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

Running on...

Used in...

Huge

Compression incurs expensive decompression

Log(Graph): A Near-Optimal High-Performance Graph Representation
MIT 6.827 Algorithm Engineering | Qing Feng
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.

<table>
<thead>
<tr>
<th>Node</th>
<th>Outdegree</th>
<th>Successors</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>15, 16, 17, 22, 23, 24, 315, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>13, 15, 16, 17, 50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th>Copy list</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>...</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>01110011010</td>
<td>22, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td>...</td>
<td>50</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>11110000000</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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 $S$. 

$S = \{x_1, x_2, x_3, \ldots \}$

$x_1 \rightarrow 0 \ldots 01$

$x_2 \rightarrow 0 \ldots 10$

$x_3 \rightarrow 0 \ldots 11$

$\ldots$
High-level Approach

Logarithm-ize different parts of graph representation accordingly.

Key idea

- Log (Vertex labels)
- Log (Edge weights)
- Log (Adjacency arrays (edges adjacent to each vertex))
- Log (Offsets (locations) of adj. arrays)

Encode different parts of a graph representation using (logarithmic) storage lower bounds.
Adjacency Array Representation

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

---

**Representation**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Physical realization**

```
Log ( 1 2 | 0 3 | 0 3 | 1 2 4 | 3 5 | 4 )
```

Adjacency arrays (one contiguous array)

```
Log ( 0 2 4 6 9 11 )
```

Offsets (another contiguous array)

---

Log(Graph): A Near-Optimal High-Performance Graph Representation
MIT 6.827 Algorithm Engineering | Qing Feng
Fine Elements: eg. Vertex Label

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

**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: $\hat{N}_v \ll n$

Thus, use the local bound $\lceil \log \hat{N}_v \rceil$
Fine Elements: eg. Vertex Label

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

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: $\bar{N}_v \ll n$
- ...one neighbor has a large ID:

\[
\begin{align*}
\left\lfloor \log 2^{20} \right\rfloor &= 20 \\
&=
\begin{cases}
0\ldots10 & 0\ldots11 \\
0\ldots100 & 0\ldots101
\end{cases}
\end{align*}
\]

17 zeros!
Fine Elements: ILP

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

Lower bounds (local) enhanced with ILP

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

Permuted labels: 2 3 4 5 1M

Intuition: maximum labels in new neighborhoods will be smaller

Heuristics: \( \min \sum_{v \in V} \frac{1}{d_v} \)

Inverse of the neighborhood size

(Permuted simultaneously for all other neighborhoods)

\[ \leq 100? \]
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

```c
/* Input: graph G, Output: a new relabeling $N(v), \forall v \in V$. */
void relabel(G) {
    ID[0..n-1] = [0..n-1]; // An array with vertex IDs.
    D[0..n-1] = [d_0..d_{n-1}]; // An array with degrees of vertices.
    // An auxiliary array for determining if a vertex was relabeled:
    visit[0..n-1] = [false..false];
    nl = 1; // An auxiliary variable ```new label''.
    sort(ID); sort(D);
    for(int i = 1; i < n; ++i) // For each vertex...
        for(int j = 0; j < D[i]; ++j) { // For each neighbor...
            int id = $N_i,ID[i]$; // $N_i,ID[i]$ is jth neighbor of vertex with ID $ID[i]
            if(visit[id] == false) {
                $N(id) = nl++;
                visit[id] = true;
            } }
    for(int i = 1; i < n; ++i)
        if(visit[i] == false)
            $N(id) = nl++;
}
```

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

Compressing fine elements consistently reduces storage.

**Power-law graphs**

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

$$\alpha d^\beta$$

Expected size of the adjacency array

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

**Random uniform graphs**

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

$$pd$$

Expected size of the adjacency array

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

**Symbols**

- $\hat{W}$ : max edge weight,
- $n$ : #vertices,
- $p, \alpha, \beta$ : constants
Offset Structure

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

**Bit vectors instead of offset arrays**

How many 1s are set before a given i-th bit?

\[
\begin{array}{cccccccc}
1 & 2 & 0 & 3 & 0 & 3 & 1 & 2 & 4 & 3 & 5 & 4 \\
101010100101
\end{array}
\]

\(i\)-th set bit has a position \(x\) ➔ the adjacency array of a vertex \(i\) starts at a word \(x\)
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.

\[ n + o(n) + o(n) + \ldots = n + o(n) \]

They use \([Q] + o(Q)\) bits ([Q] - lower bound), they answer various queries in \(o(Q)\) time.

\[
\begin{align*}
\log^2 n &= t_1 \\
\log^2 n &= t_1 \\
\log^2 n &= t_1 \\
\frac{1}{2} \log n &= \frac{1}{2} \log n \\
\frac{1}{2} \log n &= \frac{1}{2} \log n \\
\frac{1}{2} \log n &= \frac{1}{2} \log n \\
\end{align*}
\]

\[
\begin{align*}
\frac{n}{t_1} \log n &= o(n) \\
\frac{n}{t_2} \log^{\log n} n &= o(n) \\
\end{align*}
\]

## Analysis

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

<table>
<thead>
<tr>
<th>$O$</th>
<th>ID</th>
<th>Asymptotic size [bits]</th>
<th>Exact size [bits]</th>
<th>select or $O[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer array</td>
<td>ptrW</td>
<td>$O(Wn)$</td>
<td>$W(n + 1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Plain [44]</td>
<td>bvPL</td>
<td>$O\left(\frac{Wm}{B}\right)$</td>
<td>$\frac{2Wm}{B}$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Interleaved [44]</td>
<td>bvIL</td>
<td>$O\left(\frac{Wm}{B} + \frac{Wm}{L}\right)$</td>
<td>$2Wm \left(\frac{1}{B} + \frac{64}{L}\right)$</td>
<td>$O\left(\log \frac{Wm}{B}\right)$</td>
</tr>
<tr>
<td>Entropy based [31, 78]</td>
<td>bvEN</td>
<td>$O\left(\frac{Wm}{B} \log \frac{Wm}{B}\right)$</td>
<td>$\approx \log \left(\frac{2Wm}{Bn}\right)$</td>
<td>$O\left(\log \frac{Wm}{B}\right)$</td>
</tr>
<tr>
<td>Sparse [76]</td>
<td>bvSD</td>
<td>$O\left(n + n \log \frac{Wm}{Bu}\right)$</td>
<td>$\approx n \left(2 + \log \frac{2Wm}{Bn}\right)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>B-tree based [1]</td>
<td>bvBT</td>
<td>$O\left(\frac{Wm}{B}\right)$</td>
<td>$\approx 1.1 \cdot \frac{2Wm}{B}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Gap-compressed [1]</td>
<td>bvGC</td>
<td>$O\left(\frac{Wm}{B} \log \frac{Wm}{Bn}\right)$</td>
<td>$\approx 1.3 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn}$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Table 4: (§ 4.3) Theoretical analysis of various types of $O$ 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)
Overall Framework
Overall Framework

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

Looks complex 😊

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

... 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 $O$ (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.

(a) Twitter graph tw.  
(b) California road graph rca.  

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)

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

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