Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics

Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger,
Andrey Ivanov, Yishai Oltchik, Torsten Hoefler
Department of Computer Science; ETH Zurich

ABSTRACT

We propose Slim Graph: the first programming model and framework for practical lossy graph compression that facilitates high-performance approximate graph processing, storage, and analytics. Slim Graph enables the developer to express numerous compression schemes using small and programmable compression kernels that can access and modify local parts of input graphs. Such kernels are executed in parallel by the underlying engine, isolating developers from complexities of parallel programming. Our kernels implement novel graph compression schemes that preserve numerous graph properties, for example connected components, minimum spanning trees, or graph spectra. Finally, Slim Graph uses statistical divergences and other metrics to analyze the accuracy of lossy graph compression. We illustrate both theoretically and empirically that Slim Graph accelerates numerous graph algorithms, reduces storage used by graph datasets, and ensures high accuracy of results. Slim Graph may become the common ground for developing, executing, and analyzing emerging lossy graph compression schemes.

ACM Reference Format:
Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger, Andrey Ivanov, Yishai Oltchik, Torsten Hoefler. 2019. Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3295500.3356182

Slim Graph website:
https://spcl.inf.ethz.ch/Research/Parallel_Programming/SlimGraph

1 INTRODUCTION

Large graphs are a basis of many problems in machine learning, medicine, social network analysis, computational sciences, and others [15, 25, 106]. The growing graph sizes, reaching one trillion edges in 2015 (the Facebook social graph [48]) and 12 trillion edges in 2018 (the Sogou webgraph [101]), require unprecedented amounts of compute power, storage, and energy. For example, running PageRank on the Sogou webgraph using 38,656 compute nodes (10,050,560 cores) on the Sunway TaihuLight supercomputer [71] (nearly the full scale of TaihuLight) takes 8 minutes [101]. The sizes of such datasets will continue to grow; Sogou Corp. expects a ≈60 trillion edge graph dataset with whole-web crawling. *Lowering the size of such graphs is increasingly important for academia and industry*: It would offer speedups by reducing the number of expensive I/O operations, the amount of data communicated over the network [19, 21, 29] and by storing a larger fraction of data in caches.

There exist many *lossless* schemes for graph compression, including WebGraph [33], k^2-trees [37], and others [24]. They provide various degrees of storage reductions. Unfortunately, the majority of these schemes incur *expensive decompression* in performance-critical kernels and *high preprocessing costs* that throttle performance [33, 37]. Moreover, there also exist *succinct* graph representations that *approach the associated graph storage lower bounds* [68, 113, 124, 157]. However, they are mostly theoretical structures with large hidden constants. In addition, as shown recently, the associated storage reductions are not large, at most 20–35%, because *today's graph codes already come close to theoretical storage lower bounds* [31].

In this work, we argue that the next step towards *significantly* higher performance and storage reductions in graph analytics can be enabled by *lossy graph compression* and the resulting *approximate graph processing*. As the size of graph datasets grows larger, a question arises: **Does one need to store and process the exact input graph datasets to ensure precise outcomes of important graph algorithms?** We show that, as with the JPEG compression (see Figure 1), one *may not always* need the full precision while processing graphs.

Our analogy between compressing graphs and bitmaps brings more questions. First, **what is the criterion (or criteria?) of the accuracy of lossy graph compression?** It is no longer a simple visual similarity as with bitmaps. Next, **what is the actual method of lossy compression** that combines large *storage reductions*, high *accuracy*, and *speedups* in graph algorithms running over compressed datasets? Finally, **how to easily implement compression schemes?** To answer these questions, we develop ***Slim Graph: the first programming model and framework for lossy graph compression***.

The first core idea and element of Slim Graph is a programming model that enables straightforward development of different compression schemes for graphs. Here, a developer constructs a simple program called a *compression kernel*. A compression kernel is similar to a vertex program in systems

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11. . . $15.00
https://doi.org/10.1145/3295500.3356182

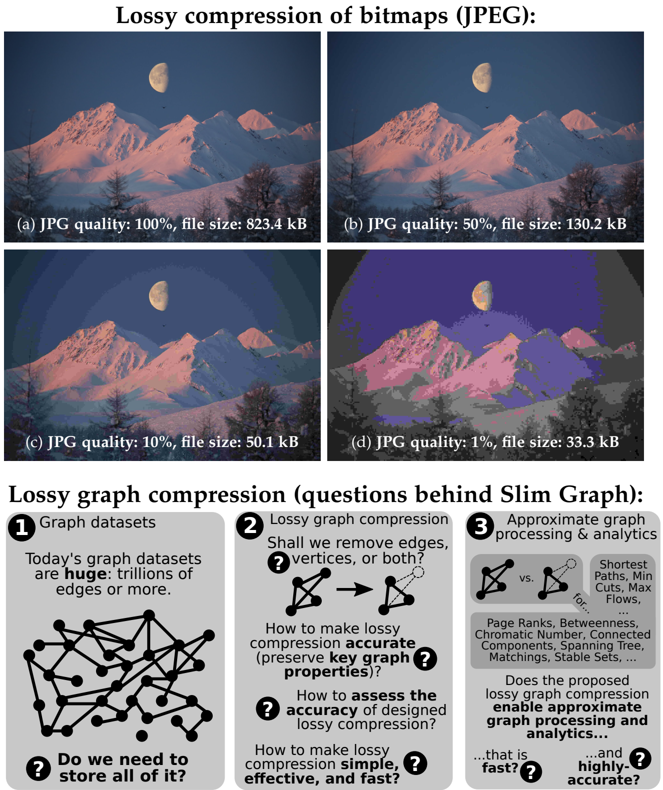

Lossy compression of bitmaps (JPEG):
(a) JPG quality: 100%, file size: 823.4 kB
(b) JPG quality: 50%, file size: 130.2 kB
(c) JPG quality: 10%, file size: 50.1 kB
(d) JPG quality: 1%, file size: 33.3 kB
Lossy graph compression (questions behind Slim Graph):
1 Graph datasets
Today's graph datasets are huge: trillions of edges or more.
? Do we need to store all of it?
2 Lossy graph compression
Shall we remove edges, vertices, or both?
How to make lossy compression accurate (preserve key graph properties)?
How to assess the accuracy of designed lossy compression?
How to make lossy compression simple, effective, and fast?
3 Approximate graph processing & analytics
vs.
for...
Shortest Paths, Min Cuts, Max Flows, ...
Page Ranks, Betweenness, Chromatic Number, Connected Components, Spanning Tree, Matchings, Stable Sets, ...
Does the proposed lossy graph compression enable approximate graph processing and analytics...
...that is fast?
...and highly-accurate?

Figure 1: The comparison of different compression levels of the JPG format and the resulting file sizes (the photos illustrate Chersky Mountains in Yakutia (North-East Siberia), in January, with the Moon and an owl caught while flying over taiga forests). **Can one apply a similar approach to storing complex graph structures?**

such as Pregel [107] or Galois [116] in that it enables accessing local graph elements, such as neighbors of a given vertex. However, there are two key differences. First, the scope of a single kernel is more general than a single vertex — it can be an edge, a triangle, or even an arbitrary subgraph. Second, the goal of a compression kernel is to *remove certain elements of a graph*. The exact elements to be removed are determined by the body of a kernel. In this work, we introduce kernels that preserve graph properties as different as Shortest Paths or Coloring Number while removing significant fractions of edges; these kernels constitute *novel graph compression schemes*. We also illustrate kernels that implement **spanners** [120] and **spectral sparsifiers** [148], established structures in graph theory. These are graphs with edges removed in such a way that, respectively, the distances between vertices and the graph spectra are preserved up to certain bounds. Finally, for completeness, we also express and implement a recent variant of **lossy graph summarization** [141]. *Based on the analysis of more than 500 papers on graph compression, we conclude that **Slim Graph enables expressing and implementing all major classes of lossy graph compression***, including sampling, spectral sparsifiers, spanners, graph summarization, and others.

Next, Slim Graph contributes **metrics for assessing the accuracy of lossy graph compression**. For algorithms that assign certain values to each vertex or edge that impose some vertex or edge ordering (e.g., Brandes Algorithm for Betweenness Centrality [36]), we analyze the numbers of vertex or edge pairs that switched their location in the order after applying compression. Moreover, for graph algorithms with output that can be interpreted as a probability distribution (e.g., PageRank [117]), we propose to use **statistical divergences**, a powerful tool used in statistics to assess the similarity and difference of two probability distributions. We analyze a large number of difference divergence measures and we select the *Kullback-Leibler divergence* [92] as the most suitable tool in the context of comparing graph structure.

We conduct a **theoretical analysis**, presenting or deriving *more than 50 bounds* that illustrate how graph properties change under different compression methods. We also evaluate Slim Graph for different algorithms, on both shared-memory high-end servers and distributed supercomputers. Among others, we were able to use Slim Graph to compress Web Data Commons 2012, the largest publicly available graph that we were able to find (with ≈3.5 billion vertices and ≈128 billion edges), reducing its size by 30-70% using distributed compression. *Slim Graph may become a common ground for developing, executing, and analyzing emerging lossy graph compression schemes on shared- and distributed-memory systems.*

2 NOTATION AND BACKGROUND

We first summarize the necessary concepts and notation. Table 1 presents the used abbreviations.

BFS, SSSP	Breadth-First Search, Single Source Shortest Path [51]
MST, PR, CC	Min. Spanning Tree, PageRank [117], Connected Components
BC, TC	Betweenness Centrality [36, 145], Triangle Counting [142]
TR, EO, CT, SG	Triangle Reduction, Edge Once, Count Triangles, Slim Graph
KL, SVD	Kullback-Leibler, Singular Value Decomposition

Table 1: The most important abbreviations used in the paper.

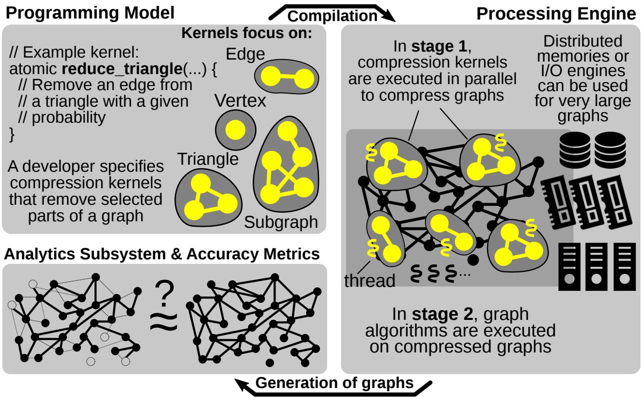
We **model** an undirected graph G as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$, $|E| = m$. N_v and d_v denote the neighbors and the degree of a vertex v, respectively. We also consider weighted and directed graphs and mention this appropriately. The shortest path length between vertices u and v in a graph G is $dist_G(u, v)$. G's maximal degree and diameter are d and D. T is the total number of triangles in a graph.

We list considered well-known **graph problems** in Table 1. Due to space constrains, we describe them in detail in the extended report (see the link on page 1). Importantly, the complexity of algorithms (both sequential and parallel) solving these problems is proportional to m. *Thus, removing graph edges would directly accelerate the considered graph algorithms.*

We also clarify **naming**: we use the term "lossy graph compression" to refer to any scheme that removes some parts of graphs: *sparsification* and *sparsifiers* [16, 148], *sketches* [2], *synopses* [75], *sampling* [79, 99, 160], *spanners* [120], *low-rank approximation* [133, 149], *bounded-error summarization* [115], *lossy compression* [78], and *reduction* [This work].

3 SLIM GRAPH ARCHITECTURE

We now describe the architecture of Slim Graph. An overview is presented in Figure 2. Slim Graph consists of three key elements: (1) a programming model, (2) an execution engine, and (3) an analytics subsystem with accuracy metrics.


Programming Model
Compilation
Processing Engine
Kernels focus on:
Edge
Vertex
Triangle
Subgraph
A developer specifies compression kernels that remove selected parts of a graph
In stage 1, compression kernels are executed in parallel to compress graphs
Distributed memories or I/O engines can be used for very large graphs
thread
Analytics Subsystem & Accuracy Metrics
?
≈
In stage 2, graph algorithms are executed on compressed graphs
Generation of graphs

```
// Example kernel:
atomic reduce_triangle(...) {
  // Remove an edge from
  // a triangle with a given
  // probability
}
```

Figure 2: The overview of the architecture of Slim Graph.

3.1 Part One: Programming Model

The first core part of Slim Graph is a *programming model for graph compression*. The model provides a developer with a set of programmable compression kernels that can be used to *express and implement graph compression schemes*. Intuitively, the developer can program the kernel by providing a small code snippet that uses the information on graph structure (provided by the kernel arguments) to remove certain parts of the graph. These kernels are then executed by an underlying engine, where multiple *instances of kernels* run in parallel.

Thus, a developer has a "local" view of the input graph [151], similar to that of vertex-centric processing frameworks such as Pregel [107] or Galois [116]. Still, Slim Graph enables *several* types of kernels where the "local view of the graph" is (1) a vertex and its neighbors, but it can also be (2) an edge with adjacent vertices, (3) a triangle with neighboring vertices, and (4) a subgraph with a list of pointers to vertices within the subgraph and pointers to neighboring vertices. As we show in detail in § 4, *each type of kernel is associated with a certain class of graph compression algorithms*. For example, a subgraph is used to implement spanners while a triangle is associated with Triangle Reduction, a class proposed in this work. Each of these classes can be used to reduce the graph size while preserving *different* properties; we provide more details in § 6 and § 7. *Slim Graph offers multiple compression schemes because no single compression method can be used to preserve many graph properties deemed important in today's graph computations.*

The developer can indicate whether different parts of a compression kernel will execute *atomically* [135]. The developer can also specify if a given element should be considered for removal *only once* or *more than once* (i.e., by more than one kernel instance). This enables various tradeoffs between performance, scope (i.e., number of removed graph elements), and accuracy of compression. More details are in § 4.

3.2 Part Two: Execution Engine

Second, Slim Graph's processing engine executes compression kernels over input graphs, performing the actual compression. The engine consists of a two-stage pipeline. In stage 1, a graph is compressed with a selected method. In stage 2, a selected graph algorithm is executed on the compressed graph to verify how compression impacts the graph structure. Many considered real-world graphs fit in a memory of a single "fat" server and we use established in-memory techniques and integrate Slim Graph with high-performance shared-memory processing infrastructure, namely GAP Benchmark Suite [13], to deliver fast graph compression routines (we extend GAPBS with new graph algorithms whenever necessary, e.g., to compute matchings, spanning trees, and others). However, if graphs do not fit into the memory of one server, we use a separate pipeline with I/O and distributed-memory tools. Currently, we use a distributed-memory implementation of edge compression kernels, based on MPI Remote Memory Access [20, 23, 57, 73, 134].

Challenges behind designing fast graph processing engines were studied thoroughly in the last decade and summarized in numerous works [11, 22, 26–28, 30, 58, 77, 96, 105, 109, 140, 165, 166]. Thus, in the following, we focus on the novel contributions, which are (1) kernel abstractions for graph compression, (2) novel graph compression methods, (3) novel accuracy metrics, and (4) theoretical and empirical evaluation.

3.3 Part Three: Analytics Subsystem

Slim Graph also provides methods and tools for analyzing the accuracy of graph compression schemes. The proposed metrics can be used to compare the outcomes of graph algorithms that generate a scalar output (e.g., a number of Connected Components), a vector (e.g., Betweenness Centrality), or a probability distribution (e.g., PageRank). The results of this analytics can be used by the Slim Graph user to provide feedback while implementing graph compression routines. We discuss these metrics in detail in § 5.

4 SLIM GRAPH: COMPRESSING GRAPHS

We now show how to **develop lossy graph compression schemes using Slim Graph abstraction of compression kernels**. Table 2 summarizes schemes considered in this work. We (1) describe each scheme and (2) provide the pseudocode of the corresponding kernels and any other required structures. Throughout this section, we use Figure 3 (overview of kernels) and Listing 1 ("C++ style" pseudocode). We present the code of seven key kernels; more examples (a total of 16) can be found in the extended technical report. Finally, we propose **Triangle Reduction, a tunable class of graph compression schemes**, together with corresponding kernels.

4.1 Compression Kernels: Syntax + Semantics

We summarize selected parts of Slim Graph syntax and semantics. To implement a kernel, one first specifies a kernel name and a single kernel argument `x`; `x` can be a vertex, an edge, a triangle, or a subgraph. Within kernel's body, `x` offers properties and methods that enable accessing and modifying local graph structure, e.g., edges adjacent to `x`. Slim Graph also provides a global container object `SG`. `SG` offers various functions and parameters for accessing or modifying *global* graph structure, for example `del(a)` (delete a graph element `a`) or `out_edges(X)` (return all edges with a source vertex in a subgraph induced by elements `X`). `SG` also contains properties of the used compression scheme, for example values of sampling parameters. Finally, Slim Graph syntax includes a

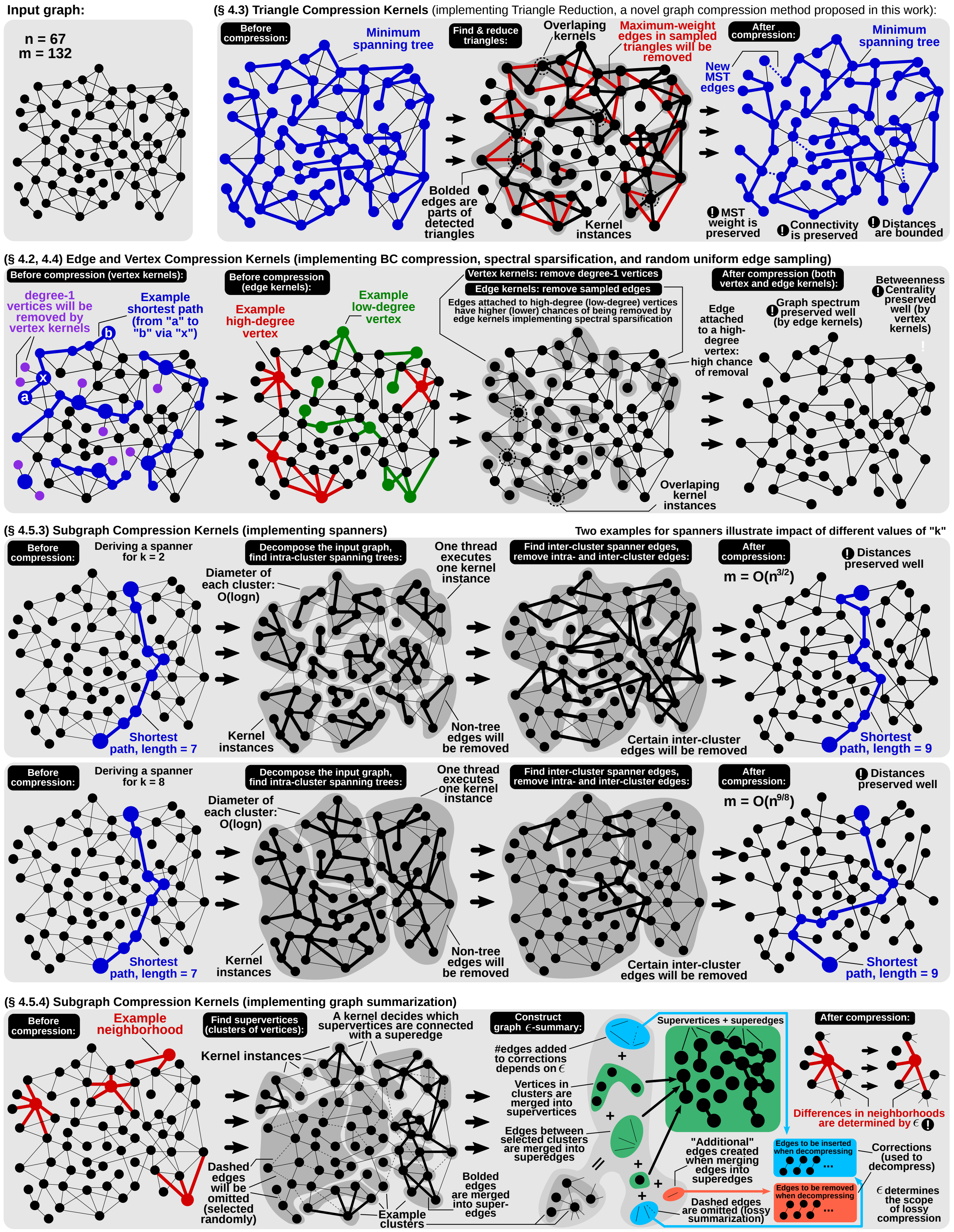

Input graph:
n = 67
m = 132
(§ 4.3) Triangle Compression Kernels (implementing Triangle Reduction, a novel graph compression method proposed in this work):
Before compression:
Minimum spanning tree
Find & reduce triangles:
Overlaping kernels
Maximum-weight edges in sampled triangles will be removed
After compression:
Minimum spanning tree
New MST edges
Bolded edges are parts of detected triangles
Kernel instances
MST weight is preserved
Connectivity is preserved
Distances are bounded
(§ 4.2, 4.4) Edge and Vertex Compression Kernels (implementing BC compression, spectral sparsification, and random uniform edge sampling)
Before compression (vertex kernels):
degree-1 vertices will be removed by vertex kernels
Example shortest path (from "a" to "b" via "x")
b
x
a
Before compression (edge kernels):
Example high-degree vertex
Example low-degree vertex
Vertex kernels: remove degree-1 vertices
Edge kernels: remove sampled edges
Edges attached to high-degree (low-degree) vertices have higher (lower) chances of being removed by edge kernels implementing spectral sparsification
Overlaping kernel instances
Edge attached to a high-degree vertex: high chance of removal
After compression (both vertex and edge kernels):
Graph spectrum preserved well (by edge kernels)
Betweenness Centrality preserved well (by vertex kernels)
(§ 4.5.3) Subgraph Compression Kernels (implementing spanners)
Two examples for spanners illustrate impact of different values of "k"
Before compression:
Deriving a spanner for k = 2
Shortest path, length = 7
Decompose the input graph, find intra-cluster spanning trees:
Diameter of each cluster: O(logn)
Kernel instances
One thread executes one kernel instance
Non-tree edges will be removed
Find inter-cluster spanner edges, remove intra- and inter-cluster edges:
Certain inter-cluster edges will be removed
After compression:
m = O(n^{3/2})
Distances preserved well
Shortest path, length = 9
Before compression:
Deriving a spanner for k = 8
Shortest path, length = 7
Decompose the input graph, find intra-cluster spanning trees:
Diameter of each cluster: O(logn)
Kernel instances
One thread executes one kernel instance
Non-tree edges will be removed
Find inter-cluster spanner edges, remove intra- and inter-cluster edges:
Certain inter-cluster edges will be removed
After compression:
m = O(n^{9/8})
Distances preserved well
Shortest path, length = 9
(§ 4.5.4) Subgraph Compression Kernels (implementing graph summarization)
Before compression:
Example neighborhood
Find supervertices (clusters of vertices):
Kernel instances
A kernel decides which supervertices are connected with a superedge
Dashed edges will be omitted (selected randomly)
Example clusters
Bolded edges are merged into super-edges
Construct graph ϵ-summary:
#edges added to corrections depends on ϵ
Vertices in clusters are merged into supervertices
Edges between selected clusters are merged into superedges
Supervertices + superedges
"Additional" edges created when merging edges into superedges
Dashed edges are omitted (lossy summarization)
Edges to be inserted when decompressing
Edges to be removed when decompressing
Corrections (used to decompress)
ϵ determines the scope of lossy compression
After compression:
Differences in neighborhoods are determined by ϵ

Figure 3: The overview of Slim Graph compression kernels and their impact on various graph properties. Vertex/edge kernels are shown together due to space constraints.

Compression scheme	#remaining edges	Work	W, D†	Storage$^\$$	Preserves best...
		Lossy compression schemes that are a **part of Slim Graph**.			
(§ 4.2.1) Spectral sparsification (“High-conductance” sampling [148])	$\propto \max(\log \frac{3}{p}, \log n)n$	$m + O(1)$	👎, 👍	$O(m+n)$	Graph spectra
(§ 4.2.2) Edge sampling (simple random-uniform sampling)	$(1-p)m$	$m + O(1)$	👎, 👍	$O(m+n)$	Triangle count
(§ 4.3) Triangle reduction (several variants are described in § 4.3)	$m - pT$ (more in § 6)	$O(nd^2)$ or $O(m^{3/2})$	👍, 👍	$O(m+n)$	Several (§ 6)
(§ 4.5.3) Spanners ($O(k)$–spanner [111])	$O(n^{1+1/k} \log k)$	$O(m)$	👍, 👎	$O(m+n)$	Distances
(§ 4.5.4) Lossy summarization with Jaccard similarity (SWeG [141])	$m \pm 2\epsilon m^\ddagger$	$O(mI)^\ddagger$	👎, 👎*	$O(m+n)$	Count of common neighbors
		Past schemes for lossy graph compression (some might be integrated with Slim Graph in future versions):			
(§ 4.6) Lossy summarization with the MDL principle (ApxMdl [115])	$\epsilon m^\ddagger$	$O(C^2 \log n + nm_S)^\ddagger$	👎, 👍	$O(m+n)$	Unknown
(§ 4.6) Lossy linearization [108]	$2kn^*$	$O(mdIT)^*$	👎, 👎	$O(m+n)$	Unknown
(§ 4.6) Low-rank approximation (clustered SVD [133, 149])	—	$O(n_c^3)^\ddagger$	👍, 👍	$O(n_c^2)$	[High error rates]
(§ 4.6) Cut sparsification (Benczúr–Karger [16])	$O(n \log n\, \epsilon^2)$	$O(m \log^3 n + m \log n/\epsilon^2)^\ddagger$	👍, 👎	$O(n+m)$	Cut sizes

Table 2: (§ 4) **Considered lossy compression schemes.** †**W,D** indicate support for weighted or directed graphs, respectively. Symbols used in Slim Graph schemes (p, k) are explained in corresponding sections. $^\$$Storage needed to conduct compression. In the **SWeG lossy summarization** [141], ϵ controls the approximation ratio while I is the number of iterations (originally set to 80 [141]). *SWeG covers undirected graphs but uses a compression metric for directed graphs. In **ApxMdl** [115], ϵ controls the approximation ratio, $C \in O(m)$ is the number of “corrections”, $m_S \in O(m)$ is the number of “corrected” edges. In **lossy linearization** [108], $k \in O(n)$ is a user parameter, I is the number of iterations of a “re-allocation process” (details in Section V.C.3 in the original work [108]), while T is a number of iterations for the overall algorithm convergence. In **clustered SVD approximation** [133, 149], $n_c \leq n$ is the number of vertices in the largest cluster in low-rank approximation. In **cut sparsifiers** [16], ϵ controls the approximation ratio of the cuts.

```
 1 /********** Single-edge compression kernels (§ 4.2) ****************/
 2 spectral_sparsify(E e) { //More details in § 4.2.1
 3   double Y = SG.connectivity_spectral_parameter();
 4   double edge_stays = min(1.0, Y / min(e.u.deg, e.v.deg));
 5   if(edge_stays < SG.rand(0,1)) atomic SG.del(e);
 6   else e.weight = 1/edge_stays;
 7 }
 8 random_uniform(E e) { //More details in § 4.2.2
 9   double edge_stays = SG.p;
10   if(edge_stays < SG.rand(0,1)) atomic SG.del(e);
11 }
12 /************* Triangle compression kernels (§ 4.3) ****************/
13 p-1-reduction(vector<E> triangle) {
14   double tr_stays = SG.p;
15   if(tr_stays < SG.rand(0,1))
16     atomic SG.del(rand(triangle)); }
17 p-1-reduction-EO(vector<E> triangle) {
18   double tr_stays = SG.p;
19   if(tr_stays < SG.rand(0,1)) {
20     E e = rand(triangle);
21     atomic {if(!e.considered) SG.del(e);
22             else e.considered = true; } } }
23 /*********** Single-vertex compression kernel (§ 4.4) ***************/
24 low_degree(V v) {
25   if(v.deg==0 or v.deg==1) atomic SG.del(v); }
26 /************* Subgraph compression kernels (§ 4.5) ****************/
27 derive_spanner(vector<V> subgraph) { //Details in § 4.5.3
28   //Replace "subgraph" with a spanning tree
29   subgraph = derive_spanning_tree(subgraph);
30   //Leave only one edge going to any other subgraph.
31   vector<set<V>> subgraphs(SG.sgr_cnt);
32   foreach(E e: SG.out_edges(subgraph)) {
33     if(!subgraphs[e.v.elem_ID].empty()) atomic del(e);
34 } }
35 derive_summary(vector<V> cluster) { //Details in § 4.5.4
36   //Create a supervertex "sv" out of a current cluster:
37   V sv = SG.min_id(cluster);
38   SG.summary.insert(sv); //Insert sv into a summary graph
39   //Select edges (to preserve) within a current cluster:
40   vector<E> intra = SG.summary_select(cluster, SG.ϵ);
41   SG.corrections_plus.append(intra);
42   //Iterate over all clusters connected to "cluster":
43   foreach(vector<V> cl: SG.out_clusters(out_edges(cluster))) {
44     [E, vector<E>] (se, inter) = SG.superedge(cluster,cl,SG.ϵ);
45     SG.summary.insert(se);
46     SG.corrections_minus.append(inter);
47   }
48   SG.update_convergence();
49 }
```

Listing 1: Implementing lossy graph compression schemes with Slim Graph.

keyword atomic (it indicates atomic execution) and opaque reference types for vertices and edges (V and E, respectively). Example V fields are deg (degree) and parent_ID (ID of the containing graph element, e.g., a subgraph). Example E fields are u (source vertex), v (destination vertex), and weight.

4.2 Single-Edge Kernels

We start from a simple kernel where the Slim Graph programming model provides the developer with access to each edge together with the adjacent vertices and their properties, such as degrees. In Slim Graph, we use this kernel to express two important classes of compression schemes: spectral sparsification and random uniform sampling.

4.2.1 ***Spectral Sparsification with Slim Graph.*** In spectral sparsifiction, one removes edges while preserving (with high accuracy) *graph spectrum* (i.e., the eigenvalues of graph Laplacian). *The graph spectrum determines various properties, for example* ***bipartiteness*** *or* ***spectral clustering coefficient***, which may be important for Slim Graph users. All formal definitions are in the extended report. Now, there exist many works on spectral sparsifiers [6, 12, 39, 50, 69, 83, 89, 91, 95, 97, 146–148, 161, 167]. We *exhaustively analyzed these works*[1] and we identified a method that needs only $O(m+n)$ storage and $O(m)$ time (others require $\Omega(n^2)$ storage or have large hidden constants). Here, edges are sampled according to probabilities different for each edge. These probabilities are selected in such a way that *every vertex in the compressed graph has edges attached to it w.h.p.*. The fraction Y of remaining edges adjacent to each vertex can be proportional to $\log(n)$ [148] ($\mathrm{Y} = p\log(n)$) or to the average vertex degree [82] ($\mathrm{Y} = pm/n$); p is a user parameter. Then, each edge (u,v) stays in the compressed graph with probability $p_{u,v} = \min(1, \mathrm{Y}/\min(d_u, d_v))$. If the output graph must be weighted, then we set $W(u,v) = 1/p_{u,v}$. Now, one can prove that a graph compressed according to the presented scheme *preserves spectrum well* [148].

Slim Graph Implementation In the corresponding kernel spectral_sparsify (Lines 2–6), each edge e (provided as the kernel argument) is processed concurrently. edge_stays (the probability $p_{i,j}$ of sampling e) is derived based on Y (a parameter maintained in SG and pre-initialied by the user) and degrees of vertices u and v attached to e. Then, e is either atomically deleted or appropriately re-weighted.

4.2.2 ***Uniform Sampling with Slim Graph.*** We also express and implement random uniform sampling in Slim Graph.

[1] Our exhaustive review on lossy graph compression is in an accompanying survey that will be released upon publication of this work.

Here, each edge remains in the graph with a probability p. This simple scheme can be used to rapidly compress a graph while preserving accurately the ***number of triangles*** [156].

Slim Graph Implementation The kernel for this scheme is shown in Lines 8–10. Its structure is analogous to `spectral_sparsify`. The main difference is that the sampling probability `edge_stays` (p) is identical for each edge.

4.3 Triangle Kernels for Triangle Reduction

The next class of compression kernels uses triangles (3-cycles) as the "smallest unit of graph compression". Triangle kernels implement *Triangle Reduction* (TR): a class of compression schemes that generalizes past work [87]. In TR, a graph is compressed by removing certain parts of a selected *fraction of triangles*, sampled u.a.r. (uniformly at random). Specific triangle parts to be removed are specified by the developer. Thus, we "reduce" triangles in a specified way.

We focus on triangles because – as we also show later (§ 6, § 7) – TR is *versatile*: *removing certain parts of triangles does not significantly impact a* ***surprisingly large*** *number of graph properties*. For example, removing an edge from each triangle does not increase the ***number of connected components***, while removing the maximum-weight edge from each triangle does not change the ***weight of the minimum spanning tree***. Second, the relatively low computational complexity of mining all the triangles ($O(m^{3/2})$ or $O(nd^2)$), combined with the existing bulk of work on fast triangle listing [54, 74, 80, 123, 142, 155, 159, 162, 164], enables lossy compression of even the largest graphs available today. Further, numerous approximate schemes find *fractions of all triangles in a graph much faster than $O(m^{3/2})$ or $O(nd^2)$* [17, 38, 52, 81, 86, 110, 119] [63, 70, 84, 85, 136–138], further reducing the cost of lossy compression based on TR.

In the basic TR variant, we select pT triangles from a graph u.a.r., $p \in (0;1)$. In each selected triangle, we remove x edges ($x \in \{1,2\}$), chosen u.a.r.. We call this scheme **Triangle** p-x-**Reduction**, where p and x are input parameters.

We advocate the versatility, extensibility, and flexibility of TR by discussing variants of the basic TR scheme that enable tradeoffs between compression performance, accuracy in preserving graph properties, and storage reductions. One variant is ***Edge-Once*** **Triangle** p-x-**Reduction** (EO p-x-TR). Here, we consider each edge *only once* for removal. When a triangle is selected for reduction for the first time (by some kernel instance), if a random edge is not removed, it will *not* be considered for removal in another kernel instance. This protects edges that are a part of *many* triangles (that would otherwise be considered for deletion more often) and thus *may be more important, e.g., they may be a part of multiple* ***shortest paths***. Another example is EO p-1-Triangle Reduction with a modification in which we remove an edge with the highest weight. *This preserves the* ***exact weight of the minimum spanning tree***.

Certain Slim Graph users may be willing to sacrifice more accuracy in exchange for further storage reductions. In such cases, we offer **Triangle** p**-2-Reduction**. Finally, we propose the **Triangle** p**-Reduction by Collapse** scheme in which triangles *are collapsed to single vertices*, each with a probability p. This scheme changes the vertex set in addition to the edge set, offering even more storage reduction.

Slim Graph Implementation The kernel for the basic TR scheme (for $x = 1$) is in Lines 13–16; the EO variant is presented in Lines 17–22. In both cases, the kernel argument `triangle` is implemented as a vector of edges. `SG.`p is a probability of sampling a triangle. We select an edge to be removed with `rand` (an overloaded method that returns – in this case – a random element of a container provided as the argument). Here, by selecting an edge for removal in a different way, one could straightforwardly implement other TR variants. For example, selecting an edge with a maximum weight (instead of using `rand(triangle)`) would preserve the MST weight. The deletion is performed with the overloaded `SG.del` method.

4.4 Single-Vertex Kernels

We enable the user to modify a single vertex. Our example kernel (Lines 24-25) removes all vertices with degree zero and one. The code is intuitive and similar to above-discussed edge kernels. This enables compressing a graph while preserving the exact values of ***betweenness centrality***, because degree-1 vertices do not contribute any values to shortest paths between vertices with degrees higher than one [132].

4.5 Subgraph Kernels

Slim Graph allows for **executing a kernel on an arbitrary *subgraph***. *This enables expressing and implementing different sophisticated compression schemes*, such as spanners (graphs that preserve **pairwise distances**) and lossy graph summarization (graphs that preserve **neighborhoods**).

4.5.1 ***Overview of Slim Graph Runtime.*** To clarify subgraph kernels, we first summarize the general Slim Graph runtime execution, see Listing 2. After initializing `SG`, assuming subgraph kernels are used, Slim Graph constructs `SG.mapping`, a structure that maps each vertex to its subgraph. Mappings are discussed in § 4.5.2; they enable *versatility and flexibility* in implementing lossy compression schemes in Slim Graph. Next, a function `run_kernels` executes each kernel concurrently. These two steps are repeated until a convergence condition is achieved. The convergence condition (and thus executing all kernels more than once) is only necessary for graph summarization. All other lossy compression schemes expressed in Slim Graph require only a single execution of `run_kernels`.

```
SG.init(G); //Init the SG object using the input graph G.
/* In addition, here the user can initialize various parameters
related to the selected lossy compression, etc. */
while(!SG.converged) { //"converged" is updated in "run_kernels"
  if(SG.kernel == SUBGRAPH) SG.construct_mapping();
  SG.run_kernels(); } //Execute all kernels concurrently
SG.free(); //Perform any necessary cleanup.
```

Listing 2: Overview of Slim Graph runtime execution for subgraph kernels.

4.5.2 ***Mappings.*** While analyzing lossy graph compression, we discovered that many representative spanner and graph summarization schemes first decompose a graph into disjoint subgraphs. Next, these schemes use the obtained intra- and inter-subgraph edges to achieve higher compression ratios or to ensure that the compression preserves some graph properties (e.g., diameter). Details of such graph decompositions are algorithm-specific, but they can *all* be defined by a mapping that assigns every vertex to its subgraph. Thus, to express any such compression algorithm in Slim Graph, we enable constructing arbitrary mappings.

Example Mappings Two important mappings used in Slim Graph are based on low-diameter decomposition [111] (takes $O(n+m)$ work) and clustering based on Jaccard similarity [125] (takes $O(mN)$ work; N is #clusters). In the former (used for spanners), resulting subgraphs have (provably) low diameters. In the latter (used for graph summarization), resulting subgraphs consist of vertices that are similar to one another with respect to the Jaccard measure. Both schemes are extensively researched and we omit detailed specifications.

Implementing Mappings To develop mappings, a user can use either the established vertex-centric abstraction or simply access the input graph (maintained as adjacency arrays) through the `SG` container. Implementation details are straightforward; they directly follow algorithmic specifications of low-diameter decompositions [111] or clustering [141]. From populated mappings, the Slim Graph runtime derives subgraphs that are processed by kernel instances.

*4.5.3 **Spanners with Slim Graph**.* An (α, β)**-spanner** [121] is a subgraph $H = (V, E')$ of $G = (V, E)$ such that $E' \subset E$ and

$$dist_G(u,v) \leq dist_H(u,v) \leq \alpha \cdot dist_G(u,v) + \beta, \quad \forall u, v \in V.$$

We exhaustively analyzed works on spanners [3–5, 9, 10, 41, 42, 59, 64, 98, 111, 118, 120, 122] and we select a state-of-the-art scheme by Miller et al. [111] that provides best known work-depth bounds. It first decomposes a graph into low-diameter subgraphs. An input parameter $k \geq 1$ controls how large these subgraphs are. Then, it derives a spanning tree of each subgraph; these trees have low diameters ($k \log(n)$ w.h.p.). After that, for each subgraph C and each vertex v belonging to C, if v is connected to any other subgraph with edges $e_1, ..., e_l$, only one of these edges is added to the resulting $O(k)$-spanner that has $O(n^{1+1/k})$ edges.

Slim Graph Implementation The corresponding kernel is in Lines 27–33 (Listing 1). First, one derives a spanning tree of `subgraph` that is the argument of the compression kernel `derive_spanner`. Then, by iterating over edges outgoing from `subgraph`, the implementation leaves only one edge between any two subgraphs (here, we use `sgr_cnt`, a field of `SG` that maintains the number of subgraphs.

*4.5.4 **Lossy Summaries with Slim Graph**.* We enable Slim Graph to support **lossy ϵ-summarization (ϵ-summaries)**. The general idea behind these schemes is to *summarize* a graph by merging specified subsets of vertices into *supervertices*, and merge parallel edges between supervertices into *superedges*. A parameter ϵ bounds the error (details are algorithm-specific). We exhaustively analyzed existing schemes [14, 40, 47, 61, 67, 93, 103, 104, 115, 126, 130, 141, 152–154, 168]. We focus on SWeG, a recent scheme [141] that constructs supervertices with a generalized Jaccard similarity.

Slim Graph Implementation The corresponding kernel is in Lines 35–48 (Listing 1). It first creates a supervertex `sv` out of a processed cluster; `sv` is added to the summary graph. Next, an algorithm-specific `summary_select` method returns edges selected from `cluster`; ϵ determines the scope of lossy compression (i.e., how many intra-cluster edges are irreversibly dropped). The returned edges are kept in a data structure `corrections_plus` (they are used to better preserve neighborhoods). Finally, one iterates over neighboring clusters (using simple predefined methods that appropriately aggregate edges). For each neighboring cluster, a superedge may be created inside method `SG.superedge`. This method (1) drops certain sampled inter-cluster edges (for lossy compression), (2) returns a newly-created superedge `se` (or a null object, if no superedge was created), and (3) a vector `inter` with edges that *do not belong to the created superedge* (assuming `se` is created) and thus *must be removed whenever one accesses edges that form superedge* `se`. Thus, edges in `inter` are added to `corrections_minus`, a data structure with corrections.

4.6 Slim Graph vs Other Schemes

Other forms of lossy graph compression could be used in future Slim Graph versions as new compression kernels. First, **cut sparsifiers** [16] only target the problem of graph cuts and they are a *specific case of spectral sparsification*: a good spectral sparsifier is also a good cut sparsifier. Second, other schemes target specifically *dynamic and weighted graphs* [78, 102]. Third, **low-rank approximation** [133] of clustered Singular Value Decomposition (SVD) was shown to yield very high error rates [133, 149]; we confirm this (§ 7). Moreover, it has a prohibitive time and space complexity of $O\left(n_c^3\right)$ and $O(n_c^2)$ where n_c is the size of the largest cluster $n_c \in O(n)$. Finally, **lossy summarization** based on the **Minimum Description Length principle** [115] and **Lossy Linearization** [108] have high time complexities of $O(m^2 \log n)$ and $O(mdIT)$, respectively, making them infeasible for today's graphs.

4.7 Kernel Strengths: Takeaways

Compression kernels are *simple*: the "local" (e.g., vertex-centric) view of the graph simplifies designing compression algorithms. Slim Graph implementations of compression schemes based on vertex, edge, or triangle kernels use 3–10× fewer lines of code than the corresponding standard baselines. Subgraph kernels use up to 5× fewer code lines (smaller gains are due to the fact that compression schemes that must be expressed with subgraph kernels are inherently complex and some part of this complexity must also be implemented within Slim Graph mappings). Second, kernels are *flexible*: one easily extends a kernel to cover a different graph property (e.g., preserving the exact MST weight with TR only needs removing an edge with the highest weight). Third, different kernels offer a *tradeoff in compression speed, simplicity, and flexibility*. Vertex kernels have limited expressiveness (as is vertex-centric graph processing [131, 163]), but they are simple to use and reason about, and running all vertex kernels takes $\Omega(n)$ work. Edge kernels are less limited but they take $\Omega(m)$ work. Triangle kernels are even more expressive but take $O(m^{3/2})$ work. Finally, subgraph kernels are the most expressive but also complex to use. We recommend using them if global knowledge of the graph structure is needed. Currently, we use them with spanners and summarization.

5 SLIM GRAPH: ACCURACY METRICS

We now establish metrics for assessing the impact of graph compression on algorithm outcomes. We present the most interesting metrics and omit simple tools such as relative scalar changes (full description is in the report). *Our metrics are generic and can be used with any compression methods.*

Counts of Reordered Pairs For algorithms that output a vector of n values associated with vertices (e.g., in PageRank), we count the number of vertex pairs that are reordered with respect to the considered score such as rank. This count equals $|P_{RE}/\binom{n}{2}|$ where P_{RE} is the number of vertex pairs that are reordered after applying compression; we divide it by the maximum possible number of reordered pairs $\binom{n}{2}$. We also count reordered *neighboring* vertices: it is less accurate but easier to compute ($O(m)$ instead of $O\left(n^2\right)$).

Statistical Divergences Some graph properties and results of algorithms can be modeled with certain *probability distributions*. For example, in PageRank, one assigns each vertex (that models a web page) the probability (rank) of a random surfer landing on that page. In such cases, *we observe that one can use the concept of a* ***divergence****: a statistical tool that measures the distance between probability distributions*. Divergence generalizes the notion of “distance”: it does not need not be symmetric and need not satisfy the triangle inequality. There are dozens of divergences [8, 43]; many belong to two groups: so called f-divergences and Bregman divergences [8].

In order to develop Slim Graph, we analyzed various divergences to understand which one is best suited for Slim Graph. We select the *Kullback-Leibler (KL) divergence* [92], which originated in the field of information theory. The reasons are as follows. First, the Kullback-Leibler divergence is generic and applicable to many problems as it is *the only Bregman divergence which is also an f-divergence* [92]. Moreover, it has been used to measure the information loss while approximating probability distributions [53, 92]. Finally, it has recently been used to find differences between brain networks by analyzing distributions of the corresponding graph spectra [150]. *Thus, Kullback-Leibler divergence can be used to analyze the information loss in graphs compressed by Slim Graph when considering graph properties such as PageRank distributions.*

Formally, Kullback-Leibler divergence measures the deviation of one probability distribution from another one. The deviation of distribution Q from P is defined as $\sum_i P(i) \log_2 \frac{P(i)}{Q(i)}$. The Kullback-Leibler divergence is a non-negative number, equal to zero if and only if P and Q are identical. The lower Kullback-Leibler divergence between probability distributions is, the closer a compressed graph is to the original one, regarding the considered probability distribution.

Algorithm-Specific Measures: BFS BFS is of particular importance in the HPC community as it is commonly used to test the performance of high-performance systems for irregular workloads, for example in the Graph500 benchmark [112]. BFS is also a special case for Slim Graph metrics. Its outcome that is important for Graph500 is a vector of *predecessors* of every vertex in the BFS traversal tree. Thus, we cannot use simple metrics for vector output as they are suitable for centrality-related graph problems where a swapped pair of vertices indicates that a given compression scheme impacts vertex ordering; no such meaning exists in the context of vertex predecessors. Moreover, we cannot use divergences because a vector of predecessors does not form a distribution.

To understand how a given compression scheme impacts the BFS outcome, we first identify various types of edges used in BFS. The core idea is to identify how many *critical* edges that may constitute the BFS tree are preserved after sparsification. For a given BFS traversal, the set of critical edges E_{cr} contains the edges from the actual output BFS traversal tree (*tree edges*) *and* the edges that could potentially be included in the tree by replacing any of the tree edges (*potential edges*). We illustrate an example in Figure 4. $\widetilde{E}_{cr}$ are critical edges in the compressed graph, for a traversal starting from the same root. Now, the fraction $|\widetilde{E}_{cr}|/|E_{cr}|$ indicates the change in the number of critical edges.

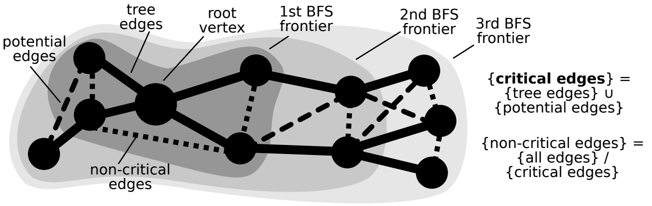

tree edges
root vertex
1st BFS frontier
2nd BFS frontier
3rd BFS frontier
potential edges
non-critical edges
{**critical edges**} = {tree edges} ∪ {potential edges}
{non-critical edges} = {all edges} / {critical edges}

Figure 4: Edge types considered in Slim Graph when analyzing the outcome of BFS.

6 THEORETICAL ANALYSIS

We analyze theoretically how Slim Graph impacts graph properties. Our main result are *novel bounds* (more than 20 non-trivial ones) for *each combination of 12 graph properties and 7 compression schemes*. We show selected results (Table 3); our report details all bounds omitted from the following text.

6.1 Triangle Kernels: Edge-Once p-1-TR

Edge Count We expect to sample pT triangles. Each triangle shares an edge with at most $3d$ other triangles. Thus, an edge is deleted from at least $pT/3d$ triangles (in expectation).

Shortest Path Length At most one edge is deleted from every triangle. Thus, the length of the shortest s-t path does not increase by more than 2×, as we can always use the two edges remaining in the triangle. Moreover, we can show that the shortest s-t path (previously of length $\mathcal{P}$) has length at most $\mathcal{P}(1+p/3)$ in expectation. As we consider each triangle for deletion at most once, the probability of deleting an edge along the shortest path is at most $1/3$. Thus, we expect to delete at most $p\mathcal{P}/3$ edges, increasing the length of the shortest path by the same amount. We can obtain high probability concentration bounds by using Chernoff bounds [35], showing that the shortest path has length at most $\mathcal{P}(1+p)$ w.h.p., if $\mathcal{P}$ is larger than a constant times $\log n$. A similar reasoning gives the bounds for **Diameter**.

Vertex Degree A vertex of degree d' is contained in at most $d'/2$ edge-disjoint triangles. Hence, TR decreases its degree by at most $d'/2$. As this bound holds for every vertex, it also holds for the maximum degree and average degree.

	$\|V\|$	$\|E\|$	Shortest s-t path length	Average path length	Diameter	Average degree	Maximum degree	#Triangles	#Connected components	Coloring number	Max. indep. set size	Max. cardinal. matching size
Original graph	n	m	$\mathcal{P}$	$\overline{P}$	D	$\overline{d}$	d	T	$\mathcal{C}$	C_G	$\widehat{I}_S$	$\widehat{M}_C$
Lossy ϵ-summary	n	$m \pm 2\epsilon m$	$1, \ldots, \infty$	$1, \ldots, \infty$	$1, \ldots, \infty$	$\overline{d} \pm \epsilon\overline{d}$	$d \pm \epsilon d$	$T \pm 2\epsilon m$	$\mathcal{C} \pm 2\epsilon m$	$C_G \pm 2\epsilon m$	$\widehat{I}_S \pm 2\epsilon m$	$\widehat{M}_C \pm 2\epsilon m$
Simple p–sampling	n	$(1-p)m$	∞	∞	∞	$(1-p)\overline{d}$	$(1-p)d$	$(1-p^3)T$	$\leq \mathcal{C} + pm$	$\geq \frac{1-p}{2} C_G$	$\leq \widehat{I}_S + pm$	$\geq (1-p)\widehat{M}_C$
Spectral ϵ-sparsifier	n	$\tilde{O}(n/\epsilon^2)$	$\leq n$	$\leq n$	$\leq n$	$\tilde{O}(1/\epsilon^2)$	$\geq d/2(1+\epsilon)$	$\tilde{O}(n^{3/2}/\epsilon^3)$	$\overset{w.h.p.}{=} \mathcal{C}$	≥ 0	$\leq n$	≥ 0
$O(k)$–spanner	n	$O(n^{1+1/k})$	$O(k\mathcal{P})$	$O(k\overline{P})$	$O(kD)$	$O(n^{1/k})$	$\leq d$	$O(n^{1+2/k})$	$\mathcal{C}$	$O(n^{1/k} \log n)$	$\Omega\left(\frac{n^{1-1/k}}{\log n}\right)$	≥ 0
EO p–1–Triangle Red.	n	$\leq m - \frac{pT}{3d}$	$\overset{w.h.p.}{\leq} \mathcal{P} + p\mathcal{P}$	$\leq \overline{P} + \frac{pT}{n(n-1)}$	$\overset{w.h.p.}{\leq} D + pD$	$\leq \overline{d} - \frac{pT}{dn}$	$\geq d/2$	$\leq (1 - \frac{p}{d})T$	$\mathcal{C}$	$\geq \frac{1}{3} C_G$	$\leq \widehat{I}_S + pT$	$\geq \frac{2}{3} \widehat{M}_C$
remove k deg-1 vertices	$n-k$	$m-k$	$\mathcal{P}$	$\geq \overline{P} - \frac{kD}{n}$	$\geq D-2$	$\geq \overline{d} - \frac{k}{n}$	d	T	$\mathcal{C}$	$\geq C_G - 1$	$\geq \widehat{I}_S - k$	$\geq \widehat{M}_C - k$

Table 3: The impact of various compression schemes on the outcome of selected graph algorithms. Bounds that do not include inequalities hold deterministically. If not otherwise stated, the other bounds hold in expectation. Bounds annotated with w.h.p. hold w.h.p. (if the involved quantities are large enough). Note that since the listed compression schemes (except the scheme where we remove the degree 1 vertices and ϵ-summaries) return a subgraph of the original graph, m, C_G, $\overline{d}$, d, T, and $\widehat{M}_C$ never increase. Moreover, $\mathcal{P}$, $\overline{P}$,D, $\mathcal{C}$, and $\widehat{I}_S$ never decrease during compression. ϵ is a parameter that controls how well a spectral sparsifier approximates the original graph spectrum.

Maximum Cardinality Matching[2] In every triangle, a matching [18] of the original graph can contain at most one of its three edges. Since we delete at most one of the three edges in a triangle uniformly at random, the probability that an edge in a particular maximum matching of the original graph is deleted is at most 1/3. Hence, the expected number of edges that is deleted from the maximum matching (originally of size $\widehat{M}_C$) is at most $1/3\widehat{M}_C$.

Coloring Number In a *greedy coloring*, vertices are colored by visiting the vertices in some predetermined ordering. The coloring number [65] gives the smallest number of colors obtained among all such vertex orderings by a greedy coloring. This best ordering is closely related to the densest subgraph, which is characterized by the *arboricity* [114, 169].

Let $m(S)$ be the number of edges in the subgraph of G induced by the vertex set S. The arboricity [114] is given by

$$\alpha = \max_{\emptyset \subset S \subseteq V} \left\lceil \frac{m(S)}{|S|-1} \right\rceil .$$

The arboricity relates to the coloring number C_G by the inequalities $\alpha \leq C_G \leq 2\alpha$ [169].

Now, consider a set S that obtains the maximum value. The expected number of deleted edges from the subgraph induced by S is at most $m(S)/3$. Hence, the expected arboricity (and coloring number) of the compressed graph is at least $\frac{2}{3}\alpha$, which is at least $\frac{1}{3}C_G$.

Others We observe that all connected components and the minimum spanning tree are preserved (assuming that considered triangles are edge-disjoint and (in MST) the removed edge has maximum weight in the triangle).

6.2 Subgraph Kernels: Spanners

#Triangles A spanner consists of clusters that are trees and each vertex has an edge to $O(n^{1/k})$ clusters (in expectation) [111]. As clusters are acyclic, a triangle with vertex v has to contain one or two vertices that are in a different cluster than v. There are $O(n^{2/k})$ possibilities to choose such two vertices (in expectation). Hence, summing over all vertices, there are $O(n^{1+2/k})$ triangles in expectation.

Coloring Number Within each cluster, the edges form a tree. Any greedy coloring that colors each of these trees bottom-up uses at most $O(n^{1/k} \log n)$ colors. We prove this by bounding the number of edges to different clusters.

The probability that a vertex has an edge to more than l clusters is at most $(1 - n^{-1/k})^{l-1}$ [111]. Setting $l = n^{1/k} 2 \log n + 1$ and using $1 - x \leq e^x$, we get for the probability that a fixed vertex has an edge to more than l clusters: $(1 - n^{-1/k})^{l-1} \leq e^{\frac{l-1}{n^{1/k}}} = n^{-2}$. By a union bound over all vertices, the probability that a vertex has edges to more than $l = O(n^{1/k} \log n)$ clusters is at most n^{-1}. Hence, there is a greedy coloring which uses at most $O(n^{1/k} \log n)$ colors.

6.3 Discussion and Takeaways

With **random uniform sampling**, the ***#connected components*** is not necessarily preserved. Thus, the length of a shortest path between any two vertices has unbounded expectation. Yet, it can be shown that if p is large enough, the compressed graph *does* preserve #connected components w.h.p. and the size of a minimum cut also obtains its expected value [88]. All other considered schemes, except graph summarization, preserve the number of connected components, at least w.h.p..

$O(k)$**-Spanners** preserve well ***lengths of shortest paths*** and also the ***diameter***. Spanners compress the ***edge count*** to close to linear in vertex count when a large stretch k is allowed. Yet, for small k (e.g., $k = 2$) the graph can have many edges (up to $\min(m, n^{3/2})$). Spanners also allow for a ***coloring*** with relatively few colors and have a large ***independent set***.

Edge-Once Triangle p**-1-Reduction** gives nontrivial bounds for *all* considered graph properties (except independent sets). Compressed graphs are 2-spanners and, w.h.p., $\alpha = p, \beta = O(\log n)$ spanners. Moreover, the compressed graph approximates the size of the ***largest matching*** up to a factor 2/3 and the ***coloring number*** up to a factor 1/3. If there are many triangles, the scheme can eliminate up to a third of the ***number of edges***. This is significant because k-spanners do not guarantee compression for $k \leq 2$.

Spectral sparsification preserves the value of ***minimum cuts*** and ***maximum flows*** [88, 148]. Moreover, there is a relationship between the maximum degree of a graph and its Laplacian eigenvalues, meaning that the ***maximum degree*** is preserved up to a factor close to 2. Thus, the compressed graph admits a ***coloring*** with $O(d)$ (d is the maximum degree of the original graph). Spectral sparsifiers always return a sparse graph, achieving a ***number of edges*** that is close to linear in the number of vertices.

[2] (11.2019) bound updated

ϵ**-Summary** bounds the size of the ***symmetric difference between neighborhoods*** *in the compressed and original graph. Its bounds are not competitive with others as this scheme can arbitrarily disconnect the graph and insert new edges*, see Table 3.

7 EVALUATION

Lossy graph compression enables tradeoffs in three key aspects of graph processing: *performance*, *storage*, and *accuracy*. We now illustrate several of these tradeoffs. *Our goal is **not** to advocate a single compression scheme, but to (1) confirm pros and cons of different schemes, provided in § 6, and (2) illustrate that **Slim Graph enables analysis of the associated tradeoffs**.*

Algorithms, Schemes, Graphs We consider algorithms and compression schemes from § 2 and Table 2, and all associated parameters. We also consider all large graphs from SNAP [100], KONECT [94], DIMACS [56], Web Data Commons [1], and WebGraph datasets [33]; see Table 4 for details. *This creates a **very large evaluation space** and we only summarize selected findings; full data is in the extended report.*

Evaluation Methodology For algorithmic execution we use the arithmetic mean for data summaries. We treat the first 1% of any performance data as warmup and we exclude it from the results. We gather enough data to compute the mean and 95% non-parametric confidence intervals.

Machines We use CSCS Piz Daint, a Cray with various XC* nodes. Each XC50 compute node contains a 12-core HT-enabled Intel Xeon E5-2690 CPU with 64 GiB RAM. Each XC40 node contains two 18-core HT-enabled Intel Xeons E5-2695 CPUs with 64 GiB RAM. We also use high-end servers, most importantly a system with Intel Xeon Gold 6140 CPU @ 2.30GHz, 768GB DDR4-2666, 18 cores, and 24.75MB L3.

Friendships: Friendster (**s-frs**, 64M, 2.1B), Orkut (**s-ork**, 3.1M, 117M), LiveJournal (**s-ljn**, 5.3M, 49M), Flickr (**s-flc**, 2.3M, 33M), Pokec (**s-pok**, 1.6M, 30M), Libimseti.cz (**s-lib**, 220k, 17M), Catster/Dogster (**s-cds**, 623k, 15M), Youtube (**s-you**, 3.2M, 9.3M), Flixster (**s-flx**, 2.5M, 7.9M),

Hyperlink graphs: Web Data Commons 2012 (**h-wdc**, 3.5B, 128B), EU domains (2015) (**h-deu**, 1.07B, 91.7B), UK domains (2014) (**h-duk**, 787M, 47.6B), ClueWeb12 (**h-clu**, 978M, 42.5B), GSH domains (2015) (**h-dgh**, 988M, 33.8B), SK domains (2005) (**h-dsk**, 50M, 1.94B), IT domains (2004) (**h-dit**, 41M, 1.15B), Arabic domains (2005) (**h-dar**, 22M, 639M), Wikipedia/DBpedia (en) (**h-wdb**, 12M, 378M), Indochina domains (2004) (**h-din**, 7.4M, 194M), Wikipedia (en) (**h-wen**, 18M, 172M), Wikipedia (it) (**h-wit**, 1.8M, 91.5M), Hudong (**h-hud**, 2.4M, 18.8M), Baidu (**h-bai**, 2.1M, 17.7M), DBpedia (**h-dbp**, 3.9M, 13.8M),

Communication: Twitter follows (**m-twt**, 52.5M, 1.96B), Stack Overflow interactions (**m-stk**, 2.6M, 63.4M), Wikipedia talk (en) (**m-wta**, 2.39M, 5M),

Collaborations: Actor collaboration (**l-act**, 2.1M, 228M), DBLP co-authorship (**l-dbl**, 1.82M, 13.8M), Citation network (patents) (**l-cit**, 3.7M, 16.5M), Movie industry graph (**l-acr**, 500k, 1.5M)

Various: UK domains time-aware graph (**v-euk**, 133M, 5.5B), Webbase crawl (**v-wbb**, 118M, 1.01B), Wikipedia evolution (de) (**v-ewk**, 2.1M, 43.2M), USA road network (**v-usa**, 23.9M, 58.3M), Internet topology (Skitter) (**v-skt**, 1.69M, 11M),

Table 4: Considered graphs with $n > 2M$ or $m > 10M$ from established datasets [1, 33, 56, 94, 100]. **Graph are sorted by m in each category.** For each graph, we show its "(**symbol used later**, n, m)".

7.1 Storage and Performance

We start with storage and performance tradeoffs. Figure 5 shows the impact of different compression parameters on m and performance (we use smaller graphs to analyze in detail a large body of parameters). Plotted graphs are selected to cover different edge sparsity and number of triangles per vertex (T/n is 1052 (s-cds), 20 (s-pok), and 80 (v-ewk)). In most cases, *spanners and p-1-TR ensure the largest and smallest **storage reductions***, respectively. This is because subgraphs in spanners become spanning trees while p-1-TR removes only as many edges as the count of triangles. Uniform and spectral sampling offer a middle ground — depending on p, they can offer arbitrarily small or large reductions of m. Moreover, *respective storage reductions entail similar **performance effects** (fewer edges indicates faster algorithmic execution).* Still, there are some effects specific to each scheme. Spanners offer mild performance improvements for small k that increase by a large factor after a certain threshold of k is reached. Other schemes steadily accelerate all algorithms with growing p.

We also test TR on weighted graphs (resulted excluded due to space constraints). For very sparse graphs, such as the US road network, compression ratio and thus speedups (for both MST and SSSP) from TR is very low. MST's performance is in general not influenced much because it depends mostly on n. In other graphs, such as v-ewk, SSSP speedups follow performance patterns for BFS. For some graphs and roots, very high p that significantly enlarges diameter (and iteration count) may cause slowdowns. Changing Δ can help but needs manual tuning. *Lossy compression may also degrade performance if a selected scheme is unsuitable for targeted algorithms.*

We also analyze variants of proposed Slim Graph compression kernels. Figure 6 shows size reductions in graphs compressed with spectral sparsification variants, in which the number of remaining edges is proportional to the average degree or $\log(n)$. We also analyze variants of TR; "CT" is an additional variant of "EO" in which we not only consider an edge for removal *at most once*, but also *we remove edges starting from ones that belong to the fewest triangles*. Spectral variants result in different size reductions, depending on graphs. *Contrarily, the "CT" and "EO" TR variants consistently deliver smaller m than a simple p-1-TR (for a fixed $p = 0.5$).*

7.2 Accuracy

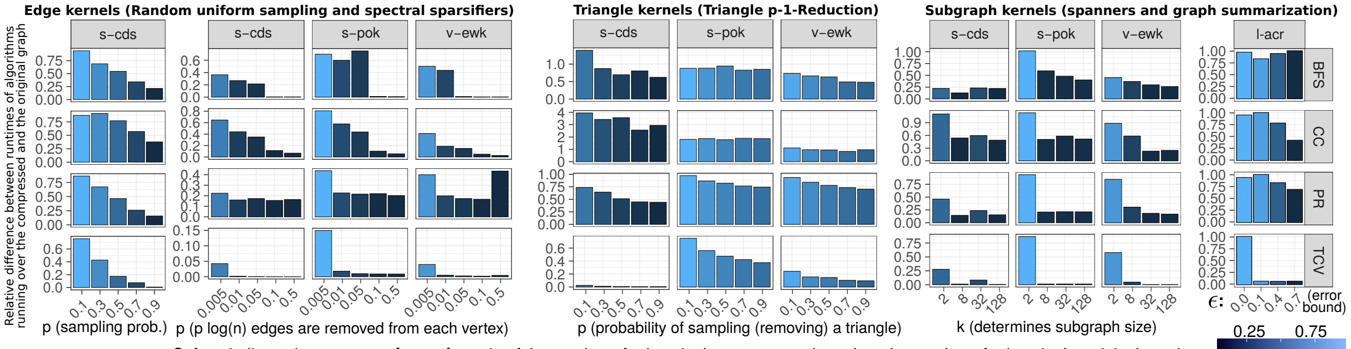
We use Slim Graph metrics to analyze the accuracy of graph algorithms after compressing graphs. First, we show that the Kullback-Leibler divergence can assess information loss due to compression, see Table 5. *In all the cases, the higher compression ratio is (lower m), the higher KL divergence becomes.*

Graph	EO 0.8-1-TR	EO 1.0-1-TR	Uniform ($p = 0.2$)	Uniform ($p = 0.5$)	Spanner ($k = 2$)	Spanner ($k = 16$)	Spanner ($k = 128$)
s-you	0.0121	0.0167	0.1932	0.6019	0.0054	0.2808	0.2993
h-hud	0.0187	0.0271	0.0477	0.1633	0.0340	0.2794	0.3247
il-dbl	0.0459	0.0674	0.0749	0.2929	0.0080	0.1980	0.2005
v-skt	0.0410	0.0643	0.0674	0.2695	0.0311	0.1101	0.2950
v-usa	0.0089	0.0100	0.1392	0.5945	0.0000	0.0074	0.0181

Table 5: Kullback-Leibler divergences between PageRank probability distributions on the original and compressed graphs, respectively.

Another proposed metric is the number of pairs of neighboring vertices that swapped their order (with respect to a certain property) after compression. We test this metric for BC and TC per vertex. Note that this metric should be used *when the compared schemes remove the same number of edges (possibly in expectation)*. Otherwise, numbers of reordered vertices may differ simply because one of compared graphs has fewer vertices left. With this metric, we discover that *spectral sparsification preserves TC per vertex better than other methods*.

We also discover that used $O(k)$-spanners preserve the accuracy of the BFS traversal trees surprisingly well. For example, for the s-pok graph, respectively, removing 21%



Colors indicate the **compression ratio**: ratio of the number of edges in the compressed graph to the number of edges in the original graph

Figure 5: Analysis of **storage and performance** tradeoffs of various lossy compression schemes implemented in Slim Graph (**when varying compression parameters**).

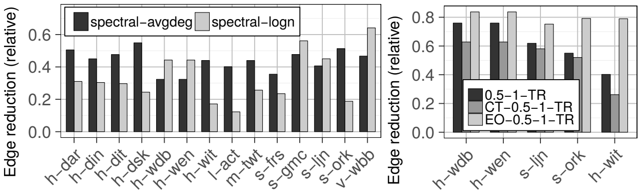


Figure 6: Compression ratio analysis: different variants of spectral sparsification (left) and triangle reduction (right), for a fixed $p = 0.5$. Extending results from Figure 5 (panels "spectral sparsification" and "TR", argument $p = 0.5$) to (1) graphs of different sizes, sparsities, classes, degree distributions, and (2) multiple compression variants.

($k = 2$), 73% ($k = 8$), 89% ($k = 32$), and 95% ($k = 128$) of edges preserves 96%, 75%, 57%, and 27% of the critical edges that constitute the BFS tree. *The accuracy is maintained when different root vertices are picked and different graphs are selected*.

We also investigate how triangle count (T) is reduced with lossy compression. Intuitively, TR should significantly impact T. While this is true, we also illustrate that *almost all schemes, especially spanners, eliminate a large fraction of triangles*, see Table 6. This is because spanners, especially for large k, remove most of cycles while turning subgraphs into spanning trees.

Graph	Original	0.2-1-TR	0.9-1-TR	Uniform ($p = 0.8$)	Uniform ($p = 0.5$)	Uniform ($p = 0.2$)	Spanner ($k = 2$)	Spanner ($k = 16$)	Spanner ($k = 128$)	Spectral ($p = 0.5$)	Spectral ($p = 0.05$)	Spectral ($p = 0.005$)
s-you	11.38	1.544	0.037	0.091	1.416	5.825	7.626	0.071	0.000	0	0.007	0.426
s-flx	9.389	0.645	0.017	0.075	1.173	4.802	6.933	0.000	0.070	0	0.001	0.219
s-flc	1091	6.845	0.164	8.765	136.6	557.9	250.7	1.327	0.001	0	0.016	1.517
s-cds	3157	18.56	0.561	25.24	394.8	1615	844.5	45.392	0.001	0	0.015	4.821
s-lib	938.3	31.51	0.902	7.569	116.9	480.2	82.59	167.0	5.708	0	0.000	0.042
s-pok	59.82	10.25	0.280	0.480	7.494	30.58	41.27	0.362	0.000	0	0.005	1.962
h-dbp	6.299	1.158	0.072	0.051	0.822	3.218	2.295	0.440	0.002	0	0.020	1.981
h-hud	14.71	1.832	0.083	0.117	1.839	7.538	7.373	0.001	0.000	0	0.005	2.495
l-cit	5.973	1.994	0.091	0.048	0.747	3.059	5.128	0.240	0.000	0	0.007	1.931
l-dbl	45.57	6.144	0.257	0.365	5.671	23.33	22.64	0.033	0.004	0	0.066	8.572
v-ewk	235.2	14.13	0.422	1.886	29.33	120.3	110.0	0.034	0.000	0	0.008	2.436
v-skt	50.88	2.642	0.099	0.395	6.455	26.01	22.24	5.777	0.502	0	0.016	2.376

Table 6: (Accuracy) Analysis of the average number of triangles per vertex.

Further tradeoffs between accuracy and size reductions are related to other graph properties. For example, the MM size is least affected by TR. Similarly, the MST is preserved best by TR (assuming a variant that always removes the maximum weight edge in a triangle), followed by spanners. In SSSP, spanners best preserve lengths of shortest paths, followed by TR. Finally, spanners and the "EO" variant of TR maintain the number of CC. Contrarily, random uniform sampling and spectral sparsification disconnect graphs. Graph summarization acts similarly to random uniform sampling (also with respect to other properties), because it can also arbitrarily remove edges. However, for a fixed p, the latter generates significantly fewer (by >10×) components than the former; this is because *used spectral sparsification schemes were designed to minimize graph disconnectedness*.

In Slim Graph, we also analyze the impact of compression kernels on degree distributions. As degree distributions determine many structural and performance properties of a graph, *such analysis is a visual method of assessing the impact of compression on the graph structure*. This method is also *applicable to graphs with different vertex counts*. We illustrate the impact from spanners on three popular graphs often used in graph processing works (Twitter, Friendster, .it domains) in Figure 7. Interestingly, spanners "strengthen the power law": the higher k is, the closer to a straight line the plot is. *One could use such observations to accelerate graph processing frameworks that process compressed graphs, by navigating the design of data distribution schemes, load balancing methods, and others.*

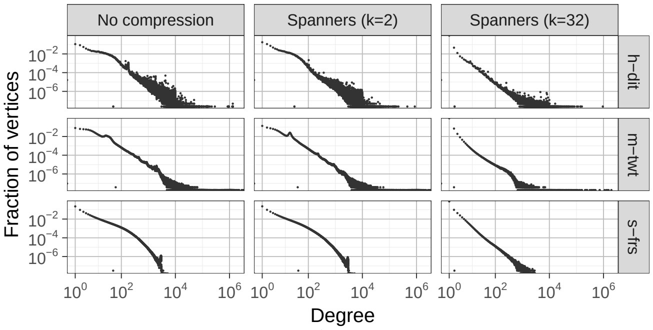


Figure 7: Accuracy analysis (varying k): impact of spanners on the degree distribution of popular graph datasets, Twitter communication (m-twt), Friendster social network (s-frs), and .it domains (h-dit). Extending results from Figure 5 (panel "spanners", arguments $k \in \{2, 32\}$) to degree distribution details.

7.3 Distributed Compression of Large Graphs

To the best of our knowledge, *we present the first results from distributed lossy graph compression*. In a preliminary analysis, we compressed the *five largest publicly available graphs* using edge kernels (random uniform sampling) and we analyze their degree distributions in Figure 8. Random uniform sampling "removes the clutter": scattered points that correspond to specific fractions of vertices with different degrees. *This suggests that random uniform sampling could be used as preprocessing for more efficient investigation into graph power law properties.*

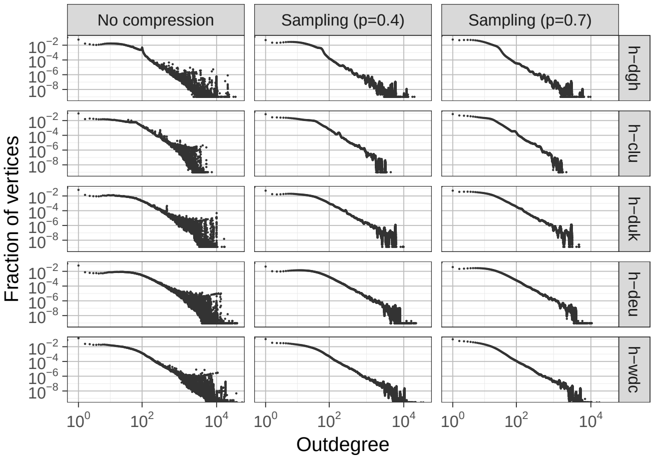


Figure 8: (Accuracy) Impact of random uniform sampling on the degree distribution of large graphs (the largest, h-wdc, has ≈128B edges). #Compute nodes used for compression: 100 (h-wdc), 50 (h-deu), 20 (h-duk), 13 (h-clu), and 10 (h-dgh).

7.4 Other Analyses

We also compared Slim Graph kernels against low-rank approximation (of adjacency or Laplacian graph matrix). It entails significant storage overheads (cf. Table 2) and consistently very high error rates. We also **timed the compression routines**. The compression time is not a bottleneck and it follows asymptotic complexity ($O(m)$ for uniform sampling, spectral sparsification, and spanners, $O(Im)$ for summarization, and $O(m^{3/2})$ for TR). In all cases, sampling is the fastest; spectral sparsification is negligibly slower as each kernel must access degrees of attached vertices. Spanners are >20% slower due to overheads from low-diameter decomposition (larger constant factors in $O(m)$). TR is slower than spanners by >50% ($O(m^{3/2}$ vs. $O(m)$). Summarization is >200% slower than TR due to large constant factors and a complex design.

7.5 How To Select Compression Schemes?

We **summarize our analyses** by providing guidelines on selecting a compression scheme for a specific algorithm. Overall, **empirical analyses follow our theoretical predictions**. Thus, as **the first step**, we recommend to consult Table 3 and select a compression scheme that ensures best accuracy. **Second**, one should verify whether a selected method is feasible, given the input graph size *and* graph type, e.g., whether a scheme supports weighted or directed graphs. Here, we offer Table 2 for overview and Section 7.4 with remarks on empirical performance. **Third**, to select concrete parameter values, one should consult Figure 5, key insights from § 7.1–§ 7.3, and – possibly – the report with more data.

8 RELATED WORK

Lossy graph compression is outlined in § 2, § 4.6, and in Table 2. *We analyze its feasibility for practical usage and we express and implement representative schemes as Slim Graph compression kernels*, covering spanners [120], spectral sparsifiers [148], graph summarization [141], and others [108]. Our TR schemes generalize past work that removes two edges from triangles in weighted graphs to preserve exact shortest paths [87]. *Most of the remaining schemes could be implemented as Slim Graph kernels.* Second, **lossless graph compression** is summarized in a recent survey [24]; it is outside the Slim Graph scope. Third, many **approximation graph algorithms** have been develop to alleviate NP-Completeness and NP-Hardness of graph problems [49, 55, 55, 66, 76, 90, 158]. Contrarily to Slim Graph, *these works are usually hard to use in practice and they do not compress input graphs*. More recently, **approximate graph computations** dedicated to a single algorithm were proposed [7, 32, 34, 44–46, 60, 62, 72, 81, 127–129, 129, 144]. Some works consider general approximate graph processing [82, 139, 143]; *they do not focus on lossy compression and they do not analyze metrics for different algorithm classes.*

9 CONCLUSION

We introduce Slim Graph: the first framework and programming model for lossy graph compression. The core element of this model are *compression kernels*: small code snippets that modify a local part of the graph, for example a single edge or a triangle. Compression kernels can express and implement multiple methods for lossy graph compression, for example spectral sparsifiers and spanners. To ensure that Slim Graph is versatile, we exhaustively analyzed a large body of works in graph compression theory. Users of Slim Graph could further extend it towards novel compression methods.

Slim Graph introduces metrics for assessing the quality of lossy graph compression. Our metrics target different classes of graph properties, e.g., vectors of numbers associated with each vertex, or probability distributions. For the latter, we propose to use statistical divergences, like the Kullback-Leibler divergence, to evaluate information loss caused by compression. Slim Graph could be extended with other metrics.

In theoretical analysis, we show how different compression methods impact different graph properties. We illustrate or derive more than 50 bounds. For example, we constructively show that a graph compressed with Triangle Reduction (TR) has a maximum cardinality matching (MCM) of size at least half of the size of MCM in the uncompressed graph. TR is a novel class of compression methods, introduced in Slim Graph, that generalizes past work and is *flexible*: one can easily tune it to preserve accurately various graph properties.

We use Slim Graph to evaluate different schemes in terms of (1) reductions in graph sizes, (2) performance of algorithms running over compressed graphs, and (3) accuracy in preserving graph properties. We also conduct the first distributed lossy compression of the largest publicly available graphs. We predict that *Slim Graph may become a platform for designing and analyzing today's and future lossy graph compression methods, facilitating approximate graph processing, storage, and analytics.*

ACKNOWLEDGMENTS

We thank Mark Klein, Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank David Schmidig for help with analyzing low-rank approximation, and Timo Schneider for his immense help with computing infrastructure at SPCL. This project has received funding from the European Research Council (ERC) under the European Union's Horizon2020 programme (grant agreement DAPP, No.678880).

REFERENCES

[1] [n.d.]. Hyperlink Graph 2012. http://webdatacommons.org/hyperlinkgraph/2012-08/download.html.

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 459–467.

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. ACM, 5–14.

[4] Stephen Alstrup, Søren Dahlgaard, Arnold Filtser, Morten Stöckel, and Christian Wulff-Nilsen. 2017. Constructing light spanners deterministically in near-linear time. *arXiv preprint arXiv:1709.01960* (2017).

[5] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. 1993. On sparse spanners of weighted graphs. *Discrete & Computational Geometry* 9, 1 (1993), 81–100.

[6] David G Anderson, Ming Gu, and Christopher Melgaard. 2014. An efficient algorithm for unweighted spectral graph sparsification. *arXiv preprint arXiv:1410.4273* (2014).

[7] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 124–137.

[8] Michèle Basseville. 2010. Divergence measures for statistical data processing. (2010).

[9] Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. 2010. Additive spanners and (α, β)-spanners. *ACM Transactions on Algorithms (TALG)* 7, 1 (2010), 5.

[10] Surender Baswana and Sandeep Sen. 2007. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms* 30, 4 (2007), 532–563.

[11] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, et al. 2015. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing* 18, 3 (2015), 1189–1213.

[12] Joshua Batson, Daniel A Spielman, Nikhil Srivastava, and Shang-Hua Teng. 2013. Spectral sparsification of graphs: theory and algorithms. *Commun. ACM* 56, 8 (2013), 87–94.

[13] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[14] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. 2018. Scalable Approximation Algorithm for Graph Summarization. *pacific-asia conference on knowledge discovery and data mining* (2018), 502–514.

[15] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. *arXiv preprint arXiv:1901.10183* (2019).

[16] András A Benczúr and David R Karger. 1996. Approximating st minimum cuts in Õ (n 2) time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 47–55.

[17] Suman K Bera and Amit Chakrabarti. 2017. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[18] Maciej Besta, Marc Fischer, Tal Ben-Nun, Johannes De Fine Licht, and Torsten Hoefler. 2019. Substream-Centric Maximum Matchings on FPGA. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 152–161.

[19] Maciej Besta, Syed Minhaj Hassan, Sudhakar Yalamanchili, Rachata Ausavarungnirun, Onur Mutlu, and Torsten Hoefler. 2018. Slim NoC: A low-diameter on-chip network topology for high energy efficiency and scalability. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 43–55.

[20] Maciej Besta and Torsten Hoefler. 2014. Fault tolerance for remote memory access programming models. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 37–48.

[21] Maciej Besta and Torsten Hoefler. 2014. Slim fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 348–359.

[22] Maciej Besta and Torsten Hoefler. 2015. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 161–172.

[23] Maciej Besta and Torsten Hoefler. 2015. Active access: A mechanism for high-performance distributed data-centric computations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 155–164.

[24] Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *arXiv preprint arXiv:1806.01799* (2018).

[25] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2019. Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons. *arXiv preprint arXiv:1911.04200* (2019).

[26] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 32–41.

[27] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *arXiv preprint arXiv:1910.09017* (2019).

[28] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 93–104.

[29] Maciej Besta, Marcel Schneider, Karolina Cynk, Marek Konieczny, Erik Henriksson, Salvatore Di Girolamo, Ankit Singla, and Torsten Hoefler. 2019. FatPaths: Routing in Supercomputers, Data Centers, and Clouds with Low-Diameter Networks when Shortest Paths Fall Short. *arXiv preprint arXiv:1906.10885* (2019).

[30] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. *arXiv preprint arXiv:1903.06697* (2019).

[31] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. 2018. Log (graph): a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 7.

[32] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. 2011. HyperANF: Approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th international conference on World wide web*. ACM, 625–634.

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

[34] Michele Borassi and Emanuele Natale. 2016. KADABRA is an adaptive algorithm for betweenness via random approximation. *arXiv preprint arXiv:1604.08553* (2016).

[35] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. 2013. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press.

[36] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[37] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k2-Trees for Compact Web Graph Representation.. In *SPIRE*, Vol. 9. Springer, 18–30.

[38] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 253–262.

[39] Daniele Calandriello, Ioannis Koutis, Alessandro Lazaric, and Michal Valko. 2018. Improved large-scale graph learning through ridge spectral sparsification. In *International Conference on Machine Learning*.

[40] Stéphane Campinas, Renaud Delbru, and Giovanni Tummarello. 2013. Efficiency and precision trade-offs in graph summary algorithms. In *Proceedings of the 17th International Database Engineering and Applications Symposium on*. 38–47.

[41] Keren Censor-Hillel and Michal Dory. 2018. Distributed spanner approximation. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 139–148.

[42] Keren Censor-Hillel, Ami Paz, and Noam Ravid. 2018. The Sparsest Additive Spanner via Multiple Weighted BFS Trees. *arXiv preprint arXiv:1811.01997* (2018).

[43] Sung-Hyuk Cha. 2007. Comprehensive survey on distance/similarity measures between probability density functions. *City* 1, 2 (2007), 1.

[44] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. 2005. Approximating the minimum spanning tree weight in sublinear time. *SIAM Journal on computing* 34, 6 (2005), 1370–1379.

[45] Shiri Chechik, Daniel H Larkin, Liam Roditty, Grant Schoenebeck, Robert E Tarjan, and Virginia Vassilevska Williams. 2014. Better approximation algorithms for the graph diameter. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial

and Applied Mathematics, 1041–1052.

[46] Mostafa Haghir Chehreghani, Albert Bifet, and Talel Abdessalem. 2018. Efficient Exact and Approximate Algorithms for Computing Betweenness Centrality in Directed Graphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 752–764.

[47] Chen Chen, Cindy Xide Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. 2009. Mining graph patterns efficiently via randomized summaries. *very large data bases* 2, 1 (2009), 742–753.

[48] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[49] Nicos Christofides. 1976. *Worst-case analysis of a new heuristic for the travelling salesman problem*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

[50] Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. 2018. Graph sparsification, spectral sketches, and faster resistance computation, via short cycle decompositions. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 361–372.

[51] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[52] Graham Cormode and Hossein Jowhari. 2017. A second look at counting triangles in graph streams (corrected). *Theoretical Computer Science* 683 (2017), 22–30.

[53] Thomas M Cover and Joy A Thomas. 2012. *Elements of information theory*. John Wiley & Sons.

[54] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. 2017. Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[55] Etienne de Klerk, Dmitrii V Pasechnik, and Joost P Warners. 2004. On approximate graph colouring and max-k-cut algorithms based on the θ-function. *Journal of Combinatorial Optimization* 8, 3 (2004), 267–294.

[56] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. American Math. Soc.

[57] Salvatore Di Girolamo, Konstantin Taranov, Andreas Kurth, Michael Schaffner, Timo Schneider, Jakub Beránek, Maciej Besta, Luca Benini, Duncan Roweth, and Torsten Hoefler. 2019. Network-accelerated non-contiguous memory transfers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 56.

[58] Niels Doekemeijer and Ana Lucia Varbanescu. 2014. A survey of parallel graph processing frameworks. *Delft University of Technology* (2014), 21.

[59] Michal Dory. 2018. Distributed Approximation of Minimum k-edge-connected Spanning Subgraphs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 149–158.

[60] Stefania Dumbrava, Angela Bonifati, Amaia Nazabal Ruiz Diaz, and Romain Vuillemot. 2018. Approximate Evaluation of Label-Constrained Reachability Queries. *arXiv preprint arXiv:1811.11561* (2018).

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

[62] Ghizlane ECHBARTHI and Hamamache KHEDDOUCI. 2017. Lasas: an aggregated search based graph matching approach. In *The 29th International Conference on Software Engineering and Knowledge Engineering*.

[63] Talya Eden, Amit Levi, Dana Ron, and C Seshadhri. 2017. Approximately counting triangles in sublinear time. *SIAM J. Comput.* 46, 5 (2017), 1603–1646.

[64] Michael Elkin and Ofer Neiman. 2018. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Transactions on Algorithms (TALG)* 15, 1 (2018), 4.

[65] Paul Erdős and András Hajnal. 1966. On chromatic number of graphs and set-systems. *Acta Mathematica Hungarica* 17, 1-2 (1966), 61–99.

[66] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. 1999. Fast approximate graph partitioning algorithms. *SIAM J. Comput.* 28, 6 (1999), 2187–2214.

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

[68] Arash Farzan and J Ian Munro. 2008. Succinct representations of arbitrary graphs. In *European Symposium on Algorithms*. Springer, 393–404.

[69] Zhuo Feng. 2016. Spectral graph sparsification in nearly-linear time leveraging efficient spectral perturbation analysis. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 57.

[70] Jacob Fox, Tim Roughgarden, C Seshadhri, Fan Wei, and Nicole Wein. 2018. Finding cliques in social networks: A new distribution-free model. *arXiv preprint arXiv:1804.07431* (2018).

[71] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (2016), 072001.

[72] Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 90–100.

[73] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2014. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Scientific Programming* 22, 2 (2014), 75–91.

[74] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 1–8.

[75] Sudipto Guha and Andrew McGregor. 2012. Graph synopses, sketches, and streams: A survey. *Proceedings of the VLDB Endowment* 5, 12 (2012), 2030–2031.

[76] Magnús M Halldórsson. 1993. A still better performance guarantee for approximate graph coloring. *Inform. Process. Lett.* 45, 1 (1993), 19–23.

[77] Safiollah Heidari, Yogesh Simmhan, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 60.

[78] Wilko Henecka and Matthew Roughan. 2015. Lossy compression of dynamic, weighted graphs. In *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, 427–434.

[79] Pili Hu and Wing Cheong Lau. 2013. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865* (2013).

[80] Yang Hu, Hang Liu, and H Howie Huang. 2018. High-Performance Triangle Counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–5.

[81] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. {ASAP}: Fast, Approximate Graph Pattern Mining at Scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 745–761.

[82] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. 2018. Bridging the GAP: towards approximate graph analytics. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, 10.

[83] Arun Jambulapati and Aaron Sidford. 2018. Efficient Õ (n/epsilon) Spectral Sketches for the Laplacian and its Pseudoinverse. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2487–2503.

[84] Madhav Jha, Ali Pinar, and C Seshadhri. 2015. Counting triangles in real-world graph streams: Dealing with repeated edges and time windows. In *2015 49th Asilomar Conference on Signals, Systems and Computers*. IEEE, 1507–1514.

[85] Madhav Jha, C Seshadhri, and Ali Pinar. 2015. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 9, 3 (2015), 15.

[86] Hossein Jowhari and Mohammad Ghodsi. 2005. New streaming algorithms for counting triangles in graphs. In *International Computing and Combinatorics Conference*. Springer, 710–716.

[87] Vasiliki Kalavri, Tiago Simas, and Dionysios Logothetis. 2016. The shortest path is not always a straight line: leveraging semi-metricity in graph analysis. *Proceedings of the VLDB Endowment* 9, 9 (2016), 672–683.

[88] David R. Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76. https://doi.org/10.1145/331605.331608

[89] Jonathan A Kelner and Alex Levin. 2013. Spectral sparsification in the semi-streaming setting. *Theory of Computing Systems* 53, 2 (2013), 243–262.

[90] Subhash Khot and Oded Regev. 2008. Vertex cover might be hard to approximate to within 2- ε. *J. Comput. System Sci.* 74, 3 (2008), 335–349.

[91] Ioannis Koutis and Shen Chen Xu. 2016. Simple parallel and distributed algorithms for spectral graph sparsification. *ACM Transactions on Parallel Computing (TOPC)* 3, 2 (2016), 14.

[92] Solomon Kullback. 1997. *Information theory and statistics*. Courier Corporation.

[93] K. Ashwin Kumar and Petros Efstathopoulos. 2018. Utility-driven graph summarization. *very large data bases* 12, 4 (2018), 335–347.

[94] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proc. of Intl. Conf. on World Wide Web (WWW)*. ACM, 1343–1350.

[95] Rasmus Kyng and Zhao Song. 2018. A Matrix Chernoff Bound for Strongly Rayleigh Distributions and Spectral Sparsifiers from a few Random Spanning Trees. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 373–384.

[96] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. *AcM sIGMoD Record* 40, 4 (2012), 11–20.

[97] Yin Tat Lee and He Sun. 2018. Constructing linear-sized spectral sparsification in almost-linear time. *SIAM J. Comput.* 47, 6 (2018), 2315–2336.

[98] Christoph Lenzen and Reut Levi. 2018. A Centralized Local Algorithm for the Sparse Spanning Graph Problem. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

[100] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[101] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. ShenTu: processing multi-trillion edge graphs on millions of cores in seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 56.

[102] Wei Liu, Andrey Kan, Jeffrey Chan, James Bailey, Christopher Leckie, Jian Pei, and Ramamohanarao Kotagiri. 2012. On compressing weighted time-evolving graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2319–2322.

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

[104] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *Comput. Surveys* 51, 3 (2018), 62.

[105] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.

[106] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. *Par. Proc. Let.* 17, 1 (2007), 5–20.

[107] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Manag. of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[108] Hossein Maserrat and Jian Pei. 2012. Community Preserving Lossy Compression of Social Networks. In *2012 IEEE 12th International Conference on Data Mining*. 509–518.

[109] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25.

[110] Andrew McGregor, Sofya Vorotnikova, and Hoa T Vu. 2016. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 401–411.

[111] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 192–201.

[112] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray User's Group (CUG)* (2010).

[113] Moni Naor. 1990. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics* 28, 3 (1990), 303–307.

[114] C. S. J. A Nash-Williams. 1961. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society* 1, 1 (1961), 445–450.

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

[116] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[117] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[118] Merav Parter, Ronitt Rubinfeld, Ali Vakilian, and Anak Yodpinyanee. 2018. Local Computation Algorithms for Spanners. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[119] Aduri Pavan, Srikanta Tirthapura, et al. 2013. Counting and sampling triangles from a graph stream. (2013).

[120] David Peleg and Alejandro A Schäffer. 1989. Graph spanners. *Journal of graph theory* 13, 1 (1989), 99–116.

[121] David Peleg and Jeffrey D Ullman. 1989. An optimal synchronizer for the hypercube. *SIAM Journal on computing* 18, 4 (1989), 740–747.

[122] Seth Pettie. 2010. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing* 22, 3 (2010), 147–166.

[123] Adam Polak. 2016. Counting triangles in large graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 740–746.

[124] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)* 3, 4 (2007), 43.

[125] Raimundo Real and Juan M Vargas. 1996. The probabilistic basis of Jaccard's index of similarity. *Systematic biology* 45, 3 (1996), 380–385.

[126] Matteo Riondato, David García-Soriano, and Francesco Bonchi. 2017. Graph summarization with quality guarantees. *Data Mining and Knowledge Discovery* 31, 2 (2017), 314–349.

[127] Matteo Riondato and Evgenios M Kornaropoulos. 2016. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery* 30, 2 (2016), 438–475.

[128] Matteo Riondato and Eli Upfal. 2018. ABRA: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 5 (2018), 61.

[129] Liam Roditty and Virginia Vassilevska Williams. 2013. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM, 515–524.

[130] Amin Sadri, Flora D. Salim, Yongli Ren, Masoomeh Zameni, Jeffrey Chan, and Timos Sellis. 2017. Shrink: Distance preserving graph compression. *Information Systems* 69 (2017), 180–193.

[131] Semih Salihoglu and Jennifer Widom. 2014. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment* 7, 7 (2014), 577–588.

[132] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 76–85.

[133] Berkant Savas and Inderjit S Dhillon. 2011. Clustered low rank approximation of graphs in information science applications. In *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM, 164–175.

[134] Patrick Schmid, Maciej Besta, and Torsten Hoefler. 2016. High-performance distributed rma locks. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 19–30.

[135] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 445–456.

[136] C Seshadhri. 2015. A simpler sublinear algorithm for approximating the triangle count. *arXiv preprint arXiv:1505.01927* (2015).

[137] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, Vol. 4. 5.

[138] C Seshadhri, Ali Pinar, and Tamara G Kolda. 2014. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7, 4 (2014), 294–307.

[139] Zechao Shang and Jeffrey Xu Yu. 2014. Auto-approximation of graph computing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1833–1844.

[140] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 81.

[141] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. 2019. Sweg: Lossless and lossy summarization of web-scale graphs. In *Proceedings of the 28th International Conference on World Wide Web. ACM*, Vol. 1. 1–2.

[142] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 149–160.

[143] Somesh Singh and Rupesh Nasre. 2018. Scalable and Performant Graph Processing on GPUs Using Approximate Computing. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018), 190–203.

[144] George M Slota and Kamesh Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 405–414.

[145] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 47.

[146] Tasuku Soma and Yuichi Yoshida. 2019. Spectral Sparsification of Hypergraphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2570–2581.

[147] Daniel A Spielman and Nikhil Srivastava. 2011. Graph sparsification by effective resistances. *SIAM J. Comput.* 40, 6 (2011), 1913–1926.

[148] Daniel A Spielman and Shang-Hua Teng. 2011. Spectral sparsification of graphs. *SIAM J. Comput.* 40, 4 (2011), 981–1025.

[149] Xin Sui, Tsung-Hsien Lee, Joyce Jiyoung Whang, Berkant Savas, Saral Jain, Keshav Pingali, and Inderjit Dhillon. 2012. Parallel clustered low-rank approximation of graphs and its application to link prediction. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 76–95.

[150] Daniel Yasumasa Takahashi, Joao Ricardo Sato, Carlos Eduardo Ferreira, and André Fujita. 2012. Discriminating different classes of biological networks by analyzing the graphs spectra distribution. *PLoS One* 7, 12 (2012), e49949.

[151] Adrian Tate, Amir Kamil, Anshu Dubey, Armin Größlinger, Brad Chamberlain, Brice Goglin, Carter Edwards, Chris J Newburn, David Padua, Didem Unat, et al. 2014. Programming abstractions for data locality. PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center.

[152] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. 2011. Compression of weighted graphs. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 965–973.

[153] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. 2012. Network compression by node and edge mergers. *Bisociative Knowledge Discovery* (2012), 199–217.

[154] Ioanna Tsalouchidou, Francesco Bonchi, Gianmarco De Francisci Morales, and Ricardo Baeza-Yates. 2018. Scalable Dynamic Graph Summarization. *IEEE Transactions on Knowledge and Data Engineering* (2018), 1–1.

[155] Charalampos E Tsourakakis. [n.d.]. Fast Counting of Triangles in Large Real Networks: Algorithms and Laws. *cis. temple. edu* ([n. d.]), 608–617.

[156] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 837–846.

[157] György Turán. 1984. On the succinct representation of graphs. *Discrete Applied Mathematics* 8, 3 (1984), 289–294.

[158] Jason TL Wang, Kaizhong Zhang, and Gung-Wei Chirn. 1995. Algorithms for approximate graph matching. *Information Sciences* 82, 1-2 (1995), 45–74.

[159] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. 2016. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 1–8.

[160] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. 2011. Understanding graph sampling algorithms for social network analysis. In *2011 31st international conference on distributed computing systems workshops*. IEEE, 123–128.

[161] Yongyu Wang and Zhuo Feng. 2017. Towards scalable spectral clustering via spectrum-preserving sparsification. *arXiv preprint arXiv:1710.04584* (2017).

[162] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[163] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.

[164] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. 2018. Fast Triangle Counting Using Cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[165] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

[166] Yunquan Zhang, Ting Cao, Shigang Li, Xinhui Tian, Liang Yuan, Haipeng Jia, and Athanasios V Vasilakos. 2016. Parallel processing systems for big data: a survey. *Proc. IEEE* 104, 11 (2016), 2114–2136.

[167] Ying Zhang, Zhiqiang Zhao, and Zhuo Feng. 2018. Towards Scalable Spectral Sparsification of Directed Graphs. *arXiv preprint arXiv:1812.04165* (2018).

[168] Fang Zhou, Qiang Qu, and Hannu Toivonen. 2017. Summarisation of weighted networks. *Journal of Experimental and Theoretical Artificial Intelligence* 29, 5 (2017), 1023–1052.

[169] Xiao Zhou and Takao Nishizeki. 1994. Edge-Coloring and f-Coloring for Various Classes of Graphs. In *Algorithms and Computation, 5th International Symposium, ISAAC '94, Beijing, P. R. China, August 25-27, 1994, Proceedings*. 199–207. https://doi.org/10.1007/3-540-58325-4_182