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:

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], k²-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 graph 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...
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.

2. Lossy graph compression
   Shall we remove edges, vertices, or both?

3. Approximate graph processing & analytics
   How to make lossy compression accurate (preserve key graph properties)?
   How to make lossy compression simple, effective, and fast?

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.

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 constraints, 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.
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
Deriving a spanner for $k = 2$

Input graph:

$\begin{align*}
\text{n} &= 67 \\
\text{m} &= 132
\end{align*}$

**Graph Properties**

- Diameter of each cluster: $O(\log n)$
- $m = O(n^{3/2})$

**Kernel Instances**

- One thread executes one kernel instance

**Before Compression**

- Shortest path, length = 7
- Shortest path, length = 9

**After Compression**

- Distances preserved well
- Connectivity is preserved
- Distances are bounded

---

**Graph Compression Kernels**

- **Subgraph Compression Kernels**
  - Implementing spanners
  - Find & reduce triangles
  - Maximum-weight edges in sampled triangles will be removed
  - New MST edges
  - Minimum spanning tree weight is preserved

- **Triangle Compression Kernels**
  - Implementing Triangle Reduction, a novel graph compression method proposed in this work:
  - Overlap kernel instances
  - Edges attached to high-degree (low-degree) vertices have higher (lower) chances of being removed by edge kernels implementing spectral sparsification

- **Edge and Vertex Compression Kernels**
  - Implementing BC compression, spectral sparsification, and random uniform edge sampling

- **Betweenness Centrality**
  - Well-preserved by vertex kernels

**Example shortest path**

- From "a" to "b" via "c"

**Examples**

- Minimum spanning tree

**Differences in neighborhoods**

- Determined by

**Example neighborhood**

- Supervertices + superedges

**Example clusters**

- Dashed edges will be omitted (selected randomly)

**Before compression:**

- Shortest path, length = 7
  - Kernel instances

**After compression:**

- Shortest path, length = 9
  - Certain inter-cluster edges will be removed

**Kernel Decisions**

- A kernel decides which supernodes are connected with a superedge

**Corrections (used to decompress)**

- Differences in neighborhoods are determined by $c$

**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.
### Table 2: Considered compression schemes

<table>
<thead>
<tr>
<th>Compression scheme</th>
<th>#remaining edges</th>
<th>Work</th>
<th>W, D</th>
<th>Storage</th>
<th>Preserves best...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral sparsification</td>
<td>( \min(\log n, \text{log} n \cdot m + O(1)) )</td>
<td>( m + O(1) )</td>
<td>( O(m + n) )</td>
<td>Graph spectra</td>
<td></td>
</tr>
<tr>
<td>Edge sampling</td>
<td>( (1 - \epsilon) \cdot m )</td>
<td>( m + O(1) )</td>
<td>( O(m + n) )</td>
<td>Triangle count</td>
<td></td>
</tr>
<tr>
<td>Triangle reduction</td>
<td>( m - \epsilon \cdot T )</td>
<td>( O(n^2) ) or ( O(m^{3/2}) )</td>
<td>( O(m) )</td>
<td>Several ( (\delta) ) Distances</td>
<td></td>
</tr>
<tr>
<td>Spanners ((\mathcal{E}k))-spanner</td>
<td>( O(n^{k/2}) )</td>
<td>( m \leq 2m^k )</td>
<td>( O(m^2) )</td>
<td>( O(m + n) ) Count of common neighbors</td>
<td></td>
</tr>
</tbody>
</table>

Past schemes for lossy graph compression (some might be integrated with Slim Graph in future versions):

<table>
<thead>
<tr>
<th>(§ 4.6) Lossy summarization with Jaccard similarity (SWG [141])</th>
<th>( O(n \log n e^2) )</th>
<th>( O(m + n) )</th>
<th>Unknown</th>
</tr>
</thead>
</table>

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

```c
1 /********** Single-edge compression kernels (§ 4.2) **********/
2 spectral_sparsify(E e) { /* More details in § 4.2.1 */
3 double V = SG.connectivity(spectral_parameter());
4 double edge_stays = min(1.0, V / 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(triangle);
17 }
18 p-1-reduction EO(vector<E> triangle) {
19 double tr_stays = SG.p;
20 if(tr_stays < SG.rand(0,1))
21 atomic (if(e.considered) SG.del(e);
22 else e.considered = true; }
23 /********** Single-vertex compression schemes (§ 4.4) **********/
24 low_degree(V v) {
25 if(v.deg=v or v.deg==1) atomic SG.del(v);
26 }
27 /********** Subgraph compression schemes (§ 4.5) **********/
28 derive_spanner(vector<E> subgraph) { /* Details in § 4.5.3 */
29 // Remove “subgraph” with a spanning tree
30 // Leave only one edge going to any other subgraph
31 vector<vector<E>> 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<E> cluster) { /* Details in § 4.5.4 */
36 // Create a supernode “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 subgraph
40 vector<E> intra = SG.summary.select(cluster, SG.e);
41 SG.corrections_plus.append(intra);
42 if(intra) // Iterate over all clusters connected to “cluster”:
43 foreach(vector<E> cl: SG.out_clusters(out_edges(cluster)))
44 // [E, vector<E>(se, inter) = SG.superedge(cluster, cl, SG.e);
45 SG.summary.insert(se);
46 SG.corrections_minus.append(inter);
47 }
48 SG.update_convergence();
49 }
```

### 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 sparsification, 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 bipartitens or spectral sparsifiers, 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 and identified a method that needs only \( O(m + n) \) storage and \( O(m) \) time (others require \( O(n^2) \) 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 \( \log(n) [148] \) \((Y = p \log(n)) or to the average vertex degree \([82] \( Y = pm/n; \) \( p \) is a user parameter. Then, each edge \((u,v)\) stays in the compressed graph with probability \( p_{uv} = \min(1, Y/\min(d_u, d_v)) \). If the output graph must be weighted, then we set \( W(u,v) = 1/p_{uv} \). 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 \( p_{uv} \) of sampling \( e \)) is derived based on \( Y \) (a parameter maintained in \( SG \) and pre-initialized 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 uniform sampling in Slim Graph.
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 removal 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. 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

```c
Listing 2: Overview of Slim Graph runtime execution for subgraph kernels.
```
We focus on SWeG, a recent scheme that exhaustively analyzed works on spanners. After that, for each subgraph, we derive_spanner of low-diameter decompositions or clustering. From populated mappings, the Slim Graph runtime derives subgraphs that are processed by kernel instances.

4.5.3 Spanners with Slim Graph. An \((\alpha, \beta)\)-spanner [121] is a subgraph \(H = (V, E')\) of \(G = (V, E)\) such that \(E' \subseteq E\) and
\[ \text{dist}_G(u,v) \leq \text{dist}_H(u,v) \leq \alpha \cdot \text{dist}_G(u,v) + \beta, \quad \forall u,v \in V. \]
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 \(e_1, ..., e_t\) 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 e-summaries (e-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 \(\epsilon\) 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; \(\epsilon\) 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 sgr_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 \(O(n^2)\) and \(O(n^2)\) 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 \(O(n)\) work. Edge kernels are less limited but they take \(O(m)\) work. Triangle kernels are even more expressive but take \(O(m^2/2)\) 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 \((O(m) \text{ instead of } O(n^2))\).

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. \( \bar{E}_{cr} \) are critical edges in the compressed graph, for a traversal starting from the same root. Now, the fraction \( |\bar{E}_{cr}|/|E_{cr}| \) indicates the change in the number of critical edges.

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 3\( d \) 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 \( 2x \), 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 \( P \)) has length at most \( P(1 + p/3) \) 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 \( pP/3 \) 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 \( P(1 + p) \) w.h.p., if \( P \) 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.

![Figure 4: Edge types considered in Slim Graph when analyzing the outcome of BFS.](image-url)
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 \(M_C\)) is at most 1/3\(M_C\).

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

\[
\alpha = \max_{\emptyset \subsetneq S \subseteq V} \left\lfloor \frac{m(S)}{|S|-1} \right\rfloor.
\]

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

Now, consider a set \(S\) of vertices. The maximum number of edges that can be deleted 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 \(2\alpha / 3\), which is at least \(\frac{4}{3}C_G\).

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 \(O(n^{2+2/k})\) 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 \(l = \frac{n^{1/2} \log n}{n} + 1\) and using \(1 - x \leq e^{-x}\), we get for the probability that a single vertex has an edge to more than \(l\) clusters:

\[
\frac{1}{e} \left(1 - \frac{n^{-1/k}}{\log n}\right)^{n^{1/2} \log n} = n^{-2}\text{.}
\]

A union bound over all vertices, the probability that a vertex has edges to more than \(l = O(n^{1/2} \log n)\) clusters is at most \(n^{-2}\). Hence, there is a greedy coloring which uses at most \(O(n^{1/2} \log n)\) 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.\(^3\). 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 \(O(d)\) (for \(d\) is the 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

---

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 \(\epsilon\)-summaries) return a subgraph of the original graph, \(m, C_G, \alpha, d, T, \text{ and } M_C\) never increase. Moreover, \(P, \mathcal{T}, D, C, \epsilon\) and \(E_{1-}\) do not decrease during compression. \(\epsilon\) is a parameter that controls how well a spectral sparsifier approximates the original graph spectrum.

| Scheme                  | \(|V|\) | \(|E|\) | Shortest path length | Average path length | Diameter | Average degree | \#Triangles | #Connected components | Coloring number | Max. indep. set size | Max. cardinal matching size |
|-------------------------|--------|--------|----------------------|--------------------|----------|---------------|--------------|----------------------|----------------|----------------------|--------------------------|
| Original graph          | \(n\)  | \(m\)  | \(P\)               | \(\mathcal{T}\)     | \(D\)    | \(\alpha\)    | \(T\)        | \(C\)                     | \(E_{1-}\)           | \(E_{2-}\)           | \(E_{3-}\)               |
| Lossy \(\epsilon\)-summary | \(n \pm 2\epsilon m\) | \(1, \ldots, n\) | \(P\)               | \(\mathcal{T}\)     | \(D\)    | \(\epsilon\)    | \(T\)        | \(\epsilon\)         | \(E_{1-}\)           | \(E_{2-}\)           | \(E_{3-}\)               |
| Simple \(p\)-sampling   | \((1 - p)m\) | \(\infty\) | \(\infty\)          | \(\mathcal{T}\)     | \(\infty\) | \(\infty\)    | \(\infty\) | \(\infty\)         | \(E_{1-}\)           | \(E_{2-}\)           | \(E_{3-}\)               |
| Spectral \(\epsilon\)-sparsifier | \(O((n/\epsilon^2))\) | \(\leq n\) | \(\leq n\)          | \(O(1/\epsilon^2)\) | \(\geq d/(2 + \epsilon)\) | \(O(\alpha/\epsilon^2)\) | \(\geq 0\) | \(\leq n\)          | \(E_{1-}\)           | \(E_{2-}\)           | \(E_{3-}\)               |
| \(O(\kappa)\)-spanner   | \(O(n^{1-1/k})\) | \(O(\kappa P)\) | \(O(\mathcal{T})\) | \(O(kD)\) | \(O(n^{1/k})\) | \(\leq d\) | \(O(n^{1-1/k})\) | \(O(\alpha/\epsilon^2)\) | \(\geq 0\) | \(\leq n\)          | \(E_{1-}\)           | \(E_{2-}\)           | \(E_{3-}\)               |
\textbf{c}-Summary bounds the size of the \textit{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 \textbf{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 \textit{not} to advocate a single compression scheme, but to (1) confirm pros and cons of different schemes, provided in § 6, and (2) illustrate that \textit{Slim Graph} enables analysis of the associated tradeoffs.

\textbf{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 \textit{very large evaluation space} and we only summarize selected findings; full data is in the extended report.

\textbf{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.

\textbf{Machines} We use CSCS Piz Daint, a Cray with various XC\textsuperscript{*} 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, mostly a system with Intel Xeon Gold 6140 CPU @ 2.30GHz, 768GB DDR4-2666, 18 cores, and 24.75MB L3.


\textbf{Collaborations:} Actor collaboration (l-act, 2.1M, 228M), DBLP co-authorship (l-db, 1.82M, 13.6M), Citation network (patents) (l-cit, 3.7M, 16.5M), Movie industry graph (l-face, 500k, 1.5M), USA road network (l-usa, 23.9M, 58.3M), Internet topology (skitter) (l-sk, 1.69M, 118M).

\textbf{Table 1:} Considered graphs with \( n \geq 2M \) or \( m \geq 10M \) from established datasets [1, 33, 56, 94, 100]. Graphs are sorted by \( m \) in each category. For each graph, we show its “\textit{symbol used later, } n, m”.

7.1 \textbf{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 \( A \) 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 \textbf{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 \( (m) \), the higher KL divergence becomes.

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 vertices (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 results surprisingly well. For example, for the s-pok graph, respectively, removing 21%
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.

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

Further tradeoffs between accuracy and compression ratios 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 $>10\times$) 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-twi), Friendster social network (s-fsr), 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.
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, $O(\text{Im})$ for summarization, and $O(n^{3/2})$ 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% ($O(n^{3/2})$ 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).
and Applied Mathematics, 1041–1052.


[47] Chen Chen, Cindy Xide Lin, Matt Fredrickson, Mihai Chistodorescu, Xifeng Yan, and Jiawei Han. 2009. Mining graph patterns efficiently via randomized summarizations. very large data bases 2, 1 (2009), 743–783.


