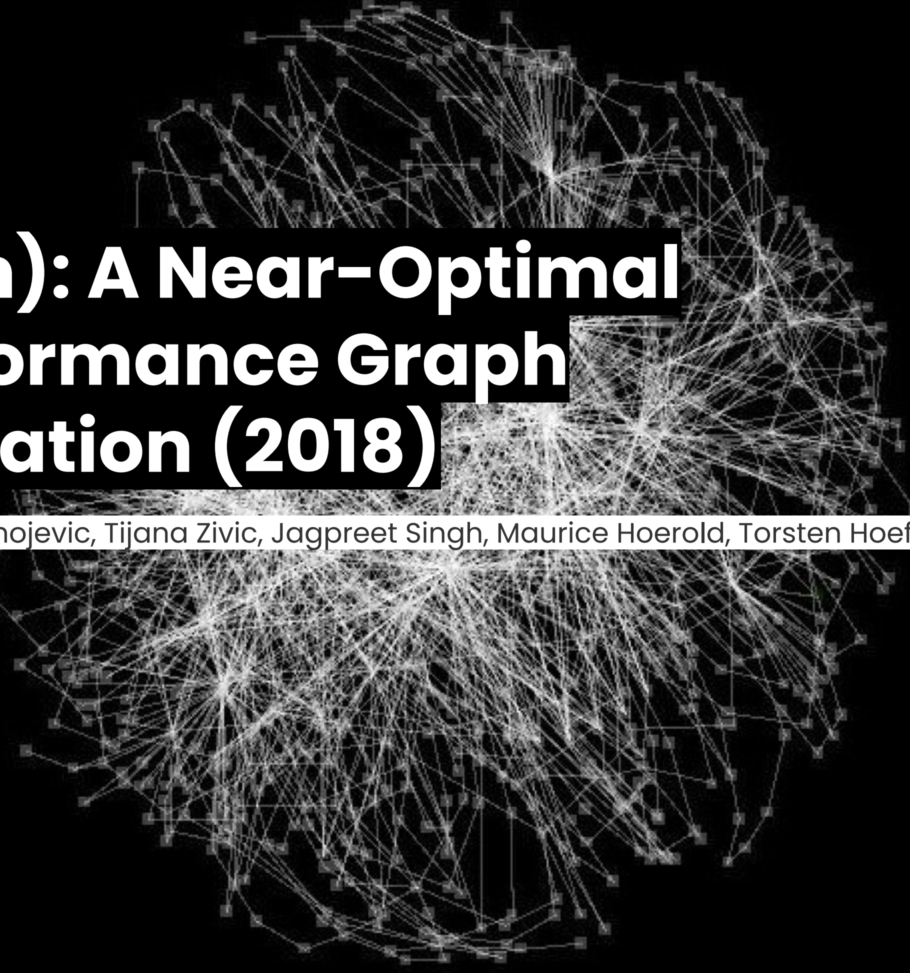


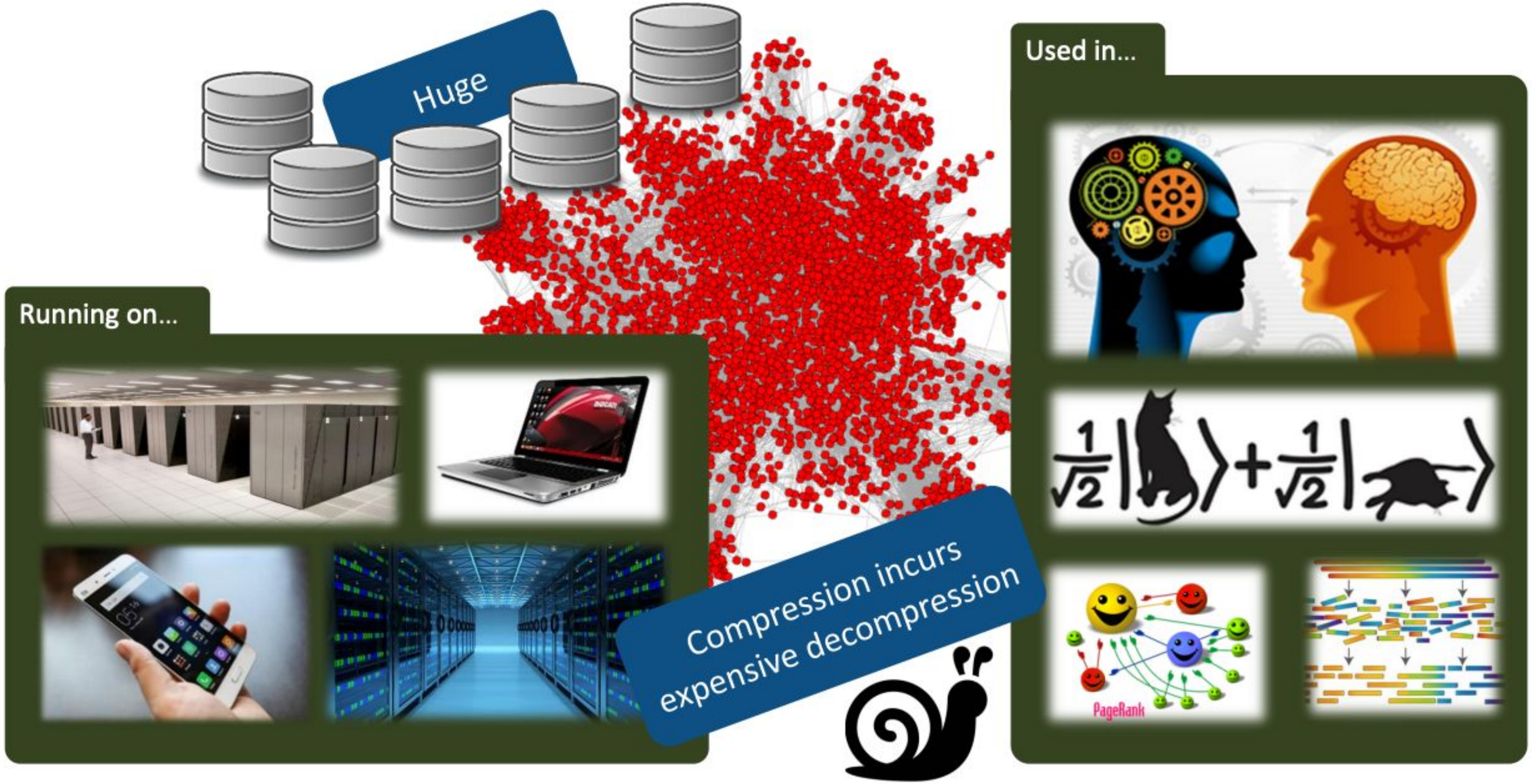
Log(Graph): A Near-Optimal High-Performance Graph Representation (2018)

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

Presentation by Qing Feng

Apr 19 2022

Big Graphs



Huge
Used in...
Running on...
Compression incurs expensive decompression
PageRank

Graph Compression

State-of-the-art graph compression framework, such as **Web-Graph**, uses reference encoding or interval encoding that requires **expensive decompression** due to pointer chasing caused by arbitrary nested encoding structure.

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Storage Lower Bound

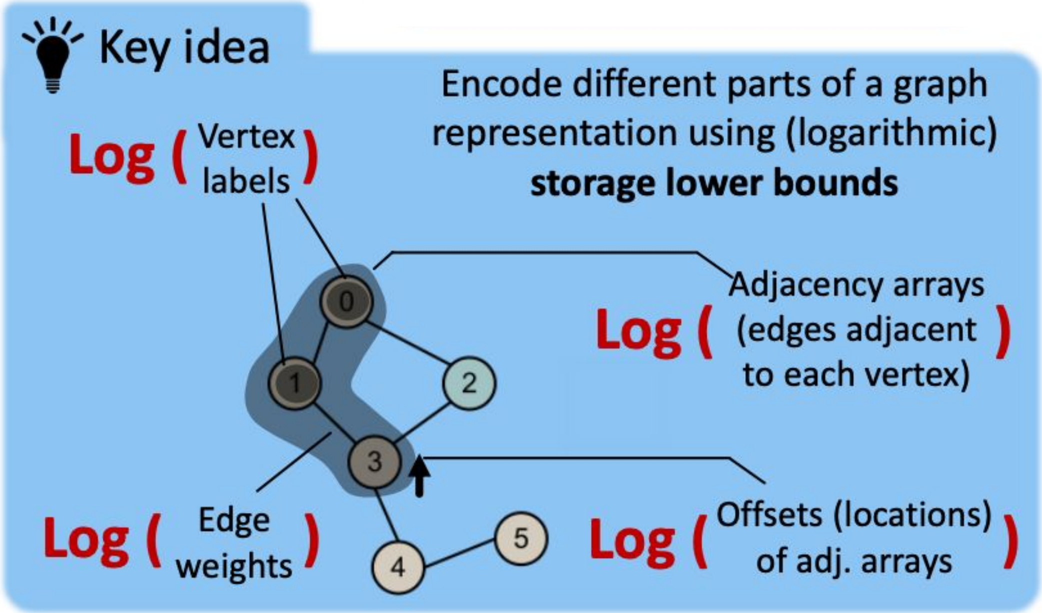
What is the lowest storage we can hope to achieve?

Logarithm-ization : one needs at least $\lceil \log |S| \rceil$ bits to store an object from a set S.

$$S = \{x_1, x_2, x_3, \ldots\} \qquad \begin{array}{l} x_1 \rightarrow 0 \ldots 01 \\ x_2 \rightarrow 0 \ldots 10 \\ x_3 \rightarrow 0 \ldots 11 \\ \quad\;\ldots \end{array}$$

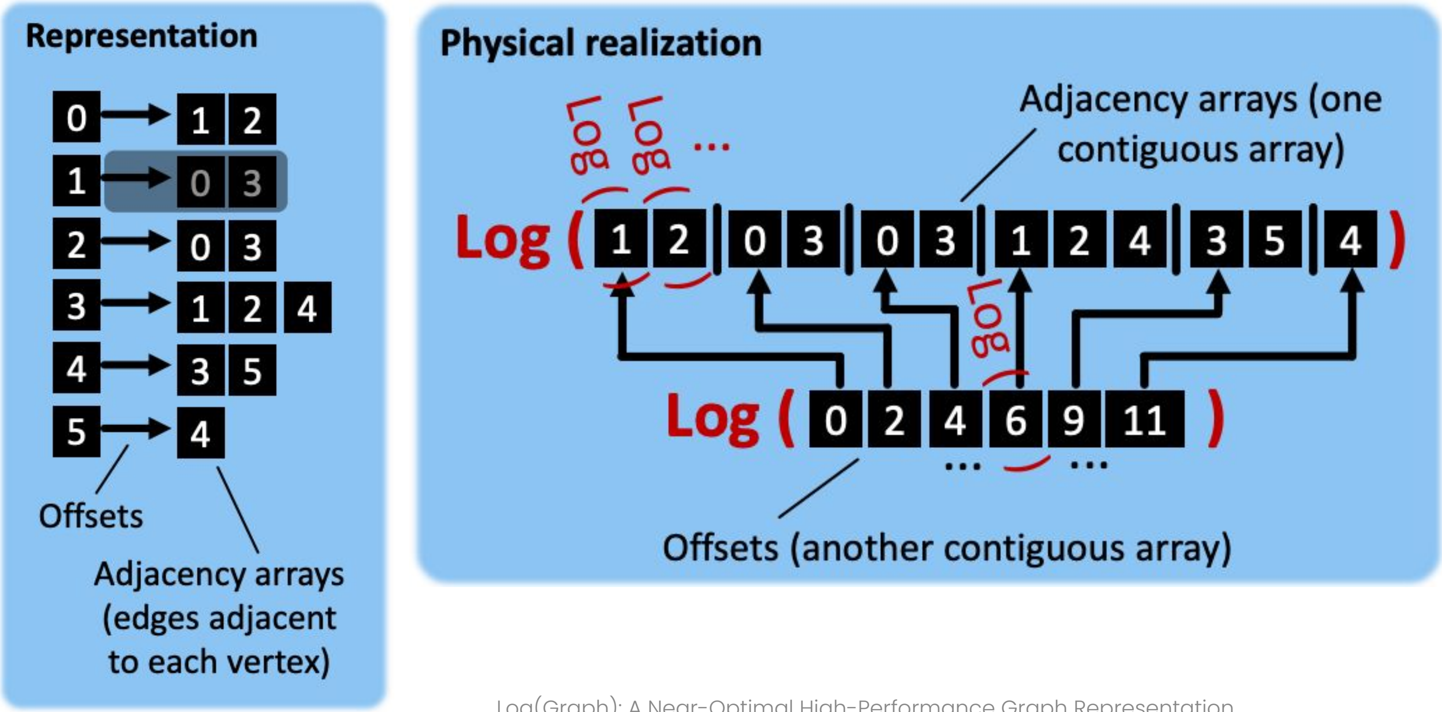
High-level Approach

Logarithm-ize different parts of graph representation accordingly.



Adjacency Array Representation

The Log(Graph) representation builds on the traditional **Adjacency Array** structure(similar to CSR?).

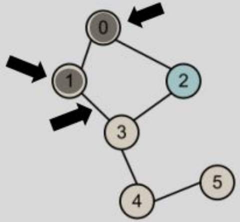


Fine Elements: eg. Vertex Label

Intuitively, **global lower bounds for vertex labels are O(log |V|).**
However, can further optimize for local cases:

Symbols

n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local)

Assume:

- a graph, e.g., $V = \{1, \ldots, 2^{22}\}$
- A vertex v with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\widehat{N_v} \ll n$

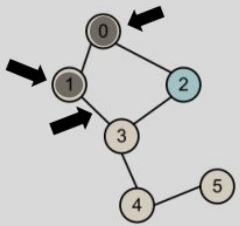
$$\lceil \log 2^{22} \rceil = 22$$

Thus, use the local bound $\lceil \log \widehat{N_v} \rceil$

Fine Elements: eg. Vertex Label

For **possible problem cases**, use Integer Linear Programming (ILP) to optimize.

Symbols



n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v

Lower bounds (local): problem

What if:

- a graph, e.g., $V = \{1, \ldots, 2^{22}\}$
- A vertex v with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\widehat{N_v} \ll n$
- ...one neighbor has a large ID:

$v \rightarrow$ | 2 | 3 | 4 | 5 | 1M |

$$\lceil \log 2^{20} \rceil = 20$$

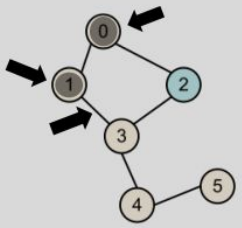
$v \rightarrow$ | 0...10 | 0...11 | 0...100 | 0...101 |

17 zeros!

Fine Elements: ILP

The main idea is to relabel vertices so that **the weighted sum of vertex labels are minimized**.

Symbols



n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v

Lower bounds (local) enhanced with ILP

Permute vertex labels **to reduce such maximum labels** in as many neighborhoods as possible

Permute(2 | 3 | 4 | 5 | 1M) = ? | ? | ? | ? | ?

(simultaneously for all other neighborhoods)

$\leq 100?$

v → 2 | 3 | 4 | 5 | 1M

Intuition: maximum labels in **new** neighborhoods will be **smaller**

Heuristics:

$$\min \sum_{v \in V} \widehat{N_v} \frac{1}{d_v}$$

Inverse of the neighborhood size

Fine Elements: ILP

The paper provides a polynomial greedy heuristic for relabeling:

Sort vertices in Line 8

Traverse starting from the smallest |Av| and assign a new smallest ID possible in Line 9

Relabel remaining vertices in Line 18

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

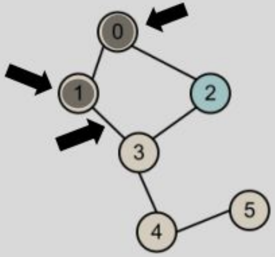
Listing 1: (§ 3.6) The greedy heuristic for vertex relabeling.

Analysis

Compressing fine elements consistently reduces storage.

Symbols

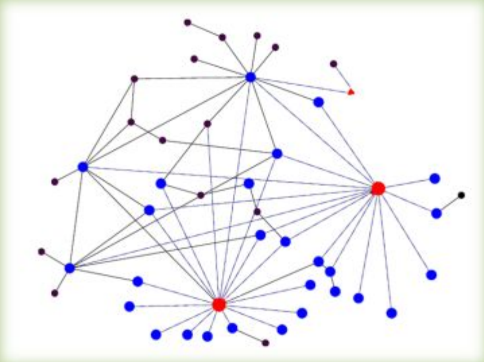
$\widehat{W}$: max edge weight,
n : #vertices,
p, α, β : constants



Power-law graphs

The probability that a vertex has degree *d* is:

$$\alpha d^{\beta}$$



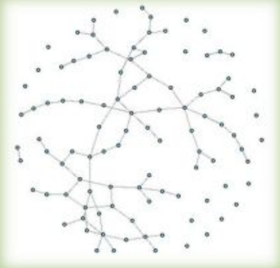
Expected size of the adjacency array

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

Random uniform graphs

The probability that a vertex has degree *d* is:

$$pd$$

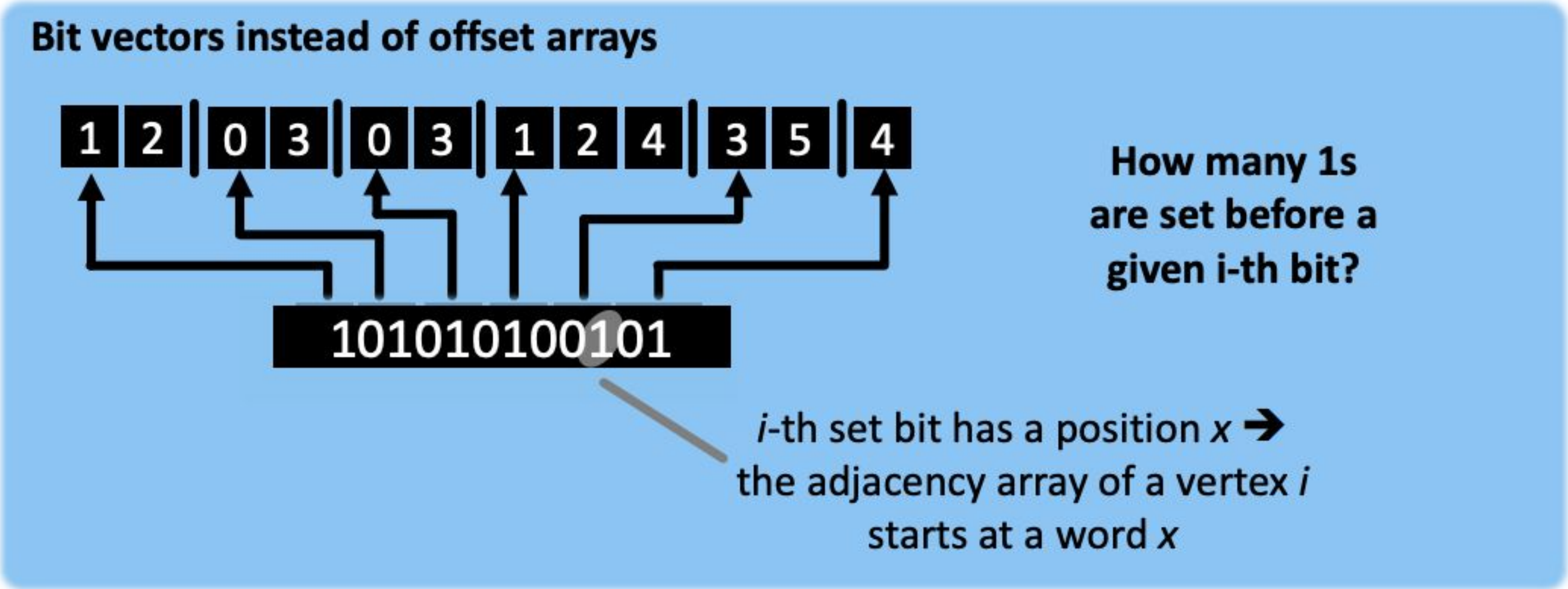


Expected size of the adjacency array

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

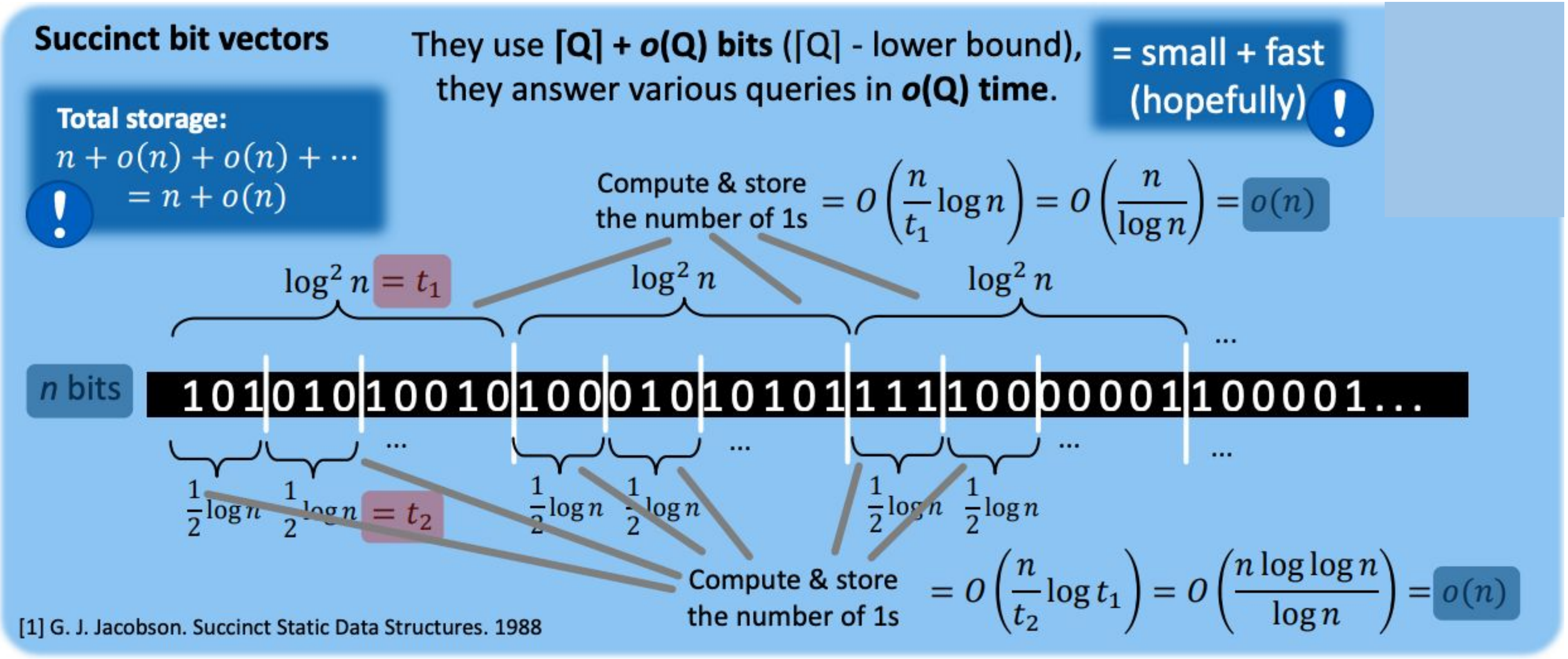
Offset Structure

The high-level idea is to represent the offset array using a **bit vector**.



Offset Structure

Specifically, use Succinct Bit Vectors that supports quick queries. The main idea is to **divide the bit vector to small chunks, group them in a table, and represent chunks using indices into the table**.



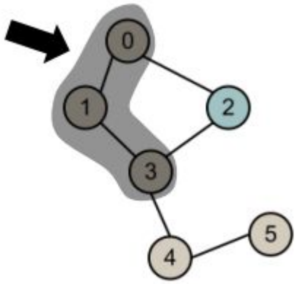
Analysis

Some structures support constant-time query and are considered candidates.

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

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

Adjacency Structure



Relabel the vertices using Degree-Minimizing + Gap Encoding.

Degree-Minimizing: Targeting general graphs (no assumptions on graph structure)

Permute(2 | 3 | 4 | 5 | 1M) = v | w | x | y | z

(simultaneously for all other neighborhoods)

(1) The more often a label occurs (i.e., the higher vertex degree), the smaller permuted value it receives

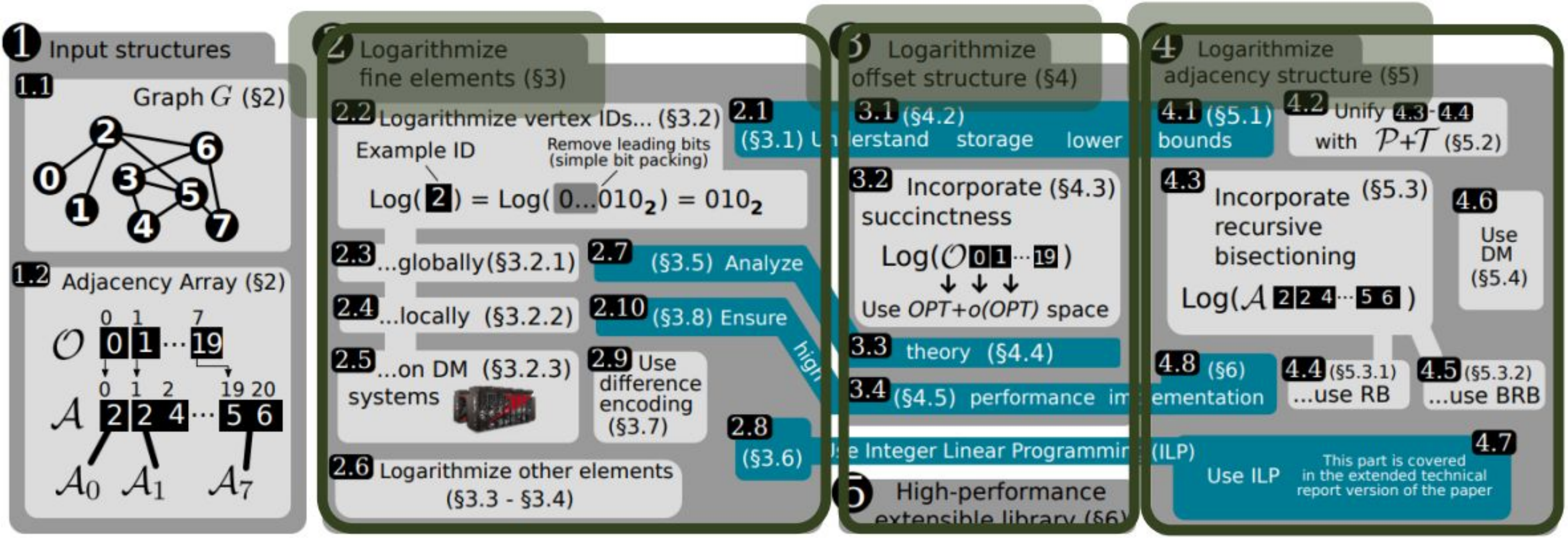
Gap-encode(v | w | x | y | z) = v | w-v | x-w | y-x | z-y

(2) Encode new labels with gap encoding (differences between consecutive labels instead of full labels)

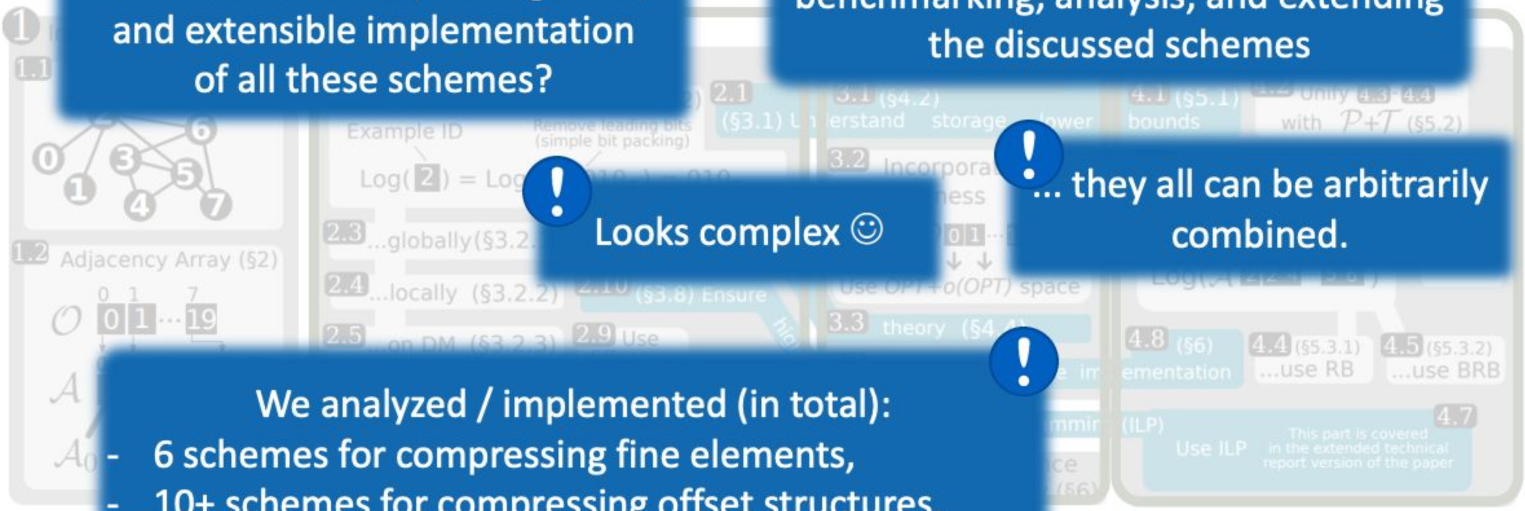
More schemes that assume specific classes of graphs

...

Overall Framework



Overall Framework



How to ensure fast, manageable, and extensible implementation of all these schemes?

We use C++ templates to develop a library that facilitates implementation, benchmarking, analysis, and extending the discussed schemes

Looks complex ☺

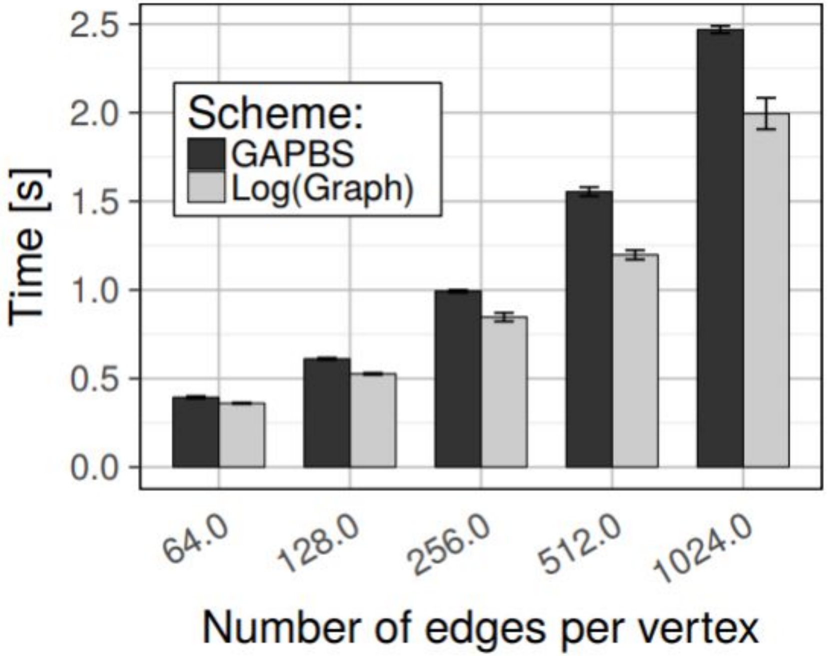
… they all can be arbitrarily combined.

We analyzed / implemented (in total):

- 6 schemes for compressing fine elements,
- 10+ schemes for compressing offset structures,
- 4+ schemes for compressing adjacency structures

Evaluation: Fine Elements

On both synthetic and real-world graphs, running various algorithms, compared with GAPBS, Log(Graph) **consistently reduces storage overhead by 20-35% while outperforming it**.



Evaluation: Storage

On real-world graphs, **Succinct bit vectors consistently ensure best storage reductions,** mainly because real-world graphs are typically sparse.

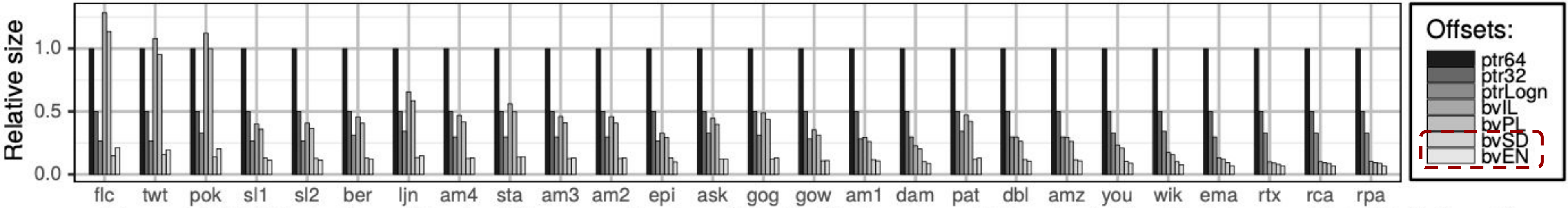


Figure 6: (§ 7.3) Illustration of the size differences of various $\mathcal{O}$ (both offset arrays and bit vectors). The offset sizes are $W \in \{32, 64, \lceil \log n \rceil\}$.

ptr64, ptr32: traditional array of offsets
ptrLogn: separate compression of each offset
bvPL: plain bit vectors
bvIL: compact bit vectors
bvEN, bvSD: succinct bit vectors

Evaluation: Performance

On accessing random selected neighbors, once parallelism overheads kick in, **performance of accessing succinct bit vectors and offset arrays becomes comparable**. The bvSD scheme is usually the fastest and the smallest.

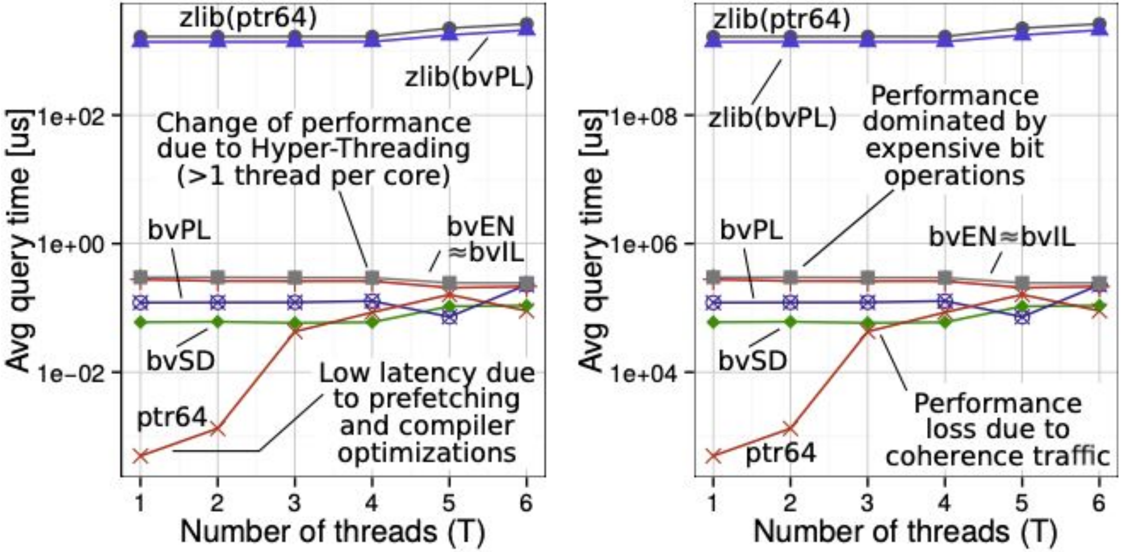


Figure 7: (§ 7.3) Performance analysis of various types of $\mathcal{O}$.

ptr64: traditional array of offsets
bvPL: plain bit vectors
bvIL: compact bit vectors
bvEN, bvSD: succinct bit vectors
zlib(.): zlib-compressed variants

Evaluation: Tunable Combination

Key insight (vertex labels)

20-35% storage reductions (compared to uncompressed data) and **negligible** decompression overheads

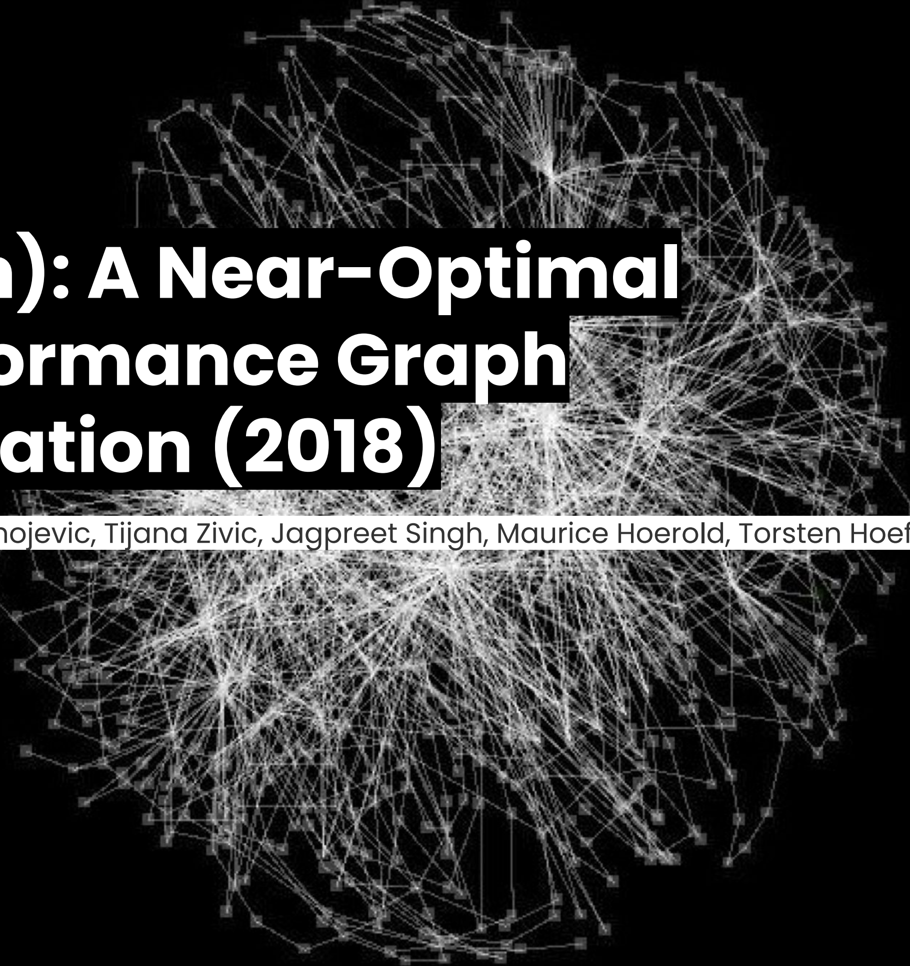
Key insight (adjacency data)

80% storage reductions (compared to uncompressed data) and up to >2x speedup over modern graph compression schemes (Webgraph)

Takeaway (Results): Log(Graph) ensures Space-Performance sweetspot (tunable!)

Key insight (offsets)

Up to >90% storage reductions (compared to uncompressed data) and comparable performance to that of uncompressed data accesses (in parallel environments)



Log(Graph): A Near-Optimal High-Performance Graph Representation (2018)

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

Presentation by Qing Feng

Apr 19 2022