of the neighborhood. With $k = 256$, G^* construction time took less than 5 secs for 13 of our 15 datasets, and for our largest dataset Orkut, took 15.5 seconds.

7.5 Application Algorithms on In-core Sketch

7.5.1 Clustering Coefficient

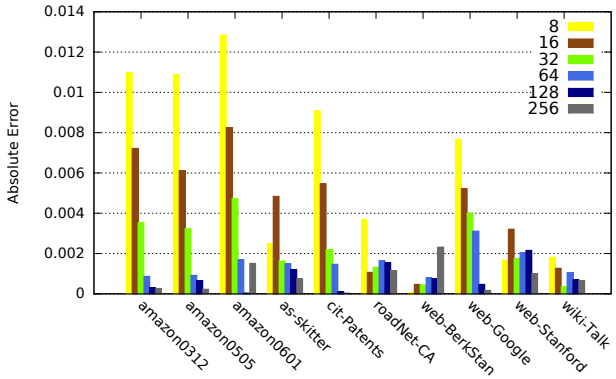


Figure 5: Absolute GCC Error varying k

Figures 5 and 6 show respectively the absolute error for GCC and LCC computed using our hybrid strategy (Section 6.1) while varying k wrt results reported by Kolda et

Dataset	Orig ϕ	LSpar Spar Ratio	LSpar ϕ	LSpar Time (I/O)	LSpar Time (Spars)	LSSpar Spar Ratio	LSSpar ϕ	LSSpar Time (I/O)	LSSpar Time (Spars)
Orkut	0.749	0.424	0.737	14.25	269.463	0.416	0.737	10.49	147.21
Twitter	0.957	0.230	0.945	6.555	162.653	0.089	0.952	1.15	18.51
Flickr	0.782	0.451	0.767	0.060	1.172	0.436	0.770	0.039	0.478

Table 3: **Conductance of sketch based sparsified graph at $k = 256$ wrt original sparsified graph.**

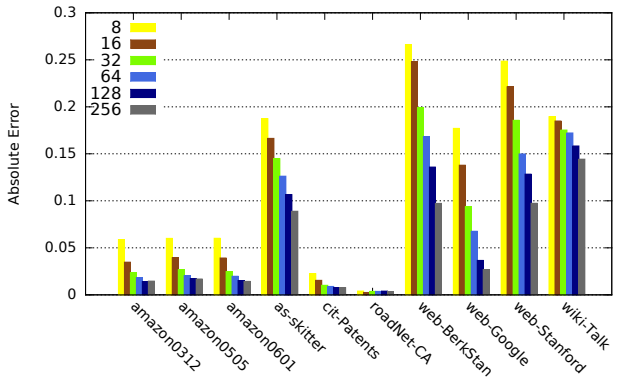


Figure 6: Absolute LCC Error varying k

al. [16]. Since we use randomness for hashing and sampling, experiments have been run for 45 iterations with different seeds, and the results have been averaged across all runs.In general, the quality of the estimation improves (absolute error decreases) as k is increased, since there are more samples per node neighborhood. The absolute error of GCC is small even at low k, as we use the count vector C of the original graph to sample a vertex for GCC computation. However, due to lack of any such guiding global information for selecting a node, the absolute error of LCC is high. Runtime for GCC and LCC estimation is fast in both cases, taking no more than 0.07 seconds on any dataset to find and test 32K sample wedges for triangle closure.

7.5.2 Local Sparsification

Table 3 reports LSpar [25] and LSSpar (Section 6.2) results for three datasets when $k = 256$. LSSpar conductance for Twitter and Flickr falls between ground-truth and LSpar, even though the sparsification ratio is lower. This shows that for a high value of $k = 256$ the minwise sampling step retains most important neighbors of each node, such that our modified sparsification strategy generates a graph which has competitive conductance value, with the number of edges being reduced. The performance of LSSpar is always better than LSpar, as it can directly read the sampled subgraph from the sketch matrix already loaded in memory, as opposed to reading from disk.

Besides the application algorithms which can be run directly on in-memory sketch as discussed above, we next discuss about some of the derivatives that can be generated from our sketch with improved neighborhood quality.

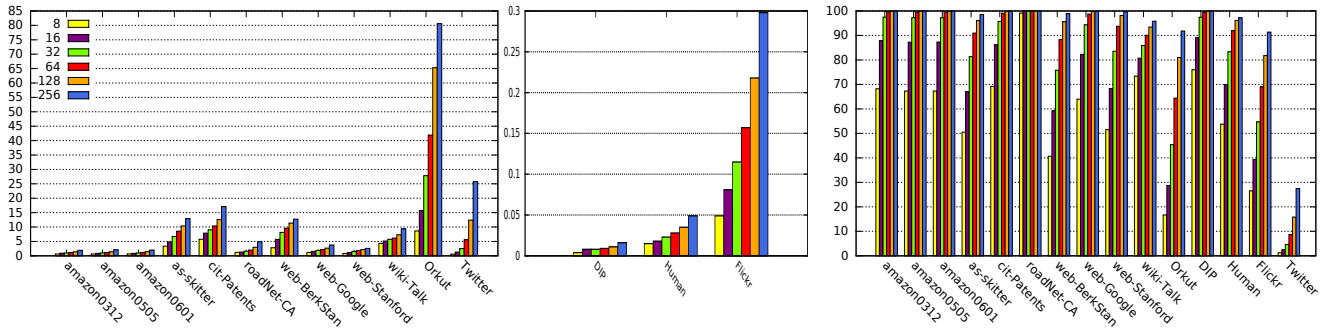
7.6 Recovering G_m from M_k

G^* is a per-node minwise neighborhood sample of the original graph, which can make it a directed subgraph with lesser number of neighbors, when the sampling rate is high ($k < d_{avg}$). Edge augmentation of G^* using $(A + A')$-symmetrization converts it back to an undirected subgraph G_m (without self-loop), with many edges of G recovered per node through the symmetrization process. Figure 7a and 7b show the time taken to symmetrize and construct G_m. Figure 7c shows the amount of the original graph (in terms of per node neighborhood) that can be recovered with such a simple operation. A qualitative assessment of G_m w.r.t clustering quality (conductance) and node importance (PageRank) preservation is presented too.

Performance: Generally, for small value of k (like 8, 16), a simple symmetrization results in recovery of close to 20% of extra neighbors for most of the datasets with very little performance overhead (less than 2.040 secs for $k = 8$ and 3.142 secs for $k = 16$ on average). For a small value of $k = 8$, symmetrization is able to regenerate more than 50% of the graph for eleven out of fifteen datasets whose $d_{avg} \leq 16$. The time taken for these recovery is below 1 sec for six datasets. For Twitter with $d_{avg} = 1172.6$ and $k = 256$, about 27.445% of the whole graph can be recovered from sketch which initially sampled only 15.082% of the graph, resulting in a neighborhood recovery of about 12% w.r.t the original graph, which is high considering the average degree of this dataset.

Conductance: After recovering G_m, we use METIS (v 5.0) [14] for clustering and then compute conductance. Figure 8 indicates that as k increases from 8, in general, G_m's conductance decrease indicating improving clustering quality. The difference of G_m's conductance (ϕ_{G_m}) from G's conductance (ϕ_{GT}) is very small for varying values of k on all the 3 data sets. For Flickr with $d_{avg} = 33.5$, ϕ_{G_m} is lower than ϕ_{GT} for all $k \geq 32$. This means the clustering property of the data set is approximately retained and can potentially improve (indicated by decreasing ϕ values for some k) as we perform minwise sampling of per node neighborhood and symmetrize to construct G_m from original graph. For Twitter, since G_m contains only 27.445% of the full graph, ϕ_{G_m} still remains higher than ϕ_{GT}. But the user gets a minwise sampled subgraph very small in size compared to original graph, with competitive clustering property, as indicated by ϕ_{G_m} value ($\phi_{G_m 256} = 0.964$ while $\phi_{GT} = 0.957$). All recovered graphs can be directly reused by any other program needing an undirected graph (with no self-loops) as input. Next, we focus on evaluating a very important graph property i.e, PageRank on G_m.

PageRank: PageRank has been computed using SNAP toolkit [19] on G_m for varying k and on the corresponding undirected full graphs [18]. The ranked order of vertices in G_m is compared with the ranked order of those vertices in the corresponding graphs using NDCG score. Figure 9 shows that NDCG score for all data set is consistently very high even for $k = 8$. Note that, at such a small value of k, for all the web graphs majority of the nodes have some



(a) Time taken (Secs) (b) Time taken (Secs) (c) Percentage of the graph regenerated

Figure 7: **Generation of G_m from G^* using $(A + A')$-symmetrization**. For a small value of $k = 8$, symmetrization is able to regenerate more than 50% of the graph for eleven out of fifteen datasets whose $d_{avg} \leq 16$.

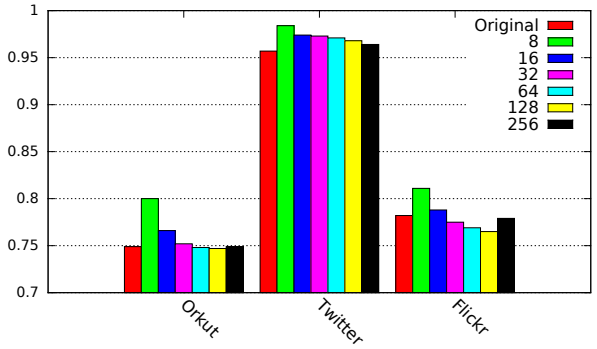


Figure 8: Conductance (ϕ) of G_m for 3 data sets.

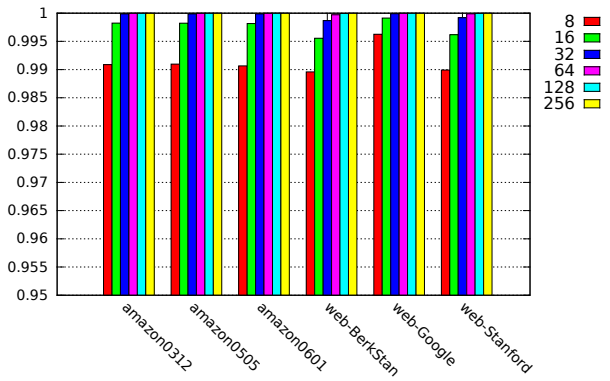


Figure 9: NDCG score for PageRank on G_m wrt G.

subset of edges sparsified since $k < d_{avg}$. In general, as k increases, the per node neighborhood of G_m resembles more closely to that of G due to higher edge retention, resulting in improving NDCG score. This indicates that our sampling strategy followed by subsequent symmetrization technique preserves the *relative importance of a node* (which is a function of the degree distribution) in a graph with acceptable accuracy. Overall, even at a high sampling rate, the quality of preservation of topological properties of G_m is impressive, bolstering the success of minwise sampling and consequent symmetrization based edge recovery technique.

7.7 Generating G_s from G_m

As discussed in Section 4.3 under very aggressive sampling rate ($k \ll d_{avg}$), it is beneficial to recover substantial amount of G while retaining much of the original graph measures in the recovered graph. Hence G_m is augmented with Jaccard similarity induced edges to give it approximately similar degree distribution per node as G using Algorithm 2. This Jaccard similarity induced edge-augmentation uses the theory of Strong Triadic Closure proposed by Granovetter [10] for social networks, in which triangle-dense subgraphs with a lot of common neighbors are prevalent.

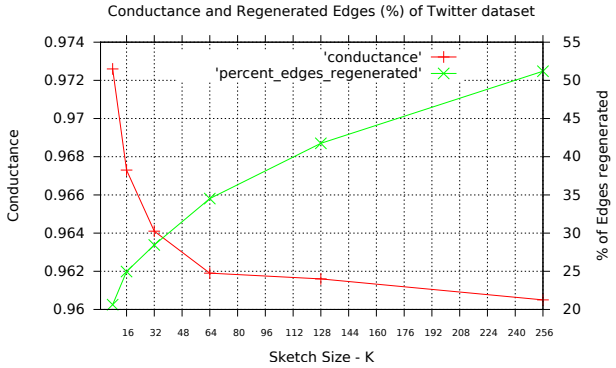


Figure 10: Properties of G_s for Twitter dataset.

Twitter Dataset: Since Twitter has $d_{avg} = 1172.6$, the problem of recovering a substantial portion of the original graph is very relevant and challenging specially for $k \ll d_{avg}$. Figure 10 shows for varying values of k, the percentage of original graph edges our similarity based augmentation algorithm could recover and the corresponding quality w.r.t clustering property (using conductance) of G_s. As k increases, the percentage of true edges regenerated increases while conductance decreases. Interestingly, for $k = 256$, G^* has 15.082% and G_m contains 27.445% of G. However, our similarity based augmentation algorithm is able to regenerate about 51.195% of G, which is a substantial improvement over both G^* and G_m, given that $d_{avg} \simeq 4.5{*}k$. Across all k, $\phi_{G_s} < \phi_{G_m}$, showing similarity based graph augmentation results in improved clustering (lower conductance) of G_s. However, this marginal decrease in conductance value is due

to formation of more triangle-dense subgraphs in G_s (some of which are absent in G), as Jaccard similarity based edge augmentation has a bias towards forming more triangles.

As $|E_s| > |E|$, initially at low k, the number of false edges is very high, but the number decreases gradually with increasing k. This can be attributed to 1) more edges of G being retained by G_m, resulting in lesser number of edges to be augmented per node and 2) Jaccard similarity between pairs of nodes improving and predicting better similarity-induced edges. Specifically for Twitter with $k = 256$ and $d_{avg} = 1172.6$, we are able to recover over 50% of G spending much less of full graph storage cost, to maintain G^* and C.

8. CONCLUSION

In this paper, to answer the four questions pertinent to streaming graphs, we propose a framework that efficiently and incrementally builds minwise neighborhood sampled subgraph of a large graph with streamed edges, within user-configurable fixed memory limit, and able to handle near real time edge streaming rate. Simple symmetrization and similarity-based techniques can recover a significant portion of the original undirected graph. We provide theoretical analysis of our proposed sampling strategy and empirically validate the efficacy and efficiency of our framework to estimate important graph properties.

Future Work: When the edge retention rate is very low (due to small k), using this sketch we can generate a set of bootstrapped variants (G_s) of the original graph (G) similar to Kleiner et al. [15], and estimate the properties of G from this set of regenerated graphs. Our compact sketch matrix is amenable to embarrassingly parallel, lock-less concurrent implementation and can be realized in multi-core CPU, GPU and MIC architectures for further scalability.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback and Ohio Supercomputing Center (Project PAS0166) for access to MATLAB and resources. This work is supported in part by NSF grants DMS-1418265, IIS-1550302 and IIS-1629548.

10. REFERENCES

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *DDC'01*.

[2] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella. Graph sample and hold: A framework for big-graph analytics. In *KDD '14*.

[3] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: Sparsification, spanners, and subgraphs. PODS '12.

[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. KDD '08.

[5] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. STOC '15.

[6] T. Bohman, C. Cooper, and A. Frieze. Min-wise independent linear permutations. *Electronic Journal of Combinatorics*, 7:R26, 2000.

[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *JCSS*, 60:327–336, 1998.

[8] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. VLDB '05.

[9] A. C. Gilbert and K. Levchenko. Compressing network graphs. In *LinkKDD workshop at KDD'04*.

[10] M. S. Granovetter. The strength of weak ties. *American journal of sociology*, pages 1360–1380, 1973.

[11] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260):663–685, 1952.

[12] P. Indyk. A small approximately min-wise independent family of hash functions. SODA '99.

[13] D. R. Karger. Random sampling in cut, flow, and network design problems. In *STOC'94*.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC'98*.

[15] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan. The Big Data Bootstrap. In *ICML'12*.

[16] T. G. Kolda, A. Pinar, and C. Seshadhri. Triadic measures on graphs: The power of wedge sampling. In *SDM'13*.

[17] J. Leskovec and C. Faloutsos. Sampling from large graphs. KDD '06.

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

[19] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. http://snap.stanford.edu/snap, June 2014.

[20] L. Lü and T. Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, 2011.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. TR 1999-66, Stanford InfoLab, 1999.

[22] Replies and retweets on twitter. http://sysomos.com/insidetwitter/engagement/.

[23] Y. Ruan, D. Fuhry, and S. Parthasarathy. Efficient community detection in large networks using content and links. WWW '13.

[24] V. Satuluri and S. Parthasarathy. Symmetrizations for clustering directed graphs. EDBT'11.

[25] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. SIGMOD '11.

[26] T. Suel and J. Yuan. Compressing the graph structure of the web. DCC '01.

[27] N. Tang, Q. Chen, and P. Mitra. Graph stream summarization: From big bang to big crunch. SIGMOD '16.

[28] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. KDD '09.

[29] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):409–10, 1998.

[30] P. Zhao, C. C. Aggarwal, and M. Wang. gsketch: On query estimation in graph streams. *VLDB'11*.