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

Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh,
Maurice Hoerold, Torsten Hoefler

Presentation by: Eric Wang
What is Log(Graph)?

A **Near-Optimal** **High Performance** *Graph Representation*

**Near-Optimal:** Graph encoding approaches storage lower bounds
**High Performance:** Enables fast operations/algorithms on graphs
*Graph Representation:* Technique to store graph in computer memory

Implemented as a modular C++ library
Why do we need Log(Graph)?

1. Modern graphs are huge
2. Traditional graph representations are inefficient or waste space
3. Traditional compression is slow

Smaller Graph Representation:
- Enables better performance
- Consumes fewer hardware resources
Understanding Lower Storage Bounds

If we have a set of $S$ elements, how many bits do we need to store any given element?
Understanding Lower Storage Bounds

If we have a set of $S$ elements, how many bits do we need to store any given element?

$\lceil \log(S) \rceil$
Understanding Lower Storage Bounds

If we have a set of $S$ elements, how many bits do we need to store any given element?

$\lceil \log(S) \rceil$

If we have a $n$ vertices in a graph, how many bits do we need to store any given vertex?
Understanding Lower Storage Bounds

If we have a set of $S$ elements, how many bits do we need to store any given element?

$\lceil \log(S) \rceil$

If we have $n$ vertices in a graph, how many bits do we need to store any given vertex?

$\lceil \log(n) \rceil$
We need ⌈log(n)⌉ bits to store a vertex if there are n vertices.

Let’s say in our graph we have n = 1024, so our vertices are 0, 1, 2, ... 1023.

We need ⌈log(1024)⌉ = 10 bits to store a given element.

However, a memory word can be 32 or 64 bits! Meaning that we are wasting a lot of space potentially if we store these vertices many times.
Understanding Lower Storage Bounds

If we have a graph with $n$ nodes and $m$ edges, what is the theoretical storage lower bound?

$$\log \left( \binom{n}{2} \right)$$
Applying Lower Storage Bounds

Let’s say \( n = 2^{40} = \sim 1.09 \) trillion vertices

We have our adjacency array:

0 | 2 3 5 7 11 ... 97
1 | ...
...

**Idea:** Use 7 bits for 0’s neighborhood, saving 25 * 33 = 825 bits
Applying Lower Storage Bounds

Our adjacency array:

<table>
<thead>
<tr>
<th></th>
<th>2 3 5 7 11 ... 97 2^30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**Idea:** Relabel the vertex with ID $2^{30}$ to a smaller ID so we can use < 30 bits
Heuristic Examples

Our adjacency array:

0   |   2   3   5   7   11   …   97   2^30
1   |   …
…

1. Assign vertices that appear often smaller vertex IDs to leverage local storage bounds
2. Use ILP to minimize the maximum vertex IDs of neighborhoods
Technical Definitions

- Log(Graph) structure utilizes unique vertex IDs, an adjacency array (edgeArray), and an offset array (vertexArray)
- A neighborhood is an adjacency array for a single vertex
- A permuter is a function that relabels vertex IDs
- A transformer is a function that maps vertex IDs to bits, modifies AA
- A data structure is compact if it uses $O(OPT)$ bits and succinct if it uses $OPT + o(OPT)$ where OPT is the optimal # of bits
## Technical Definitions

<table>
<thead>
<tr>
<th>Graph model</th>
<th>( G )</th>
<th>A graph ( G = (V, E) ); ( V ) and ( E ) are sets of vertices and edges.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( n, m )</td>
<td>Numbers of vertices and edges in ( G ); (</td>
</tr>
<tr>
<td></td>
<td>( W_{(u,v)} ), ( D )</td>
<td>The weight of an edge ( W_{(u,v)} ) and the diameter of ( G ).</td>
</tr>
<tr>
<td></td>
<td>( d_u, N_u, N_{i,v} )</td>
<td>Degree and neighbors and ( i )-th neighbor of a vertex ( v ); ( N_{0,v} \equiv v ).</td>
</tr>
<tr>
<td></td>
<td>( \bar{x}, \hat{x} )</td>
<td>The average and the maximum among ( x ).</td>
</tr>
<tr>
<td></td>
<td>( \alpha, \beta; p )</td>
<td>Parameters of a power-law graph and an Erdős-Rényi graph.</td>
</tr>
</tbody>
</table>

| 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. |
|              | \( T_{x}, P, W \) | The number of threads/processes and the memory word size. |
|              | \( T_x \) | Time to do a given operation \( x \). |

| Adjacency array | \( \mathcal{A}, \mathcal{A}_v \) | The adjacency array of a given graph and a given vertex. |
|                | \( \Theta, \Theta_v \) | The offset structure of a given graph and an offset to \( \mathcal{A}_v \). |
|                | \( |\mathcal{A}|, |\Theta| \) | The sizes of \( \mathcal{A}, \Theta \). |
|                | \( C[\mathcal{A}], C[\Theta] \) | Compression schemes acting upon \( \mathcal{A}, \Theta \). |
|                | \( B, L \) | Various parameters of \( \mathcal{A} \) and \( \Theta \); see § 4.3 for details. |

| Schemes for \( \mathcal{A} \) | \( \mathcal{P} \) | Permuter: function that relabels vertices. |
|                            | \( \mathcal{T}_x, \mathcal{T} \) | Transformers: functions that arbitrarily modify \( \mathcal{A} \). |
|                            | \( G_x \) | Subgraphs of \( G \) constructed in recursive partitioning. |

**Table 1:** Symbols used in the paper.
Log(Graph) Overview

Organized into three main components/modules:
1. Logarithmize **Fine Elements**
2. Logarithmize **Offset Structure**
3. Logarithmize **Adjacency Structure**

Each component can take on numerous variants and be combined with other components to form many possible Log(Graph) implementations

<table>
<thead>
<tr>
<th>$Θ$</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer array</td>
<td>ptrW</td>
</tr>
<tr>
<td>Plain [36]</td>
<td>bvPL</td>
</tr>
<tr>
<td>Interleaved [36]</td>
<td>bvIL</td>
</tr>
<tr>
<td>Entropy based [24, 66]</td>
<td>bvEN</td>
</tr>
<tr>
<td>Sparse [64]</td>
<td>bvSD</td>
</tr>
<tr>
<td>B-tree based [1]</td>
<td>bvBT</td>
</tr>
<tr>
<td>Gap-compressed [1]</td>
<td>bvGC</td>
</tr>
</tbody>
</table>

Table 4: (§ 4.3) Theore
Figure 2: (§ 2.4) The roadmap of incorporated schemes. The green areas indicate analyzes and themes shared by multiple logarithmization areas.
Log(Graph) Implementation

```cpp
1 template<typename θ, typename C[θ], typename Φ> 
2 class GraphR : public BaseGraphR { // Class template.
3   θ* offsets; C[θ]* compressor; Φ* transformer; 
4 
5 template<typename θ, typename C[θ], typename Φ> // Constructor.
6 GraphR<θ, C[θ], Φ>::GraphR(Permutation Ψ, AA* al) {
7   al->permute(Ψ); // Note that Ψ is not a type.
8   transformer = new Φ(); transformer->transform(&al);
9   offsets = new θ(al);
10  compressor = new C[θ](); compressor->compress(&offsets); }
11
12 template<typename θ, typename C[θ], typename Φ> 
13 v_id* GraphR<θ, C[θ], Φ>::getNeighbors(v_id v) { // Resolve N_v.
14   v_id offset = offsets->getOffset(v);
15   v_id* neighbors = tr->decodeNeighbors(v, offset);
16   return neighbors; }
```

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

Implemented as C++ Library - templates are used for performance reasons and to control complexity
The `bextr` operation consumes 2 CPU cycles and extracts a contiguous sequence of bits. For each neighborhood, we simply store the bit length next to the offset.
Logarithmize Fine Elements

Fine elements are vertices and edges
We can apply storage lower bounds to both

For vertex IDs, we can apply storage lower bounds globally based on $n$ or locally based on the largest vertex in a neighborhood

For edges, we apply storage lower bounds globally or locally based on maximal edge weight

Vertex Id Example:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>...</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Idea: Use 7 bits for 0’s neighborhood

$$|\mathcal{E}| = \sum_{v \in V} \left( d_v \left[ \log \hat{N}_v \right] + \left\lfloor \log \log \hat{N}_v \right\rfloor \right)$$
Logarithmize Fine Elements Strategy #1

Incorporate ILP

Use ILP to reduce maximal IDs in as many neighborhoods as possible - maximal IDs are weighted based on inverse of neighborhood size

\[
\min \sum_{v \in V} \hat{N}_v \cdot \frac{1}{d_v}
\]
Logarithmize Fine Elements Strategy #2

Incorporate Fixed-Size Gap Encoding

**AA Structure:** \([a \ (b - a) \ (c - b)]\)

Maximum difference within a given domain determines number of bits used to encode - we can aim to minimize differences if the numbers themselves are very large but close in value.
Logarithmize Fine Elements Strategy #3

Greedy Vertex Labeling

Sort vertices in non-decreasing order of their degrees - then, traverse the vertices in sorted order and assign smallest ID possible to vertex and neighborhood.

Used as a heuristic for ILP due to ILP being NP-hard.
Logarithmize Offset Array - Bit Vector

Use A Bit Vector

Idea: Instead of storing the offsets in an array, we can use bit vectors to represent

If arr[i] == 1 and this is the jth set bit, then the neighborhood for vertex j starts at the ith block of AA
Logarithmize Offset Array - Bit Vector

But ...

Using this bit vector can potentially be very slow if we have to iterate over it linearly to calculate

We can use an additional $O(n)$ space in order to significantly speed up query operations on this bit vector, so the bit vector structure remains succinct
Succinct Bit Vector Example

Uses $o(n)$ additional bookkeeping space to enable efficient select($x$) and rank($x$) queries
Techniques on Separable Graphs

A graph is separable if we can divide a graph into two sets of vertices so that the size of the cut separating the vertices is much smaller than $|V|$.

The two techniques we will examine are Recursive Bisectioning and Binary Recursive Bisectioning.
**Recursive Bisectioning**: Relabel vertices to minimize differences between labels of consecutive neighbors

1. Bisect recursively on vertices/edges
2. Perform inorder traversal on resulting binary separator tree
3. Label vertices IDs with increasing values
Logarithmize Adjacency Array Strategy #2

**Binary Recursive Bisectioning:** When bisecting recursively, label subgraphs with 0 or 1 appended to existing prefix - clusters will have large common prefixes.

End up with a **hierarchical AA** that incurs less overhead than Recursive Bisectioning.
RB vs BRB Comparison

1. Input graph with initial labels (decimal and binary)

2. Recursive bisectioning and the induced separator tree

3. The input graph with labels as imposed by an inorder traversal

4. A default AA

5. An AA with ID differences

6. Binary recursive bisectioning and the induced labels

7. The input graph with hierarchical labels

8. A default AA

9. A hierarchical AA

Number of bits to store edges: 48

Number of bits to store edges: 28
Distributed Setting

We assume **hierarchical machines** where computation is distributed among them.

We can divide a vertex ID into an **intra** part that is unique within a machine and an **inter** part that encodes the vertex in the distributed-memory structure.

The "**intra-node**" vertex label thus takes \([\log \frac{n}{H}]\) bits.

|\mathcal{A}| = n \left\lceil \log \frac{n}{H_{node}} \right\rceil + H_{node} \left\lceil \log H_{node} \right\rceil

The "**inter-node**" vertex label is unique for a whole node and it takes \([\log H]\) bits.

|\mathcal{A}| = n \left\lceil \log \frac{n}{H_N} \right\rceil + \sum_{j=2}^{N-1} H_j \left\lceil \log H_j \right\rceil
Figure 1: (§ 1, § 7.2) The performance of Log(Graph) with the Single Source Shortest Path algorithm when logarithmizing vertex IDs.
Evaluation Strategy

Examined algorithms in the **GAP benchmark suite** such as BFS, PageRank, SSSP, Betweenness Centrality, Connected Components, and Triangle Counting.

Compared Log(Graph) against **Zlib** (a traditional compression scheme), **Webgraph Library**, and other forms of **Recursive Partitioning**.
Key Findings

- Logarithmizing fine elements reduces storage while ensuring high-performance.
- Logarithmizing the offset array with succinct bit vectors reduces the size of the offset array while matching performance for higher thread counts.
- Logarithmizing the adjacency array with DMd (degree-minimizing with differences encoded) offers a strong space/performance tradeoff as it trades a small amount of storage for faster access but is still very small.
- If we have frequent accesses to neighbors, use RB - if instead we have a large or constantly evolving graph, use BRB.
- Overall, felt that Log(Graph) was a pretty cool paper
- Unfortunate that the C++ implementation has still not been released yet
- Paper overall does a good job of explaining concepts
- However, doesn’t explain how Log(Graph) handles a graph that evolves quickly
- Possible directions for future work might be exploring how different component variants work with each other and if certain variants are specialized for certain graph types/properties

Any Questions?