MACIEJ BESTA, DIMITRI STANOJEVIC, TIJANA ZIVIC, JAGPREET SINGH, MAURICE HOEROLD, TORSTEN HOEFLER

Log(Graph): A Near-Optimal High-Performance Graph Representation
Large graphs...
Large graphs...
Large graphs...
Large graphs...
Large graphs...
Large graphs...
Large graphs...
Large graphs ...

Running on ...

Used in ...
Large graphs...
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>Wikipedia edits (en)</td>
<td>30,757,442</td>
<td>172,591,272</td>
</tr>
<tr>
<td>TW</td>
<td>Twitter (WWW)</td>
<td>41,652,230</td>
<td>1,468,365,182</td>
</tr>
<tr>
<td>TF</td>
<td>Twitter (MPI)</td>
<td>52,579,682</td>
<td>1,903,263,821</td>
</tr>
<tr>
<td>FR</td>
<td>Friendster</td>
<td>68,349,466</td>
<td>2,586,147,869</td>
</tr>
<tr>
<td>UL</td>
<td>UK domain (2007)</td>
<td>105,153,952</td>
<td>3,301,876,564</td>
</tr>
</tbody>
</table>

KONECT graph datasets
## Graph500 Benchmark

### Top Ten from June 2018 BFS

<table>
<thead>
<tr>
<th>RANK</th>
<th>MACHINE</th>
<th>VENDOR</th>
<th>INSTALLATION SITE</th>
<th>LOCATION</th>
<th>COUNTRY</th>
<th>YEAR</th>
<th>NUMBER OF NODES</th>
<th>NUMBER OF CORES</th>
<th>SCALE</th>
<th>GTEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>K computer</td>
<td>Fujitsu</td>
<td>RIKEN Advanced Institute for Computational Science (AICS)</td>
<td>Kobe Hyogo</td>
<td>Japan</td>
<td>2011</td>
<td>82944</td>
<td>663552</td>
<td>40</td>
<td>38621.4</td>
</tr>
<tr>
<td>2</td>
<td>Sunway TaihuLight</td>
<td>NRPC</td>
<td>National Supercomputing Center in Wuxi</td>
<td>Wuxi</td>
<td>China</td>
<td>2015</td>
<td>40768</td>
<td>10599680</td>
<td>40</td>
<td>23755.7</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL Sequoia</td>
<td>IBM</td>
<td>Lawrence Livermore National Laboratory</td>
<td>Livermore CA</td>
<td>USA</td>
<td>2012</td>
<td>98304</td>
<td>1572864</td>
<td>41</td>
<td>23751</td>
</tr>
<tr>
<td>4</td>
<td>DOE/SC/Argonne National Laboratory Mira</td>
<td>IBM</td>
<td>Argonne National Laboratory</td>
<td>Chicago IL</td>
<td>USA</td>
<td>2012</td>
<td>49152</td>
<td>786432</td>
<td>40</td>
<td>14982</td>
</tr>
</tbody>
</table>
Large graphs...

Graph500 Benchmark

KONECT graph datasets

Webgraph datasets

<table>
<thead>
<tr>
<th>Graph</th>
<th>Crawl date</th>
<th>Nodes</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>uk-2014</td>
<td>2014</td>
<td>787801471</td>
<td>47614527250</td>
</tr>
<tr>
<td>eu-2015</td>
<td>2015</td>
<td>1070557254</td>
<td>91792261600</td>
</tr>
<tr>
<td>gsh-2015</td>
<td>2015</td>
<td>988490691</td>
<td>33877399152</td>
</tr>
<tr>
<td>uk-2014-host</td>
<td>2014</td>
<td>4769354</td>
<td>50829923</td>
</tr>
<tr>
<td>eu-2015-host</td>
<td>2015</td>
<td>11264052</td>
<td>386915963</td>
</tr>
<tr>
<td>gsh-2015-host</td>
<td>2015</td>
<td>6866014</td>
<td>1802747600</td>
</tr>
<tr>
<td>uk-2014-tpd</td>
<td>2014</td>
<td>1766010</td>
<td>18244650</td>
</tr>
<tr>
<td>eu-2015-tpd</td>
<td>2015</td>
<td>6650532</td>
<td>170145510</td>
</tr>
<tr>
<td>gsh-2015-tpd</td>
<td>2015</td>
<td>30809122</td>
<td>602119716</td>
</tr>
<tr>
<td>clueweb12</td>
<td>2012</td>
<td>978408098</td>
<td>42574107469</td>
</tr>
<tr>
<td>uk-2002</td>
<td>2002</td>
<td>18520486</td>
<td>298113762</td>
</tr>
</tbody>
</table>
Graph500 Benchmark

Web data commons datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Granularity</th>
<th>#Nodes</th>
<th>#Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td></td>
<td>3.563 million</td>
<td>128,736 million</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>101 million</td>
<td>2,043 million</td>
</tr>
<tr>
<td>Pay-Level-Domain</td>
<td></td>
<td>43 million</td>
<td>623 million</td>
</tr>
</tbody>
</table>

KONECT graph datasets

Webgraph datasets

<table>
<thead>
<tr>
<th>Graph</th>
<th>Crawl date</th>
<th>Nodes</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>uk-2014</td>
<td>2014</td>
<td>787801471</td>
<td>47614527250</td>
</tr>
<tr>
<td>eu-2015</td>
<td>2015</td>
<td>1070557254</td>
<td>91792261600</td>
</tr>
<tr>
<td>gsh-2015</td>
<td>2015</td>
<td>988490691</td>
<td>33877399152</td>
</tr>
<tr>
<td>uk-2014-host</td>
<td>2014</td>
<td>4769354</td>
<td>50829923</td>
</tr>
<tr>
<td>eu-2015-host</td>
<td>2015</td>
<td>11264052</td>
<td>386915963</td>
</tr>
<tr>
<td>gsh-2015-host</td>
<td>2015</td>
<td>68660142</td>
<td>1802747600</td>
</tr>
<tr>
<td>uk-2014-tpd</td>
<td>2014</td>
<td>1766010</td>
<td>18244650</td>
</tr>
<tr>
<td>eu-2015-tpd</td>
<td>2015</td>
<td>6650532</td>
<td>170145510</td>
</tr>
<tr>
<td>gsh-2015-tpd</td>
<td>2015</td>
<td>30809122</td>
<td>602119716</td>
</tr>
<tr>
<td>clueweb12</td>
<td>2012</td>
<td>978408098</td>
<td>42574107469</td>
</tr>
<tr>
<td>uk-2002</td>
<td>2002</td>
<td>18520486</td>
<td>298113762</td>
</tr>
</tbody>
</table>
Large graphs...

**Graph500 Benchmark**

**Webgraph datasets**

<table>
<thead>
<tr>
<th>Graph</th>
<th>Crawl date</th>
<th>Nodes</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>uk-2014</td>
<td>2014</td>
<td>787,801,471</td>
<td>47,614,527,250</td>
</tr>
<tr>
<td>eu-2015</td>
<td>2015</td>
<td>1,070,557,254</td>
<td>91,792,261,600</td>
</tr>
<tr>
<td>gsh-2015</td>
<td>2015</td>
<td>988,490,069</td>
<td>33,877,399,152</td>
</tr>
<tr>
<td>uk-2014-host</td>
<td>2014</td>
<td>4,769,354</td>
<td>50,829,923</td>
</tr>
<tr>
<td>eu-2015-host</td>
<td>2015</td>
<td>11,264,052</td>
<td>386,915,963</td>
</tr>
<tr>
<td>gsh-2015-host</td>
<td>2015</td>
<td>68,660,142</td>
<td>1,802,747,600</td>
</tr>
<tr>
<td>uk-2014-tpd</td>
<td>2014</td>
<td>1,766,010</td>
<td>18,244,650</td>
</tr>
<tr>
<td>eu-2015-tpd</td>
<td>2015</td>
<td>6,650,532</td>
<td>17,014,551</td>
</tr>
<tr>
<td>gsh-2015-tpd</td>
<td>2015</td>
<td>30,809,122</td>
<td>602,119,716</td>
</tr>
<tr>
<td>clueweb12</td>
<td>2012</td>
<td>978,408,098</td>
<td>42,574,107,469</td>
</tr>
<tr>
<td>uk-2002</td>
<td>2002</td>
<td>18,520,486</td>
<td>298,113,762</td>
</tr>
</tbody>
</table>

**Web data commons datasets**

<table>
<thead>
<tr>
<th>Granularity</th>
<th>#Nodes</th>
<th>#Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td>3.563 million</td>
<td>128,736 million</td>
</tr>
<tr>
<td>Host</td>
<td>101 million</td>
<td>2,043 million</td>
</tr>
<tr>
<td>Pay-Level-Domain</td>
<td>43 million</td>
<td>623 million</td>
</tr>
</tbody>
</table>

**KONECT graph datasets**

- Twitter (WWW)
- Twitter (MPI)
- Friendster
- UK domain (2007)
Large graphs...

Running on...

Used in...

\[ \frac{1}{\sqrt{2}} |\text{cat}\rangle + \frac{1}{\sqrt{2}} |\text{mouse}\rangle \]
Large graphs...

Compression incurs expensive decompression
Large graphs...

Running on...

Used in...

Log(Graph): effective compression with low-overhead decompression!

Compression incurs expensive decompression
What is **the lowest storage** we can (hope to) use to store a graph?
What is the lowest storage we can (hope to) use to store a graph?
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound $\Omega$

Which one? 😊
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound $\Omega$.

Which one? 😊

Counting bounds. They are logarithmic (one needs at least $\log |S|$ bits to store an object from an arbitrary set $S$).
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound \( \Omega \)

Which one? 😊

Counting bounds. They are logarithmic (one needs at least \( \log|S| \) bits to store an object from an arbitrary set \( S \))

\[ S = \{x_1, x_2, x_3, \ldots \} \]

\[ x_1 \rightarrow 0 \ldots 01 \]
\[ x_2 \rightarrow 0 \ldots 10 \]
\[ x_3 \rightarrow 0 \ldots 11 \]

...
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound \( \Omega \)

Which one? 😊

Counting bounds. They are logarithmic (one needs at least \( \log |S| \) bits to store an object from an arbitrary set \( S \))

Key idea

\[ S = \{ x_1, x_2, x_3, \ldots \} \]

\[
\begin{align*}
    x_1 & \rightarrow 0 \ldots 01 \\
    x_2 & \rightarrow 0 \ldots 10 \\
    x_3 & \rightarrow 0 \ldots 11 \\
    \vdots
\end{align*}
\]
What is **the lowest storage** we can (hope to) use to store a graph?

The storage **lower bound** $\Omega$

Which one? 😊

**Counting bounds.** They are **logarithmic** (one needs at least $\log|S|$ bits to store an object from an arbitrary set $S$)

**Key idea**

Encode different parts of a graph representation using (logarithmic) **storage lower bounds**

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

- $x_1 \rightarrow 0 \ldots 01$
- $x_2 \rightarrow 0 \ldots 10$
- $x_3 \rightarrow 0 \ldots 11$
- ...

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

$$S = \{x_1, x_2, x_3, \ldots\}$$
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound $\Omega$

Which one? 😊

Counting bounds. They are logarithmic (one needs at least $\log |S|$ bits to store an object from an arbitrary set $S$)

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

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

- $x_1 \rightarrow 0 \ldots 01$
- $x_2 \rightarrow 0 \ldots 10$
- $x_3 \rightarrow 0 \ldots 11$

$\cdots$

Vertex labels

0

1

2

3

4

5
What is the **lowest storage** we can (hope to) use to store a graph?

The storage **lower bound** \( \Omega \)

Which one? 😊

Counting bounds. They are **logarithmic** (one needs at least \( \log |S| \) bits to store an object from an arbitrary set \( S \))

**Key idea**

Encode different parts of a graph representation using (logarithmic) **storage lower bounds**

\[ S = \{ x_1, x_2, x_3, \ldots \} \]

\[ x_1 \rightarrow 0 \ldots 01 \]
\[ x_2 \rightarrow 0 \ldots 10 \]
\[ x_3 \rightarrow 0 \ldots 11 \]

...
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound \( \Omega \)

Which one? 😊

Counting bounds. They are logarithmic (one needs at least \( \log |S| \) bits to store an object from an arbitrary set \( S \))

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

\[ S = \{x_1, x_2, x_3, \ldots \} \]

- \( x_1 \rightarrow 0 \ldots 01 \)
- \( x_2 \rightarrow 0 \ldots 10 \)
- \( x_3 \rightarrow 0 \ldots 11 \)
- \( \ldots \)

Adjacency arrays (edges adjacent to each vertex)

Edge weights

Vertex labels
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound \( \Omega \)

Which one? 😊

Counting bounds. They are logarithmic (one needs at least \( \log |S| \) bits to store an object from an arbitrary set \( S \))

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

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

Graph example:

\[ S = \{ x_1, x_2, x_3, \ldots \} \]

\[ x_1 \rightarrow 0 \ldots 01 \]

\[ x_2 \rightarrow 0 \ldots 10 \]

\[ x_3 \rightarrow 0 \ldots 11 \]

\[ \ldots \]
What is the **lowest storage** we can (hope to) use to store a graph?

**Key idea**

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

**Counting bounds.** They are logarithmic (one needs at least \( \log |S| \) bits to store an object from an arbitrary set \( S \))

**The storage lower bound** \( \Omega \)

Which one? 😊

\[
S = \{ x_1, x_2, x_3, \ldots \} \\
\begin{align*}
x_1 &\to 0 \ldots 01 \\
x_2 &\to 0 \ldots 10 \\
x_3 &\to 0 \ldots 11 \\
\ldots
\end{align*}
\]
What is **the lowest storage** we can (hope to) use to store a graph?

The storage **lower bound** $\Omega$

Which one? 😊

Counting bounds. They are **logarithmic** (one needs at least $\log |S|$ bits to store an object from an arbitrary set $S$)

Key idea

Encode different parts of a graph representation using (logarithmic) **storage lower bounds**

<table>
<thead>
<tr>
<th>Vertex labels</th>
<th>$S = {x_1, x_2, x_3, \ldots }$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \to 0 \ldots 01$</td>
<td></td>
</tr>
<tr>
<td>$x_2 \to 0 \ldots 10$</td>
<td></td>
</tr>
<tr>
<td>$x_3 \to 0 \ldots 11$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Adjacency arrays (edges adjacent to each vertex)

Offsets (locations) of adj. arrays

Log (Vertex labels)

Log (Edge weights)
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound $\Omega$

Which one? 😊

Counting bounds. They are logarithmic (one needs at least $\log |S|$ bits to store an object from an arbitrary set $S$)

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

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

- $x_1 \rightarrow 0 \ldots 01$
- $x_2 \rightarrow 0 \ldots 10$
- $x_3 \rightarrow 0 \ldots 11$

- Adjacency arrays (edges adjacent to each vertex)
- Offsets (locations) of adj. arrays

- Log (Edge weights)
- Log (Vertex labels)
- Log (Adjacency arrays)
What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound $\Omega$

Which one? 😊

Counting bounds. They are logarithmic (one needs at least $\log |S|$ bits to store an object from an arbitrary set $S$)

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

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

- $x_1 \rightarrow 0 \ldots 01$
- $x_2 \rightarrow 0 \ldots 10$
- $x_3 \rightarrow 0 \ldots 11$
- $\ldots$

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

- $x_1 \rightarrow 0 \ldots 01$
- $x_2 \rightarrow 0 \ldots 10$
- $x_3 \rightarrow 0 \ldots 11$
- $\ldots$

Adjacency arrays (edges adjacent to each vertex)

Offsets (locations) of adj. arrays
ADJACENCY ARRAY GRAPH REPRESENTATION
ADJACENCY ARRAY GRAPH REPRESENTATION

Representation
**ADJACENCY ARRAY GRAPH REPRESENTATION**

Representation

0
1
2
3
4
5
## Adjacency Array Graph Representation

**Representation**

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

**Offsets**

Adjacency arrays (edges adjacent to each vertex)
### Adjacency Array Graph Representation

**Representation**

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

**Offsets**

**Adjacency arrays (edges adjacent to each vertex)**
**Adjacency Array Graph Representation**

### Representation

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

**Offsets**

**Adjacency arrays**
(edges adjacent to each vertex)

### Physical realization
Adjacency Array Graph Representation

**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>

**Offsets**

Adjacency arrays (edges adjacent to each vertex)

**Physical realization**

Adjacency arrays (one contiguous array)
**Adjacency Array Graph Representation**

**Representation**

<table>
<thead>
<tr>
<th>Vertex (A)</th>
<th>Adjacency Arrays (edges adjacent to each vertex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>0, 3</td>
</tr>
<tr>
<td>2</td>
<td>0, 3</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>4</td>
<td>3, 5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Physical realization**

- **Offsets**: Another contiguous array
- **Adjacency Arrays**: One contiguous array

Adjacency arrays form the basis of the representation, where each vertex is connected to its neighbors. The offsets provide the starting positions in the array for each vertex's adjacent vertices.
Adjacency Arrays (edges adjacent to each vertex)

Offsets

Adjacency arrays
( one contiguous array )

Offsets (another contiguous array)
**Adjacency Array Graph Representation**

<table>
<thead>
<tr>
<th>Representation</th>
<th>Physical realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 1 2</td>
<td>Adjacency arrays (one contiguous array)</td>
</tr>
<tr>
<td>1 → 0 3</td>
<td>Offsets (another contiguous array)</td>
</tr>
<tr>
<td>2 → 0 3</td>
<td></td>
</tr>
<tr>
<td>3 → 1 2 4</td>
<td></td>
</tr>
<tr>
<td>4 → 3 5</td>
<td></td>
</tr>
<tr>
<td>5 → 4</td>
<td></td>
</tr>
</tbody>
</table>

Adjacency arrays (edges adjacent to each vertex)
Adjacency arrays (edges adjacent to each vertex)

Offsets

Adjacency arrays (one contiguous array)

Offsets (another contiguous array)
**Adjacency Array Graph Representation**

### Representation

- 0 → 1 2
- 1 → 0 3
- 2 → 0 3
- 3 → 1 2 4
- 4 → 3 5
- 5 → 4

**Adjacency arrays** (edges adjacent to each vertex)

**Offsets**

### Physical realization

**Adjacency arrays** (one contiguous array)

**Offsets** (another contiguous array)
1 \text{ Log ( Vertex labels ), Log ( Edge weights )}
Log (Vertex labels), Log (Edge weights)
1. $\log(\text{Vertex labels}), \log(\text{Edge weights})$
Vertex labels

Edge weights

Log (Vertex labels), Log (Edge weights)

Symbols

\[ n \] : #vertices,
\[ m \] : #edges,
\[ d_v \] : degree of vertex \( v \),
\[ N_v \] : neighbors (adj. array) of vertex \( v \),
\[ \bar{N}_v \] : maximum among \( N_v \)
Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

Symbols

\( n \) : \#vertices,
\( m \) : \#edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \bar{N}_v \) : maximum among \( N_v \)
Log (Vertex labels), Log (Edge weights)

Lower bounds (global) \[ \log n \]

Symbols

- \( n \) : \#vertices,
- \( m \) : \#edges,
- \( d_v \) : degree of vertex \( v \),
- \( N_v \) : neighbors (adj. array) of vertex \( v \),
- \( \overline{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global) \([\log n]\)

This is it? Not really 😊

Symbols

- \(n\) : \#vertices,
- \(m\) : \#edges,
- \(d_v\) : degree of vertex \(v\),
- \(N_v\) : neighbors (adj. array) of vertex \(v\),
- \(\overline{N}_v\) : maximum among \(N_v\)
1. **Log (Vertex labels), Log (Edge weights)**

Lower bounds (global)

[log \( n \)]

This is it? Not really 😊

Lower bounds (local)

**Symbols**

\[ n : \#\text{vertices}, \]
\[ m : \#\text{edges}, \]
\[ d_v : \text{degree of vertex } v, \]
\[ N_v : \text{neighbors (adj. array) of vertex } v, \]
\[ \overline{N}_v : \text{maximum among } N_v \]
1 Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
[log n]

This is it? Not really 😊

Lower bounds (local)
Assume:

Symbols

\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \overline{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
\[ \log n \]

This is it? Not really 😊

Lower bounds (local)
Assume:
- a graph, e.g., \( V = \{1, \ldots, 2^{22}\} \)

Symbols
- \( n \) : \#vertices,
- \( m \) : \#edges,
- \( d_v \) : degree of vertex \( v \),
- \( N_v \) : neighbors (adj. array) of vertex \( v \),
- \( \bar{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

[log \( n \)]

This is it? Not really 😊

Lower bounds (local)

Assume:
- a graph, e.g., \( V = \{1, \ldots, 2^{22}\} \)
- A vertex \( v \) with few neighbors: \( d_v \ll n \)

Symbols

\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \overline{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

[log n]

This is it? Not really 😊

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: $\overline{N}_v \ll n$

Symbols

$n$ : #vertices,
$m$ : #edges,
$d_v$ : degree of vertex $v$,
$N_v$ : neighbors (adj. array) of vertex $v$,
$\overline{N}_v$ : maximum among $N_v$
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

[log n]

This is it? Not really 😊

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: $\overline{N}_v \ll n$

Symbols

- $n$ : #vertices,
- $m$ : #edges,
- $d_v$ : degree of vertex $v$,
- $N_v$ : neighbors (adj. array) of vertex $v$,
- $\overline{N}_v$ : maximum among $N_v$
Lower bounds (global)

\[ \log n \]

This is it? Not really 😊

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: \( \overline{N_v} \ll n \)

\[ \left\lfloor \log 2^{22} \right\rfloor = 22 \]
Lower bounds (global)

\[ \log n \]

This is it? Not really 😊

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: \( \widetilde{N}_v \ll n \)

\[ \log 2^{22} = 22 \]

Symbols

\[ n : \text{#vertices}, \]
\[ m : \text{#edges}, \]
\[ d_v : \text{degree of vertex } v, \]
\[ N_v : \text{neighbors (adj. array) of vertex } v, \]
\[ \widetilde{N}_v : \text{maximum among } N_v \]
1. \[ \log (\text{Vertex labels}), \log (\text{Edge weights}) \]

Lower bounds (global)

\[ \log n \]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

\[ \log 2^{22} = 22 \]

\[ \hat{N}_v \]

19 zeros!
1 Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
\[ \log n \]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

\[ \log 2^{22} = 22 \]

Thus, use the local bound \[ \log \hat{N}_v \]
1. Log (Vertex labels), Log (Edge weights)

This is it? Not really 😊

Symbols

- $n$ : number of vertices,
- $m$ : number of edges,
- $d_v$ : degree of vertex $v$,
- $N_v$ : neighbors (adj. array) of vertex $v$,
- $\overline{N}_v$ : maximum among $N_v$

Lower bounds (local): problem

- 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: $\overline{N}_v \ll n$
1. \( \log(\text{Vertex labels}), \log(\text{Edge weights}) \)

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

Symbols

\[
\begin{align*}
n & : \#\text{vertices}, \\
m & : \#\text{edges}, \\
d_v & : \text{degree of vertex } v, \\
N_v & : \text{neighbors (adj. array) of vertex } v, \\
\hat{N}_v & : \text{maximum among } N_v
\end{align*}
\]
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: \( \widetilde{N}_v \ll n \)
- ...one neighbor has a large ID:

Symbols

\begin{align*}
\text{n} & : \#\text{vertices}, \\
\text{m} & : \#\text{edges}, \\
\text{d}_v & : \text{degree of vertex } v, \\
\text{N}_v & : \text{neighbors (adj. array) of vertex } v, \\
\text{\widetilde{N}_v} & : \text{maximum among } N_v
\end{align*}
Vertex labels

Log ( ), Log ( )

Edge labels

This is it?
Not really 😊

Symbols

\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \overline{N}_v \) : 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: \( \overline{N}_v \ll n \)
- ...one neighbor has a large ID:

\[
\begin{array}{c}
\text{v} \\
2 & 3 & 4 & 5 & \text{1M}
\end{array}
\]

\[
\begin{array}{c}
\text{v} \\
0...10 & 0...11
\end{array}
\]

\[
\begin{array}{c}
\text{v} \\
0...100 & 0...101
\end{array}
\]
1. Log (Vertex labels), Log (Edge weights)

This is it? Not really 😊

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:

$$[\log 2^{20}] = 20$$

Symbols

$n$ : #vertices,
$m$ : #edges,
$d_v$ : degree of vertex $v$,
$N_v$ : neighbors (adj. array) of vertex $v$,
$\widehat{N}_v$ : maximum among $N_v$
1 Log (Vertex labels), Log (Edge weights)

This is it? Not really 😊

Lower bounds (local): problem

What if:
- A graph, e.g., $V = \{1, ..., 2^{22}\}$
- 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:

$$\left\lfloor \log 2^{20} \right\rfloor = 20$$

Symbols

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

This is it? Not really 😊
1 Log (Vertex labels), Log (Edge weights)

Symbols

\( n \) : \#vertices,
\( m \) : \#edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \overline{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

...Use Integer Linear Programming (ILP)!

**Symbols**

- $n$: #vertices,
- $m$: #edges,
- $d_v$: degree of vertex $v$,
- $N_v$: neighbors (adj. array) of vertex $v$,
- $\overline{N}_v$: maximum among $N_v$
1. Log (Vertex labels), Log (Edge weights)

...Use Integer Linear Programming (ILP)!

Lower bounds (local) enhanced with ILP

Symbols

- $n$: #vertices,
- $m$: #edges,
- $d_v$: degree of vertex $v$,
- $N_v$: neighbors (adj. array) of vertex $v$,
- $\overline{N}_v$: maximum among $N_v$
Lower bounds (local) enhanced with ILP

Permute vertex labels to reduce such maximum labels in as many neighborhoods as possible
**Log (Vertex labels), Log (Edge weights)**

...Use **Integer Linear Programming (ILP)**!

**Lower bounds (local) enhanced with ILP**

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

\[ \nu \rightarrow 2, 3, 4, 5, 1M \]

**Symbols**

\[ n : \#\text{vertices}, \]
\[ m : \#\text{edges}, \]
\[ d_v : \text{degree of vertex } v, \]
\[ N_v : \text{neighbors (adj. array) of vertex } v, \]
\[ \overline{N}_v : \text{maximum among } N_v \]
Lower bounds (local) enhanced with ILP

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

Symbols

- $n$: number of vertices,
- $m$: number of edges,
- $d_v$: degree of vertex $v$,
- $N_v$: neighbors (adj. array) of vertex $v$,
- $\overline{N}_v$: maximum among $N_v$
1 Log (Vertex labels), Log (Edge weights)

...Use Integer Linear Programming (ILP)!

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)
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)

\[ \leq 100? \]

- **Log** (Vertex labels), **Log** (Edge weights)
- Use Integer Linear Programming (ILP)!

Symbols:

\[ n : \text{#vertices}, \quad m : \text{#edges}, \quad d_v : \text{degree of vertex } v, \quad N_v : \text{neighbors (adj. array) of vertex } v, \quad \overline{N}_v : \text{maximum among } N_v \]
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 ) = ?? ?? ?? ?? ?? ≤ 100?

Heuristics:
\[ \min_{v \in V} \sum \frac{\bar{N}_v}{d_v} \frac{1}{d_v} \]

Symbols

- \( n \) : #vertices,
- \( m \) : #edges,
- \( d_v \) : degree of vertex \( v \),
- \( N_v \) : neighbors (adj. array) of vertex \( v \),
- \( \bar{N}_v \) : 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] ) = [?] [?] [?] [?] [?] [?] [?] ≤ 100?

Heuristics:
\[
\min \sum_{v \in V} \frac{1}{N_v} \frac{1}{d_v}
\]

Inverse of the neighborhood size
1. Log (Vertex labels), Log (Edge weights)

...Use Integer Linear Programming (ILP)!

Lower bounds (local) enhanced with ILP

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

Intuition: maximum labels in new neighborhoods will be smaller

Heuristics:
\[
\min \sum_{v \in V} \frac{\overline{N}_v}{d_v}
\]

Symbols

\[n \,: \text{#vertices,} \quad m \,: \text{#edges,} \quad d_v \,: \text{degree of vertex } v, \quad N_v \,: \text{neighbors (adj. array) of vertex } v, \quad \overline{N}_v \,: \text{maximum among } N_v\]
1. Log (Vertex labels), Log (Edge weights)

Formal analyses

Symbols

\( \hat{W} \) : max edge weight,

\( n \) : #vertices,

\( p, \alpha, \beta \) : constants
1 Log (Vertex labels), Log (Edge weights)

Formal analyses

Symbols

\( \hat{W} \): max edge weight,
\( n \): #vertices,
\( p, \alpha, \beta \): constants

Power-law graphs

Random uniform graphs
1. Log (Vertex labels), Log (Edge weights)

Formal analyses

Power-law graphs
The probability that a vertex has degree $d$ is:

$$\alpha d^\beta$$

Symbols

$\hat{W}$ : max edge weight,

$n$ : #vertices,

$p, \alpha, \beta$ : constants

Random uniform graphs
1. **Log (Vertex labels), Log (Edge weights)**

Formal analyses

---

**Power-law graphs**

The probability that a vertex has degree \( d \) is:

\[
\alpha d^\beta
\]

---

**Random uniform graphs**

---

**Symbols**

\( \hat{W} \) : max edge weight, 
\( n \) : #vertices, 
\( p, \alpha, \beta \) : constants
1. Log (Vertex labels), Log (Edge weights)

Formal analyses

Power-law graphs
The probability that a vertex has degree \( d \) is:
\[ \alpha d^\beta \]

Random uniform graphs
The probability that a vertex has degree \( d \) is:
\[ pd \]

Symbols
\( \tilde{W} \) : max edge weight,
\( n \) : #vertices,
\( p, \alpha, \beta \) : constants
**Formal analyses**

**1. Log (Vertex labels), Log (Edge weights)**

**Power-law graphs**

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

$$\alpha d^\beta$$

**Random uniform graphs**

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

$$pd$$

**Symbols**

- $\hat{W}$: max edge weight,
- $n$: #vertices,
- $p, \alpha, \beta$: constants
1. Log (Vertex labels), Log (Edge weights)

Formal analyses

Power-law graphs
The probability that a vertex has degree $d$ is:

$$\alpha d^\beta$$

Expected size of the adjacency array

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

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
1 Log (Vertex labels), Log (Edge weights)

Formal analyses: more (check the paper 😊)
Log (Vertex labels), Log (Edge weights)

Formal analyses: more (check the paper 😊)

\[ |\mathcal{A}| = \sum_{v \in V} \left( d_v \left[ \log \hat{N}_v \right] + \left[ \log \log \hat{N}_v \right] \right) \]

\[ |\mathcal{A}| = n \left[ \log \frac{n}{\mathcal{H}} \right] + \mathcal{H} \left[ \log \mathcal{H} \right] \]

\[ E[|\mathcal{O}|] = n \left[ \log \left( 2pn^2 \right) \right] = n \left[ \log 2p + 2 \log n \right] \]

\[ \forall v, u \in V \ (u \in N_v) \Rightarrow \left[ \mathcal{N}(u) \leq \hat{N}_v \right] \]

\[ |\mathcal{A}| = 2m \left( \left[ \log n \right] + \left[ \log \hat{W} \right] \right) \]

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

\[ E[|\mathcal{A}|] = \left( \left[ \log n \right] + \left[ \log \hat{W} \right] \right) pn^2 \]
**1. Log (Vertex labels), Log (Edge weights)**

Formal analyses: more (check the paper 😊)

\[
|\mathcal{A}| = \sum_{v \in V} \left( d_v \left[ \log \hat{N}_v \right] + \left[ \log \log \hat{N}_v \right] \right)
\]

\[
|\mathcal{A}| = n \left[ \log \frac{n}{\mathcal{H}} \right] + \mathcal{H} \left[ \log \mathcal{H} \right]
\]

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

\[
E[|\mathcal{O}|] = n \left[ \log \left( 2pn^2 \right) \right] = n \left[ \log 2p + 2 \log n \right]
\]

A Cray XE/XT supercomputer...
1. Log(Vertex labels), Log(Edge weights)

Key methods
1 Log (Vertex labels), Log (Edge weights)

Key methods

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits
Key methods

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits

```c
/* v_ID is an opaque type for IDs of vertices. */
v_ID N_{i,v}(v_ID v, int32_t i, int64_t* O, int64_t* A, int8_t s){
  int64_t exactBitOffset = s * (O[v] + i);
  int8_t* address = (int8_t*) A + (exactBitOffset >> 3);
  int64_t distance = exactBitOffset & 7;
  int64_t value = ((int64_t*) (address))[0];
  return _bextr_u64(value, distance, s);
}
```
**Key methods**

Return $i$-th neighbor of vertex $v$.

```
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); }
```
1. Log(Vertex labels), Log(Edge weights)

Key methods

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits

Return i-th neighbor of vertex v

Pointer to the offset array

```c
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* 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); }
```
Vertex labels

Edge weights

Key methods

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits

Return $i$-th neighbor of vertex $v$

```
1 /* v_ID is an opaque type for IDs of vertices. */
2 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, $s$); }
```
# Key methods

Return $i$-th neighbor of vertex $v$

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits

```c
1 /* v_ID is an opaque type for IDs of vertices. */
2 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, $s$);
```
1. **Log (Vertex labels), Log (Edge weights)**

Key methods

- Return $i$-th neighbor of vertex $v$
- Derive exact offset (in bits) to the neighbor label
- Pointer to the offset array
- Pointer to the adjacency array
- $s = \lfloor \log n \rfloor$

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits.

```c
1 /* v_ID is an opaque type for IDs of vertices. */
2 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); }
```
Key methods

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits

Return $i$-th neighbor of vertex $v$

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

Get the closest byte alignment

```
/* v_ID is an opaque type for IDs of vertices. */
v_ID N_{i,v}(v_ID v, int32_t i, int64_t* O, int64_t* A, int8_t* s){
  int64_t exactBitOffset = s * (O[v] + i);
  int8_t* address = (int8_t*) A + (exactBitOffset >> 3);
  int64_t distance = exactBitOffset & 7;
  int64_t value = ((int64_t*) (address))[0];
  return _bextr_u64(value, distance, s); }
```
**Key methods**

Return $i$-th neighbor of vertex $v$

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array $s = \lfloor \log n \rfloor$

Get the closest byte alignment

Get the distance from the byte alignment

---

```
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 \gg 3));
5   \int64_t \ distance = exactBitOffset \& 7;
6   \int64_t \ value = ((\int64_t^*)\ (address))[0];
7   return _bextr_u64(value, distance, s); }
```
**Key methods**

- **Log (Vertex labels)**, **Log (Edge weights)**

Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits.

Return $i$-th neighbor of vertex $v$

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

$s = [\log n]$ Get the closest byte alignment

Get the distance from the byte alignment

Access the derived 64-bit value

```c
/* v_ID is an opaque type for IDs of vertices. */
v_ID N_{i,v}(v_ID v, int32_t i, int64_t* O, int64_t* A, int8_t* s){
  int64_t exactBitOffset = s * (O[v] + i);
  int8_t* address = (int8_t*) (A + (exactBitOffset >> 3));
  int64_t distance = exactBitOffset & 7;
  int64_t value = (*((int64_t*) (address))[0];
  return _bextr_u64(value, distance, s); }
```
Use the BEXTR bitwise operation to help extract an arbitrary sequence of bits.

Return $i$-th neighbor of vertex $v$

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array $s = \lceil \log n \rceil$

Get the closest byte alignment

Get the distance from the byte alignment

Shift the derived 64-bit value by $d$ bits and mask it with BEXTR

Access the derived 64-bit value

Key methods

Return $i$-th neighbor of vertex $v$

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array $s = \lceil \log n \rceil$

Get the closest byte alignment

Get the distance from the byte alignment

Shift the derived 64-bit value by $d$ bits and mask it with BEXTR

Access the derived 64-bit value

Key methods

Log (Vertex labels), Log (Edge weights)
2 Log ( Offset structure )
2 Log ( Offset structure )
Use a **bit vector** instead of an array of offsets...
Use a **bit vector** instead of an array of offsets...

Bit vectors instead of offset arrays
Use a **bit vector** instead of an array of offsets...

**Bit vectors instead of offset arrays**

1 2 0 3 0 3 1 2 4 3 5 4

0 2 4 6 9 11
Log (Offset structure)

Use a **bit vector** instead of an array of offsets...

Bit vectors instead of offset arrays:

```
1 2 0 3 0 3 1 2 4 3 5 4
0 2 4 6 9 11
```
Use a **bit vector** instead of an array of offsets...

**Bit vectors instead of offset arrays**

```
1 2 | 0 3 | 0 3 | 1 2 4 | 3 5 | 4
```

101010100101
Use a **bit vector** instead of an array of offsets...

**Bit vectors instead of offset arrays**

```
0 1 2 3 0 3 1 2 4 3 5 4
```

The adjacency array of a vertex $i$ starts at a word $x$. 

101010100101

$i$-th set bit has a position $x$
Use a **bit vector** instead of an array of offsets...

Bit vectors instead of offset arrays

```
1 2 0 3 0 3 1 2 4 3 5 4
```

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

```
101010100101
```

i-th set bit has a position x → the adjacency array of a vertex i starts at a word x

Offset structure

Log (Offset structure)
2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors [1]

Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

Succinct bit vectors

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

Succinct bit vectors

They use $\lceil Q \rceil + o(Q)$ bits ($\lceil Q \rceil$ - lower bound), they answer various queries in $o(Q)$ time. = small + fast (hopefully) [1]

Succinct bit vectors

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

10101010010100010101011110000001100001…

2 **Log ( Offset structure )**

 Encode the resulting bit vectors as **succinct bit vectors** [1]

**Succinct bit vectors**

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

$n$ bits: 101010100101000101010111110000001100001...

Succinct bit vectors

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

$10101010010100000001000001100001\ldots$

2 Log (Offset structure)  

...Encode the resulting bit vectors as succinct bit vectors [1]  

Succinct bit vectors  

They use ⌈Q⌉ + o(Q) bits (⌈Q⌉ - lower bound), they answer various queries in o(Q) time. = small + fast (hopefully)  

Succinct bit vectors: They use $[Q] + o(Q)$ bits ($[Q]$ - lower bound), they answer various queries in $o(Q)$ time. = small + fast (hopefully)

$n$ bits: 1010101001010001010101000000001100001...

$\log^2 n = t_1$  
$\log^2 n$  
$\log^2 n$  

Succinct bit vectors

They use $[Q] + o(Q)$ bits ($[Q]$ - lower bound), they answer various queries in $o(Q)$ time. = small + fast (hopefully)

$n$ bits

$$\log^2 n = t_1$$

\[ 10101010010\ldots \]
Succinct bit vectors

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

Encode the resulting bit vectors as succinct bit vectors [1]

$n$ bits

$10101010010101011111100000011000011...$

$\frac{1}{2}\log n$

$\log^2 n = t_1$

$\log^2 n$

$\log^2 n$

2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

= small + fast (hopefully)

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

\[
\log^2 n = t_1
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n = t_2
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n
\]

... = small + fast (hopefully)

Succinct bit vectors

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

Compute & store the number of 1s

\[
\log^2 n = t_1 \\
\log^2 n \quad \log^2 n \quad \log^2 n \\
10101010010100101010111110000011000001\ldots
\]

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n \\
\frac{1}{2} \log n \quad \frac{1}{2} \log n \\
\frac{1}{2} \log n \quad \frac{1}{2} \log n \\
\]

$= t_2$

Log (Offset structure)

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

\[ \log^2 n = t_1 \]

\[ \log^2 n \]

Compute & store the number of 1s

\[ \frac{1}{2} \log n \] \[ \frac{1}{2} \log n \]

Compute & store the number of 1s

\[ \frac{1}{2} \log n \] \[ \frac{1}{2} \log n \] \[ \frac{1}{2} \log n \] \[ \frac{1}{2} \log n \]

\[ 10101010010100010101011110000001100001 \ldots \]

…

2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

Compute & store the number of 1s

$n$ bits

$101010100101000101010111110000001100001…$

$\log^2 n = t_1$

$\log^2 n$

$\log^2 n$

$\log^2 n$

Compute & store the number of 1s

$\frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n$

$\frac{1}{2} \log n \quad \frac{1}{2} \log n$

$t_2$

$t_1$

$n = t_1$

$t_2$

$= small + fast$

(hopefully)

2 Log (Offset structure)

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

Compute & store the number of 1s

\[ n \log n = t_1 \]

\[ \frac{1}{2} \log n = t_2 \]

Compute & store the number of 1s

\[ n \log n = t_1 \]

\[ \frac{1}{2} \log n = t_2 \]

\[ o(n) \]

\[ o(n) \]

\[ o(n) \]

\[ o(n) \]

\[ o(n) \]

\[ o(n) \]

\[ o(n) \]

2 Offset structure

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

$n$ bits

$10101010010100010101011111100000011000001...$

Compute & store the number of 1s

$\log^2 n = t_1$

$\frac{1}{2} \log n \cdot \frac{1}{2} \log n = t_2$

Compute & store the number of 1s

$\log^2 n = o(n)$

$t_2 \log t_1 = o(n)$

$t_1 \log \frac{n}{t_1} = o(n)$

$\frac{n \log \log n}{\log n} = o(n)$

2 Log (Offset structure)

...Encode the resulting bit vectors as succinct bit vectors [1]

Succinct bit vectors

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

Total storage:
$n + o(n) + o(n) + \cdots = n + o(n)$

Compute & store the number of 1s

$\log^2 n = t_1$

$n$ bits

$\frac{1}{2} \log n \frac{1}{2} \log n = t_2$

$1 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 \cdots$

$\log^2 n$

$\log^2 n$

$\log^2 n$

Compute & store the number of 1s

$= O \left( \frac{n}{t_1 \log n} \right) = O \left( \frac{n}{\log n} \right) = o(n)$

Compute & store the number of 1s

$= O \left( \frac{n}{t_2 \log t_1} \right) = O \left( \frac{n \log \log n}{\log n} \right) = o(n)$

$n + o(n) + o(n) + \cdots = n + o(n)$

Log (Offset structure)

...Encode the resulting bit vectors as succinct bit vectors
2. $\text{Log ( Offset structure )}$

...Encode the resulting bit vectors as succinct bit vectors

Formal analyses
Formal analyses

<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>$O(\frac{Wm}{B})$</td>
<td>$2Wm\frac{1}{B}$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Interleaved [44]</td>
<td>bvIL</td>
<td>$O(\frac{Wm}{B} + \frac{Wm}{L})$</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>$\approx \log \frac{2Wm}{Bn}$</td>
<td>$O\left(\log \frac{Wm}{B}\right)$</td>
</tr>
<tr>
<td>Sparse [76]</td>
<td>bvSD</td>
<td>$O\left(n + n \log \frac{Wm}{Bn}\right)$</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>
### Formal analyses

<table>
<thead>
<tr>
<th>$\mathcal{O}$</th>
<th>ID</th>
<th>Asymptotic size [bits]</th>
<th>Exact size [bits]</th>
<th>select or $\mathcal{O}[v]$</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>$2Wm$</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>$O\left(n + n \log \frac{Wm}{Bn}\right)$</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>

...Encode the resulting bit vectors as succinct bit vectors

Check the paper for details 😊
2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors

Formal analyses

<table>
<thead>
<tr>
<th>Offset structure</th>
<th>Log</th>
<th>Exponent size [bits]</th>
<th>Exact size [bits]</th>
<th>select or $\mathcal{O}[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer array</td>
<td>$\text{ptrW}$</td>
<td>$O(Wn)$</td>
<td>$W(n+1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Plain [44]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interleaved [44]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entropy based [3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse [76]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-tree based [1]</td>
<td>$\text{BVT}$</td>
<td>$O\left( \frac{W_m}{B} \right)$</td>
<td>$\approx 1.1 \cdot \frac{W_m}{B}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Gap-compressed [1]</td>
<td>$\text{bvGC}$</td>
<td>$O\left( \frac{W_m}{B} \log \frac{W_m}{Bn} \right)$</td>
<td>$\approx 1.3 \cdot \frac{2W_m}{B} \log \frac{2W_m}{Bn}$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

We will show that some are in practice both small and fast!
3 Log ( Adjacency structure )
3 Log ( Adjacency structure )

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

More schemes that assume specific classes of graphs
...
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)

More schemes that assume specific classes of graphs ...

Use different relabelings
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

More schemes that assume specific classes of graphs ...

Use different relabelings
Use different relabelings

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

More schemes that assume specific classes of graphs...

...
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 ...
OVERVIEW OF FULL LOG(GRAPH) DESIGN
OVERVIEW OF FULL LOG(GRAPH) DESIGN

1 Input structures

1.1 Graph $G$ ($\S 2$)

1.2 Adjacency Array ($\S 2$)

$O$, $A$, $A_0$, $A_1$, $A_7$
OVERVIEW OF FULL LOG(GRAPH) DESIGN

1. Input structures
   1.1 Graph $G$ ($\S$2)
   1.2 Adjacency Array ($\S$2)

2. Logarithmize fine elements ($\S$3)
   2.1 Logarithmize vertex IDs... ($\S$3.2)
      Example ID: Remove leading bits (simple bit packing)
      \[
      \log(2) = \log(0...0102) = 0102
      \]
   2.2 Logarithmize vertex IDs... ($\S$3.2)
   2.3...globally ($\S$3.2.1)
   2.4...locally ($\S$3.2.2)
   2.5...on DM ($\S$3.2.3) systems
   2.6 Logarithmize other elements ($\S$3.3 - $\S$3.4)
   2.7 ($\S$3.5) Analyze
   2.8 ($\S$3.6) Ensure high
   2.9 Use difference encoding ($\S$3.7)
   2.10 ($\S$3.8) Ensure
OVERVIEW OF FULL LOG(GRAPH) DESIGN

1. Input structures
   1.1 Graph $G$ (§2)
   1.2 Adjacency Array (§2)

2. Logarithmize fine elements (§3)
   2.1 Logarithmize vertex IDs... (§3.2)
      Example ID
      \[
      \text{Log}(2) = \text{Log}(0010_2) = 0102
      \]
   2.2 Logarithmize globally (§3.2.1)
   2.3 Logarithmize locally (§3.2.2)
   2.4 Logarithmize on DM (§3.2.3) systems
   2.5 Logarithmize other elements (§3.3 - §3.4)

3. Logarithmize offset structure (§4)
   3.1 Understand storage lower
   3.2 Incorporate succinctness
   3.3 Theory (§4.4)
   3.4 (§4.5) Performance improvements

4. High-performance extensible library (§6)

Use Integer Linear Programming

Log(01010101010101010101010101010101)...
OVERVIEW OF FULL LOG(GRAPH) DESIGN

1. Input structures
   - Graph $G$ ($\S 2$)

2. Logarithmize fine elements ($\S 3$)
   - Logarithmize vertex IDs... ($\S 3.2$)
     - Example ID
     - Remove leading bits (simple bit packing)
     - $\log(2) = \log(0\ldots010_2) = 010_2$
   - Logarithmize globally ($\S 3.2.1$)
   - Logarithmize locally ($\S 3.2.2$)
   - Logarithmize on DM ($\S 3.2.3$)

3. Logarithmize offset structure ($\S 4$)
   - Logarithmize of $O$ (§3.1)
   - Understand storage lower bounds
   - Incorporate succinctness
     - $\log(O_{\log-19})$
     - Use $OPT+o(OPT)$ space
   - Incorporate recursive bisectioning
     - Use DM ($\S 5.4$)

4. Logarithmize adjacency structure ($\S 5$)
   - Unify $P+T$ ($\S 5.2$)
   - Incorporate recursive bisectioning
     - Log($A_{2,4,5,6}$)

5. High-performance extensible library ($\S 6$)
   - Use ILP
   - This part is covered in the extended technical report version of the paper

6. Use Integer Linear Programming (ILP)
OVERVIEW OF FULL $\log(G)$ DESIGN

1. Input structures
   1.1 Graph $G$ ($\S$2)
   1.2 Adjacency Array ($\S$2)

2. Logarithmize fine elements ($\S$3)
   2.2 Logarithmize vertex IDs... ($\S$3.2)
      Example ID
      \[ \log(2) = \log(0...010_2) = 010_2 \]
   2.3...globally ($\S$3.2.1)
   2.4...locally ($\S$3.2.2)
   2.5...on DM ($\S$3.2.3)

3. Logarithmize offset structure ($\S$4)
   3.1 ($\S$4.2) Understand storage lower bounds
   3.2 Incorporate ($\S$4.3) succinctness
   \[ \log(O_1 + \cdots + 19) \]
   3.3 Incorporate ($\S$5.3) recursive bisectioning
   3.4 ($\S$4.5) performance implementation

4. Logarithmize adjacency structure ($\S$5)
   4.1 ($\S$5.1) Unify with $P + T$
   4.2 Logarithmize ($\S$5.2)
   4.3 Incorporate ($\S$5.3) recursive bisectioning
   4.4 ($\S$5.3.1) Use DM
   4.5 ($\S$5.3.2) ...use RB
   4.6 ($\S$5.3.3) ...use BRB

5. High-performance extensible library ($\S$6)
   5.1 Use ILP
   5.2 This part is covered in the extended technical report version of the paper
OVERVIEW OF FULL LOG(GRAPH) DESIGN

1. Input structures
   1.1 Graph $G$ ($\S2$)
   1.2 Adjacency Array ($\S2$)

2. Logarithmize
   2.2 Logarithmize vertex IDs... ($\S3.2$)
      Example ID
      Remove leading bits (simple bit packing)
   2.3...globally ($\S3.2$)
   2.4...locally ($\S3.2$)
   2.5...on DM ($\S3.2$)
   2.6 Logarithmize other elements ($\S3.3 - \S3.4$)

3. Logarithmize
   3.1 offset structure ($\S4$)
      Understand storage lower bounds
   3.2 Incorporate ($\S4.3$)

4. Logarithmize
   4.1 adjacency structure ($\S5$)
      Incorporate ($\S5.3$)
      Recursive bisectioning
   4.2 unify $\mathcal{P} + \mathcal{T}$ ($\S5.2$)
   4.3...DM ($\S5.4$)
   4.4...use RB
   4.5...use BRB
   4.6...ILP

5. High-performance
   5.1 extensible library ($\S6$)

---

Looks complex 😊
Overview of Full Log(Graph) Design

Looks complex 😊

We analyzed / implemented (in total):
- 6 schemes for compressing fine elements,
- 10+ schemes for compressing offset structures,
- 4+ schemes for compressing adjacency structures
OVERVIEW OF FULL LOG(GRAPH) DESIGN

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
OVERVIEW OF FULL LOG(GRAPH) DESIGN

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

Looks complex 😊

We analyzed / implemented (in total):
- 6 schemes for compressing fine elements,
- 10+ schemes for compressing offset structures,
- 4+ schemes for compressing adjacency structures

...they all can be arbitrarily combined.
OVERVIEW OF FULL LOG(GRAPH) DESIGN

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

Looks complex 😊

We analyzed / implemented (in total):
- 6 schemes for compressing fine elements,
- 10+ schemes for compressing offset structures,
- 4+ schemes for compressing adjacency structures

... they all can be arbitrarily combined.

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

TYPES OF MACHINES

CSCS Cray Piz Daint
PERFORMANCE ANALYSIS

TYPES OF MACHINES

CSCS Cray Piz Daint
PERFORMANCE ANALYSIS

TYPES OF MACHINES

CSCS Cray Piz Daint

HP “fat server” (DL360)
PERFORMANCE ANALYSIS
TYPES OF GRAPHS
PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs
PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]

PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]

Erdős-Rényi [2]

PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Real-world graphs (SNAP [3], KONECT [4], Webgraph [5], DIMACS [6])

Synthetic graphs

- Kronecker [1]
- Erdős-Rényi [2]


PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Real-world graphs (SNAP [3], KONECT [4], Webgraph [5], DIMACS [6])

Synthetic graphs

Kronecker [1]

Erdös-Rényi [2]

Road networks

Social networks

Web graphs

Purchase networks

Communication graphs

Citation graphs

[5] DIMACS Challenge
PERFORMANCE ANALYSIS
ALGORITHMS
PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])
PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])


PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])


PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])


Performance Analysis Algorithms

Connected Components (Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

PERFORMANCE ANALYSIS ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Triangle Counting

**PERFORMANCE ANALYSIS ALGORITHMS**

Connected Components (Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Triangle Counting

---


PERFORMANCE ANALYSIS

ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Triangle Counting


**PERFORMANCE ANALYSIS**

**ALGORITHMS**

- **Connected Components** (Shiloach-Vishkin [1])
- **BFS** (direction optimization [2])
- **SSSP** (Delta-Stepping [3])
- **Triangle Counting**

---


PERFORMANCE ANALYSIS ALGORITHMS

Connected Components (Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Triangle Counting

PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Triangle Counting

**PERFORMANCE ANALYSIS ALGORITHMS**

- Connected Components (Shiloach-Vishkin [1])
- BFS (direction optimization [2])
- SSSP (Delta-Stepping [3])
- Triangle Counting
- Betweenness Centrality (Brandes’ Algorithm [4])

---

**PERFORMANCE ANALYSIS ALGORITHMS**

- **Connected Components** (Shiloach-Vishkin [1])
- **BFS** (direction optimization [2])
- **SSSP** (Delta-Stepping [3])
- **Triangle Counting**
- **Betweenness Centrality** (Brandes’ Algorithm [4])

References:

**Performance Analysis of Algorithms**

- Connected Components (Shiloach-Vishkin [1])
- BFS (Direction Optimization [2])
- SSSP (Delta-Stepping [3])
- Triangle Counting
- Betweenness Centrality (Brandes’ Algorithm [4])

---


**Performance Analysis of Algorithms**

- **Connected Components** (Shiloach-Vishkin [1])
- **BFS** (direction optimization [2])
- **SSSP** (Delta-Stepping [3])
- **Triangle Counting**
- **Betweenness Centrality** (Brandes’ Algorithm [4])

**Performance Analysis of Algorithms**

- **Connected Components** (Shiloach-Vishkin [1])
- **BFS** (direction optimization [2])
- **SSSP** (Delta-Stepping [3])
- **PageRank** (variant with no atomics)
- **Triangle Counting**
- **Betweenness Centrality** (Brandes’ Algorithm [4])

---


**Performance Analysis Algorithms**

- **Connected Components** (Shiloach-Vishkin [1])
- **BFS** (direction optimization [2])
- **SSSP** (Delta-Stepping [3])
- **PageRank** (variant with no atomics)
- **Triangle Counting**
- **Betweenness Centrality** (Brandes’ Algorithm [4])

---


**PERFORMANCE ANALYSIS**

**ALGORITHMS**

Connected Components (Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

PageRank (variant with no atomics)

Betweenness Centrality (Brandes’ Algorithm [4])

---


PERFORMANCE ANALYSIS
ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

PageRank (variant with no atomics)

Betweenness Centrality
(Brandes’ Algorithm [4])

Triangle Counting

PERFORMANCE ANALYSIS

COMPARISON TARGETS
PERFORMANCE ANALYSIS

COMPARISON TARGETS

GAPBS: Graph Algorithm Platform Benchmark Suite [1]. Comparison to a traditional adjacency array implementation

PERFORMANCE ANALYSIS

COMPARISON TARGETS

Zlib [2].
Comparison to a traditional compression scheme

GAPBS: Graph Algorithm Platform Benchmark Suite [1].
Comparison to a traditional adjacency array implementation

PERFORMANCE ANALYSIS

COMPARISON TARGETS

Zlib [2].
Comparison to a traditional compression scheme

GAPBS: Graph Algorithm Platform Benchmark Suite [1].
Comparison to a traditional adjacency array implementation

WebGraph Library [3]
Comparison to a state-of-the-art graph compression scheme

PERFORMANCE ANALYSIS
COMPARISON TARGETS

Zlib [2].
Comparison to a traditional compression scheme

GAPBS: Graph Algorithm Platform Benchmark Suite [1].
Comparison to a traditional adjacency array implementation

WebGraph Library [3]
Comparison to a state-of-the-art graph compression scheme

Recursive Partitioning [4].
Comparison to a tuned scheme for compressing adjacency data

1. Log (Vertex labels), Log (Edge weights) Storage, Performance

Kronecker graphs
Number of vertices: 4M

SSSP
1. Log (Vertex labels), Log (Edge weights) Storage, Performance

![Graph](image)

Kronecker graphs
Number of vertices: 4M

**Scheme:**
- GAPBS
- Log(Graph)

**Time [s]:**
- 64.0
- 128.0
- 256.0
- 512.0
- 1024.0

**Number of edges per vertex**
1 Log (Vertex labels), Log (Edge weights) Storage, Performance

Log(Graph) consistently reduces storage overhead (by 20-35%)
1 Log (Vertex labels), Log (Edge weights) Storage, Performance

Log(Graph) accelerates GAPBS

Log(Graph) consistently reduces storage overhead (by 20-35%)

Number of vertices: 4M
### Log (Vertex labels), Log (Edge weights) Storage, Performance

- **Kronecker graphs**
  - Number of vertices: 4M

- **Log(Graph) accelerates GAPBS**
  - Both storage and performance are improved *simultaneously*

- **Log(Graph) consistently reduces storage overhead** (by 20-35%)

---

### Graph Scheme

<table>
<thead>
<tr>
<th>Number of edges per vertex</th>
<th>GAPBS</th>
<th>Log(Graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Time [s]**

- 0.0
- 0.5
- 1.0
- 1.5
- 2.0
- 2.5

---

**Log(Graph)**

- Consistently reduces storage overhead (by 20-35%)
2 Log ( Offset structure ) Storage
Various real-world graphs
Various real-world graphs

Lots of data 😊

Conclusions:
Various real-world graphs

Lots of data 😊

Conclusions:
Lots of data 😊

Conclusions:

- **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
Various real-world graphs

Offset structure

Log

Storage

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

Lots of data 😊

Conclusions:
Various real-world graphs

Offset structure

Log

Storage

Lots of data 😊

Conclusions:

Succinct bit vectors consistently ensure best storage reductions

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
Lots of data 😊

Conclusions:

Succinct bit vectors consistently ensure best storage reductions

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

The main reason: succinct designs work well for sparse bit vectors, and graphs „that matter” are sparse
Log ( Offset structure ) Performance

Accessing randomly selected neighbors

Kronecker graphs
Number of vertices: 4M
2 Log (Offset structure) Performance

Accessing randomly selected neighbors

Kronecker graphs
Number of vertices: 4M
Accessing randomly selected neighbors

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

Kronecker graphs

Number of vertices: 4M

---

2 \text{Log ( Offset structure ) Performance}

![Diagram showing performance of different offset structures]
2 Log (Offset structure) Performance

Lots of data again 😊 Conclusions:

Accessing randomly selected neighbors

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

Change of performance due to Hyper-Threading (>1 thread per core)
Low latency due to prefetching and compiler optimizations

Kronecker graphs
Number of vertices: 4M
In sequential settings (or settings with low parallelism), simple offset arrays are the fastest.

Lots of data again 😊 Conclusions:

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

**Accessing randomly selected neighbors**

![Graph showing performance comparison]

**Kronecker graphs**

Number of vertices: 4M
In sequential settings (or settings with low parallelism), simple offset arrays are the fastest. Once parallelism overheads kick in, performance of accessing succinct bit vectors and offset arrays becomes comparable.

Lots of data again 😊 Conclusions:

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

Accessing randomly selected neighbors

Kronecker graphs

Number of vertices: 4M
In sequential settings (or settings with low parallelism), simple offset arrays are the fastest.

Once parallelism overheads kick in, performance of accessing succinct bit vectors and offset arrays becomes comparable.

Lots of data again 😊 Conclusions:

- \[ \text{ptr64}: \text{traditional array of offsets} \]
- \[ \text{bvPL}: \text{plain bit vectors} \]
- \[ \text{bvIL}: \text{compact bit vectors} \]
- \[ \text{bvEN, bvSD}: \text{succinct bit vectors} \]
- \[ \text{zlib(.)}: \text{zlib-compressed variants} \]

**Accessing randomly selected neighbors**

- \[ \text{bvSD}: \text{the fastest and (usually) the smallest} \]
3 Log ( Adjacency structure ) Storage, Performance
Various real-world graphs

3 Log (Adjacency structure) Storage, Performane

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs
Adjacency structure Log ( ) Storage, Performance

- **Trad**: Traditional adjacency array
- **DMd / DMf**: Degree Minimizing (without / with gap encoding)
- **WG**: WebGraph compression
- **BRB, RB**: Schemes targeting certain specific classes of graphs

Various real-world graphs

Relative size

Scheme:
- Trad
- DMd
- DMf
- BRB
- RB
- WG
Adjacency structure

Storage, Performance

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs

Lots of data 😊

Conclusions:
WebGraph is best for web graphs.

Lots of data 😊

Conclusions:

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs
**Log (Adjacency structure)**

**Storage, Performance**

- **Trad**: Traditional adjacency array
- **DMd / DMf**: Degree Minimizing (without / with gap encoding)
- **WG**: WebGraph compression
- **BRB, RB**: Schemes targeting certain specific classes of graphs

**Lots of data 😊**

**Conclusions:**

- BRB, RB: various tradeoffs but very expensive preprocessing (details in the paper)
3 Log (Adjacency structure) Storage, Performance

**Trad**: Traditional adjacency array
**DMd / DMf**: Degree Minimizing (without / with gap encoding)
**WG**: WebGraph compression
**BRB, RB**: Schemes targeting certain specific classes of graphs

Lots of data 😊

Conclusions:

- **DMd**: much better than **DMf**, often comparable to **WG**
- **BRB, RB**: various tradeoffs but very expensive preprocessing (details in the paper)

**WebGraph** best for web graphs 😊
Lot of data 😊

**Conclusions:**

- **WebGraph best for web graphs 😊**
- **DMd: much better than DMf, often comparable to WG**
- **BRB, RB: various tradeoffs but very expensive preprocessing (details in the paper)**

**Trad:** Traditional adjacency array

**DMd / DMf:** Degree Minimizing (without / with gap encoding)

**WG:** WebGraph compression

**BRB, RB:** Schemes targeting certain specific classes of graphs
3 Log (Adjacency structure)  Storage, Performance

Trad: Traditional adjacency array  
DMd / DMf: Degree Minimizing (without / with gap encoding)  
WG: WebGraph compression  
BRB, RB: Schemes targeting certain specific classes of graphs

Lots of data 😊

Conclusions:

WebGraph best for web graphs 😊
DMd: much better than DMf, often comparable to WG
BRB, RB: various tradeoffs but very expensive preprocessing (details in the paper)

WebGraph is the slowest, DM somewhat slower than Trad
Takeaway (Results): Log(Graph) ensures Space-Performance sweetspot (tunable!)
Key insight (vertex labels)

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

Takeaway (Results): Log(Graph) ensures Space-Performance sweetspot (tunable!)
Key insight (vertex labels)

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

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!)
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!)
OTHER RESULTS
OTHER RESULTS
WHAT IS LOG(GRAPH)?
A NEAR-OPTIMAL GRAPH REPRESENTATION

What is the lowest storage we can hope to use to store a graph?

Key idea

Encode different parts of a graph representation using logarithmic storage lower bounds

Log (edge weights)
Log (edges adjacent to each vertex)

Adjacency arrays

Log (offsets (locations) of adj. arrays)

What is Log(Graph)?

The storage lower bound

Ω

Counting bounds: They are logarithmic (one needs at least log(|V|) bits to store an object from an arbitrary set |V|)
A NEAR-OPTIMAL GRAPH REPRESENTATION

What is the lowest storage we can hope to use to store a graph?

The storage lower bound

Key idea

Encode different parts of a graph representation using logarithmic storage lower bounds

Log (edges)
Log (weights)

Log (vertices)

Log (labels)

Adjacency arrays (edges adjacent to each vertex)

Counting bounds: They are logarithmic (one needs at least \( \log n \) bits to store an object from an arbitrary set)

\( S = \{ e_1, e_2, \ldots \} \)

\( s_1 = 0 \ldots 0 \)

\( s_2 = 0 \ldots 1 \)

\( s_3 = 0 \ldots 1 \)

\( s_4 = 1 \)

What is \( \log(\text{Graph}) \)?

AN EXTENSIBLE GRAPH REPRESENTATION

Overview of Full Log(Graph) Design

[Diagram showing various components and their relationships]
**WHAT IS LOG(GRAPH)?**

A NEAR-OPTIMAL GRAPH REPRESENTATION

- **The storage lower bound**
  - There are logarithmic (one needs at least \( \log n \) bits to store an object from an arbitrary set \( S \))

**Key idea**
- Encode different parts of a graph representation using logarithmic storage lower bounds

- **Adjacency arrays** (edges adjacent to each vertex)

- **Offsets** (locations) of adj. arrays

AN EXTENSIBLE GRAPH REPRESENTATION

Overview of Full Log(Graph) Design

**A HIGH-PERFORMANCE GRAPH REPRESENTATION**

A comparison of different graph schemes and their performance on Sparse and Dense graphs.

- Log(Graph) accelerates GAPBS

- Both storage and performance are improved simultaneously

Graph schemes include:
- Log(Graph)
- Traditional (non-compressed, GAPBS)
- Log(Graph) with optimization

**Performance metrics**
- Time (s)
- Number of edges per vertex
- Number of vertices

Log(Graph) shows significant advantages in both storage and performance compared to traditional schemes.
WHAT IS LOG(GRAPH)?

A NEAR-OPTIMAL GRAPH REPRESENTATION

What is the lowest storage we can (hope to) use to store a graph?

The storage lower bound

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

Counting bounds: They are logarithmic (one needs at least log (n) bits to store an object from an arbitrary set S)

A HIGH-PERFORMANCE GRAPH REPRESENTATION

A CONDENSED GRAPH REPRESENTATION

A NEAR-OPTIMAL GRAPH REPRESENTATION

Log (|V|, |E|), Log (ω |E|)

Log(Graph) accelerates GAPBS

BFS

S = {v₀, v₁, v₂, ..., vᵢ}

Log (vertices labels)

Log (edge weights)

Log (adjacency arrays)

Log (offsets, locations of siblings)

AN EXTENSIBLE GRAPH REPRESENTATION

Overview of Full Log(Graph) Design

AN EXTENSIBLE GRAPH REPRESENTATION

Log (adjacency structure)

Storage

WGS: WebGraph compression

IBL: IBL: Schemes targeting certain specific classes of graphs

Degree slicing

Degree slicing

WebGraph best for web graphs

DMG: much better than DAG

BRL: BRL: various tradeoffs but very expensive preprocessing (details in the paper)
A NEAR-OPTIMAL GRAPH REPRESENTATION

What is the lowest storage we can hope to use to store a graph?

The storage lower bound \( \Omega \) which is:

- Counting bounds: They are logarithmic (one needs at least \( \log(\Omega) \) bits to store an object from an arbitrary set \( \Omega \))
- Key idea: Log (Vertices labels)Encode different parts of a graph representation using (logarithmic) storage lower bounds
- Log (Edge weights)
- Log (Offsets [locations] of adj. arrays)

WHAT IS LOG(GRAPH)?

AN EXTENSIBLE GRAPH REPRESENTATION

Overview of Full Log(Graph) Design

A HIGH-PERFORMANCE GRAPH REPRESENTATION

A CONDENSED GRAPH REPRESENTATION

Website:

http://spcl.inf.ethz.ch/
Research/
Performance/
LogGraph
A NEAR-OPTIMAL GRAPH REPRESENTATION

What is the lowest storage we can hope to use to store a graph?

The storage lower bound $\Omega$

Key idea

Encode different parts of a graph representation using (logarithmic) storage lower bounds

Counting bounds: They are logarithmic (one needs at least $\log n$ bits to store an object from an arbitrary set $S$)

A NEAR-OPTIMAL GRAPH REPRESENTATION

A HIGH-PERFORMANCE GRAPH REPRESENTATION

A CONDENSED GRAPH REPRESENTATION

WHAT IS LOG(GRAPH)?

A NEAR-OPTIMAL GRAPH REPRESENTATION

A HIGH-PERFORMANCE GRAPH REPRESENTATION

A CONDENSED GRAPH REPRESENTATION

Website:
http://spcl.inf.ethz.ch/Research/Performance/LogGraph
Thank you for your attention.

**What is Log(Graph)?**

A **Near-Optimal Graph Representation**

- What is the lowest storage we can hope to use to store a graph?
- The storage lower bound \( \Omega \)
- Counting bounds: They are logarithmic (one needs at least \( \log n \) bits to store an object from an arbitrary set \( S \))
- \( S = \{x_1, x_2, x_3, \ldots\} \)
  - \( x_1 = 0 \cdots 0 \)
  - \( x_2 = 0 \cdots 1 \)
  - \( x_3 = 0 \cdots 1 \)

A **High-Performance Graph Representation**

- Log (edges, weights)
- Log (edges, labels)
- Log (vertex labels)
- Key idea: Encode different parts of a graph representation using logarithmic storage lower bounds
- Adjacency arrays (edges adjacent to each vertex)
- Offsets (locations) of adj. arrays

A **Condensed Graph Representation**

- Log (adjacency structure)
- Log (adjacency structure)
- Log (adjacency structure)
- Log (adjacency structure)

A **Extensible Graph Representation**

**Overview of Full Log(Graph) Design**

- \( V \) = \{0, 1\} \( k \times \) \( n \) bits
- \( \log \) global scheme
- \( \log \) local scheme
- \( \log \) local scheme
- \( \log \) local scheme

**Near-Optimal Graph Representation**

- Log (edges, weights)
- Log (edges, labels)
- Log (vertex labels)

**High-Performance Graph Representation**

- Log (edges, weights)
- Log (edges, labels)
- Log (vertex labels)

**Condensed Graph Representation**

- Log (adjacency structure)

**Extensible Graph Representation**

- Overview of Full Log(Graph) Design
- \( V \) = \{0, 1\} \( k \times \) \( n \) bits
- \( \log \) global scheme
- \( \log \) local scheme
- \( \log \) local scheme
- \( \log \) local scheme

**Website:**

http://spcl.inf.ethz.ch/Research/Performance/LogGraph

**COMING SOON**
1 \text{Log (Vertex labels)}, \text{Log (Edge weights)}
$\log(\text{Vertex labels}), \log(\text{Edge weights})$
1 \text{ Log ( Vertex labels ), Log ( Edge weights )}
1 Log (Vertex labels), Log (Edge weights)

Symbols
\[ \begin{align*}
\tilde{W} & : \text{max edge weight,} \\
n & : \text{#vertices,} \\
m & : \text{#edges,} \\
d_v & : \text{degree of vertex } v, \\
N_v & : \text{neighbors (adj. array) of vertex } v, \\
\tilde{N}_v & : \text{maximum among } N_v
\end{align*} \]
1 Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

Symbols

\( \bar{W} \): max edge weight,
\( n \): #vertices,
\( m \): #edges,
\( d_v \): degree of vertex \( v \),
\( N_v \): neighbors (adj. array) of vertex \( v \),
\( \bar{N}_v \): maximum among \( N_v \)
1. $\log(n)$ Vertex labels, $\log(m)$ Edge weights

Lower bounds (global) $[\log n]$
1. \( \log(\text{Vertex labels}), \ \log(\text{Edge weights}) \)

Lower bounds (global)

\[ [\log n] [\log \hat{W}] \]

Symbols

- \( \hat{W} \): max edge weight,
- \( n \): #vertices,
- \( m \): #edges,
- \( d_v \): degree of vertex \( v \),
- \( N_v \): neighbors (adj. array) of vertex \( v \),
- \( \hat{N}_v \): maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
$$[\log n] \ [\log \hat{W}]$$

This is it? Not really 😊

Symbols
- $\hat{W}$: max edge weight,
- $n$: #vertices,
- $m$: #edges,
- $d_v$: degree of vertex $v$,
- $N_v$: neighbors (adj. array) of vertex $v$,
- $\hat{N}_v$: maximum among $N_v$
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
$[\log n] \quad [\log \widehat{W}]$

Lower bounds (local)

Symbols
- $\widehat{W}$: max edge weight,
- $n$: #vertices,
- $m$: #edges,
- $d_v$: degree of vertex $v$,
- $N_v$: neighbors (adj. array) of vertex $v$,
- $\widehat{N}_v$: maximum among $N_v$

This is it? Not really 😊
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

[log $n$] [log $\widehat{W}$]

This is it? Not really 😊

Symbols

$\widehat{W}$ : max edge weight,
$n$ : #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:
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

\[ \log n \text{ and } \log \tilde{W} \]

This is it? Not really 😊

Lower bounds (local)

Assume:
- a graph, e.g., \( V = \{1, \ldots, 2^{22}\} \)

Symbols

\( \tilde{W} \) : max edge weight,
\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \tilde{N}_v \) : maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

Lower bounds (local)

Assume:
- a graph, e.g., \( V = \{1, \ldots, 2^{22}\} \)
- A vertex \( v \) with few neighbors: \( d_v \ll n \)

Symbols

- \( \hat{W} \): max edge weight,
- \( n \): #vertices,
- \( m \): #edges,
- \( d_v \): degree of vertex \( v \),
- \( N_v \): neighbors (adj. array) of vertex \( v \),
- \( \hat{N}_v \): maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

\[ \log n \quad \log \hat{W} \]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

Symbols

\( \hat{W} \) : max edge weight,
\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \hat{N}_v \) : maximum among \( N_v \)
Lower bounds (global)

\[ \log n \quad \log \hat{W} \]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

Symbols

\( \hat{W} \) : max edge weight,
\( n \) : #vertices,
\( m \) : #edges,
\( d_v \) : degree of vertex \( v \),
\( N_v \) : neighbors (adj. array) of vertex \( v \),
\( \hat{N}_v \) : maximum among \( N_v \)
1 Log (Vertex labels), Log (Edge weights)

Lower bounds (global)
$[\log n], [\log \tilde{W}]$

This is it? Not really 😊

Lower bounds (local)
Assume:
- a graph, e.g., $V = \{1, \ldots, 2^{2^2}\}$
- A vertex $v$ with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\overline{N}_v \ll n$

Symbols
$\tilde{W}$: max edge weight,
$n$: #vertices,
$m$: #edges,
$d_v$: degree of vertex $v$,
$N_v$: neighbors (adj. array) of vertex $v$,
$\overline{N}_v$: maximum among $N_v$

$[\log 2^{2^2}] = 22$
1. \( \log(\text{Vertex labels}) \), \( \log(\text{Edge weights}) \)

**Lower bounds (global)**

\[ \lceil \log n \rceil \lceil \log \tilde{W} \rceil \]

**Questions:**

This is it? Not really 😊

**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: \( \tilde{N}_v \ll n \)

\[ \lceil \log 2^{22} \rceil = 22 \]

Symbols:
- \( \tilde{W} \): max edge weight,
- \( n \): #vertices,
- \( m \): #edges,
- \( d_v \): degree of vertex \( v \),
- \( N_v \): neighbors (adj. array) of vertex \( v \),
- \( \tilde{N}_v \): maximum among \( N_v \)
1. Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

\[ \log n \quad \log \hat{W} \]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

\[ \log 2^{22} = 22 \]

\( v \rightarrow 2 \quad 3 \quad 4 \quad 5 \)

\( v \rightarrow 0...10 \quad 0...11 \)

\( 0...100 \quad 0...101 \)

19 zeros!

Symbols

- \( \hat{W} \): max edge weight,
- \( n \): #vertices,
- \( m \): #edges,
- \( d_v \): degree of vertex \( v \),
- \( N_v \): neighbors (adj. array) of vertex \( v \),
- \( \hat{N}_v \): maximum among \( N_v \)
1. \( \log(\text{Vertex labels}), \log(\text{Edge weights}) \)

Lower bounds (global)
\[
[\log n] \quad [\log \hat{W}]
\]

This is it? Not really 😊

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: \( \hat{N}_v \ll n \)

\[
\log 2^{22} = 22
\]

Thus, use the local bound \( \left[ \log \hat{N}_v \right] \)
1. Log (Vertex labels), Log (Edge weights)

Symbols:

- $n$: #vertices,
- $m$: #edges,
- $H$: number of compute nodes,
- $H_i$: number of machine elements at level $i$,
- $N$: number of machine levels.
1 \text{Log ( Vertex labels ), Log ( Edge weights )}

This is it? Still not really 😊

Symbols

$n$: #vertices,
$m$: #edges,
$H$: number of compute nodes,
$H_i$: number of machine elements at level $i$,
$N$: number of machine levels
Vertex labels \( \text{Log (        )} \), Edge weights \( \text{Log (        )} \)

Lower bounds (local): distributed memories

Symbols:
- \( n \) : number of vertices,
- \( m \) : number of edges,
- \( H \) : number of compute nodes,
- \( H_i \) : number of machine elements at level \( i \),
- \( N \) : number of machine levels
1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Lower bounds (local):
distributed memories

A Cray XE/XT supercomputer

Symbols

\( n \): #vertices,
\( m \): #edges,
\( H \): number of compute nodes,
\( H_i \): number of machine elements at level \( i \),
\( N \): number of machine levels
Lower bounds (local): distributed memories

1. Log (Vertex labels), Log (Edge weights)

Symbols:

- $n$: number of vertices,
- $m$: number of edges,
- $H$: number of compute nodes,
- $H_i$: number of machine elements at level $i$,
- $N$: number of machine levels

A Cray XE/XT supercomputer: 4 cabinets
1. $\log(n)$, $\log(m)$

This is it? Still not really 😊

Lower bounds (local): distributed memories

A Cray XE/XT supercomputer

4 cabinets:

3 chassis:

Symbols:

$n$: number of vertices,
$m$: number of edges,
$H$: number of compute nodes,
$H_i$: number of machine elements at level $i$,
$N$: number of machine levels
1. Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Lower bounds (local): distributed memories

Symbols:
- $n$: #vertices,
- $m$: #edges,
- $H$: number of compute nodes,
- $H_i$: number of machine elements at level $i$,
- $N$: number of machine levels

A Cray XE/XT supercomputer

- 4 cabinets:
- 3 chassis:
- 8 blades:
Log (Vertex labels), Log (Edge weights)

Symbols
\begin{align*}
    n &= \text{#vertices,} \\
    m &= \text{#edges,} \\
    H &= \text{number of compute nodes,} \\
    H_i &= \text{number of machine elements at level } i, \\
    N &= \text{number of machine levels}
\end{align*}

Lower bounds (local): distributed memories

A Cray XE/XT supercomputer

- 4 cabinets:
- 3 chassis:
- 8 blades:
- 4 nodes:
1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Lower bounds (local): distributed memories

A Cray XE/XT supercomputer

- 4 cabinets:
- 3 chassis:
- 8 blades:
- 4 nodes:
- 32 cores:

Symbols:

\[ n : \text{#vertices}, \]
\[ m : \text{#edges}, \]
\[ H : \text{number of compute nodes}, \]
\[ H_i : \text{number of machine elements at level } i, \]
\[ N : \text{number of machine levels} \]
1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Lower bounds (local):
distributed memories

- 4 cabinets:
- 3 chassis:
- 8 blades:
- 4 nodes:
- 32 cores:

A Cray XE/XT supercomputer

Symbols:

\( n \) : #vertices,
\( m \) : #edges,
\( H \) : number of compute nodes,
\( H_i \) : number of machine elements at level \( i \),
\( N \) : number of machine levels
Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Lower bounds (local): distributed memories

A Cray XE/XT supercomputer

4 cabinets: ...
3 chassis: ...
8 blades: ...
4 nodes: ...

4 cabinets: ...
3 chassis: ...
8 blades: ...
4 nodes: ...

A Cray XE/XT supercomputer

Symbols

\( n \) : #vertices,
\( m \) : #edges,
\( H \) : number of compute nodes,
\( H_i \) : number of machine elements at level \( i \),
\( N \) : number of machine levels
1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😊

Symbols

\( n \) : #vertices,
\( m \) : #edges,
\( H \) : number of compute nodes,
\( H_i \) : number of machine elements at level \( i \),
\( N \) : number of machine levels

Lower bounds (local): distributed memories

The number of vertices that can be stored in the memory of one node:

- 4 cabinets:
- 3 chassis:
- 8 blades:
- 4 nodes:
  \( H = 4 \)
- 32 cores:

A Cray XE/XT supercomputer
1. Log (Vertex labels), Log (Edge weights)

This is it? Still not really 😃

**Lower bounds (local): distributed memories**

The number of vertices that can be stored in the memory of one node: \( \frac{n}{H} \)

- 4 cabinets: 4 nodes: \( H = 4 \)
- 3 chassis: 8 blades: 32 cores:

**Symbols**

- \( n \) : number of vertices
- \( m \) : number of edges
- \( H \) : number of compute nodes
- \( H_i \) : number of machine elements at level \( i \)
- \( N \) : number of machine levels

A Cray XE/XT supercomputer
Lower bounds (local):

distributed memories

The number of vertices that can be stored in the memory of one node:
\[ \frac{n}{H} \]

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

Symbols:
- \( n \): #vertices,
- \( m \): #edges,
- \( H \): number of compute nodes,
- \( H_i \): number of machine elements at level \( i \),
- \( N \): number of machine levels
**Vertices**

- **Log (Vertex labels), Log (Edge weights)**

**Symbols**

- $n$: number of vertices,
- $m$: number of edges,
- $H$: number of compute nodes,
- $H_i$: number of machine elements at level $i$,
- $N$: number of machine levels

**Lower bounds (local): distributed memories**

The number of vertices that can be stored in the memory of one node: $\frac{n}{H}$

The "**intra-node**" vertex label thus takes [bits]: $\lceil \log \frac{n}{H} \rceil$

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

**A Cray XE/XT supercomputer**

- 4 cabinets:
- 3 chassis:
- 8 blades:
- 4 nodes: $H = 4$
- 32 cores:
Lower bounds (local): distributed memories

The number of vertices that can be stored in the memory of one node: \( n \frac{n}{H} \)

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

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

Symbols:

- \( n \): #vertices,
- \( m \): #edges,
- \( H \): number of compute nodes,
- \( H_i \): number of machine elements at level \( i \),
- \( N \): number of machine levels

The total size of the adjacency arrays is thus [bits]:

\[ n \left\lceil \log \frac{n}{H} \right\rceil + H \lceil \log H \rceil \]
1. \( \log(n) \), \( \log(m) \)

This is it? Still not really 😁

**Lower bounds (local): distributed memories**

The number of vertices that can be stored in the memory of one node:

\[
\frac{n}{H}
\]

The "intra-node" vertex label thus takes [bits]:

\[
\log \left( \frac{n}{H} \right)
\]

The "inter-node" vertex label is unique for a whole node and it takes [bits]:

\[
\log H
\]

A Cray XE/XT supercomputer

The total size of the adjacency arrays is thus [bits]:

\[
n \left\lceil \log \frac{n}{H} \right\rceil + H \lfloor \log H \rfloor
\]

We also generalize this to arbitrarily many levels (details in the paper ☺️) and derive the total size:

**Symbols**

- \( n \) : #vertices,
- \( m \) : #edges,
- \( H \) : number of compute nodes,
- \( H_i \) : number of machine elements at level \( i \),
- \( N \) : number of machine levels
1 Log (Vertex labels), Log (Edge weights)

Lower bounds (local): distributed memories

The number of vertices that can be stored in the memory of one node: \( n \frac{H}{H} \)

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

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

A Cray XE/XT supercomputer

The total size of the adjacency arrays is thus [bits]:
\[
n \left\lceil \log \frac{n}{H} \right\rceil + H \left\lceil \log H \right\rceil
\]

We also generalize this to arbitrarily many levels (details in the paper 😊) and derive the total size:
\[
n \left\lceil \log \frac{n}{H} \right\rceil + \sum_{j=2}^{N-1} H_j \left\lceil \log H_j \right\rceil
\]
Log (Vertex labels), Log (Edge weights)

Formal analyses: more (check the paper 😊)
1. **Log (Vertex labels), Log (Edge weights)**

Formal analyses: more (check the paper 😊)

\[ |\mathcal{A}| = \sum_{v \in V} \left( d_v \left( \log \hat{N}_v \right) + \left\lfloor \log \log \hat{N}_v \right\rfloor \right) \]

\[ |\mathcal{A}| = n \left( \log \frac{n}{\mathcal{H}} \right) + \mathcal{H} \left( \log \mathcal{H} \right) \]

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

\[ E[|\mathcal{O}|] = n \left( \log \left( 2pn^2 \right) \right) = n \left( \log 2p + 2 \log n \right) \]

\[ \forall v, u \in V \ (u \in \mathcal{N}_v) \Rightarrow \left[ \mathcal{N}(u) \leq \hat{N}_v \right] \]

\[ |\mathcal{A}| = \sum_{v \in V} \left( d_v \left( \log \hat{N}_v \right) + \left\lfloor \log \log \hat{N}_v \right\rfloor \right) \]

\[ |\mathcal{A}| = 2m \left( \left\lfloor \log n \right\rfloor + \left\lfloor \log \mathcal{W} \right\rfloor \right) \]

\[ |\mathcal{A}| = \sum_{v \in V} \left( d_v \left( \left\lfloor \log \hat{N}_v \right\rfloor + \left\lfloor \log \mathcal{W} \right\rfloor \right) + \left\lfloor \log \log \hat{N}_v \right\rfloor + \left\lfloor \log \log \mathcal{W} \right\rfloor \right) \]

\[ E[|\mathcal{A}|] = \left( \left\lfloor \log n \right\rfloor + \left\lfloor \log \mathcal{W} \right\rfloor \right) pn^2 \]
Vertex labels

\[ E[|\mathcal{O}|] = n \left[ \log \left( 2pn^2 \right) \right] = n \left[ \log 2p + 2 \log n \right] \]

\[ \forall v,u \in V \left( u \in N_v \right) \Rightarrow \left[ N(u) \leq \hat{N}_v \right] \]

\[ |\mathcal{A}| = 2m \left[ \left\lfloor \log n \right\rfloor + \left\lfloor \log \hat{W} \right\rfloor \right] \]

\[ E[|\mathcal{A}|] = \left( \left\lfloor \log n \right\rfloor + \left\lfloor \log \hat{W} \right\rfloor \right)pn^2 \]
2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors
2 Log ( Offset structure )

...Encode the resulting bit vectors as succinct bit vectors

Formal analyses
### Formal analyses

<table>
<thead>
<tr>
<th>Offset structure</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>$O \left( \frac{W_m}{B} \right)$</td>
<td>$\frac{2W_m}{B}$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Interleaved [44]</td>
<td>bvIL</td>
<td>$O \left( \frac{W_m}{B} + \frac{W_m}{L} \right)$</td>
<td>$2Wm \left( \frac{1}{B} + \frac{64}{L} \right)$</td>
<td>$O \left( \log \frac{W_m}{B} \right)$</td>
</tr>
<tr>
<td>Entropy based [31, 78]</td>
<td>bvEN</td>
<td>$O \left( \frac{W_m}{B} \log \frac{W_m}{B} \right)$</td>
<td>$\approx \log \left( \frac{2W_m}{B} \right)$</td>
<td>$O \left( \log \frac{W_m}{B} \right)$</td>
</tr>
<tr>
<td>Sparse [76]</td>
<td>bvSD</td>
<td>$O \left( n + n \log \frac{W_m}{Bn} \right)$</td>
<td>$\approx n \left( 2 + \log \frac{2W_m}{Bn} \right)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>B-tree based [1]</td>
<td>bvBT</td>
<td>$O \left( \frac{W_m}{B} \right)$</td>
<td>$\approx 1.1 \cdot \frac{2W_m}{B}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Gap-compressed [1]</td>
<td>bvGC</td>
<td>$O \left( \frac{W_m}{B} \log \frac{W_m}{Bn} \right)$</td>
<td>$\approx 1.3 \cdot \frac{2W_m}{B} \log \frac{2W_m}{Bn}$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
### Formal analyses

Check the paper for details 😊

<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>$O\left( \frac{Wm}{B} \log \frac{Wm}{Bn} \right)$</td>
<td>$\approx \log \left( \frac{2Wm}{n} \right)$</td>
<td>$O \left( \log \frac{Wm}{B} \right)$</td>
</tr>
<tr>
<td>Sparse [76]</td>
<td>bvSD</td>
<td>$O\left( n + n \log \frac{Wm}{Bn} \right)$</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>
## Formal analyses

<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>$O\left(n + n \log \frac{Wm}{Bn}\right)$</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>

Check the paper for details 😊

---

### Key methods

Use the sdsl-lite sequential library of succinct bit vectors [1] and investigate if it fares well when being accessed by multiple threads.

1 Log (Vertex labels), Log (Edge weights) Storage

**Power-law model**

- Design: $32 + 8 \log n + \log W$
- $n = 4.29B$

**Random-uniform model**

- Design: $32 + 8 \log n + \log W$
- $\alpha = 1$
1 Log (Vertex labels), Log (Edge weights) Storage

Power-law model

Random-uniform model

#bits per vertex label

Design: $32+8 \log n + \log W$

Design: $32+8 \log n + \log W$

Size [TiB]

Size [TiB]
1. Log (Vertex labels), Log (Edge weights)  

Storage

Power-law model

Random-uniform model

#bits per edge weight

#bits per vertex label

Design:

32+8 logn+logW

Size [TiB]

n = 4.29B

p (edge probability)

α = 1

β

Log ( ), Log ( )
Log(Graph) consistently reduces storage overhead (by 20-35%).

- **Vertex labels**
  - Power-law model: \( n = 4.29B \)
  - Random-uniform model: \( \alpha = 1 \)

- **Edge weights**
  - Both models use \( \log n + \log W \) for storage.
1. Log (Vertex labels), Log (Edge weights) Performance

The diagram shows the performance of the SSSP algorithm for different numbers of edges per vertex, comparing GapBS and Log(Graph) schemes. The x-axis represents the number of edges per vertex, ranging from 64.0 to 1024.0, while the y-axis indicates the time in seconds. The graph illustrates that the performance improves as the number of edges increases for both schemes.
1. \[ \log(\text{Vertex labels}), \log(\text{Edge weights}) \] Performance

Log(Graph) accelerates GAPBS

<table>
<thead>
<tr>
<th>Number of edges per vertex</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>64.0</td>
<td>1.0</td>
</tr>
<tr>
<td>128.0</td>
<td>1.5</td>
</tr>
<tr>
<td>256.0</td>
<td>2.0</td>
</tr>
<tr>
<td>512.0</td>
<td>2.3</td>
</tr>
<tr>
<td>1024.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Kronecker graphs
Number of vertices: 4M

SSSP
1. Log (Vertex labels), Log (Edge weights) → Performance

Log(Graph) accelerates GAPBS

Both storage and performance are improved simultaneously

Kronecker graphs
Number of vertices: 4M
1. Log (Vertex labels), Log (Edge weights)  Performance

Betweenness Centrality

“LG”: Log(Graph)
Trad: Traditional (non compressed, GAPBS)
“g”: global scheme
“l”: local scheme
“gap”: additional gap encoding

Kronecker graphs
Number of vertices: 4M
Betweenness Centrality

“LG”: Log(Graph)
Trad: Traditional (non compressed, GAPBS)
“g”: global scheme
“l”: local scheme
“gap”: additional gap encoding

Sparse graphs

Dense graphs

Kronecker graphs

Number of vertices: 4M
1. **Log (Vertex labels), Log (Edge weights)**

**Performance**

**Betweenness Centrality**

“LG”: Log(Graph)

Trad: Traditional (non compressed, GAPBS)

“g”: global scheme

“l”: local scheme

“gap”: additional gap encoding

---

**Kronecker graphs**

Number of vertices: 4M

Number of edges per vertex:

- 1
- 2
- 4
- 8
- 16
- 32

Number of edges per vertex:

- 64
- 128
- 256
- 512
- 1024

Log(Graph) incurs negligible overheads
1. Log (Vertex labels), Log (Edge weights) Performance

BFS

“LG”: Log(Graph)
Trad: Traditional (non compressed, GAPBS)
“g”: global scheme
“l”: local scheme
“gap”: additional gap encoding

Kronecker graphs
Number of vertices: 4M
Sparse graphs

<table>
<thead>
<tr>
<th>Scheme:</th>
<th>LG-(g)</th>
<th>LG-(g)-gap</th>
<th>LG-(l)</th>
<th>LG-(l)-gap</th>
<th>Trad</th>
</tr>
</thead>
</table>

Dense graphs

<table>
<thead>
<tr>
<th>Scheme:</th>
<th>LG-(g)</th>
<th>LG-(g)-gap</th>
<th>LG-(l)</th>
<th>LG-(l)-gap</th>
<th>Trad</th>
</tr>
</thead>
</table>

Performance

- "LG": Log(Graph)
- Trad: Traditional (non compressed, GAPBS)
- "g": global scheme
- "l": local scheme
- "gap": additional gap encoding

BFS

Kronecker graphs

Number of vertices: 4M

Number of edges per vertex:

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256
- 512
- 1024
1. \( \log(\text{vertex labels}), \log(\text{edge weights}) \) Performance

Sparse graphs

<table>
<thead>
<tr>
<th>Scheme:</th>
<th>Log (g), Log (g-gap), Log (f), Log (l-gap), Trad</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Number of edges per vertex:</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [s]</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
<td>0.05</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Dense graphs

<table>
<thead>
<tr>
<th>Scheme:</th>
<th>Log (g), Log (g-gap), Log (f), Log (l-gap), Trad</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Number of edges per vertex:</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [s]</td>
<td>10</td>
<td>5</td>
<td>2.5</td>
<td>1.25</td>
<td>0.625</td>
</tr>
</tbody>
</table>

Both storage and performance are improved simultaneously.
Sparse graphs

Log(Graph), Log(Edge weights)

Performance

Log(Graph) accelerates GAPBS

Dense graphs

Both storage and performance are improved simultaneously

“LG”: Log(Graph)
“g”: global scheme
“l”: local scheme

Trad: Traditional (non compressed, GAPBS)

Number of vertices: 4M

Kronecker graphs

Number of edges per vertex:

1, 2, 4, 8, 16, 32

Number of edges per vertex:

64, 128, 256, 512, 1024
1. Log (Vertex labels), Log (Edge weights) Communicated data

Various real-world and synthetic graphs

Communicated data [MB]

1.0e+05
1.0e+03
1.0e+01

Scheme:
- Log(Graph)
- Trad

1024 compute nodes
The amount of communicated data is consistently reduced by ~37%.
Log (Adjacency structure) Storage
3 Log ( Adjacency structure ) Storage

- **Trad**: Traditional adjacency array
- **DMd / DMf**: Degree Minimizing (without / with gap encoding)
- **WG**: WebGraph compression
- **BRB, RB**: Schemes targeting certain specific classes of graphs
Log (Adjacency structure) Storage

**Trad**: Traditional adjacency array
**DMd / DMf**: Degree Minimizing (without / with gap encoding)
**WG**: WebGraph compression
**BRB, RB**: Schemes targeting certain specific classes of graphs
Lots of data 😊

Conclusions:

3 Log (Adjacency structure) Storage

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs
Lots of data 😊

Conclusions: **WebGraph best for web graphs 😊**

**Storage**

- **Log (Adjacency structure)**

- **Trad**: Traditional adjacency array
- **DMd / DMf**: Degree Minimizing (without / with gap encoding)
- **WG**: WebGraph compression
- **BRB, RB**: Schemes targeting certain specific classes of graphs

![Graph](image.png)

- Various real-world graphs
- Relative size

---

[Graph data and schemes explained]
Lots of data 😊

Conclusions:

WebGraph best for web graphs 😊

BRB, RB: various tradeoffs but very expensive preprocessing (details in the paper)

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs
Lots of data 😊

Conclusions:

- WebGraph best for web graphs 😊
- DMd: much better than DMf, often comparable to others
- BRB, RB: various tradeoffs but very expensive preprocessing (details in the paper)

**Storage**

**Adjacency structure**

**Trade-offs**

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs
Log (Adjacency structure) Performance
Performance

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
RB: Scheme targeting certain specific classes of graphs
WebGraph is the slowest

**Performance**

- **Trad**: Traditional adjacency array
- **DMd / DMf**: Degree Minimizing (without / with gap encoding)
- **WG**: WebGraph compression
- **RB**: Scheme targeting certain specific classes of graphs

### Graphs

**BFS**

- gho, orm, tw, usrn

**PageRank**

- gho, orm, tw, usrn

**Triangle Counting**

- ask, rca, lj2, pok, pat, ber

<table>
<thead>
<tr>
<th>Scheme:</th>
<th>Trad</th>
<th>RB</th>
<th>DMd</th>
<th>DMf</th>
<th>WG</th>
</tr>
</thead>
</table>

WebGraph is the slowest
WebGraph is the slowest
DM, RB: comparable

Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
RB: Scheme targeting certain specific classes of graphs
Log(Graph) full design...
Log(Graph) full design...
Log(Graph) full design...

1. Input structures
   - Graph $G$ ($§2$)

2. Logarithmize fine elements ($§3$)
   - Logarithmize vertex IDs... ($§3.2$)
     Example ID
     Remove leading bits (simple bit packing)
     \[
     \log(2) = \log(0...010_2) = 010_2
     \]
   - ...globally ($§3.2.1$)
   - ...locally ($§3.2.2$)
   - ...on DM ($§3.2.3$) systems

3. Logarithmize other elements ($§3.3 - §3.4$)

4. (§3.5) Analyze
   - (§3.6) Use difference encoding ($§3.7$)
   - (§3.8) Ensure
   - (§3.9) High
Log(Graph) full design...
Log(Graph) full design...
Log(Graph) full design...

1. Input structures
   - Graph $G$ ($\S2$)

2. Logarithmize fine elements ($\S3$)
   - Logarithmize vertex IDs... ($\S3.2$)
     - Example ID
       - $\text{Log}(2) = \text{Log}(0..010_2) = 010_2$

3. Logarithmize offset structure ($\S4$)
   - ($\S4.2$) Understand storage lower bounds

4. Logarithmize adjacency structure ($\S5$)
   - ($\S5.1$) Unify with $P+T$ ($\S5.2$)

5. High-performance extensible library ($\S6$)

6. Use ILP
   - This part is covered in the extended technical report version of the paper
Log(Graph) full design...

1. Input structures
   1.1 Graph $G$ (§2)
   1.2 Adjacency Array (§2)

2. Logarithmize fine elements (§3)
   2.2 Logarithmize vertex IDs... (§3.2)
      Example ID
      $\log(2) = \log(0...010_2) = 010_2$
   2.3...globally (§3.2.1)
   2.4...locally (§3.2.2)
   2.5...on DM (§3.2.3)
   2.6 Logarithmize other elements
      (§3.3 - §3.4)

3. Logarithmize offset structure (§4)
   3.1 Understand storage lower bounds
   3.2 Incorporate (§4.3)
      succintness
      $\log(OPT + o(OPT))$
   3.3 Theory (§4.4)
   3.4 Performance implementation
      (§4.5)

4. Logarithmize adjacency structure (§5)
   4.1 Unify bounds
   4.2 With $P + T$ (§5.2)
   4.3 Incorporate recursive bisectioning
   4.4...use RB
   4.5...use BRB

5. High-performance extensible library (§6)
   Use ILP
   This part is covered in the extended technical report version of the paper

---

Use Integer Linear Programming (ILP)
Log(Graph) full design...

Understand storage lower bounds and the theory
Log(Graph) full design...

Understand storage lower bounds and the theory

Ensure high-performance implementation
Log(Graph) full design...

Understand storage lower bounds and the theory

Ensure high-performance implementation

Use Integer Linear Programming (ILP) for more storage reductions
Key method (vertex labels)
Key method (vertex labels)

**Bit packing:** use $\lceil \log n \rceil$ bits for one vertex label
Key method (vertex labels)

Bit packing: use \([\log n]\) bits for one vertex label

Modern bitwise operations
Key method (vertex labels)

Bit packing: use $\lceil \log n \rceil$ bits for one vertex label

Modern bitwise operations

Key method (offsets)
Key method (vertex labels)

**Bit packing:** use $[\log n]$ bits for one vertex label

Modern bitwise operations

Key method (offsets)

**Succinct bit vectors:** understand state-of-the-art designs and use the best ones in a given context
Key method (vertex labels)

**Bit packing:** use $\lceil \log n \rceil$ bits for one vertex label

Modern bitwise operations

Key method (offsets)

**Succinct bit vectors:**
understand state-of-the-art designs and use the best ones in a given context

<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>$2\frac{Wm}{B}$</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>$2\frac{Wm}{B} \left(\frac{1}{B} + \frac{64}{L}\right)$</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>$\approx \log \left(\frac{2\frac{Wm}{B}}{n}\right)$</td>
<td>$O\left(\log \frac{Wm}{B}\right)$</td>
</tr>
<tr>
<td>Sparse [76]</td>
<td>bvSD</td>
<td>$O\left(n + n \log \frac{Wm}{Bn}\right)$</td>
<td>$\approx n \left(2 + \log \frac{2\frac{Wm}{Bn}}{Bn}\right)$</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>$\approx 1.1 \cdot \frac{2\frac{Wm}{Bn}}{Bn}$</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>$\approx 1.3 \cdot \frac{2\frac{Wm}{Bn}}{Bn} \log \frac{2\frac{Wm}{Bn}}{Bn}$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Key method (vertex labels)

**Bit packing**: use $\lceil \log n \rceil$ bits for one vertex label

Modern bitwise operations

Key method (neighborhoods)

**Recursive partitioning**: use representations that assume more about graph structure to enable better bounds

Key method (offsets)

**Succinct bit vectors**: understand state-of-the-art designs and use the best ones in a given context
Key method (vertex labels)

**Bit packing:** use \( \lceil \log n \rceil \) bits for one vertex label

Modern bitwise operations

---

Key method (neighbors)

**Recursive partitioning:** use representations that assume more about graph structure to enable better bounds

---

Key method (offsets)

**Succinct bit vectors:**
understand state-of-the-art designs and use
the best ones in a given context

---

<table>
<thead>
<tr>
<th>( O )</th>
<th>ID</th>
<th>Asymptotic size [bits]</th>
<th>Exact size [bits]</th>
<th>select or ( O[\text{tr}] )</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>( 2\frac{Wm}{B} + 64 )</td>
<td>( O \left( \log \frac{Wm}{B} \right) )</td>
</tr>
<tr>
<td>Interleaved [44]</td>
<td>bvIL</td>
<td>( \frac{Wn}{B} + \frac{Wm}{B} )</td>
<td>( 2\frac{Wm}{B} \left( \frac{1}{B} + \frac{64}{n} \right) )</td>
<td>( O \left( \log \frac{Wm}{B} \right) )</td>
</tr>
<tr>
<td>Entropy based [31, 78]</td>
<td>bvEN</td>
<td>( \frac{Wm}{B} \log \frac{Wm}{B} )</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>( \frac{Wm}{B} \log \frac{Wm}{Bn} )</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>( \frac{Wm}{B} \log \frac{Wm}{Bn} )</td>
<td>( \approx 1.1 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn} )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Gap-compressed [1]</td>
<td>bvGC</td>
<td>( \frac{Wm}{B} \log \frac{Wm}{Bn} )</td>
<td>( \approx 1.3 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn} )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>