# 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)?

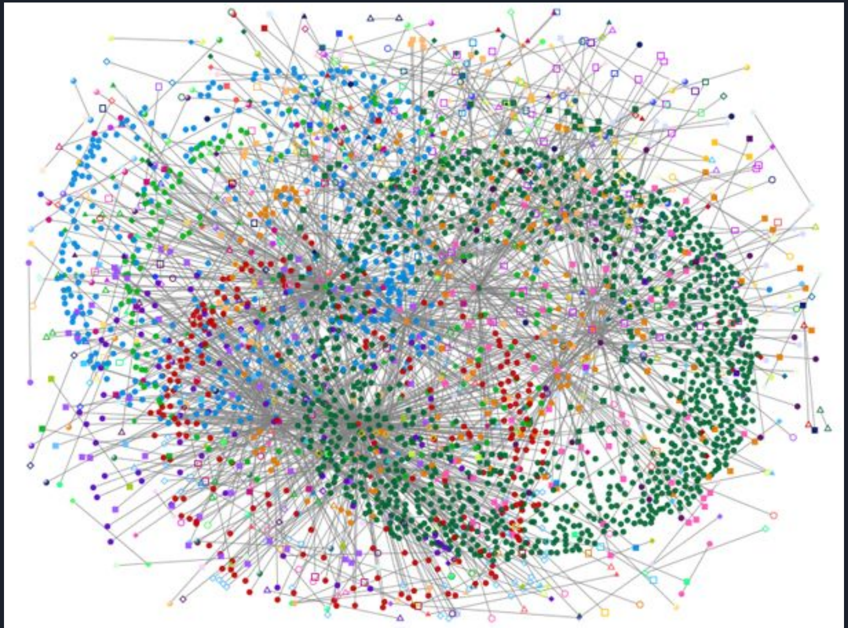
A **Near-Optimal** <u>High Performance</u> *Graph Representation*

**Near-Optimal:** Graph encoding approaches storage lower bounds
<u>High Performance:</u> 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 a n vertices in a graph, how many bits do we need to store any given vertex?

$\lceil \log(n) \rceil$

# Understanding Lower Storage Bounds

We need $\lceil \log(n) \rceil$ 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 $\lceil \log(1024) \rceil = 10$ 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?

$$\log \binom{\binom{n}{2}}{m}$$

# Applying Lower Storage Bounds

Let's say n = 2^40 = ~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:

0 | 2 3 5 7 11 … 97 2^30

1 | …

…

**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

| | | |
|---|---|---|
| Graph model | $G$ | A graph $G = (V, E)$; $V$ and $E$ are sets of vertices and edges. |
| | $n, m$ | Numbers of vertices and edges in $G$; $\lvert V\rvert = n, \lvert E\rvert = m$. |
| | $\mathcal{W}_{(v,w)}, D$ | 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 $\mathscr{A}_v$. |
| | $\lvert\mathscr{A}\rvert, \lvert\mathscr{O}\rvert$ | 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.**

# 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

| $\mathcal{O}$ | ID |
|---|---|
| Pointer array | ptr$W$ |
| Plain [36] | bvPL |
| Interleaved [36] | bvIL |
| Entropy based [24, 66] | bvEN |
| Sparse [64] | bvSD |
| B-tree based [1] | bvBT |
| Gap-compressed [1] | bvGC |

**Table 4: (§ 4.3) Theore**

# Log(Graph) Overview

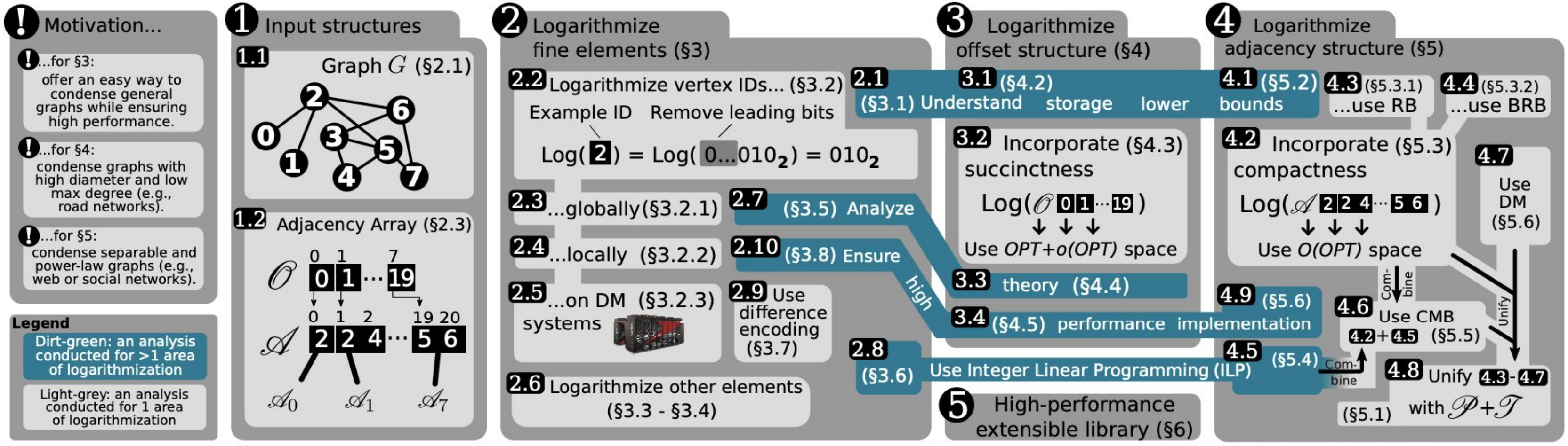



Figure 2: (§ 2.4) The roadmap of incorporated schemes. The green areas indicate analyzes and themes shared by multiple logarithmization areas.

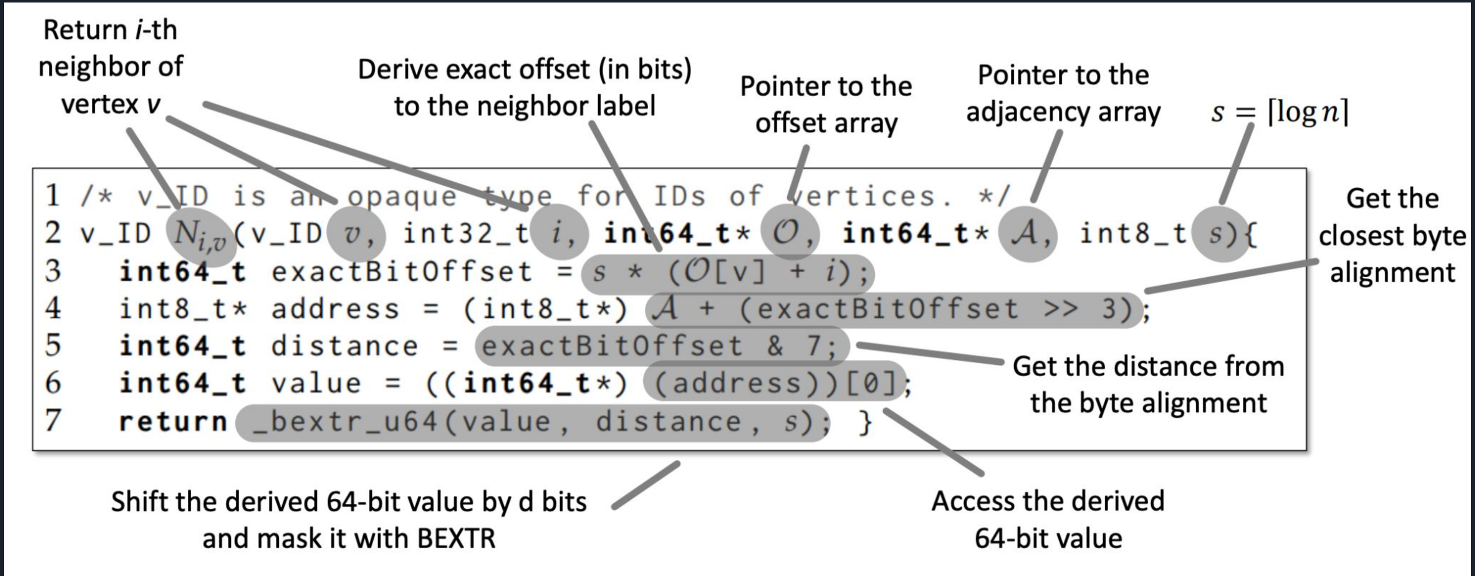
# Log(Graph) Implementation

```
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 Nv.
  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.**

Implemented as C++ Library - **templates** are used for performance reasons and to control complexity

# Accessing Values




**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 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:

0 | 2 3 5 7 11 … 97
1 | …
…

**Idea:** Use 7 bits for 0's neighborhood

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

# 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

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

# 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

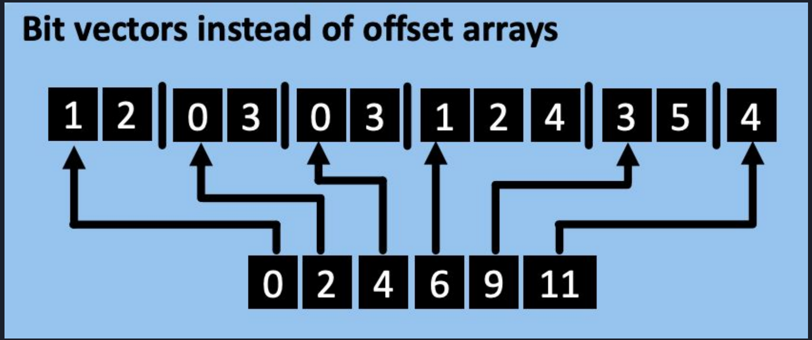
Used as a heuristic for ILP due to ILP being NP-hard

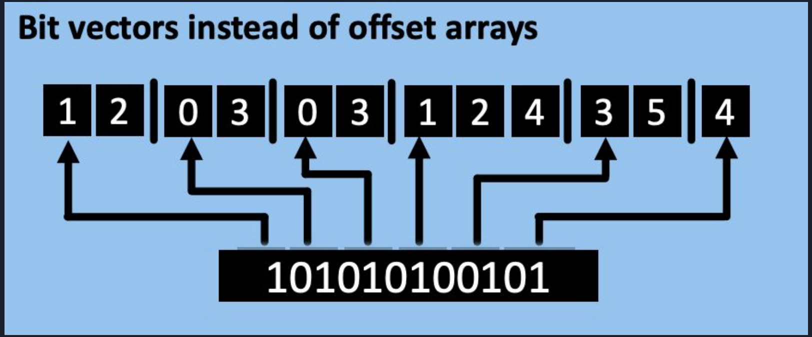
# Logarithmize Offset Array - Bit Vector

**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**



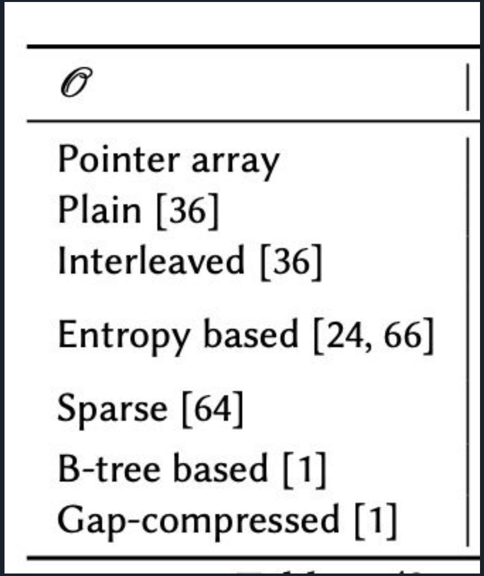



# Logarithmize Offset Array - Bit Vector

**But ...**

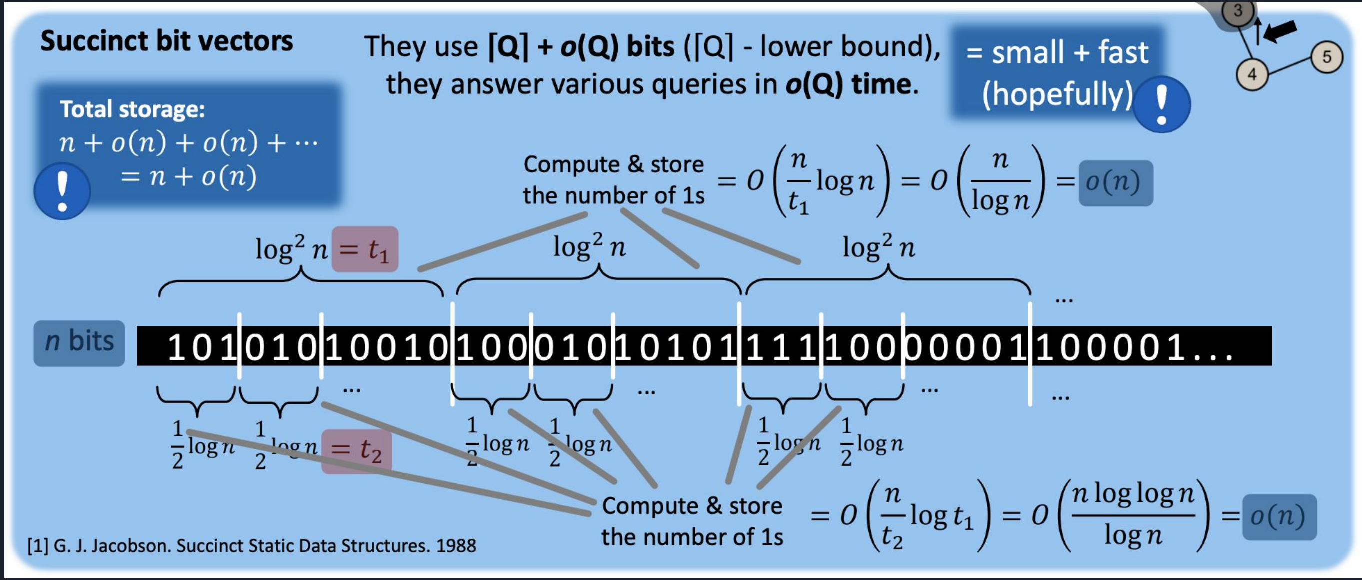
Using this bit vector can potentially be **very slow** if we have to iterate over it linearly to calculate

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**



| $\mathcal{O}$ |
|---|
| Pointer array |
| Plain [36] |
| Interleaved [36] |
| Entropy based [24, 66] |
| Sparse [64] |
| B-tree based [1] |
| Gap-compressed [1] |

# Succinct Bit Vector Example




**Uses o(n) additional bookkeeping space to enable efficient select(x) and rank(x) queries**

# Logarithmize Adjacency Array

**Techniques on Separable Graphs**

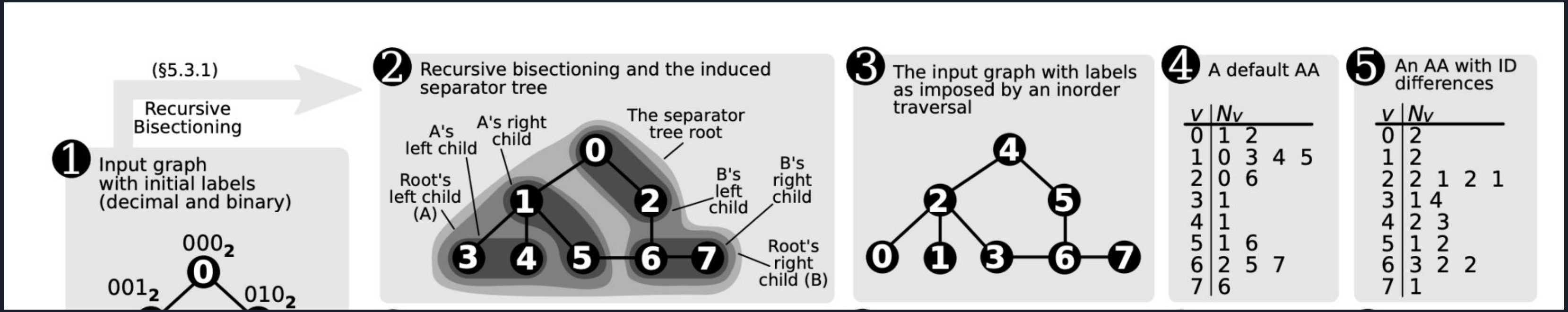
A graph is **separable** if we can divide a graph into two sets of vertices so that the size of the cut separating the vertices is much smaller than $|V|$

The two techniques we will examine are **Recursive Bisectioning** and **Binary Recursive Bisectioning**

# Logarithmize Adjacency Array Strategy #1

**Recursive Bisectioning**: Relabel vertices to minimize differences between labels of consecutive neighbors

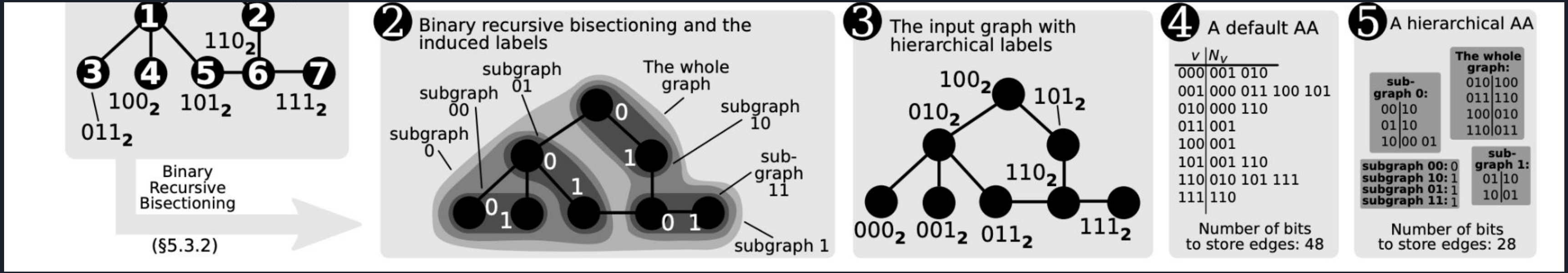
1. Bisect recursively on vertices/edges
2. Perform inorder traversal on resulting binary separator tree
3. Label vertices IDs with increasing values



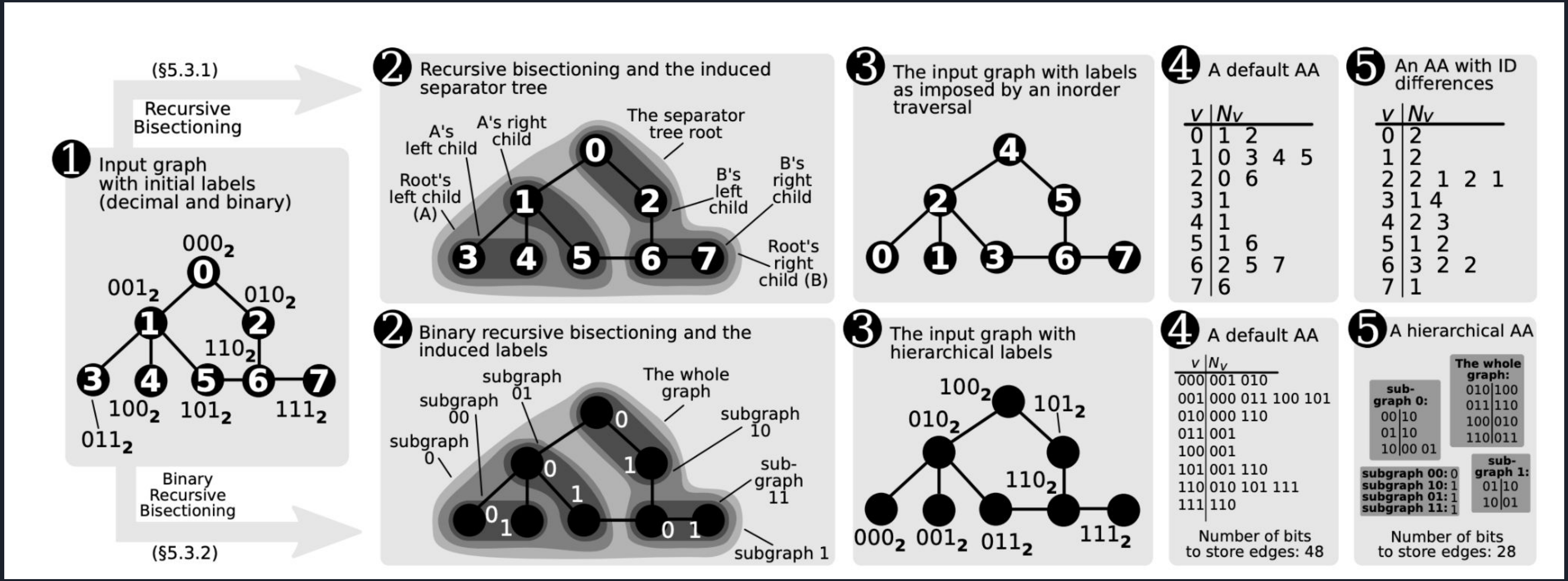
# Logarithmize Adjacency Array Strategy #2

**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



# 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 [bits]: $\left\lceil \log \frac{n}{H} \right\rceil$

The „**inter-node**” vertex label is unique for a whole node and it takes [bits]: $\lceil \log H \rceil$

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

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

# Evaluation Example

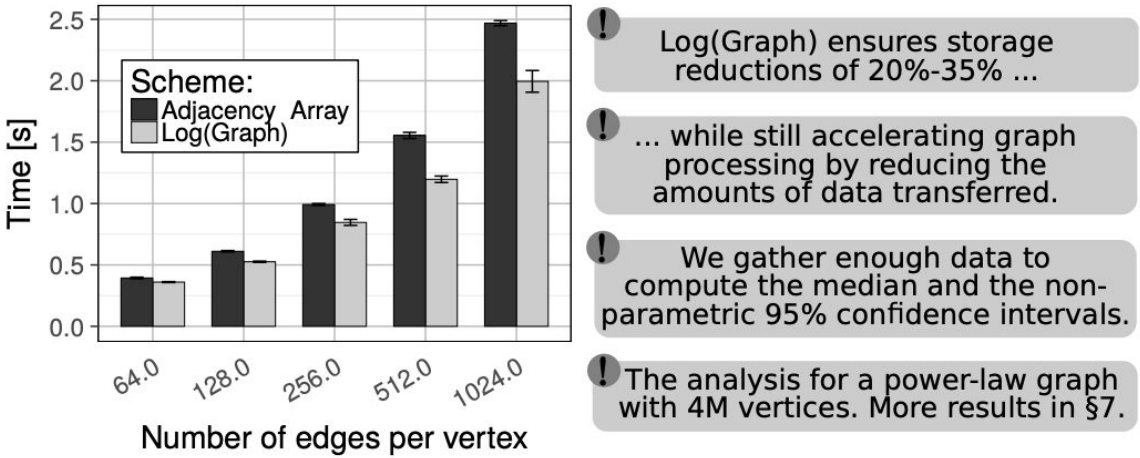



**Figure 1: (§ 1, § 7.2) The performance of Log(Graph) with the Single Source Shortest Path algorithm when logarithmizing vertex IDs.**

# Evaluation Strategy

Examined algorithms in the **GAP benchmark suite** such as BFS, PageRank, SSSP,  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

# Thoughts & Questions

- 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?