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

This is a full version of a paper published at PACT ’18 under the same title

Code:
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 [70]. 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 [22], use techniques such as reference encoding or interval encoding that may require costly decompression. For example, consider two vertices, v1 and v2, and assume that some of the neighbors of v1 and v2 are identical. In reference encoding, these shared neighbors are stored only once, in the adjacency array of either v1 or v2. 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 $\log_2|S|$ 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 $>2\times$) 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, 94], in which one uses parametric 95% confidence intervals.

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

We first describe the used concepts and notation; see Table 1 for summarized symbols. In Log(Graph), we use multiple techniques and we postpone describing some of them to their related sections for better readability.

2.1 Used Models

We first describe models used in this work.

2.1.1 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 $\{1, \ldots, n\}$. $N_G$ and $d_G$ denote the neighbors and the degree of a vertex $v$. $N_G(v)$ is $v$’s in-neighbor. $\bar{N}_G$ is the maximal value in a given set or sequence $X$, for example $\bar{N}_G$ is the maximum ID among $v$’s neighbors. $\bar{W}$ is the maximal edge weight in a given graph $G$.

2.1.2 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 $W$.

2.2 Used Concepts

We next explain concepts related to the structures used in Log(Graph) and their size; see also Figure 2 for an overview.

First, we discuss succinctness. Assume $OPT$ is the optimal number of bits to store some data. A representation of this data is compact if it uses $O(OPT)$ bits and succinct if it uses $OPT + o(OPT)$ bits. They all should support a reasonable set of queries in (ideally) $O(1)$ time [20]. Now, they differ from compression mechanisms such as zlib as they do not entail expensive decompression. To avoid confusion, we use the term condensing to refer in general to reducing the size of some data.

Finally, we describe graph separability. Intuitively, $G$ is vertex (or edge) separable (i.e., has good separators) if we can divide $V$ 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 much smaller than $|V|$.

2.3 Used Data Structures

2.3.1 Adjacency Array Data Structures. Log(Graph) builds upon the traditional adjacency array (AA) representation. One part of AA is an array (denoted as $A$) with adjacency data. $A$ consists of $n$ subarrays $A_i$, $i \in V$. Subarray $A_i$ contains neighbors
We divide the Log(Graph) schemes into three categories, (§ 4.1, § 4.3). A bit vector

2.3.2 Bit Vectors. Next, we use simple bit vectors to enhance O

3 LOGARITHMIZING FINE ELEMENTS

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 \(|\log n|\) 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 \(|\log 2m|\); 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 \(\{0, ..., \bar{w}\}\) is \(|\log \bar{w}|\) (for continuous weights, we first scale them appropriately to become elements of a discrete set).

3.2 Logarithmization of Vertex IDs

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

2.4 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 O (§ 4), and adjacency data 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 to the traditional AA while delivering performance close to or matching or even exceeding that of tuned graph processing codes. Second, compressing offset structures 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 O even by >90%. Finally, adjacency data 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×.

| Graph model | A graph \(G = (V, E)\); \(V\) and \(E\) are sets of vertices and edges. |
|-------------|------------------------------------------------------------------|
| \(n, m\)    | Numbers of vertices and edges in \(G\); \(|V| = n, |E| = m\).        |
| \(w_{\ell, \psi}\) | The weight of an edge \((v, \psi)\).                            |
| \(d_{\ell, v}, N_v, N_{\ell, v}\) | Degree, neighbors, and \(l\)th neighbor of a vertex \(v\); \(N_{\ell, v} \equiv v\). |
| \(\bar{x}, \bar{X}\) | 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 | The adjacency array of a given graph and a given vertex. |
|-----------------|---------------------------------------------------------|
| \(A, A_v\)     | The adjacency structure of \(A\) and \(A_v\).            |
| \(O, O_v\)     | The offset structure of a given graph and an offset to \(A_v\). |
| \(|A|, |O|\)     | The sizes of \(A\) and \(O\).                          |
| \(|\mathcal{L}(A)|, |\mathcal{L}(O)|\) | Logarithmization schemes acting upon \(A\) and \(O\). |
| \(B, E, W\)    | Various parameters of \(A\) and \(O\); see § 4.2–§ 4.3 for details. |

| Machine model | The number of levels in a hierarchical machine. |
|---------------|----------------------------------------------|
| \(N\)        | The number of elements at level \(i\), the number of compute nodes. |
| \(T, \ell\)  | The number of threads/processes.             |
| \(W\)        | The memory word size [bits].                 |
| \(t_x\)      | Time to do a given operation \(x\).          |

| Others         | Permuter: function that relabels vertices. |
|----------------|-------------------------------------------|
| \(P\)          | Transformers: functions that arbitrarily modify \(A\). |
| \(G_x\)        | Subgraphs of \(G\) constructed in recursive bisectioning. |

Table 1: Symbols used in the paper gathered for the reader’s convenience.
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 (\( \tilde{N}_v \ll n \)). Here, the optimum number of bits for a vertex ID in \( A_v \) is \( \left\lceil \log \tilde{N}_v \right\rceil \), 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 \( \lceil \log \log \tilde{N}_v \rceil \) bits. The size of \( A \) is in this case

\[
|A| = \sum_{v \in V} \left( d_v \left[ \log \tilde{N}_v \right] + \left\lceil \log \log \tilde{N}_v \right\rceil \right)
\]

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 \( \frac{n}{N} \). The intra ID part takes \( \lceil \log \frac{n}{N} \rceil \) bits, the inter one takes \( \lceil \log H \rceil \). As the inter part is unique for a given node, it is stored once per node. Thus

\[
|A| = n \left[ \log \frac{n}{N} \right] + H \left[ \log H \right]
\]

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 \( \frac{n}{N^j} \). Thus, the intra ID part requires \( \lceil \log \frac{n}{N^j} \rceil \) bits. The inter part needs \( \sum_{j \in \{2..N-1\}} \left[ \log H_j \right] \) bits and has to be stored once per each machine element, thus

\[
|A| = n \left[ \log \frac{n}{N^N} \right] + \sum_{j=2}^{N-1} H_j \left[ \log H_j \right]
\]

3.3 Logarithmization of Edge Weights

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

\[
|A| = 2m \left( \left\lceil \log n \right\rceil + \left\lceil \log W \right\rceil \right)
\]

Finally, one can also “logarithmize” other AA elements, including offsets. Each offset must be able to address any position in \( A \) that may reach \( 2m \). Thus, the related lower bound is \( \left\lceil \log 2m \right\rceil \), giving \(|O| = n \left\lceil \log 2m \right\rceil\).

3.5 Theoretical Storage Analysis

3.5.1 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

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

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

3.5.2 Power-Law Graphs. We next analyze power-law graphs; the derivation is in the Appendix. Here, the probability that a vertex has degree \( d \) is \( f(d) = ad^{-\beta} \), and

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

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 $|A|$. Consider any $A_v$. We observe that a single neighbor in $A_v$ with a large ID may vastly increase $|A_v|$; we use $\hat{N}_v$ bits to store each neighbor in $A_v$ 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. (5). It minimizes the weighted sum of $\hat{N}_v, v \in V$. Each maximal ID is given a positive weight which is the inverse of the neighborhood size; this intuitively decreases $\hat{N}_v$ in smaller $A_v$.

$$\min \sum_{v \in V} \hat{N}_v \cdot \frac{1}{|A_v|} \quad (1)$$

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

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

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 $|A_v|$, 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.

```c
/* Input: graph G, Output: a new relabeling N(v),\forall v \in V. */
2 void relabel(G) {
3 ID[0..n-1] = [0..n-1]; //An array with vertex IDs.
4 D[0..n-1] = [0..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];
7 ml = 1; //An auxiliary variable ‘’new label’’.
8 sort(D); sort(D);
9 for(int i = 1; i < n; ++i) //For each vertex...
10 for(int j = D[i]; j < D[i]; ++j) //For each neighbor...
11 if (visit[i] == false) {
12 visit[i] = true;
13 if (visit[j] == false) {
14 ml = ml + 1;
15 } // visit[j] == true;
16 if (visit[j] == false) {
17 if (ml < ml) ml = ml;
18 } // ml = ml;
19 }
20 }
```

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 $|A|$. Traditionally, in gap encoding one calculates differences between all consecutive neighbors in $A_v$. These differences are then encoded with a variable-length code such as Varint [20]. 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 $A_v$ determines the number of bits to encode any difference in the same $A_v$.

### 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.

#### 3.8.1 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 [52].

| 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      |
| ADD       | +          | Performs a bitwise AND operation. | 1      |
| ADD       | +          | Performs an addition between two values. | 2      |

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

#### 3.8.2 Accessing An Edge ($N_v(v)$)

We first describe how to access a given edge in $A$ that corresponds to a given neighbor of $v$ ($N_v(v)$); 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 $O[v]$ of $v$’s neighbors’ array (Line 3). This usually involves a cache miss, taking $t_{cm}$. Second, we derive $o = s \cdot O[v]$ (the exact bit position of $N_{cm}(v)$); it takes $t_{mem}$ (Line 3). Third, we find the closest byte alignment before $o$ by right-shifting $o$ by 3 bits, taking $t_{shift}$ (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 $N_v(v)$. Here, we use the x86 bextr instruction that combines these two operations and takes $t_{bext}$. 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.

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

#### 3.8.3 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.

#### 3.8.4 Accessing a Degree ($d_v$)

$d_v$ is simply calculated as the difference between two offsets: $O[v + 1] - O[v]$.
We now logarithmize $O$. A bit vector that serves as an array of a size $j$ position of an array of a can be stored 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 $O$ as a bitvector and then encode it as a succinct bit vector [48] that approaches the storage lower bound while providing fast accesses to its contents.

4.1 Arrays of Offsets vs. Bit Vectors for $O$

Usually, $O$ is an array of $n$ offsets and the size of $O$, $|O|$, is much smaller than $|A|$. Still, in sparse graphs with low maximal degree $d$, $|O| \approx |A|$ or even $|O| > |A|$. For example, for the USA road network, if $O$ contains 32-bit offsets, $|O| \approx 0.83|A|$. To reduce $|O|$ in such cases, one can use a bit vector instead of an array of offsets. For this, $A$ is divided into blocks (e.g., bytes or words) of a size $B$ [bits]. Then, if the $i$th bit of $O$ is set (i.e., $O[i]=1$) and if this is the $j$th set bit in $O$, then $A_j$ starts at the $i$th block. The key insight is that getting the position of $j$th set bit and thus the offset of the array $A_j$ is equivalent to performing a certain operation called $select(j)$ that returns the position of the $j$th one in $O$. 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 [49]. They both trade some space for a faster $select$. bvPL uses up to $0.2|O|$ 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 $O(\log |O|)$ time [49]. 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 $O$.

4.2 Understanding Storage Lower Bounds

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

4.3 Incorporating Succinct Bit Vectors

To reduce the size of $O$ and improve the performance of its query, we use succinct bit vectors.

4.3.1 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 [20]. Thus, succinct designs differ from compression mechanisms such as zlib as they do not entail expensive decompression.

4.3.2 Succinct Bit Vectors: Preliminaries. Succinct bit vectors use $|Q| + o(Q)$ bits assuming a storage lower bound of $|Q|$ and they answer the $select$ query in $o(Q)$ time. Many such designs exist [83] 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 [17]. 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 $|Q| + o(Q)$. 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.

4.3.3 Succinct Bit Vectors in Log(Graph). First, we use the entropy based bit vector (bvEN) [83]. The key idea behind bvEN is to use a dictionary data structure [27] that achieves the lower bound for storing bit vectors of length $2Wm/B$ with $n$ ones. Second, we use sparse succinct bit vectors (bvSD) [81]. 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 $O_v$. Now, $ptrW$ (array of offsets) together with bvPL and bvSD feature the fastest $O_v$, we select them as the most promising candidates for $O$ in Log(Graph).
| $\mathcal{O}$ | ID | Asymptotic size [bits] | Exact size [bits] | rank | select | Deriving $\mathcal{O}_v$ |
|----------|-----|------------------------|-------------------|-------|--------|----------------------|
| Pointer array | ptrW | $O(W(n))$ | $W(n + 1)$ | - | - | $O(1)$ |
| Plain [49] | bvPL | $O\left(\frac{Wm}{B}\right)$ | $\frac{2Wm}{B} + \left(\frac{1}{B} + 64\right)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Interleaved [49] | bvIL | $O\left(\frac{Wm}{B} + \frac{Wm}{B^2}\right)$ | $2Wm\left(\frac{1}{B} + \frac{64}{B^2}\right)$ | $O(1)$ | $O\left(\log\frac{Wm}{B}\right)$ | $O\left(\log\frac{Wm}{B}\right)$ |
| Entropy based [34, 83] | bvEN | $O\left(\frac{Wm}{B}\log\frac{Wm}{B}\right)$ | $\approx \log\left(\frac{2Wm}{n}\right)$ | $O(1)$ | $O\left(\log\frac{Wm}{B}\right)$ | $O\left(\log\frac{Wm}{B}\right)$ |
| Sparse [81] | bvSD | $O(n + n\log\frac{Wm}{B})$ | $\approx n\left(2 + \log\frac{2Wm}{B}\right)$ | $O\left(\log\frac{Wm}{B}\right)$ | $O(1)$ | $O(1)$ |
| B-tree based [1] | bvBT | $O\left(\frac{Wm}{B}\right)$ | $\approx 1.1\cdot\frac{2Wm}{B}$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Gap-compressed [1] | bvGC | $O\left(\frac{Wm}{B}\log\frac{Wm}{B}\right)$ | $\approx 1.3\cdot\frac{2Wm}{B}\log\frac{2Wm}{B}$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

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

4.5 High-Performance Implementation

For high performance, we use the sdsl-lite library [50] 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 LOGARITHMING 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 $(P, T)$. $P$ is the permuter: a function that relabels the vertices. We introduce $P$ to explicitly capture the notion that appropriate labeling of vertices significantly reduces $|\mathcal{A}|$. We have $P: V \rightarrow \mathbb{N}$ such that (the condition enforces the uniqueness of IDs):

$$\forall_{v, u \in V} (v \neq u) \Rightarrow [P(v) \neq P(u)] \quad (3)$$

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

$$T_x : V \times \ldots \times V \rightarrow \{0, 1\} \times \ldots \times \{0, 1\}$$

We introduce $T$ to enable arbitrary operations on sequences of relabeled vertices, for example be the Varint encoding [38].
5.2 Understanding Storage Lower Bounds

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 $\left\lceil \log_2 \left( \frac{n!}{m! (n-m)!} \right) \right\rceil$ (Table 2). Now, today’s graph codes already approach this bound. For example, the Graph500 benchmark [78] requires $\approx1,126$ TB for a graph with $2^{32}$ vertices and $2^{48}$ edges while the corresponding lower bound is merely $\approx350$ TB. We thus propose to assume more about $G$’s structure on top of the number of vertices and edges. We now target separable graphs (cf. Section 2).

5.3 Incorporating Compactness

We use compact graph representations that take $O(n)$ bits to encode graphs. The main technique that ensures compactness is recursive bisectioning. We first describe an existing recursive bisectioning 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. [20] (referred to as the RB scheme) that requires $O(n)$ bits to store a graph that is separable. Figure 4 contains an example. The basic method is to relabel vertices of a given graph $G$ to minimize differences between the labels of consecutive neighbors of each vertex $v$ in each adjacency list. Then, the differences are recorded with any variable-length gap encoding scheme such as Varint [38]. Assuming that the new labels of $v$’s neighbors do not differ significantly, the encoded gaps use less space than the IDs [20]. Now, to reassign labels in such a way that the storage is reduced, the graph (see 1 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. Recursive partitions form a binary separator tree with the leaves being single vertices 2. Then, the vertices are relabeled as imposed by an inorder traversal over the leaves of the separator tree 3. 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 4.

Using Permuters and Transformers

One can easily express RB using $P$ and $T$. First, $P$ relabels the vertices according to the order in which they appear as leaves in the inorder traversal of the separator tree obtained after recursive graph bipartitioning. Here, we partition graphs to make subgraphs [almost] equal $<0.1\%$ of difference) in size. Second, each transformer $T = \{ T_v(v, N_v) \}$ takes as input $\{v\}$ and $N_v$. It then encodes the differences between consecutive vertex labels using Varint. The respective differences are: $|N_{i,v} - N_{0,v}|$, $N_{2,v} - N_{1,v}$, ..., and $N_{i,v} - N_{i-1,v}$.

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

5.3.2 Binary Recursive Bisectioning (BRB).

The core idea is to relabel vertices so that vertices in clusters have large common prefixes (clusters are identified during partitioning). One prefix is stored only once per each cluster.

What Does It Fix? BRB alleviates two issues inherent to RB. First, there is no costly inorder traversal over the separator tree. More importantly, there is no expensive derivation of the full separator tree. Instead, one sets the number of partitioning levels upfront to control the preprocessing overhead.

Permuter (Relabeling Vertices)

We present an example in Figure 4. First, we recursively bipartition the input graph $G$ to identify common prefixes and uniquely relabel the vertices. After the first partitioning, 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 partitioned 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 $|A|$. 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.

5.4 Incorporating Integer Linear Programming

We next logarithmize $G$ with ILP to target generic graphs and not just the ones that are separable. We first illustrate a simple existing scheme that uses ILP for graph storage reductions (§ 5.4.1) and then accelerate it (§ 5.4.2).

5.4.1 Optimal Difference-Based (ODB).

There are several variants of ILP-based schemes [42] where the objective function minimizes: the sum of differences between consecutive neighbors in adjacency lists (minimum gap arrangement (MGapA)), the sum of logarithms of differences from MGapA (minimum logarithmic gap arrangement (MLogGapA)), the sum of logarithms of differences from MLogA (minimum logarithmic arrangement (MLogA)), and the sum of logarithms of differences from MLinA (minimum logarithmic arrangement (MLogA)).

Using Permuters and Transformers

Now, ODB’s $T$ is identical to that of RB as it encodes ID differences while $P$ determines the relabeling obtained by solving a respective ILP problem. Consider a vector $v = (P(v_1), ..., P(v_n))$ that models new vertex labels (where $v_1, ..., v_n \in V$). For example, the MGapA and MLogA objective functions are respectively

$$\min_{P(\cdot)} \sum_{v \in V} \sum_{i=0}^{N_v-1} |P(N_{i+1,v}) - P(N_{i,v})|$$

(4)

$$\min_{P(\cdot)} \sum_{v \in V} \sum_{u \in N_v} \log |P(v) - P(u)|$$

(5)

Both functions use the uniqueness Constraint (3).
5.4.2 Positive Optimal Differences (POD). We now enhance the ODB MGapA (§ 5.4.1) by removing the absolute value $| \cdot |$ from the objective function, which accelerates relabeling. Yet, this requires additional constraints to enforce that the neighbors of each vertex are sorted according to their IDs. We present the constraints below; readers who are not interested in the mathematical details may proceed to § 5.5.

\[
\forall v \in V \forall i,j \in \{1..d\} \left[ \mathcal{P} \left( N^i_{j,v} \right) - \left( x^i_{v,j} - 1 \right) \cdot n \leq N_{i,v} \right] \tag{6}
\]

\[
\forall v \in V \forall i,j \in \{1..d\} \left[ \mathcal{P} \left( N^i_{j,v} \right) + \left( 1 - x^i_{v,j} \right) \cdot n \geq N_{i,v} \right] \tag{7}
\]

\[
\forall v \in V \forall i,j \in \{1..d\} \left[ \sum_{j=1}^{n} x^i_{v,j} = 1 \right] \tag{8}
\]

\[
\forall v \in V \forall i,j \in \{1..d\} \left[ N_{i,v} < N_{i,v+1} \right] \tag{9}
\]

$N^i_{j,v}$ is the initial labeling of $N_v$, $x^i_{v,j}$ is a boolean variable that determines if neighbor $j \in N^i_v$ must be $i$th neighbor in $N_v$ according to relabeling $\mathcal{P}$. If $x^i_{v,j} = 1$, constraints (6), (7) use $N_{i,v} = \mathcal{P} \left( N^i_v \right)$, otherwise they are trivially satisfied. Constraint (8) selects each neighbor once. Finally, constraint (9) sorts $N_v$ in the increasing label order.

5.5 Combining Compactness and ILP (CMB)

Finally, we design combining (CMB) schemes that use the compact recursive partitioning approach to enhance ODB and others. The core idea is to first bisect the graph $k$ times (within the given time constraints), and then encode independently each subgraph (cluster) with a selected ILP scheme. We illustrate an example of this scheme in Figure 5.

![Combining Compactness and ILP](image)

**Figure 5: An example of Hybridization (§ 5.5).**

**What Does It Fix?** First, the initial partitioning does not dominate the total runtime. Second, the NP-hardness of ODB is alleviated as it now runs on subgraphs that are $k$ times smaller than the initial graph. Finally, it is generic and one can use an arbitrary scheme instead of ODB.

**Using Permuters and Transformers** The exact design of $\mathcal{P}$ and $T$ depend on the scheme used for condensing subgraphs. For example, consider ODB. The most significant $k$ bits are now determined by $G$’s partitioning. The remaining bits are derived from ODB independently for each subgraph. Their combination gives each final label. $T$ can be, e.g., Varint.

5.6 Incorporating Degree-Minimizing (DM)

The final step is to relabel vertices so that those with the highest degrees (and thus occurring more often in $A$) 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), similarly to RB. Thus, $|A|$ is decreased as the edges that occur most often are stored using fewer bits.

**What Does It Fix?** First, DM trades some space reductions for faster accesses to $A$ compared to BRB (BRB’s hierarchical encoding entails expensive queries). Second, it does not require costly recursive partitioning. Finally, we later (§ 7.4) show that DM significantly outperforms DMf and matches the compression ratios of the WebGraph library [23].

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

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.

6.1 Modular Design and Extensibility

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

6.2 High Performance

Combinations of the variants of $O$, $L[O]$, $A$, and $L[A] = (\mathcal{P}, T)$ give many possible designs. For example, $O$ can be any succinct bit vector. Now, selecting a specific variant takes place in a performance-critical region such as querying $d$. 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.

1https://spclinf.ethz.ch/Research/Performance/LogGraph
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 \( (O) \) type, the offset compression structure \( (L(O)) \) type, and the transformer \( (T) \) type.

| Type  | ID   | Name                     | \( n \) | \( m \) | \( d \) |
|-------|------|--------------------------|--------|--------|--------|
| uku   | Union of .uk domain      | 133M   | 4.65B | 34.9   |
| uk    | .uk domain               | 118M   | 3.45B | 31.3   |
| sk    | .sk domain               | 59.6M  | 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          | 12.5M  | 288M  | 23.72  |
| tra   | Trackers                 | 27.6M  | 140M  | 5.08   |

Others: tr, ber, gog, sta

| Affiliation     | Graphs           | Social networks | Road networks |
|-----------------|------------------|-----------------|--------------|
| arm             | Orkut Memberships| fr Friendster   | usrn USA road network |
| jfn             | LiveJournal Memberships| tw Twitter | 23.9M 28.3M 1.2 |
|                 |                  | ork Orkut      | Others: rca, rtx, rpa |
|                 |                  | Others: ljn, pok, flc, gow, sl2, epi, you, dbl, amz |

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

- **PR**: A variant without atomic operations [11, 85, 86].
- **TC**: An optimized algorithm that reduces the computational complexity by preprocessing the input graph [31].

7.1.3 Considered Graphs. We analyze synthetic power-law (the Kronecker model [67]), synthetic uniform (the Erdős-Rényi model [43]), and real-world datasets (including SNAP [68], KONECT [65], DIMACS [40], and WebGraph [23]); 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 follows similar patterns.

7.1.4 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 [44, 62].
- **Monte Leone** is an HP DL 360 Gen 9 system. One node has: two Intel E5-2667 v3 @ 3.2GHz Haswells (8 cores), 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.

7.1.5 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.
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.

7.2.1 Log(Graph) Variants and Comparison Targets. We consider four variants of Log(Graph): LG-g (the global approach),
We show that logarithmizing \( \sum_{i=1}^{n} a_i \) reduces \(|A|\) for graphs of various sparsities \( \overline{d} \); see Figure 8. Static succinct bit vectors consistently use the least space. Interestingly, bvSD uses more space than bvEN for graphs with lower \( \overline{d} \leq 15 \). This is because the term \( \log \left( \frac{\log n}{n} \right) \) grows faster with the number of edges than that of bvEN.

7.3.3 Size: Which Bit Vector is the Smallest? We first compare the size of all bit vectors for graphs of various sparsities \( \overline{d} \); see Figure 8. Static succinct bit vectors consistently use the least space. Interestingly, bvSD uses more space than bvEN for graphs with lower \( \overline{d} \leq 15 \). This is because the term \( \log \left( \frac{\log n}{n} \right) \) grows faster with the number of edges than that of bvEN.

| Design  | Dynamic/succinct | Static/simple | Static+succinct |
|---------|-------------------|---------------|-----------------|
| ptr64C  |                   |               |                 |
| ptrLogn |                   |               |                 |
| ptr32  |                   |               |                 |
| ptr16  |                   |               |                 |
| ptrLogn |                   |               |                 |
| ptr32  |                   |               |                 |
| ptr16  |                   |               |                 |
| ptrLogn |                   |               |                 |
| ptr32  |                   |               |                 |
| ptr16  |                   |               |                 |

Figure 7: (§ 7.3) Illustration of the size differences of various O (both offset arrays and bit vectors). The offset sizes are \( W \in \{32, 64, [\log n]\} \).

Figure 8: Analysis of the sizes of bit vectors (plots sorted by the average degree \( \overline{d} \)).
7.3.6 Performance. Finally, we analyze the performance of various $O$ designs queried by $T$ threads in parallel. Each thread fetches offsets of 1,000 random vertices. The results for twt and rca (representing graphs with high and low $d$) are in Figure 11. First, bvEN is consistently slowest due to its complex design; this dominates any advantages from its small size and better cache reuse. Surprisingly, bvEN is followed by bvL that has the biggest $|O|$ (cf. Figure 18); 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 18) and bvSD uses little storage and fits well in cache. Next, we study offset arrays. ptr64 is the fastest for $T \leq 4$ due to least memory operations. Interestingly, the smaller $T$, the lower the latency of ptr64. We conjecture this is because fewer threads cause less traffic caused by the coherence protocol. As for zlib, it entails costly performance overheads as it requires decompression. We tried a modified blocked zlib variant without significant improvements.

7.3.7 Further Analyses. We vary the block size $B$ that controls the granularity of $A$ to be an 8-bit byte or a 64-bit word; see Figure 9. First, larger $B$ reduces each $|O|$ (as $|O|$ is proportional to $B$). Next, $|A|$ grows with $B$. This phenomenon is similar to the internal fragmentation in memory allocation. Here, each $A_x$ is aligned with respect to $B$. The larger $B$, the more space may be wasted at the end of each array.

Other analyses are included in the Appendix (§10.3).

7.3.8 Key Insights and Answers. We conclude that succinct bit vectors are a good match for $O$. First, they reduce $|O|$ 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 multitudes of parameters, both related to input zlib ($d$) and to the utilized AA ($B$ and $A$). They can enhance any system for condensing static or slowly changing graphs that uses $O$.

7.4 Logarithmizing Adjacency Structure $A$

Finally, we evaluate the logarithmization of $A$ 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.

7.4.1 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 (§5.6), the traditional adjacency array (Trad), the state-of-the-art WebGraph (WG) [22] compression system, POD (§5.4.2), and the combination of these two (§5.5). 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.

7.4.2 BRB: Alleviating RB’s Preprocessing. We start with illustrating that BRB alleviates preprocessing overhead inherent to RB. Table 6 shows the overhead from RB compared to a simple AA. Now, BRB’s preprocessing takes equally long if we build the full separator tree. The idea is to build a given limited number of the separator tree levels. We illustrate this analysis in Figure 12. Using fewer partitioning levels increases $|A|$ but also reduces the preprocessing time (it approximately doubles for each new level). Interestingly, the storage overhead from preserving the recursive graph structure begins to dominate at a certain level, annihilating further $|A|$ reductions.

Yet, BRB comes with overheads while resolving $N_v$ because one must construct vertex IDs from bit strings. This results in a 2-2.5x slowdown of obtaining $N_v$, depending on the graph. We conclude that whether to use RB or BRB should depend on the targeted workload: for frequent accesses to $N_v$ one
should use RB while to handle large or evolving graphs that require continual preprocessing one should use BRB.

7.4.3 DMd: Approaching the Time/Space Sweetspot. Next, we illustrate that DMd significantly reduces $|A|$, resolves $N_v$ fast, outperforms WG, uses less storage than DMf, and can be generated fast. The size analysis is shown in Figure 14. 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 $A$ than Trad, with DMd outperforming DMf, being comparable or in many cases better than either RB or BRB (e.g., for ljm). Now, in various cases DMd closely matches WebGraph, for example for tw, fr, ljm. For others, it gives slightly larger $A$ (e.g., for wik). Next, we also derive time to
obtain $N_v$. WG is consistently slower (>2x) than DMd; more results are in Figure 13 and the Appendix (§10.4.4). We conclude that DMd offers the storage/performance sweetspot: it ensures high level of condensing, trades a little storage for fast $N_v$, and finally takes significantly (>10x for RB) less time to generate than any other scheme $A$.

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

7.4.5 Further Analyses. Other analyses include: investigating the ILP schemes, using various types of cuts while building the separator tree, and varying the maximum allowed difference in the sizes of subgraphs derived while partitioning. These analyses are included in the Appendix (§10.4). Here, we conclude that ILP does improve upon RB and DM by reducing the sums of differences between consecutive IDs.

7.4.6 Key Insights and Answers. We conclude that BRB alleviates RB’s preprocessing overheads while DMd offers the best space/performance tradeoff.

7.5 The Log(Graph) Library

We finally evaluate the Log(Graph) library and show that it ensures high performance.

7.5.1 Performance: Graph Algorithms. We use the Log(Graph) library to implement graph algorithms. We present the results for BFS, PR, and TC. We use succinct bit vectors (bvSD) as $O$ and various schemes for $A$. Our modular design based on the established model enables quick and easy implementation of $A$: each variant requires at most 20 lines of code. The results are shown in Figure 15. The BFS and PR analyses for large graphs (gho, orm, tw, usrn) illustrate that DMd is comparable to RB and DMf, merely up to 2x slower than the uncompressed Trad, and significantly faster (e.g., >3x for orm) than WebGraph. The relative differences for TC are smaller because the high computational complexity of TC makes decomposition overheads less severe. Finally, we also study the differences between DMd and RB as well as DMf in more detail in Figure 15b) for a broader set of SNAP graphs. We conclude that DMd offers performance comparable to the state-of-the-art RB as well as DMf, while avoiding costly overheads from recursive partitioning.

7.5.2 Performance: Graph Accesses. We also evaluate obtaining $d_v$ and $N_v$. This also enables understanding the performance of succinct structures in a parallel setting, which is of independent interest. Full results are in the Appendix (§10.4.4). Trad is the fastest (no decoding). The difference is especially visible for bvSD and $f_0$ due to the complex $O$ design. DMd, DMf, and RB differ only marginally (1-3%) due to decoding.

7.6 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 encoding methods. 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 $\approx$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 $O$. First, they reduce $|O|$ 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 multitue of parameters.

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 [73], HAMA [87], GraphLab [69], Spark [102], Galois [63], PBGL [53], GAPS [11], Ligra [90], Gemini [103], Tux$^2$ [100], Green-Mari [56], and others [15, 19, 47]. It could also be used to enhance systems and schemes where graphs are modeled with their adjacency matrix [18, 29, 75, 92]. For example, one could use logarithmized vertex IDs to accelerate graph processing and reduce the pressure on the memory subsystem [12] or network in distributed-memory environments [13, 14, 16, 46, 85].

8.1 Log(Graph) and Compact Schemes

A graph representation based on recursive partitioning, proposed by Blandford et al. [20], was proved to be compact: it takes $O(n)$ bits for an input graph with $n$ vertices. It reduces $|A|$ for several real-world graphs. Yet, its preprocessing is costly. Log(Graph) alleviates it with the BRB scheme.

8.2 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, 21, 34, 39, 50, 51, 57–60, 77, 81, 83, 97]

| Graph | uku | gho | orm | tw | usrn | ema | aml |
|-------|-----|-----|-----|----|------|-----|-----|
| Generation of RB       | 98.15 | 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].
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 [89] construct succinct structures in parallel, but they do not process them in parallel. Finally, there are several libraries of succinct data structures [2, 37, 45, 48, 49, 54]. Contrarily to our work, none of these designs enhances graph processing and they do not address parallel processing of a succinct data structure.

8.3 Log(Graph) and Compression Schemes

A mature compression system for graphs is WebGraph [22]. There are also other works [3, 6, 7, 17, 25, 26, 28, 30, 32, 33, 35, 42, 55, 61, 66, 74, 79, 80, 82, 84, 93, 94, 96]. Some mention encoding some vertex IDs with the logarithmic number of bits [3, 94]: Log(Graph) extends them with schemes such as local logarithmization § 3.2.2. Several works use ILP to relabel vertices to reduce \(|A|\) [30, 42]. Others collapse specified subgraphs into supervertices and merge edges between them into a superedge [28, 82, 93]. These systems come with complex compression and costly decompression. Next, Ligra+ [91] 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 be used to enhance Ligra+ (e.g., with local vertex ID or \(O\) logarithmization) and ensure even more performance. Moreover, G-Store [64] is a storage system for graphs that, among others, removes most significant bit (MSB) zeros of vertex IDs within one tile of 2D partitioning. In general, removing MSB zeros was proposed even before G-Store [3, 94]. 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. [98] 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 [22] 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 \(\approx 20\%\) while reducing the required storage by 20-35%.

To logarithmize offset or adjacency data, we use succinct data structures [58, 81] 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.

ACKNOWLEDGEMENTS

We thank Juraj Hromkovič for inspiring discussions and useful comments that helped us improve the quality of the paper. We thank Guy Blelloch for providing us with the source code of some succinct and compact designs, and Simon Gog for help and useful information. We thank the CSCS and ALCF teams granting access to the Piz Dora and Vesta machines, and for their excellent technical support.

10 APPENDIX

10.1 Theory: Additional Analyses

Here, we first provide the derivation of the expressions for \(\Theta\) and \(\mathcal{A}\) for power-law graphs. We assume that the minimum degree is 1 and also use the recent result that bounds the maximum degree in a power-law graph with high \((1 - \frac{1}{\log n})\)

\[ p \approx \frac{\alpha}{2 - \beta} \left( \frac{an \log n}{\beta - 1} \right)^{\frac{1 + \beta}{\beta}} - 1 \]

This can be approximated with an integral

\[ m \approx \frac{1}{2} \int_{x=1}^{d} ax^{1-\beta} dx = \frac{a}{2} \left( \frac{d^{2-\beta} - 1}{2-\beta} \right) \]

Plugging this into the storage expression, we obtain

\[ E[|\mathcal{A}|] \approx n \left[ \log \left( \frac{a}{2 - \beta} \left( \frac{an \log n}{\beta - 1} \right)^{\frac{1 + \beta}{\beta}} - 1 \right) \right] \]

\[ E[|\Theta|] \approx n \left[ \log \left( \frac{a}{2 - \beta} \left( \frac{an \log n}{\beta - 1} \right)^{\frac{1 + \beta}{\beta}} - 1 \right) \right] \]
We now analyze in more detail how logarithmizing fine elements impacts aspects such as scalability or communicated data. This approach benefits from explicitly considering the locality of data [95]. The results are illustrated in Figure 17.

10.2.1 Investigating Scalability. We also provide the results of scalability analyses. We vary the number of threads $T$ for various real-world graphs, see in Figure 16.

10.2.2 Investigating Distributed-Memory Settings. Finally, we also present results that show how Log(Graph) reduces the amount of communicated data in a distributed-memory environment when logarithmizing fine-grained graph elements. This approach benefits from explicitly considering the locality of data [95]. The results are illustrated in Figure 17.

10.3 Logarithmizing $\theta$: Additional Analyses

In the main body of the work, we have only analyzed the influence of $\theta$’s properties on $|\theta|$. Yet, the scope of the interplay between AA’s parameters is much broader: $|\theta|$ is also impacted by $\mathcal{A}$’s design. Specifically, if $\mathcal{A}$ is encoded using a scheme that shrinks adjacency arrays, such as Blandford’s RB scheme, then the size of $\theta$ based on pointer arrays should remain the same (as $W$ is fixed) while bvPL and bvIL should shrink. Succinct designs are harder to predict due to more...
Figure 18: (§ 10.3) Illustration of the size of various $O$ with and without relabeling vertex IDs (with a traditional adjacency data structure $A$).

Figure 19: (§ 10.4.1) An illustration of sums of edge and vertex cuts at various levels of separator trees in different types of graphs.

complex dependencies; for example $bwSD$’s length also gets smaller but as the ratio of ones to zeros gets higher, $|O|$ may as well increase; a similar argument applies to $bwEN$. We now analyze these effects by comparing the original $|O|$ to $|O|$ after relabeling of vertices according to the inorder traversal of the separator tree as performed in the Blandford’s scheme; see Figure 18. As expected, all offset arrays remain identical because they only depend on fixed parameters. Contrarily, all the bit vectors shrink (e.g., $bvPL$ and $bvSD$ are on average $\approx 25\%$ and $\approx 12\%$ smaller). This is because the relabelled $A$ uses less storage, requiring shorter bit vectors.

10.4 Logarithmizing $A$: Additional Analyses

We also illustrate more analyses related to logarithmizing $A$.

10.4.1 Using Vertex Cuts Instead of Edge Cuts. So far, we have only considered edge cuts (ECs) in the considered recursive partitioning schemes (RB and BRB). Yet, as explained in § 5.3.1, vertex cuts (VCs) can also be incorporated to enhance RB. They seem especially attractive as it can be proven that they are always smaller or equal than the corresponding ECs [99]. In our setting, this relationship is more complicated as we partition graphs recursively and the correspondence between ECs and VCs is lost. Thus, we first compute the total sums of sizes of ECs and VCs at various levels of respective separator trees, see Figure 19. We ensure that the respective partitions are of almost equal sizes. We did not find strong correlation between cut sizes and sparsities $d$. We group selected representative graphs into social networks (SNs), purchase networks (PNs), and road graphs (RNs). For most SNs, ECs start from numbers much larger than in the corresponding
VCs, and then steadily decrease. This is due to vertices with very high degrees whose edges are cut early during recursive partitioning. Contrarily, VCs in SNs start from very small values and then grow. This is because it is easy to partition initial input SNs using VCs as they are rich in communities [101], but later on, as communities become harder to find, cut sizes grow. Then, ECs and VCs in PNs follow similar patterns; they increase together with levels of separator trees. This suggests that in both cases it becomes more difficult to find clusters after several initial partitioning rounds. Finally, ECs and VCs in RNs differ marginally because these graphs are almost planar.

We conclude that, in most cases, VCs are significantly smaller than ECs, being potentially a more appealing tool in reducing $|\mathcal{A}|$ because less information crosses partitions and has to be encoded as large differences in RB. Yet, $\mathcal{A}$ based on VCs introduces redundancy as now some vertices are present in graph separators as well as in graph partitions. It requires additional lookup structures with shadow pointers [20] for mapping between such vertex clones (i.e., shadow...
vertex trees [20]). We calculated the number of shadow pointers in such structures and the resulting storage overheads in $|\alpha|$ based on VCs. All the graphs follow similar trends. For example, twt requires 3.53M additional pointers, giving 4.81MB for $|\alpha|$ (VCs, ptrLogn), as opposed to 3.1MB (ECs). Thus, the additional complexity from shadow vertex trees removes advantages from smaller cuts, motivating us to focus on ECs.

10.4.2 Relaxing Balancedness of Partitions. While bisecting $G$, we now let the maximum relative imbalance between the partition sizes be at most $D$. We first analyze how changing $D$ influences sizes of ECs and VCs for each level of separator trees. We plot the findings for representative graphs in Figure 20. As expected, cuts become smaller with growing $D$: more imbalance more often prevents partitioning clusters. Yet, these differences in sizes of both ECs and VCs are surprisingly small. For example, the difference between ECs for twt (level 1) is only around $\pm 2\%$; other cases follow similar patterns. They do not impact the final $|\alpha|$, resulting in minor ($\pm 1\%$) differences.

10.4.3 Approaching the Optimal Labeling with POD/CMB. We now use POD and HYB to approach optimal labeling and outperform RB and DM. We use IBM CPLEX [36] to solve the ILP problems formulated in § 5.4.1 and § 5.4.2. We use two graphs $g_1$ and $g_2$, both consisting of two communities with few ($<0.2m$) edges in-between. Here, we illustrate (Figure 22) that POD/HYB do improve upon RB and DM by reducing sums of differences between consecutive labels. Each scheme is denoted as ODB-$y$-$z$: $y$ indicates the variant of the objective function ($y=1$ for Eq. (4) and $y=2$ for Eq. (5)) and $z$ determines if we force the obtained labels to be contiguous ($z=c$) or not ($z=u$). Each proposed scheme finds a better labeling than RB (by $5-10\%$) and DM (by $30-40\%$).

10.4.4 Investigating Performance of Graph Accesses. Here, we present the full results of the performance of obtaining $d_v$ and $N_v$ that we discussed briefly in § 7.4.

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. A. 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, M. Uehara, 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. Barbagli, 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:1508.03639, 2015.
[12] M. Besta, S. Chu, J. Cheng, and P. Raghavan. On compressing social networks. In Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery and data mining, pages 237–246. ACM, 2010.
[13] M. Besta, J. K. Ooi, and M. Mitzenmacher. Towards compressing web graphs. In Proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining, pages 672–680. ACM, 2011.
[14] S. Boldi and S. Vigna. The WebGraph Framework I: compression techniques. WWW, 2004.
[15] S. Boldi and S. Vigna. The WebGraph Framework II: compression techniques. In World Wide Web Conf. (WWW), pages 595–601, 2004.
[16] J. Barbay, L. Castelli Aleardi, M. He, and J. Munro. Succinct Representations of Separable Graphs. SODA, 2003.
[17] G. Blelloch and A. Farzan. Succinct representations of separable graphs. In Amir and M. Parida, editors, Combinatorial Pattern Matching, volume 17 of Lecture Notes in Computer Science, pages 138–150. Springer Berlin Heidelberg, 2010.
[18] 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.
[19] F. Ciaramella, S. Lucarelli, A. Mitzenmacher, S. 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.
[20] 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.
[21] F. Claude and S. Ladra. Practical representations for web and social graphs. In Proceedings of the 25th ACM international conference on Information and knowledge management, pages 1185–1190. ACM, 2011.
| Scheme       | (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 23: Log(Graph) performance analysis, logarithmizing fine elements, \( n = 2^{24}, \ T = 16 \) (full parallelism), Kronecker graphs.

[33] 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.
[34] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In In Proc. 15th SPIRE, LNCS 5280, pages 176–187, 2008.
[35] F. Claude and G. Navarro. Extended compact web graph representations. Algorithms and Applications, 6060:77–91, 2010.
[36] I. I. CPLEX. V12. 1: User’s Manual for CPLEX. International Business Machines Corporation, 46(53):157, 2009.
[37] Daisuke Okanohara. rsdic - Compressed Rank Select Dictionary. available at: http://code.google.com/p/rsdic.
[38] 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.
[39] E. Demaine. Advanced Data Structures, 2012. Lecture Notes.
[40] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. The Shortest Path Problem: Ninth DIMACS Implementation Challenge, volume 74. American Math. Soc., 2009.
[101] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[102] 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.

[103] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.