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

## ABSTRACT

Various graphs such as web or social networks may contain up to hundreds of billions of edges, yet they are not necessarily stored and thus managed efficiently. To address this, we propose Log(Graph): a graph representation that enables cheaper and faster graph processing for various families of graphs. The core idea is to encode the graph so that various elements approach or match respective storage lower bounds. We call our approach "graph logarithmization" because logarithmic terms are applied to various graph elements to derive the respective bounds. We use various techniques and notions to achieve this, including succinctness, compactness, and integer linear programming. We also show a high-performance design of Log(Graph) that uses modern bitwise operations, performance models, and parallel implementation for higher speedups. Our evaluation illustrates that Log(Graph) achieves simplicity and tunable performance as well as compression ratios. Log(Graph) can be used to improve the design of any graph processing framework, algorithm implementation, or library on both fat shared-memory NUMA nodes and distributed-memory data centers and supercomputers.

## 1 INTRODUCTION

Large graphs are behind many problems in machine learning, medicine, social network analysis, and computational sciences [54]. Lowering the size of such graphs (e.g., through compression) is becoming increasingly important for HPC and Big Data. First, it reduces expensive I/O. Next, it potentially improves performance by storing a larger fraction of data in caches. Third, it decreases the amount of hardware resources required to store a given graph.

Traditional lossless graph compression systems such as the well-known WebGraph [12] introduce costly decompression. This is often caused by the underlying complexity (e.g., WebGraph combines various techniques, such as sophisticated variable-length codes, reference encoding, and storing intervals instead of vertex IDs). Thus, one needs both *simplicity* and *high performance* for *lossless* compression.

In this work, we develop Log(Graph): a graph representation that achieves the above goals. Log(Graph) is founded on two fundamental ideas. First, to enable significant storage reductions, it *applies storage lower bounds* to various elements of the popular adjacency array graph representation. We call this approach "logarithmization" because the well-known combinatorial argument states that a lower bound to store an object from some set (or class) is the logarithm of the cardinality of this set (or class); the name Log(Graph) indicates that we "logarithmize" various elements of a graph representation such as vertex IDs or adjacency offsets. Second, we ensure that all the "logarithmization" schemes maintain high performance. The result is a representation that combines storage reductions with fast accesses to the graph data.

To motivate and explain the "logarithmization" idea for vertex IDs, consider using a fixed-size 64- and 32-bit memory word to respectively store a vertex ID and an edge weight (a strategy used in miscellaneous graph processing codes such as the GAP Benchmark Suite [8]). Yet, we note that the storage lower bound (for a graph with $n$ vertices and the maximum edge weight $\mathcal{W}_{max}$) associated with vertex IDs and edge weights is $\lceil \log n \rceil$ and $\lceil \log \mathcal{W}_{max} \rceil$ bits. Now, for a graph with $n = 2^{37}$ and $\mathcal{W}_{max} = 2^{10}$, 42% of each word with a vertex ID and 69% of each word with an edge weight is wasted. We instead "logarithmize" IDs and use $\lceil \log n \rceil$ bits to store an ID and $\lceil \log \mathcal{W}_{max} \rceil$ bits to store an edge weight.

Other elements of a graph representation use more sophisticated designs. For example, we "logarithmize" an offset array by implementing it with a bit vector and then using an interesting family of *succinct* designs [45, 64] that approach theoretical storage lower bounds while enabling constant-time queries, without compressing or decompressing the bit vector data. We show that they can provide storage-efficient offsets into the adjacency data, asymptotically reducing the usual $O(n \log n)$ bits used in traditional offset arrays.

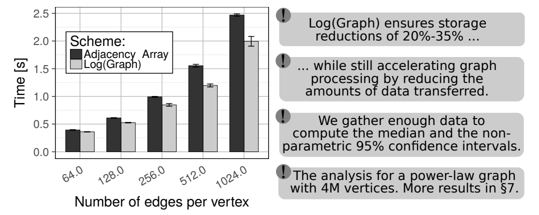


**Figure 1: (§ 1, § 7.2) The performance of Log(Graph) with the Single Source Shortest Path algorithm when logarithmizing vertex IDs.**

Now, these ideas offer simplicity but come with challenges to be solved and questions to be answered to achieve high performance. For example, if we use $\lceil \log n \rceil$ bits for a vertex ID instead of a fixed-size memory word, how do we ensure fast extraction of the graph connectivity information? Can we use fast bitwise operations available in today's architectures? Which ones? How to logarithmize other graph elements?

In this work, we answer these and other questions and propose a set of schemes that are portable and straightforward to implement in any graph processing code, framework, or library. We motivate Log(Graph) in Figure 1 that illustrates example results for the Single Source Shortest Path (SSSP) algorithm running over traditional adjacency arrays and a selected Log(Graph) variant. Log(Graph) does not only reduce storage requirements, but also reduces data transfers, accelerating SSSP. The specific contributions are as follows:

*Conference'17, Washington, DC, USA*
© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

| | | |
|---|---|---|
| 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 $G$; $\|V\| = n, \|E\| = m$. |
| | $\mathcal{W}_{(v,w)}, D$ | The weight of an edge $\mathcal{W}_{(v,w)}$ and the diameter of $G$. |
| | $d_v, N_v, N_{i,v}$ | Degree and neighbors and $i$th neighbor of a vertex $v$; $N_{0,v} \equiv v$. |
| | $\overline{x}, \widehat{x}$ | The average and the maximum among $x$. |
| | $\alpha, \beta; p$ | Parameters of a power-law graph and an Erdős-Rényi graph. |
| Machine model | $N$ | The number of levels in a hierarchical machine. |
| | $H_i, H_{node}$ | Total number of elements from level $i$ and compute nodes. |
| | $T, P, W$ | The number of threads/processes and the memory word size. |
| | $\mathcal{T}_x$ | Time to do a given operation $x$. |
| Adjacency array | $\mathscr{A}, \mathscr{A}_v$ | The adjacency array of a given graph and a given vertex. |
| | $\mathscr{O}, \mathscr{O}_v$ | The offset structure of a given graph and an offset to $\mathscr{A}_v$. |
| | $\|\mathscr{A}\|, \|\mathscr{O}\|$ | The sizes of $\mathscr{A}, \mathscr{O}$. |
| | $C[\mathscr{A}], C[\mathscr{O}]$ | Compression schemes acting upon $\mathscr{A}, \mathscr{O}$. |
| | $B, L$ | Various parameters of $\mathscr{A}$ and $\mathscr{O}$; see § 4.3 for details. |
| Schemes for $\mathscr{A}$ | $\mathscr{P}$ | Permuter: function that relabels vertices. |
| | $\mathscr{T}_x, \mathscr{T}$ | Transformers: functions that arbitrarily modify $\mathscr{A}$. |
| | $G_x$ | Subgraphs of $G$ constructed in recursive partitioning. |

**Table 1: Symbols used in the paper.**

- We propose Log(Graph): a representation that losslessly reduces storage required for graphs, ensuring both simplicity and high performance. The core idea is to "logarithmize" (i.e., apply logarithmic storage lower bounds to) various parts of adjacency arrays such as vertex IDs or offset arrays.
- We illustrate how to implement Log(Graph) to ensure high performance and portability using various bitwise operations available in state-of-the-art architectures and optimized succinct and compact data structures.
- We enhance Log(Graph) with fixed-size gap encoding and Integer Linear Programming (ILP).
- We extend Log(Graph) to distributed-memory settings.
- We evaluate Log(Graph) on various graphs (synthetic and real-world) with a broad set of algorithms (BFS, PageRank, Connected Components, Betweenness Centrality, Triangle Counting, Single Source Shortest Paths). We show that graphs compressed with Log(Graph) can be processed as fast as uncompressed graphs with tuned graph processing codes. We also illustrate significant speedups over state-of-the-art compression systems such as WebGraph [12].

*Finally, our work presents the first analysis of succinct data structures in a parallel setting, which is of independent interest.*

## 2 BACKGROUND, NOTATION, CONCEPTS

We first describe the used concepts and notation; the most important symbols are gathered in Table 1 for clarity. To design Log(Graph) we use multiple techniques and data structures and we postpone describing some of the used specific schemes to their related sections for better readability.

### 2.1 Used Models

We start with the used models of graphs and machines.

**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; $|V| = n$, $|E| = m$. We assume that vertices have contiguous IDs ($\equiv$ labels) from the set $\{0, ..., n-1\}$. $d_v$ and $N_v$ denote the degree and the neighbors of a vertex $v$. The $i$th neighbor of $v$ (in the order of increasing labels) is denoted as $N_{i,v}$; $N_{0,v} \equiv v$. $D$ is $G$'s diameter. Now, $\widehat{X}$ and $\overline{X}$ indicate the maximum and average value in a given set or sequence, for example $\widehat{N_v}$ is the maximum ID among $v$'s neighbors, $\widehat{W}$ is the maximal edge weight in $G$, $\widehat{W_v}$ is the maximal weight among edges originating at $v$, $\widehat{d}$ and $\overline{d}$ are $G$'s maximal and average degree.

**Representation Model** We present a unified model for graph representations. A representation is modeled as a tuple $(G, \mathscr{O}, \mathscr{A}, C[\mathscr{O}], C[\mathscr{A}])$. $\mathscr{O}$ is an offset structure that keeps the location of the adjacency structure $\mathscr{A}_v$ of each vertex $v$. $\mathscr{A} = \bigcup_{v \in V} \mathscr{A}_v$ is the adjacency data. Both $\mathscr{O}$ and $\mathscr{A}$ may be arbitrary data structures. $C[\mathscr{O}]$ and $C[\mathscr{A}]$ are compression schemes for $\mathscr{O}$ and $\mathscr{A}$. The size of the representation is $|\mathscr{O}| + |\mathscr{A}|$ [bits] ($|\mathscr{O}|$ and $|\mathscr{A}|$ denote the sizes of $\mathscr{O}$ and $\mathscr{A}$).

**Machine and System Model** For more storage reductions on today's hardware, we cover 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 $H_{node}$. 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.

**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, *succinct* if it uses $OPT + o(OPT)$ bits, and *implicit* if it takes $OPT + O(1)$ bits [29]. They all should support *a reasonable set of queries in (ideally)* $O(1)$ *time* [10].

**Compression** Traditional compression mechanisms such as zlib [31] differ from succinct schemes as the latter can be accessed without expensive decompression.

**Condensing** To avoid confusion, we use the term *condensing* to refer in general to reducing the size of some data.

We also describe the concept of **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

Finally, we describe structures used in Log(Graph): adjacency array that is the Log(Graph) foundation and succinct as well as compact bit vectors used to condense $G$.

**Graph Adjacency Arrays** Log(Graph) builds upon the traditional adjacency array (AA) representation. An AA consists of a contiguous array with the adjacency data ($\mathscr{A}$) and an array with offsets to the neighbors of each vertex ($\mathscr{O}$). The neighbors of each $v$ form an array $\mathscr{A}_v$ sorted by vertex IDs; all such arrays form a contiguous array $\mathscr{A}$. $\mathscr{O}_v$ is the offset to $v$'s adjacency data. An example AA is in Figure 2 (❶).

**Bit Vectors and Rank/Select Queries** The next data structure are simple bit vectors that we will use to enhance $\mathscr{O}$ (§ 4.1, § 4.3). A bit vector $S$ of length $L$ takes only $L$ bits, but uses $O(L)$ time to answer two important queries: $rank_S(x)$ and $select_S(x)$. For a given $S$, $rank_S(x)$ returns the number of ones in $S$ up to and including the $x$th bit. Conversely, $select_S(x)$ returns the position of the $x$th one in $S$. Now, many designs with *rank* and *select* answering in $o(L)$ time have been proposed. An important family are *succinct bit vectors*.

**Succinct Bit Vectors** Succinct bit vectors of length $L$ use $L + o(L)$ bits. The first term is the amount of space for storing the actual data. The second term is the space for an additional structure that enables answering *rank*/*select* in

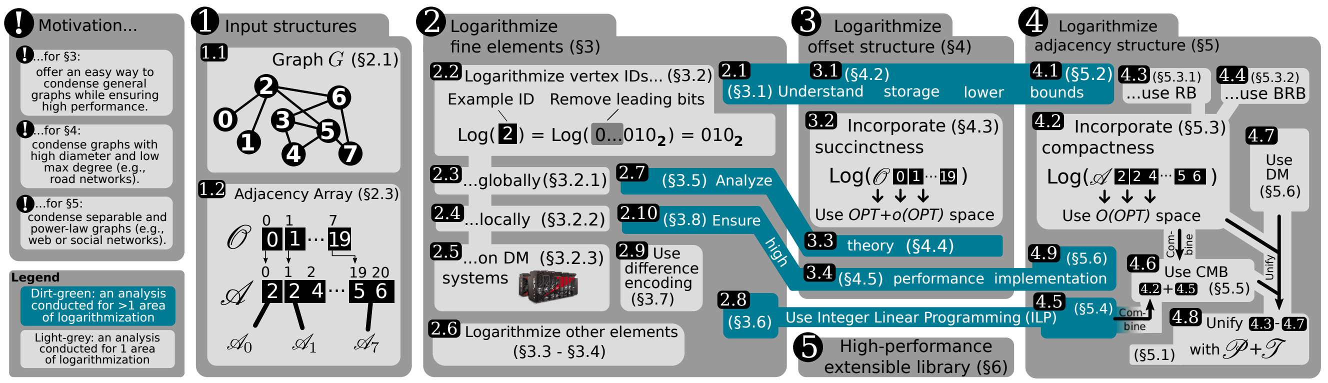

Motivation...
...for §3: offer an easy way to condense general graphs while ensuring high performance.
...for §4: condense graphs with high diameter and low max degree (e.g., road networks).
...for §5: condense separable and power-law graphs (e.g., web or social networks).
Legend
Dirt-green: an analysis conducted for >1 area of logarithmization
Light-grey: an analysis conducted for 1 area of logarithmization
1 Input structures
1.1 Graph G (§2.1)
1.2 Adjacency Array (§2.3)
2 Logarithmize fine elements (§3)
2.2 Logarithmize vertex IDs... (§3.2)
Example ID Remove leading bits
Log(2) = Log(0...010₂) = 010₂
2.3 ...globally (§3.2.1)
2.4 ...locally (§3.2.2)
2.5 ...on DM (§3.2.3) systems
2.6 Logarithmize other elements (§3.3 - §3.4)
2.7 (§3.5) Analyze
2.10 (§3.8) Ensure high
2.9 Use difference encoding (§3.7)
2.8 (§3.6) Use Integer Linear Programming (ILP)
2.1 (§3.1) Understand storage lower bounds
3 Logarithmize offset structure (§4)
3.1 (§4.2)
3.2 Incorporate (§4.3) succinctness
Use OPT+o(OPT) space
3.3 theory (§4.4)
3.4 (§4.5) performance implementation
4 Logarithmize adjacency structure (§5)
4.1 (§5.2)
4.3 (§5.3.1) ...use RB
4.4 (§5.3.2) ...use BRB
4.2 Incorporate (§5.3) compactness
Use O(OPT) space
4.7 Use DM (§5.6)
4.9 (§5.6)
4.6 Use CMB 4.2+4.5 (§5.5)
4.5 (§5.4)
Com-bine
Unify
4.8 Unify 4.3-4.7 with 𝒫+𝒯 (§5.1)
5 High-performance extensible library (§6)

**Figure 2: (§ 2.4) The roadmap of incorporated schemes. The green areas indicate analyzes and themes shared by multiple logarithmization areas.**

$o(L)$ time. Many such designs exist [38, 44, 45, 66, 70] and are widely used in space-efficient trees and other schemes such as dictionaries [29, 66]. A common idea behind many of these schemes is to precompute answers to some *rank* and *select* queries so that neither space nor time exceeds $o(L)$. For example, to achieve *rank* answering in $O(1)$ time, one divides $S$ into *mini* blocks of size $\log^2 L$ bits. The rank of the first bit of each mini block is precomputed and stored in an auxiliary table; each such entry takes $\log L$ bits and there are $L/\log^2 L$ entries, giving the total of $L/\log L \in o(L)$ bits. Each mini block is then divided into *micro* blocks of size $\frac{1}{2}\log L$. Again, the rank of the first bit is stored in another auxiliary table. However, now only the differences from the rank of the first bit in the containing mini block must be stored. Thus, each entry takes $\log\log^2 L = 2\log\log L$ bits and there are $L/\frac{1}{2}\log L$ such entries, giving the total of $4L\log\log L/\log L$ bits of storage. Finally, a lookup table with $o(n)$ bits stores an answer to each possible *rank* on a bit vector of length $\log n/2$. Thus, all tables take $o(L)$ bits: $L/\log L$ (for mini blocks) + $4L\log\log L/\log L$ (for micro blocks) + $o(L)$ (for the lookup table). They answer *rank* [29] in $O(1)$ time with three accesses into: a mini block (to get the offset of a targeted micro block), a micro block (to get the offset of the entry in the lookup table), and a lookup table (to get the targeted data).

## 2.4 Motivation and Roadmap of Schemes

Log(Graph) uses a plethora of schemes for various graphs and elements of AA. To enhance readability, we present a motivation summary and roadmap in Figure 2. The numbers in circles indicate the best order of viewing the figure. Section numbers refer to the associated parts of the paper.

## 3 LOGARITHMIZING FINE ELEMENTS

We start by applying lower bounds to fine elements of a graph representation: vertex IDs, vertex offsets, and edge weights.

## 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 Figure 2 (❷) and 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 a graph with $m$ edges there are $2m$ cells. Third, a

| | 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 graph. |
| | Edge weight | $\log \widehat{\mathcal{W}}$ | - |
| $\mathscr{O}$ (§ 4.2) | Bit vector | $\log \binom{\frac{2Wm}{B}}{n}$ | $n$ set bits among $\frac{2Wm}{B}$. |
| $\mathscr{A}$ (§ 5.2) | Graph | $\log \binom{\binom{n}{2}}{m}$ | The graph is undirected. |

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

storage lower bound of an edge weight from a discrete set $\{0,\ldots,\widehat{\mathcal{W}}\}$ is $\lceil\log\widehat{\mathcal{W}}\rceil$ (for continuous weights, we first scale them appropriately to become elements of a discreet set).

## 3.2 Logarithmization of Vertex IDs

We first consider vertex IDs.

*3.2.1 Vertex IDs (The Global Approach).* The first and simplest step in Log(Graph) is to use the storage lower bound of a vertex ID in a graph with $n$ vertices. Assume that a vertex ID uses $W$ bits of space. In various AA designs, $W \in \{32, 64\}$ bits and it thus corresponds to the size of 32- or 64-bit memory word. Log(Graph) extends this approach and uses the lowest applicable value: $W = \lceil\log n\rceil$ bits, giving

$$|\mathscr{A}| = 2m\lceil\log n\rceil \qquad (1)$$

*3.2.2 Vertex IDs: The Local Approach.* The advantage of the global approach described above lies in simplicity. Yet, even if it uses an optimal number or bits to store $n$ vertex IDs, it may be far from optimal when considering subsets of these vertices. For example, consider a vertex $v$ such that $d_v \ll n$ and $\widehat{N}_v \ll n$. Here, using $\lceil\log n\rceil$ bits for a vertex ID in $v$'s adjacency list results in unnecessary storage overheads as one may need much fewer then $\lceil\log n\rceil$ bits for a short adjacency list. Here, we use $\lceil\log\widehat{N}_v\rceil$ bits to store a vertex ID in $\mathscr{A}_v$. The tradeoff is that one must also keep the information on the number of bits required for each $v$. We use the fixed number of bits; it is lower bounded by $\lceil\log\log\widehat{N}_v\rceil$ bits. Then

$$|\mathscr{A}| = \sum_{v\in V}\left(d_v\left\lceil\log\widehat{N}_v\right\rceil + \left\lceil\log\log\widehat{N}_v\right\rceil\right) \qquad (2)$$

*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 $\frac{n}{H_{node}}$. The intra ID part takes $\left\lceil \log \frac{n}{H_{node}} \right\rceil$; the inter one takes $\lceil \log H_{node} \rceil$. As the inter part is unique for a given node, it is stored once per node. Thus

$$|\mathscr{A}| = n \left\lceil \log \frac{n}{H_{node}} \right\rceil + H_{node} \lceil \log H_{node} \rceil \tag{3}$$

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 $\sum_{j \in \{2..N-1\}} \lceil \log H_j \rceil$ bits and has to be stored once per each machine element, thus

$$|\mathscr{A}| = n \left\lceil \log \frac{n}{H_N} \right\rceil + \sum_{j=2}^{N-1} H_j \lceil \log H_j \rceil \tag{4}$$

## 3.3 Logarithmization of Edge Weights

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

$$|\mathscr{A}| = 2m \left( \lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) \tag{5}$$

$$|\mathscr{A}| = \sum_{v \in V} \left( d_v \left( \left\lceil \log \widehat{N}_v \right\rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) + \left\lceil \log \log \widehat{N}_v \right\rceil + \left\lceil \log \log \widehat{\mathcal{W}} \right\rceil \right) \tag{6}$$

## 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 $\mathscr{A}$ that may reach $2m$. Thus, the related lower bound is $\lceil \log 2m \rceil$, giving $|\mathscr{O}| = n \lceil \log 2m \rceil$.

## 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

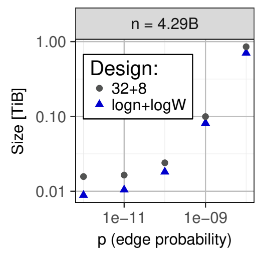
$$E[|\mathscr{A}|] = \left( \lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) pn^2 \tag{7}$$

$$E[|\mathscr{O}|] = n \left\lceil \log \left( 2pn^2 \right) \right\rceil = n \lceil \log 2p + 2 \log n \rceil \tag{8}$$

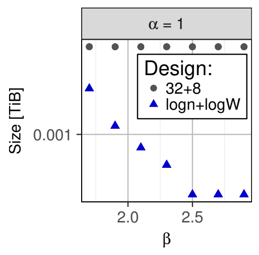
**Power-Law Graphs** We next analyze power-law graphs; the full derivation is in the Appendix (§ 10.1). Here, the probability that a vertex has degree $d$ is $f(d) = \alpha d^{-\beta}$, and

$$E[|\mathscr{A}|] \approx \frac{\alpha}{2-\beta} \left( \left( \frac{\alpha n \log n}{\beta - 1} \right)^{\frac{2-\beta}{\beta-1}} - 1 \right) \left( \lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil \right) \tag{9}$$

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). Logarithmization 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



**(a) Random-uniform graphs.**



**(b) Power-law graphs.**

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

with ILP and gap encoding (for less storage) and efficient design (for more performance).

## 3.6 Incorporating Integer Linear Programming

Here, we reduce the size of vertex IDs by augmenting local logarithmization (§ 3.2). Consider any $\mathscr{A}_v$. We observe that a single neighbor in $\mathscr{A}_v$ with a large ID may vastly increase $|\mathscr{A}_v|$: we use $\left\lceil \widehat{N}_v \right\rceil$ bits to store each neighbor in $\mathscr{A}_v$ but other neighbors may have much lower IDs. Thus, we reorder vertex IDs to *preserve the graph structure* but *reduce maximal IDs in as many neighborhoods as possible*. We illustrate an ILP formulation and then propose a heuristics. The new objective function is presented in Eq. (15). It minimizes the weighted sum of $\widehat{N}_v$ for all $v \in V$ (Eq. 15). Each maximal ID is given a positive weight which is the inverse of the neighborhood size ($d_v$); this intuitively decreases $\widehat{N}_v$ in smaller $\mathscr{A}_v$.

$$\min \sum_{v \in V} \widehat{N}_v \cdot \frac{1}{d_v} \tag{10}$$

In Constraint (11), we set $\widehat{N}_v$ to be the maximum of the new IDs assigned to the neighbors of $v$.

$$\forall_{v,u \in V} \left( u \in N_v \right) \Rightarrow \left[ \mathscr{P}(u) \leq \widehat{N}_v \right] \tag{11}$$

Listing 1 describes a greedy polynomial-time heuristic for changing IDs. We sort vertices in non-decreasing order of their degrees (Line 5). Next, we traverse vertices in the sorted order, beginning with the smallest $|\mathscr{A}_v|$ and assign a new smallest ID possible (Line 6). The remaining vertices that are not a part of any $N_v$ are relabeled in Line 10. This scheme acts similarly to the proposed ILP.

```
1 /* Input: $G$, Output: a new relabeling $\mathscr{P}(v), \forall v \in V$. */
2 void relabel($G$) {
3   $ID[0..n-1]$ = $[0..n-1]$; $D[0..n-1]$ = $[d_0..d_{n-1}]$;
4   $visit[0..n-1]$ = $[false..false]$; $nl$ = 1;
5   sort($ID$); sort($D$);
6   for(int $i$ = 1; $i$ < $n$; ++$i$)
7     for(int $j$ = 0; $j$ < $D[i]$; ++$j$) {
8       int $id$ = $N_{j,ID[i]}$;
9       if($visit[id]$ == $false$) { $\mathscr{P}(id)$ = $nl$++; $visit[id]$ = $true$; } }
10  for(int $i$ = 1; $i$ < $n$; ++$i$)  if($visit[i]$ == $false$) $\mathscr{P}(id)$ = $nl$++; }
```

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

## 3.7 Incorporating Fixed-Size Gap Encoding

We next use the popular gap encoding technique to further reduce $|\mathscr{A}|$. Traditionally, in gap encoding one calculates the difference between all consecutive neighbors in each $\mathscr{A}_v$. As it may be arbitrary, it is then encoded with a variable-length code such as Varint [19]. We observe that this may entail significant decoding overheads. To alleviate this, we use *fixed-size gap encoding* where the maximum difference within a given domain determines the number of bits used to encode this and other differences. In the global approach, the maximum difference for the whole $\mathscr{A}$ is used to calculate the number of bits used to encode any other difference. In the local approach, the maximum difference in each $\mathscr{A}_v$ determines the number of bits to encode any difference in the same $\mathscr{A}_v$.

## 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 technical report[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 [39].

| 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 $\mathscr{A}$ 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 $\mathscr{O}[v]$ of $v$'s neighbors' array (Line 3). This usually involves a cache miss, taking $\mathcal{T}_{cm}$. Second, we derive $o = s \cdot \mathscr{O}[v]$ (the exact bit position of $N_{i,v}$); it takes $\mathcal{T}_{mul}$ (Line 3). Third, we find the closest byte alignment before $o$ by right-shifting $o$ by 3 bits, taking $\mathcal{T}_{shf}$ (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 $\mathcal{T}_{and}$. We can then access the derived 64-bit value; this involves another cache miss ($\mathcal{T}_{cm}$). 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 $\mathcal{T}_{bxr}$. 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: $\mathscr{O}[v+1] - \mathscr{O}[v]$.

[1]To be released after the publication due to double-blindness.

**Accessing an Edge Weight** We store the weight of each edge directly after the corresponding vertex ID in $\mathscr{A}$. 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 $\mathcal{T}_{cm}$.

**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

$$\mathcal{T}_{edge} = 2\mathcal{T}_{cm} + \mathcal{T}_{mul} + \mathcal{T}_{shf} + \mathcal{T}_{and} + \mathcal{T}_{bxr}$$

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

$$\mathcal{T}_{neigh}(v) = \mathcal{T}_{edge} + (d_v - 1)(\mathcal{T}_{add} + \mathcal{T}_{shf} + \mathcal{T}_{and} + \mathcal{T}_{bxr})$$

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

$$\mathcal{T}_{degree} = 2\mathcal{T}_{cm} + \mathcal{T}_{sub}$$

## 4 LOGARITHMIZING THE $\mathscr{O}$ STRUCTURES

We now logarithmize *the whole offset structure* $\mathscr{O}$, treating it as a single entity with its own associated storage lower bound; see Figure 2 (❸). For this, we use bit vectors instead of offset arrays and then *apply succinctness* to approach the storage lower bound, significantly reducing space for some graphs.

### 4.1 Offset Arrays vs. Bit Vectors

Usually, $\mathscr{O}$ is an array of $n$ offsets and $|\mathscr{O}|$ is much smaller than $|\mathscr{A}|$. Still, in sparse graphs with low maximal degree $\hat{d}$, $|\mathscr{O}| \approx |\mathscr{A}|$ or even $|\mathscr{O}| > |\mathscr{A}|$. For example, for the USA road network, if $\mathscr{O}$ contains 32-bit offsets, $|\mathscr{O}| \approx 0.83|\mathscr{A}|$.

To reduce $|\mathscr{O}|$, one can use a bit vector. For this, $\mathscr{A}$ is divided into blocks (e.g., bytes or words) of a size $B$ [bits]. Then, if $i$th bit of $\mathscr{O}$ is set (i.e., $\mathscr{O}[i] = 1$) and if this is $j$th set bit in $\mathscr{O}$, then $\mathscr{A}_j$ starts at $i$th block. The key insight is that getting the position of $j$th set bit and thus the offset of vector $j$ is equivalent to performing $select_{\mathscr{O}}(j)$ (cf. § 2.3).

Yet, *select* on a raw bit vector takes $O(n)$ time. Thus, we first incorporate two designs that enhance *select*: *Plain* (bvPL) and *Interleaved* (bvIL) bit vectors [36]. They both trade some space for faster *select*. bvPL uses up to $0.2|\mathscr{O}|$ 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 $O(\log |\mathscr{O}|)$ time [36]. Neither bvPL nor bvIL achieve succinctness; we use them (1) as reference points and (2) because they also enable a smaller yet simple $\mathscr{O}$.

### 4.2 Understanding Storage Lower Bounds

A bit vector that serves as an $\mathscr{O}$ 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 $\mathscr{A}$, each stored using $W$ bits) grouped in blocks of size $B$ bits. Now, there are exactly $\mathscr{L} = \binom{\frac{2Wm}{B}}{n}$ bit vectors of length $\frac{2Wm}{B}$ with $n$ ones and the storage lower bound is thus $\lceil \mathscr{L} \rceil$ bits.

### 4.3 Incorporating Succinctness

To condense $\mathscr{O}$ further, we propose to use succinct bit vectors. Note that these are simply various forms of $\mathscr{O}$ and they do *not* entail traditional compression. First, we use the *entropy*

| $\mathscr{O}$ | ID | **Asymptotic size** [bits] | **Exact size** [bits] | *rank* | *select* | Deriving $\mathscr{O}_v$ |
|---|---|---|---|---|---|---|
| Pointer array | ptr$W$ | $O(Wn)$ | $W(n+1)$ | - | - | $O(1)$ |
| Plain [36] | bvPL | $O\left(\frac{Wm}{B}\right)$ | $\frac{2Wm}{B}$ | $O\left(\frac{Wm}{B}\right)$ | $O(1)$ | $O(1)$ |
| Interleaved [36] | bvIL | $O\left(\frac{Wm}{B} + \frac{Wm}{L}\right)$ | $2Wm\left(\frac{1}{B} + \frac{64}{L}\right)$ | $O(1)$ | $O\left(\log \frac{Wm}{B}\right)$ | $O\left(\log \frac{Wm}{B}\right)$ |
| Entropy based [24, 66] | bvEN | $O\left(\frac{Wm}{B} \log \frac{Wm}{B}\right)$ | $\approx \log \binom{\frac{2Wm}{B}}{n}$ | $O(1)$ | $O\left(\log \frac{Wm}{B}\right)$ | $O\left(\log \frac{Wm}{B}\right)$ |
| Sparse [64] | bvSD | $O\left(n + n \log \frac{Wm}{Bn}\right)$ | $\approx n\left(2 + \log \frac{2Wm}{Bn}\right)$ | $O\left(\log \frac{Wm}{Bn}\right)$ | $O(1)$ | $O(1)$ |
| 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 | $O\left(\frac{Wm}{B} \log \frac{Wm}{Bn}\right)$ | $\approx 1.3 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn}$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

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

*based* bit vector (bvEN) [24, 66]. The key idea behind bvEN is to use a dictionary data structure [16] that achieves the lower bound for storing bit vectors of length $2Wm/B$ with $n$ ones that is presented in § 4.2. Second, we use *sparse* succinct bit vectors (bvSD) [64]. 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 illustrate 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 $\mathscr{O}_v$ but also *rank* and *select* queries. Now, ptrW together with bvPL and bvSD feature the fastest $\mathscr{O}_v$. Yet, we show in the evaluation (§ 7) that the hidden constant factors entail overheads for bit vectors. Simultaneously, their size outperforms that of ptrW for various sparse graphs.

## 4.5 High-Performance Implementation

For high performance, we use the sdsl-lite library [37] 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 applications and families, illustrating that the empirical results follow the theoretical analysis from § 4.4.

## 5 LOGARITHMIZING THE $\mathscr{A}$ STRUCTURE

We now turn our attention to $\mathscr{A}$, the second part of AA; see Figure 2 (❹). As $\mathscr{A}$ is more complex than $\mathscr{O}$, we first develop a formal model for logarithmizing $\mathscr{A}$ and show that various past schemes are merely its special cases. We illustrate that they all entail inherent performance or storage issues and we then design novel schemes to overcome these problems.

## 5.1 A Model for Logarithmizing $\mathscr{A}$

Log(Graph) comes with many logarithmization schemes $C[\mathscr{A}]$ for condensing $\mathscr{A}$ that target various classes of graphs. To facilitate a unified reasoning about them, we can define any such logarithmization scheme to be a tuple $(\mathscr{P}, \mathscr{T})$. $\mathscr{P}$ is the *permuter*: a function that relabels the vertices. We introduce $\mathscr{P}$ to explicitly capture the notion that appropriate labeling of vertices significantly reduces $|\mathscr{A}|$. We have $\mathscr{P} : V \to \mathbb{N}$ such that (the condition enforces the uniqueness of IDs)):

$$\forall_{v,u \in V} \ (v \neq u) \Rightarrow [\mathscr{P}(v) \neq \mathscr{P}(u)] \quad (12)$$

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

$$\mathscr{T}_x : \overbrace{\hat{V} \times ... \times \hat{V}}^{x \text{ times}} \to \{0,1\} \times ... \times \{0,1\} \quad (13)$$

We introduce $\mathscr{T}$ to enable arbitrary operations on sequences of relabeled vertices, for example be the Varint encoding [28].

## 5.2 Understanding Storage Lower Bounds

$\mathscr{A}$ 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 [61] requires ≈1,126 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 assume more about $G$'s structure on top of the number of vertices and edges. We now target *separable* graphs (§ 2.2).

## 5.3 Incorporating Compactness

We use compact graph representations that take $O(n)$ bits to encode graphs. The main technique that ensures compactness that we incorporate 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. [10] (referred to as the *RB* scheme) that requires $O(n)$ bits to store a graph that is *separable* (see § 2.2). 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 [28]. Assuming that the new labels of $v$'s neighbors do not differ significantly, the encoded gaps use less space than the IDs [10]. 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.

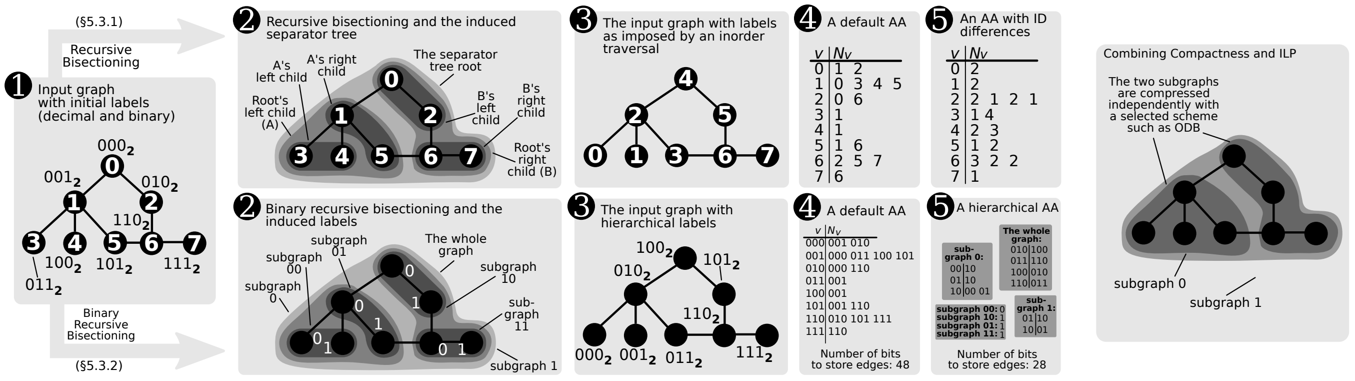


**Figure 4: An example graph representation logarithmized with compact Recursive Bisectioning (RB [10], § 5.3.1), an example of Binary Recursive Bisectioning (BRB, § 5.3.2) and an example of combining compactness and ILP (CMB, § 5.5).**

**Using Permuters and Transformers** One can easily express RB using $\mathscr{P}$ and $\mathscr{T}$. First, $\mathscr{P}$ 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 $\mathscr{T} = \{\mathscr{T}_v(v, N_v)\}$ takes as input $v$ and $N_v$. It then encodes the differences between consecutive vertex labels using Varint. The respective differences are: $|N_{1,v} - N_{0,v}|$, $N_{2,v} - N_{1,v}$, ..., and $N_{d_v,v} - N_{d_v-1,v}$.

**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 $|\mathscr{A}|$. 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 [32] 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 differences of each pair of neighbors (minimum linear arrangement (MLinA)), and the sum of logarithms of differences from MLinA (minimum logarithmic arrangement (MLogA)).

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

$$\min_{\mathscr{P}(v), \forall v \in V} \sum_{v \in V} \sum_{i=0}^{|N_v|-1} |\mathscr{P}(N_{i+1,v}) - \mathscr{P}(N_{i,v})| \tag{14}$$

$$\min_{\mathscr{P}(v), \forall v \in V} \sum_{v \in V} \sum_{u \in N_v} \log |\mathscr{P}(v) - \mathscr{P}(u)| \tag{15}$$

Both functions use the uniqueness Constraint (12).

**Problems** All of the above schemes except MLogGapA were proved to be NP-hard [20] and do not scale with $n$. MLogGapA is still an open problem.

*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.

$$\forall_{v \in V} \forall_{i,j \in \{1..d_v\}} \left[ \mathscr{P}\left(N^I_{j,v}\right) + (x_{vij} - 1) \cdot n \leq N_{i,v} \right] \tag{16}$$

$$\forall_{v \in V} \forall_{i,j \in \{1..d_v\}} \left[ \mathscr{P}\left(N^I_{j,v}\right) + (1 - x_{vij}) \cdot n \geq N_{i,v} \right] \tag{17}$$

$$\forall_{v \in V} \forall_{i \in \{1..d_v\}} \left[ \sum_{j=1}^{d_v} x_{vij} = 1 \right] \tag{18}$$

$$\forall_{v \in V} \forall_{i \in \{1..d_v\}} \left[ N_{i,v} < N_{i+1,v} \right] \tag{19}$$

$N^I_v$ is the initial labeling of $N_v$, $x_{vij}$ is a boolean variable that determines if neighbor $j \in N^I_v$ must be $i$th neighbor in $N_v$ according to relabeling $\mathscr{P}$. If $x_{vij} = 1$, constraints (16), (17) use $N_{i,v} = \mathscr{P}\left(N^I_{j,v}\right)$, otherwise they are trivially satisfied. Constraint (18) selects each neighbor once. Finally, constraint (19) 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 4.

**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 $\mathcal{T}$ 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. $\mathcal{T}$ 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 $\mathscr{A}$) 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, $|\mathscr{A}|$ 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 $\mathscr{A}$ compared to BRB (BRB's hierarchical encoding entails expensive queries). Second, it does not require costly recursive partitioning. Finally, we later (§ 7.4, § 7.5) show that DMd significantly outperforms DMf and matches the compression ratios of the WebGraph library [12].

**Permuter/Transformer** DM's $\mathcal{T}$ 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. Log(Graph) implements the provided model (§ 2.1, § 5.1). The code is available online[2].

**Extensibility** We achieve this by implementing the model from § 2.1, § 5.1. Log(Graph) is divided into four *modules* (well-separated parts of code) that group variants of $\mathcal{O}$, $C[\mathcal{O}]$, $\mathscr{A}$, and $C[\mathscr{A}]$. The $C[\mathscr{A}]$ module further manages submodules for various types of $\mathcal{P}$ and $\mathcal{T}$. This enables us to seamlessly implement, analyze, and compare the described AA variants.

**High Performance** The combinations of the variants of $\mathcal{O}$, $C[\mathcal{O}]$, $\mathscr{A}$, $\mathcal{P}$, and $\mathcal{T}$ give many possible AA designs. For example, $\mathcal{O}$ can be any succinct bit vector. Now, selecting a specific variant takes place in a performance-critical code part 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: the generic template class, the constructor of a representation, and a function for resolving $N_v$. A new representation GraphR only requires defining the offset structure ($\mathcal{O}$), the offset compression structure ($C[\mathcal{O}]$), and the transformer ($\mathcal{T}$) types.

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 $\mathcal{P}$.

```
template<typename 𝒪, typename C[𝒪], typename 𝒯>
class GraphR : public BaseGraphR { // Class template.
  𝒪* offsets; C[𝒪]* compressor; 𝒯* transformer; };

template<typename 𝒪, typename C[𝒪], typename 𝒯> // Constructor.
GraphR<𝒪, C[𝒪], 𝒯>::GraphR(Permutation 𝒫, AA* al) {
  al->permute(𝒫); // Note that 𝒫 is not a type.
  transformer = new 𝒯(); transformer->transform(&al);
  offsets = new 𝒪(al);
  compressor = new C[𝒪](); compressor->compress(&offsets); }

template<typename 𝒪, typename C[𝒪], typename 𝒯>
v_id* GraphR<𝒪, C[𝒪], 𝒯>::getNeighbors(v_id v) { // Resolve N_v.
  v_id offset = offsets->getOffset(v);
  v_id* neighbors = tr->decodeNeighbors(v, offset);
  return neighbors; }
```

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

## 7 EVALUATION

We now illustrate the advantages of Log(Graph).

## 7.1 Evaluation Scope and Methodology

We first describe the evaluation scope and methodology.

**Considered Algorithms** We consider the following algorithms included in the GAP Benchmark Suite [8]: 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.

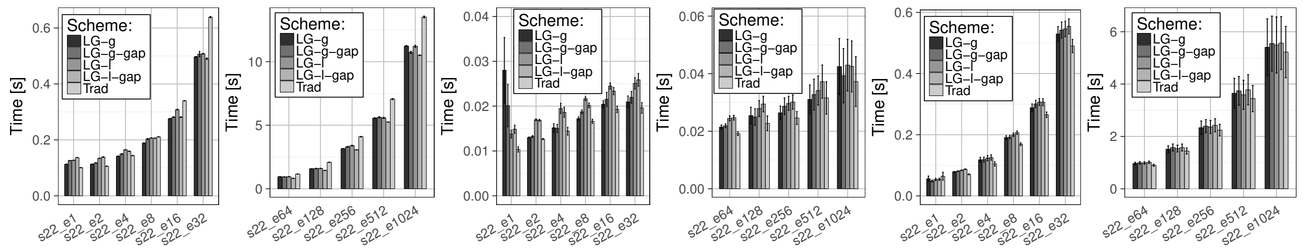
- **BFS**: A state-of-the-art variant with direction-optimization and other enhancements that reduce data transfer [7, 8].
- **SSSP**: An optimized $\Delta$-Stepping algorithm [8, 55, 59].
- **CC**: A variant of the Shiloach-Vishkin scheme [5, 67].
- **BC**: An enhanced Brandes' scheme [8, 13, 56].
- **PR**: A variant without atomic operations [8].
- **TC**: An optimized algorithm that reduces the computational complexity by preprocessing the input graph [21].

**Considered Graphs** We analyze synthetic power-law (the Kronecker model [52]), synthetic uniform (the Erdős-Rényi model [33]), and real-world datasets (including SNAP [53], KONECT [50], DIMACS [30], and WebGraph [12]); 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 small subset; the remainder is in the Appendix or technical report[2].

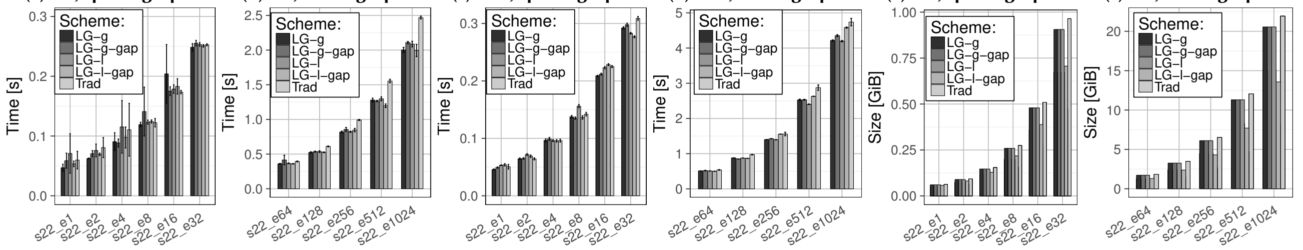
**Experimental Setup and Architectures** We use the following systems to cover various types of machines:

- **CSCS Piz Daint** is a Cray with various XC* nodes. Each XC50 compute node contains a 12-core HT-enabled Intel

[2]To be released after the publication due to double-blindness.



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

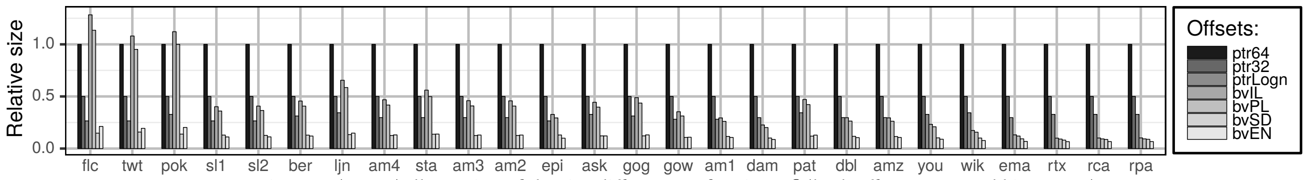


**Figure 6: (§ 7.3.2) Illustration of the size differences of various $\mathscr{O}$ (both offset arrays and bit vectors).**

| 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, am2, am3, 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.**

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 [48]. 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 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 start with logarithmizing fine graph elements.

*7.2.1 Description of Preliminaries.*

**Main Goal** We 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). We predict that the former is because of overheads from bitwise manipulations over the input data. Simultaneously, less pressure on the memory subsystem due to less data transferred to and from the CPU should result in performance improvements.

**Considered Log(Graph) Variants** We consider four variants of Log(Graph): `LG-g` (the global approach), `LG-g-gap` (the global approach with fixed-size gap encoding), `LG-l` (the local approach), and `LG-l-gap` (the local approach with fixed-size gap encoding). We also incorporate the ILP heuristics for relabeling from § 3.6 that enhances the local approach.

**Comparison Targets** We compare Log(Graph) to the traditional adjacency array (`Trad`). We do not aim to outperforms sophisticated compression schemes but rather illustrate a straightforward logarithmization scheme that does not incur additional overheads but still ensures storage reductions and can be used in any graph processing library or engine.

*7.2.2 Key Analyses.*

**Performance and Size** 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 $|\mathscr{A}|$ 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`.

**Distributed-Memories and Scalability** The advantages from Log(Graph) 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 all the studied graphs. We also conducted scalability analyses; the performance pattern is not surprising and all the Log(Graph) variants finish faster as $T$ increases. Full results can be found in the Appendix (§ 10.2.1, § 10.2.2).

*7.2.3 Further Analyses.* We also investigate the impact from ILP. It reduces the size of graphs and we obtain consistent improvements or 1-3%, for example from 0.614 GB to 0.604 GB for the ork graph. Yet, these improvements are significantly lower that those from gap-encoding.

*7.2.4 Key Insights and Answers.* The most important insights are as follows. First, logarithmizing fine elements does reduce storage for graphs while ensuring high-performance and scalability; both on shared- and distributed-memory machines. Second, both ILP (§ 3.6) and fixed-size (§ 3.7) gap encoding reduce $|\mathscr{A}|$, with the latter being a definite winner.

## 7.3 Logarithmizing Offset Structures $\mathscr{O}$

We next proceed to analyze logarithmizing $\mathscr{O}$.

*7.3.1 Description of Preliminaries.*

**Main Goal** We illustrate that logarithmizing $\mathscr{O}$ with succinctness brings large storage reductions over simple bit vectors and offset arrays, without compromising performance.

**Considered Log(Graph) Variants** We investigate all the described $\mathscr{O}$ variants of offset arrays and bit vectors (§ 4.1, § 4.3). We also incorporate a variant where we logarithmize each offset treated as a fine-grained element, as described in § 3.4.

**Investigated Parameters** We vary the parameters related to the design of various types of $\mathscr{O}$: $L$, $B$, and $W$.

**Comparison Targets** We compare the above-mentioned $\mathscr{O}$ designs to the traditional zlib compression.

*7.3.2 Key Analyses.*

**Size: Which Bit Vector is the Smallest?** We first compare the size of all bit vectors for graphs of various sparsities $\bar{d}$; see Figure 7. Static succinct bit vectors consistently use the least space. Interestingly, bvSD uses more space than bvEN for graphs with lower $\bar{d} \leq 15$. This is because the term $\log \binom{\frac{2Wm}{B}}{n}$ grows faster with the number of edges than that of bvEN.

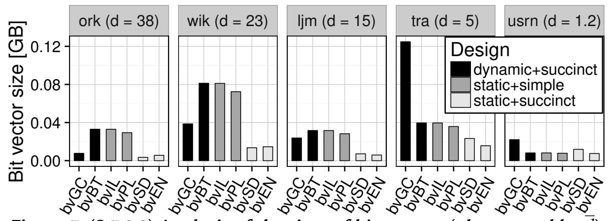
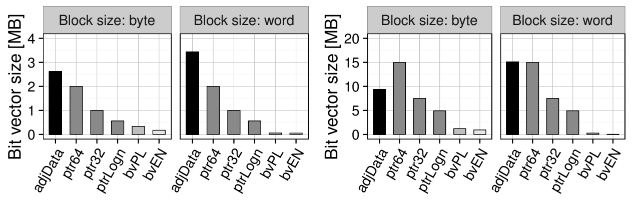


Figure 7: (§ 7.3.2) Analysis of the sizes of bit vectors (plots sorted by $\bar{d}$).

**Size: Bit Vectors or Offset Arrays?** We next compare the size of offset arrays ($W \in \{32, 64, \lceil \log n \rceil\}$) and selected bit vectors in Figure 16. As expected, offset arrays are the largest except for graphs with very high $\bar{d}$. The sparser a graph, the bigger the advantage of bit vectors. Although $|\mathscr{O}|$ grows linearly with $m$ for bit vectors, the rate of growth is low (bit vector take only one bit for a single edge, assuming $B$ is one word size). Again, bvEN is the smallest in most cases.

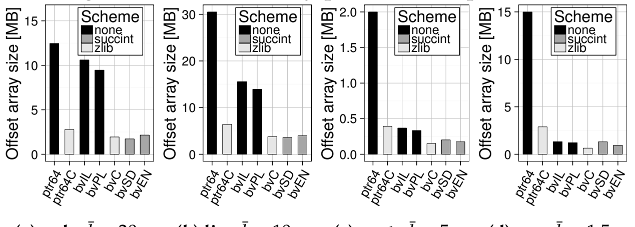
**Size: When To Condense $|\mathscr{O}|$?** For many graphs, $|\mathscr{A}| \gg |\mathscr{O}|$ and condensing $|\mathscr{O}|$ brings only little improvement. Thus, we also investigate when to condense $|\mathscr{O}|$. We analyze the listed graphs and conclude that $|\mathscr{O}| \approx |\mathscr{A}|$ if the graph is sparse enough: $\bar{d} \leq 5$. This is illustrated in Figure 8. In such cases, condensing $\mathscr{O}$ is at least as crucial as $\mathscr{A}$.



(a) Graph am1; $\bar{d} \approx 5$. (b) Graph rca; $\bar{d} \approx 1.5$.

Figure 8: (§ 7.3.2) Illustration of storage required for both $\mathscr{A}$ and $\mathscr{O}$ for graphs with various average degrees $\bar{d}$ when varying the block size $B$.

**Size: Succinctness or Compression?** We finally analyze the effect of traditional compression included in $C[\mathscr{O}]$ on the example of the well-known zlib [31]. We present the results in Figure 9; (64-bit offset arrays represent all pointer schemes that followed similar results). Compressed variants of ptr64 and bvPL are ptr64C and bvPLC. Once more, the results heavily depend on $\bar{d}$. For bit vectors, the sparser the graph, the more compressible $\mathscr{O}$ is; for $\bar{d} < 10$, bvPLC is smaller than both bvSD and bvEN. Interestingly, ptr64C is smaller than ptr64 by an identical ratio regardless of $\bar{d}$. We conclude that zlib offers comparable or slightly (by up to ≈20%) better $|\mathscr{O}|$ reductions than succinct designs. We now proceed to show that these advantages are annihilated by performance penalties.



(a) pok; $\bar{d} \approx 20$. (b) ljn; $\bar{d} \approx 10$. (c) am1; $\bar{d} \approx 5$. (d) rca; $\bar{d} \approx 1.5$.

Figure 9: (§ 7.3.2) $|\mathscr{O}|$ for various $\bar{d}$ when varying the compression scheme.

**Performance** Finally, we analyze the performance of various $\mathscr{O}$ 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 $\bar{d}$) are in Figure **??**. 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 bvIL that has the biggest $|\mathscr{O}|$ (cf. Figure 16); 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 16) 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.

*7.3.3 Further Analyses.* We vary the block size $B$ that controls the granularity of $\mathscr{A}$ to be an 8-bit byte or a 64-bit word; see Figure 8. First, larger $B$ reduces each $|\mathscr{O}|$ (as $|\mathscr{O}|$ is proportional to $B$). Next, $|\mathscr{A}|$ grows with $B$. This phenomenon is similar to the internal fragmentation in memory allocation. Here, each $\mathscr{A}_v$ 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).

*7.3.4 Key Insights and Answers.* We conclude that succinct bit vectors are a good match for $\mathscr{O}$. First, they reduce $|\mathscr{O}|$ 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, both related to input graphs ($\bar{d}$) and to the utilized AA ($B$ and $\mathscr{A}$). They can enhance any system for condensing static or slowly changing graphs that uses $\mathscr{O}$.

## 7.4 Logarithmizing Adjacency Structures $\mathscr{A}$

We also evaluate the logarithmization of $\mathscr{A}$.

*7.4.1 Description of Preliminaries.*

**Main Goal** We investigate the advantages and tradeoffs of the proposed schemes over the comparison targets.

**Considered Log(Graph) Variants** We evaluate the proposed schemes, including BRB (§ 5.3.2), POD (§ 5.4.2), and the combination of these two (§ 5.5); they are implemented using the proposed Log(Graph) library (§ 6).

**Comparison Targets** We consider all the described schemes expressed with permuters and transformers: RB (§ 5.3.1), ODB (§ 5.4.1), and DMd as well as DMf (§ 5.6). We also compare to the traditional adjacency array (Trad), and the state-of-the-art WebGraph (WG) [12] compression system.

*7.4.2 Key Analyses.*

**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 10. Using fewer partitioning levels increases $|\mathscr{A}|$ 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 $|\mathscr{A}|$ 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.

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

**Table 6: (§ 7.4.2) Illustration of preprocessing overheads of RB.**

**DMd: Approaching the Time/Space Sweetspot** Next, we illustrate that DMd significantly reduces $|\mathscr{A}|$, resolves $N_v$ fast, outperforms WG, uses less storage than DMf, and can be generated fast. The size analysis is shown in Figure 11. 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 $\mathscr{A}$ 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 $\mathscr{A}$ (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 12 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 $\mathscr{A}$.

*7.4.3 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 sums of differences between consecutive IDs.

*7.4.4 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.

**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 $\mathscr{O}$ and various schemes for $\mathscr{A}$. Our modular design based on the established model enables quick and easy implementation of $\mathscr{A}$; each variant requires at most 20 lines of code. The results are shown in Figure 13. 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 decompression overheads less severe. Finally, we also study the differences between DMd and RB as well as DMf in more detail in Figure 13b) 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.

**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 $f_v$ due to the complex $\mathscr{O}$ design. DMd, DMf, and RB differ only marginally (1-3%) due to decoding.

## 8 RELATED WORK

We now discuss how Log(Graph) differs from or complements various aspects of graph processing.

## 8.1 Graph Frameworks, Standards, Languages

*Log(Graph) aims to be a tool that enhances and complements any graph processing engine that stores graphs as adjacency arrays.* Examples could be Pregel [57], Galois [49], GraphBLAS [58], CombBLAS [18], PBGL [40], GAPS [8], and Green-Marl [43]. For example, one could use logarithmized vertex IDs and use them within Galois to accelerate graph processing and reduce the pressure on the memory subsystem.

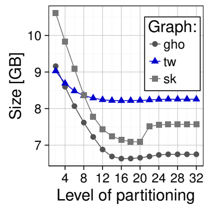


**Figure 10: (§ 7.4.2) The analysis of BRB.**

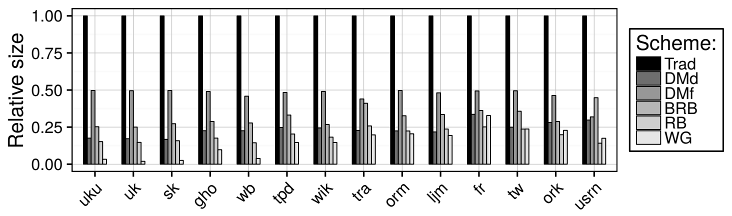


**Figure 11: (§ 7.4.2) Illustration of the storage overhead of different types of $\mathscr{A}$.**

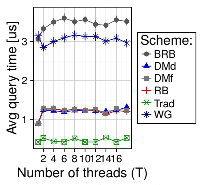
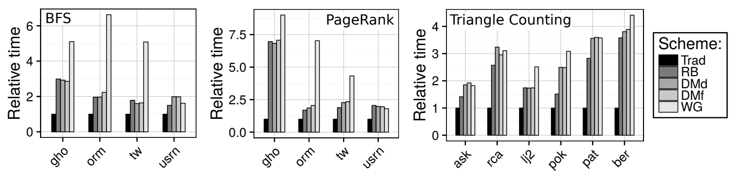


**Figure 12: (§ 7.4.2) The performance of $N_v$ for the orm graph.**


**(a) BFS, PageRank, and Triangle Counting.**

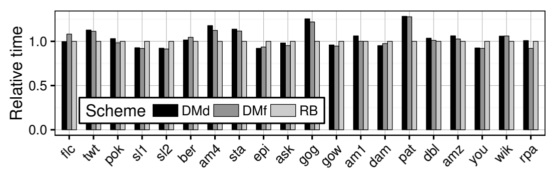

**(b) Comparison of DM and RB (for BFS) on SNAP graphs.**

**Figure 13: (§ 7.4.2) The analysis of the performance impact of various condensing schemes on parallel graph algorithms.**

## 8.2 Reducing Size of Graphs

There exist numerous works on reducing the size of graphs.

**Succinctness and Compactness** *Log(Graph) uses and improves succinct and compact designs to enhance graph storage and processing.* A compact graph representation was proposed by Blandford et al. [10]. It reduces $|\mathscr{A}|$ for several real-world graphs. Yet, its preprocessing is costly. *We alleviate this with our BRB scheme.* Next, various succinct graph representations [6, 11, 47] are all theoretical structures with large hidden constants and no practical implementations or analyses. Third, there exist numerous succinct and compact bit vectors [29, 37, 38, 44, 45, 66, 70]. There are also designs that go below $n + o(n)$ bits with $o(n)$ time queries [24, 64]. Other schemes such as tree representations also exist [46, 60]. Finally, there are several libraries of succinct data structures [2, 27, 34–36, 41]. *Contrarily to our work, none of these designs enhances graph processing and they do not address parallel settings.*

**Graph Compression** A mature compression system for graphs is WebGraph [12]. It compresses $\mathscr{A}$ with: difference encoding, sharing parts of adjacency lists between vertices with similar neighborhoods, and storing consecutive vertex IDs as intervals. Besides WebGraph, there exists a large body of works on compressing webgraphs [3, 4, 14, 15, 17, 22, 23, 25, 42, 51, 62, 65, 68]. Some mention encoding some vertex IDs with the logarithmic number of bits; no details, analyses, or enhancements such as the local approach are provided [3]. Others collapse specified subgraphs into *supervertices* and merge edges between them into a *superedge* [17, 63, 65, 69]. Some of these works aim at generating a summary of input graphs for more efficient graph processing [63, 69]. Finally, several works use ILP to relabel vertices to reduce $|\mathscr{A}|$ [20, 32]. These systems often come with complex compression that requires costly decompression. Now, Log(Graph) embraces these schemes in its model and offers more flexibility. On one hand, it enables simple and generic logarithmization of fine elements that offers inexpensive storage reductions without or with negligible performance overheads. Simultaneously, it comes with more sophisticated schemes that can be used for graphs with more specific properties such as separability.

## 9 CONCLUSION

Reducing graph storage overheads is one of the key challenges for large-scale computations. Yet, as various studies show, traditional schemes such as zlib [31] or WebGraph [12] incur costly performance penalties.

In this work, we illustrate a graph representation called Log(Graph) that unifies various graph compression schemes and applies logarithmic storage lower bounds to (aka "logarithmize"): fine-grained graph elements such as vertex IDs, offset structures, and adjacency structures.

First, logarithmizing fine-grained elements offers simplicity and negligible performance overheads or even speedups from reducing the pressure on the memory subsystem and from incorporating the modern bitwise operations. It can be applied to any graph and can enhance any graph processing engine in shared- and distributed-memory settings. For example, we accelerate SSSP in the GAP Benchmark [8] by ≈20% while reducing the required storage by 20-35%.

Second, for logarithmizing offset or adjacency data we incorporate more sophisticated succinct and compact data structures and techniques such as 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 (>3x for the `orm` graph) while nearly matching its compression ratio with various schemes. We provide a careful and extensible implementation that ensures high performance and will be available online upon publication.

Finally, to the best of our knowledge, our work is the first performance analysis of succinct data structures in a parallel environment. It illustrates surprising differences between succinct bit vectors and offset arrays. We believe that our insights can be used by both theoreticians and practitioners to develop more efficient succinct schemes that would offer higher speedups in parallel settings.

## REFERENCES

[1] DYNAMIC: a succinct and compressed dynamic data structures library. https://github.com/nicolaprezza/DYNAMIC.

[2] Sux - Implementing Succinct Data Structures. available at: http://sux.dsi.unimi.it.

[3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 203–212. IEEE, 2001.

[4] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of web graphs. *Computing and Combinatorics*, pages 1–11, 2008.

[5] 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.

[6] J. Barbay, L. Castelli Aleardi, M. He, and J. Munro. Succinct representation of labeled graphs. In T. Tokuyama, editor, *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 316–328. Springer Berlin Heidelberg, 2007.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[8] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[9] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. SlimSell: A Vectorized Graph Representation for Breadth-First Search. In *Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*. IEEE, May 2017.

[10] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 679–688, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[11] 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.

[12] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *World Wide Web Conf. (WWW)*, pages 595–601, 2004.

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

[14] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *SPIRE*, volume 9, pages 18–30. Springer, 2009.

[15] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[16] A. Brodnik and J. I. Munro. Membership in Constant Time and Almost-Minimum Space. *SIAM J. Comput.*, 28(5):1627–1640, May 1999.

[17] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM, 2008.

[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] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *In Proc. 15th SPIRE, LNCS 5280*, pages 176–187, 2008.

[25] F. Claude and G. Navarro. Extended compact web graph representations. *Algorithms and Applications*, 6060:77–91, 2010.

[26] I. I. CPLEX. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[27] Daisuke Okanohara. rsdic - Compressed Rank Select Dictionary. available at: http://code.google.com/p/rsdic.

[28] 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.

[29] E. Demaine. Advanced Data Structures, 2012. Lecture Notes.

[30] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Math. Soc., 2009.

[31] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification Version 3.3, 1996.

[32] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.

[33] P. Erdős and A. Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi*, 2:482–525, 1976.

[34] Francisco Claude. libcds. https://github.com/fclaude/libcds.

[35] Giuseppe Ottaviano. Succinct library. https://github.com/ot/succinct.

[36] S. Gog, T. Beller, A. Moffat, and M. Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[37] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

[38] R. González, S. Grabowski, V. Mäkinen, 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.

[39] T. Granlund. Instruction latencies and throughput for AMD and Intel x86 Processors, 2012.

[40] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[41] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Experimental Algorithms*, pages 5–17. Springer, 2013.

[42] C. Hernández 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.

[43] 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.

[44] 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.

[45] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1988. AAI8918056.

[46] 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.

[47] S. Kannan, M. Naor, and S. Rudich. Implicit Representation of Graphs. In *SIAM Journal On Discrete Mathematics*, pages 334–343, 1992.

[48] 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.

[49] 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.

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

[51] S. Ladra. Algorithms and compressed data structures for information retrieval. 2011.

[52] 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.

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

[54] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 17(1):5–20, 2007.

[55] 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.

[56] 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.

[57] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Manag. of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[58] 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.

[59] U. Meyer and P. Sanders. $\delta$-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[60] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, Mar. 2002.

[61] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[62] G. Navarro. Compressing web graphs like texts. Technical report, Technical Report TR/DCC-2007-2, Dept. of Computer Science, University of Chile, 2007.

[63] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 419–432. ACM, 2008.

[64] D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of ALENEX'07, ACM*. Press, 2007.

[65] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 405–416. IEEE, 2003.

[66] 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.

[67] Y. Shiloach and U. Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[68] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 213–222. IEEE,

2001.
[69] 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.
[70] S. Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms*, pages 154–168. Springer, 2008.
[71] D. B. West. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.
[72] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

## 10 APPENDIX

### 10.1 Theory: Additional Analyses

Here, we first provide the derivation of the expressions for $\mathcal{O}$ 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 $\left(1 - \frac{1}{\log n}\right)$ probability [9]: $\hat{d} \leq \left(\frac{\alpha n \log n}{\beta - 1}\right)^{\frac{1}{\beta-1}}$. We now aim to derive an expression for $m$ as a function of $\alpha$ and $\beta$. Using the degree distribution we have

$$m = \frac{1}{2}\sum_{x=1}^{\hat{d}} x f(x) = \frac{1}{2}\sum_{x=1}^{\hat{d}} \alpha x^{1-\beta} \tag{20}$$

This can be approximated with an integral

$$m \approx \frac{1}{2}\int_{x=1}^{\hat{d}} \alpha x^{1-\beta} dx = \frac{\alpha}{2}\frac{1}{(2-\beta)}\left(\hat{d}^{2-\beta} - 1\right) \tag{21}$$

Plugging this into the storage expression, we obtain

$$E[|\mathcal{A}|] \approx \frac{\alpha}{2-\beta}\left(\left(\frac{\alpha n \log n}{\beta - 1}\right)^{\frac{2-\beta}{\beta-1}} - 1\right)\left(\lceil \log n \rceil + \left\lceil \log \widehat{\mathcal{W}} \right\rceil\right) \tag{22}$$

$$E[|\mathcal{O}|] \approx n\left\lceil \log\left(\frac{\alpha}{2-\beta}\left(\left(\frac{\alpha n \log n}{\beta - 1}\right)^{\frac{2-\beta}{\beta-1}} - 1\right)\right)\right\rceil \tag{23}$$

### 10.2 Logarithmizing Fine Elements: Additional Analyses

We now analyze in more detail how logarithmizing fine elements impacts aspects such as scalability or communicated data.

*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 14.

*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. The results are illustrated in Figure 15.

### 10.3 Logarithmizing $\mathcal{O}$: Additional Analyses

In the main body of the work, we have only analyzed the influence of $\mathcal{O}$'s properties on $|\mathcal{O}|$. Yet, the scope of the interplay between AA's parameters is much broader: $|\mathcal{O}|$ 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 $\mathcal{O}$ 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 complex dependencies; for example bvSD's length also gets smaller but as the ratio of ones to zeros gets higher, $|\mathcal{O}|$ may as well increase; a similar argument applies to bvEN. We now analyze these effects by comparing the original $|\mathcal{O}|$ to $|\mathcal{O}|$ after relabeling of vertices according to the inorder traversal of the separator tree as performed in the Blandford's scheme; see Figure 16. 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 ≈25% and ≈12% smaller). This is because the relabelled $\mathcal{A}$ uses less storage, requiring shorter bit vectors.

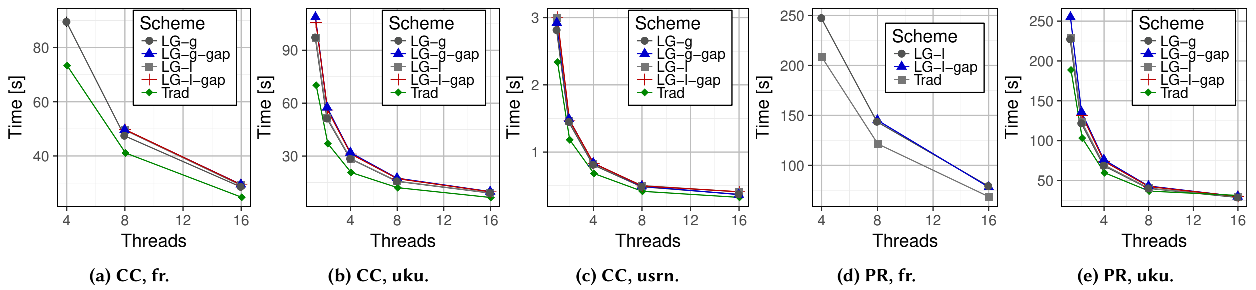
### 10.4 Logarithmizing $\mathcal{A}$: Additional Analyses

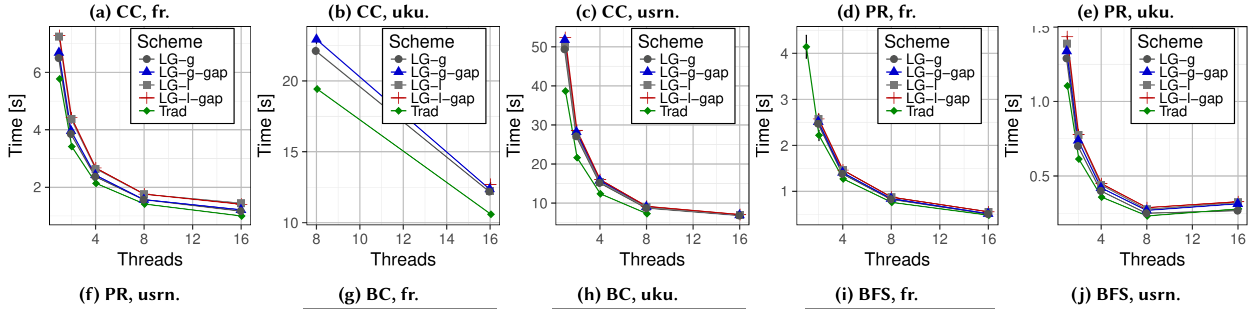
We also illustrate more analyses related to logarithmizing $\mathcal{A}$.

*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 [71]. 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 17. We ensure that the respective partitions are of almost equal sizes. We did not find strong correlation between cut sizes and sparsities $\overline{d}$. 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 [72], 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* [10] for mapping between such vertex clones (i.e., *shadow vertex trees* [10]). We calculated the number of shadow pointers in such structures and the resulting storage overheads in $|\mathcal{A}|$ based on VCs. All the graphs follow similar trends. For example, twt requires 3.53M additional pointers, giving 4.81MB for $|\mathcal{A}|$ (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


(a) CC, fr. (b) CC, uku. (c) CC, usrn. (d) PR, fr. (e) PR, uku.


(f) PR, usrn. (g) BC, fr. (h) BC, uku. (i) BFS, fr. (j) BFS, usrn.

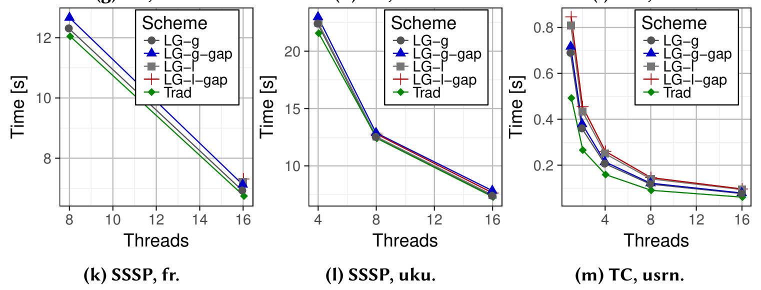

(k) SSSP, fr. (l) SSSP, uku. (m) TC, usrn.

**Figure 14: (§ 10.2.1) The scalability analysis of Log(Graph), real-world graphs.**

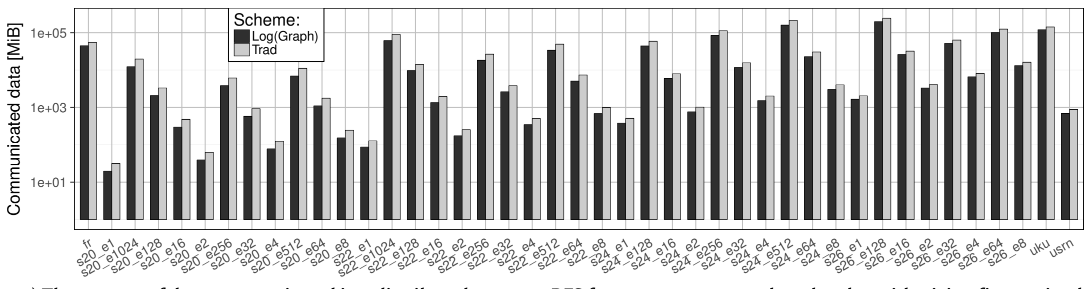


**Figure 15: (§ 10.2.2) The amount of data communicated in a distributed-memory BFS for 1,024 compute nodes when logarithmizing fine-grained graph elements.**

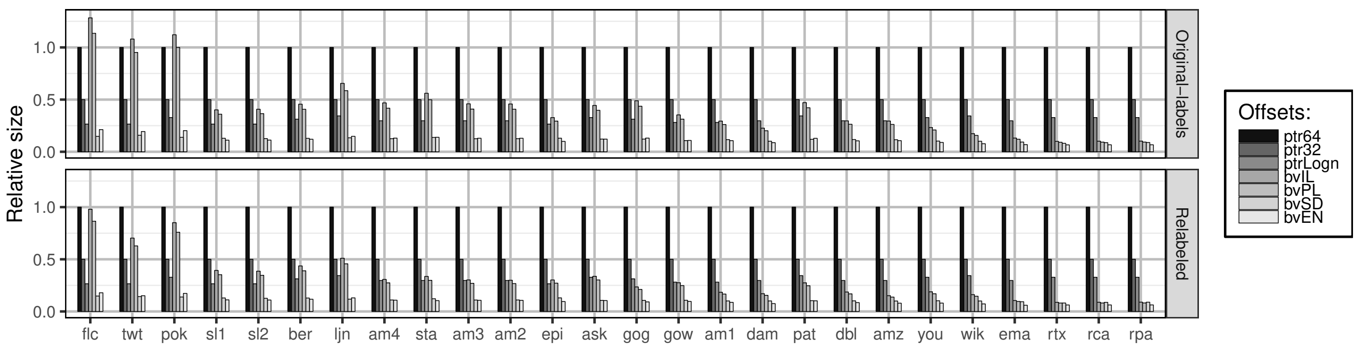
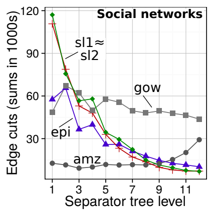
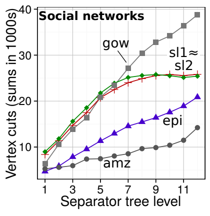


**Figure 16: (§ 10.3) Illustration of the size of various $\mathscr{O}$ with and without relabeling vertex IDs (with a traditional adjacency data structure $\mathscr{A}$).**

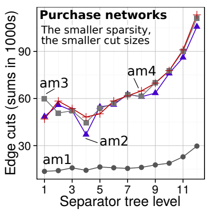
partition sizes be at most $\mathcal{D}$. We first analyze how changing $\mathcal{D}$ influences sizes of ECs and VCs for each level of separator trees. We plot the findings for representative graphs in Figure 18. As expected, cuts become smaller with growing $\mathcal{D}$: more imbalance more often 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 ≈2%; other cases follow similar



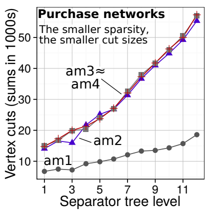
**(a) Edge cuts.**



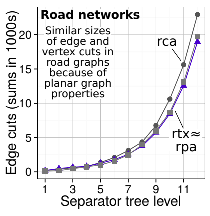
**(b) Vertex cuts.**



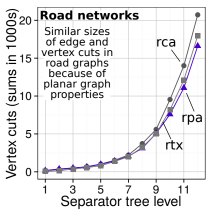
**(c) Edge cuts.**



**(d) Vertex cuts.**

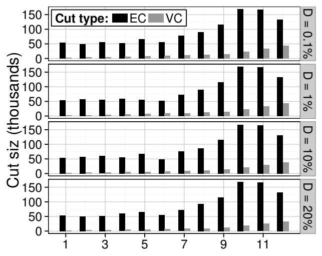


**(e) Edge cuts.**

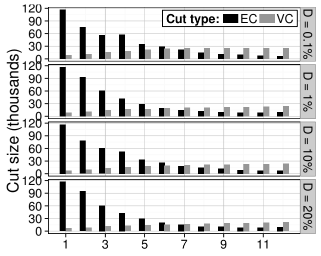


**(f) Vertex cuts.**

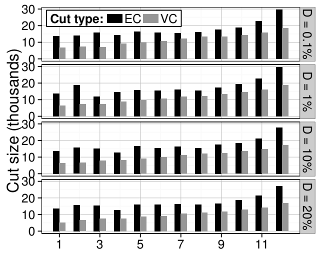
**Figure 17: (§ 10.4.1) An illustration of sums of edge and vertex cuts at various levels of separator trees in different types of graphs.**



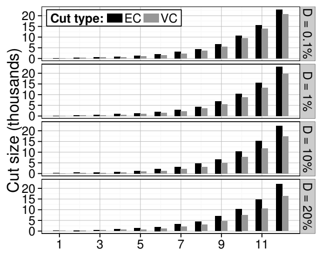
**(a) Graph twt; $\bar{d} \approx 20$.**



**(b) Graph sl2; $\bar{d} \approx 10$.**

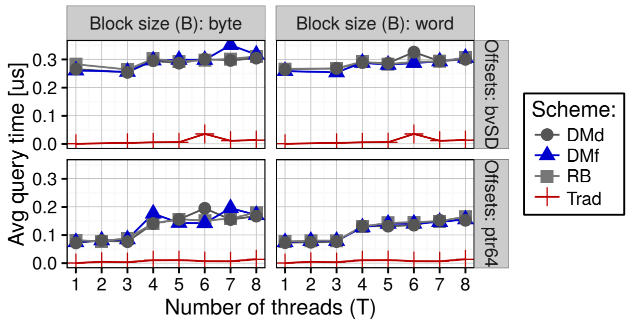


**(c) Graph am1; $\bar{d} \approx 5$.**

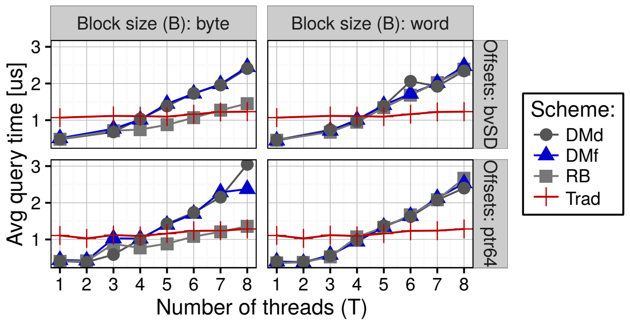


**(d) Graph rca; $\bar{d} \approx 1.5$.**

**Figure 18: (§ 10.4.2) Illustration of vertex/edge cut sizes when varying the balancedness ratio $\mathcal{D}$ and levels of respective separator trees.**



**(a) The analysis of obtaining $d_v$; pok graph.**



**(b) The analysis of obtaining $N_v$; pok graph.**

**Figure 19: (§ 10.4.4) Performance analysis of accessing a graph in a parallel setting.**

patterns. They do not impact the final $|\mathscr{A}|$, resulting in minor (≈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 [26] to solve the ILP problems formulated in § 5.4.1 and § 5.4.2. We use two graphs g1 and g2, both consisting of two communities with few (<0.2$m$) edges in-between. Here, we illustrate (Figure 20) 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. (14) and y=2 for Eq. (15)) 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.

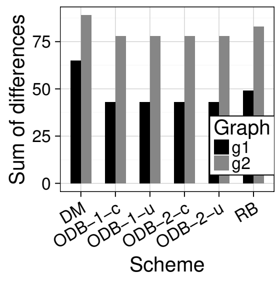


**Figure 20: (§ 10.4.3) Analysis of POD/CMB.**