Log(Graph): A Near-Optimal High-Performance Graph Representation

Maciej Besta[†], Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, Torsten Hoefler[†]
Department of Computer Science, ETH Zurich
[†]Corresponding authors (maciej.besta@inf.ethz.ch, htor@inf.ethz.ch)

ABSTRACT

Today's graphs used in domains such as machine learning or social network analysis may contain hundreds of billions of edges. Yet, they are not necessarily stored efficiently, and standard graph representations such as adjacency lists waste a significant number of bits while graph compression schemes such as WebGraph often require time-consuming decompression. To address this, we propose Log(Graph): a graph representation that combines high compression ratios with very low-overhead decompression to enable cheaper and faster graph processing. The key idea is to encode a graph so that the parts of the representation approach or match the respective storage lower bounds. We call our approach "graph logarithmization" because these bounds are usually logarithmic. Our high-performance Log(Graph) implementation based on modern bitwise operations and state-of-the-art succinct data structures achieves high compression ratios as well as performance. For example, compared to the tuned Graph Algorithm Processing Benchmark Suite (GAPBS), it reduces graph sizes by 20-35% while matching GAPBS' performance or even delivering speedups due to reducing amounts of transferred data. It approaches the compression ratio of the established WebGraph compression library while enabling speedups of up to more than 2×. Log(Graph) can improve the design of various graph processing engines or libraries on single NUMA nodes as well as distributed-memory systems.

CCS CONCEPTS

• **Information systems** → **Data structures**; **Data access methods**; **Data layout**; **Data compression**; *Storage management*; • **Theory of computation** → **Graph algorithms analysis**; **Data compression**; *Design and analysis of algorithms*; *Data structures design and analysis*; Mathematical optimization;

KEYWORDS

graph compression; graph representation; graph layout; parallel graph algorithms; ILP; succinct data structures

Code & Extended Paper Version:
http://spcl.inf.ethz.ch/Research/Performance/LogGraph

1 INTRODUCTION

Large graphs form the basis of many problems in machine learning, social network analysis, and computational sciences [65]. For example, graph clustering is important in discovering relationships in graph data. The sheer size of such graphs, up to hundreds of billions of edges, exacerbates the number of needed memory banks, increases the amount of data transferred between CPUs and memory, and may lead to I/O accesses while processing graphs. Thus, reducing the size of such graphs is becoming increasingly important.

However, state-of-the-art graph representations and compression schemes, for example the well-known WebGraph [18], use techniques such as reference encoding or interval encoding that may require costly decompression. For example, consider two vertices, v_1 and v_2, and assume that some of the neighbors of v_1 and v_2 are identical. In reference encoding, these shared neighbors are stored only once, in the adjacency array of either v_1 or v_2. The other adjacency array contains a pointer to the location of these neighbors in the first array. Such encoding may be nested arbitrarily deeply, leading to pointer chasing. Such schemes may degrade performance of *graph accesses* (e.g., verifying if an edge exists) that are performance critical operations in various graph algorithms such as triangle counting. An ideal graph representation should not only provide *high compression ratios* but also *reduce or eliminate decompression overheads when accessing a graph*.

In this work, we propose Log(Graph): a representation that achieves the above goals. The key idea is to *encode different graph elements using the associated storage lower bounds*. We apply this idea to the popular adjacency array (AA) graph representation and its elements, including vertex IDs, edge weights, offsets, and the whole arrays with offsets and with adjacency data. We call this approach "graph logarithmization" as most considered storage lower bounds are *logarithmic* (one needs at least $\lceil \log |S| \rceil$ bits to store an object from a set S). We illustrate that the main advantage of this approach is its *very low overhead of decompression* combined with *high compression ratios*. For example, the compression ratio of Log(Graph) is often negligibly lower than that of WebGraph. Yet, processing these graphs with algorithms such as BFS or PageRank is faster (up to >2×) when using the Log(Graph) schemes.

Simultaneously, we illustrate that one must be careful when selecting an element of AA to logarithmize. For example, a straightforward bit packing scheme [3, 87], in which one uses $\lceil \log n \rceil$ bits to store a vertex ID from the set of all vertices $V = \{1, ..., n\}$ in a given graph, brings only modest storage

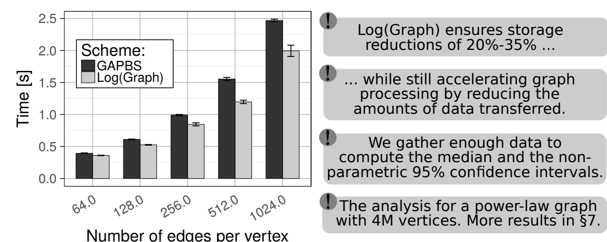


Figure 1: Log(Graph) performance with SSSP when compressing vertex IDs (the *local* approach § 3.2.2), compared to the tuned GAPBS [11].

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
PACT '18, November 1–4, 2018, Limassol, Cyprus
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5986-3/18/11...$15.00
https://doi.org/10.1145/3243176.3243198

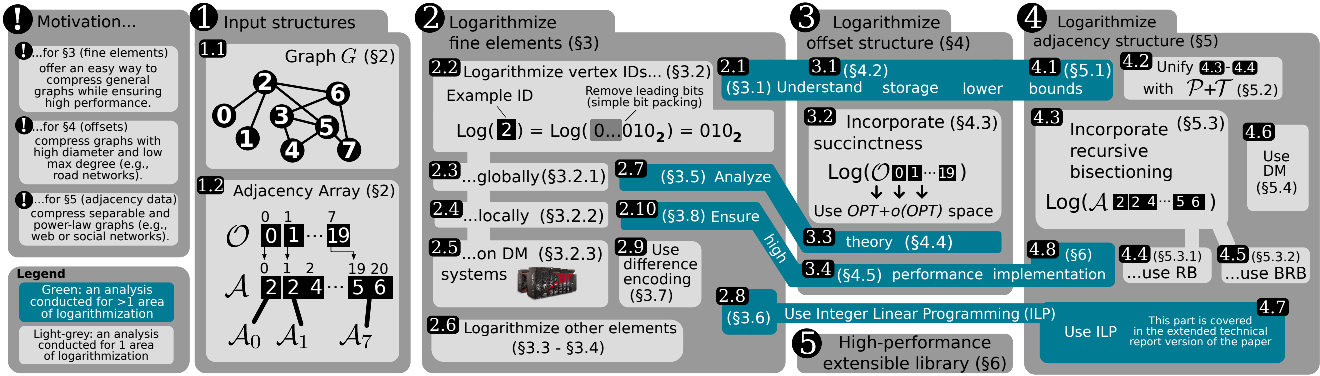


Figure 2: (§ 2) The roadmap of incorporated schemes. The green areas indicate analyzes and themes shared by multiple logarithmization areas.

reductions, as shown later (§ 7). In this particular case, we *separately consider the neighborhood N_v of each vertex v*, and encode any vertex $w \in N_v$ using $\lceil \log |N_v| \rceil$ bits.

We motivate Log(Graph) in Figure 1 with example results for the Single Source Shortest Path (SSSP) algorithm. Log(Graph) outperforms a tuned SSSP code from the GAP Benchmark Suite [11], a state-of-the-art graph benchmarking platform, by 20%, while reducing the required storage by 20-35%. Thus, Log(Graph) does *not only compress graphs with negligible preprocessing costs*, but it even *delivers speedups* by combining storage reductions with fast decompression.

To further accelerate Log(Graph) and enhance its compression ratios, we use *modern bitwise operations* to efficiently extract data from its logarithmic encoding. Moreover, we develop an Integer Linear Programming (ILP) heuristic that reorders vertex IDs to save space. Third, to compress offsets into graph adjacency data, we use *succinct data structures* [53] that approach theoretical storage lower bounds while enabling constant-time data accesses. We show that they asymptotically reduce the usual $O(n \log n)$ bits used in traditional offset arrays. In addition, we provide *the first performance analysis of succinct data structures in a parallel setting* and we conclude that, in the context of graph accesses, succinct data structures deliver nearly identical performance to that of offset arrays.

We target shared- and distributed-memory settings, and a wide selection of graph algorithms: BFS, PageRank (PR), Connected Components (CC), Betweenness Centrality (BC), Triangle Counting (TC), and SSSP. We conclude that *Log(Graph) reduces storage required for graphs while enabling low-overhead decompression, matching the performance of tuned graph processing codes, and outperforming established compression systems*.

2 BACKGROUND AND NOTATION

We first describe the used concepts and notation; see Table 1 for summarized symbols.

Graph Model 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; $|V| = n$, $|E| = m$. Vertices are identified by contiguous IDs ($\equiv$ labels) from $\{1, ..., n\}$. N_v and d_v denote the neighbors and the degree of a vertex v. $N_{i,v}$ is v's ith neighbor. $\widehat{X}$ indicates the maximum value in a given set or sequence X, for example $\widehat{N}_v$ is the maximum ID among v's neighbors. $\widehat{\mathcal{W}}$ is the maximal edge weight in a given graph G.

Adjacency Array Data Structures Log(Graph) builds upon the traditional **adjacency array** (AA) representation. One part

	Symbol	Description
Graph model	G	A graph $G = (V, E)$; V and E are sets of vertices and edges.
	n, m	Numbers of vertices and edges in G; $\lvert V\rvert = n$, $\lvert E\rvert = m$.
	$\mathcal{W}_{(v,w)}$	The weight of an edge (v, w).
	$d_v, N_v, N_{i,v}$	Degree, neighbors, and ith neighbor of a vertex v; $N_{0,v} \equiv v$.
	$\overline{x}, \widehat{x}$	The average and the maximum value in a set or sequence x.
	$\alpha, \beta; p$	Parameters of a power-law graph and an Erdős-Rényi graph.
Adjacency array	$\mathcal{A}, \mathcal{A}_v$	The adjacency array of a given graph and a given vertex.
	$\mathcal{O}, \mathcal{O}_v$	The offset structure of a given graph and an offset to $\mathcal{A}_v$.
	$\lvert\mathcal{A}\rvert, \lvert\mathcal{O}\rvert$	The sizes of $\mathcal{A}$ and $\mathcal{O}$.
	$\mathcal{L}[\mathcal{A}], \mathcal{L}[\mathcal{O}]$	Logarithmization schemes acting upon $\mathcal{A}$ and $\mathcal{O}$.
	B, L, W	Various parameters of $\mathcal{A}$ and $\mathcal{O}$; see § 4.2–§ 4.3 for details.
Machine model	N	The number of levels in a hierarchical machine.
	$H_i, \mathcal{H}$	The number of elements at level i, the number of compute nodes.
	T, P	The number of threads/processes.
	W	The memory word size [bits].
	t_x	Time to do a given operation x.
Others	$\mathcal{P}$	Permuter: function that relabels vertices.
	$\mathcal{T}_x, \mathcal{T}$	Transformers: functions that arbitrarily modify $\mathcal{A}$.
	G_x	Subgraphs of G constructed in recursive bisectioning.

Table 1: Symbols used in the paper gathered for the reader's convenience.

of AA is an array (denoted as $\mathcal{A}$) with adjacency data. $\mathcal{A}$ consists of n subarrays $\mathcal{A}_i, i \in V$. Subarray $\mathcal{A}_v$ contains neighbors of vertex v. We have $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_v\}$. Each $\mathcal{A}_v$ is sorted by vertex IDs. The second part of AA is an array $\mathcal{O}$ with offsets (or pointers) to each $\mathcal{A}_v$. $\mathcal{O}_v$ denotes the offset to $\mathcal{A}_v$. $|\mathcal{O}|$ and $|\mathcal{A}|$ denote the sizes of $\mathcal{O}$ and $\mathcal{A}$.

Machine and System Model For more storage reductions on today's hardware, we consider arbitrary hierarchical machines where, for example, cores reside on a socket, sockets constitute a node, and nodes form a rack. N is the number of hierarchy levels and H_i is the total number of elements from level i. The first level corresponds to the whole machine; thus $H_1 = 1$. We also refer specifically to the number of compute nodes as $\mathcal{H}$. Finally, the numbers of used threads per node and processes are T and P. The memory word size is W.

Roadmap of Schemes To enhance readability, we summarize Log(Graph) in Figure 2. We divide the Log(Graph) schemes into three categories, based on what they compress: fine graph elements (§ 3), offset structure $\mathcal{O}$ (§ 4), and adjacency data $\mathcal{A}$ (§ 5). Each of these sections is structured similarly. We first describe lower bounds associated with a compressed object (e.g., § 3.1) and the actual compression schemes (e.g., § 3.2–§ 3.4), then conduct a theoretical analysis (e.g., § 3.5), describe further enhancements such as an ILP heuristic (e.g., § 3.6) or gap-encoding (§ 3.7), and the implementation (e.g., § 3.8).

As we show empirically in § 7, each of the three classes of logarithmization schemes has slightly different characteristics and thus application domains. First, compressing fine graph elements (§ 3) brings *storage reductions of 20-35%* compared

	Entity	Bound	Assumptions, remarks
Fine elements (§ 3.1)	Vertex ID	$\log n$	In the global approach.
	Vertex ID	$\log \widehat{N}_v$	In the local approach.
	Offset	$\log 2m$	For unweighted graphs.
	Edge weight	$\log \widehat{\mathcal{W}}$	-
$\mathcal{O}$ (§ 4.2)	Bit vector	$\log \binom{\frac{2Wm}{B}}{n}$	n set bits among $\frac{2Wm}{B}$ bits.
$\mathcal{A}$ (§ 5.2)	Graph	$\log \binom{\binom{n}{2}}{m}$	The graph is undirected.

Table 2: Storage lower bounds for various parts of an AA. The ceiling function $\lceil \cdot \rceil$ surrounding each $\log \cdot$ was omitted for aesthetic purposes.

to the traditional AA while delivering performance close to or matching or *even exceeding* that of tuned graph processing codes. Second, compressing offset structures $\mathcal{O}$ (§ 4) can enhance *any* parallel graph processing computation because it does *not* impact performance in parallel settings while it *does* reduce storage required for offsets $\mathcal{O}$ even by >90%. Finally, adjacency data $\mathcal{A}$ (§ 5) is the most aggressive in reducing storage, in some cases by *up to* ≈*80%* compared to the adjacency array, approaching the compression ratios of modern graph compression schemes and simultaneously offering speedups of more than 2×.

3 LOGARITHMIZING FINE ELEMENTS

We first logarithmize fine elements of the adjacency array: vertex IDs, vertex offsets, and edge weights; see Figure 2 (❷).

3.1 Understanding Storage Lower-Bounds

A simple storage lower bound is the logarithm of the number of possible instances of a given entity, which corresponds to the number of bits required to distinguish between these instances. Now, bounds derived for fine-grained graph elements are illustrated in Table 2. First, a storage lower bound for a single vertex ID is $\lceil \log n \rceil$ bits as there are n possible numbers to be used for a single vertex ID. Second, a corresponding bound to store an offset into the neighborhood of a single vertex is $\lceil \log 2m \rceil$; this is because in an undirected graph with m edges there are $2m$ cells. Third, a storage lower bound of an edge weight from a discrete set $\left\{0, ..., \widehat{\mathcal{W}}\right\}$ is $\left\lceil \log \widehat{\mathcal{W}} \right\rceil$ (for continuous weights, we first scale them appropriately to become elements of a discreet set).

3.2 Logarithmization of Vertex IDs

We first logarithmize vertex IDs.

3.2.1 Vertex IDs: The Global Approach. The first and simplest step in Log(Graph) is to use $\lceil \log n \rceil$ bits to store a vertex ID in a graph with n vertices. In this case, the total size of the adjacency data $|\mathcal{A}|$ is $2m\lceil \log n \rceil$. This simple bit packing was mentioned in past work [3, 87] and it modestly reduces $|\mathcal{A}|$ (as we show in § 7). We now *extend it with schemes that deliver more storage reductions and performance* (§ 3.2.2–§ 3.2.3).

3.2.2 Vertex IDs: The Local Approach. Even if the above global approach uses an optimum number or bits to store a vertex ID, it may be far from optimum when considering *subsets* of these vertices. For example, consider a vertex v with very few neighbors ($d_v \ll n$) that all have small IDs ($\widehat{N}_v \ll n$). Here, the optimum number of bits for a vertex ID in $\mathcal{A}_v$ is $\left\lceil \log \widehat{N}_v \right\rceil$, *which may be much lower than* $\lceil \log n \rceil$. However, one must also keep the information on the number of bits required for each N_v. We use a fixed number of bits; it is lower bounded by $\left\lceil \log \log \widehat{N}_v \right\rceil$ bits. The size of $\mathcal{A}$ is in this case

$$|\mathcal{A}| = \sum_{v \in V} \left(d_v \left\lceil \log \widehat{N}_v \right\rceil + \left\lceil \log \log \widehat{N}_v \right\rceil \right)$$

3.2.3 Vertex IDs in Distributed-Memories. We now extend vertex logarithmization to the distributed-memory setting. We divide a vertex ID into an *intra* part that ensures the uniqueness of IDs within a given machine element (e.g., a compute node), and an *inter* part that encodes the position of a vertex in the distributed-memory structure. The intra part can be encoded with either the local or the global approach.

We first only consider the level of compute nodes; each node constitutes a cache-coherent domain and they are connected with non-coherent network. The number of vertices in one node is $\frac{n}{\mathcal{H}}$. The intra ID part takes $\lceil \log \frac{n}{\mathcal{H}} \rceil$; the inter one takes $\lceil \log \mathcal{H} \rceil$. As the inter part is unique for a given node, it is stored once per node. Thus

$$|\mathcal{A}| = n \left\lceil \log \frac{n}{\mathcal{H}} \right\rceil + \mathcal{H} \lceil \log \mathcal{H} \rceil$$

Next, we consider the arbitrary number of memory hierarchy levels. Here, the number of vertices in one element from the bottom of the hierarchy (e.g., a die) is $\frac{n}{H_N}$. Thus, the intra ID part requires $\left\lceil \log \frac{n}{H_N} \right\rceil$ bits. The inter part needs $\sum_{j \in \{2..N-1\}} \left\lceil \log H_j \right\rceil$ bits and has to be stored once per each machine element, thus

$$|\mathcal{A}| = n \left\lceil \log \frac{n}{H_N} \right\rceil + \sum_{j=2}^{N-1} H_j \left\lceil \log H_j \right\rceil$$

3.3 Logarithmization of Edge Weights

We similarly condense edge weights. The storage lower bound for storing a maximal edge weight is $\left\lceil \log \widehat{\mathcal{W}} \right\rceil$ bits. Thus, if G is weighted, we respectively have (for the global and local approach applied to the weights)

$$|\mathcal{A}| = 2m \left(\lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right)$$

$$|\mathcal{A}| = \sum_{v \in V} \left(d_v \left(\left\lceil \log \widehat{N}_v \right\rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) + \left\lceil \log \log \widehat{N}_v \right\rceil + \left\lceil \log \log \widehat{\mathcal{W}} \right\rceil \right)$$

3.4 Logarithmization of Single Offsets

Finally, one can also "logarithmize" other AA elements, including offsets. Each offset must be able to address any position in $\mathcal{A}$ that may reach $2m$. Thus, the related lower bound is $\lceil \log 2m \rceil$, giving $|\mathcal{O}| = n \lceil \log 2m \rceil$.

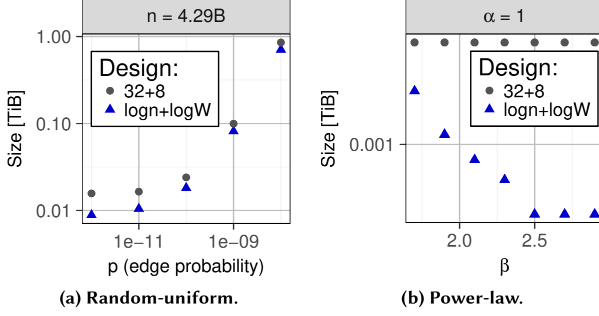
3.5 Theoretical Storage Analysis

Next, we show how the above schemes reduce the size of graphs generated using two synthetic graph models (random uniform and power-law) for the global approach.

Erdős-Rényi (Uniform) Graphs We start with Erdős-Rényi random uniform graphs. Here, every edge is present with probability p. The expected degree of any vertex is pn, thus

$$E[|\mathcal{A}|] = \left(\lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) pn^2$$

$$E[|\mathcal{O}|] = n \left\lceil \log \left(2pn^2 \right) \right\rceil = n \lceil \log 2p + 2 \log n \rceil$$



(a) Random-uniform. **(b) Power-law.**

Figure 3: (§ 3.5) The analysis of the size of random-uniform and power-law graphs with Log(Graph) and traditional adjacency array.

Power-Law Graphs We next analyze power-law graphs; the derivation is in extended version of the paper[1]. Here, the probability that a vertex has degree d is $f(d) = \alpha d^{-\beta}$, and

$$E[|\mathcal{A}|] \approx \frac{\alpha}{2-\beta}\left(\left(\frac{\alpha n \log n}{\beta - 1}\right)^{\frac{2-\beta}{\beta-1}} - 1\right)\left(\lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil\right)$$

The results are in Figure 3. "32+8" indicates an AA with 32 bits for a vertex ID and 8 bits for a weight; the other target is Log(Graph). Compressing fine-grained elements consistently reduces storage. Yet, it may offer suboptimal space and performance results as it ignores the *structure of the graph* and *the structure of the memory with fixed-size words*. We now address these issues with ILP and gap encoding (for less storage) and efficient design (for more performance).

3.6 Adding Integer Linear Programming

We now enhance the local logarithmization (§ 3.2) to further reduce $|\mathcal{A}|$. Consider any $\mathcal{A}_v$. We observe that a single neighbor in $\mathcal{A}_v$ with a large ID may vastly increase $|\mathcal{A}_v|$: we use $\left\lceil \widehat{N}_v \right\rceil$ bits to store each neighbor in $\mathcal{A}_v$ but other neighbors may have much lower IDs. Thus, we permute vertex IDs to *reduce maximal IDs in as many neighborhoods as possible* without modifying the graph structure. We illustrate an ILP formulation and then propose a heuristic.

The new objective function is shown in Eq. (1). It minimizes the weighted sum of $\widehat{N}_v, v \in V$. Each maximal ID is given a positive weight which is the inverse of the neighborhood size; this intuitively decreases $\widehat{N}_v$ in smaller $\mathcal{A}_v$.

$$\min \sum_{v \in V} \widehat{N}_v \cdot \frac{1}{d_v} \quad (1)$$

In Constraint (2), we set $\widehat{N}_v$ to be the maximum of the new IDs assigned to the neighbors of v. $\mathcal{N}(v)$ is the new ID of v.

$$\forall_{v,u \in V}\ (u \in N_v) \ \Rightarrow \left[\mathcal{N}(u) \leq \widehat{N}_v\right] \quad (2)$$

Listing 1 describes a greedy polynomial time heuristic for changing IDs. We sort vertices in the increasing order of their degrees (Line 8). Next, we traverse vertices in the sorted order, beginning with the smallest $|\mathcal{A}_v|$, and assign a new smallest ID possible (Line 9). The remaining vertices that are not renamed by now are relabeled in Line 16. This scheme acts similarly to the proposed ILP.

[1]https://spcl.inf.ethz.ch/Research/Performance/LogGraph

```
1 /* Input: graph G, Output: a new relabeling N(v),∀v ∈ V. */
2 void relabel(G) {
3  ID[0..n−1] = [0..n−1]; //An array with vertex IDs.
4  D[0..n−1] = [d_0..d_{n−1}]; //An array with degrees of vertices.
5  //An auxiliary array for determining if a vertex was relabeled:
6  visit[0..n−1] = [false..false];
7  nl = 1; //An auxiliary variable ``new label''.
8  sort(ID); sort(D);
9  for(int i = 1; i < n; ++i) //For each vertex...
10  for(int j = 0; j < D[i]; ++j) { //For each neighbor...
11   int id = N_{j,ID[i]}; //N_{j,ID[i]} is jth neighbor of vertex with ID ID[i]
12   if(visit[id] == false) {
13    N(id) = nl++;
14    visit[id] = true;
15  }}
16  for(int i = 1; i < n; ++i)
17   if(visit[i] == false)
18    N(id) = nl++;
19 }
```

Listing 1: (§ 3.6) The greedy heuristic for vertex relabeling.

3.7 Adding Fixed-Size Gap Encoding

We next use gap encoding to further reduce $|\mathcal{A}|$. Traditionally, in gap encoding one calculates differences between all consecutive neighbors in each $\mathcal{A}_v$. These differences are then encoded with a variable-length code such as Varint [16]. Now, this may entail significant decoding overheads. We alleviate these overheads with *fixed-size gap encoding* where the maximum difference within a given neighborhood determines the number of bits used to encode other differences in this neighborhood. In the local approach (§ 3.2.2), the maximum difference in each $\mathcal{A}_v$ determines the number of bits to encode any difference in the same $\mathcal{A}_v$.

3.8 High-Performance Implementation

We finally describe the high-performance implementation. We focus on the global approach due to space constraints, the local approach entails an almost identical design and is presented fully in the extended paper version[1].

Bitwise Operations We analyzed Intel bitwise operations to ensure the fastest implementation. Table 3 presents the used operations together with the number of CPU cycles that each operation requires [47].

Name	C++ syntax	Description	Cycles
BEXTR	_bextr_u64	Extracts a contiguous number of bits.	2
SHR	>>	Shifts the bits in the value to the right.	1
AND	&	Performs a bitwise AND operation.	1
ADD	+	Performs an addition between two values.	2

Table 3: (§ 3.8) The utilized Intel bitwise operations.

Accessing an Edge ($N_{i,v}$) We first describe how to access a given edge in $\mathcal{A}$ that corresponds to a given neighbor of v ($N_{i,v}$); see Listing 2 for details. The main issue is to access an s-bit value from a byte-addressable memory with $s = \lceil \log n \rceil$. In short, we fetch a 64-bit word that contains the required s-bit edge. In more detail, we first load the offset $\mathcal{O}[v]$ of v's neighbors' array (Line 3). This usually involves a cache miss, taking t_{cm}. Second, we derive $o = s \cdot \mathcal{O}[v]$ (the exact bit position of $N_{i,v}$); it takes t_{mul} (Line 3). Third, we find the closest byte alignment before o by right-shifting o by 3 bits, taking t_{shf} (Line 4). Instead of byte alignment we also considered any other alignment but it entailed negligible (<1%) performance differences. Next, we derive the distance d from this alignment with a bitwise and acting on o and binary 111, taking t_{and}. We can then access the derived 64-bit value; this involves another cache miss (t_{cm}). If we shift this value by d bits and mask it, we obtain $N_i(v)$. Here, we use the x86 bextr instruction that combines these two operations

and takes t_{bxr}. In the local approach, we also maintain the bit length for each neighborhood. It is stored next to the associated offset to avoid another cache miss.

```
1 /* v_ID is an opaque type for IDs of vertices. */
2 v_ID N_{i,v}(v_ID v, int32_t i, int64_t* O, int64_t* A, int8_t s){
3   int64_t exactBitOffset = s * (O[v] + i);
4   int8_t* address = (int8_t*) A + (exactBitOffset >> 3);
5   int64_t distance = exactBitOffset & 7;
6   int64_t value = ((int64_t*) (address))[0];
7   return _bextr_u64(value, distance, s); }
```

Listing 2: (§ 3.8) Accessing an edge in Log(Graph) ($N_i(v)$).

Accessing Neighbors (N_v) Once we have calculated the exact bit position of the first neighbor as described in Listing 2, we simply add $s = \lceil \log n \rceil$ to obtain the bit position of the next neighbor. Thus, the multiplication that is used to get the exact bit position is only needed for the first neighbour, while others are obtained with additions instead.

Accessing a Degree (d_v) d_v is simply calculated as the difference between two offsets: $\mathcal{O}[v+1] - \mathcal{O}[v]$.

Accessing an Edge Weight We store the weight of each edge directly after the corresponding vertex ID in $\mathcal{A}$. The downside is that every weight is thus stored twice for undirected graphs. However, it enables accessing the weight and the vertex ID together. We model fetching the ID and the weight as a single cache line miss overhead t_{cm}.

Performance Model We finally present the performance model that we use to understand better the behavior of Log(Graph). First, we model accessing an edge ($N_{i,v}$) as

$$t_{edge} = 2t_{cm} + t_{mul} + t_{shf} + t_{and} + t_{bxr}$$

The model for accessing N_v looks similar with the difference that only one multiplication is used:

$$t_{neigh}(v) = t_{edge} + (d_v - 1)(t_{add} + t_{shf} + t_{and} + t_{bxr})$$

As $\mathcal{A}$ is contiguous we assume there are no more cache misses from prefetching. Now, we model the latency of d_v as

$$t_{degree} = 2t_{cm} + t_{sub}$$

4 LOGARITHMIZING OFFSETS

We now logarithmize *the whole offset structure* $\mathcal{O}$, treating it as a single entity with its own associated storage lower bound. Our main technique for combining storage reductions and low-overhead decompression is to *store the offsets* $\mathcal{O}$ *as a bitvector* and then *encode it as a succinct bit vector* [43] that approaches the storage lower bound while providing fast accesses to its contents.

4.1 Arrays of Offsets vs. Bit Vectors for $\mathcal{O}$

Usually, $\mathcal{O}$ is an array of n offsets and the size of $\mathcal{O}$, $|\mathcal{O}|$, is much smaller than $|\mathcal{A}|$. Still, in sparse graphs with low maximal degree $\hat{d}$, $|\mathcal{O}| \approx |\mathcal{A}|$ or even $|\mathcal{O}| > |\mathcal{A}|$. For example, for the USA road network, if $\mathcal{O}$ contains 32-bit offsets, $|\mathcal{O}| \approx 0.83|\mathcal{A}|$. To reduce $|\mathcal{O}|$ in such cases, one can use a bit vector instead of an array of offsets. For this, $\mathcal{A}$ is divided into blocks (e.g., bytes or words) of a size B [bits]. Then, if the ith bit of $\mathcal{O}$ is set (i.e., $\mathcal{O}[i] = 1$) and if this is the jth set bit in $\mathcal{O}$, then $\mathcal{A}_j$ starts at the ith block. The key insight is that getting the position of jth set bit and thus the offset of the array $\mathcal{A}_j$ is equivalent to performing a certain operation called $select_{\mathcal{O}}(j)$ that returns the position of the jth one in $\mathcal{O}$. Yet, *select* on a raw bit vector of size L takes $O(L)$ time. Thus, we first incorporate two designs that enhance *select*: *Plain* (bvPL) and *Interleaved* (bvIL) bit vectors [44]. They both trade some space for a faster *select*. bvPL uses up to $0.2|\mathcal{O}|$ additional bits in an auxiliary data structure to enable *select* in $O(1)$ time. In bvIL, the original bit vector data is interleaved (every L bits) with 64-bit cumulative sums of set bits up to given positions; *select* has $O(\log |\mathcal{O}|)$ time [44]. Neither bvPL nor bvIL are succinct; we use them (1) as reference points and (2) because they also enable a smaller yet simple offset structure $\mathcal{O}$.

4.2 Understanding Storage Lower Bounds

A bit vector that serves as an $\mathcal{O}$ and corresponds to an offset array of a G takes $\frac{2Wm}{B}$ bits. This is because it must be able to address up to $2m \cdot W$ bits (there are $2m$ edges in $\mathcal{A}$, each stored using W bits in a memory word) grouped in blocks of size B bits. There are exactly $\mathcal{Q} = \binom{\frac{2Wm}{B}}{n}$ bit vectors of length $\frac{2Wm}{B}$ with n ones and the storage lower bound is $\lceil \mathcal{Q} \rceil$ bits.

4.3 Incorporating Succinct Bit Vectors

To reduce the size of $\mathcal{O}$ and improve the performance of its *select* query, we use *succinct bit vectors*.

Succinct Data Structures Assume OPT is the optimal number of bits to store some data. A representation of this data is *succinct* if it uses $OPT + o(OPT)$ bits *and* if it supports *a reasonable set of queries in (ideally)* $O(1)$ *time* [16]. Thus, succinct designs differ from compression mechanisms such as zlib as they do not entail expensive decompression.

Succinct Bit Vectors: Preliminaries Succinct bit vectors use $\lceil \mathcal{Q} \rceil + o(\mathcal{Q})$ bits assuming a storage lower bound of $\lceil \mathcal{Q} \rceil$ and they answer the *select* query in $o(\mathcal{Q})$ time. Many such designs exist [78] and are widely used in space-efficient trees and other schemes such as dictionaries. The high-level idea behind their design is to divide the bit vector to be encoded into *small* parts (i.e., contiguous bit vector chunks of equal sizes), group these chunks in an auxiliary table, and represent them with the indices into this table [13]. This table should contain *all* possible chunks so that *any* bit vector could be constructed from them. These small chunks are again divided into yet smaller (*tiny*) chunks, stored similarly in other auxiliary tables. Now, the size of both small and tiny chunks is selected in such a way that the sum of the sizes of all the indices and all the auxiliary tables is $\lceil \mathcal{Q} \rceil + o(\mathcal{Q})$. The central observation that enables these bounds is that the bit vector representation consisting of small and tiny chunks can be *hierarchical*: tiny chunks only need pointers

Succinct Bit Vectors in Log(Graph) First, we use the *entropy based* bit vector (bvEN) [78]. The key idea behind bvEN is to use a dictionary data structure [23] that achieves the lower bound for storing bit vectors of length $2Wm/B$ with n ones. Second, we use *sparse* succinct bit vectors (bvSD) [76]. Here, positions of ones are represented as a sequence of integers, which is then encoded using the Elias-Fano scheme for non-decreasing sequences. As bvSD specifically targets sparse bit vectors, we expect it to be a good match for various graphs where $m = O(n)$. Third, we investigate the *B-tree based* bit vector (bvBT) [1]. This data structure supports inserts, making the bit vector dynamic. It is implemented with B-trees where leaves contain the actual bit vector data while internal nodes contain meta data for more performance. Finally, the

gap-compressed (bvGC) dynamic variant is incorporated [1] that along with bvSD also compresses sequences of zeros.

4.4 Theoretical Storage & Time Analysis

We now analyze the storage/time complexity of the described offset structures in Table 4. For completeness, we present the asymptotic and the exact size as well the time to derive $\mathcal{O}_v$. Now, ptrW (array of offsets) together with bvPL and bvSD feature the fastest $\mathcal{O}_v$; we select them as the most promising candidates for $\mathcal{O}$ in Log(Graph).

$\mathcal{O}$	ID	**Asymptotic size** [bits]	**Exact size** [bits]	*select* or $\mathcal{O}[v]$
Pointer array	ptr$W$	$O(Wn)$	$W(n+1)$	$O(1)$
Plain [44]	bvPL	$O\left(\frac{Wm}{B}\right)$	$\frac{2Wm}{B}$	$O(1)$
Interleaved [44]	bvIL	$O\left(\frac{Wm}{B} + \frac{Wm}{L}\right)$	$2Wm\left(\frac{1}{B} + \frac{64}{L}\right)$	$O\left(\log \frac{Wm}{B}\right)$
Entropy based [31, 78]	bvEN	$O\left(\frac{Wm}{B} \log \frac{Wm}{B}\right)$	$\approx \log \binom{\frac{2Wm}{B}}{n}$	$O\left(\log \frac{Wm}{B}\right)$
Sparse [76]	bvSD	$O\left(n + n \log \frac{Wm}{Bn}\right)$	$\approx n\left(2 + \log \frac{2Wm}{Bn}\right)$	$O(1)$
B-tree based [1]	bvBT	$O\left(\frac{Wm}{B}\right)$	$\approx 1.1 \cdot \frac{2Wm}{B}$	$O(\log n)$
Gap-compressed [1]	bvGC	$O\left(\frac{Wm}{B} \log \frac{Wm}{Bn}\right)$	$\approx 1.3 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn}$	$O(\log n)$

Table 4: (§ 4.3) Theoretical analysis of various types of $\mathcal{O}$ and time complexity of the associated queries.

4.5 High-Performance Implementation

For high performance, we use the sdsl-lite library [45] that provides fast codes of various succinct and compact bit vectors. Yet, it is fully sequential and oblivious to the utilized workload. Thus, we evaluate its performance tradeoffs (§ 7) and identify the best designs for respective graph families, illustrating that the empirical results follow the theoretical analysis from § 4.4.

5 LOGARITHMIZING ADJACENCY DATA

In this section, we logarithmize the adjacency data $\mathcal{A}$. $\mathcal{A}$ is usually more complex than $\mathcal{O}$ as it encodes the whole structure of a graph. To facilitate logarithmizing $\mathcal{A}$, we first develop a formal model and we show that various past schemes are its special cases. We illustrate that these schemes entail inherent performance or storage issues and we then propose novel schemes to overcome these problems.

Similarly to compressing fine elements and offsets, the schemes from this section provide low-overhead decompression combined with large storage reductions, enabling high-performance graph processing running over compressed graphs. The difference is that *the main focus is on reducing storage overheads with performance being the secondary priority*, while compressing fine elements comes with *reverse* objectives. Selecting the most appropriate set of schemes depends on the specific requirements of the user of Log(Graph).

5.1 A Model for Logarithmizing $\mathcal{A}$

Log(Graph) comes with many compression schemes for $\mathcal{A}$ that target various classes of graphs. We define any such scheme to be a tuple $(\mathcal{P}, \mathcal{T})$. $\mathcal{P}$ is the *permuter*: a function that relabels the vertices. We introduce $\mathcal{P}$ to explicitly capture the notion that appropriate labeling of vertices significantly reduces $|\mathcal{A}|$. We have $\mathcal{P} : V \to \mathbb{N}$ such that (the condition enforces the uniqueness of IDs)):

$$\forall_{v,u \in V} \ (v \neq u) \Rightarrow [\mathcal{P}(v) \neq \mathcal{P}(u)]$$

Next, $\mathcal{T} = \{\mathcal{T}_x \mid x \in \mathbb{N}\}$ is a *set* of *transformers*: functions that map sequences of vertex labels into sequences of bits:

$$\mathcal{T}_x : \overbrace{\widehat{V} \times ... \times \widehat{V}}^{x \text{ times}} \to \{0,1\} \times ... \times \{0,1\}$$

We introduce $\mathcal{T}$ to enable arbitrary operations on sequences of relabeled vertices, for example be the Varint encoding [34].

5.2 Understanding Storage Lower Bounds

$\mathcal{A}$ is determined by the corresponding G and thus a simple storage lower bound is determined by the number of graphs with n vertices and m edges and equals $\left\lceil \log \binom{\binom{n}{2}}{m} \right\rceil$ (Table 2). Now, today's graph codes already approach this bound. For example, the Graph500 benchmark [73] requires ≈1126 TB for a graph with 2^{42} vertices and 2^{46} edges while the corresponding lower bound is merely ≈350 TB. We thus propose to not only focus on general graphs, but to assume more about G's structure besides the vertex and edge count, and to use this to further reduce the storage for G. A class of graphs that was shown to model well the structure of community-driven graph datasets are *separable* graphs [16]. Intuitively, a graph G is vertex (or edge) **separable** if we can divide its set of vertices V into two subsets of vertices of approximately the same size so that the size of a vertex (or edge) cut between these two subsets is, intuitively, much smaller than $|V|$ (the full formal specification can be found in the extended technical report[2]).

5.3 Recursive Bisectioning

One technique that we use to reduce $|\mathcal{A}|$ and decompression overheads for separable graphs is *recursive bisectioning*. We first describe an existing scheme (§ 5.3.1) and then enhance it for more performance (§ 5.3.2).

5.3.1 Recursive Bisectioning (RB). Here, we first illustrate a representation introduced by Blandford et al. [16] (referred to as the *RB* scheme) that requires $O(n)$ bits to store a separable graph. Figure 4 contains an example. Essentially, vertices are relabeled to minimize differences between the labels of consecutive neighbors of each vertex v in each adjacency array. Then, the differences are recorded with any variable-length gap encoding scheme such as Varint [26, 34]. Assuming that the new labels of v's neighbors do not differ significantly, the encoded gaps use less space than full labels [16]. Now, to reassign labels in such a way that the storage is reduced, the graph (see ❶ in Figure 4 for an example) is bisected recursively until the size of a partition is one vertex (for edge cuts) or a pair of connected vertices (for vertex cuts); in the example we focus on edge cuts. Respective partitions form a binary *separator tree* with the leaves being single vertices ❷. Then, the vertices are relabeled as imposed by an inorder traversal over the *leaves* of the separator tree ❸. The first leaf visited gets the lowest label (e.g., 0); the label being assigned is incremented for each new visited leaf. This minimizes the differences between the labels of the neighboring vertices (the leaves corresponding to the neighboring vertices are close to one another in the separator tree), reducing AA's size ❹, ❺.

Permuters/Transformers $\mathcal{P}$ relabels vertices according to the order in which they appear as leaves in the inorder traversal of the separator tree obtained after recursive bisectioning. Each transformer $\mathcal{T} = \{\mathcal{T}_v(v, N_v)\}$ takes as input v and N_v.

[2]https://spcl.inf.ethz.ch/Research/Performance/LogGraph

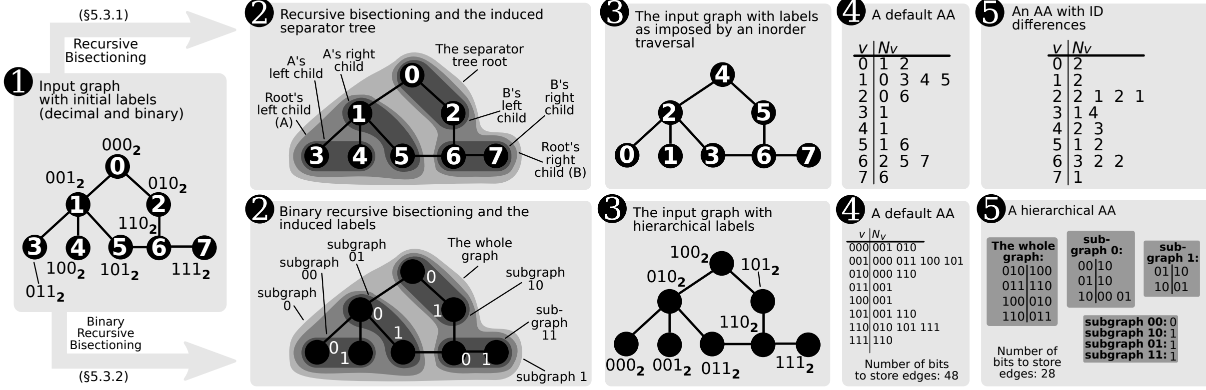


Figure 4: An example graph representation logarithmized with Recursive Bisectioning (RB [16], § 5.3.1), and Binary Recursive Bisectioning (BRB, § 5.3.2).

It then encodes the differences between consecutive vertex labels using Varint.

Problems RB suffers from expensive preprocessing, as we illustrate later in § 7 (Table 6). Generation of RB usually takes more than 20× longer than that of AA.

5.3.2 Binary Recursive Bisectioning (BRB). We now propose Binary RB (BRB), a scheme that alleviates RB's preprocessing costs. The idea is to relabel vertices so that vertices in clusters have large common prefixes (clusters are identified during bisectioning). One prefix is stored only once per each cluster.

Permuter (Relabeling Vertices) We present an example of a permuter in BRB in Figure 4 (the bottom series of numbers ❷ – ❺). First, we recursively bisect the input graph G to identify common prefixes and uniquely relabel the vertices. After the first bisectioning, we label an arbitrarily selected subgraph as 0 and the other as 1. We denote these subgraphs as G_0 and G_1, respectively. We then apply this step recursively to each subgraph for the specified number of steps or until the size of each partition is one (i.e., each partition contains only one vertex). G_0 would be bisected into subgraphs G_{00}, G_{01} with labels 00 and 01 (we refer to a partition with label X as G_X). Eventually, each vertex obtains a unique label in the form of a binary string. Each bit of this label identifies each partition that the vertex belongs to.

Transformer (Encoding Edges) Here, the idea is to group edges within each subgraph derived in the process of hierarchical vertex labeling. Several leading bits are identical in each label and are stripped off, decreasing $|\mathcal{A}|$. To make such a hierarchical adjacency list decodable, we store (for each v) such labels of v's neighbors from the same subgraph contiguously in memory, together with the common associated prefix and the neighbor count.

What Does It Fix? BRB does not derive the costly full separator tree but instead sets the number of bisectioning levels upfront *to control the preprocessing overhead.*

5.4 Degree-Minimizing (DM)

Finally, we describe a scheme that reduce the size of *general* graphs while ensuring fast decompression. The idea is to relabel vertices so that those with the highest degrees (and thus occurring more often in $\mathcal{A}$) receive the smallest labels. Then, in one scheme variant (DMf, proposed in the past [3]), full labels are encoded using Varint ("f" stands for full). In another variant (DMd, offered in this work), labels are encoded as differences ("d" indicates differences). Thus, $|\mathcal{A}|$ is decreased as frequent edges are stored using fewer bits.

Permuter/Transformer DM's $\mathcal{T}$ is identical to that of RB. DM's $\mathcal{P}$ differs: relabeling is now purely guided by vertex degrees (higher d_v enforces lower v's label).

What Does It Fix? DM trades some space for faster accesses to $\mathcal{A}$ compared to BRB (BRB's hierarchical encoding entails expensive queries). Next, it does not require costly recursive bisectioning. Finally, we later (§ 7.4) show that DMd significantly outperforms DMf and matches the compression ratios of WebGraph [18].

6 HIGH-PERFORMANCE LIBRARY

Past sections (§ 3–§ 5) illustrate a plethora of logarithmization schemes and enhancements for various graph families and scenarios. This large number poses design challenges. We now present the Log(Graph) C++ library that ensures: (1) a straightforward development, analysis, and comparison of graph representations composed of any of the proposed schemes, and (2) high-performance. The implementation of the Log(Graph) library is available online[3].

Modular Design and Extensibility Any graph compressed with Log(Graph) can be represented as a tuple $(G, \mathcal{O}, \mathcal{A}, \mathcal{L}[\mathcal{O}], \mathcal{L}[\mathcal{A}])$. These tuple elements corresponds to *modules* in the Log(Graph) library. These modules gather variants of $\mathcal{O}$, $\mathcal{A}$, and two logarithmization schemes, $\mathcal{L}[\mathcal{O}]$ and $\mathcal{L}[\mathcal{A}]$, that act upon $\mathcal{O}$ and $\mathcal{A}$. The $\mathcal{L}[\mathcal{A}]$ module contains submodules for $\mathcal{P}$ and $\mathcal{T}$. This enables us to seamlessly implement, analyze, and compare the described Log(Graph) variants.

High Performance Combinations of the variants of $\mathcal{O}$, $\mathcal{L}[\mathcal{O}]$, $\mathcal{A}$, and $\mathcal{L}[\mathcal{A}] = (\mathcal{P}, \mathcal{T})$ give many possible designs. For example, $\mathcal{O}$ can be any succinct bit vector. Now, selecting a specific variant takes place in a performance-critical region such as querying $\widehat{d}$. We identify four C++ mechanisms for such selections: #if pragmas, virtual functions, runtime branches, and templates. The first one results in unmanageably complex code. The next two entail performance overheads. We thus use templates to reduce code complexity while retaining high performance. Listing 3 illustrates: a generic `LogGraph` template class for defining different Log(Graph) representations, the constructor of a Log(Graph) representation, and a function N_v for accessing neighbors of a given vertex v. `LogGraph` only requires defining the following *types*: the offset structure ($\mathcal{O}$) type, the offset compression structure ($\mathcal{L}[\mathcal{O}]$) type, and the transformer ($\mathcal{T}$) type.

[3]https://spcl.inf.ethz.ch/Research/Performance/LogGraph

```
1 template<typename 𝒪, typename ℒ[𝒪], typename 𝒯>
2 class LogGraph : public BaseLogGraph { //Class template.
3  𝒪* offsets; //Opaque structure for storing offsets
4  ℒ[𝒪]* compressor; //Logarithmization scheme for offsets
5  𝒯* transformer; //Scheme acting upon permuted vertex IDs
6 };
7
8 template<typename 𝒪, typename ℒ[𝒪], typename 𝒯> // Constructor.
9 LogGraph<𝒪, ℒ[𝒪], 𝒯>::LogGraph(Permutation 𝒫, AA* aa) {
10  //aa is an instance of an adjacency array.
11  aa->permute(𝒫); //Relabel vertices. Note that 𝒫 isn't a type
12  transformer = new 𝒯(); //Create a new transformer object.
13  transformer->transform(&aa); //Modify vertices after relabeling
14  offsets = new 𝒪(aa); //Create a new object for offsets.
15  compressor = new ℒ[𝒪]();
16  compressor->compress(&offsets); //Logarithmize offset structure.
17 }
18
19 template<typename 𝒪, typename ℒ[𝒪], typename 𝒯>
20 //v_id is an opaque type for a vertex ID.
21 v_id* LogGraph<𝒪, ℒ[𝒪], 𝒯>::getNeighbors(v_id v) {//Resolve N_v.
22  v_id offset = offsets->getOffset(v);
23  v_id* neighbors = tr->decodeNeighbors(v, offset);
24  return neighbors;
25 }
```

Listing 3: (§ 6) A graph representation from the Log(Graph) library.

Note that the permuter $\mathcal{P}$ is an *object*, not *type*. As relabeling is executed only during preprocessing, it does not impact time-critical functions. Thus, selecting a given permutation can be done with simple branches based on the value of $\mathcal{P}$.

7 EVALUATION

We now illustrate that Log(Graph) offers the *sweetspot between low-overhead decompression and high compression ratios*, *enabling high-performance graph processing on top of compressed datasets*.

7.1 Evaluation Methodology

We first describe evaluation methodology.

Goals We illustrate that the Log(Graph) schemes offer (1) storage reductions, in many cases comparable to those of the state-of-the-art graph compression schemes, and (2) low-overhead decompression that enables high-performance graph processing running over compressed graph datasets.

Considered Algorithms We consider the following algorithms included in the GAP Benchmark Suite [11]: Breadth-First Search (BFS), PageRank (PR), Single Source Shortest Paths (SSSP), Betweenness Centrality (BC), Connected Components (CC), and Triangle Counting (TC). BFS, SSSP, and BC represent various types of traversals. PR is an iterative scheme where all the vertices are accessed in each iteration. CC represents protocols based on pointer-chasing. Finally, TC stands for non-iterative compute-intensive tasks.

- **BFS**: A state-of-the-art variant with direction-optimization and other enhancements that reduce data transfer [10, 11].
- **SSSP**: An optimized Δ-Stepping algorithm [11, 66, 71].
- **CC**: A variant of the Shiloach-Vishkin scheme [8, 81].
- **BC**: An enhanced Brandes' scheme [11, 20, 67].
- **PR**: A variant without atomic operations [11].
- **TC**: An optimized algorithm that reduces the computational complexity by preprocessing the input graph [28].

Considered Graphs We analyze synthetic power-law (the Kronecker model [62]), synthetic uniform (the Erdős-Rényi model [39]), and real-world datasets (including SNAP [63], KONECT [60], DIMACS [36], and WebGraph [19]); see Table 5 for details. Now, for Kronecker graphs, we denote them with symbols `sX_eY` where `s` is the scale (i.e., $\log_2 n$) and `e` is the average number of edges per vertex. Due to a large amount of data we present and discuss in detail a comprehensive subset; the remainder is in the technical report.

Type	ID	Name	n	m	$\bar{d}$
Web graphs	**uku**	**Union of .uk domain**	133M	4.66B	34.9
	uk	**.uk domain**	110M	3.45B	31.3
	sk	**.sk domain**	50.6M	1.81B	35.75
	gho	**Hosts of the gsh webgraph**	68.6M	1.5B	21.9
	wb	WebBase	118M	855M	7.24
	tpd	Top private domain	30.8M	490M	15.9
	wik	Wikipedia links	12.1M	288M	23.72
	tra	Trackers	27.6M	140M	5.08
	Others: tra, ber, gog, sta				
Affiliation graphs	orm	Orkut Memberships	8.73M	327M	37.46
	ljm	LiveJournal Memberships	7.48M	112M	15
Social networks	**fr**	**Friendster**	65.6M	1.8B	27.53
	tw	Twitter	49.2M	1.5B	30.5
	ork	Orkut	3.07M	117M	38.14
	Others: ljn, pok, flc, gow, sl1, sl2, epi, you, dbl, amz				
Road networks	**usrn**	**USA road network**	23.9M	28.8M	1.2
	Others: rca, rtx, rpa				
Various	Purchase networks (am1–am4), communication graphs (ema, wik)				

Table 5: The used real-world graphs (sorted by m). The details are provided for $n > 10$M or $m > 100$M. The largest ones are bolded.

Experimental Setup and Architectures We use the following systems to cover various types of machines:

- **CSCS Piz Daint** is 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. The interconnection is based on Cray's Aries and it implements the Dragonfly topology [40, 57]. The batch system is slurm 14.03.7. This machine represents massively parallel HPC machines.
- **Monte Leone** is an HP DL 360 Gen 9 system. One node has: two Intel E5-2667 v3 @ 3.2GHz Haswells (8 cores/socket), 2 hardware threads/core, 64 KB of L1 and 256 KB of L2 (per core), and 20 MB of L3 and 700 GB of RAM (per node). It represents machines with substantial amounts of memory.

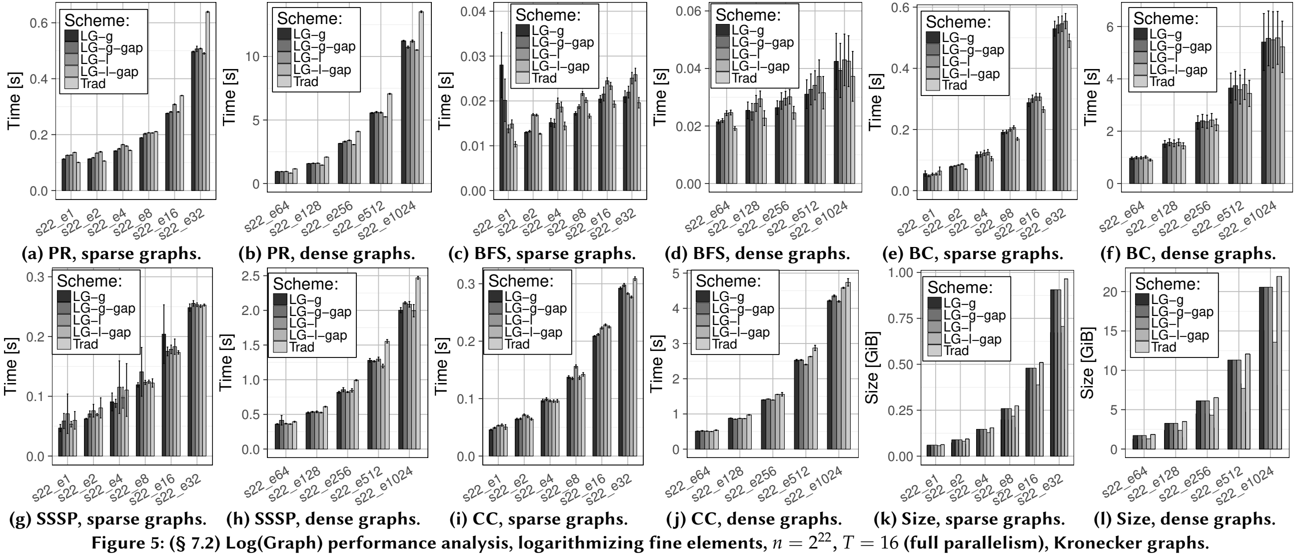
Evaluation Methodology We use the arithmetic mean for data summaries. We treat the first 1% of any performance data as warmup and we exclude it from the results. We gather enough data to compute the median and the nonparametric 95% confidence intervals.

Implementation Details

7.2 Logarithmizing Fine Elements

We first illustrate that logarithmizing fine graph elements, especially vertex IDs, reduces the size of graphs compared to the traditional adjacency arrays and incurs negligible performance overheads (in the worst case) or offers speedups (in the best case). The former is due to overheads from bitwise manipulations over the input data. Simultaneously, smaller pressure on the memory subsystem due to less data transferred to and from the CPU results in performance improvements. *This class of schemes should be used in order to maintain highest performance of graph algorithms while enabling moderate reductions in storage space for the processed graphs.*

Log(Graph) Variants and Comparison Targets We consider four variants of Log(Graph): `LG-g` (the global approach), `LG-g-gap` (the global approach with fixed-size gap encoding),



(a) PR, sparse graphs. **(b) PR, dense graphs.** **(c) BFS, sparse graphs.** **(d) BFS, dense graphs.** **(e) BC, sparse graphs.** **(f) BC, dense graphs.**

(g) SSSP, sparse graphs. **(h) SSSP, dense graphs.** **(i) CC, sparse graphs.** **(j) CC, dense graphs.** **(k) Size, sparse graphs.** **(l) Size, dense graphs.**

Figure 5: (§ 7.2) Log(Graph) performance analysis, logarithmizing fine elements, $n = 2^{22}$, $T = 16$ (full parallelism), Kronecker graphs.

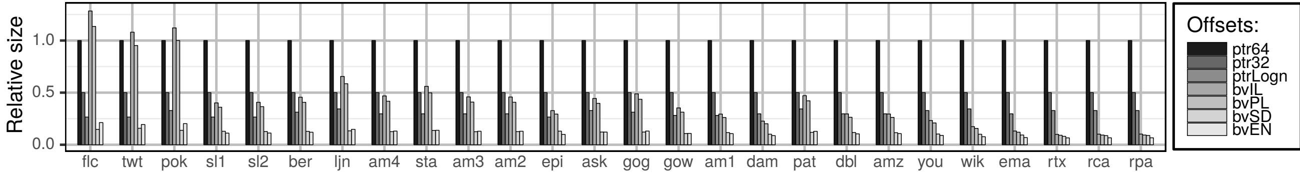


Figure 6: (§ 7.3) Illustration of the size differences of various $\mathcal{O}$ (both offset arrays and bit vectors). The offset sizes are $W \in \{32, 64, \lceil \log n \rceil\}$.

LG-l (the local approach), and LG-l-gap (the local approach with fixed-size gap encoding). We also incorporate the ILP heuristic for relabeling from § 3.6 that enhances the local approach. We compare Log(Graph) to the tuned GAPBS code that uses a traditional adjacency array (Trad).

Performance and Size on Single Nodes The results can be found in Figure 5. The collected data confirms our predictions. In many cases Log(Graph) offers performance comparable or better than that of the default adjacency array, for example for PR and SSSP. Simultaneously, it reduces $|\mathcal{A}|$ compared to Trad; the highest advantages are due to LG-l-gap (≈35% over LG-l); LG-g-gap does not improve much upon LG-g.

Scalability Log(Graph) advantages directly extend to distributed memories. Here, we measure the amount of communicated data and compare it to Trad. For example, in a distributed BFS and for 1024 compute nodes, this amount is consistently reduced by ≈37% across the considered graphs. We also conducted scalability analyses; the performance pattern is not surprising and Log(Graph) consistently finishes faster as T increases.

ILP Heuristic We also investigate the impact from the ILP heuristic from § 3.6. It reduces the size of graphs and we obtain consistent improvements or 1–4%, for example from 0.614 GB to 0.604 GB for the ork graph. The ILP heuristic could be used on top of gap-encoding when the user requires the highest compression ratio and still prefers to logarithmize fine elements (instead of logarithmizing $\mathcal{A}$) to ensure highest performance of processing graphs.

7.3 Logarithmizing Offset Structure $\mathcal{O}$

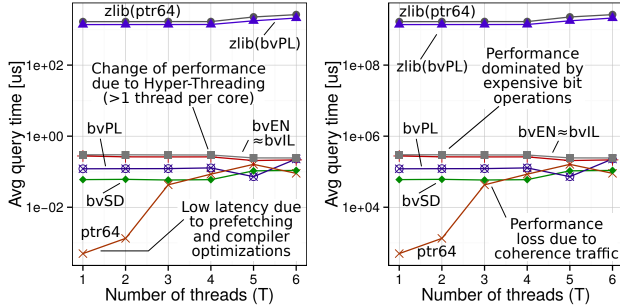
We show that logarithmizing $\mathcal{O}$ with succinct bit vectors (1) brings large storage reductions over simple bit vectors and offset arrays and (2) enables low-overhead decompression, matching performance of adjacency arrays in parallel settings.

Log(Graph) Variants and Comparison Targets We investigate all the described $\mathcal{O}$ variants of offset arrays and bit vectors presented in Table 4. We also incorporate a variant, denoted as ptrLogn, where we logarithmize each offset treated as a fine element, as described in § 3.4. We also consider compression with zlib [37].

Size We first compare the size of offset arrays and bit vectors in Figure 6. As expected, offset arrays are the largest except for graphs with high average degree $\bar{d}$. The sparser a graph, the bigger the advantage of bit vectors (especially bvEN and bvSN). This is because long sequences of unset bits are easy to compress. We also compare succinct bit vectors to bit vectors and offset arrays compressed with zlib [37] (data excluded for clarity); their size is similar (less than 1–3% of relative difference). In summary, succinct bit vectors reduce storage for most real-world graphs as such graphs *are* sparse.

Performance To show low-overhead decompression of logarithmized $\mathcal{O}$, we run a benchmark where T threads in parallel fetch offsets of 1000 random vertices. The results for tw and rca (representing graphs with high and low $\bar{d}$) are in Figure 7. First, bvEN is the slowest due to its complex design. Surprisingly, bvEN is followed by bvIL that has the biggest $|\mathcal{O}|$ (cf. Figure 6); its time/space tradeoff is thus not appealing for graph processing. Finally, bvPL and bvSD offer highest performance, with bvSD being the fastest for $T \leq 4$ (the difference becomes diluted for $T > 4$ due to more frequent cache line evictions). *The results confirm the theory*: bvPL offers $O(1)$ time accesses (while paying a high price in storage, cf. Figure 6) while bvSD uses little storage and fits well in cache. Moreover, the smaller T, the lower the latency of ptr64. This is because

fewer threads cause less traffic caused by the coherence protocol. We also consider designs compressed with blocked zlib scheme; they entail expensive decompression.



(a) Twitter graph tw. **(b) California road graph rca.**

Figure 7: (§ 7.3) Performance analysis of various types of $\mathcal{O}$.

7.4 Logarithmizing Adjacency Structure $\mathcal{A}$

Finally, we evaluate the logarithmization of $\mathcal{A}$ and show that it offers storage reductions that are in many cases comparable to that of modern graph compression schemes while providing significant speedups due to low-overhead decompression. *This class of schemes should be used in order to maintain highest reductions in storage space for graphs while enabling large speedups over the existing graph compression schemes.*

Log(Graph) Variants and Comparison Targets We evaluate all the discussed schemes: RB (§ 5.3.1), BRB (§ 5.3.2), DMd as well as DMf (§ 5.4), the traditional adjacency array (Trad), and the state-of-the-art WebGraph (WG) [18] compression system. We use the WebGraph original tuned Java implementation for gathering the data on compression ratios but, as Log(Graph) is developed in C++, we use a proof-of-a-concept C++ implementation of WG schemes for a fair C++ based performance analysis.

BRB: Alleviating RB's Preprocessing BRB alleviates RB's preprocessing overheads. Table 6 shows the overheads from RB compared to a simple adjacency array. BRB's preprocessing takes equally long if we build the full separator tree. Using fewer tree levels increases $|\mathcal{A}|$ but also reduces the preprocessing time, see Figure 8. Storage required to encode the recursive graph structure starts to dominate at some level.

Graph	uku	gho	orm	tw	usrn	ema	am1
Generation of RB	981.5	458.9	101.6	572.3	47.7	0.33	0.41
Generation of AA	19.5	5.9	1.1	5.8	0.3	0.02	0.02

Table 6: (§ 7.4) Illustration of preprocessing overheads [seconds].

DMd: Approaching the Time/Space Sweetspot Next, we illustrate that DMd significantly reduces $|\mathcal{A}|$, uses less storage than DMf, and can be generated fast. The size analysis is shown in Figure 9. We use relative sizes for clarity of presentation; the largest graphs use over 60 GB in size (in Trad). DMf and DMd generate much smaller $\mathcal{A}$ than Trad, with DMd outperforming DMf, being in various cases better than both RB and BRB (e.g., for ljm) but falling short in graph families that, e.g., are not well modeled by the power law. Now, DMd often closely matches WebGraph, for example for tw, fr, ljm. For others, it gives slightly larger $\mathcal{A}$ (e.g., for wik). We conclude that for most graphs DMd offers the storage/performance sweetspot: it ensures high level of compression, trades a little storage for high performance, and finally takes less (>10× for RB) time to generate than any other $\mathcal{A}$ scheme.

Preprocessing Log(Graph) preprocessing time is negligible, except for BRB. WebGraph is consistently slower.

Design and Performance of Algorithms We now evaluate BFS, PR, and TC. We use succinct bit vectors (bvSD) as $\mathcal{O}$ and various schemes for $\mathcal{A}$. Our modular design based on the established model enables quick and easy implementation of each graph algorithm and each combination of $\mathcal{A}$ and $\mathcal{O}$; each variant requires at most 30 lines of code (in the class definition and the constructor). The results are shown in Figure 10. The BFS and PR analyses for large graphs (gho, orm, tw, usrn) illustrate that DMd is comparable to RB and DMf, merely up to 2× slower than the uncompressed Trad, and significantly faster (e.g., ≈3× for orm) than WebGraph. The relative differences for TC are smaller because the high computational complexity of TC makes decompression overheads less significant. We conclude that DMd offers performance comparable to the state-of-the-art RB as well as DMf, while avoiding costly overheads from recursive bisectioning.

7.5 Discussion of Results

Our evaluation confirms the characteristics of three logarithmization families of schemes.

First, **logarithmizing fine elements** does deliver storage reductions (20-35%) compared to the traditional adjacency array and it enables very high performance close to or even exceeding that of tuned graph processing codes. It enables its merits on both shared- and distributed-memory machines.

Next, **logarithmizing adjacency data** is somewhat an opposite to the logarithmization of fine elements: it aggressively reduces storage, in some cases by up to ≈80% compared to the adjacency array, approaching the compression ratios of modern graph compression schemes and simultaneously offering speedups of around 3× over these schemes. Specifically, the BRB scheme alleviates RB's preprocessing overheads while the DMd scheme offers the best space/performance tradeoff. Yet, this family of schemes leads to higher overheads in performance than the logarithmization of fine elements. Thus, it should be used when reducing storage outweighs achieving highest performance.

Finally, **logarithmizing offset structures** can enhance *any* parallel graph processing computation because it does not incur performance overheads in parallel settings (for $T \geq 4$ in our tests) while it does reduce storage required for offsets (a part of the adjacency array) even by >90%. We conclude that succinct bit vectors are a good match for $\mathcal{O}$. First, they reduce $|\mathcal{O}|$ more than any offset array and are comparable to traditional compression methods such as zlib. Next, they closely match the performance of offset arrays for higher thread counts and are orders of magnitude faster than zlib. Finally, they consistently retain their advantages when varying the multitude of parameters, as illustrated in our extended technical report version of the paper.

8 RELATED WORK

We now discuss how Log(Graph) differs from or complements various aspects of graph processing and compression. As we illustrated, *Log(Graph) is a tool that can enhance any graph processing engine, benchmark, or algorithm that stores graphs as adjacency arrays*, such as GAPBS [11], Pregel [68], HAMA [80],

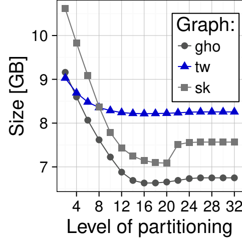


Figure 8: (§ 7.4) BRB analysis.

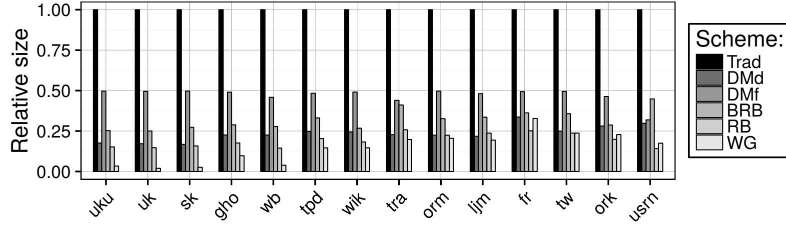


Figure 9: (§ 7.4) Illustration of the storage overhead of different types of $\mathcal{A}$.

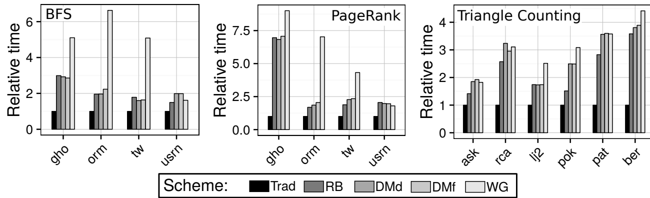


Figure 10: (§ 7.4) The analysis of the performance impact of Log(Graph) on parallel graph algorithms and comparison to the C++ implementation of WebGraph schemes. The geometric mean of ratios of DMd and WG is ≈0.5 (BFS), ≈0.6 (PageRank), and ≈0.9 (TC).

GraphLab [64], Spark [92], Galois [58], PBGL [48], GAPS [11], Ligra [83], Gemini [93], Tux^2 [91], Green-Marl [51], and others [12, 15, 42]. It could also be used to enhance systems and schemes where graphs are modeled with their adjacency matrix [14, 25, 70, 85]. For example, one could use logarithmized vertex IDs to accelerate graph processing and reduce the pressure on the memory subsystem and network.

Log(Graph) and Compact Schemes A graph representation based on recursive partitioning, proposed by Blandford et al. [16], was proved to be *compact*: it takes $O(n)$ bits for an input graph with n vertices. It reduces $|\mathcal{A}|$ for several real-world graphs. Yet, its preprocessing is costly. *Log(Graph) alleviates it with the BRB scheme.*

Log(Graph) and Succinct Schemes *Log(Graph) uses and puts in practice succinct designs to enhance graph storage and processing.* There are various succinct graph representations [5, 9, 17, 31, 35, 45, 46, 52–55, 72, 76, 78, 89] but they are mostly theoretical structures with large hidden constants, negligible asymptotic enhancements over the respective storage lower bound, or no practical codes. Succinct [4] is a data store that uses succinct data structures; yet, it does not specifically target graphs or graph processing. Some works [82] construct succinct structures in parallel, but they do *not* process them in parallel. Finally, there are several libraries of succinct data structures [2, 33, 41, 43, 44, 49]. *Contrarily to our work, none of these designs enhances graph processing and they do not address parallel processing of a succinct data structure*.

Log(Graph) and Compression Schemes A mature compression system for graphs is WebGraph [18]. There are also other works [3, 6, 7, 13, 21, 22, 24, 27, 29, 30, 32, 38, 50, 56, 61, 69, 74, 75, 77, 79, 86–88]. Some mention encoding some vertex IDs with the logarithmic number of bits [3, 87]; *Log(Graph) extends them with schemes such as local logarithmization § 3.2.2*. Several works use ILP to relabel vertices to reduce $|\mathcal{A}|$ [27, 38]. Others collapse specified subgraphs into supervertices and merge edges between them into a superedge [24, 77, 86]. These systems come with *complex compression* and *costly decompression*. Next, Ligra+ [84] compresses graphs while ensuring high performance of graph algorithms. It is orthogonal to Log(Graph) as it uses parallel compute power to provide fast decoding while Log(Graph) relies on simplicity and *it can can be used to enhance Ligra+* (e.g., with local vertex ID or $\mathcal{O}$ logarithmization) and ensure *even more performance*. Moreover, G-Store [59] is a storage system for graphs that, among others, removes most signigicant bit (MSB) zeros of vertex IDs within one tile of 2D partitioning. In general, removing MSB zeros was proposed even before G-Store [3, 87]. We enhance this technique and apply it holistically to all the considered fine graph elements. The technique in G-Store is orthogonal to ours and can be combined with the hierarchical logarithmization to reduce space even further. There are also several works that reorder vertex IDs for more performance. For example, Wei et al. [90] reduce cache miss rate. As their most important goal is to accelerate graph processing without storage reductions, we exclude this work from a more detailed discussion as less related. We conclude that Log(Graph), on one hand, *enables simple and generic logarithmization of fine elements for inexpensive storage reductions and possible performance improvements*. Simultaneously, it comes with *more sophisticated schemes for graphs with more specific properties such as separability.*

9 CONCLUSION

Reducing graph storage overheads is important in large-scale computations. Yet, established schemes such as WebGraph [18] negatively impact performance. To address this, we propose Log(Graph): a graph representation that applies logarithmic storage lower bounds to (aka "logarithmizes") various graph elements.

First, logarithmizing fine elements offers simplicity and negligible performance overheads or even speedups from reducing data transfers. It can enhance virtually any graph processing engine in shared- and distributed-memory settings. For example, we accelerate SSSP in the GAP Benchmark [11] by ≈20% while reducing the required storage by 20-35%.

To logarithmize offset or adjacency data, we use *succinct data structures* [53, 76] and ILP. We investigate the associated tradeoffs and identify as well as tackle the related issues, enhancing the processing and storing of both specific and general graphs. For example, Log(Graph) outperforms WebGraph schemes while nearly matching its compression ratio with various schemes. We provide a carefully crafted and extensible, high-performance implementation.

Finally, to the best of our knowledge, our work is the first performance analysis of accessing succinct data structures in a parallel environment. It illustrates surprising differences between succinct bit vectors and offset arrays when varying the amount of parallelism. Our insights can be used by both theoreticians and practitioners to develop more efficient succinct schemes for parallel settings.

REFERENCES

[1] DYNAMIC: a succinct and compressed dynamic data structures library.
[2] Sux - Implementing Succinct Data Structures. available at: http://sux.dsi.unimi.it.
[3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. *DCC*, 2001.
[4] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. *NSDI*, 2015.
[5] S. Alvarez-Garcia, G. de Bernardo, N. R. Brisaboa, and G. Navarro. A succinct data structure for self-indexing ternary relations. *Journal of Discrete Algorithms*, 43:38–53, 2017.
[6] A. R. Asadi, E. Abbe, and S. Verdú. Compressing data on graphs with clusters. In *Information Theory (ISIT), 2017 IEEE International Symposium on*, pages 1583–1587. IEEE, 2017.
[7] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of web graphs. *Computing and Combinatorics*, pages 1–11, 2008.
[8] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 547–556. IEEE, 2005.
[9] J. Barbay, L. Castelli Aleardi, M. He, and J. Munro. Succinct Representation of Labeled Graphs. *Algorithms and Computation*, 2007.
[10] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. *SC*, 2012.
[11] S. Beamer, K. Asanovic, and D. Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.
[12] M. Besta and T. Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 161–172. ACM, 2015.
[13] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.
[14] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *Proc. IEEE IPDPS*, volume 17, 2017.
[15] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. 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*, pages 93–104. ACM, 2017.
[16] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact Representations of Separable Graphs. *SODA*, 2003.
[17] G. Blelloch and A. Farzan. Succinct representations of separable graphs. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 138–150. Springer Berlin Heidelberg, 2010.
[18] P. Boldi and S. Vigna. The WebGraph Framework I: compression echniques. *WWW*, 2004.
[19] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *World Wide Web Conf. (WWW)*, pages 595–601, 2004.
[20] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.
[21] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *SPIRE*, volume 9, pages 18–30. Springer, 2009.
[22] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.
[23] A. Brodnik and J. I. Munro. Membership in Constant Time and Almost-Minimum Space. *SIAM J. Comput.*, 28(5):1627–1640, May 1999.
[24] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. *WSDM*, 2008.
[25] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.
[26] S. V. Chekanov, E. May, K. Strand, and P. Van Gemmeren. ProMC: Input–output data format for HEP applications using varint encoding. *Computer Physics Communications*, 185(10):2629–2635, 2014.
[27] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009.
[28] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.
[29] F. Claude and S. Ladra. Practical representations for web and social graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1185–1190. ACM, 2011.
[30] F. Claude and G. Navarro. A fast and compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 118–129. Springer, 2007.
[31] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *In Proc. 15th SPIRE, LNCS 5280*, pages 176–187, 2008.
[32] F. Claude and G. Navarro. Extended compact web graph representations. *Algorithms and Applications*, 6060:77–91, 2010.
[33] Daisuke Okanohara. rsdic - Compressed Rank Select Dictionary. available at: http://code.google.com/p/rsdic.
[34] J. Dean. Challenges in building large-scale information retrieval systems. In *Keynote of the 2nd ACM International Conference on Web Search and Data Mining (WSDM)*, 2009.
[35] E. Demaine. Advanced Data Structures, 2012. Lecture Notes.
[36] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Math. Soc., 2009.
[37] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification, 1996.
[38] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *CSUR*, 2002.
[39] P. Erdős and A. Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi*, 2:482–525, 1976.
[40] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, R. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: a scalable HPC system based on a Dragonfly network. In *SC*, page 103. IEEE/ACM, 2012.
[41] Francisco Claude. libcds. https://github.com/fclaude/libcds.
[42] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2018.
[43] Giuseppe Ottaviano. Succinct library. https://github.com/ot/succinct.
[44] S. Gog, T. Beller, A. Moffat, and M. Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. *SEA*, 2014.
[45] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 2014.
[46] R. Gonzalez, S. Grabowski, V. Makinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.
[47] T. Granlund. Instruction latencies and throughput for AMD and Intel x86 Processors. *Technical report, KTH*, 2012.
[48] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.
[49] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Experimental Algorithms*, pages 5–17. Springer, 2013.
[50] C. Hernandez and G. Navarro. Compressed representation of web and social networks via dense subgraphs. In *International Symposium on String Processing and Information Retrieval*, pages 264–276. Springer, 2012.
[51] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.
[52] G. Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.
[53] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, 1988.
[54] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 575–584, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.
[55] S. Kannan, M. Naor, and S. Rudich. Implicit Representation of Graphs. In *SIAM Journal On Discrete Mathematics*, pages 334–343, 1992.
[56] A. Khan, S. S. Bhowmick, and F. Bonchi. Summarizing static and dynamic big graphs. *Proceedings of the VLDB Endowment*, 10(12):1981–1984, 2017.
[57] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society.
[58] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.
[59] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 830–841. IEEE, 2016.
[60] J. Kunegis. Konect: the koblenz network collection. In *Proc. of Intl. Conf. on World Wide Web (WWW)*, pages 1343–1350. ACM, 2013.
[61] S. Ladra. Algorithms and compressed data structures for information retrieval. 2011.
[62] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.
[63] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.
[64] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *preprint arXiv:1006.4990*, 2010.
[65] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 2007.
[66] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.
[67] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating

betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.
[68] G. Malewicz et al. Pregel: a system for large-scale graph processing. *SIGMOD*, 2010.
[69] S. Maneth and F. Peternek. Grammar-based graph compression. *arXiv preprint arXiv:1704.05254*, 2017.
[70] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. *arXiv preprint arXiv:1408.0393*, 2014.
[71] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
[72] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, pages 762–776, 2002.
[73] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.
[74] G. Navarro. Compressing web graphs like texts. Technical report, Technical Report TR/DCC-2007-2, Dept. of Computer Science, University of Chile, 2007.
[75] S. Navlakha et al. Graph summarization with bounded error. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–432. ACM, 2008.
[76] D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. *ALENEX*, 2007.
[77] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 405–416. IEEE, 2003.
[78] R. Raman, V. Raman, and S. R. Satti. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *CoRR*, abs/0705.0552, 2007.
[79] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener. The link database: Fast access to graphs of the web. In *Data Compression Conference, 2002. Proceedings. DCC 2002*, pages 122–131. IEEE, 2002.
[80] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Intl. Conf. on Cloud Comp. Tech. and Science*, CLOUDCOM'10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.
[81] Y. Shiloach and U. Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.
[82] J. Shun. Parallel wavelet tree construction. In *Data Compression Conference (DCC), 2015*, pages 63–72. IEEE, 2015.
[83] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.
[84] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. *DCC*, 2015.
[85] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2017.
[86] N. Stanley, R. Kwitt, M. Niethammer, and P. J. Mucha. Compressing networks with super nodes. *arXiv preprint arXiv:1706.04110*, 2017.
[87] T. Suel and J. Yuan. Compressing the graph structure of the web. *DCC*, 2001.
[88] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.
[89] S. Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms*, pages 154–168. Springer, 2008.
[90] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.
[91] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou. Tux2: Distributed graph computation for machine learning. In *NSDI*, pages 669–682, 2017.
[92] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the USENIX Conf. on Net. Sys. Design and Impl.*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
[93] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.