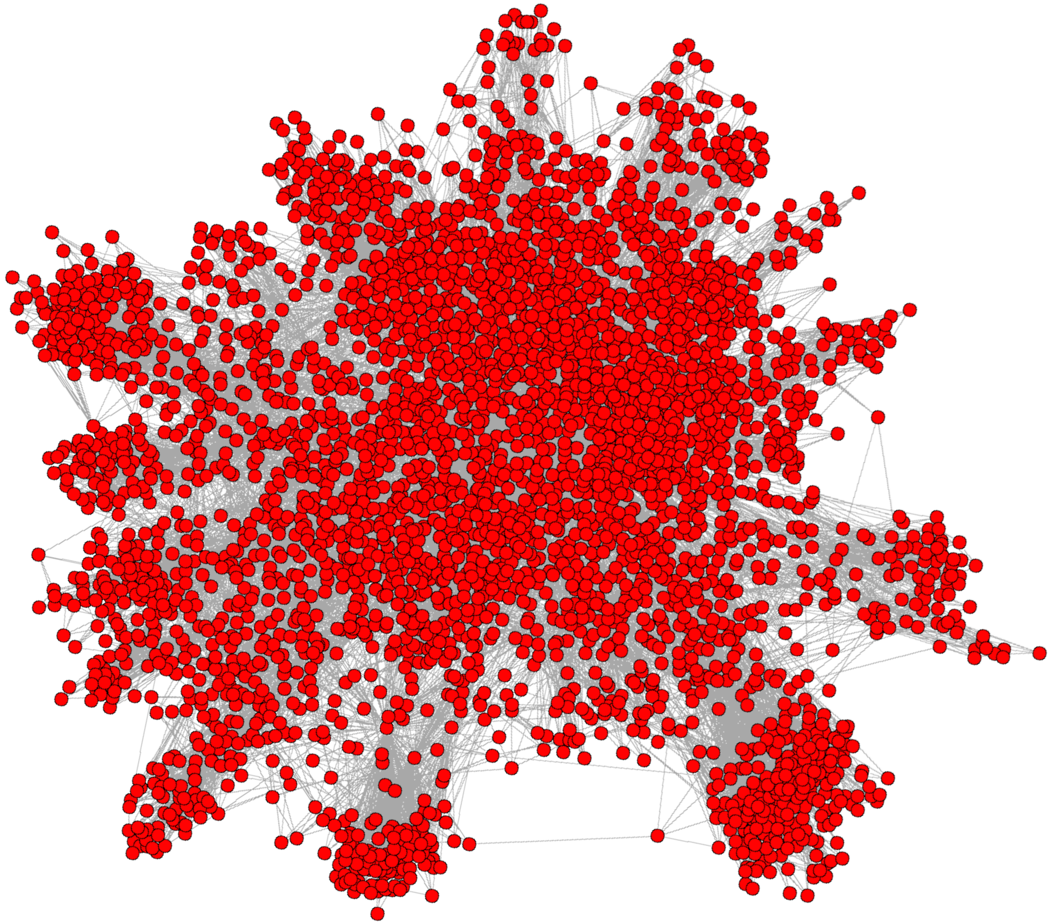
MACIEJ BESTA, DIMITRI STANOJEVIC, TIJANA ZIVIC, JAGPREET SINGH, MAURICE HOEROLD, TORSTEN HOEFLER

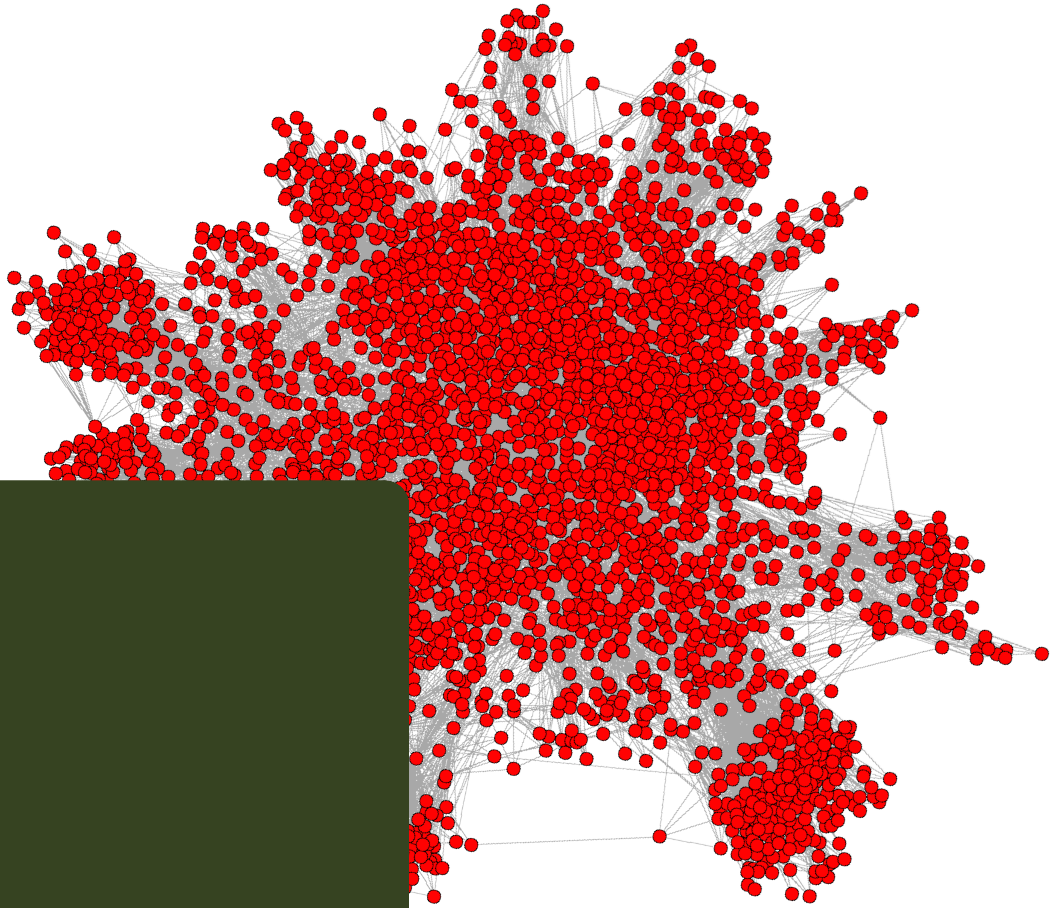
Log(Graph): A Near-Optimal High-Performance Graph Representation

Large graphs…

Large graphs…

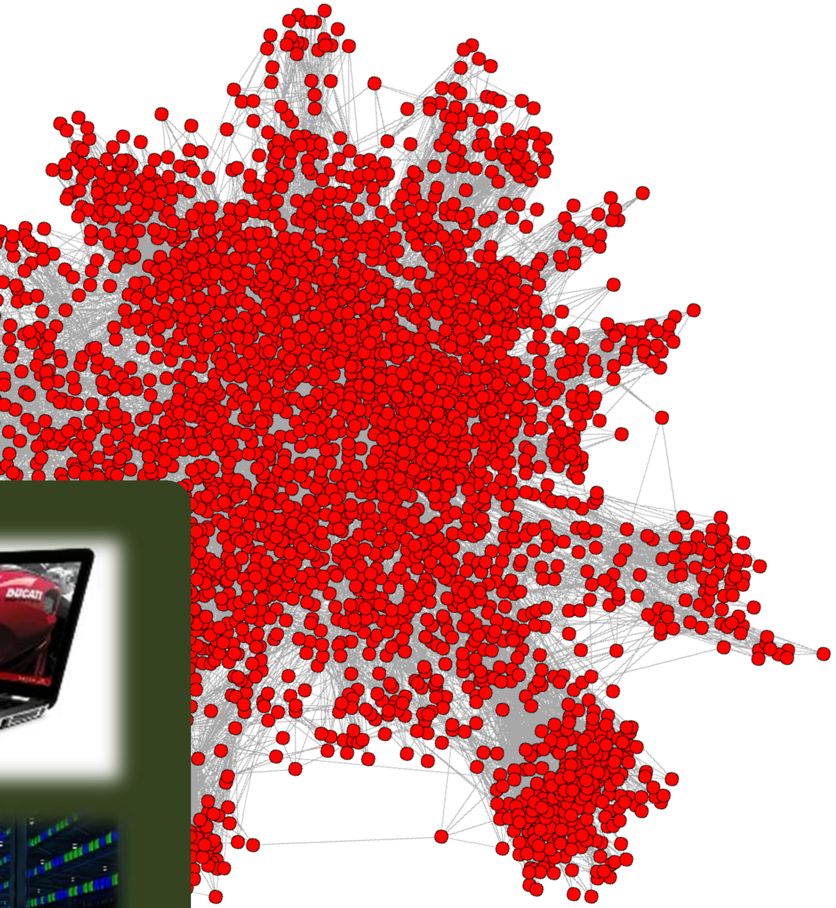


Large graphs…



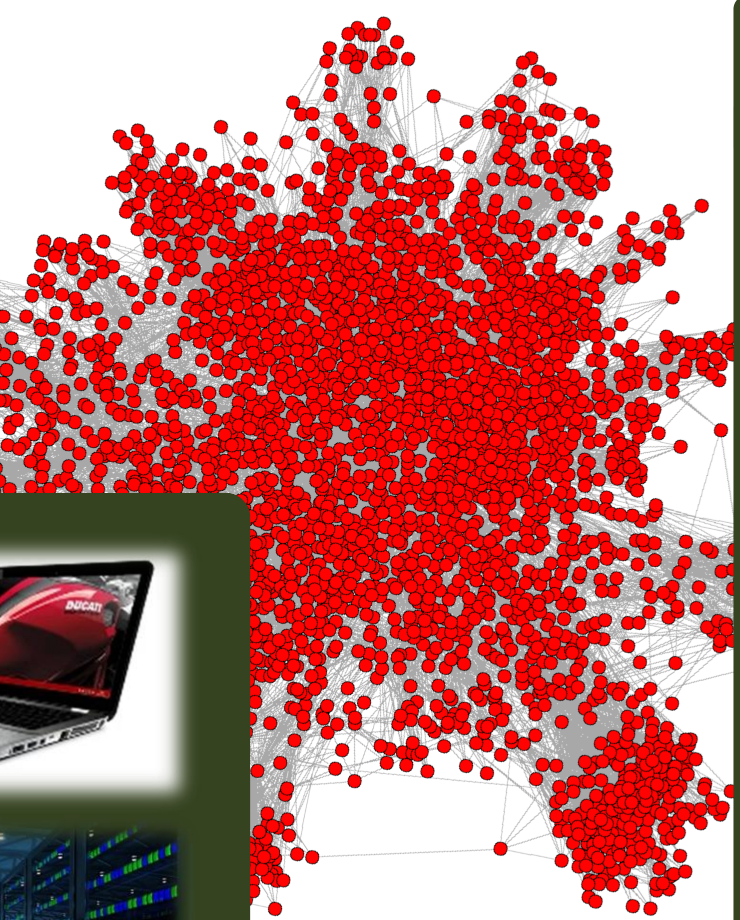
Running on…

Large graphs…



Running on…

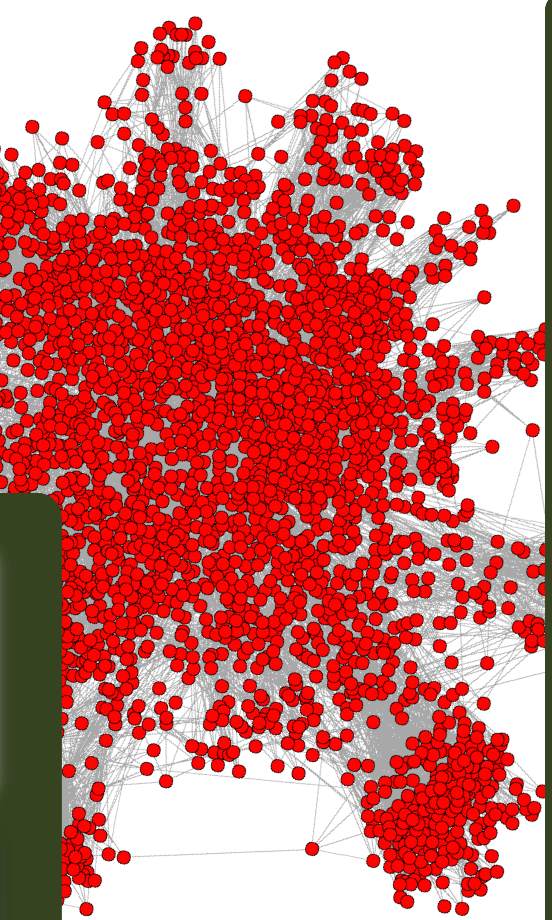
Large graphs…



Running on…

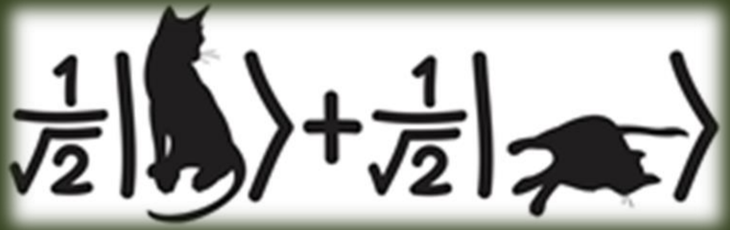
Used in…

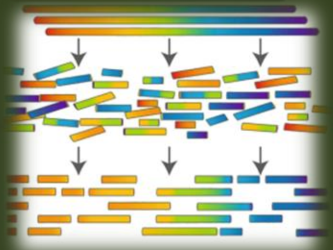
Large graphs…



Running on…

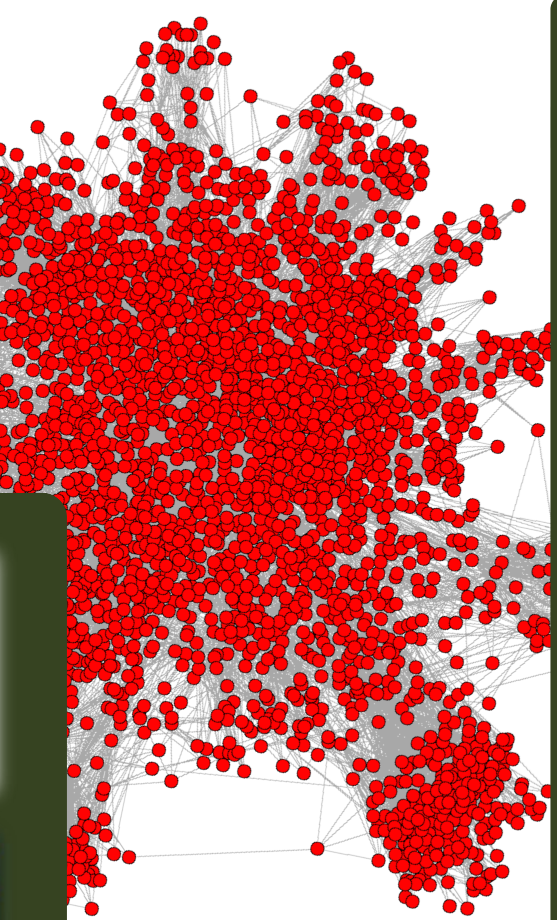
Used in…





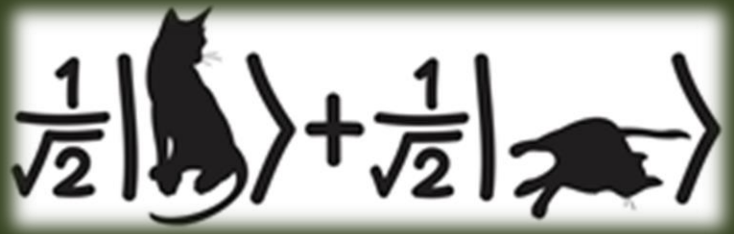
Large graphs…

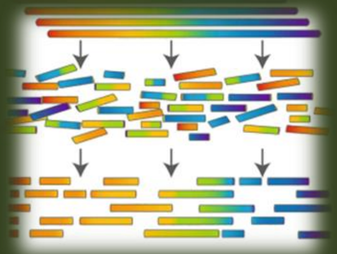
Huge



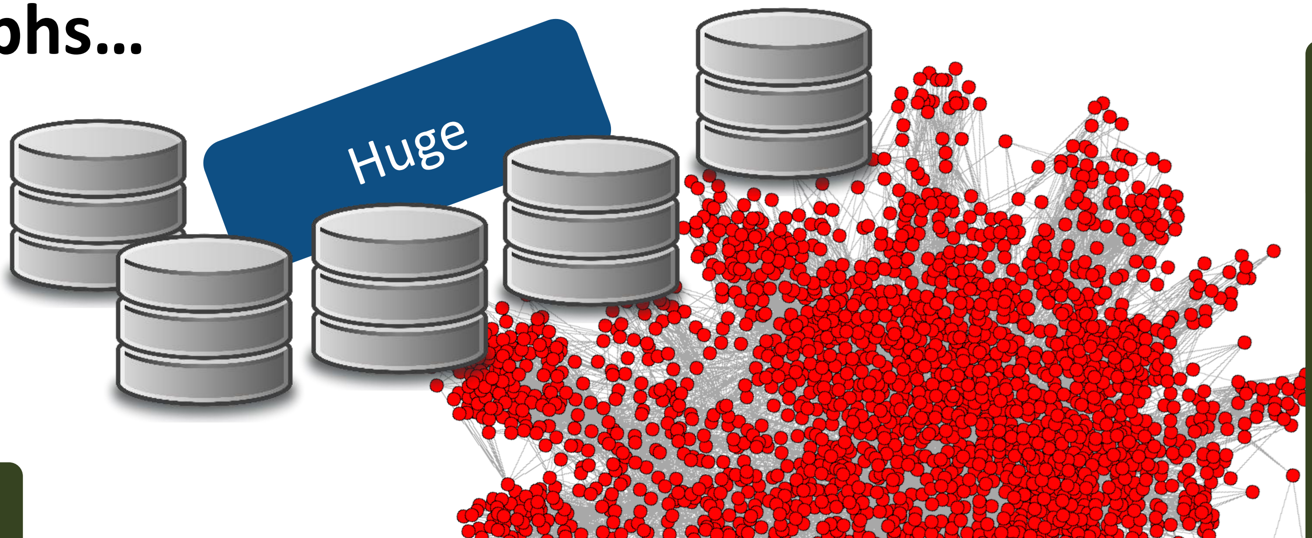
Running on…

Used in…



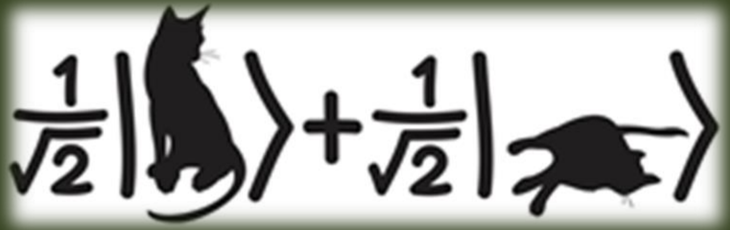


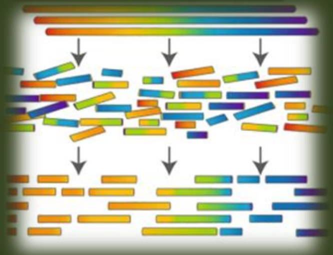
Large graphs...



Running on...

Used in...





Large graphs…

en	Wikipedia edits (en)	B	50,757,442	572,591,272
TW	Twitter (WWW)	D	41,652,230	1,468,365,182
TF	Twitter (MPI)	D	52,579,682	1,963,263,821
FR	Friendster	D	68,349,466	2,586,147,869
UL	UK domain (2007)	U	105,153,952	3,301,876,564

KONECT graph datasets

en	Wikipedia edits (en)	B	50,757,442	572,591,272
TW	Twitter (WWW)	D	41,652,230	1,468,365,182
TF	Twitter (MPI)	D	52,579,682	1,963,263,821
FR	Friendster	D	68,349,466	2,586,147,869
UL	UK domain (2007)	U	105,153,952	3,301,876,564

KONECT graph datasets

Graph500 Benchmark

Top Ten from June 2018 BFS

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

en	Wikipedia edits (en)	50,757,442	572,591,272
TW	Twitter (WWW)	41,652,230	1,468,365,182
TF	Twitter (MPI)	52,579,682	1,963,263,821
FR	Friendster	68,349,466	2,586,147,869
UL	UK domain (2007)	105,153,952	3,301,876,564

KONECT graph datasets

Graph500 Benchmark

...ON	COUNTRY	YEAR	NUMBER OF NODES	NUMBER OF CORES	SCALE	GTEPS
...e Hyogo	Japan	2011	82944	663552	40	38621.4
...xi	China	2015	40768	10599680	40	23755.7
...rmore CA	USA	2012	98304	1572864	41	23751
...ago IL	USA	2012	49152	786432	40	14982

Webgraph datasets

Graph	Crawl date	Nodes	Arcs
uk-2014	2014	787 801 471	47 614 527 250
eu-2015	2015	1 070 557 254	91 792 261 600
gsh-2015	2015	988 490 691	33 877 399 152
uk-2014-host	2014	4 769 354	50 829 923
eu-2015-host	2015	11 264 052	386 915 963
gsh-2015-host	2015	68 660 142	1 802 747 600
uk-2014-tpd	2014	1 766 010	18 244 650
eu-2015-tpd	2015	6 650 532	170 145 510
gsh-2015-tpd	2015	30 809 122	602 119 716
clueweb12	2012	978 408 098	42 574 107 469
uk-2002	2002	18 520 486	298 113 762

KONECT graph datasets

en	Wikipedia edits (en)	50,757,442	572,591,272
TW	Twitter (WWW)	41,652,230	1,468,365,182
TF	Twitter (MPI)	52,579,682	1,963,263,821
FR	Friendster	68,349,466	2,586,147,869
UL	UK domain (2007)	105,153,952	3,301,876,564

Web data commons datasets

Granularity	#Nodes	#Arcs
Page	3,563 million	128,736 million
Host	101 million	2,043 million
Pay-Level-Domain	43 million	623 million

Graph500 Benchmark

...ON	COUNTRY	YEAR	NUMBER OF NODES	NUMBER OF CORES	SCALE	GTEPS
...e Hyogo	Japan	2011	82944	663552	40	38621.4
...i	China	2015	40768	10599680	40	23755.7
...rmore CA	USA	2012	98304	1572864	41	23751
...ago IL	USA	2012	49152	786432	40	14982

Webgraph datasets

Graph	Crawl date	Nodes	Arcs
uk-2014	2014	787 801 471	47 614 527 250
eu-2015	2015	1 070 557 254	91 792 261 600
gsh-2015	2015	988 490 691	33 877 399 152
uk-2014-host	2014	4 769 354	50 829 923
eu-2015-host	2015	11 264 052	386 915 963
gsh-2015-host	2015	68 660 142	1 802 747 600
uk-2014-tpd	2014	1 766 010	18 244 650
eu-2015-tpd	2015	6 650 532	170 145 510
gsh-2015-tpd	2015	30 809 122	602 119 716
clueweb12	2012	978 408 098	42 574 107 469
uk-2002	2002	18 520 486	298 113 762

en	Wikipedia edits (en)	50,757,442	572,591,272
TW	Twitter (WWW)	41,652,230	1,468,365,182
TF	Twitter (MPI)	52,579,682	1,963,263,821
FR	Friendster	68,349,466	2,586,147,869
UL	UK domain (2007)	105,153,952	3,301,876,564

KONECT graph datasets

Web data commons datasets

Granularity	#Nodes	#Arcs
Page	3,563 million	128,736 million
Host	101 million	2,043 million
Pay-Level-Domain	43 million	623 million

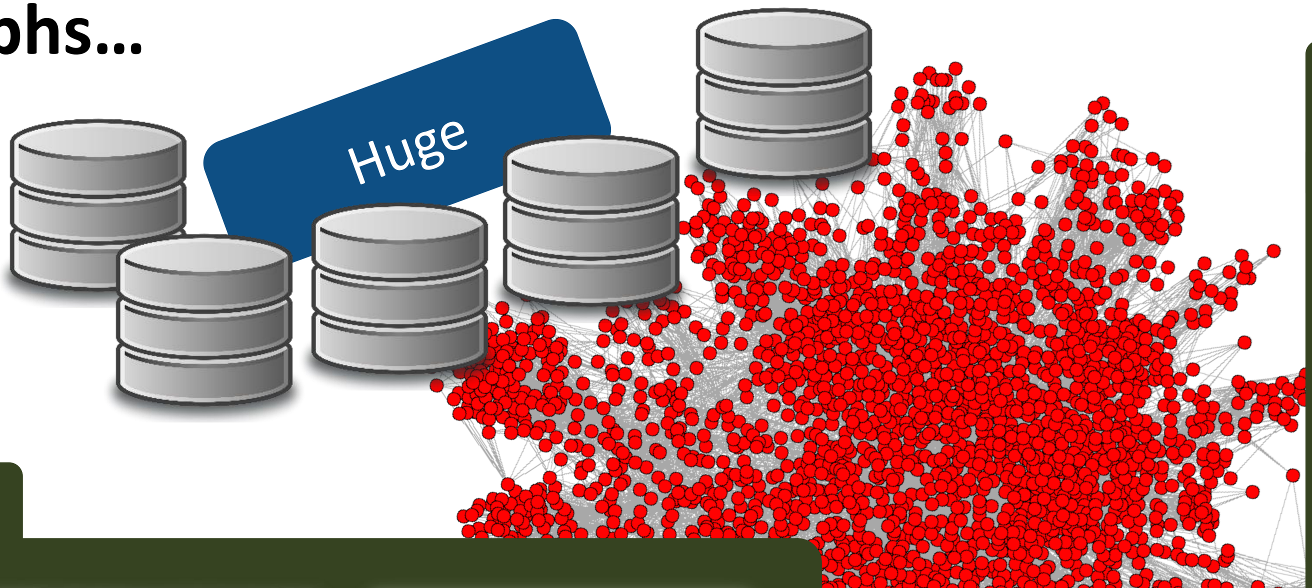
Graph500 Benchmark

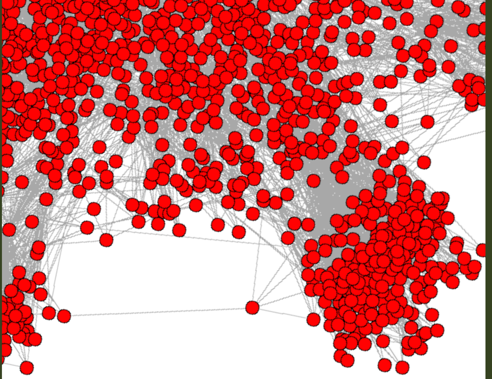
ON	COUNTRY	YEAR	NUMBER OF NODES	NUMBER OF CORES	SCALE	GTEPS
e Hyogo	Japan	2011	82944	663552	40	38621.4
i	China	2015	40768	10599680	40	23755.7
rmore CA	USA	2012	98304	1572864	41	23751
ago IL	USA	2012	49152	786432	40	14982

Webgraph datasets

Graph	Crawl date	Nodes	Arcs
uk-2014	2014	787 801 471	47 614 527 250
eu-2015	2015	1 070 557 254	91 792 261 600
gsh-2015	2015	988 490 691	33 877 399 152
uk-2014-host	2014	4 769 354	50 829 923
eu-2015-host	2015	11 264 052	386 915 963
gsh-2015-host	2015	68 660 142	1 802 747 600
uk-2014-tpd	2014	1 766 010	18 244 650
eu-2015-tpd	2015	6 650 532	170 145 510
gsh-2015-tpd	2015	30 809 122	602 119 716
clueweb12	2012	978 408 098	42 574 107 469
uk-2002	2002	18 520 486	298 113 762

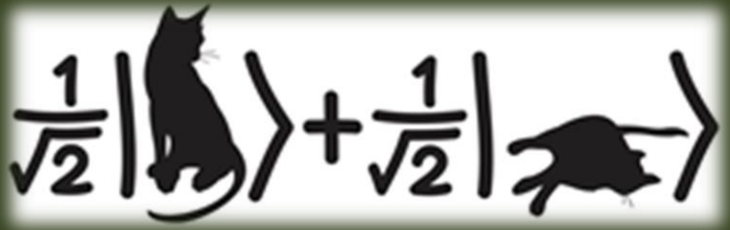
Large graphs…

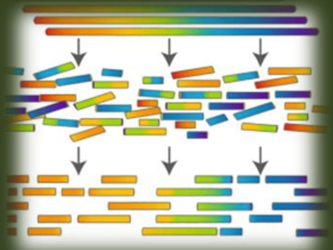




Running on…

Used in…





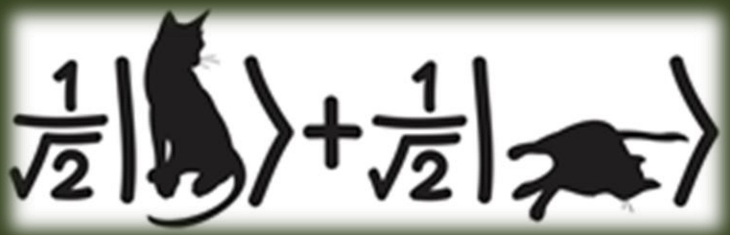
Large graphs…

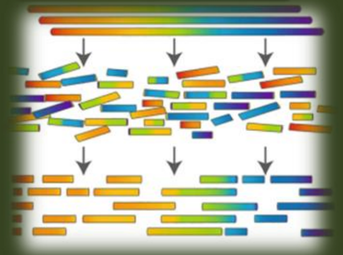
Huge

Running on…

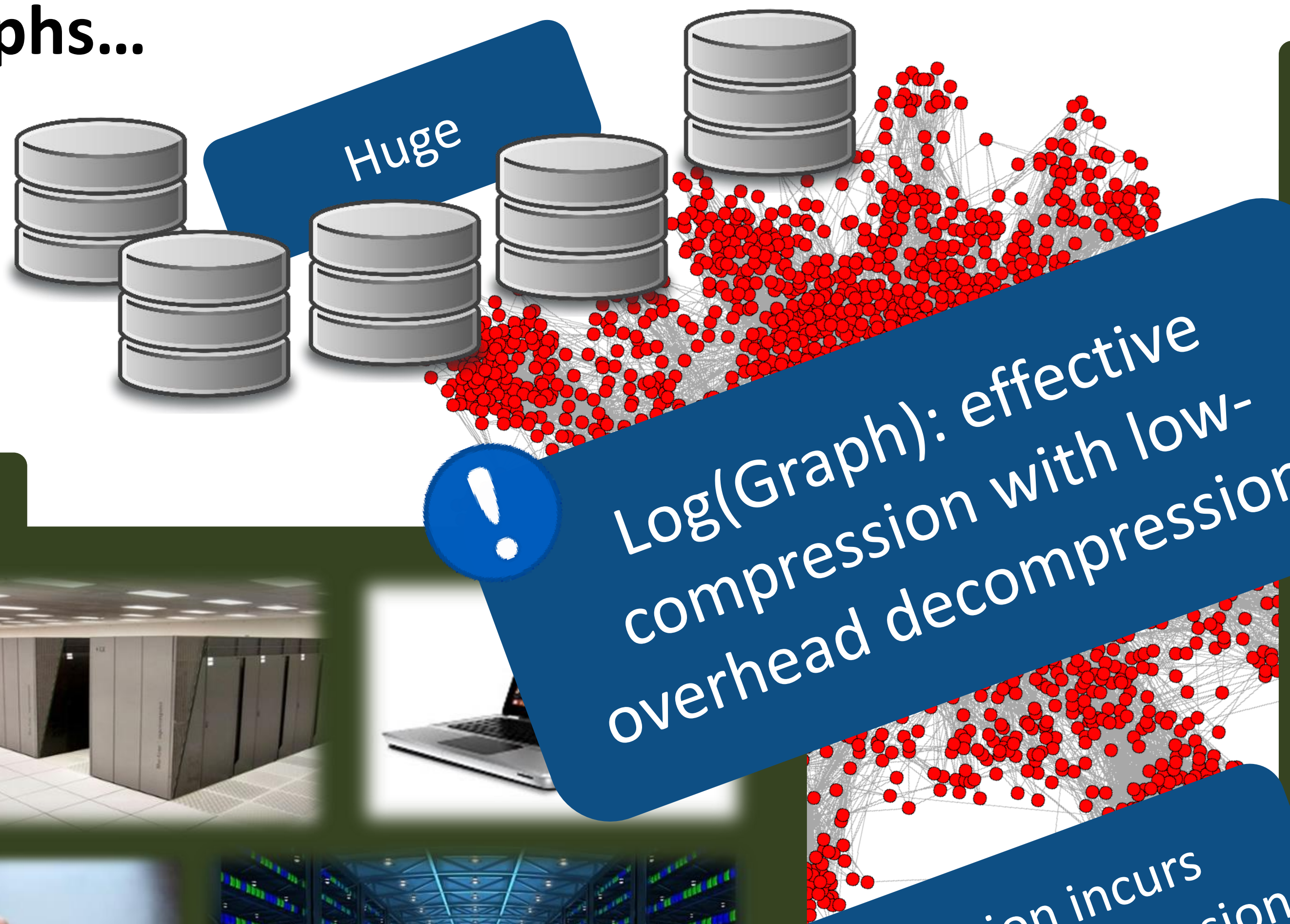
Compression incurs expensive decompression

Used in…





Large graphs…



Huge

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

Running on…

Compression incurs expensive decompression

Used in…

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** Ω

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

The storage **lower bound** Ω

Which one? ☺

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

The storage **lower bound** Ω

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** Ω

Which one? ☺

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

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
…

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

The storage **lower bound** Ω

Which one? ☺

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

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
...

Key idea

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

The storage **lower bound** Ω

Which one? ☺

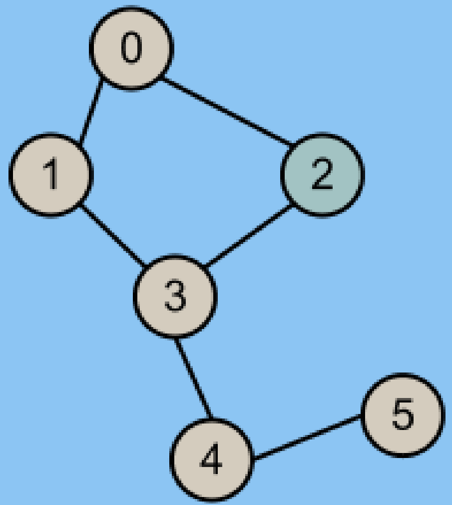
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
...

Key idea

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



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

The storage **lower bound** Ω

Which one? ☺

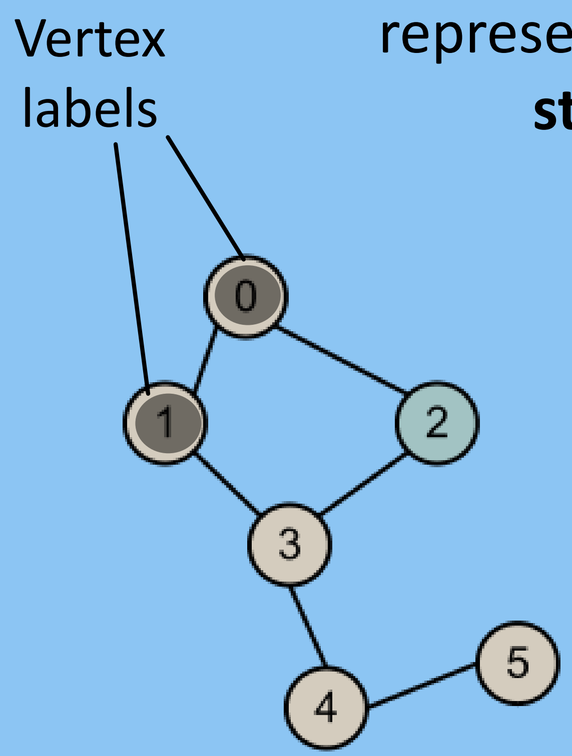
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
…

Key idea

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



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

The storage **lower bound** Ω

Which one? ☺

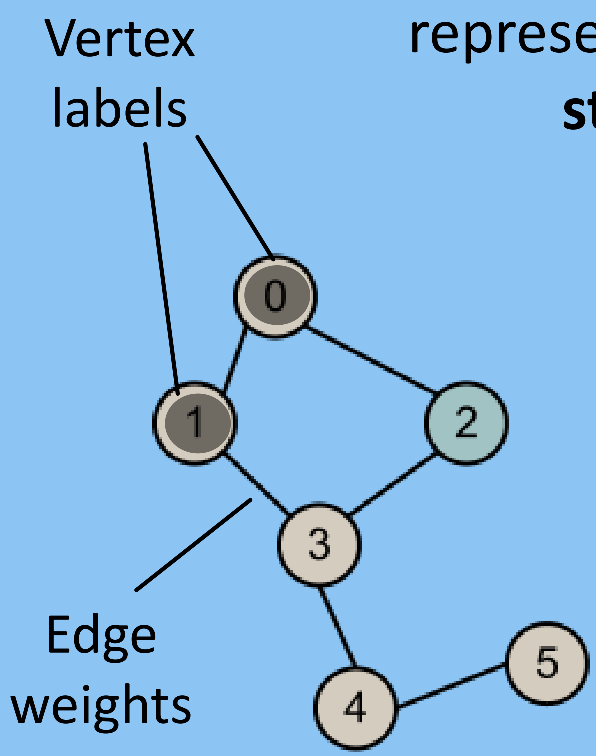
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
$\dots$

Key idea

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



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

The storage **lower bound** Ω

Which one? ☺

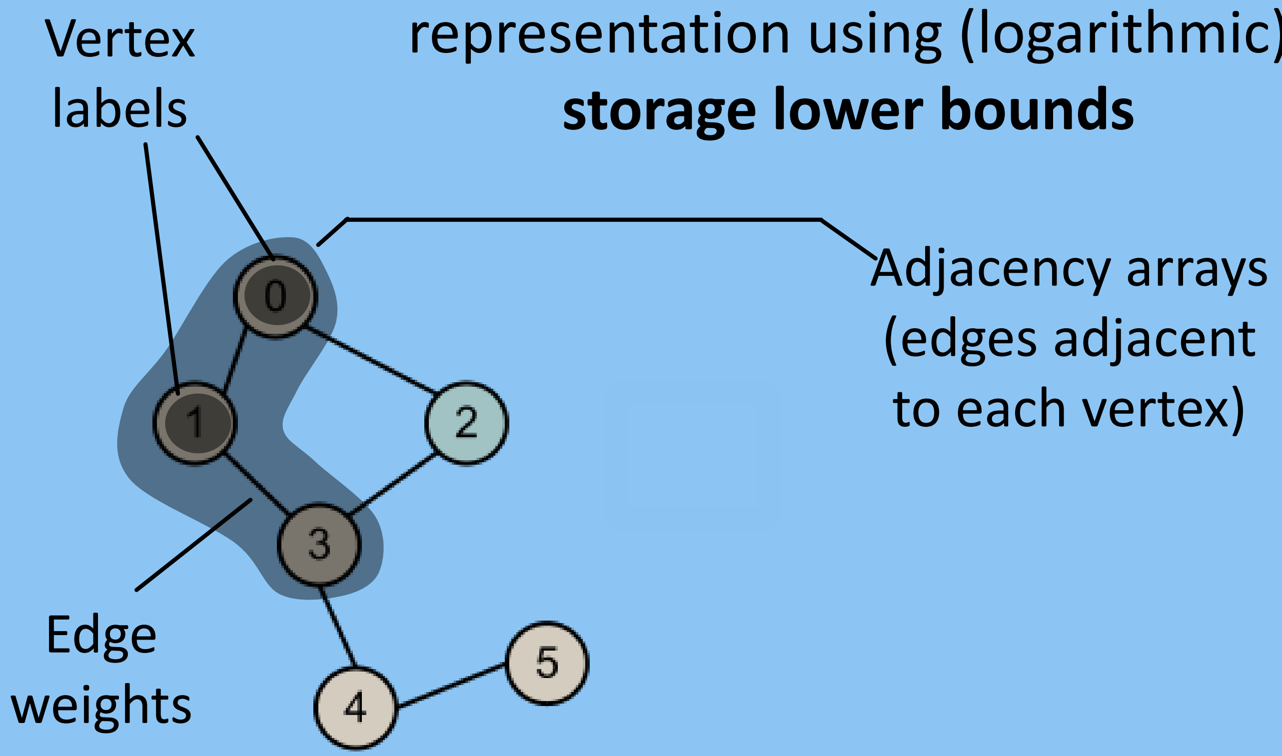
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
$\dots$

Key idea

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



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

The storage **lower bound** Ω

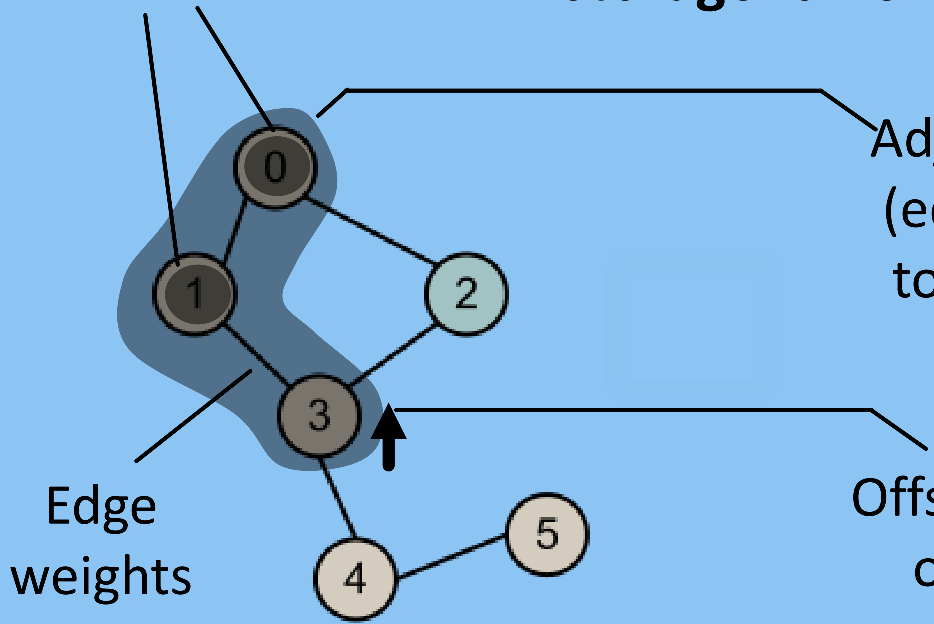
Which one? ☺

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

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

Key idea

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



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

The storage **lower bound** Ω

Which one? ☺

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

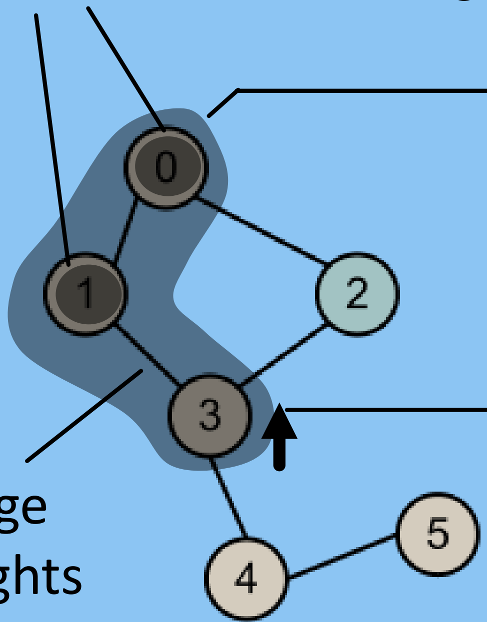
$S = \{x_1, x_2, x_3, \ldots\}$

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

Key idea

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

Log (Vertex labels **)**



Adjacency arrays (edges adjacent to each vertex)

Edge weights

Offsets (locations) of adj. arrays

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

The storage **lower bound** Ω

Which one? ☺

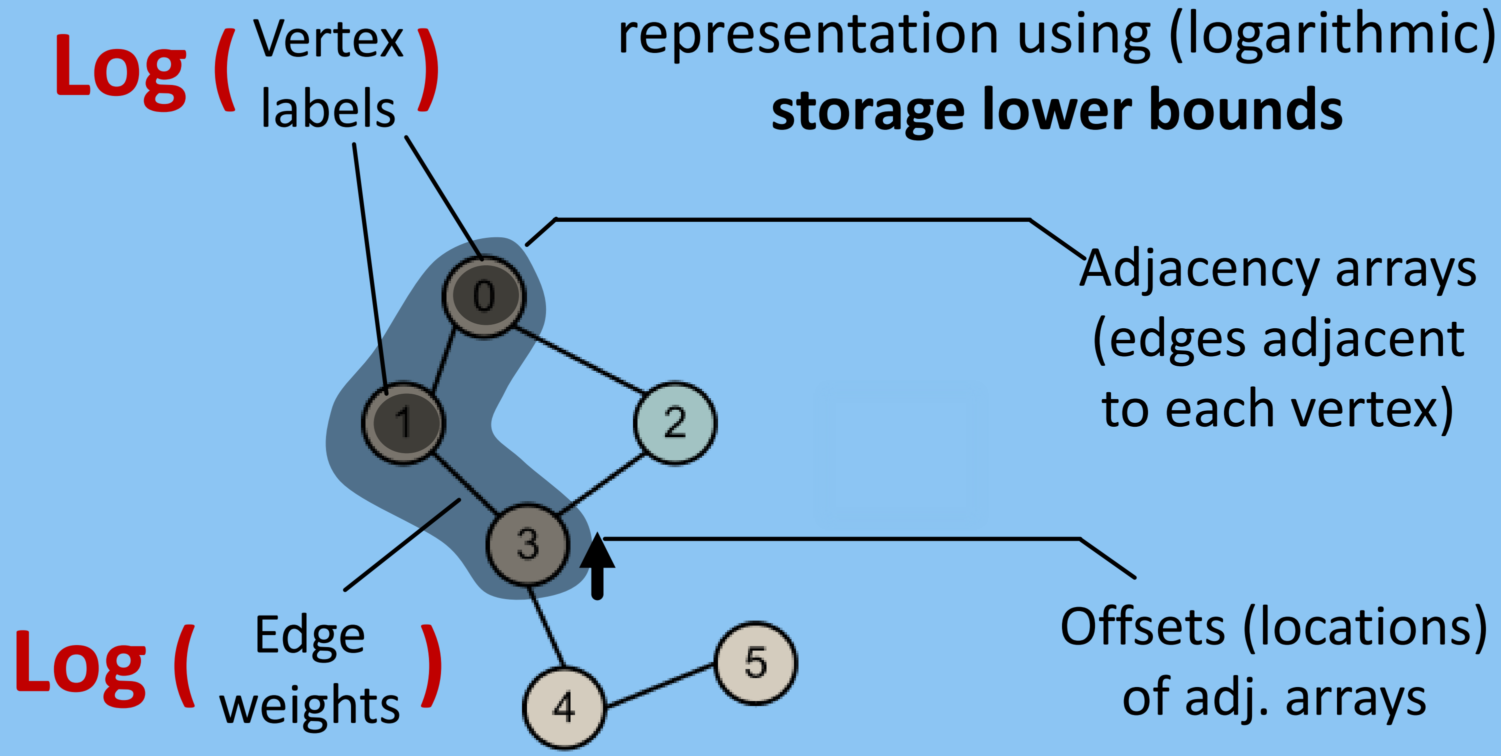
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
...

Key idea

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



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

The storage **lower bound** Ω

Which one? ☺

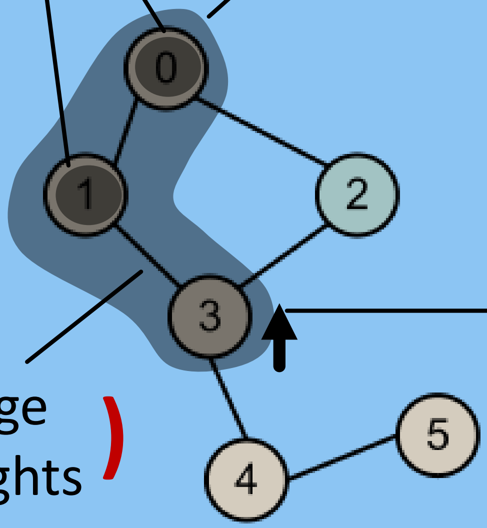
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

$$S = \{x_1, x_2, x_3, \dots\} \quad \begin{matrix} x_1 \rightarrow 0 \dots 01 \\ x_2 \rightarrow 0 \dots 10 \\ x_3 \rightarrow 0 \dots 11 \\ \dots \end{matrix}$$

Key idea

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

Log (Vertex labels **)**



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

Offsets (locations) of adj. arrays

Log (Edge weights **)**

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

The storage **lower bound** Ω

Which one? ☺

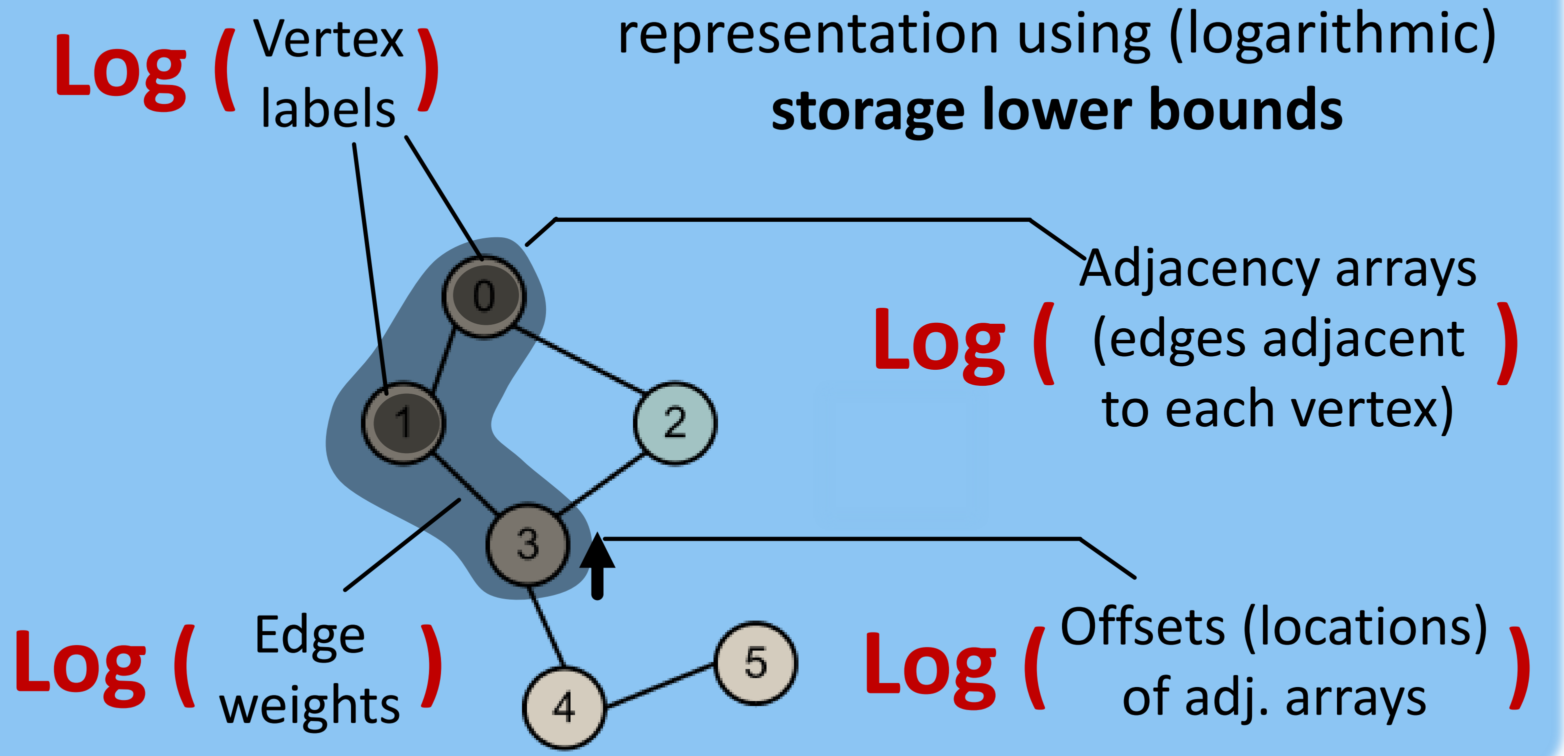
Counting bounds. They are **logarithmic** (one needs at least log|S| bits to store an object from an arbitrary set S)

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

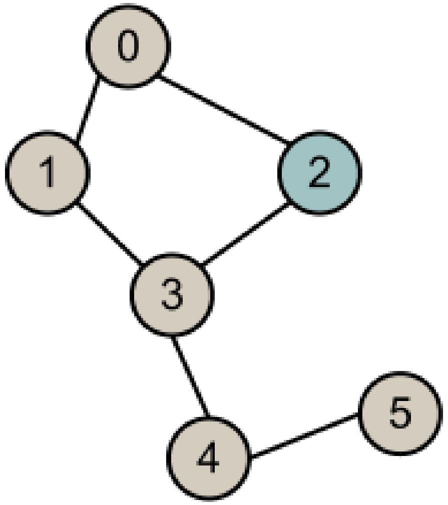
$x_1 \rightarrow 0 \dots 01$
$x_2 \rightarrow 0 \dots 10$
$x_3 \rightarrow 0 \dots 11$
...

Key idea

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

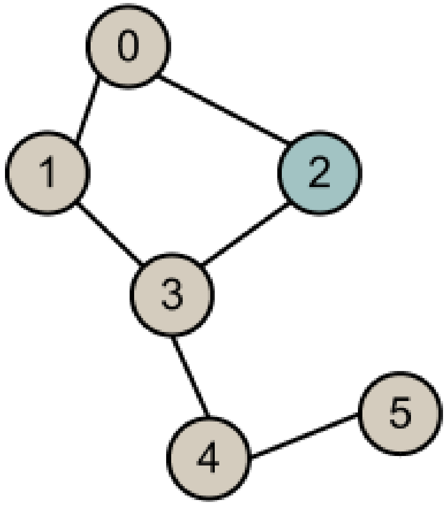


Adjacency Array Graph Representation



ADJACENCY ARRAY GRAPH REPRESENTATION

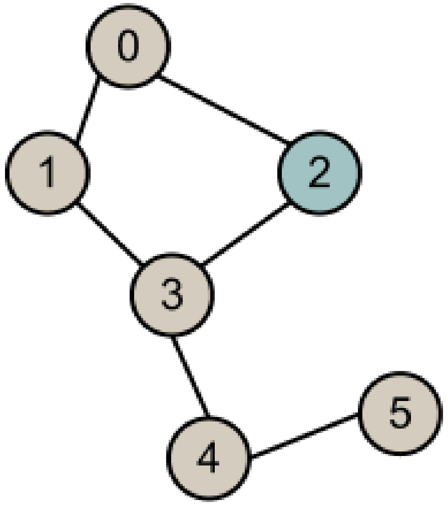
Representation



Adjacency Array Graph Representation

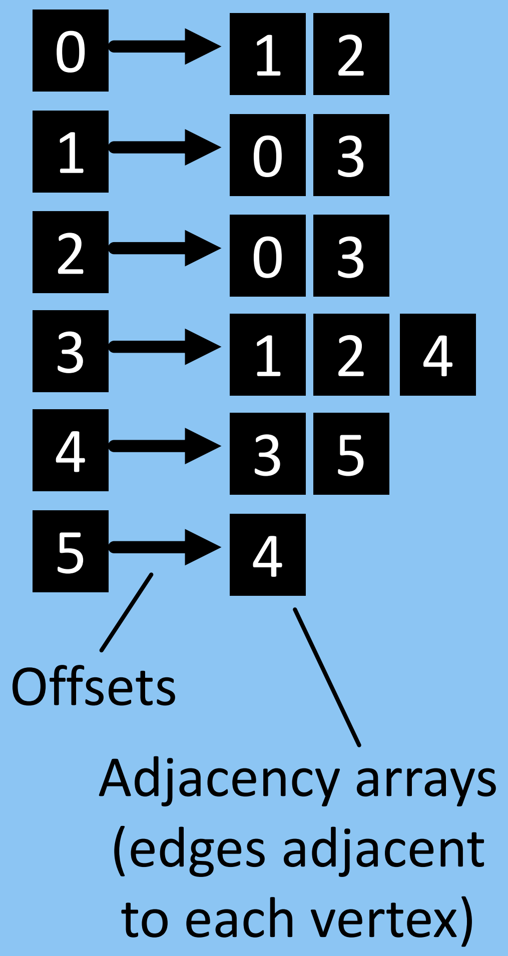
Representation

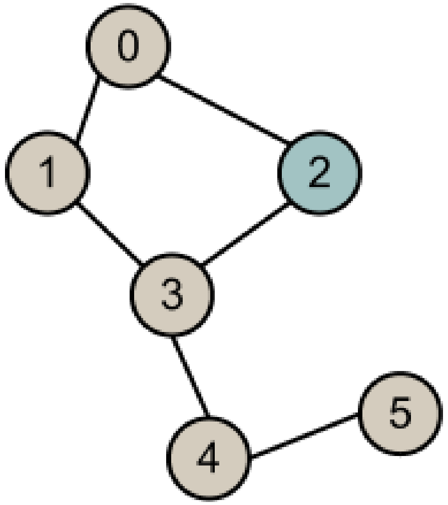
0
1
2
3
4
5



Adjacency Array Graph Representation

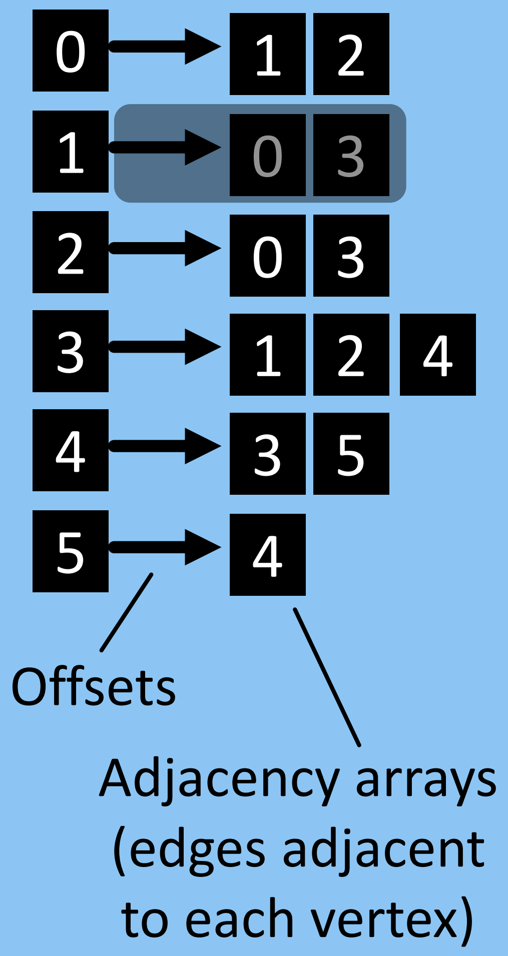
Representation

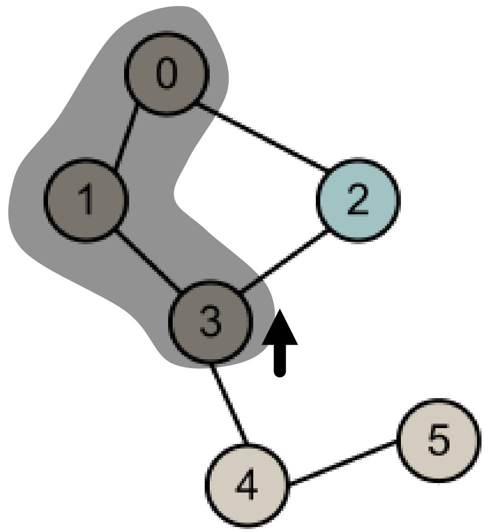




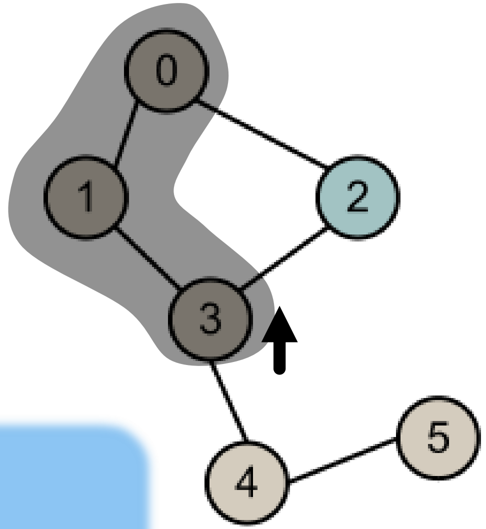
Adjacency Array Graph Representation

Representation

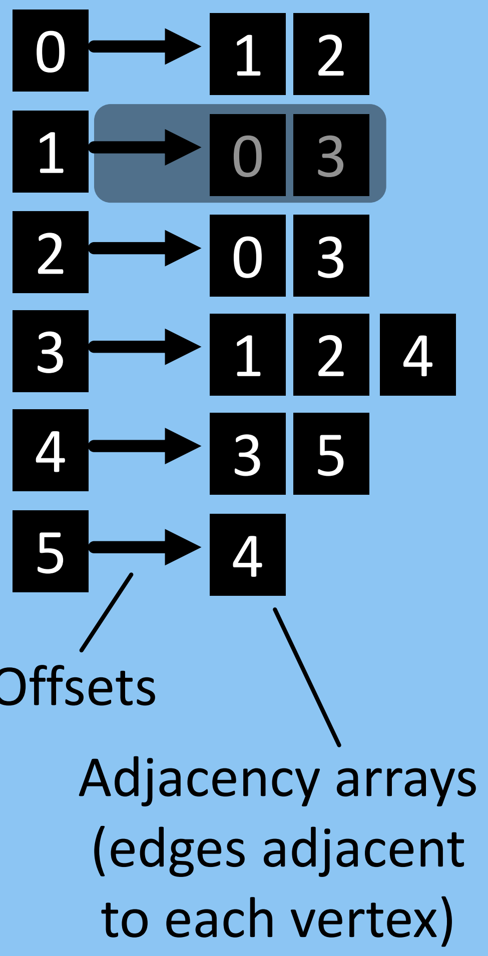




Adjacency Array Graph Representation

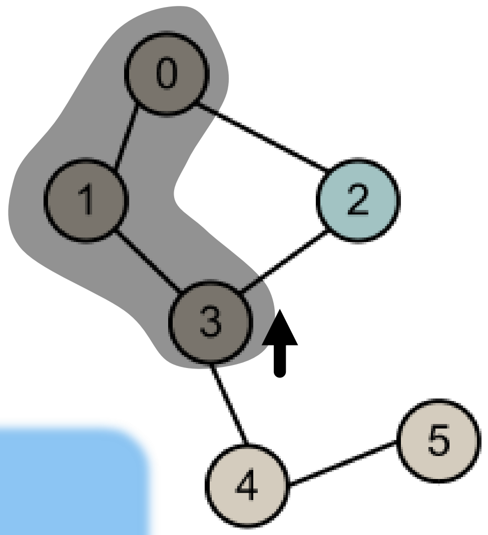


Representation

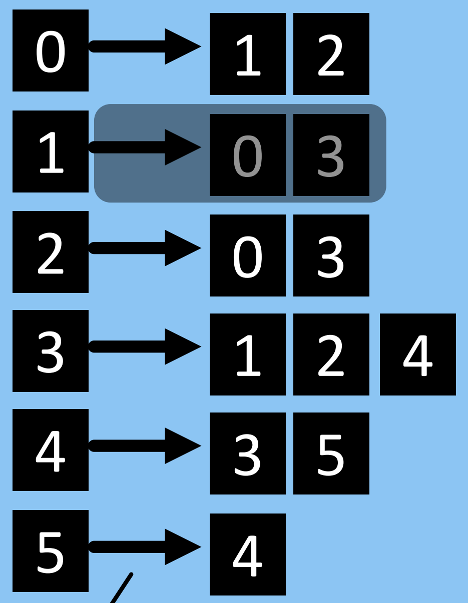


Physical realization

Adjacency Array Graph Representation



Representation



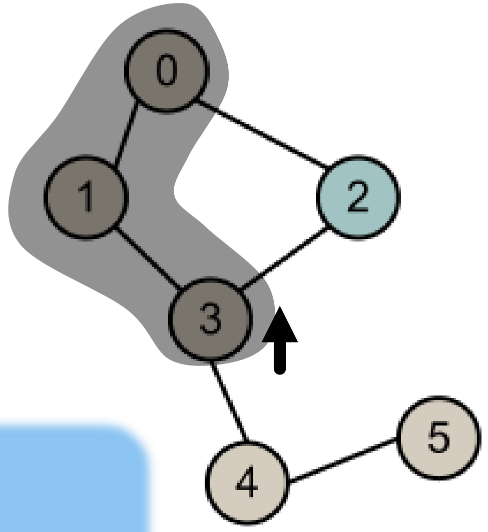
Offsets

Adjacency arrays (edges adjacent to each vertex)

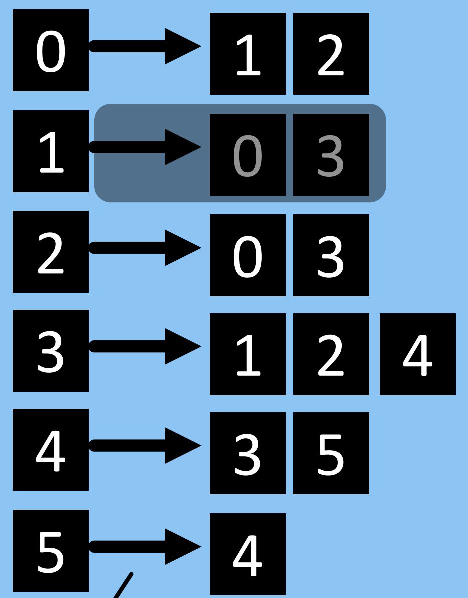
Physical realization

Adjacency arrays (one contiguous array)

Adjacency Array Graph Representation



Representation

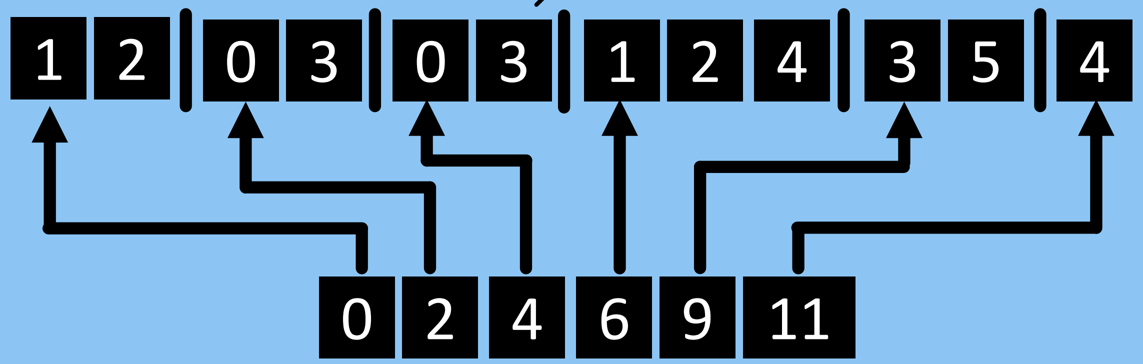


Offsets

Adjacency arrays (edges adjacent to each vertex)

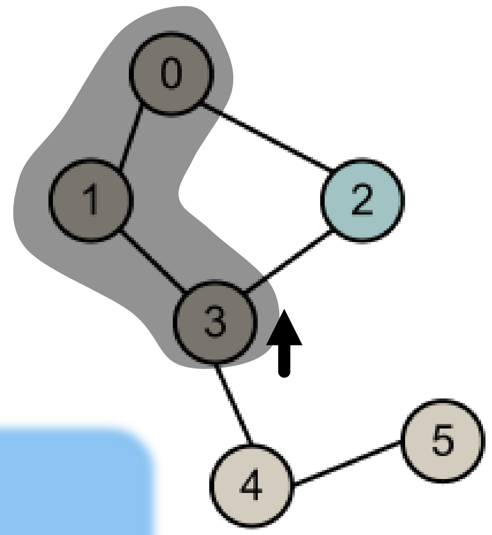
Physical realization

Adjacency arrays (one contiguous array)

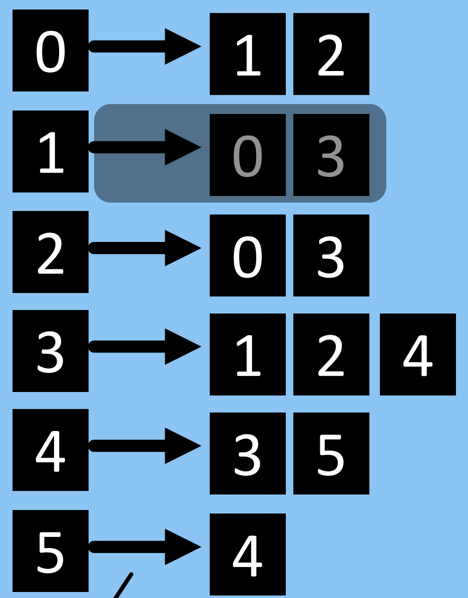


Offsets (another contiguous array)

ADJACENCY ARRAY GRAPH REPRESENTATION



Representation

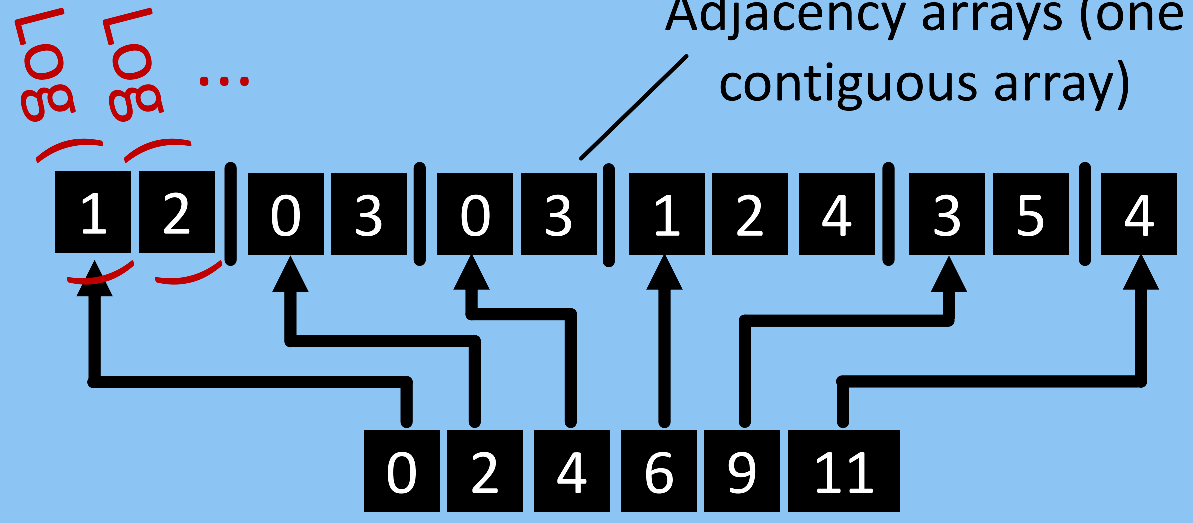


Offsets

Adjacency arrays (edges adjacent to each vertex)

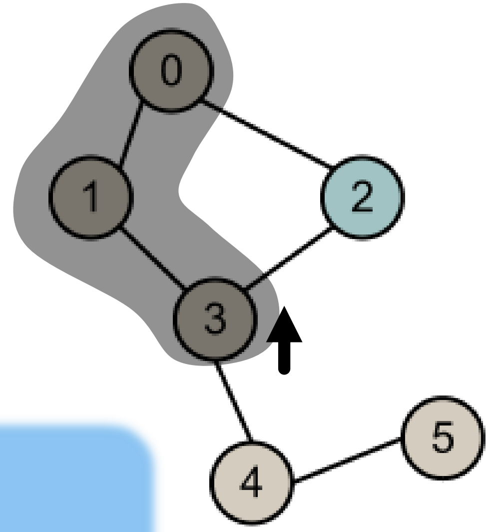
Physical realization

Adjacency arrays (one contiguous array)

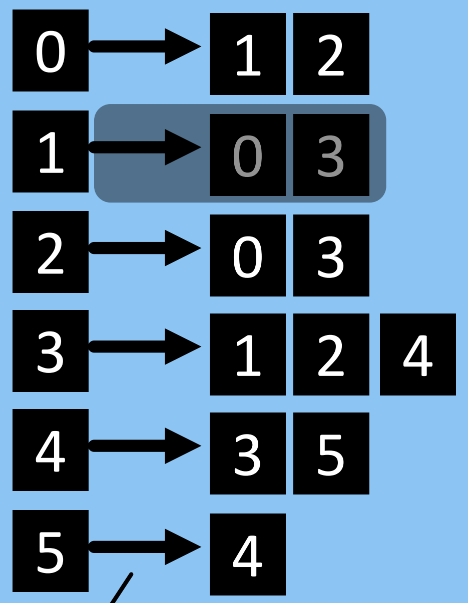


Offsets (another contiguous array)

Adjacency Array Graph Representation



Representation

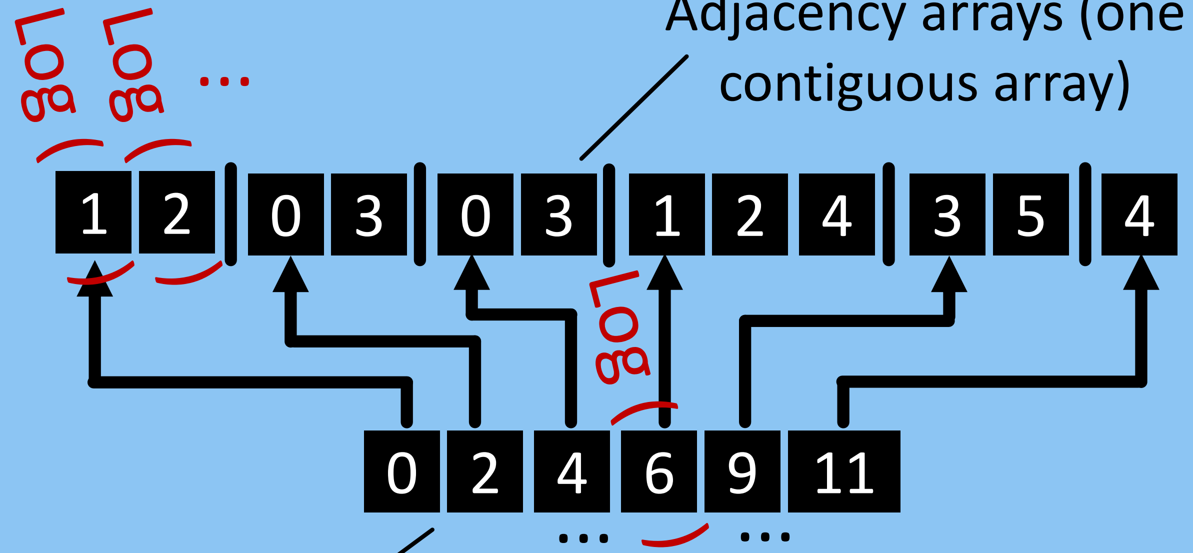


Offsets

Adjacency arrays (edges adjacent to each vertex)

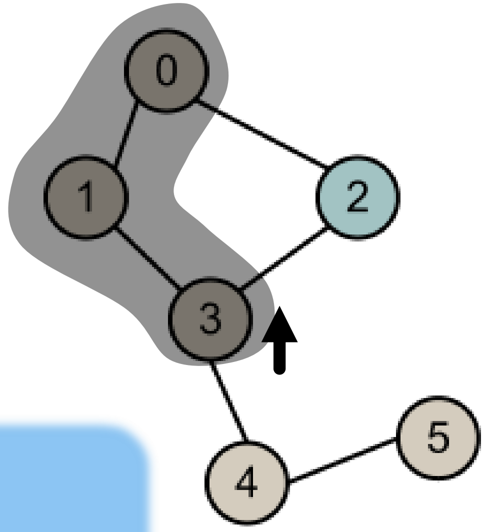
Physical realization

Adjacency arrays (one contiguous array)

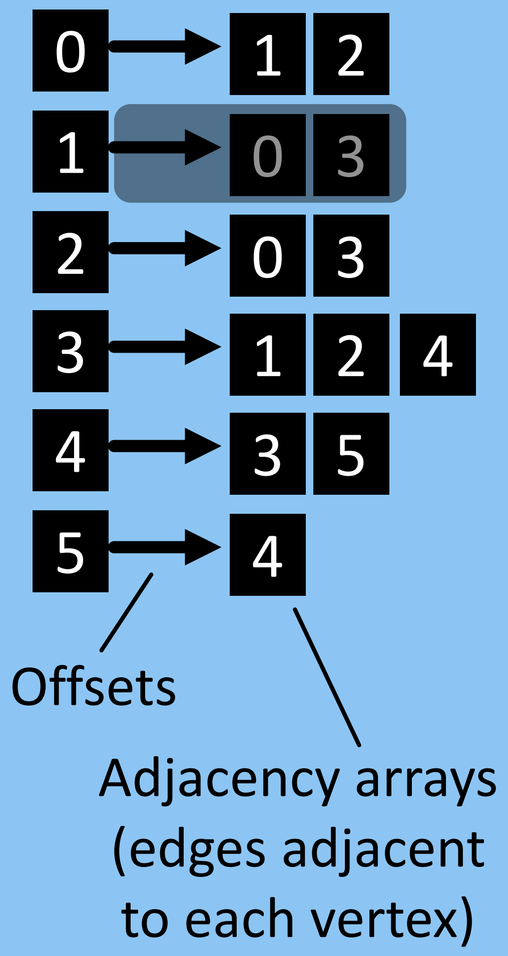


Offsets (another contiguous array)

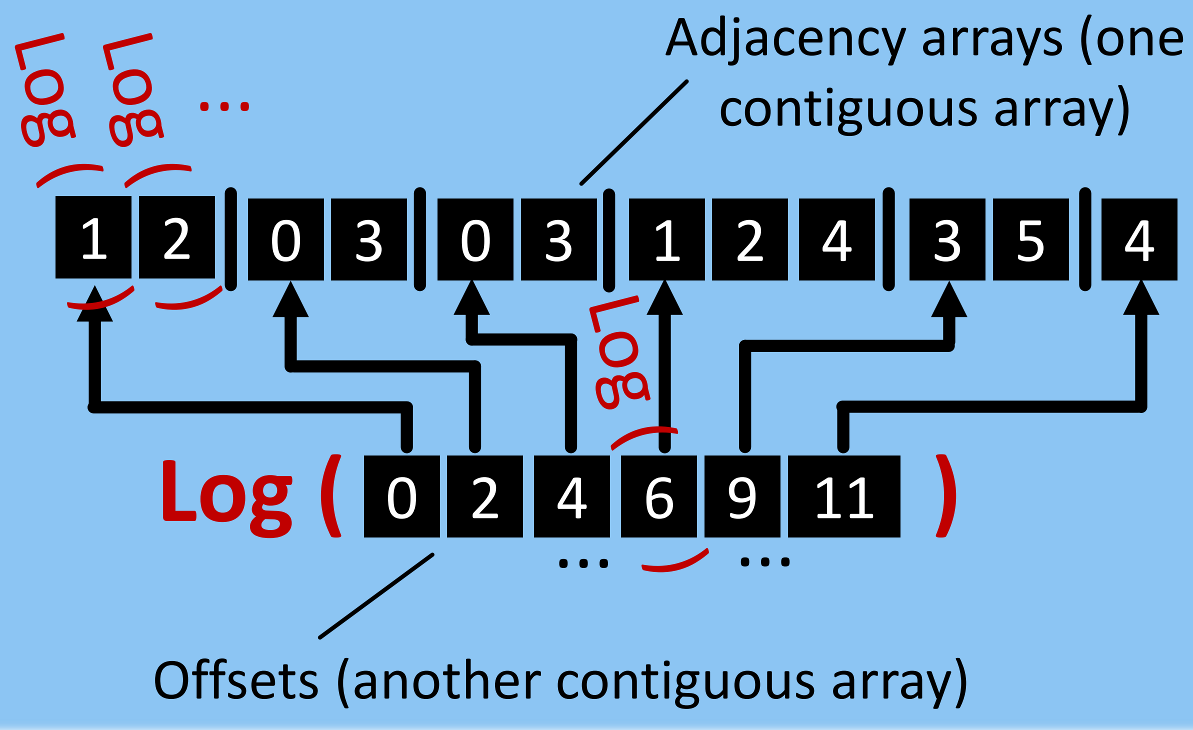
Adjacency Array Graph Representation



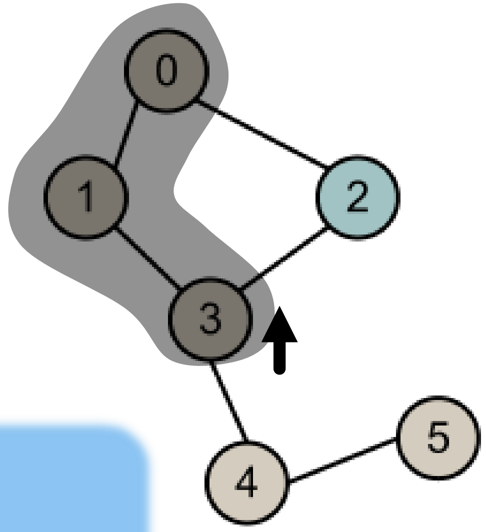
Representation



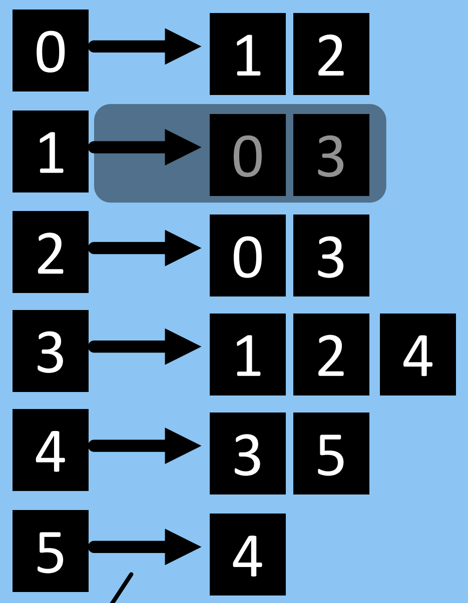
Physical realization



Adjacency Array Graph Representation



Representation

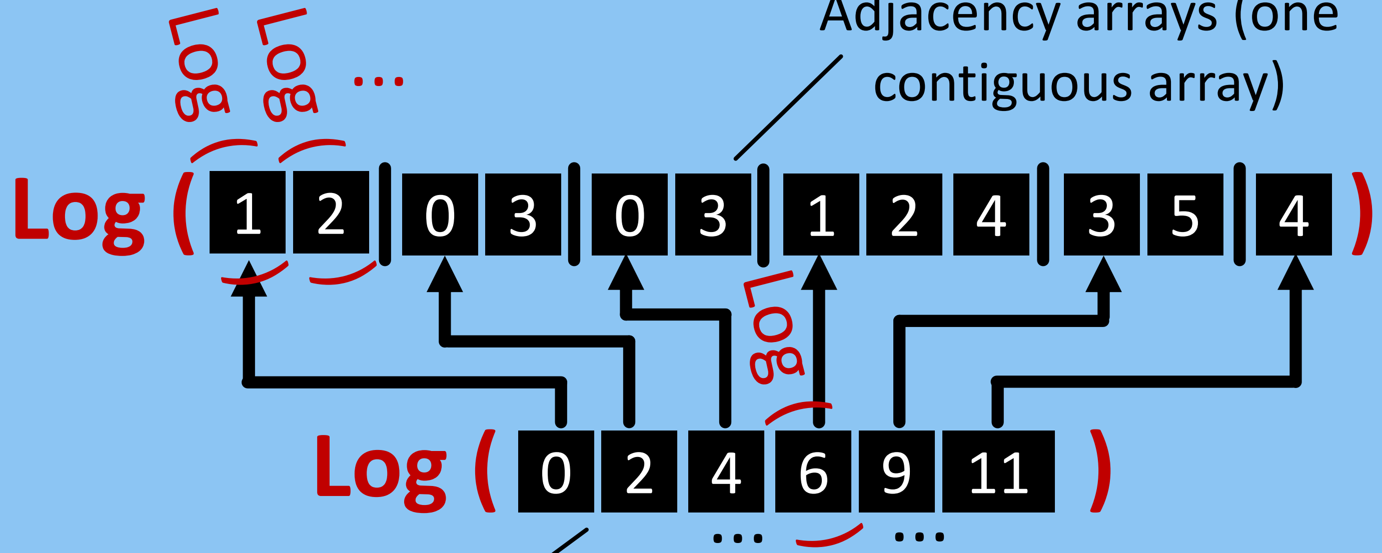


Offsets

Adjacency arrays (edges adjacent to each vertex)

Physical realization

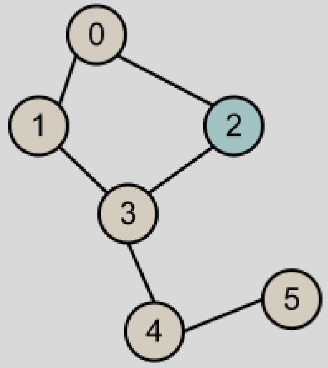
Adjacency arrays (one contiguous array)



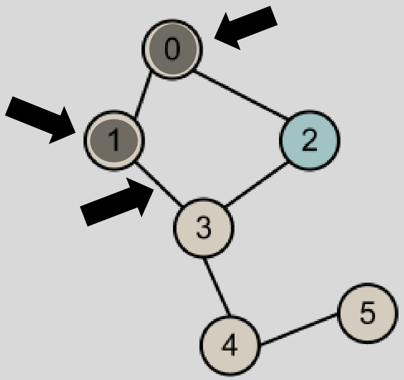
Offsets (another contiguous array)

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

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



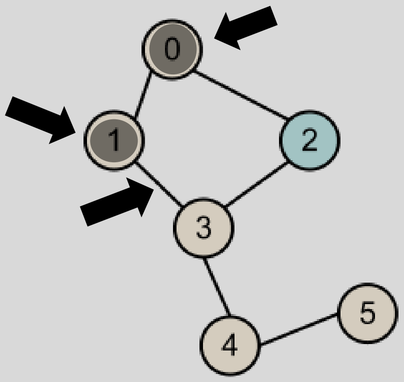
1 **Log (** Vertex labels **), Log (** Edge weights **)**



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,
$\widehat{N_v}$: maximum among N_v

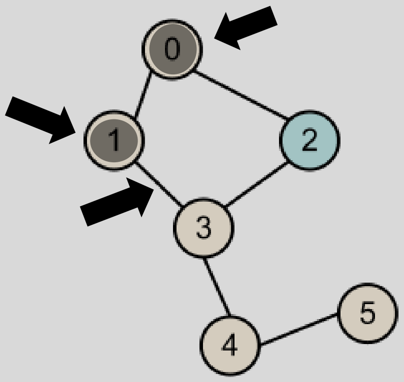


1 Log (Vertex labels), Log (Edge weights)

Lower bounds (global)

Symbols

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

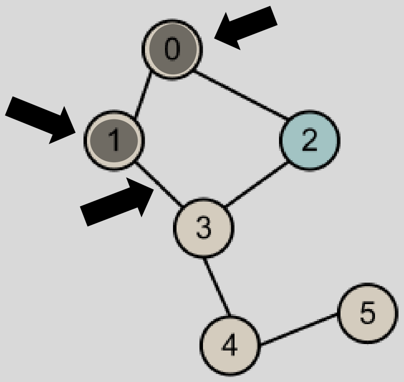


1 **Log (Vertex labels), Log (Edge weights)**

Lower bounds (global)

$$\lceil \log n \rceil$$

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)**

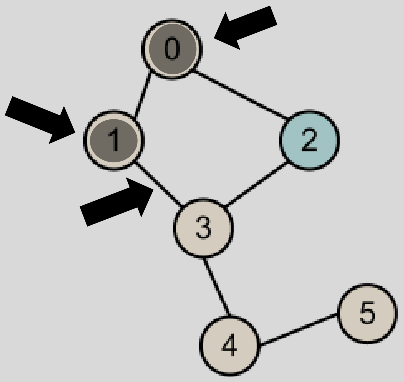
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

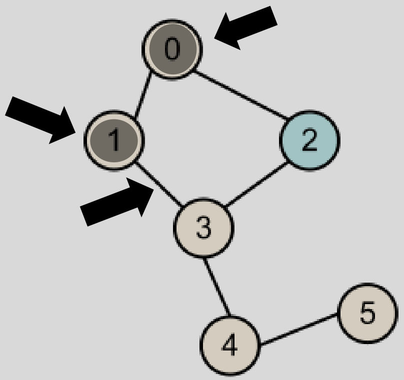
Symbols

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)

Symbols



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

Lower bounds (global)

$$\lceil \log n \rceil$$

This is it?
Not really ☺

Lower bounds (local)

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

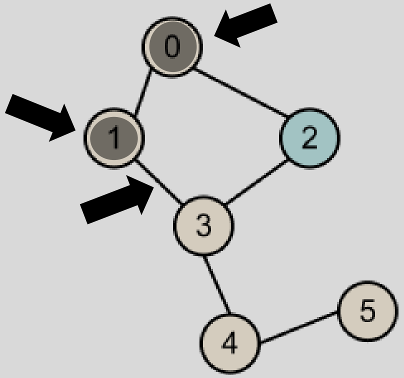
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Symbols

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



Lower bounds (local)

Assume:

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

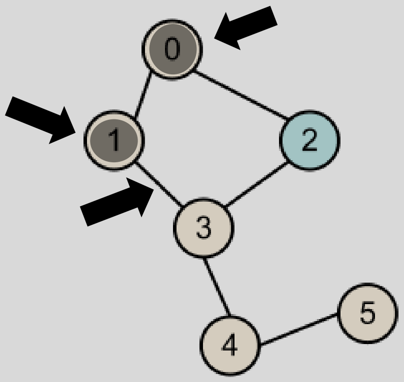
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Symbols

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



Lower bounds (local)

Assume:

- a graph, e.g., $V = \{1, \ldots, 2^{22}\}$

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

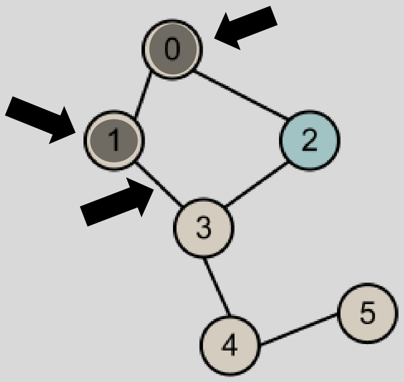
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Symbols

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



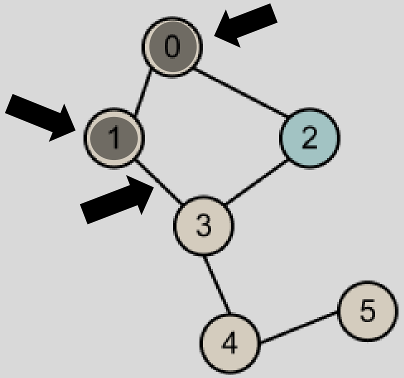
Lower bounds (local)

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

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Lower bounds (local)

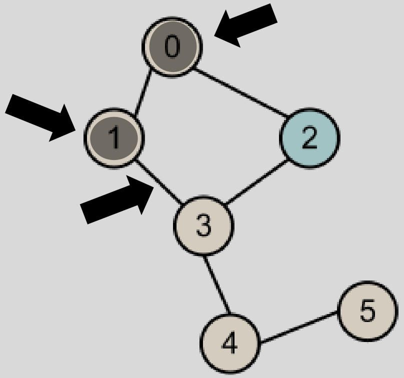
Assume:

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

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (global)

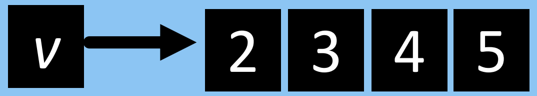
$\lceil \log n \rceil$

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: $\widehat{N_v} \ll n$



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

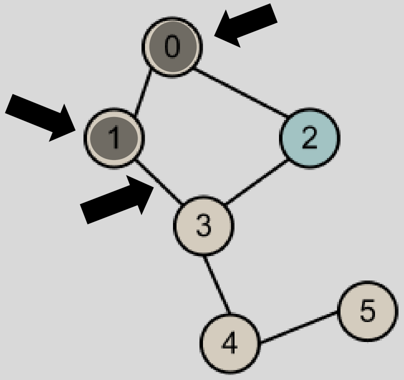
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Symbols

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

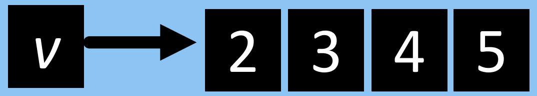


Lower bounds (local)

Assume:

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

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



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

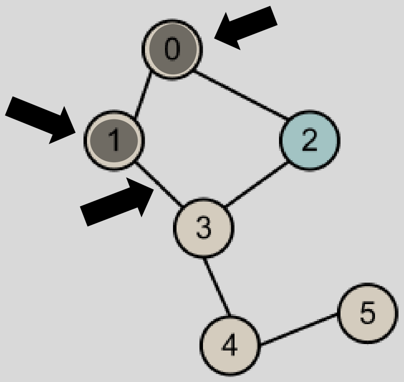
Lower bounds (global)

$\lceil \log n \rceil$

This is it?
Not really ☺

Symbols

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



Lower bounds (local)

Assume:

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

v → 2 | 3 | 4 | 5

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

v → 0...10 | 0...11 | 0...100 | 0...101

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

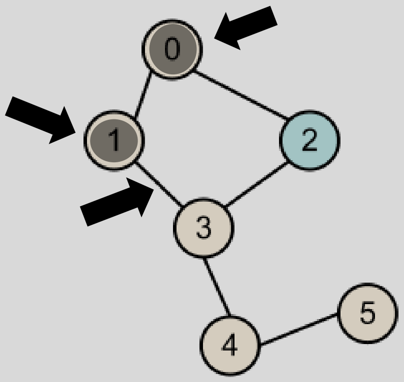
Lower bounds (global)

$$\lceil \log n \rceil$$

This is it?
Not really ☺

Symbols

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



Lower bounds (local)

Assume:

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

v → 2 | 3 | 4 | 5

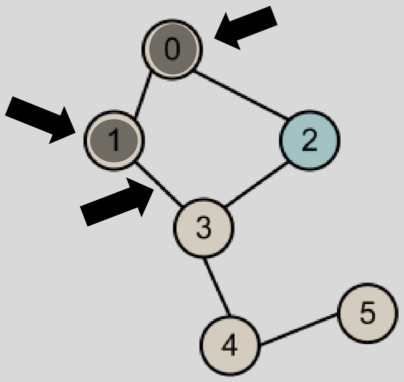
$$\lceil \log 2^{22} \rceil = 22$$

v → 0...10 | 0...11 | 0...100 | 0...101

19 zeros!

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,
$\widehat{N_v}$: maximum among N_v

Lower bounds (global)

$\lceil \log n \rceil$

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: $\widehat{N_v} \ll n$

v → 2 | 3 | 4 | 5

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

v → 0...10 | 0...11 | 0...100 | 0...101

19 zeros!

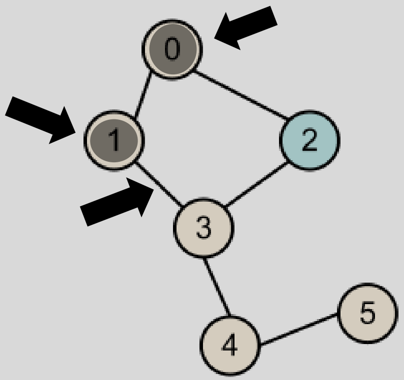
Thus, use the local bound $\lceil \log \widehat{N_v} \rceil$

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

This is it?
Not really ☺

Symbols

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



Lower bounds (local): problem

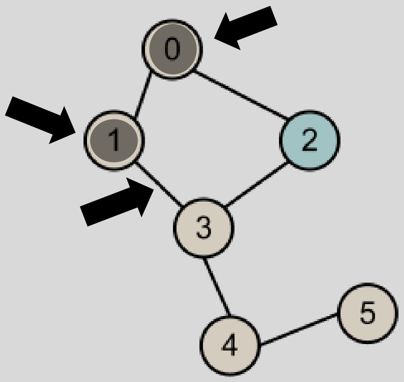
- a graph, e.g., $V = \{1, \dots, 2^{22}\}$
- A vertex v with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\widehat{N_v} \ll n$

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

This is it?
Not really ☺

Symbols

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



Lower bounds (local): problem

What if:

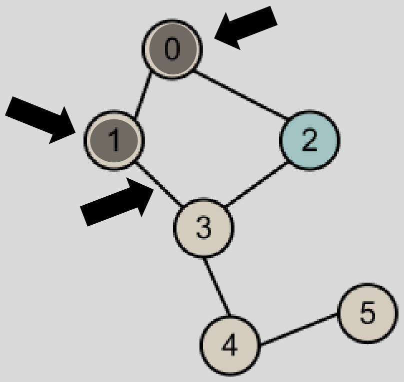
- a graph, e.g., $V = \{1, \dots, 2^{22}\}$
- A vertex v with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\widehat{N_v} \ll n$

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

This is it?
Not really ☺

Symbols

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



Lower bounds (local): problem

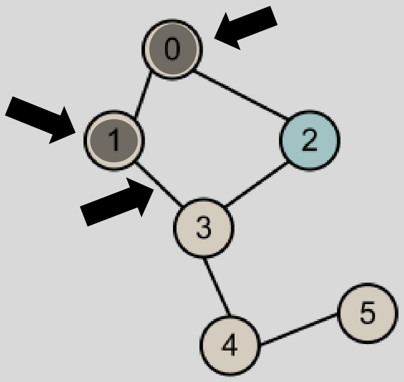
What if:

- a graph, e.g., $V = \{1, \dots, 2^{22}\}$
- 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:

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,
$\widehat{N_v}$: maximum among N_v



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:

v → | 2 | 3 | 4 | 5 | 1M |

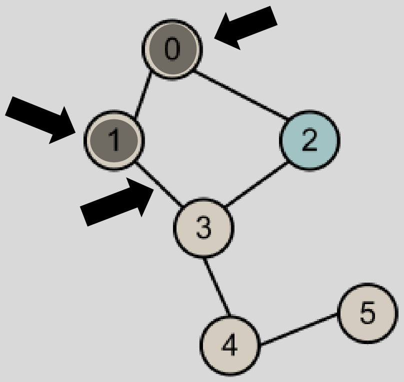
v → | 0...10 | 0...11 | 0...100 | 0...101 |

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

This is it?
Not really ☺

Symbols

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

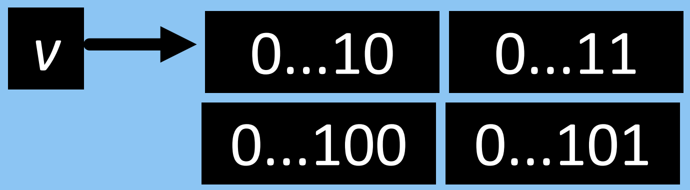


Lower bounds (local): problem

What if:

- a graph, e.g., $V = \{1, \dots, 2^{22}\}$
- 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:

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

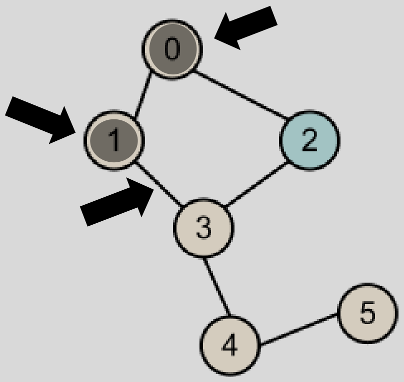


1 Log (Vertex labels), Log (Edge weights)

This is it?
Not really ☺

Symbols

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



Lower bounds (local): problem

What if:

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

v → | 2 | 3 | 4 | 5 | 1M |

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

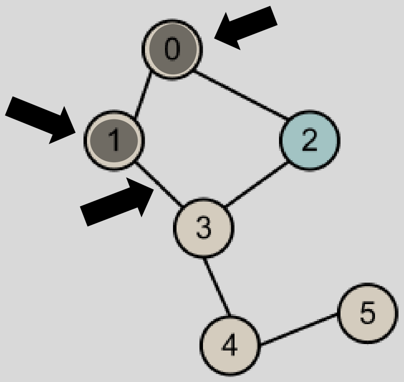
v → | 0...10 | 0...11 |
| 0...100 | 0...101 |

17 zeros!

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

Symbols

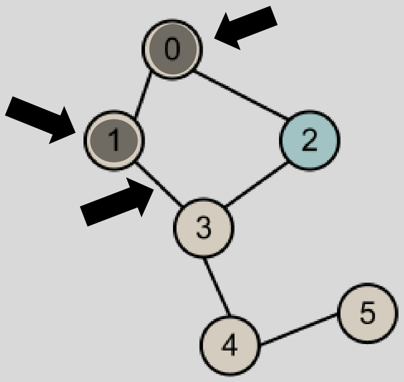
n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v



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

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

Symbols



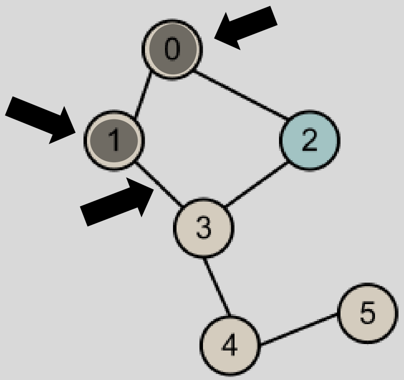
n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v

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,
$\widehat{N_v}$: maximum among N_v



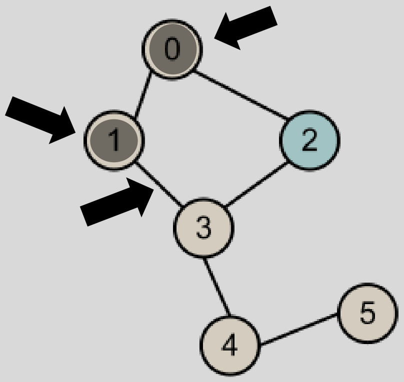
Lower bounds (local) enhanced with ILP

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

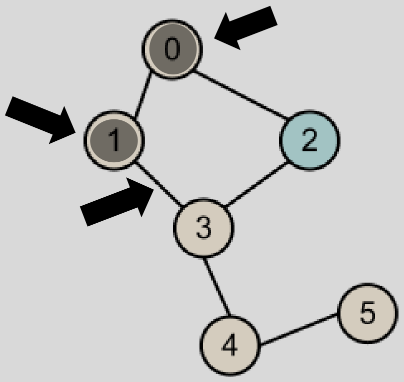
Permute vertex labels **to reduce such maximum labels** in as many neighborhoods as possible

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

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

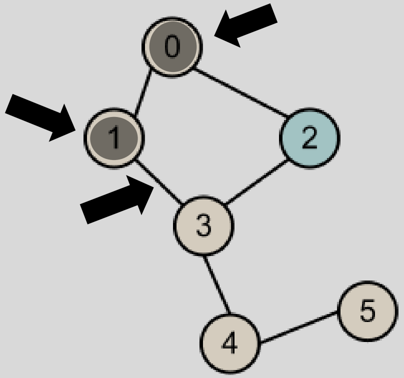
v	→	2	3	4	5	1M

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

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

2	3	4	5	1M

v →

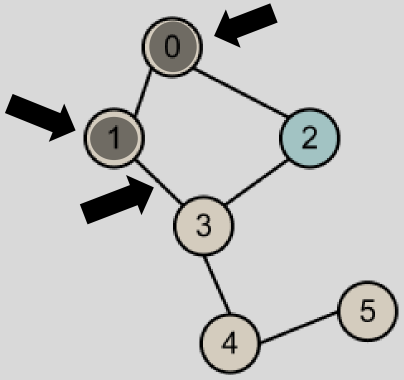
2	3	4	5	1M

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

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

Permute(2 3 4 5 1M) =

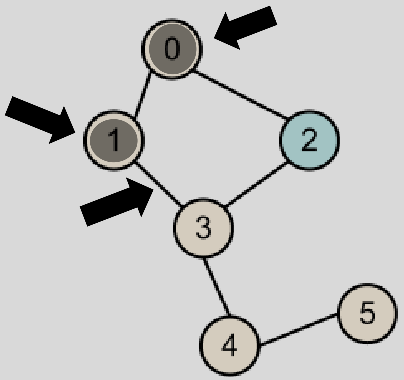
(simultaneously for all other neighborhoods)

v → 2 3 4 5 1M

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,
$\widehat{N_v}$: maximum among N_v

Lower bounds (local) enhanced with ILP

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

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

(simultaneously for all other neighborhoods)

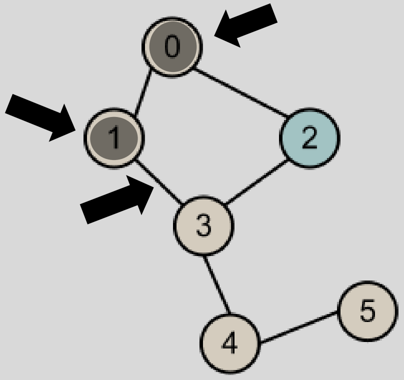
$\leq 100?$

v → 2 3 4 5 1M

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 ,
- $\widehat{N_v}$: maximum among N_v

Lower bounds (local) enhanced with ILP

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

v → 2 | 3 | 4 | 5 | 1M

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

(simultaneously for all other neighborhoods)

$\leq 100?$

Heuristics:

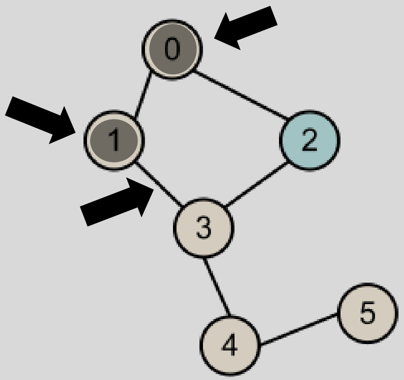
$$\min \sum_{v \in V} \widehat{N_v} \frac{1}{d_v}$$

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

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

v → | 2 | 3 | 4 | 5 | 1M |

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

(simultaneously for all other neighborhoods)

$\leq 100?$

Heuristics:

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

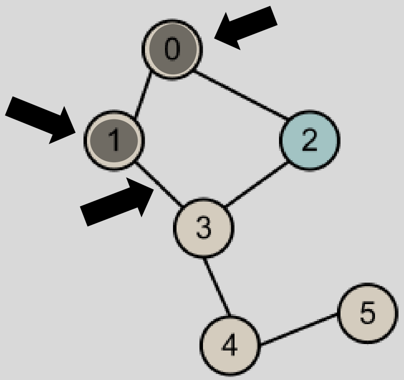
Inverse of the neighborhood size

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,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local) enhanced with ILP

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

v → | 2 | 3 | 4 | 5 | 1M |

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

(simultaneously for all other neighborhoods)

$\leq 100?$

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

Heuristics:

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

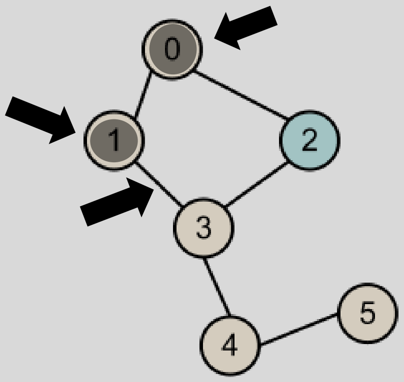
Inverse of the neighborhood size

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

∰ Formal analyses

Symbols

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

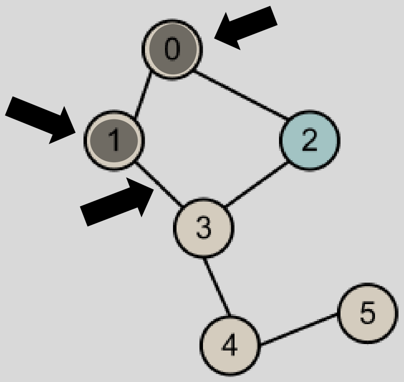


1 **Log (Vertex labels), Log (Edge weights)**

∰ Formal analyses

Symbols

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

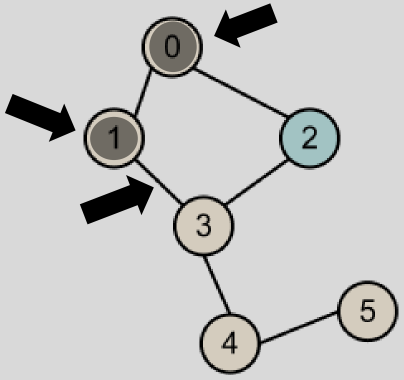
$$\alpha d^{\beta}$$

Symbols

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

n : #vertices,

p, α, β : constants



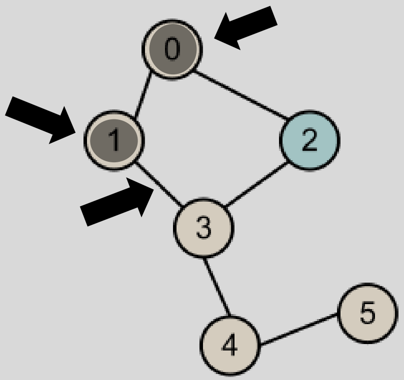
Random uniform graphs

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

Formal analyses

Symbols

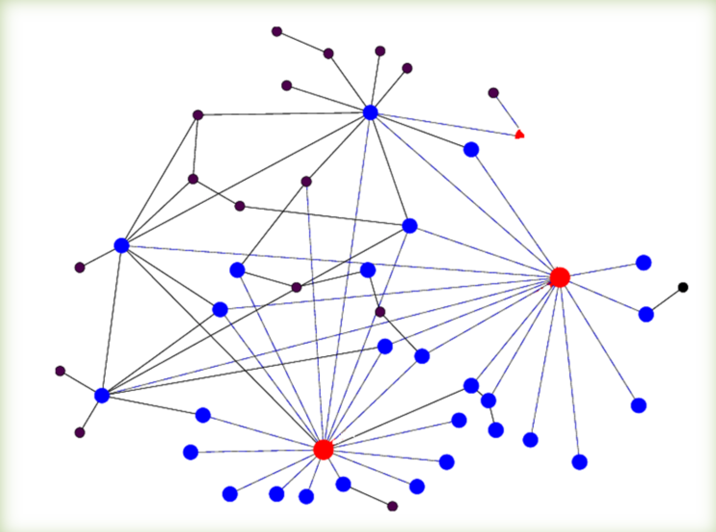
$\widehat{W}$: max edge weight,
n : #vertices,
p, α, β : constants



Power-law graphs

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

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



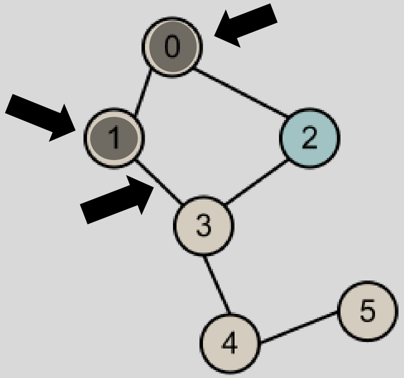
Random uniform graphs

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

∰ Formal analyses

Symbols

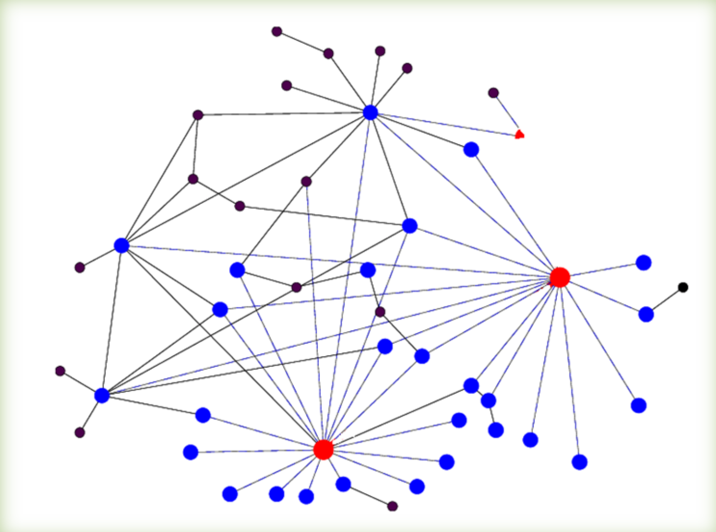
$\widehat{W}$: max edge weight,
n : #vertices,
p, α, β : constants



Power-law graphs

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

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



Random uniform graphs

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

$$pd$$

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

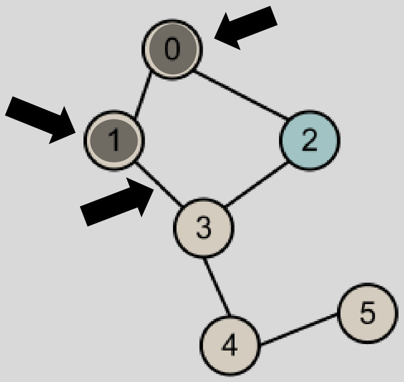
∰ Formal analyses

Symbols

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

n : #vertices,

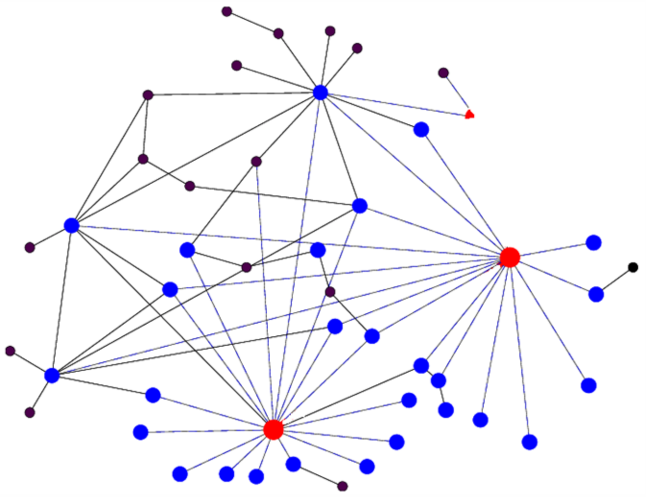
p, α, β : constants



Power-law graphs

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

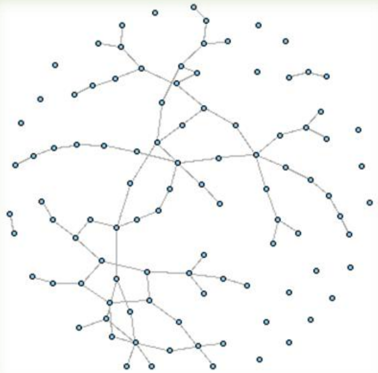
$$\alpha d^{\beta}$$



Random uniform graphs

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

$$pd$$

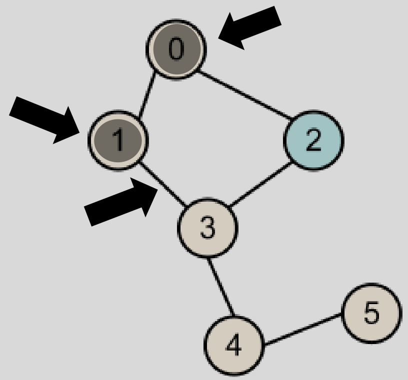


1 Log (Vertex labels), Log (Edge weights)

∰ Formal analyses

Symbols

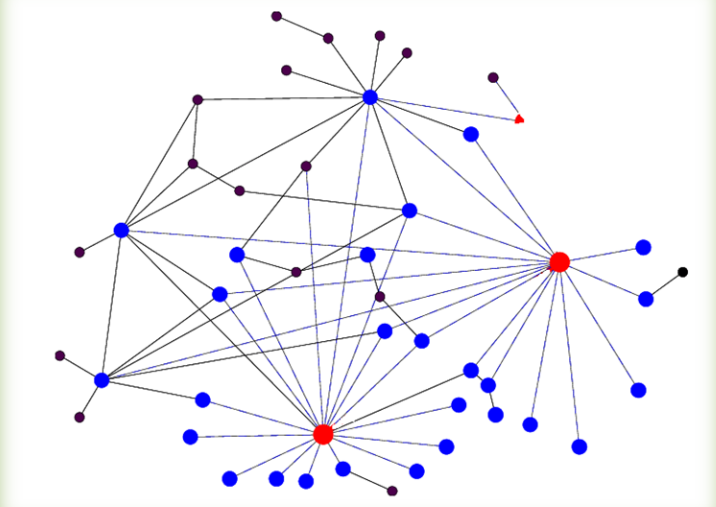
$\widehat{W}$: max edge weight,
n : #vertices,
p, α, β : constants



Power-law graphs

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

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



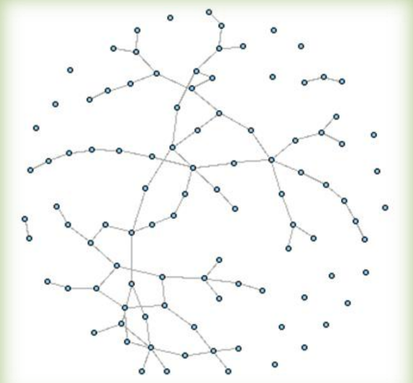
Expected size of the adjacency array

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

Random uniform graphs

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

$$pd$$



Expected size of the adjacency array

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

1 **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 ☺)

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

$$\forall_{v,u\in V}\ (u \in N_v) \ \Rightarrow \left[\mathcal{N}(u) \le \widehat{N}_v\right]$$

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

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

$$|\mathcal{A}| = n\left\lceil \log\frac{n}{\mathcal{H}}\right\rceil + \mathcal{H}\lceil \log \mathcal{H}\rceil$$

$$|\mathcal{A}| = 2m\left(\lceil \log n\rceil + \left\lceil \log \widehat{\mathcal{W}}\right\rceil\right)$$

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

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

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

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

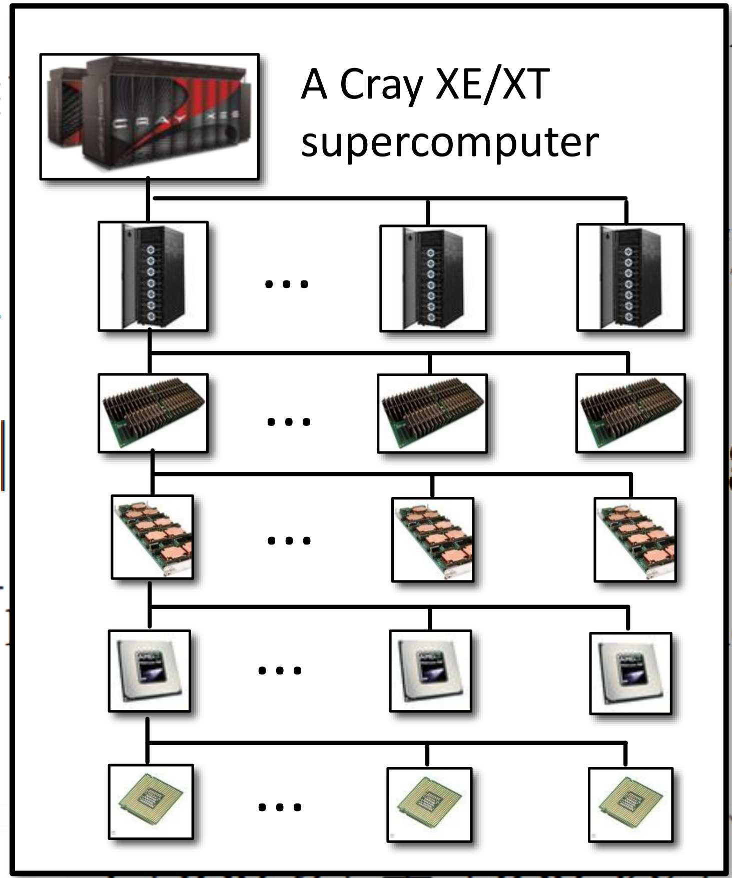
Formal analyses: more (check the paper ☺)

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

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

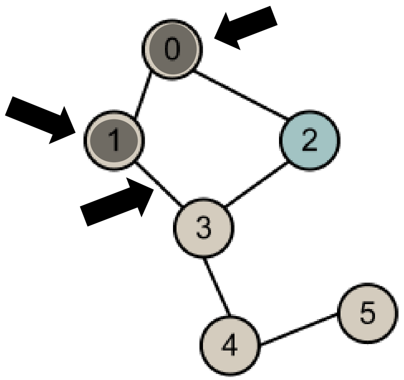
$$|\mathcal{A}| = n\left\lceil \log \frac{n}{\mathcal{H}}\right\rceil + \mathcal{H}\lceil \log \mathcal{H}\rceil$$

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



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

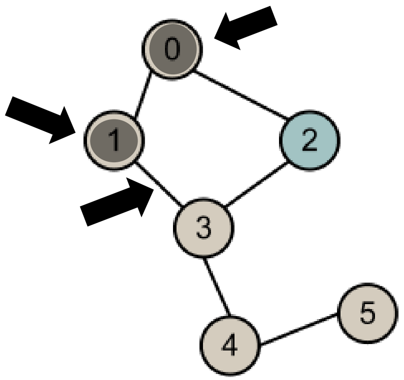
Key methods



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

Key methods

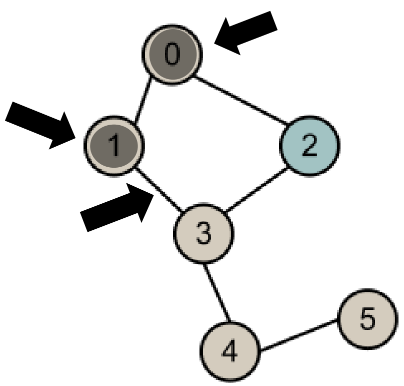
Use **the BEXTR bitwise operation** to help extract an arbitrary sequence of bits



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

Key methods

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

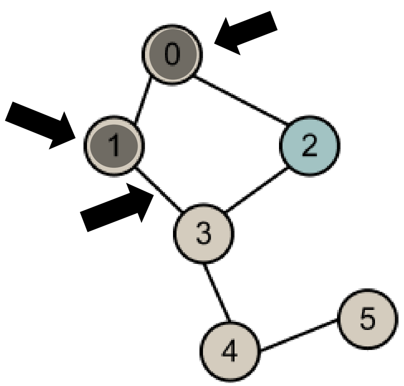



```
/* v_ID is an opaque type for IDs of vertices. */
v_ID N_{i,v}(v_ID v, int32_t i, int64_t* 𝒪, int64_t* 𝒜, int8_t s){
  int64_t exactBitOffset = s * (𝒪[v] + i);
  int8_t* address = (int8_t*) 𝒜 + (exactBitOffset >> 3);
  int64_t distance = exactBitOffset & 7;
  int64_t value = ((int64_t*) (address))[0];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

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

Key methods

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



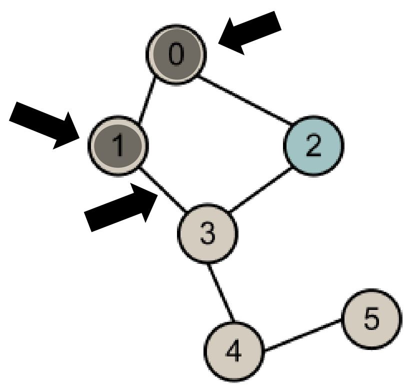
Return *i*-th neighbor of vertex *v*

```
/* 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); }
```

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*

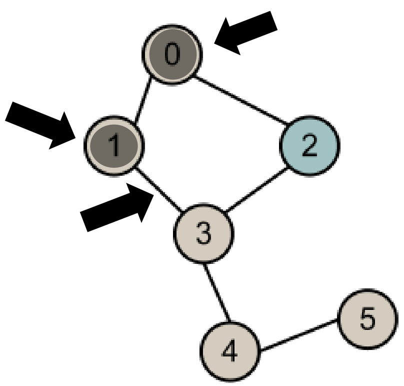
Pointer to the offset array

```
/* 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); }
```

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

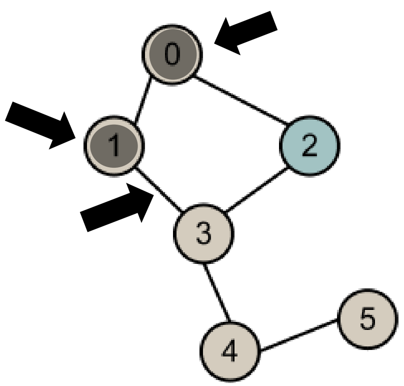
Pointer to the adjacency array

```
/* 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); }
```

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

Pointer to the adjacency array

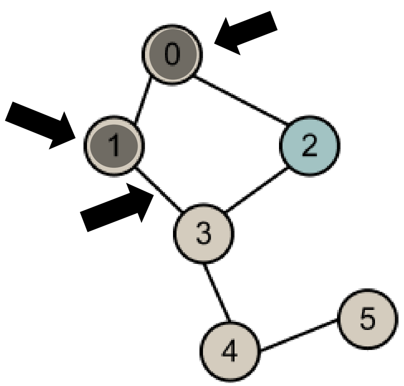
$s = \lceil \log n \rceil$

```
/* 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); }
```

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*

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

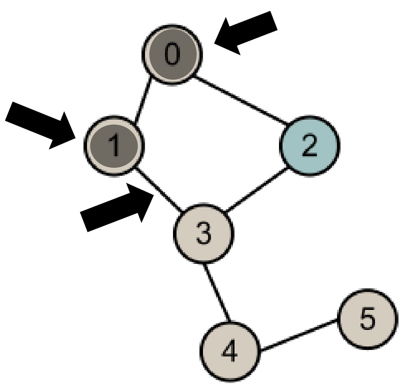
$s = \lceil \log n \rceil$

```
/* 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); }
```

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*

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

$s = \lceil \log n \rceil$

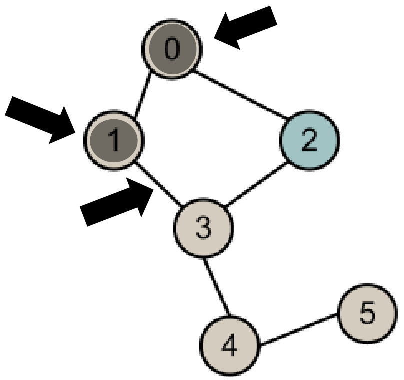
Get the closest byte alignment

```
/* 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); }
```

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*

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

$s = \lceil \log n \rceil$

```
/* 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); }
```

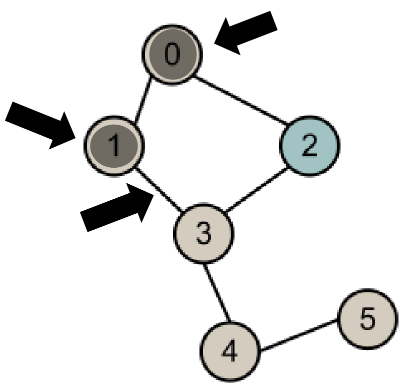
Get the closest byte alignment

Get the distance from the byte alignment

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*

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

$s = \lceil \log n \rceil$

```
/* 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); }
```

Get the closest byte alignment

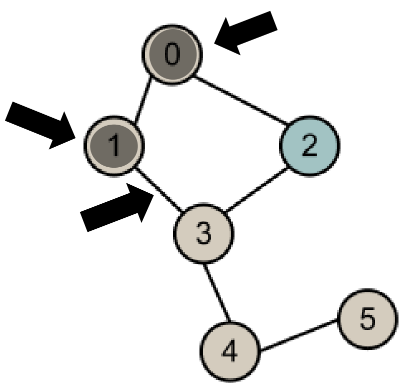
Get the distance from the byte alignment

Access the derived 64-bit value

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*

Derive exact offset (in bits) to the neighbor label

Pointer to the offset array

Pointer to the adjacency array

$s = \lceil \log n \rceil$

```
/* 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); }
```

Get the closest byte alignment

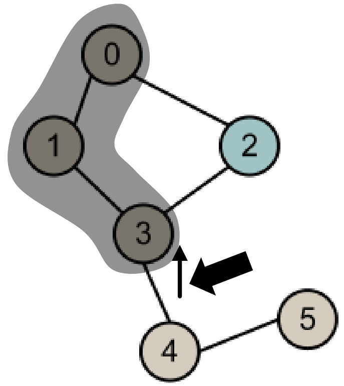
Get the distance from the byte alignment

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

Access the derived 64-bit value

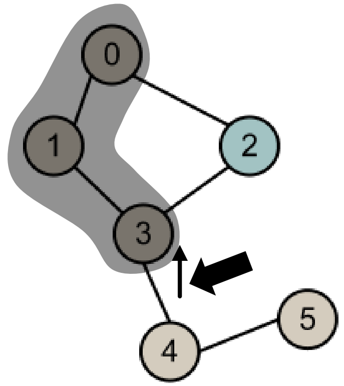
❷ Log (Offset structure)

❷ Log (Offset structure)



2 Log (Offset structure)

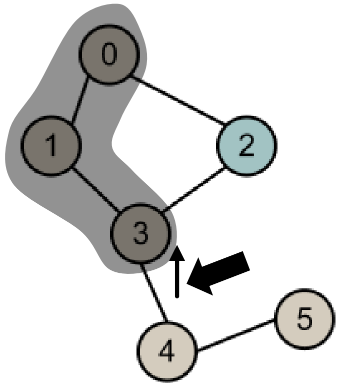
Use a **bit vector** instead of an array of offsets... !



2 Log (Offset structure)

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

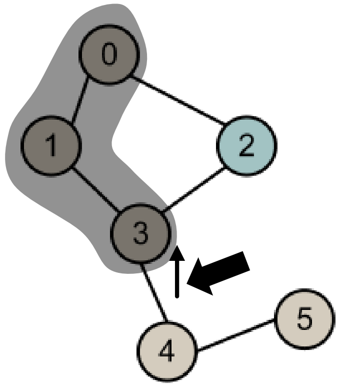
!



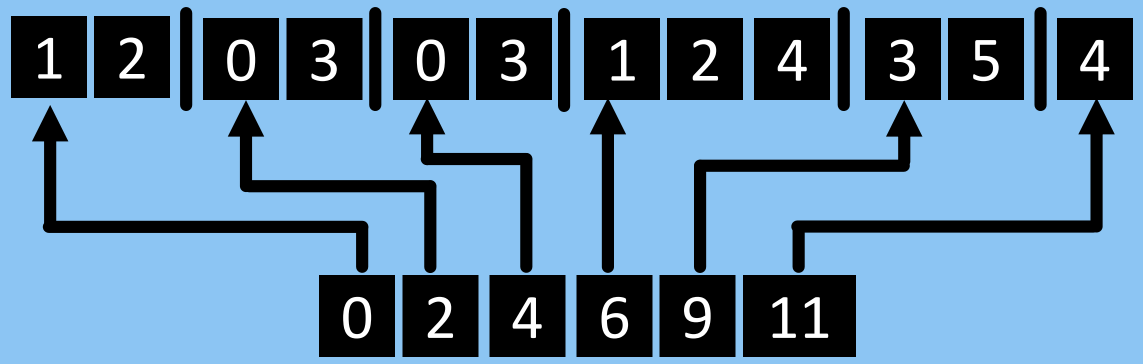
Bit vectors instead of offset arrays

2 Log (Offset structure)

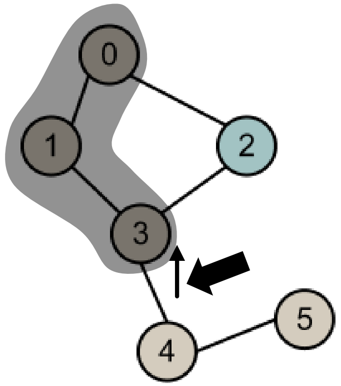
Use a **bit vector** instead of an array of offsets... !



Bit vectors instead of offset arrays

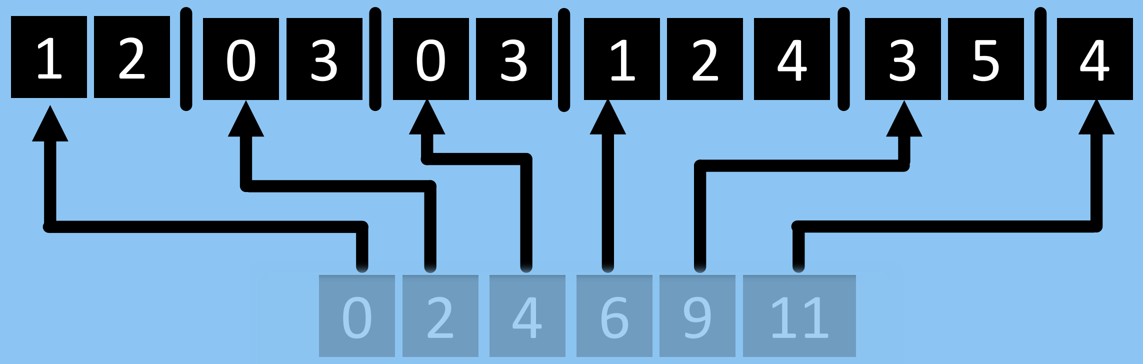


2 Log (Offset structure)



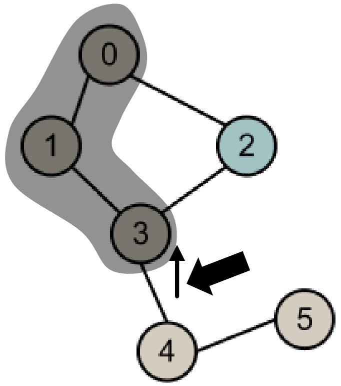
Use a **bit vector** instead of an array of offsets...

Bit vectors instead of offset arrays

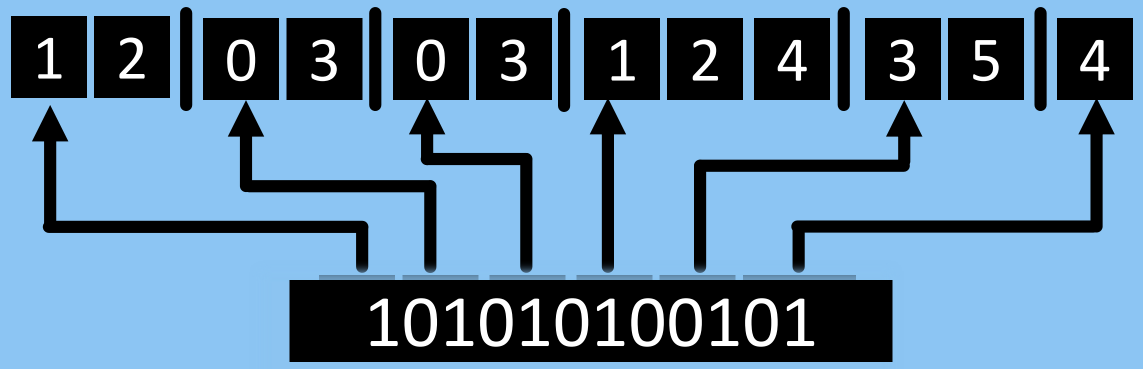


2 Log (Offset structure)

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

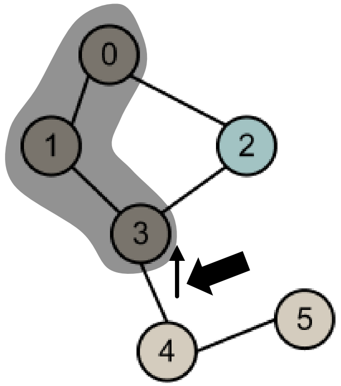


Bit vectors instead of offset arrays

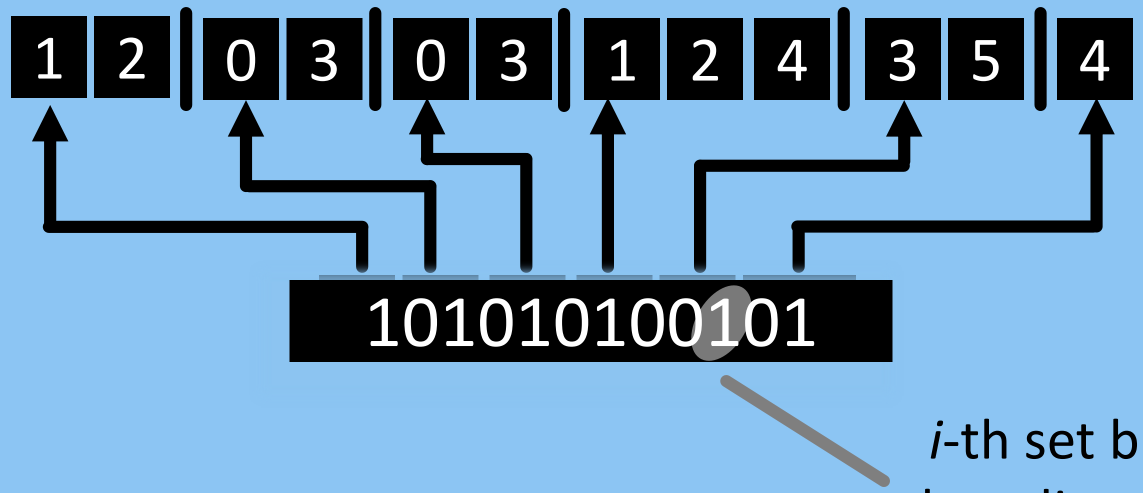


2 Log (Offset structure)

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

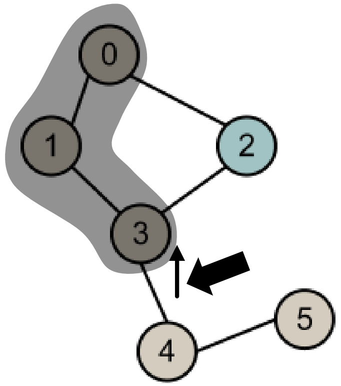


Bit vectors instead of offset arrays



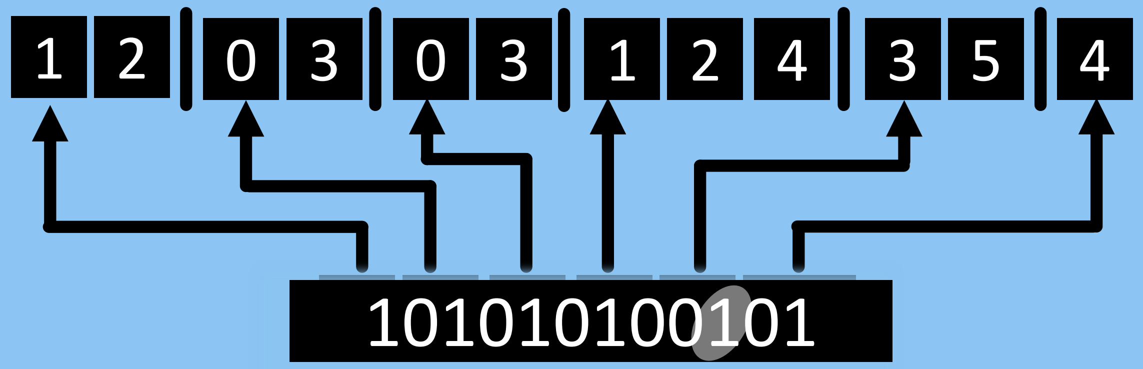
i-th set bit has a position *x* ➔ the adjacency array of a vertex *i* starts at a word *x*

2 Log (Offset structure)



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

Bit vectors instead of offset arrays

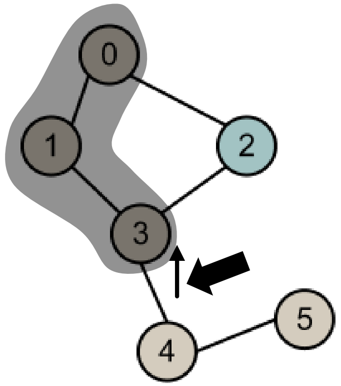


How many 1s are set before a given i-th bit?

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

2 **Log (** Offset structure **)**

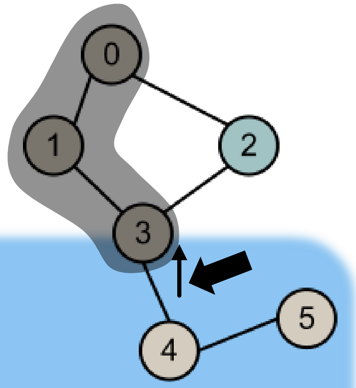
...Encode the resulting bit vectors as **succinct bit vectors [1]** !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 **Log (Offset structure)**

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



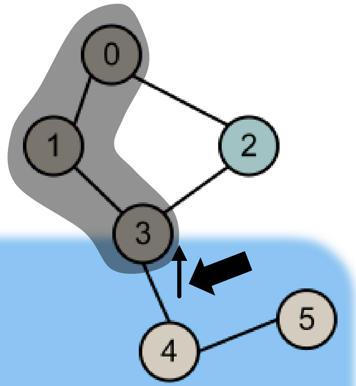
Succinct bit vectors

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 **Log (** Offset structure **)**

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

!



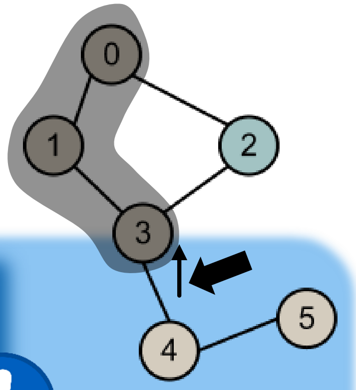
Succinct bit vectors

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

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

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



Succinct bit vectors

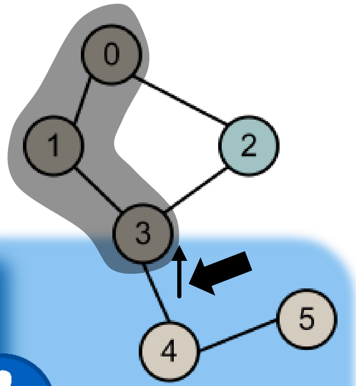
They use **[Q] + *o*(Q) bits** ([Q] - lower bound), they answer various queries in ***o*(Q) time**.

= small + fast (hopefully) !

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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**.

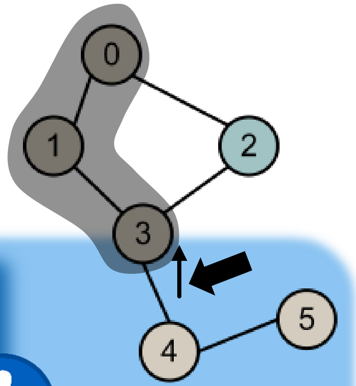
= small + fast (hopefully) !

101010100101000101010111110000001100001...

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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**.

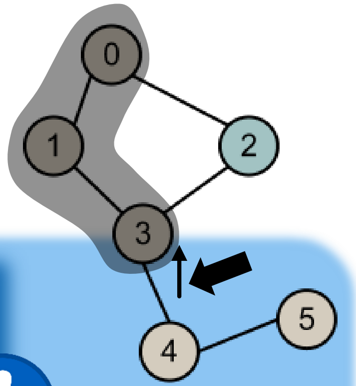
= small + fast (hopefully) !

n bits 1 0 1 0 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 1 . . .

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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**.

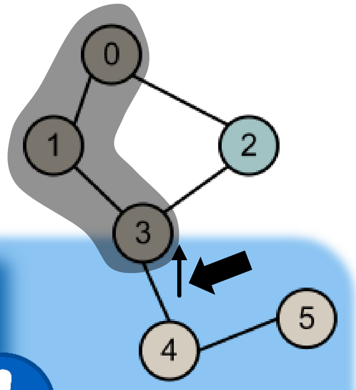
= small + fast (hopefully) !

n bits 1 0 1 0 1 0 1 0 0 1 0 | 1 0 0 0 1 0 1 0 1 0 1 | 1 1 1 1 0 0 0 0 0 0 1 | 1 0 0 0 0 1 . . .

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

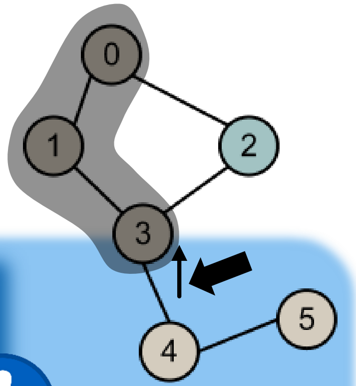
$\log^2 n$ | $\log^2 n$ | $\log^2 n$ | ...

n bits: 10101010010 | 10001010101 | 11110000001 | 100001...

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

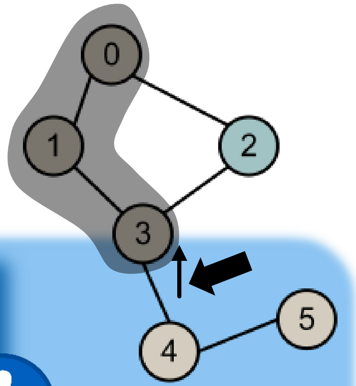
$\log^2 n = t_1$ | $\log^2 n$ | $\log^2 n$ | ...

n bits: 1 0 1 0 1 0 1 0 0 1 0 | 1 0 0 0 1 0 1 0 1 0 1 | 1 1 1 1 0 0 0 0 0 0 1 | 1 0 0 0 0 1 . . .

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

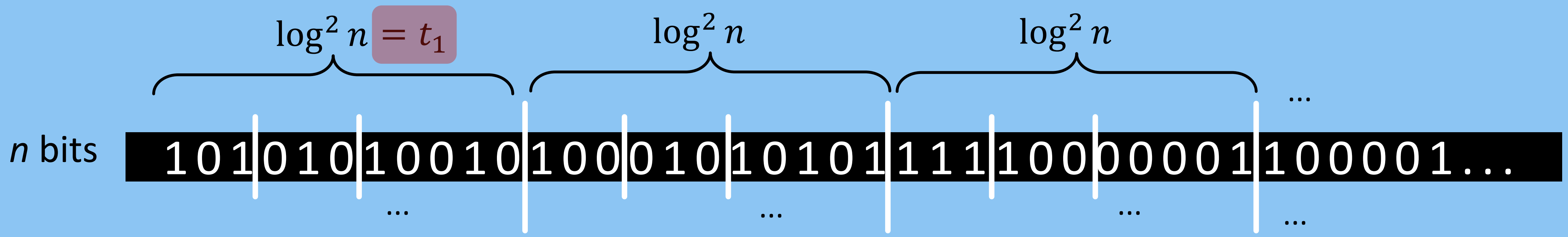
...Encode the resulting bit vectors as **succinct bit vectors [1]** !



Succinct bit vectors

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

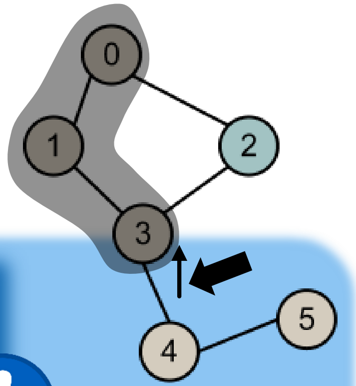
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

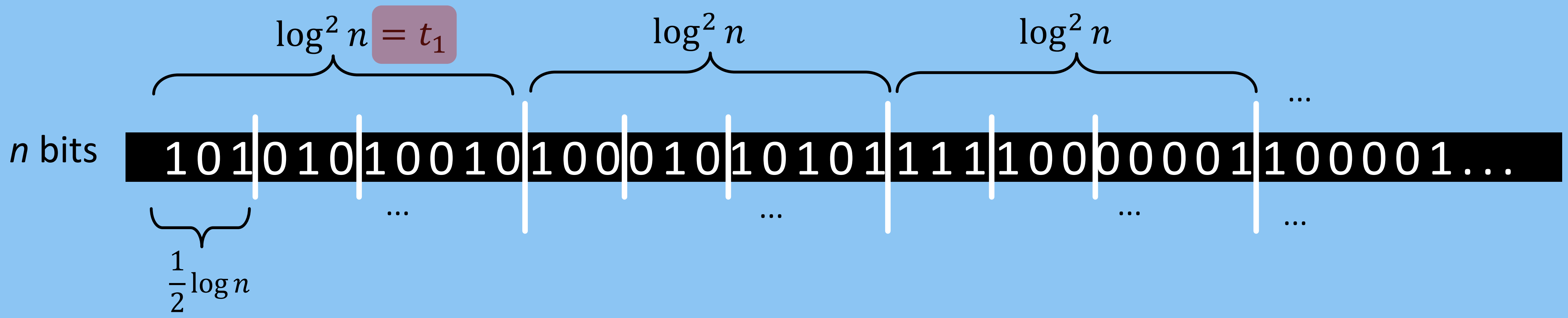
...Encode the resulting bit vectors as **succinct bit vectors [1]** !



Succinct bit vectors

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

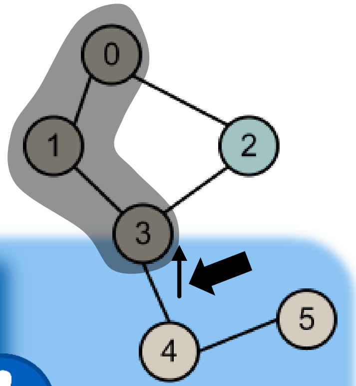
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

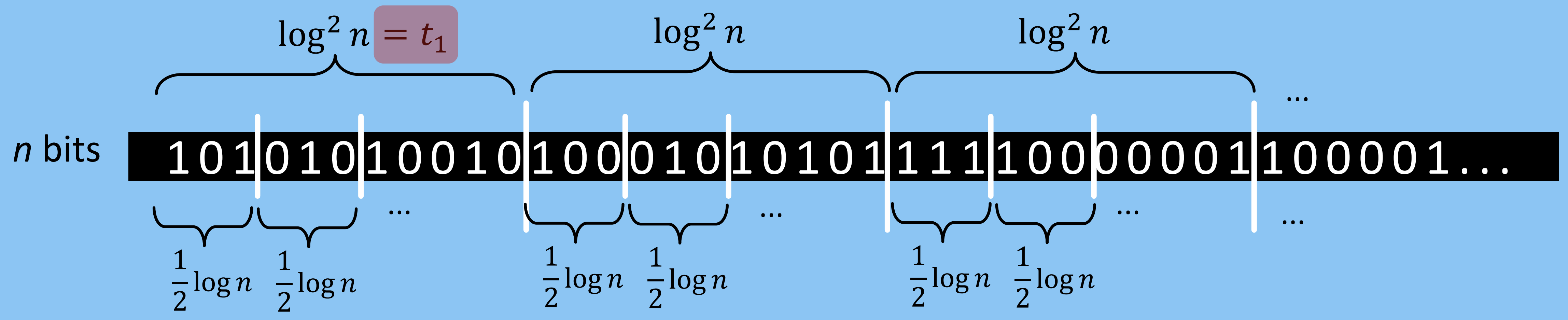
...Encode the resulting bit vectors as **succinct bit vectors [1]** !



Succinct bit vectors

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

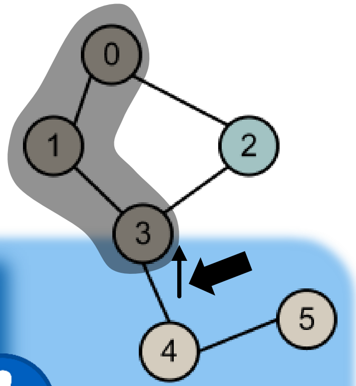
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

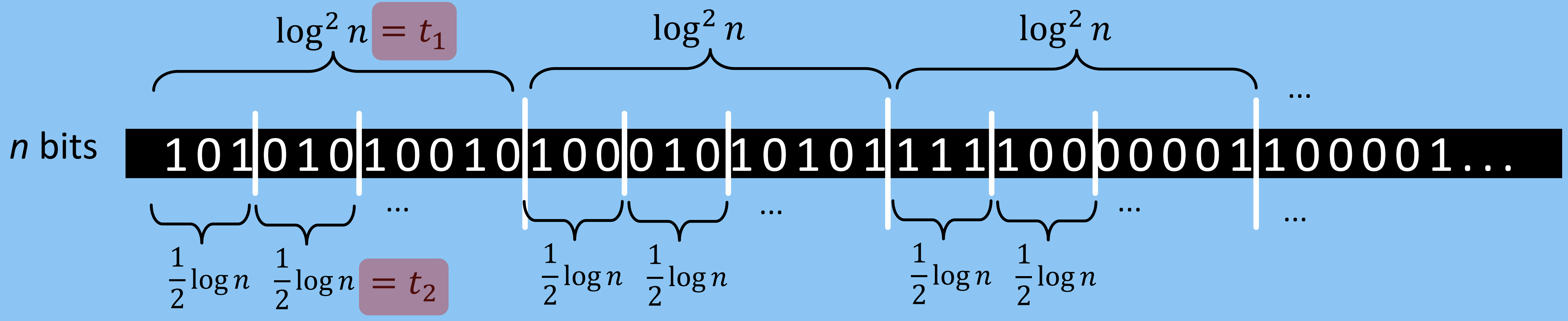
...Encode the resulting bit vectors as succinct bit vectors [1] !



Succinct bit vectors

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

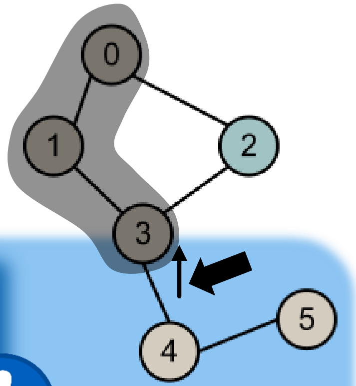
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

Compute & store the number of 1s

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

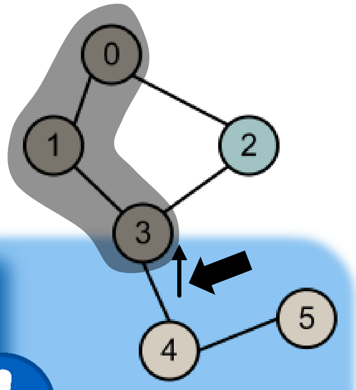
n bits: 1 0 1 | 0 1 0 | 1 0 0 1 0 | 1 0 0 | 0 1 0 | 1 0 1 0 1 | 1 1 1 | 1 0 0 | 0 0 0 0 1 | 1 0 0 0 0 1 . . .

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

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

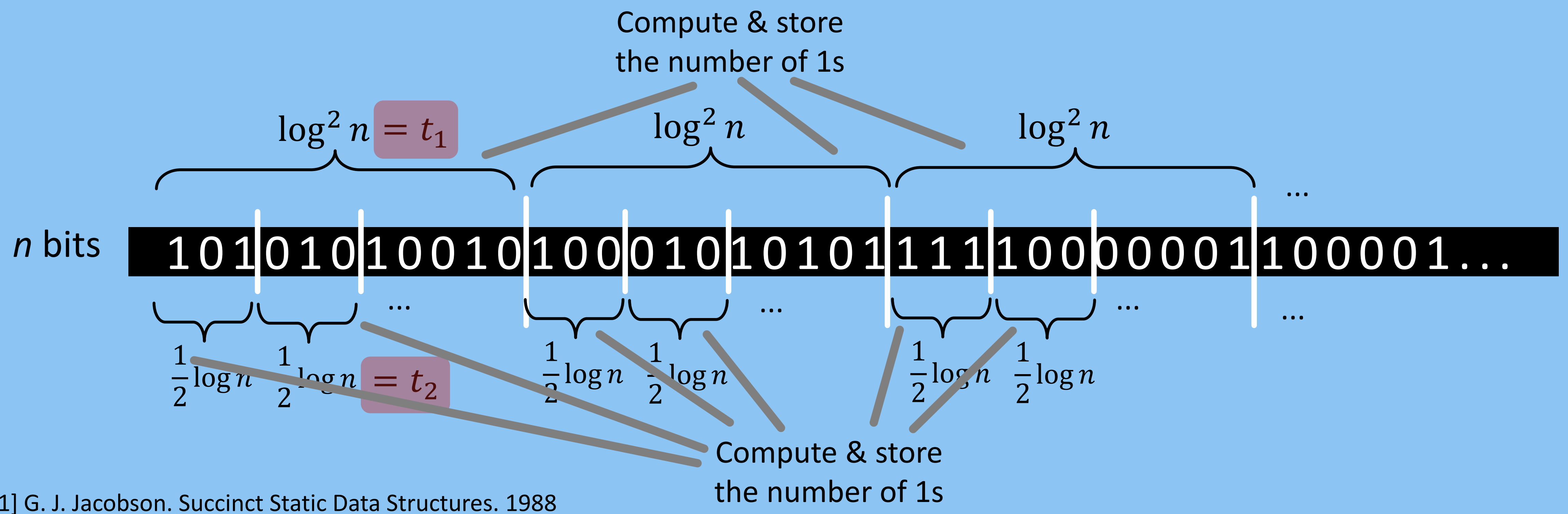
...Encode the resulting bit vectors as succinct bit vectors [1] !



Succinct bit vectors

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

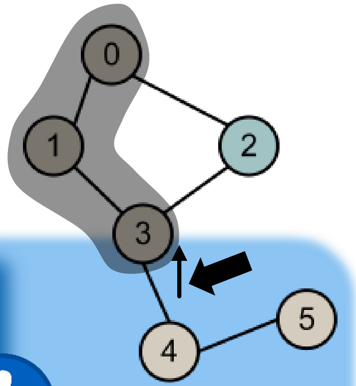
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 Log (Offset structure)

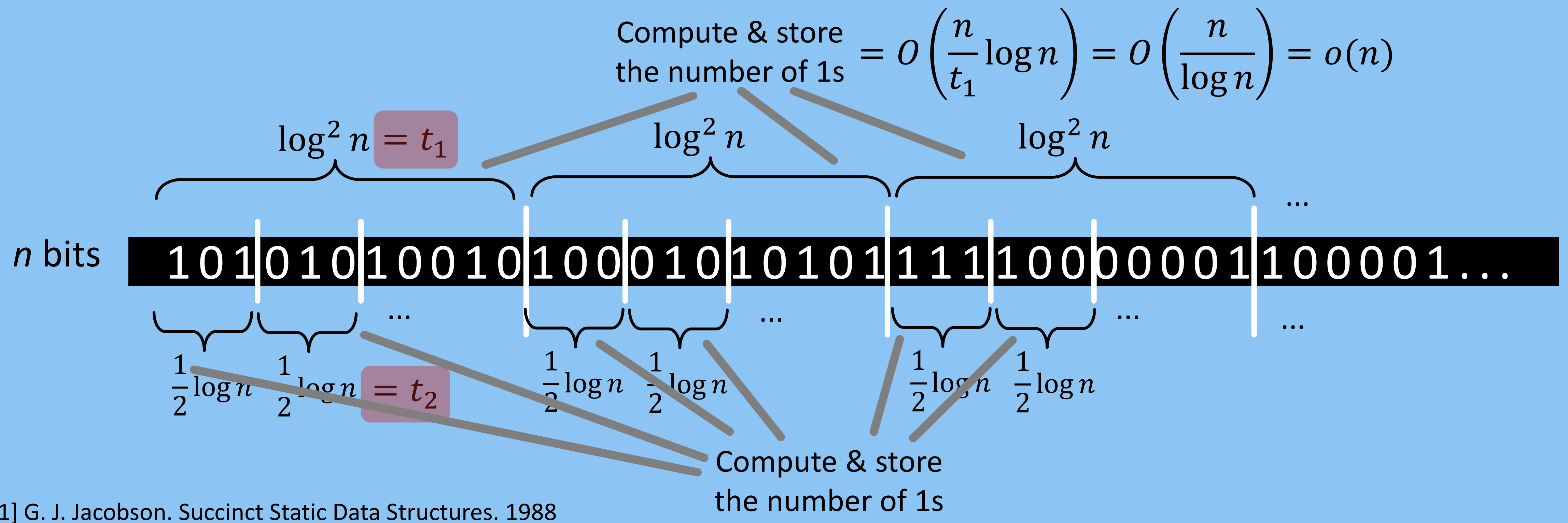
...Encode the resulting bit vectors as **succinct bit vectors [1]** !



Succinct bit vectors

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

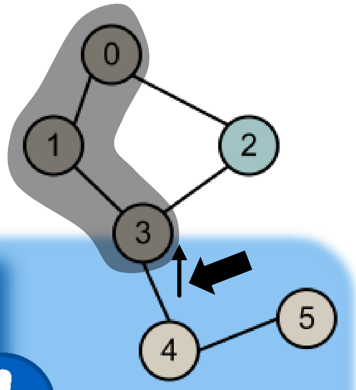
= small + fast (hopefully) !



[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

Compute & store the number of 1s $= O\left(\frac{n}{t_1}\log n\right) = O\left(\frac{n}{\log n}\right) = o(n)$

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

n bits: 1 0 1|0 1 0|1 0 0 1 0|1 0 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 . . .

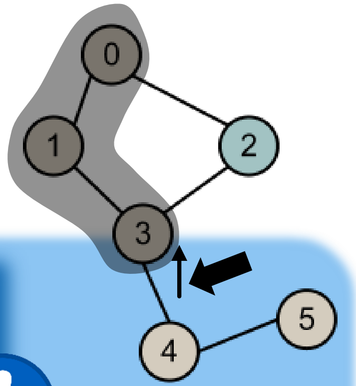
$\frac{1}{2}\log n \quad \frac{1}{2}\log n = t_2 \quad \ldots \quad \frac{1}{2}\log n \quad \frac{1}{2}\log n \quad \ldots \quad \frac{1}{2}\log n \quad \frac{1}{2}\log n \quad \ldots \quad \ldots$

Compute & store the number of 1s $= O\left(\frac{n}{t_2}\log t_1\right) = O\left(\frac{n\log\log n}{\log n}\right) = o(n)$

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

Compute & store the number of 1s $= O\left(\frac{n}{t_1}\log n\right) = O\left(\frac{n}{\log n}\right) = o(n)$

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

n bits: 1 0 1 | 0 1 0 | 1 0 0 1 0 | 1 0 0 | 0 1 0 | 1 0 1 0 1 | 1 1 1 | 1 0 0 | 0 0 0 0 1 | 1 0 0 0 0 1 . . .

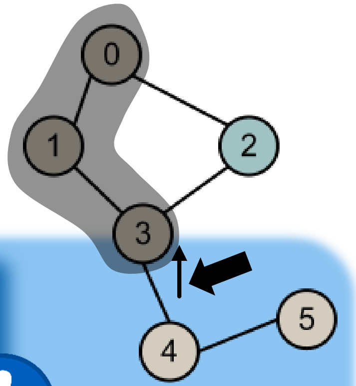
$\frac{1}{2}\log n$ $\frac{1}{2}\log n = t_2$ ... $\frac{1}{2}\log n$ $\frac{1}{2}\log n$ ... $\frac{1}{2}\log n$ $\frac{1}{2}\log n$ ... ...

Compute & store the number of 1s $= O\left(\frac{n}{t_2}\log t_1\right) = O\left(\frac{n\log\log n}{\log n}\right) = o(n)$

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

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) !

Total storage:

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

Compute & store the number of 1s $= O\left(\frac{n}{t_1}\log n\right) = O\left(\frac{n}{\log n}\right) = o(n)$

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

n bits: 1 0 1 | 0 1 0 | 1 0 0 1 0 | 1 0 0 | 0 1 0 | 1 0 1 0 1 | 1 1 1 | 1 0 0 | 0 0 0 0 1 | 1 0 0 0 0 1 . . .

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

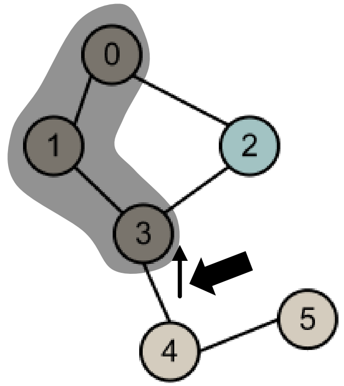
Compute & store the number of 1s $= O\left(\frac{n}{t_2}\log t_1\right) = O\left(\frac{n\log\log n}{\log n}\right) = o(n)$

[1] G. J. Jacobson. Succinct Static Data Structures. 1988

2 **Log (Offset structure)**

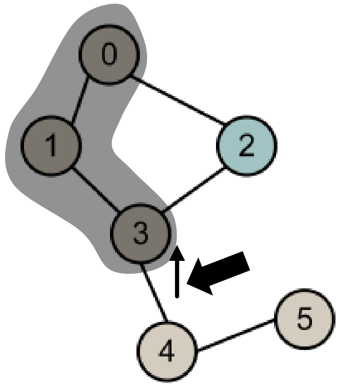
...Encode the resulting bit vectors as **succinct bit vectors**

!



2 Log (Offset structure)

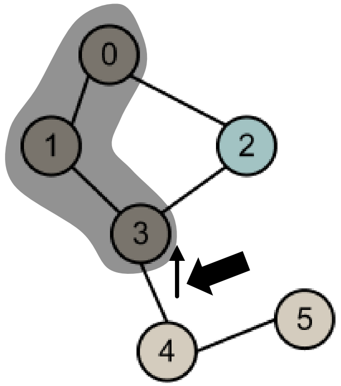
...Encode the resulting bit vectors as **succinct bit vectors** !



Formal analyses

2 Log (Offset structure)

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

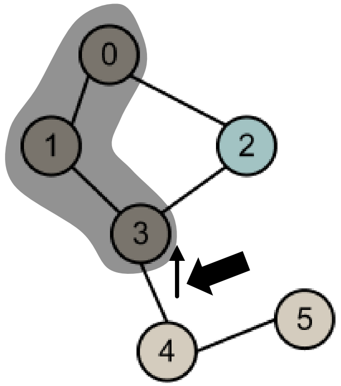


Formal analyses

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

2 Log (Offset structure)

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



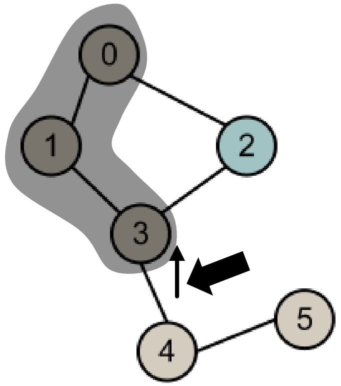
Formal analyses

Check the paper for details ☺

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

2 Log (Offset structure)

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



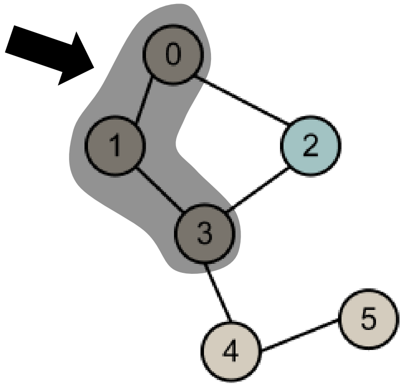
Formal analyses

Check the paper for details ☺

$\mathcal{O}$	ID	**Asymptotic size** [bits]	**Exact size** [bits]	*select* or $\mathcal{O}[v]$
Pointer array	ptrW	$O(Wn)$	$W(n+1)$	$O(1)$
Plain [44]	[illegible]	[illegible]	[illegible]	[illegible]
Interleaved [44]	[illegible]	[illegible]	[illegible]	[illegible]
Entropy based [3[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Sparse [76]	[illegible]	[illegible]	[illegible]	[illegible]
B-tree based [1]	bvBT	$O\left(\frac{Wm}{B}\right)$	$\approx 1.1 \cdot \frac{2Wm}{B}$	$O(\log n)$
Gap-compressed [1]	bvGC	$O\left(\frac{Wm}{B} \log \frac{Wm}{Bn}\right)$	$\approx 1.3 \cdot \frac{2Wm}{B} \log \frac{2Wm}{Bn}$	$O(\log n)$

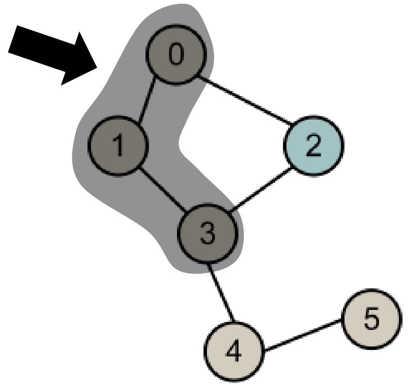
! **We will show that some are in practice both small and fast!**

3 **Log (** Adjacency structure **)**



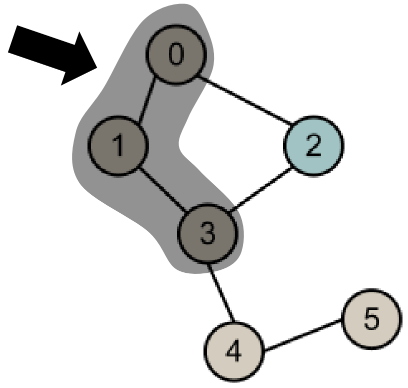
3 **Log (** Adjacency structure **)**

Use different relabelings !



3 **Log (Adjacency structure)**

Use different relabelings !

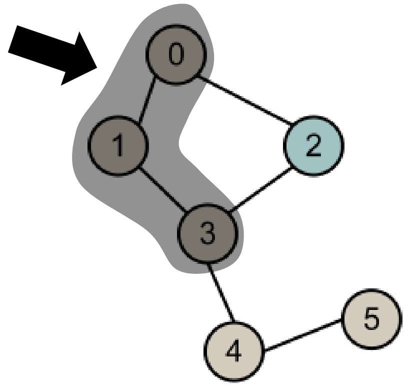


Degree-Minimizing: Targeting general graphs (no assumptions on graph structure)

3 **Log (** Adjacency structure **)**

Use different relabelings

!



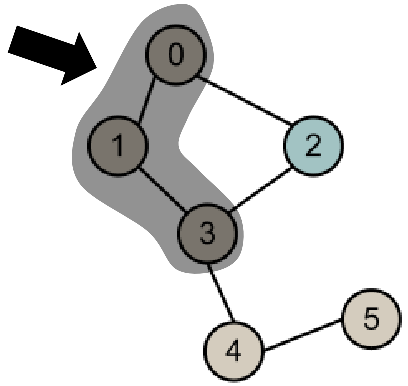
Degree-Minimizing: Targeting general graphs (no assumptions on graph structure)

More schemes that assume specific classes of graphs

...

3 **Log (** Adjacency structure **)**

Use different relabelings **!**



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

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

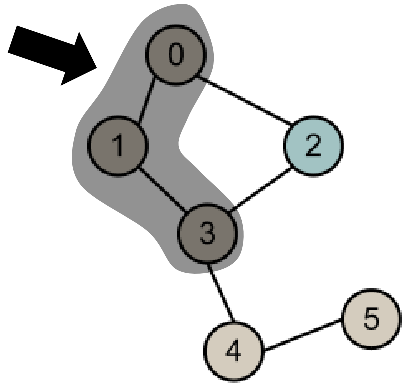
(simultaneously for all other neighborhoods)

More schemes that assume specific classes of graphs

...

3 **Log (** Adjacency structure **)**

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)

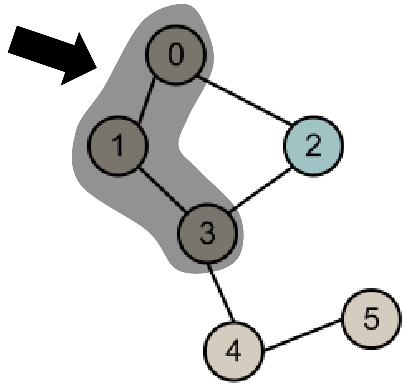
(1) The more often a label occurs (i.e., the higher vertex degree), the smaller permuted value it receives

More schemes that assume specific classes of graphs

...

3 **Log (** Adjacency structure **)**

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

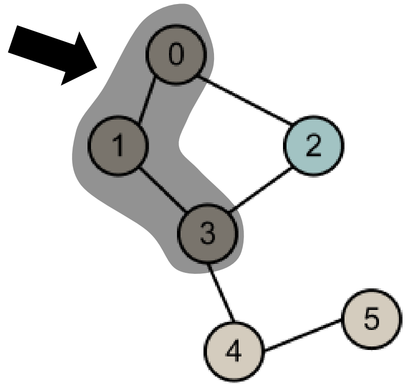
Gap-encode(v | w | x | y | z) = v | w-v | x-w | y-x | z-y

More schemes that assume specific classes of graphs

...

3 **Log (** Adjacency structure **)**

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

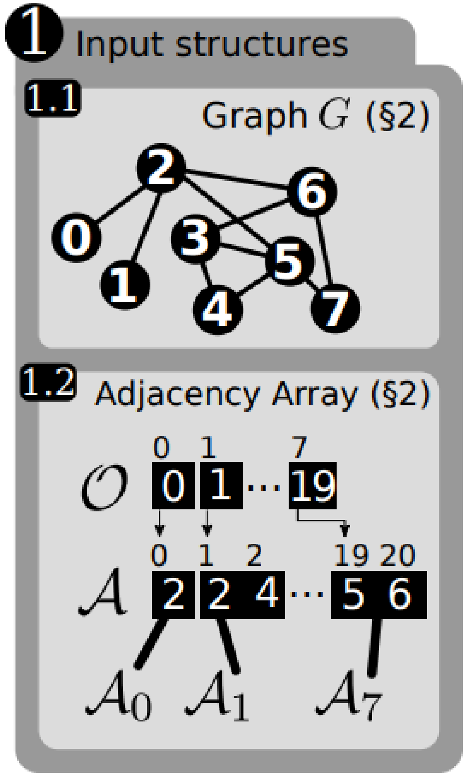
(2) Encode new labels with gap encoding (differences between consecutive labels instead of full labels)

More schemes that assume specific classes of graphs

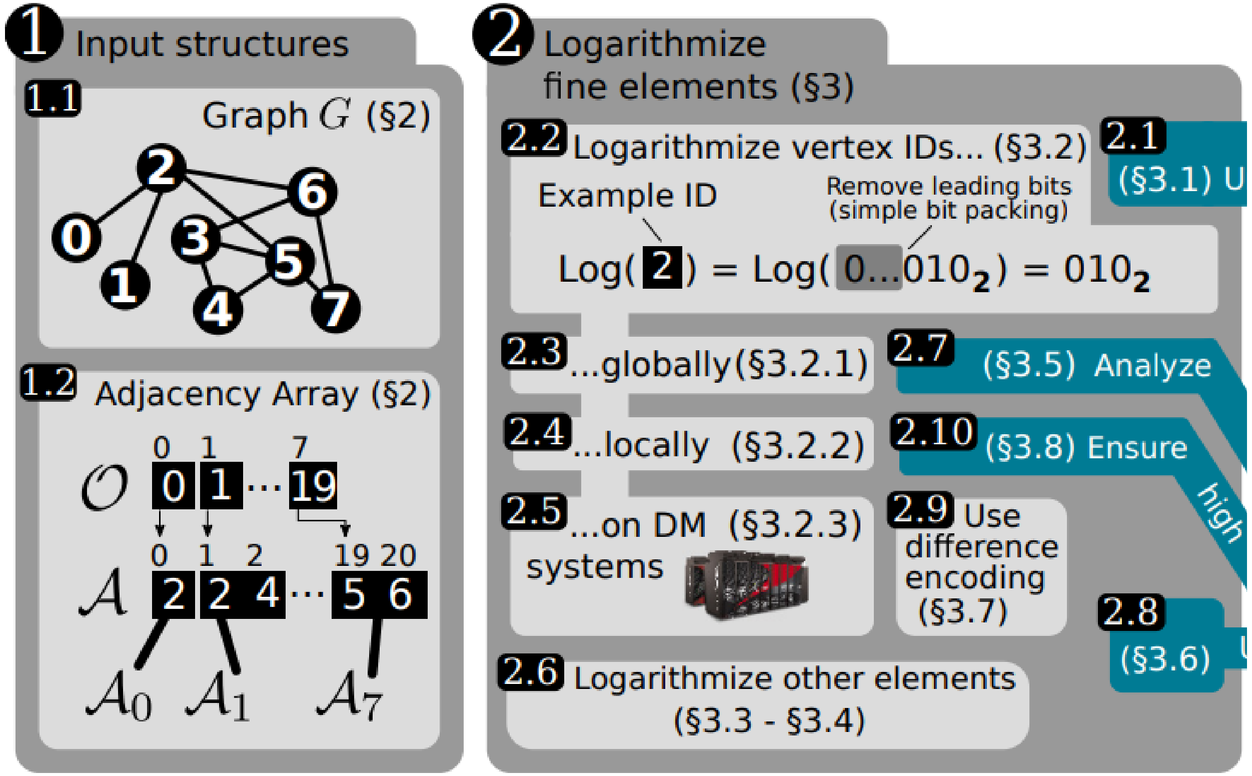
...

Overview of Full Log(Graph) Design

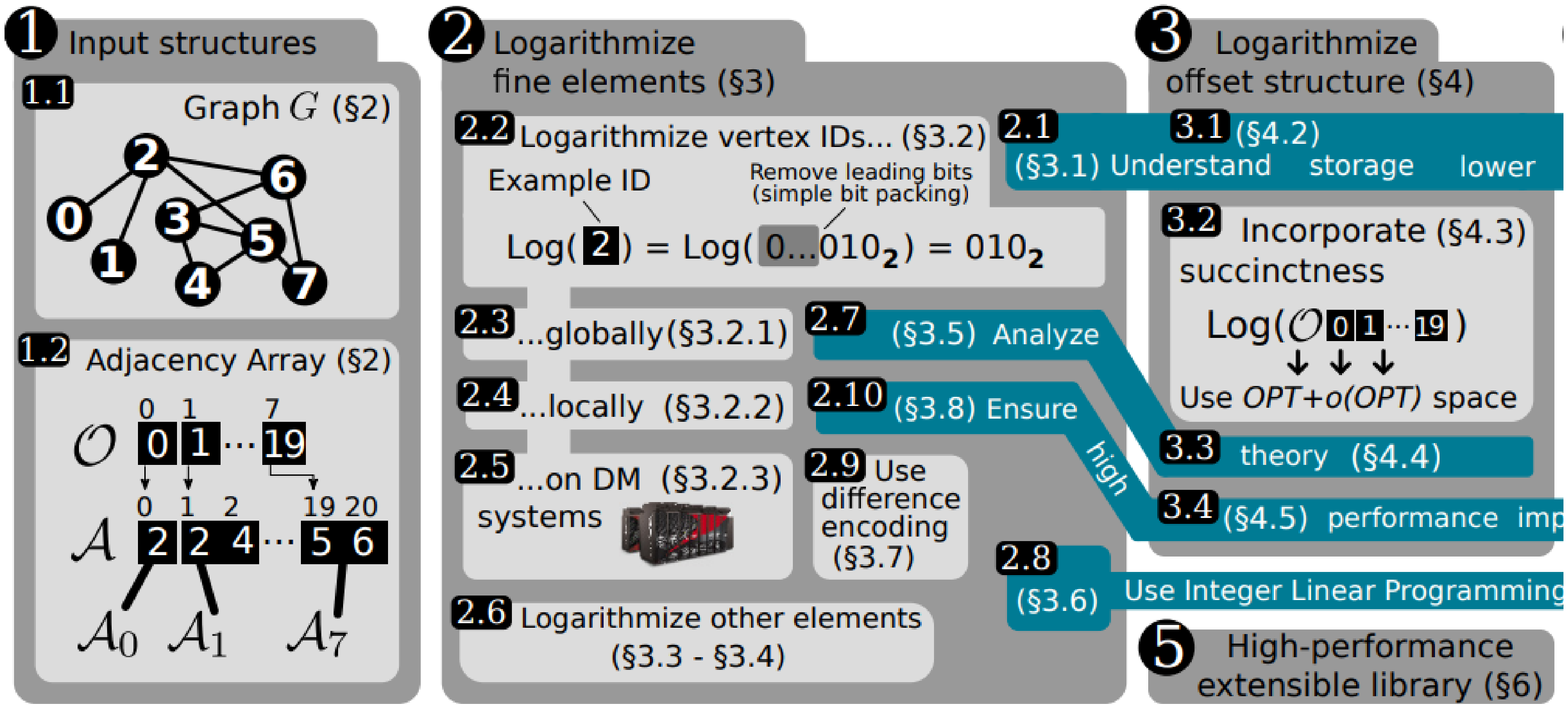
Overview of Full Log(Graph) Design



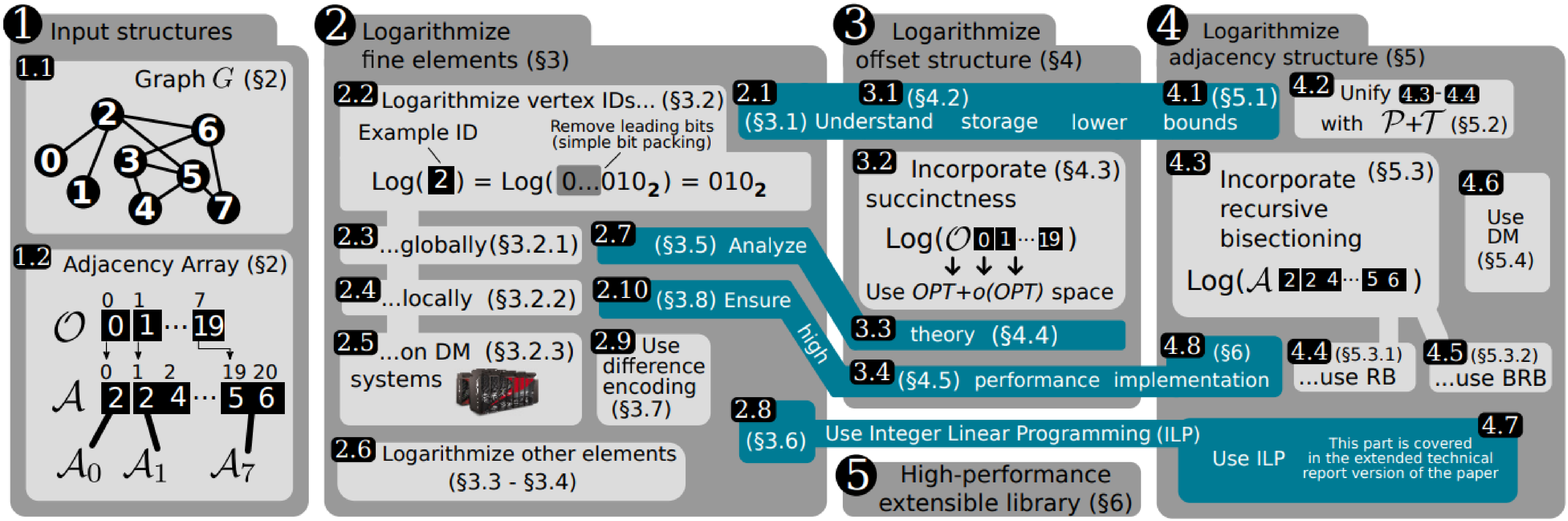
Overview of Full Log(Graph) Design



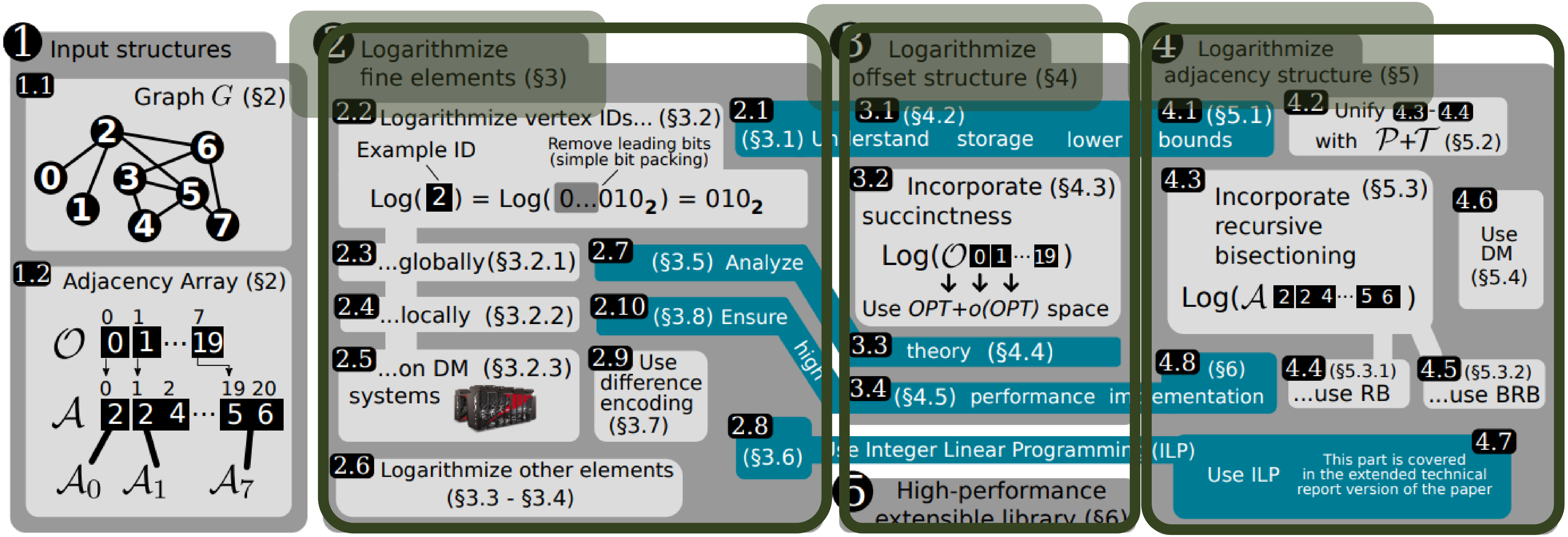
Overview of Full Log(Graph) Design



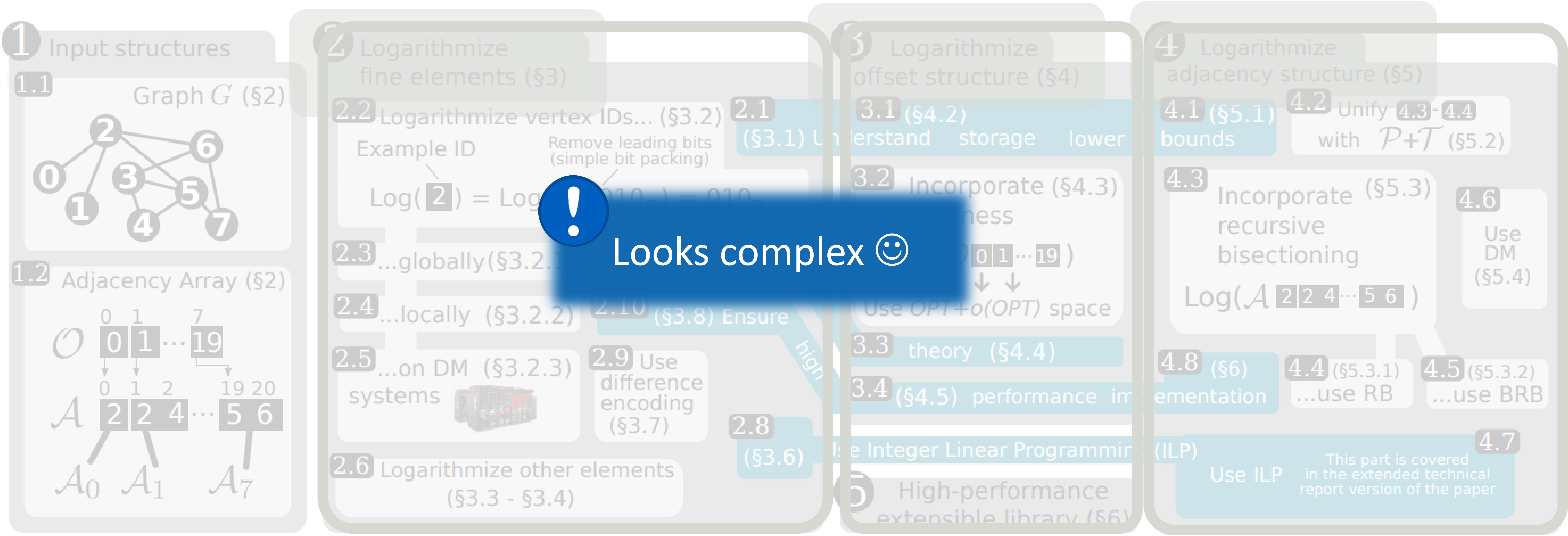
Overview of Full Log(Graph) Design



Overview of Full Log(Graph) Design

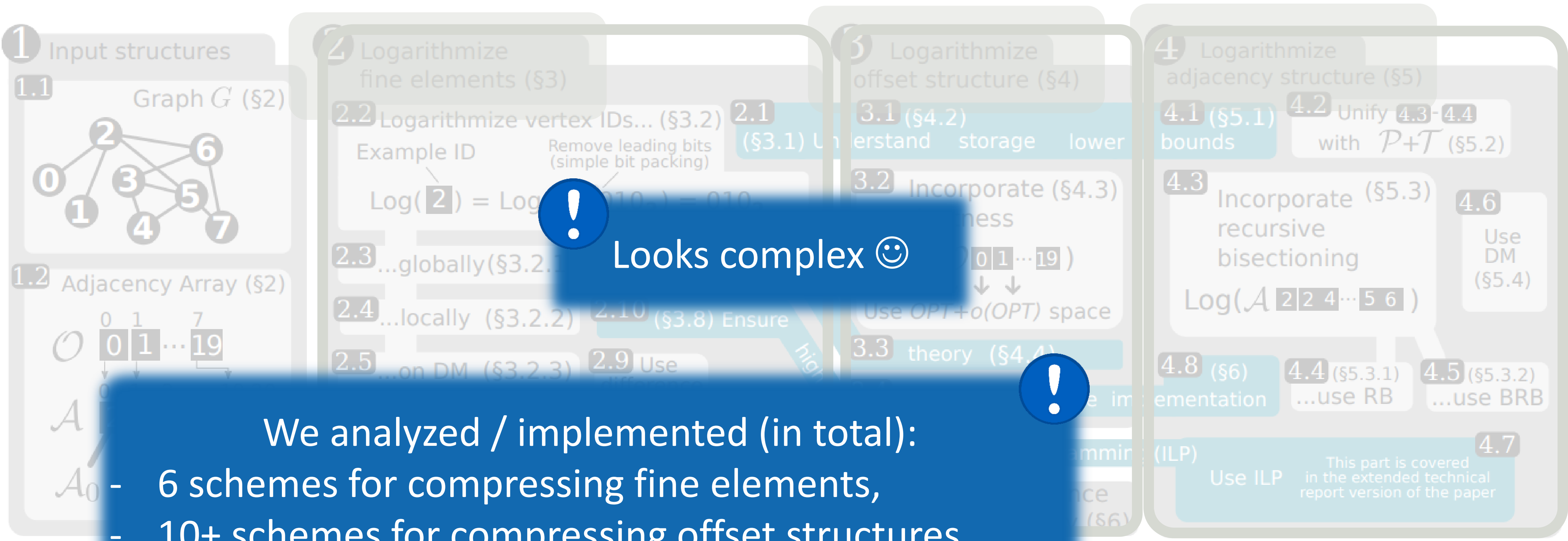


Overview of Full Log(Graph) Design



Looks complex ☺

Overview of Full Log(Graph) Design

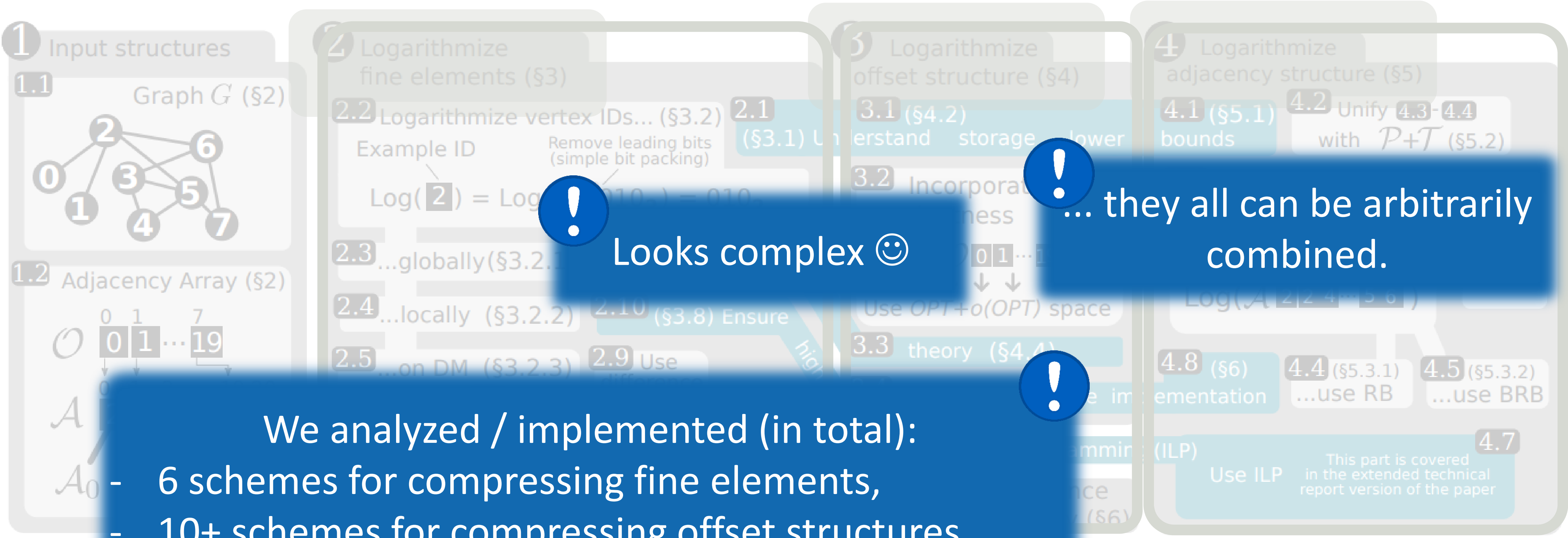


Looks complex ☺

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):

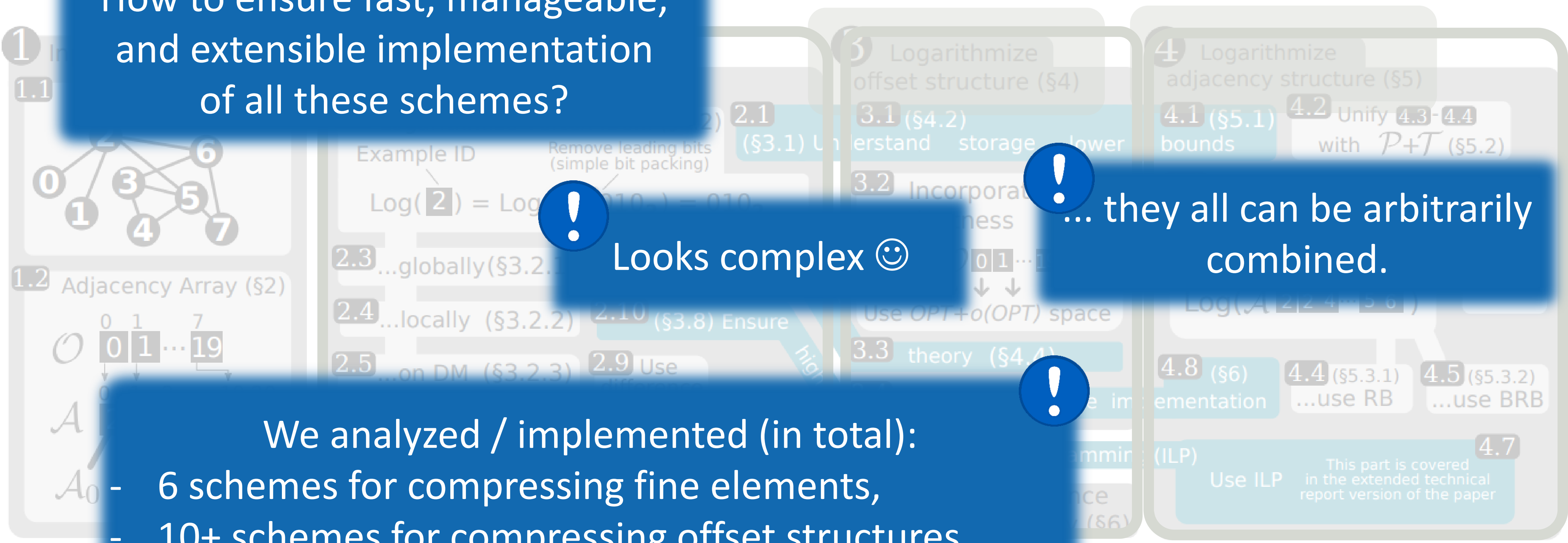
- 6 schemes for compressing fine elements,
- 10+ schemes for compressing offset structures,
- 4+ schemes for compressing adjacency structures

Overview of Full Log(Graph) Design

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

Looks complex ☺

... 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

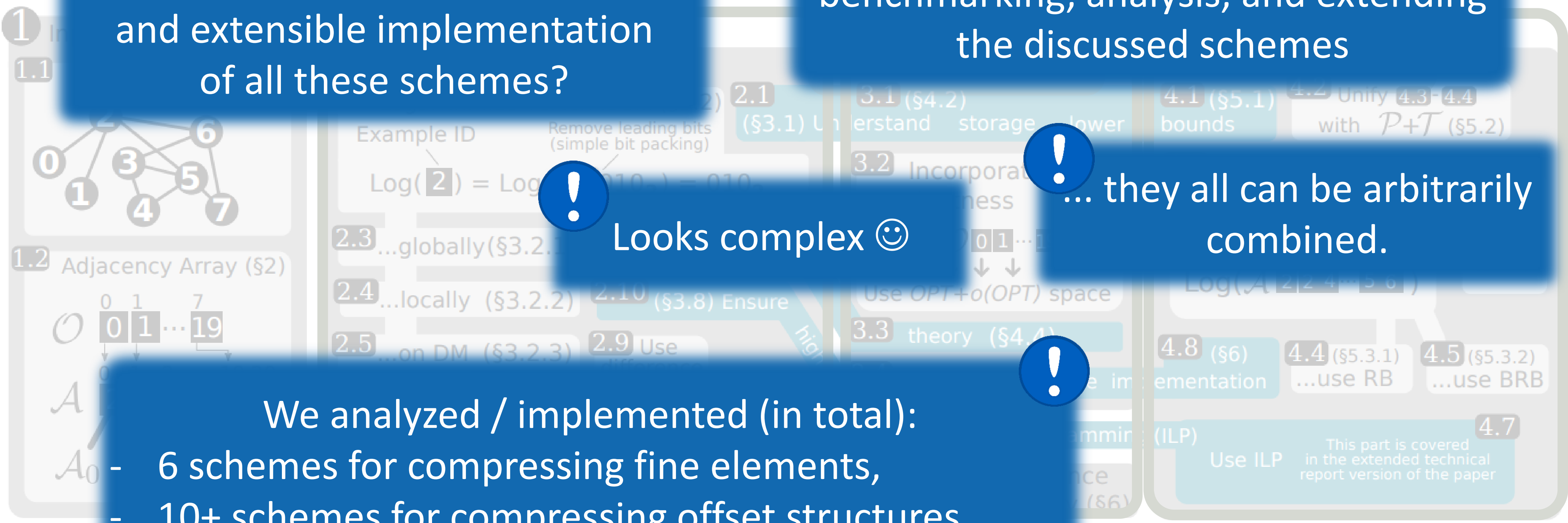
Overview of Full Log(Graph) Design

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

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

Looks complex ☺

... they all can be arbitrarily combined.



We analyzed / implemented (in total):

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

Performance Analysis

Types of machines

CSCS Cray Piz Daint

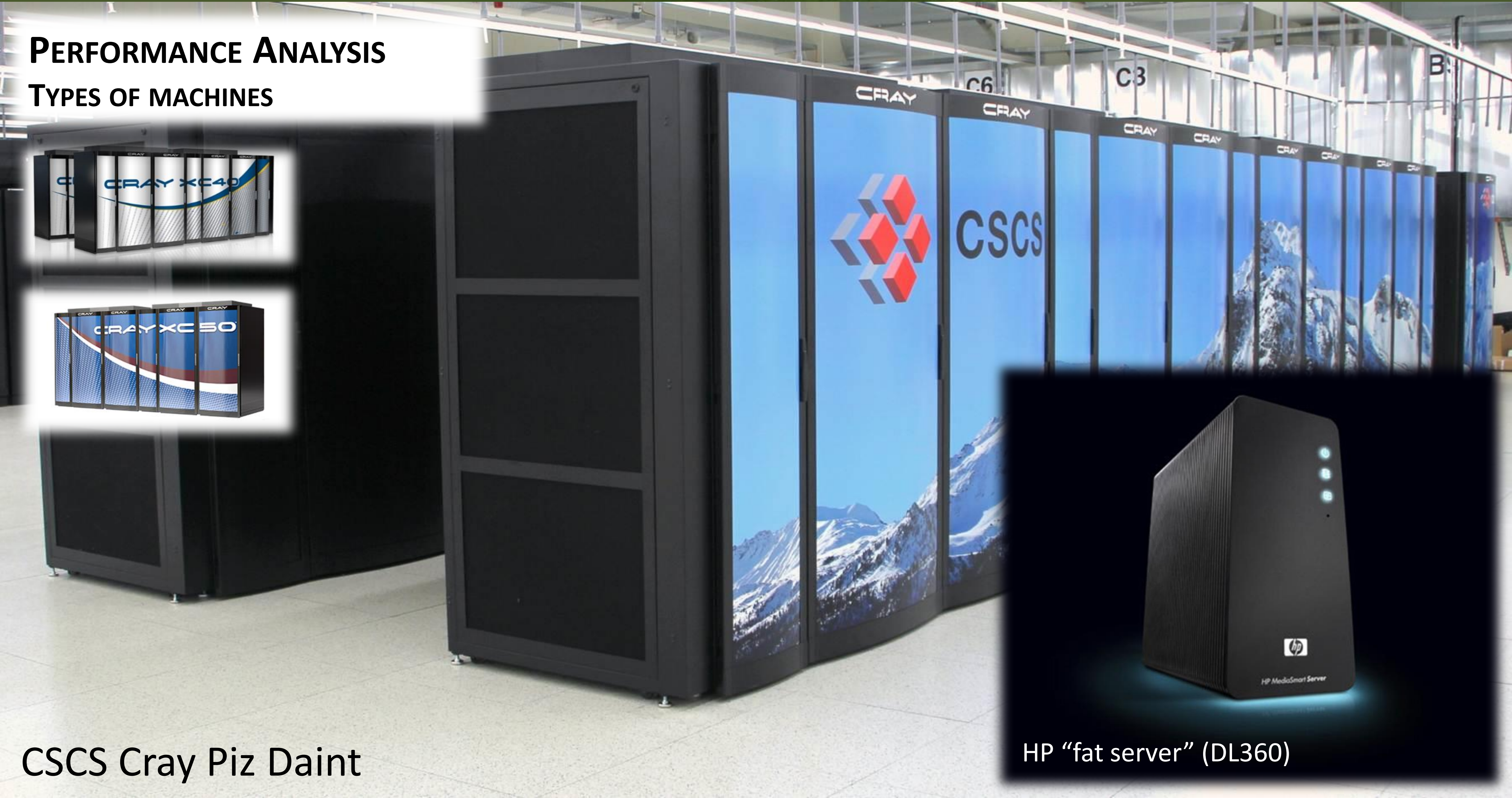
Performance Analysis

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

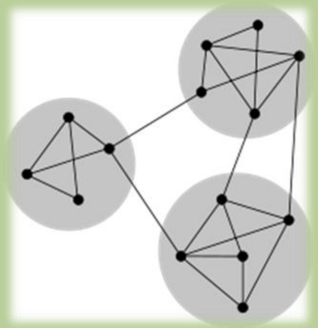
Types of graphs

Synthetic graphs

Performance Analysis

Types of graphs

Synthetic graphs



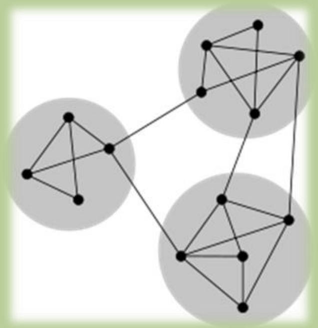
Kronecker [1]

[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.

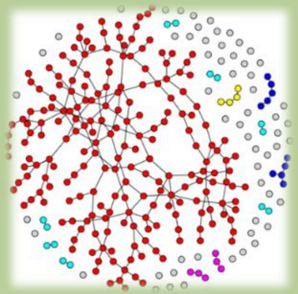
Performance Analysis

Types of graphs

Synthetic graphs



Kronecker [1]



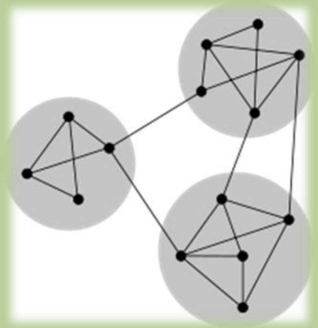
Erdös-Rényi [2]

[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.

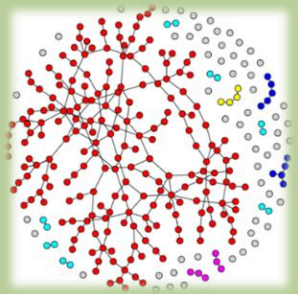
Performance Analysis

Types of graphs

Synthetic graphs



Kronecker [1]



Erdös-Rényi [2]

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

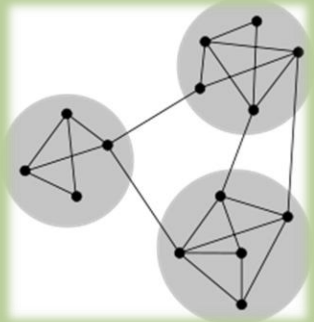
[3] SNAP. https://snap.stanford.edu

[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.

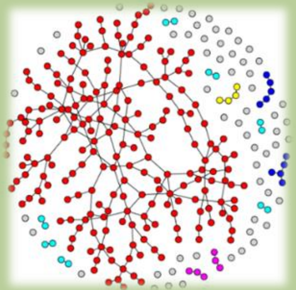
Performance Analysis

Types of graphs

Synthetic graphs

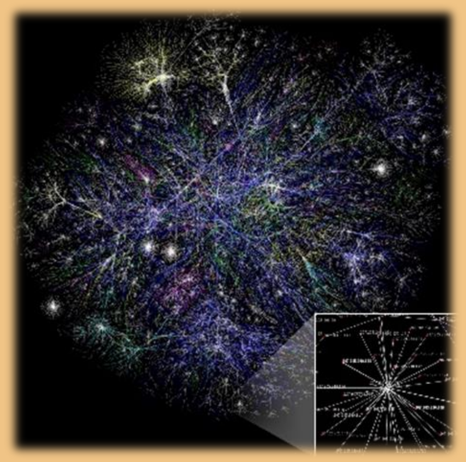


Kronecker [1]



Erdös-Rényi [2]

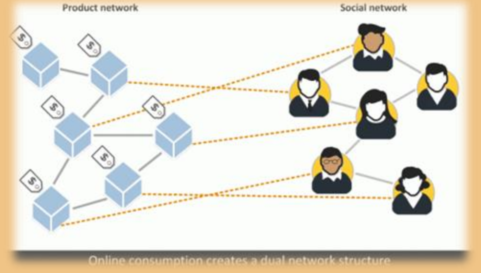
Real-world graphs (SNAP [3], KONECT [4], Webgraph [5], DIMACS [6])



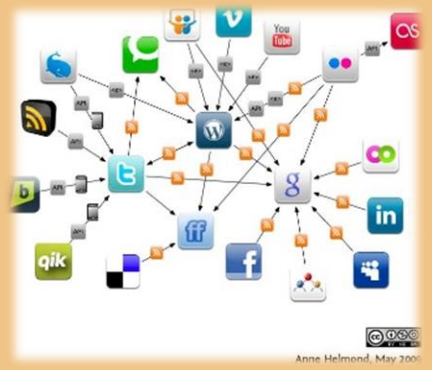
Web graphs

Road networks

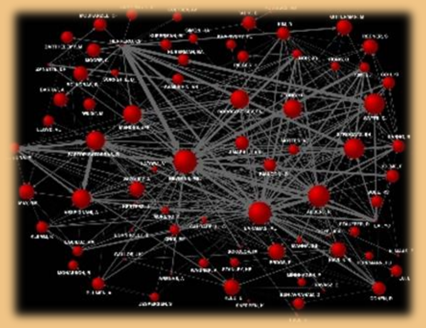
Social networks



Purchase networks



Communication graphs



Citation graphs

[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.

[3] SNAP. https://snap.stanford.edu
[4] KONECT. https://konect.cc
[5] DIMACS Challenge
[6] Webgraphs. https://law.di.unimi.it/datasets.php

Performance Analysis

Algorithms

Performance Analysis

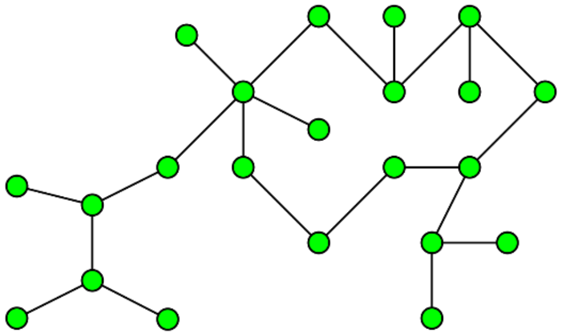
Algorithms

Connected Components
(Shiloach-Vishkin [1])

Performance Analysis

Algorithms

Connected Components
(Shiloach-Vishkin [1])

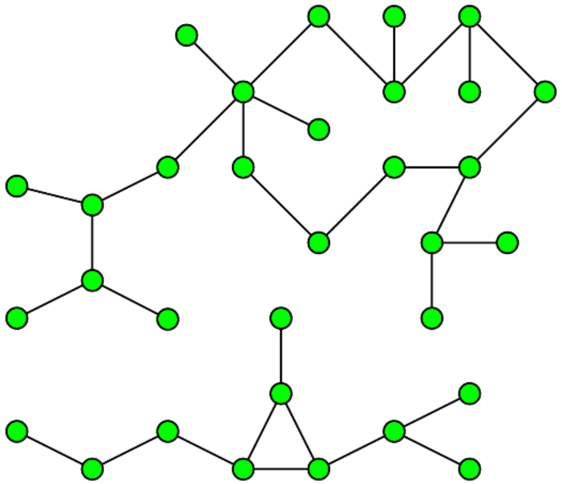


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

PERFORMANCE ANALYSIS

ALGORITHMS

Connected Components
(Shiloach-Vishkin [1])



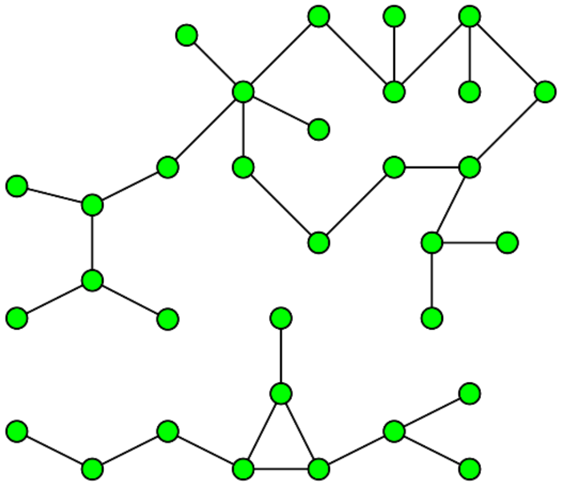
[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

Performance Analysis

Algorithms

Connected Components
(Shiloach-Vishkin [1])

BFS (direction optimization [2])

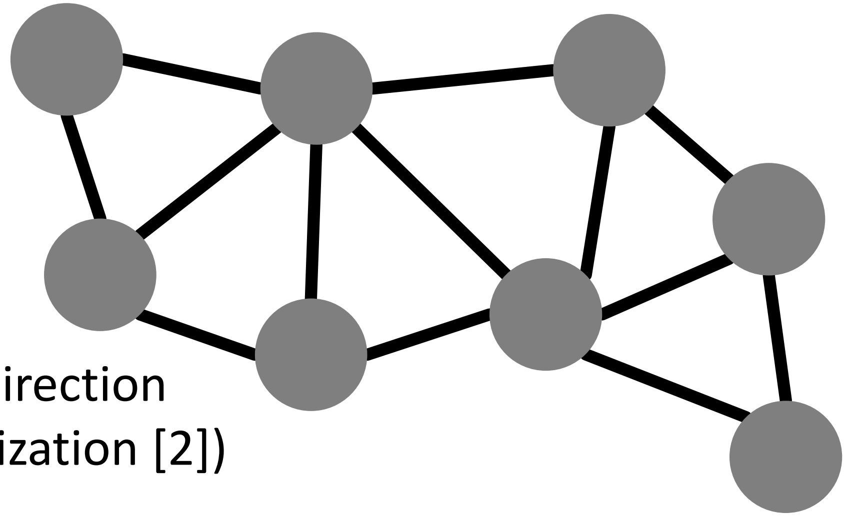


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

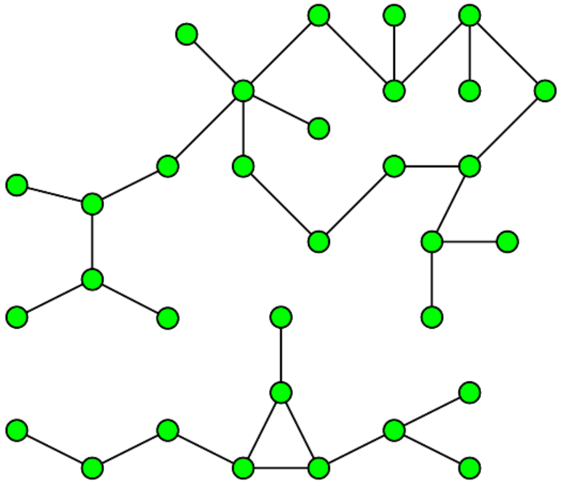
Performance Analysis

Algorithms



BFS (direction optimization [2])

Connected Components
(Shiloach-Vishkin [1])

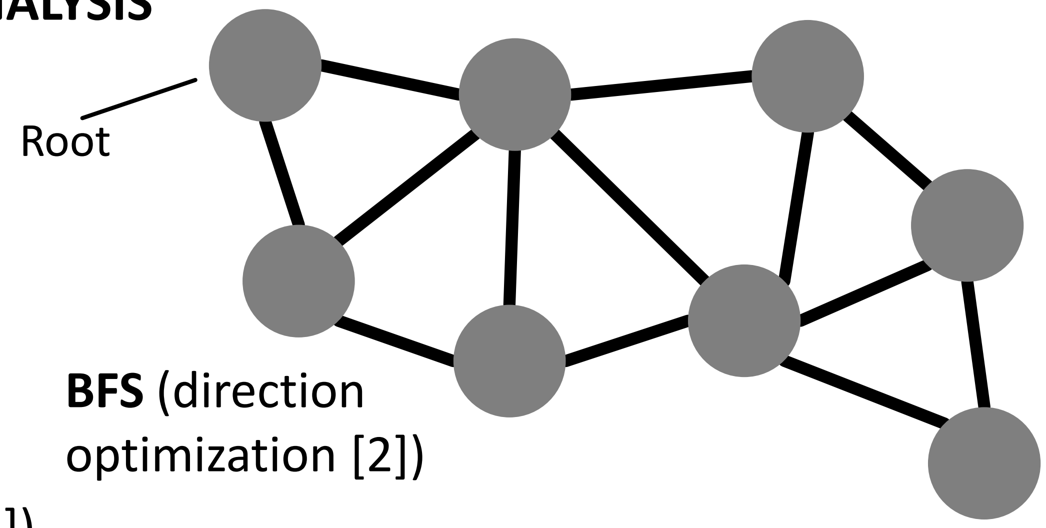


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

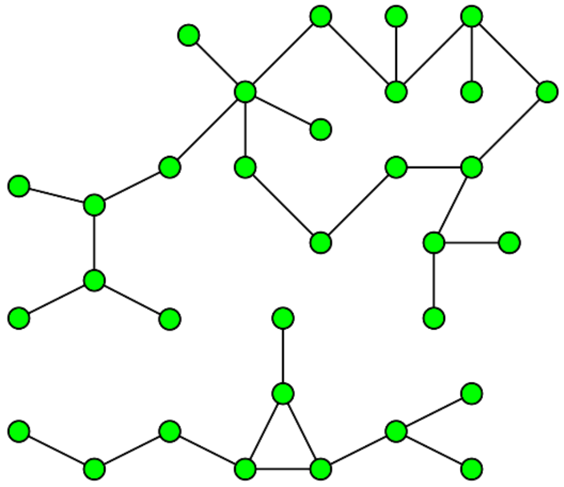
Performance Analysis

Algorithms



BFS (direction optimization [2])

Connected Components (Shiloach-Vishkin [1])

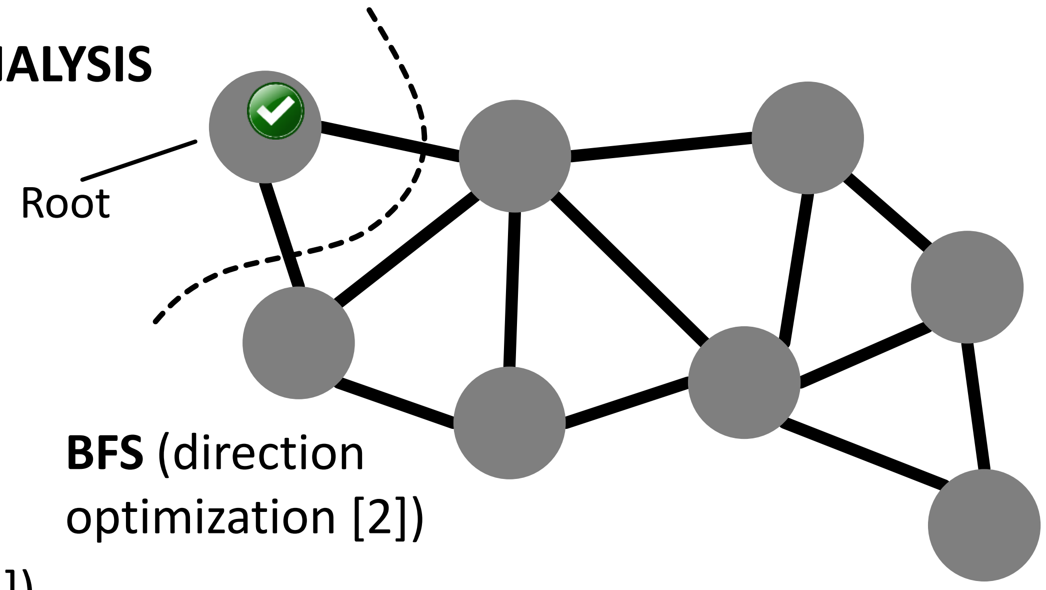


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

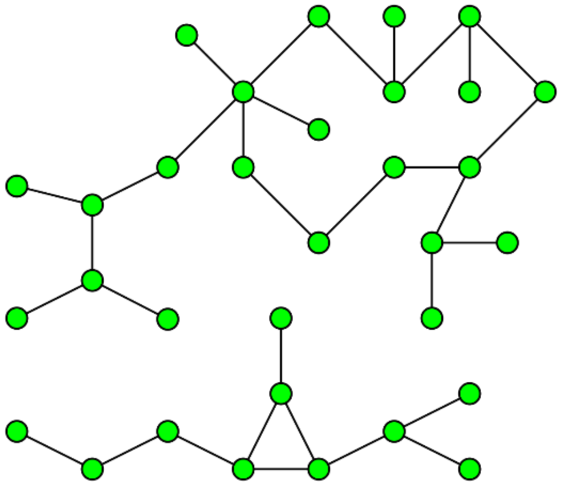
Performance Analysis

Algorithms



BFS (direction optimization [2])

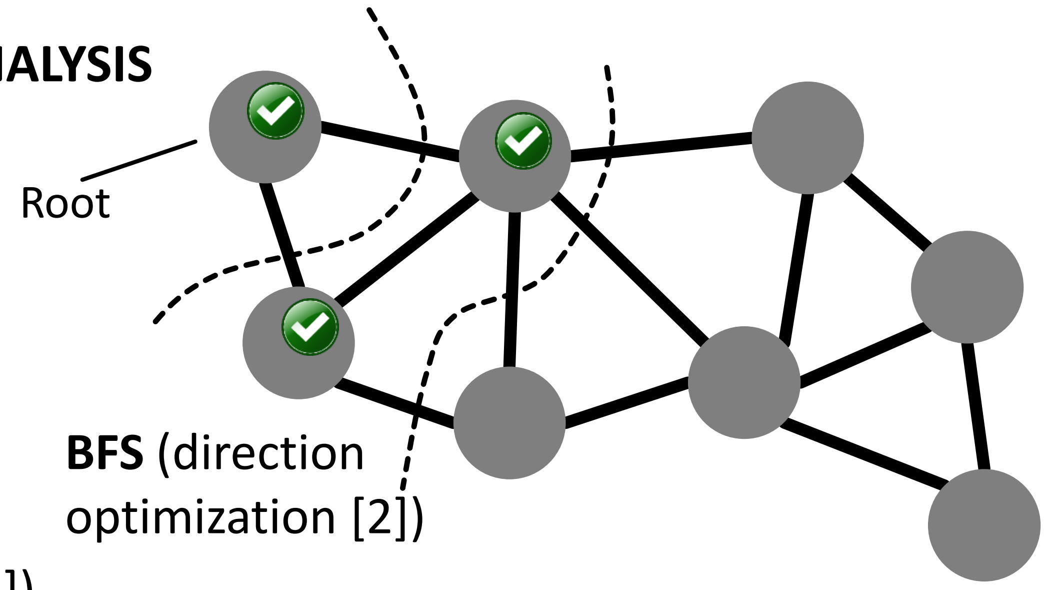
Connected Components
(Shiloach-Vishkin [1])



[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

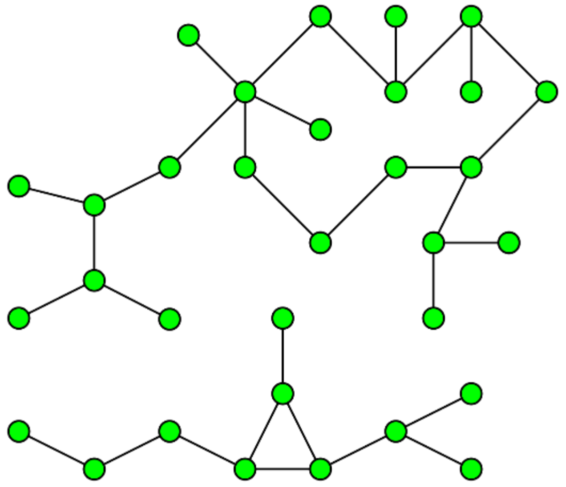
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

PERFORMANCE ANALYSIS
ALGORITHMS



BFS (direction optimization [2])

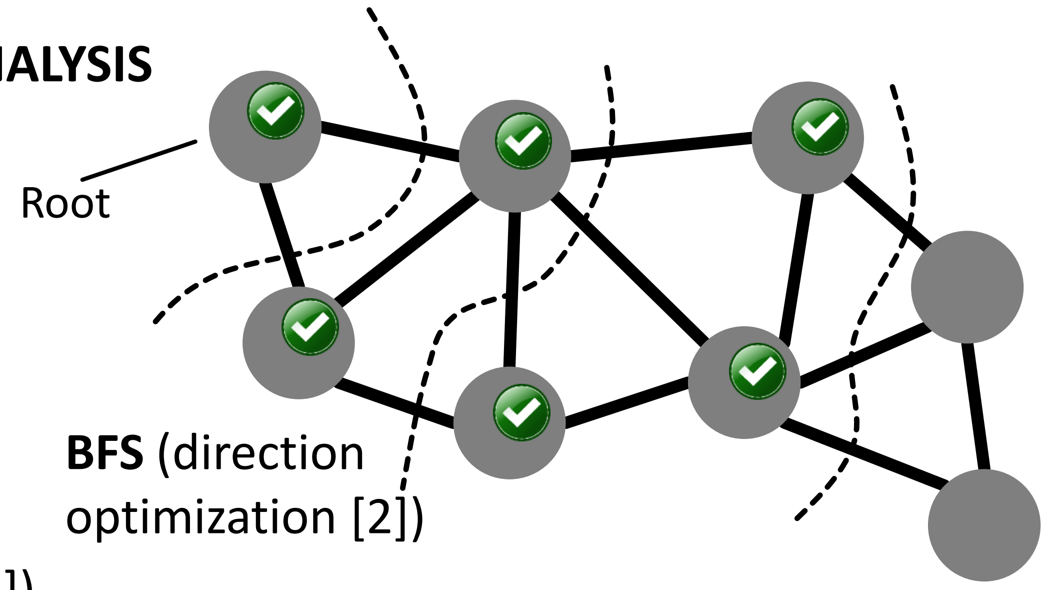
Connected Components
(Shiloach-Vishkin [1])



[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

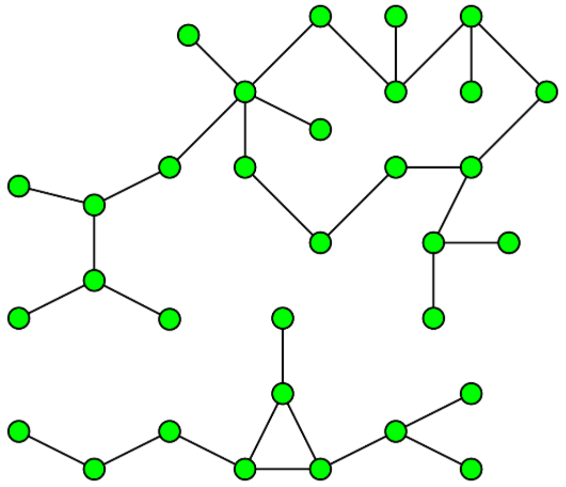
PERFORMANCE ANALYSIS

ALGORITHMS



BFS (direction optimization [2])

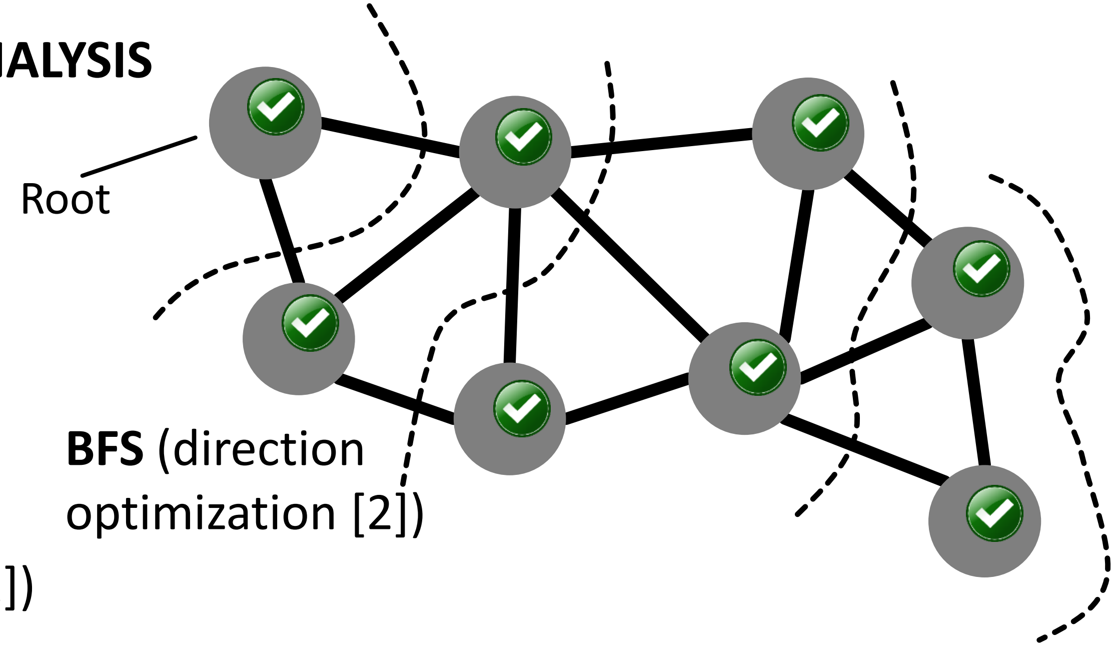
Connected Components
(Shiloach-Vishkin [1])



[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

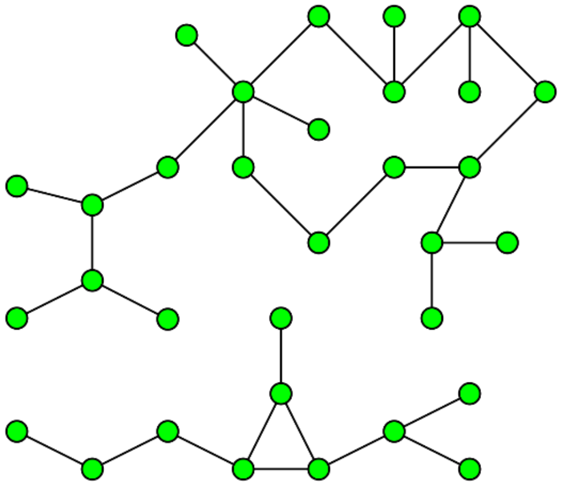
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

PERFORMANCE ANALYSIS
ALGORITHMS



BFS (direction optimization [2])

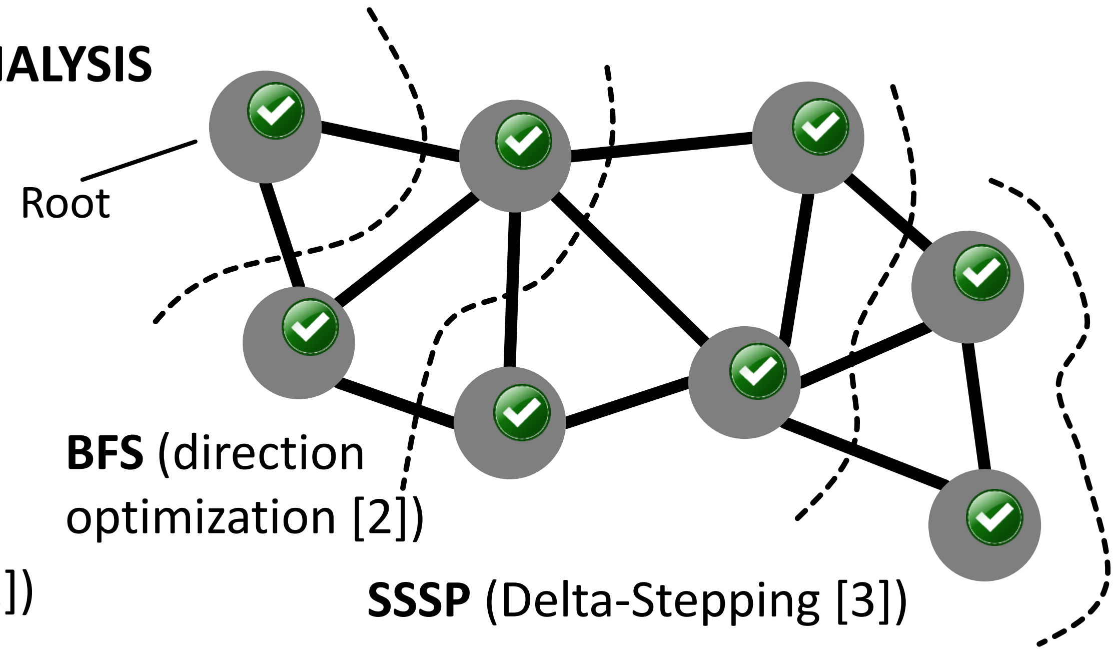
Connected Components
(Shiloach-Vishkin [1])



[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

Performance Analysis

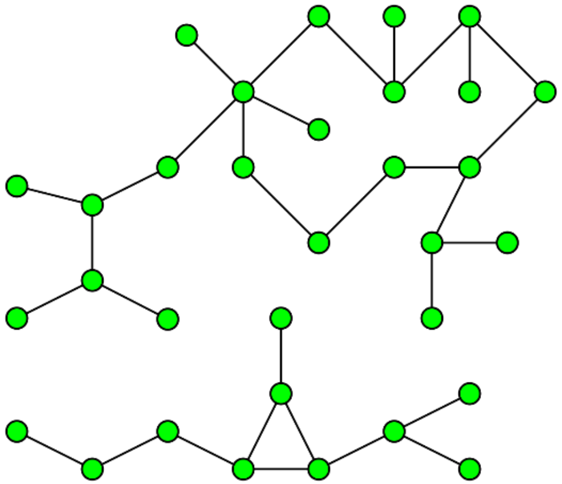
Algorithms



BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Connected Components (Shiloach-Vishkin [1])

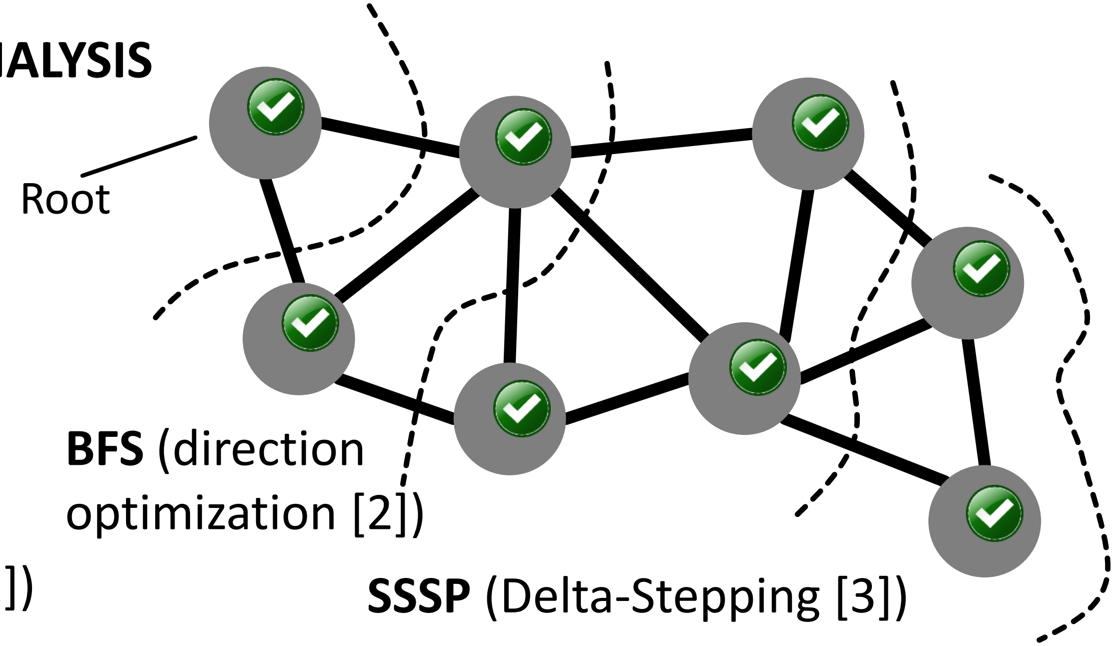


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

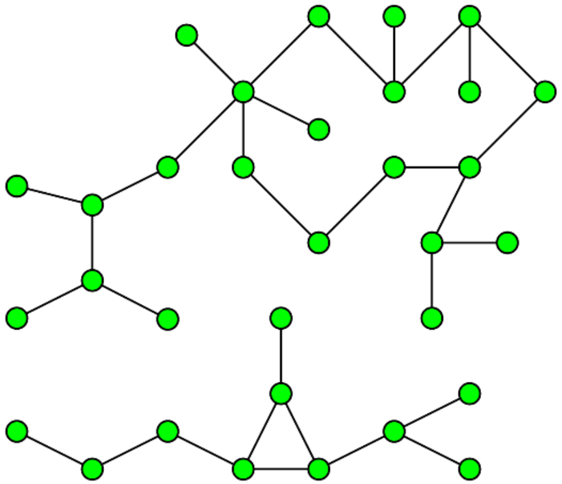
PERFORMANCE ANALYSIS
ALGORITHMS



BFS (direction optimization [2])

SSSP (Delta-Stepping [3])

Connected Components
(Shiloach-Vishkin [1])

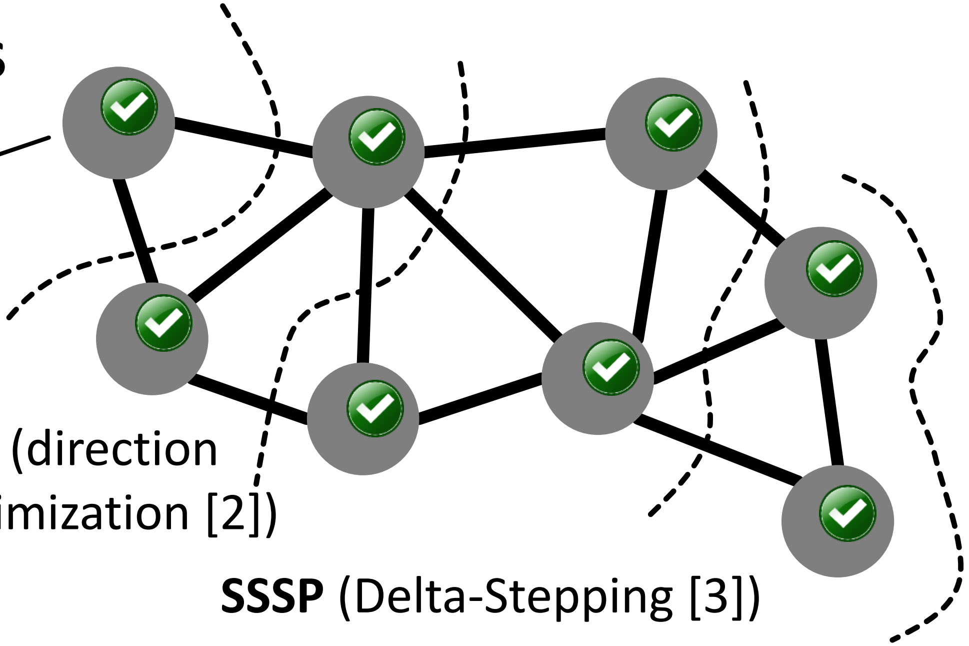


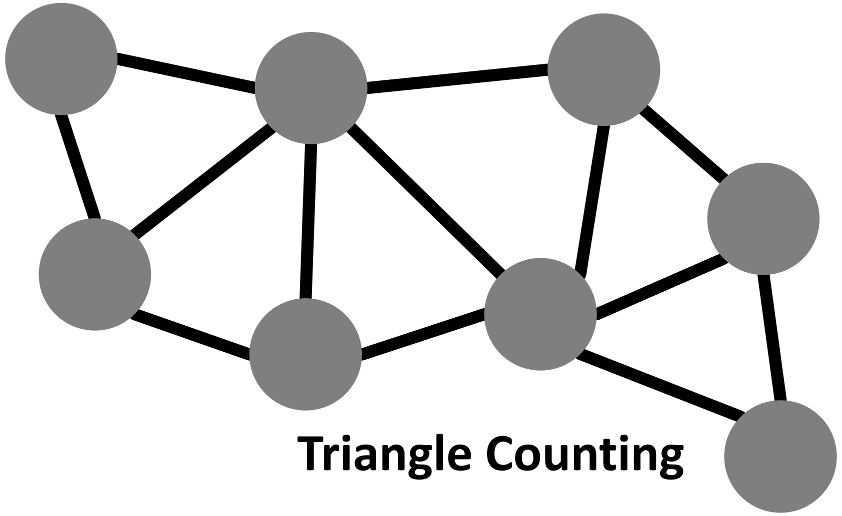
Triangle Counting

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

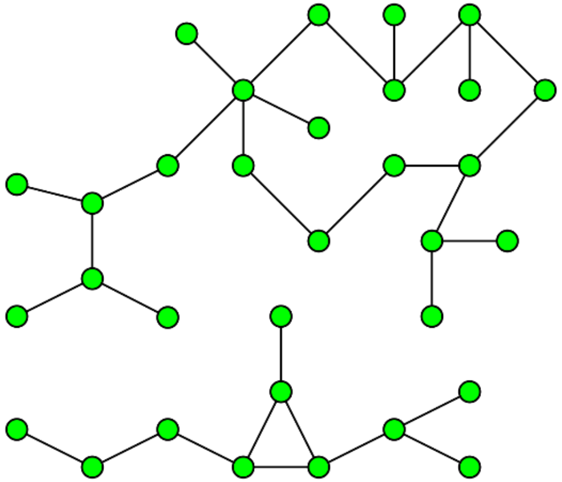
Performance Analysis

Algorithms





Connected Components
(Shiloach-Vishkin [1])



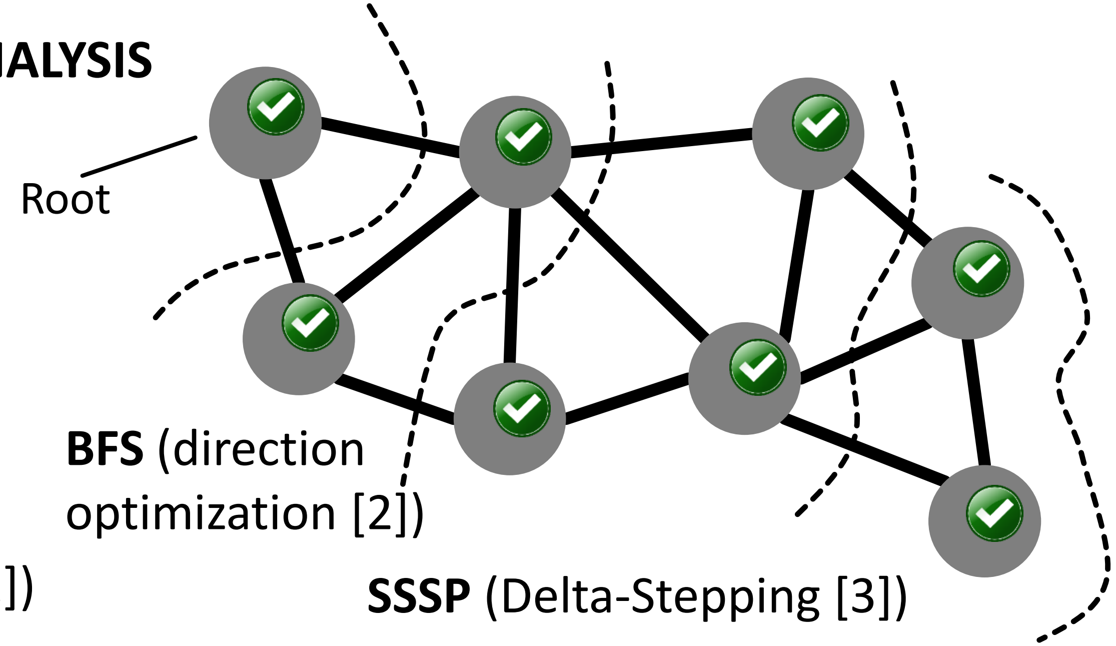
[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

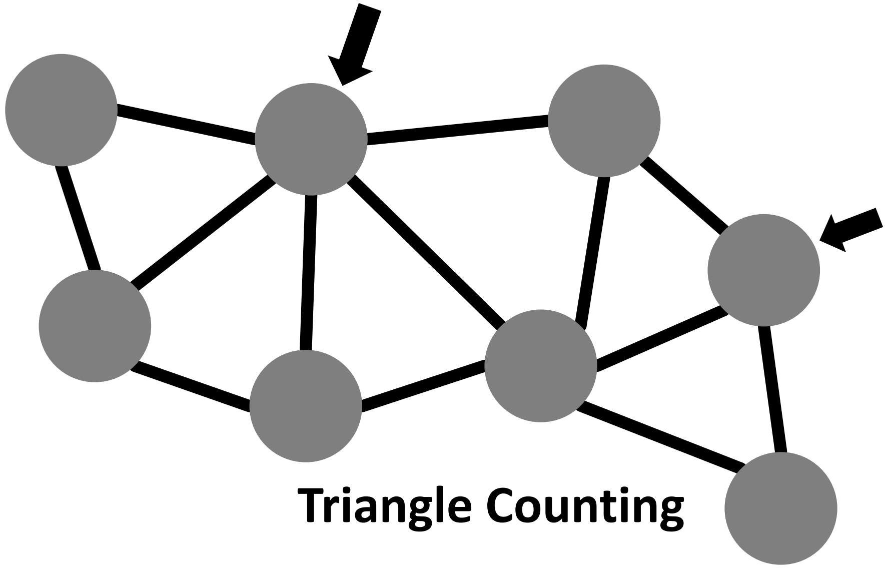
Performance Analysis

Algorithms



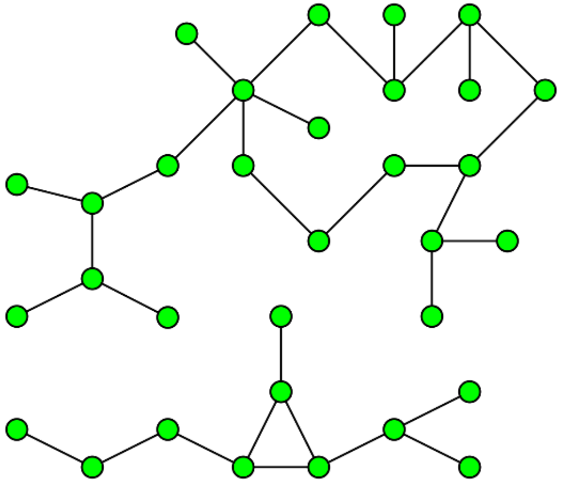
BFS (direction optimization [2])

SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components (Shiloach-Vishkin [1])



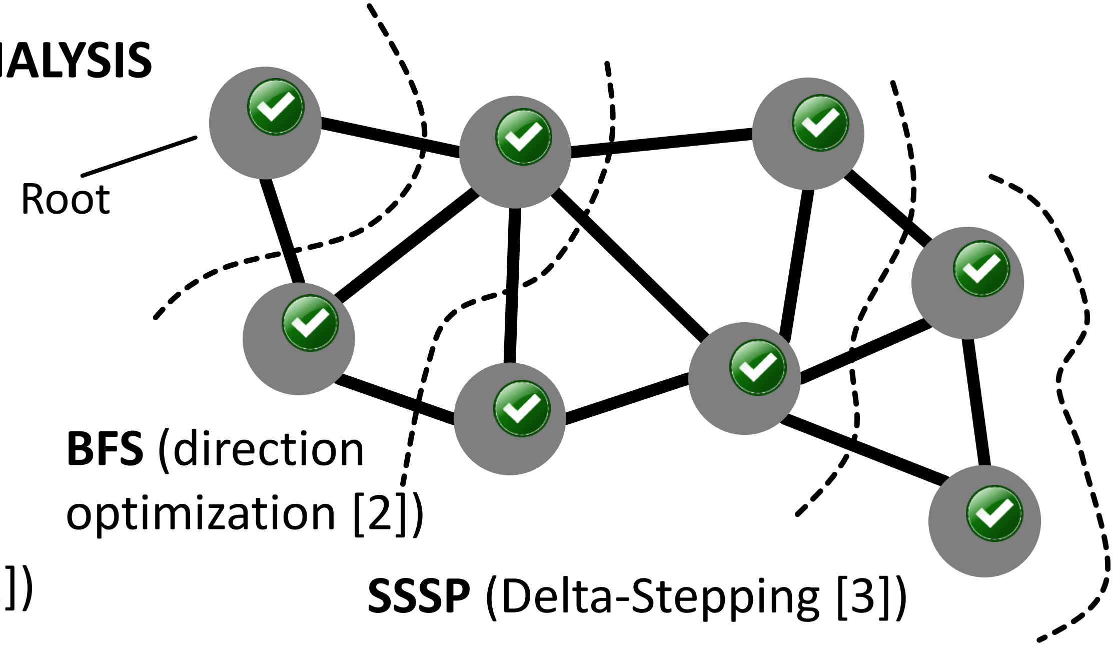
[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

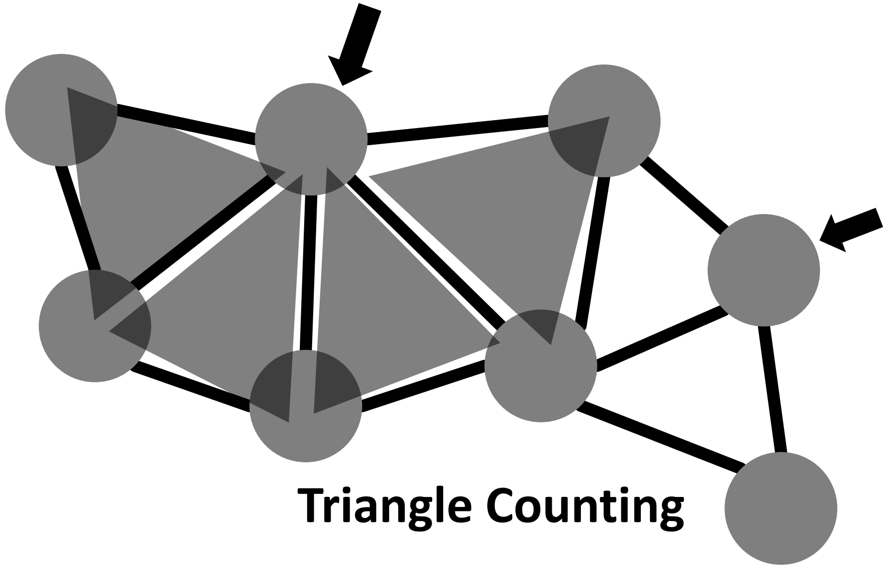
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

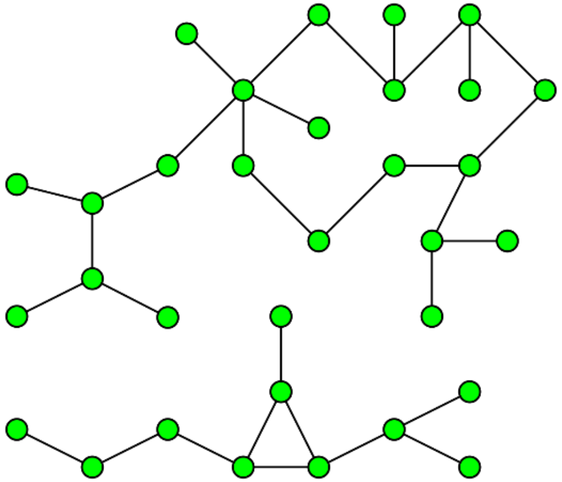
Performance Analysis

Algorithms





Connected Components (Shiloach-Vishkin [1])



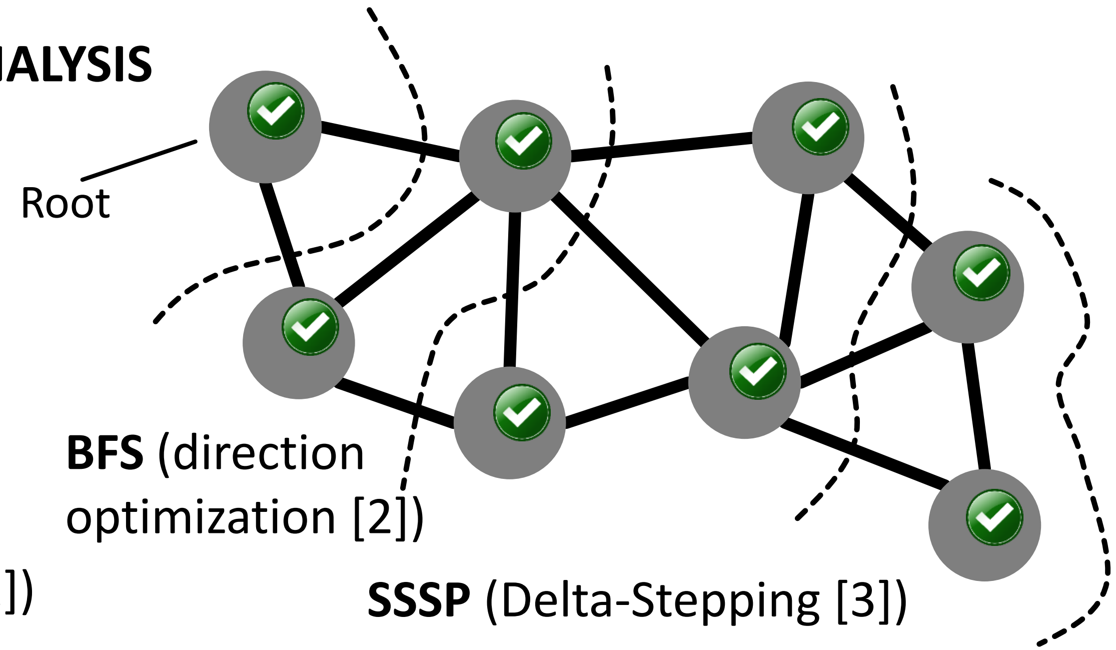
[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

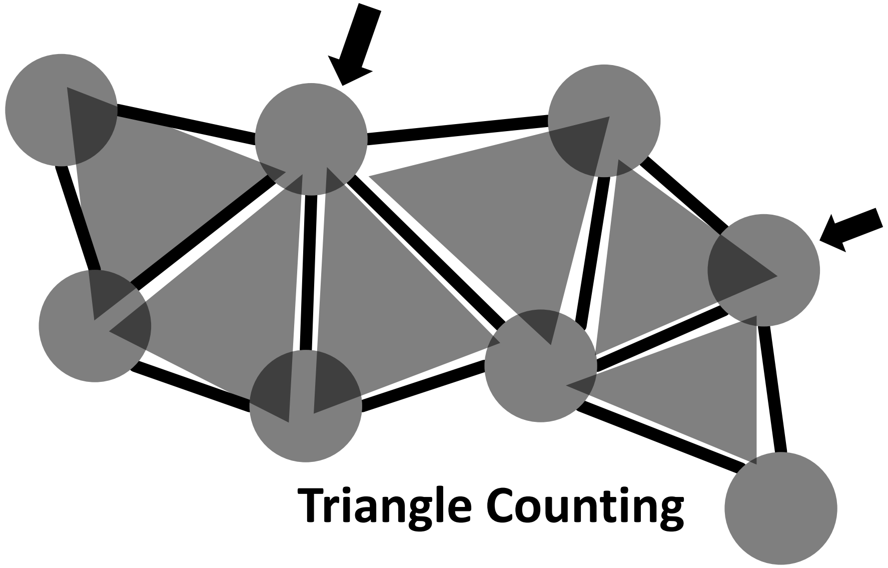
Performance Analysis

Algorithms



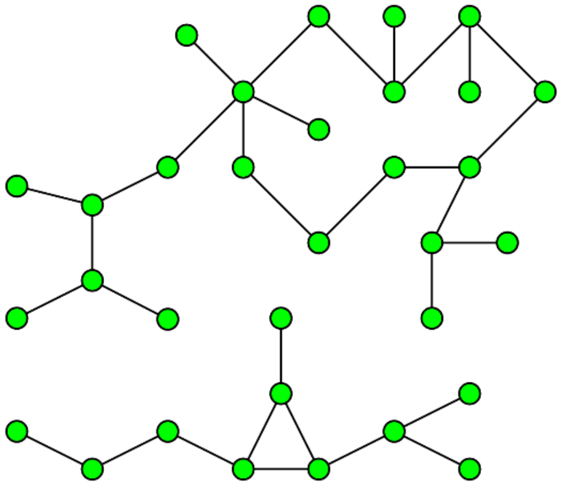
BFS (direction optimization [2])

SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components (Shiloach-Vishkin [1])

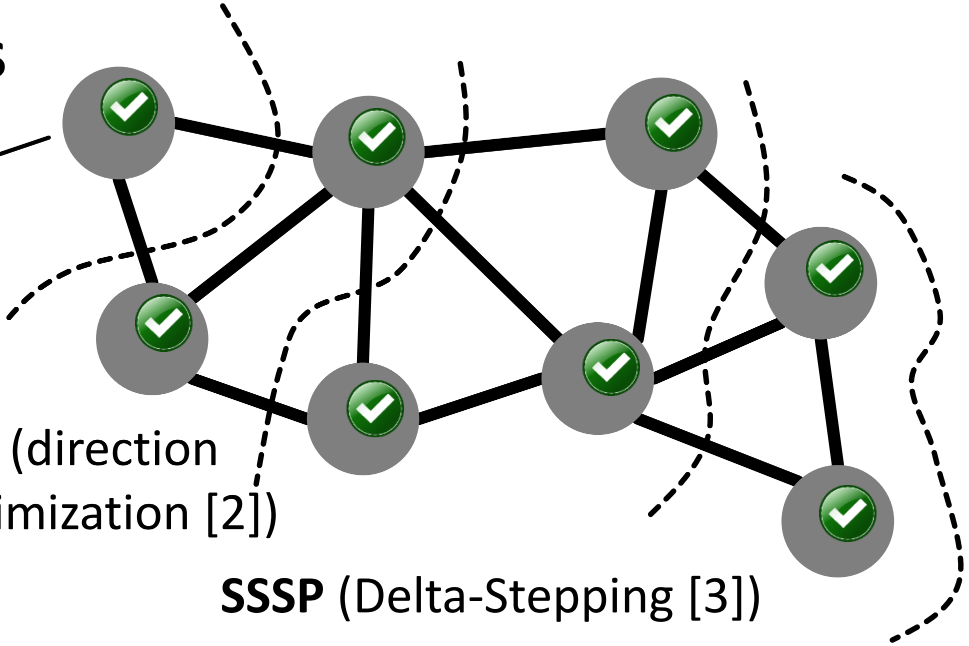


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

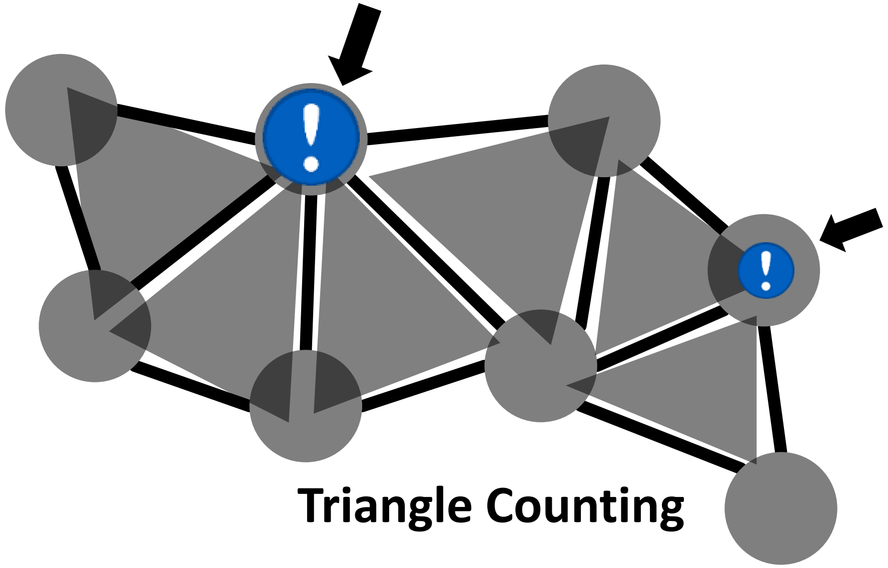
PERFORMANCE ANALYSIS
ALGORITHMS



Root

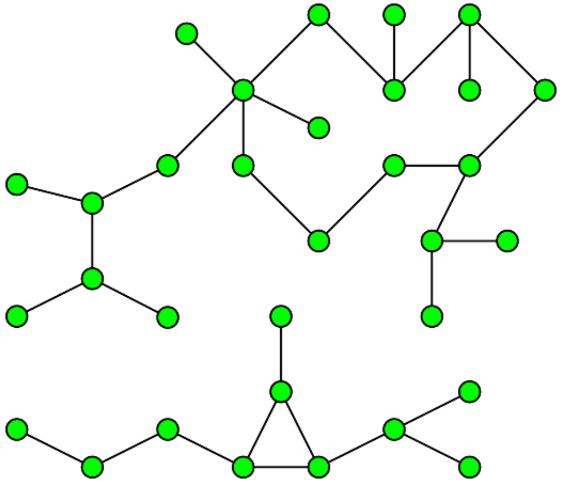
BFS (direction optimization [2])

SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components
(Shiloach-Vishkin [1])

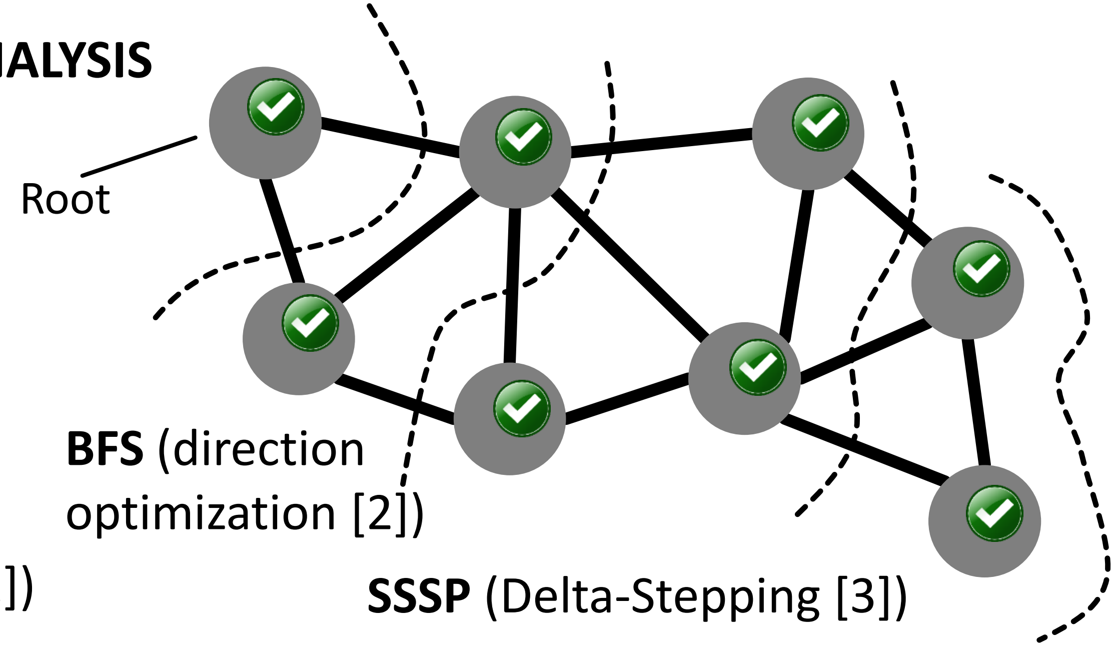


[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

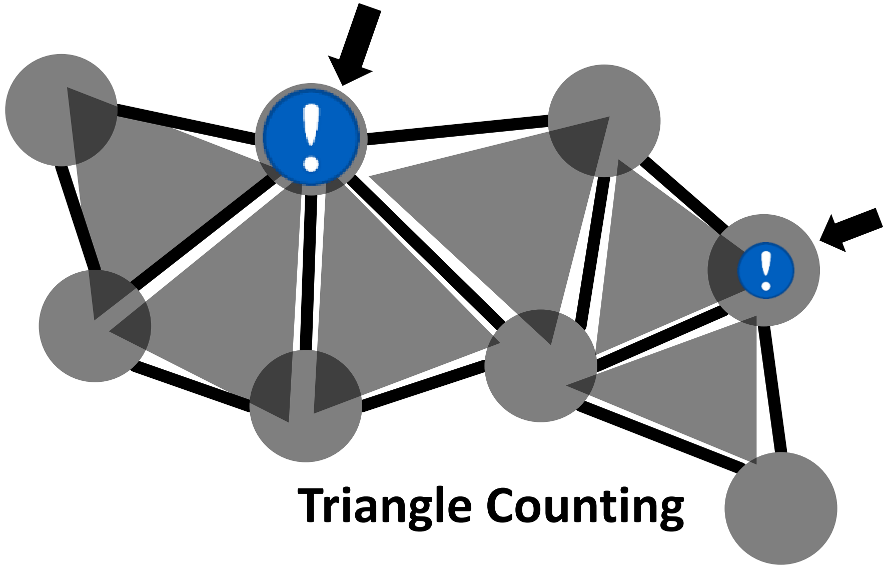
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

PERFORMANCE ANALYSIS
ALGORITHMS



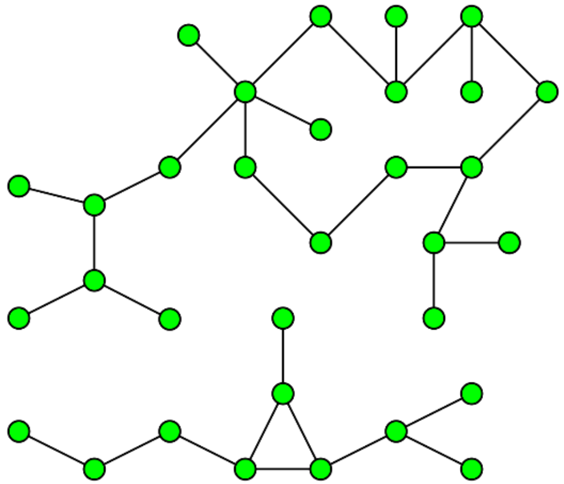
BFS (direction optimization [2])

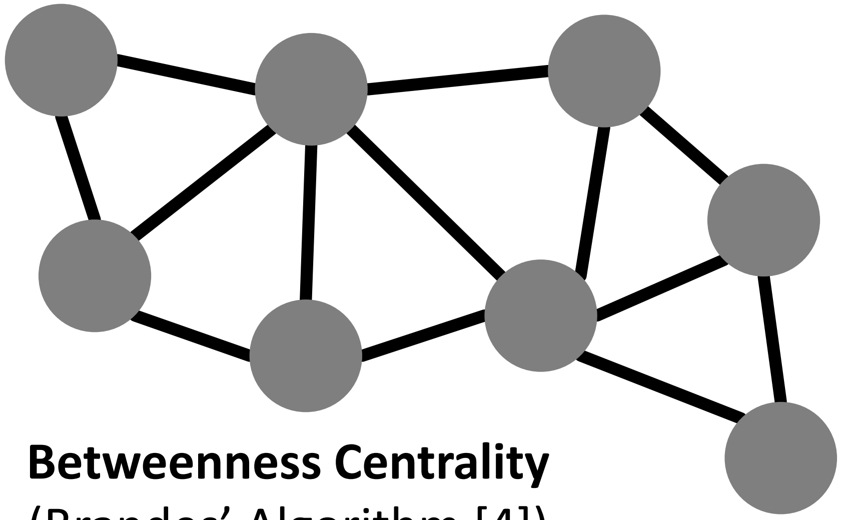
SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components (Shiloach-Vishkin [1])



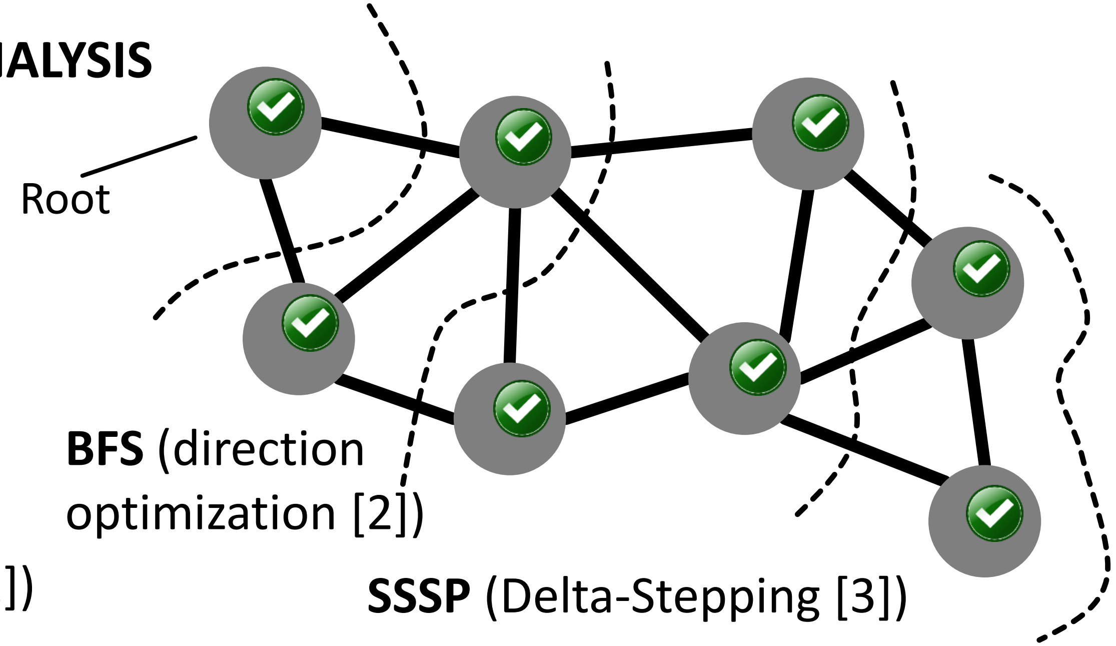


Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

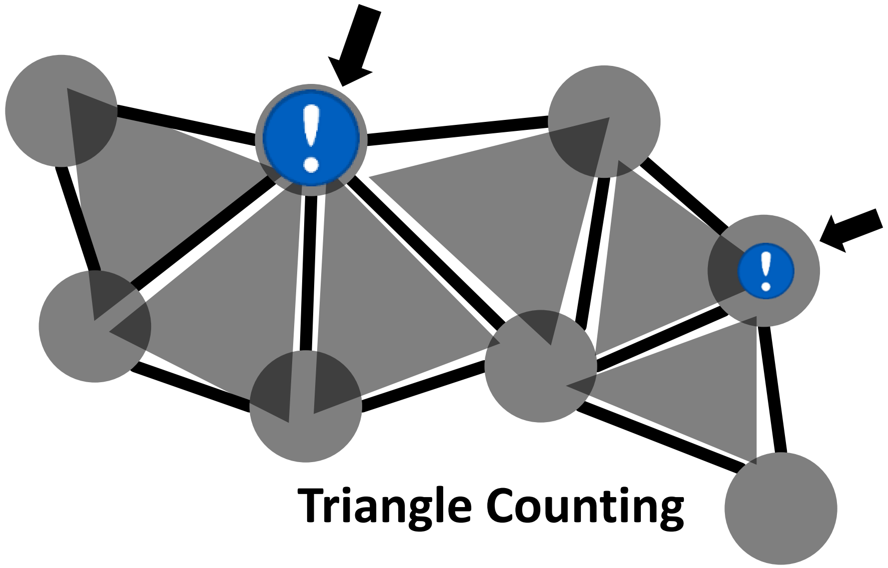
PERFORMANCE ANALYSIS

ALGORITHMS



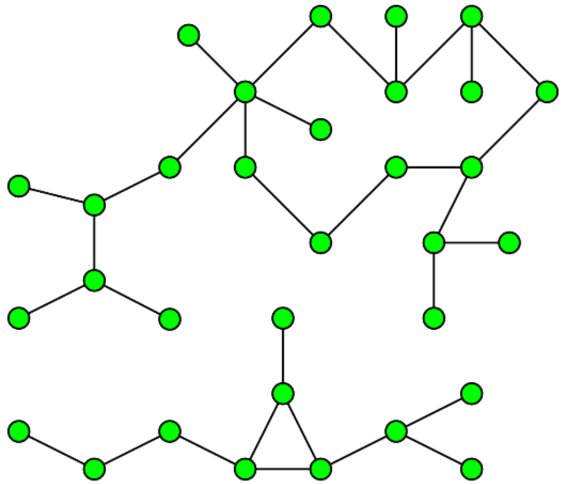
BFS (direction optimization [2])

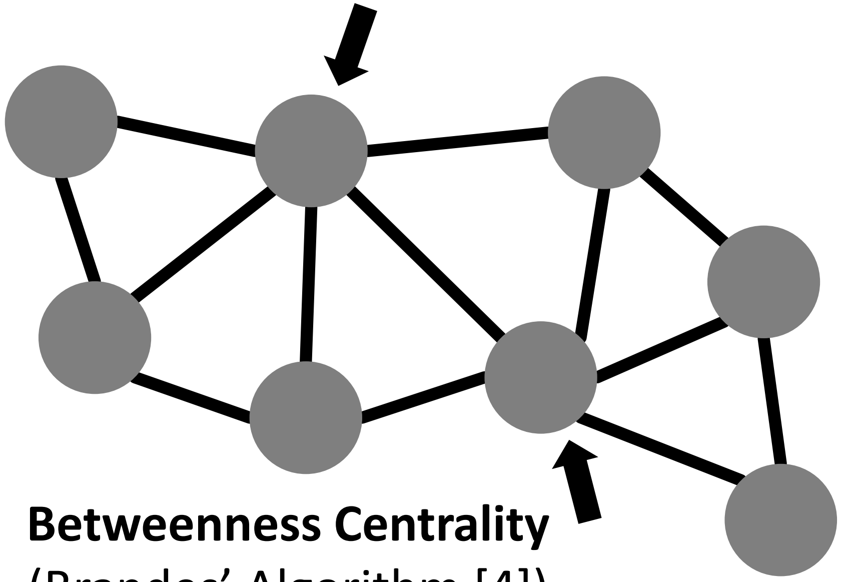
SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components (Shiloach-Vishkin [1])





Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

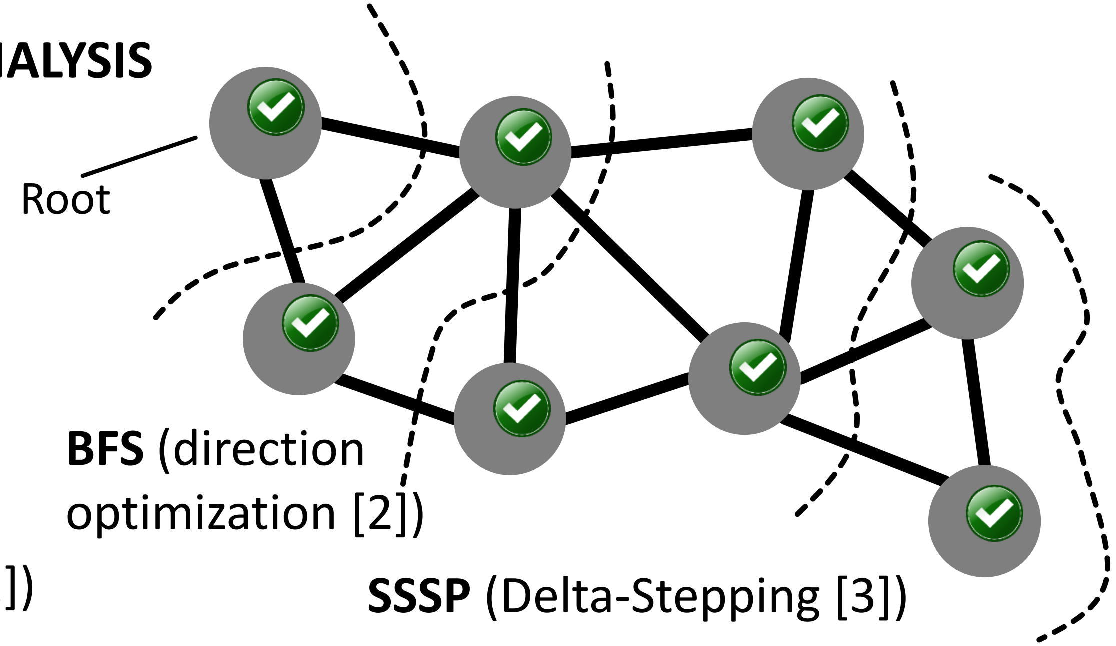
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

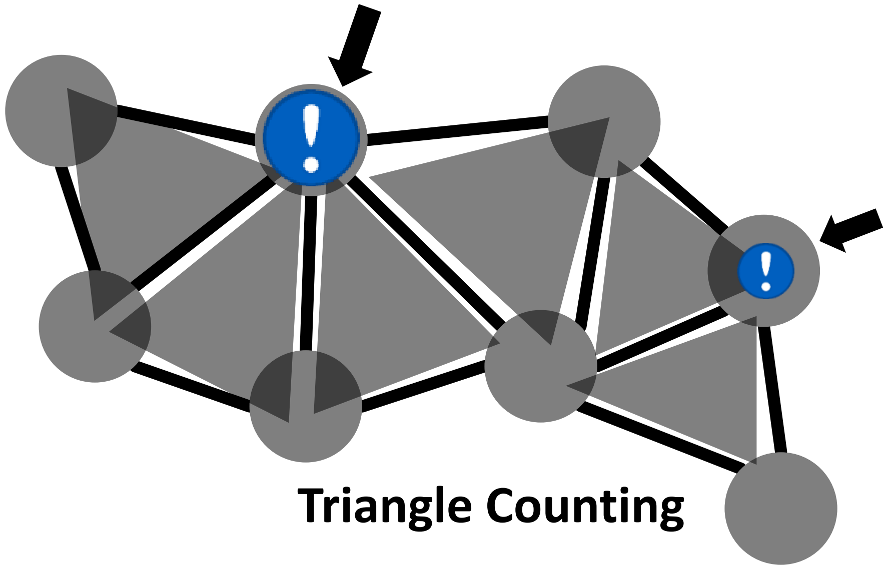
Performance Analysis

Algorithms



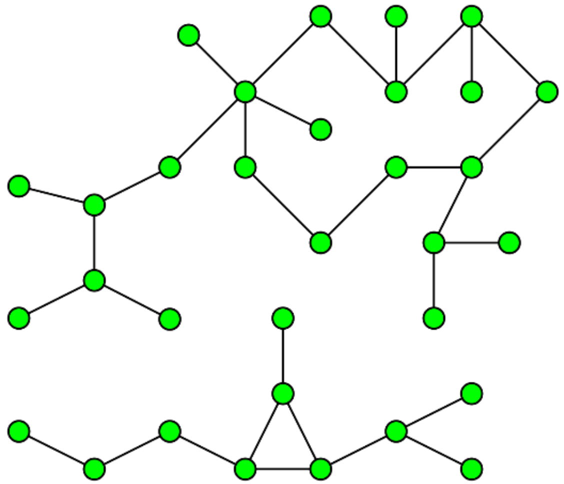
BFS (direction optimization [2])

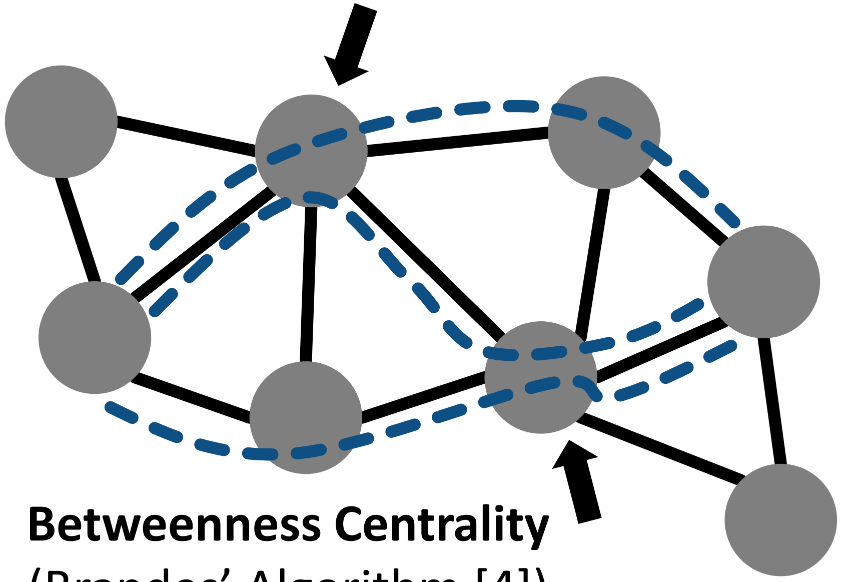
SSSP (Delta-Stepping [3])



Triangle Counting

Connected Components (Shiloach-Vishkin [1])





Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.

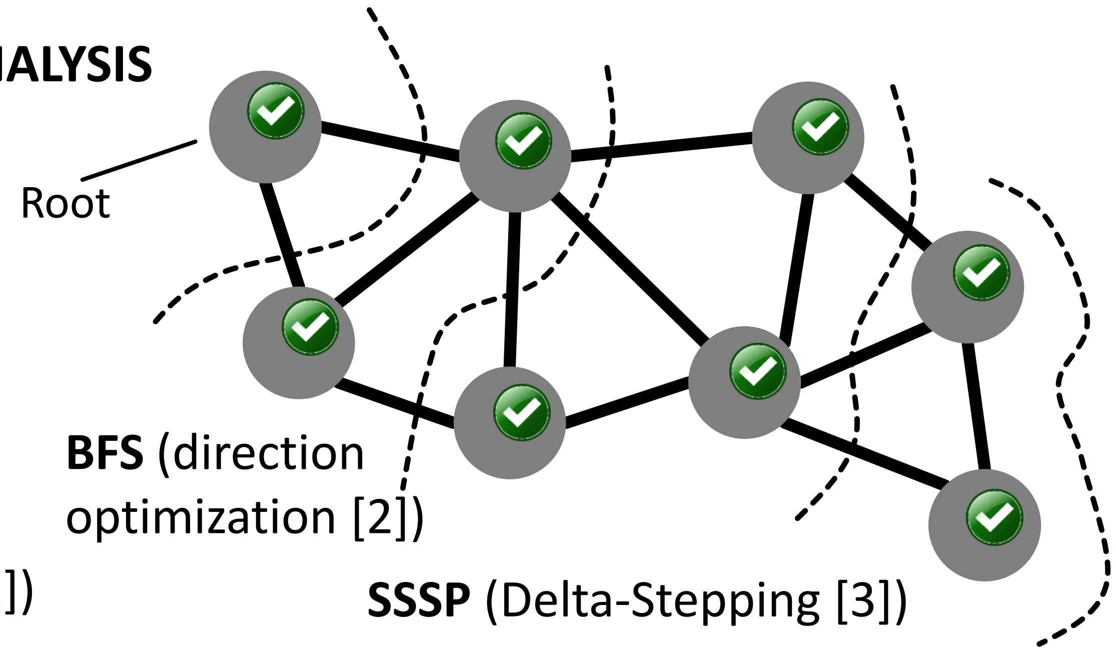
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.

[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.

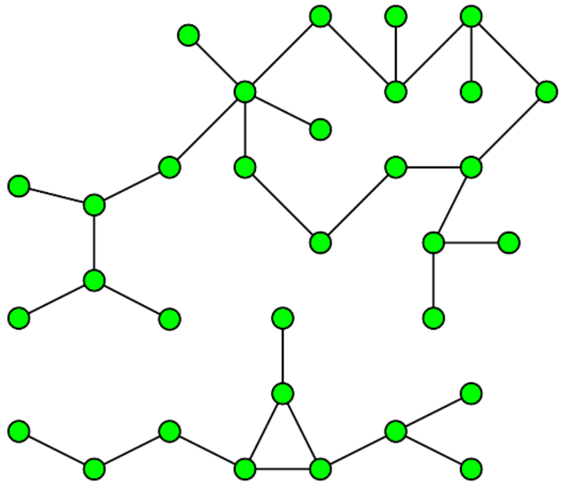
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

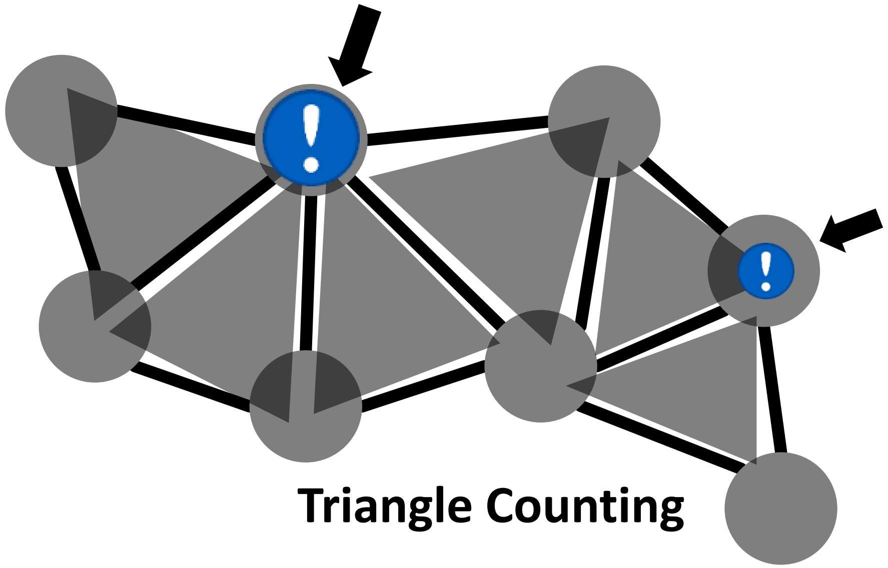
Performance Analysis

Algorithms

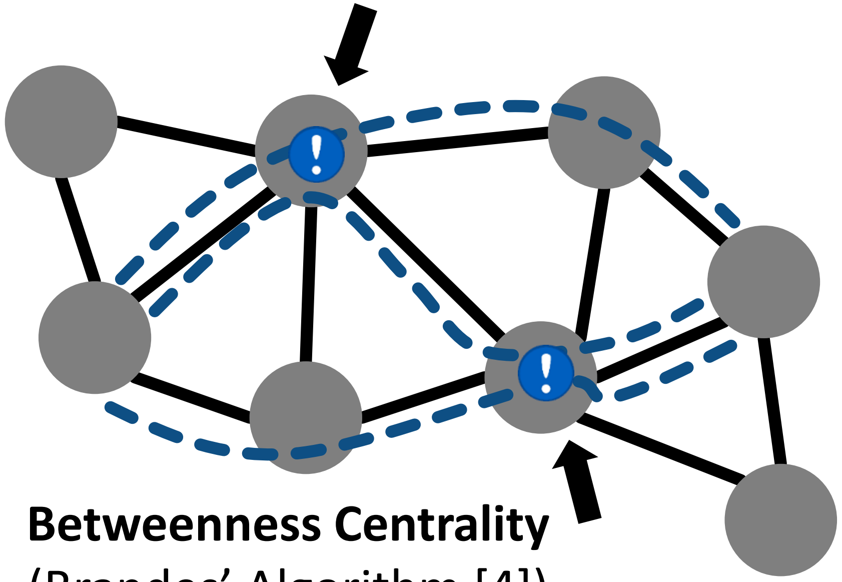


Connected Components (Shiloach-Vishkin [1])





Triangle Counting

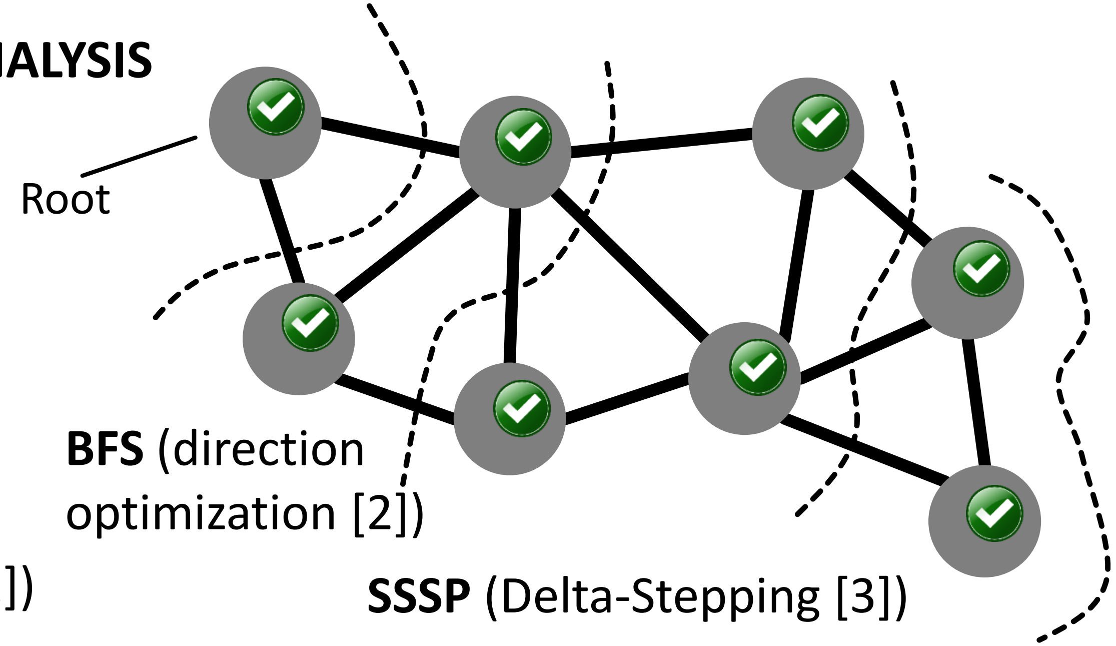


Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

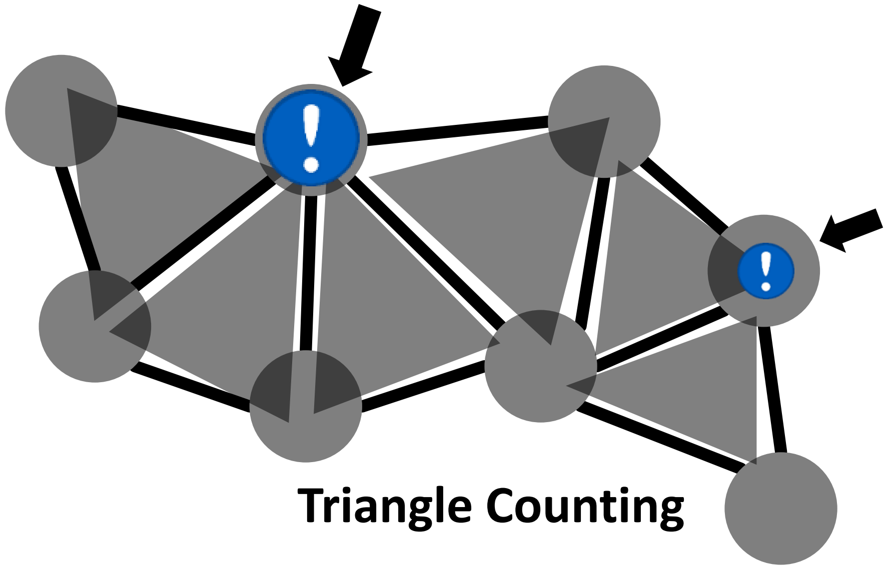
PERFORMANCE ANALYSIS

ALGORITHMS



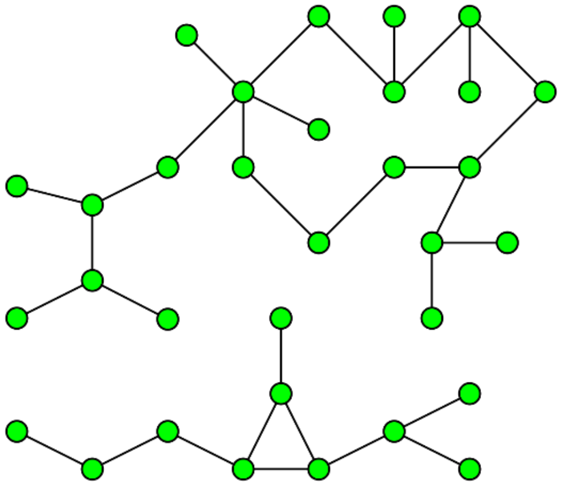
BFS (direction optimization [2])

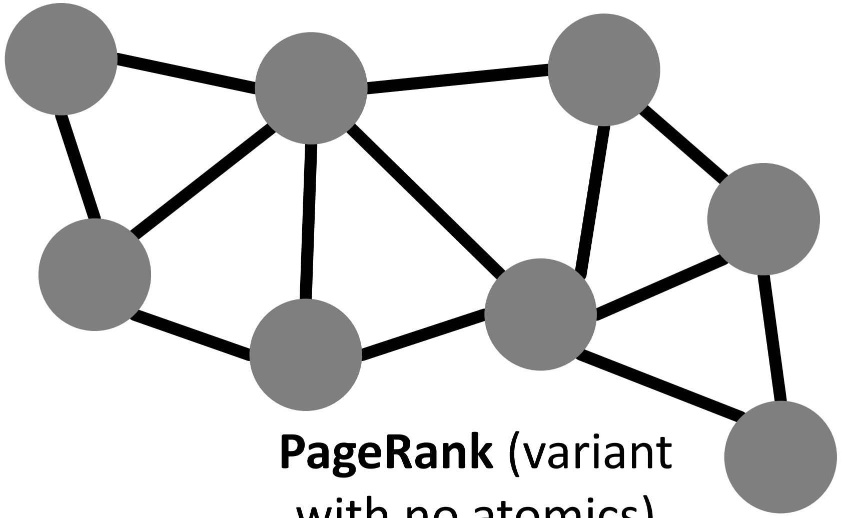
SSSP (Delta-Stepping [3])



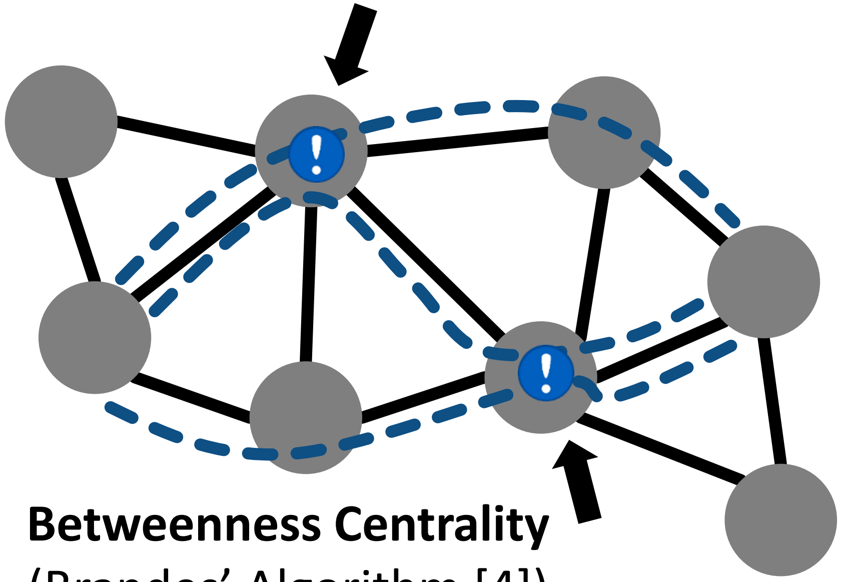
Triangle Counting

Connected Components (Shiloach-Vishkin [1])





PageRank (variant with no atomics)

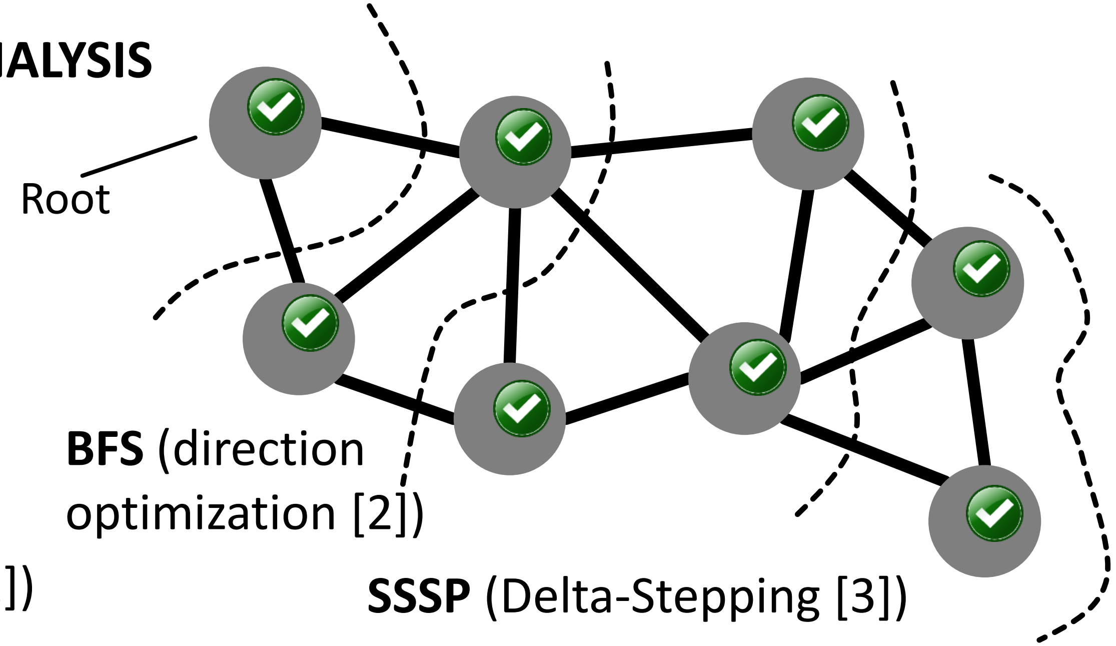


Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

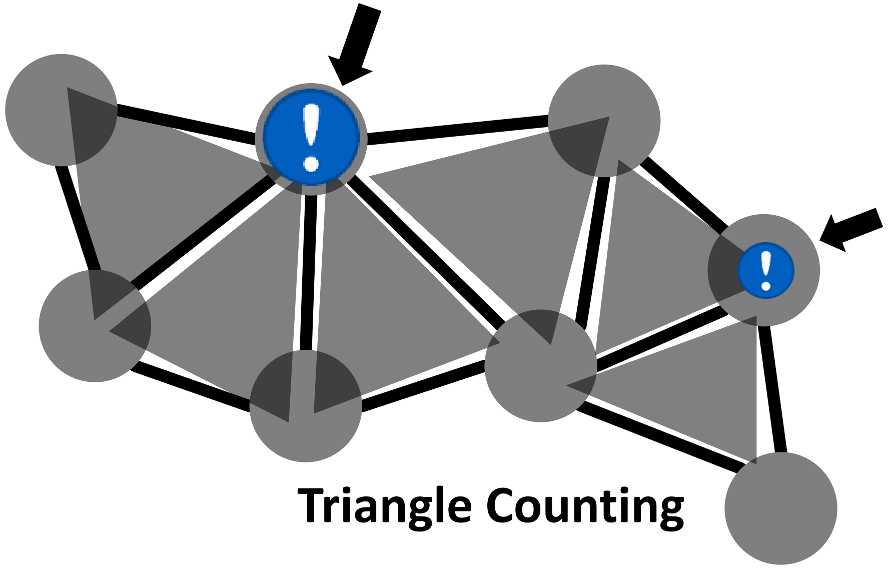
Performance Analysis

Algorithms



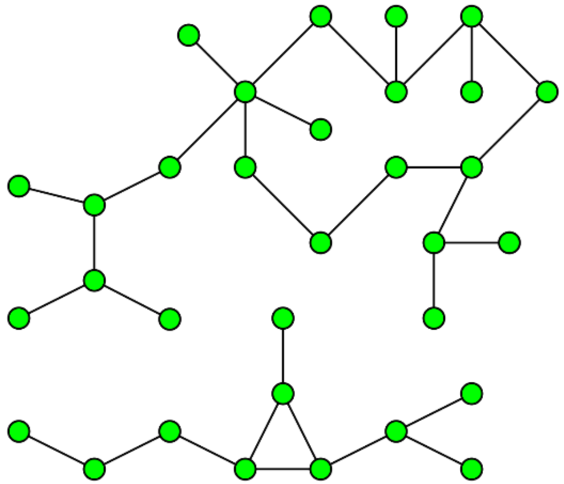
BFS (direction optimization [2])

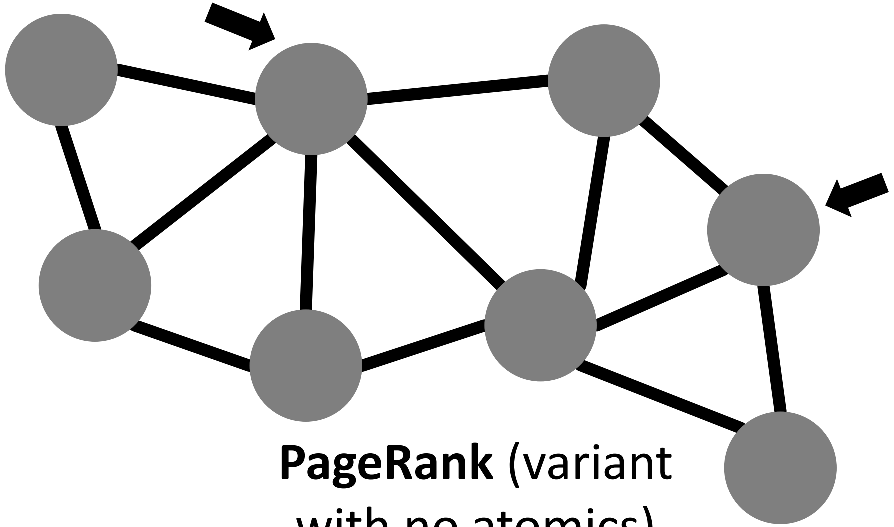
SSSP (Delta-Stepping [3])



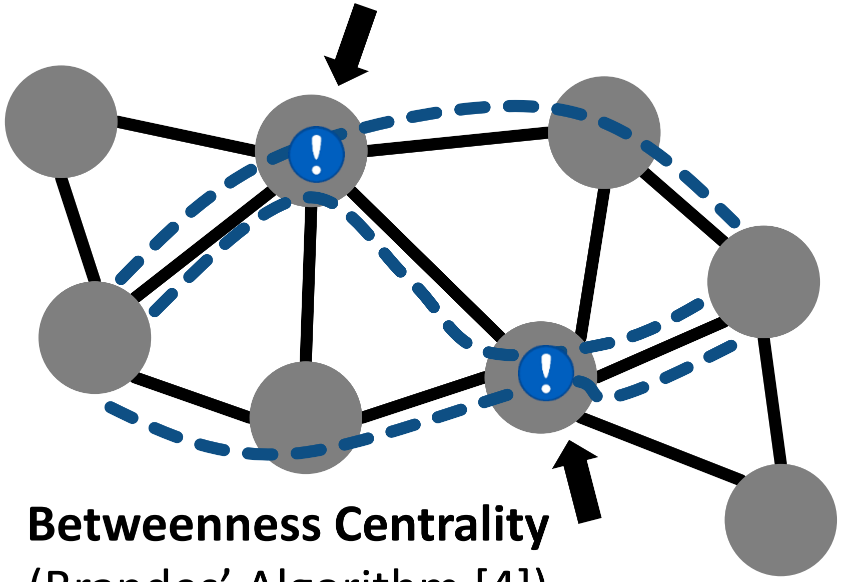
Triangle Counting

Connected Components (Shiloach-Vishkin [1])





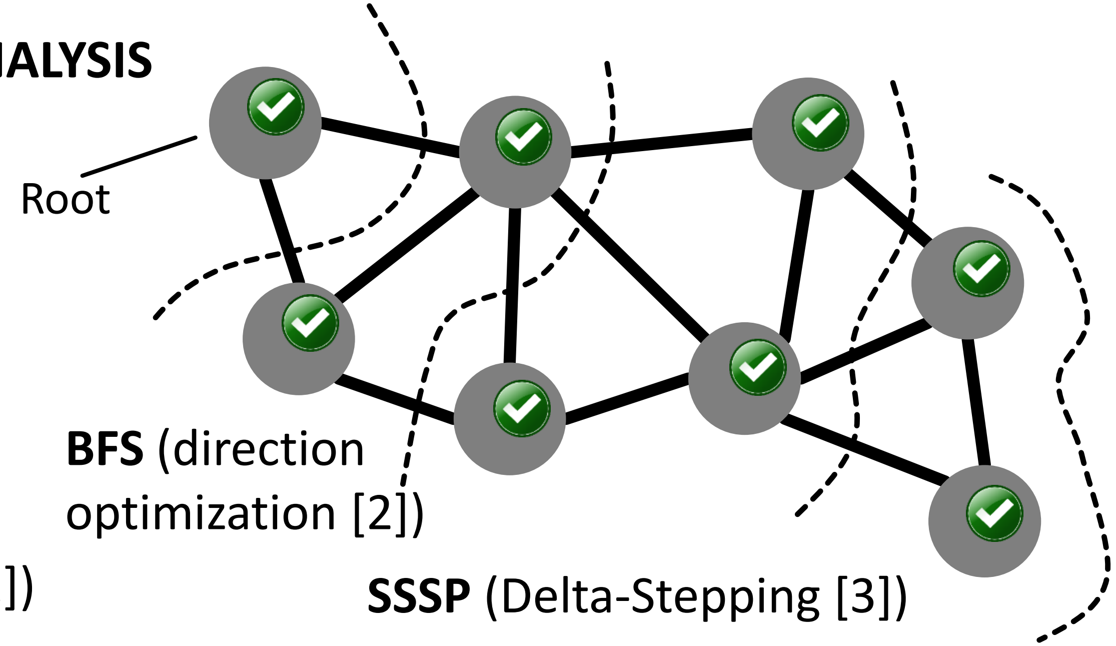
PageRank (variant with no atomics)



Betweenness Centrality (Brandes' Algorithm [4])

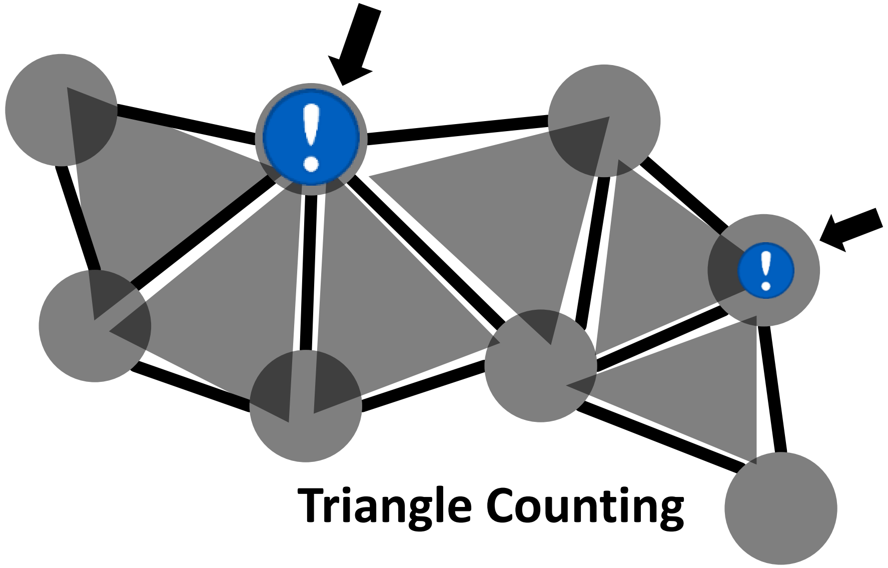
[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

Performance Analysis
Algorithms



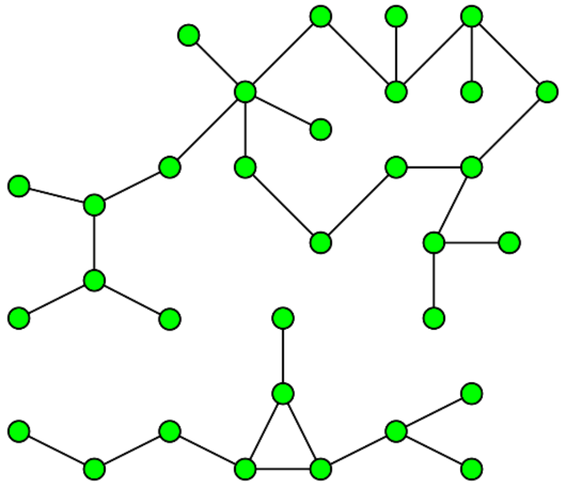
BFS (direction optimization [2])

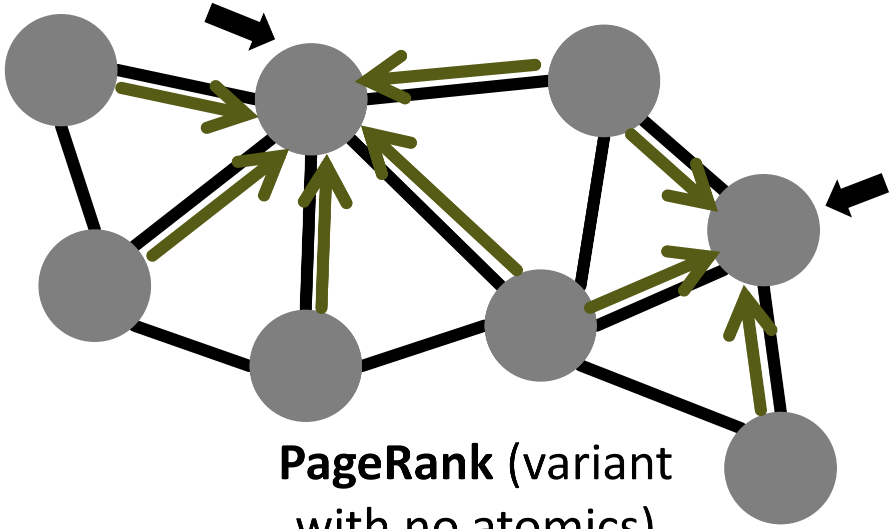
SSSP (Delta-Stepping [3])



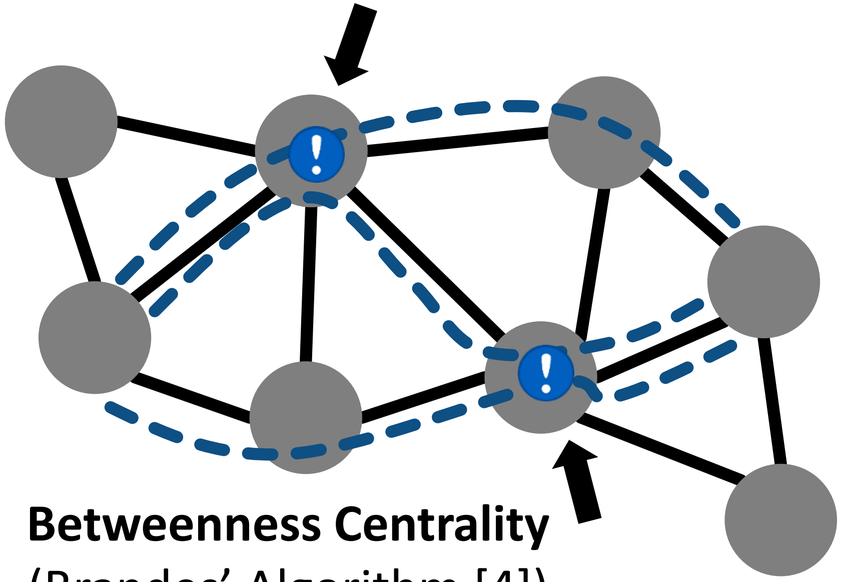
Triangle Counting

Connected Components (Shiloach-Vishkin [1])





PageRank (variant with no atomics)

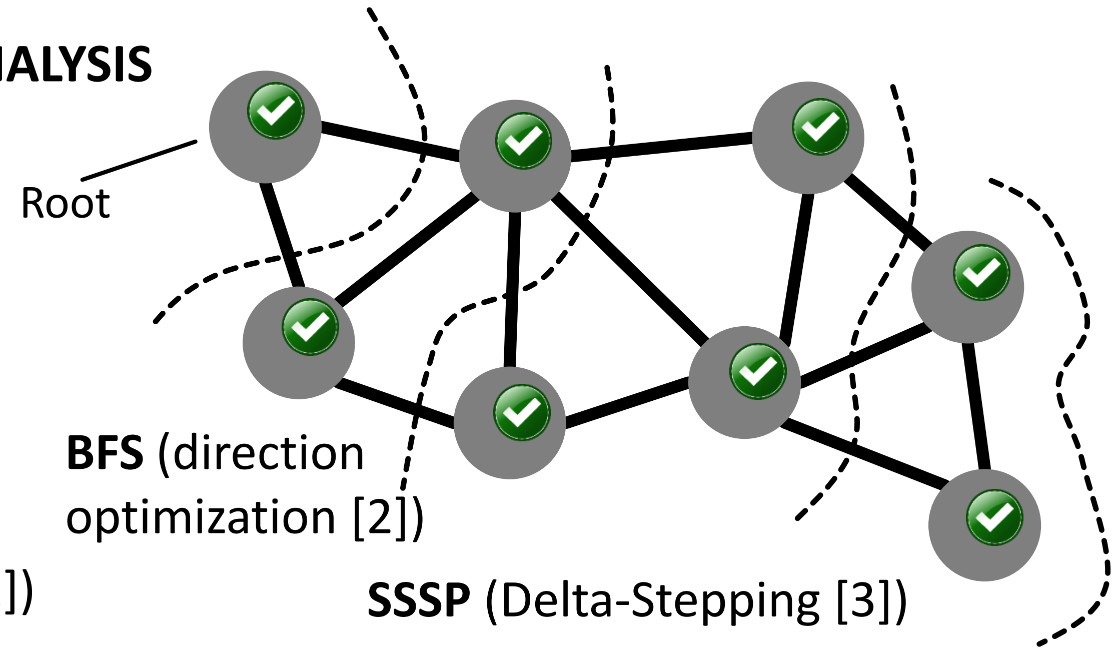


Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

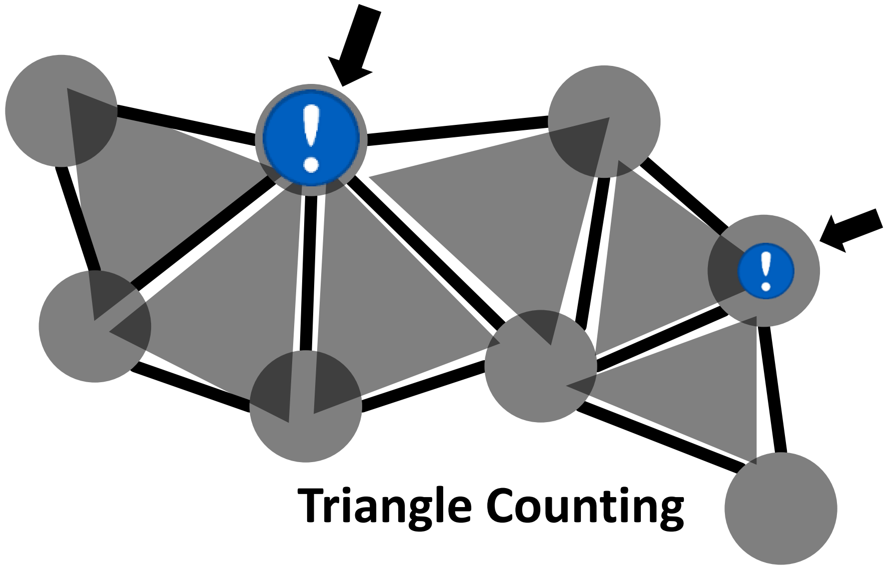
Performance Analysis

Algorithms



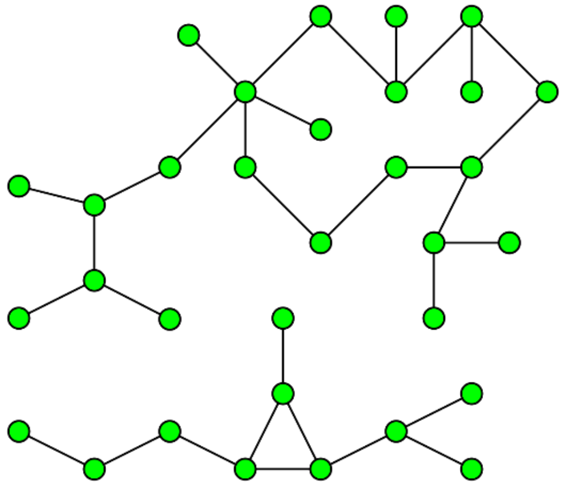
BFS (direction optimization [2])

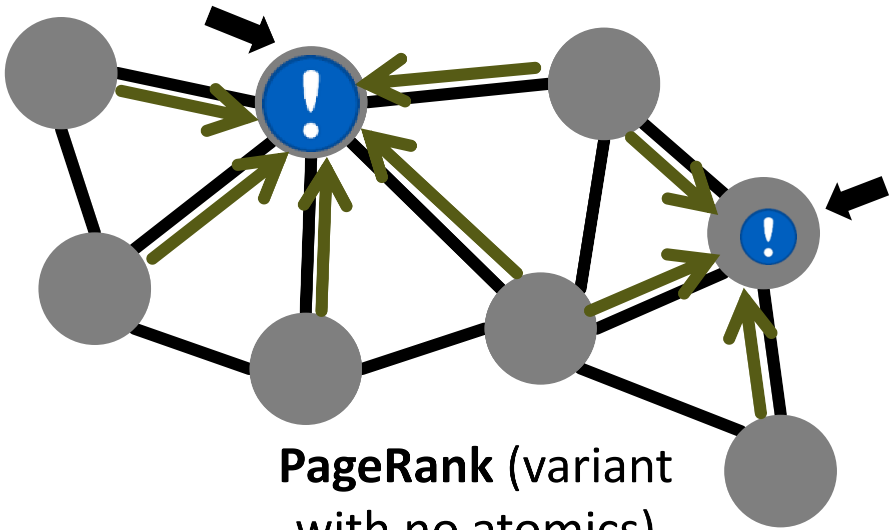
SSSP (Delta-Stepping [3])



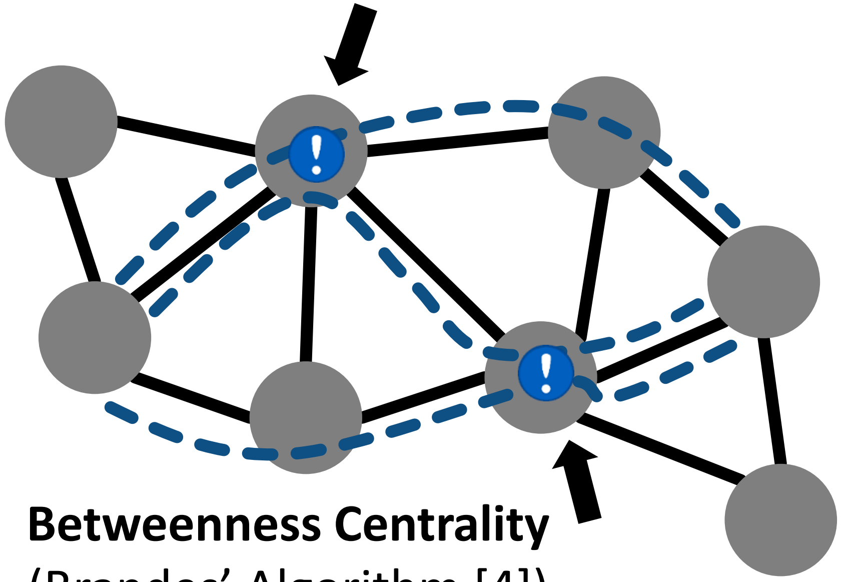
Triangle Counting

Connected Components (Shiloach-Vishkin [1])





PageRank (variant with no atomics)



Betweenness Centrality (Brandes' Algorithm [4])

[1] Y. Shiloach, U. Vishkin. An O (log n) parallel connectivity algorithm. 1980.
[2] S Beamer et al. Direction-Optimizing Breadth-First Search. 2013.
[3] U. Meyer, P. Sanders. Delta-Stepping: A Parallelizable Shortest Path Algorithm. 2003.
[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. 2001.

Performance Analysis

Comparison Targets

Performance Analysis

Comparison Targets

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

[1] S. Beamer, K. Asanovic, and D. Patterson. The GAP benchmark suite. arXiv preprint arXiv:1508.03619, 2015.

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

[1] S. Beamer, K. Asanovic, and D. Patterson. The GAP benchmark suite. arXiv preprint arXiv:1508.03619, 2015.
[2] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification, 1996.

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

[1] S. Beamer, K. Asanovic, and D. Patterson. The GAP benchmark suite. arXiv preprint arXiv:1508.03619, 2015.
[2] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification, 1996.
[3] P. Boldi and S. Vigna. The WebGraph Framework I: compression echniques. WWW, 2004.

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

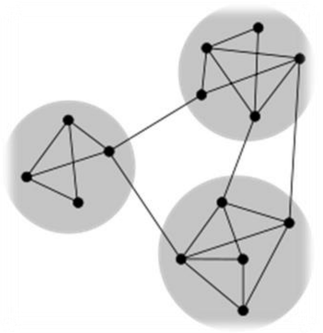
WebGraph Library [3]
Comparison to a state-of-the-art graph compression scheme

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

[1] S. Beamer, K. Asanovic, and D. Patterson. The GAP benchmark suite. arXiv preprint arXiv:1508.03619, 2015.
[2] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification, 1996.
[3] P. Boldi and S. Vigna. The WebGraph Framework I: compression echniques. WWW, 2004.
[4] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact Representations of Separable Graphs. SODA, 2003.

SSSP

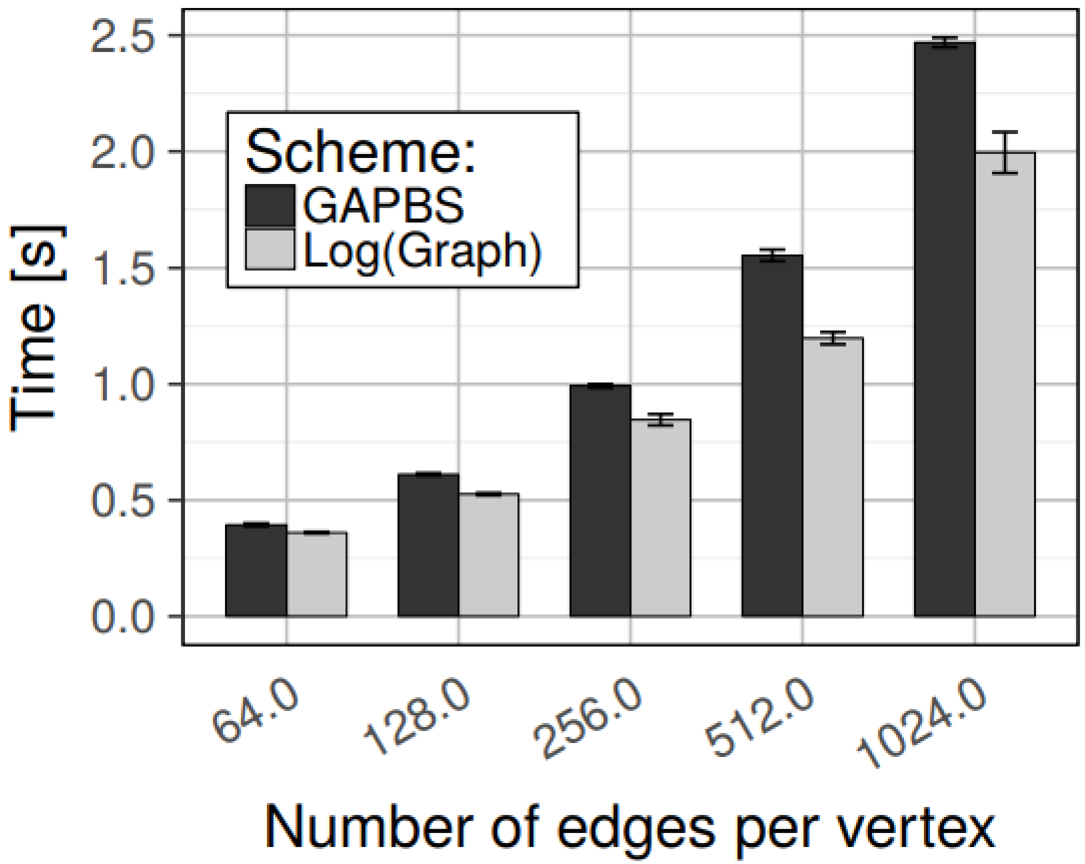
1 Log (Vertex labels), Log (Edge weights)   Storage, Performance

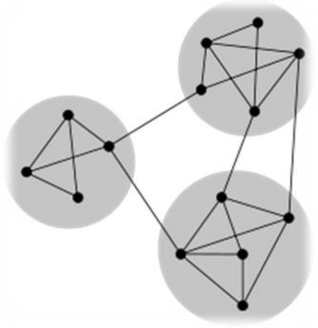


Kronecker graphs
Number of vertices: 4M

SSSP

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

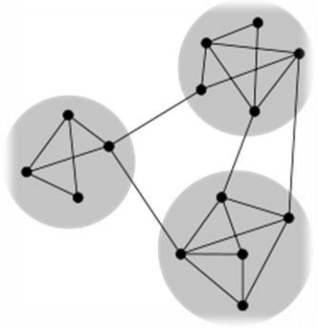




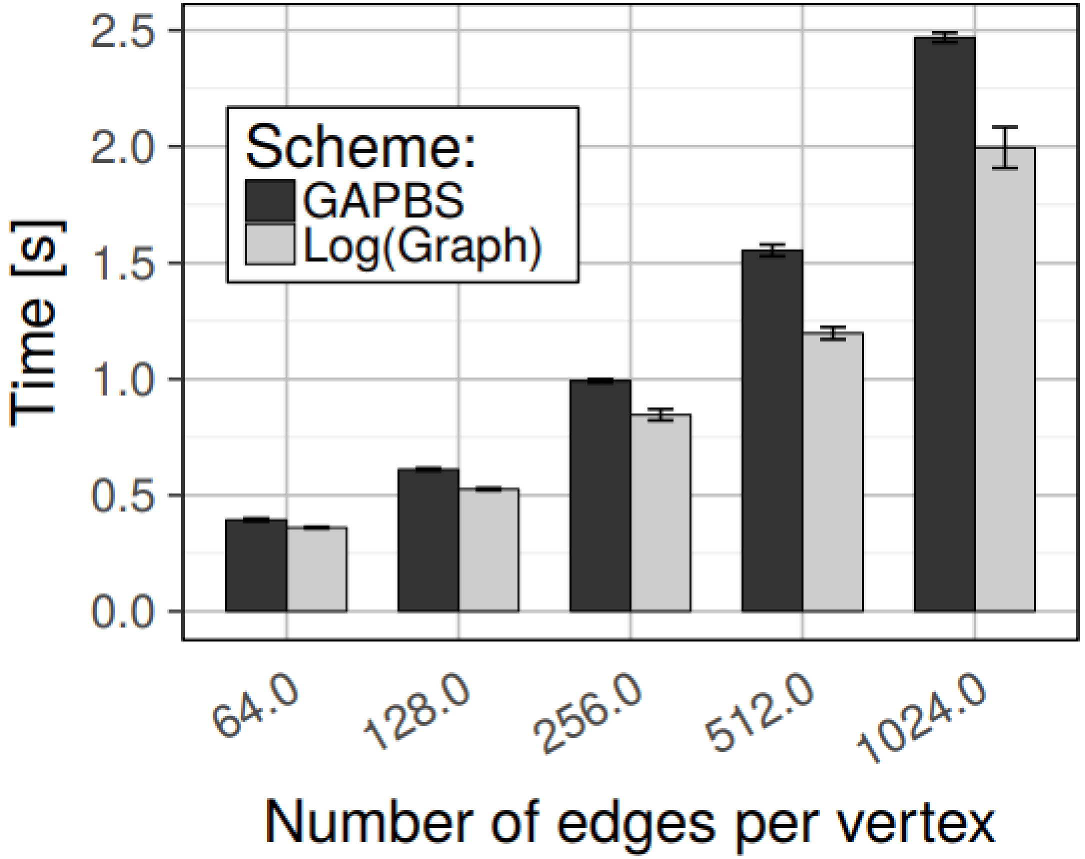
Kronecker graphs
Number of vertices: 4M

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

SSSP



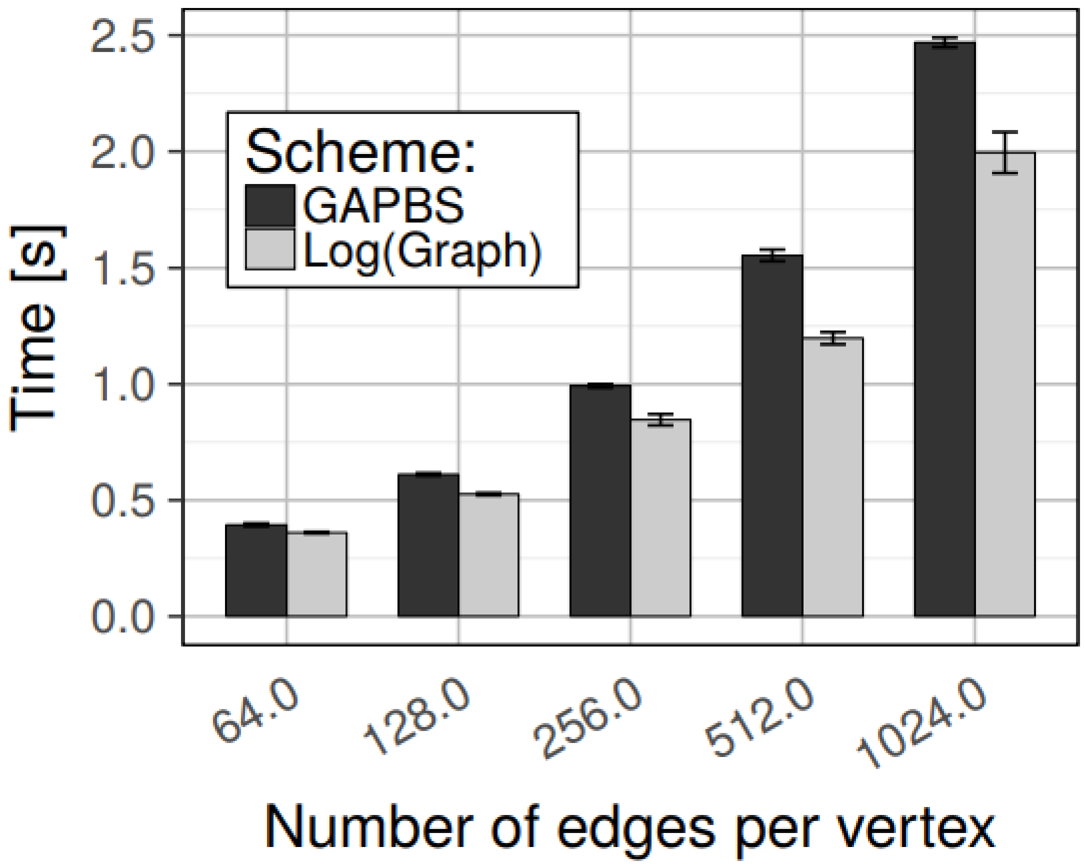
Kronecker graphs
Number of vertices: 4M



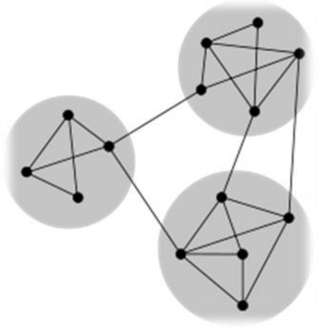
Log(Graph) consistently reduces storage overhead (by 20-35%)

SSSP

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



Log(Graph) **accelerates** GAPBS

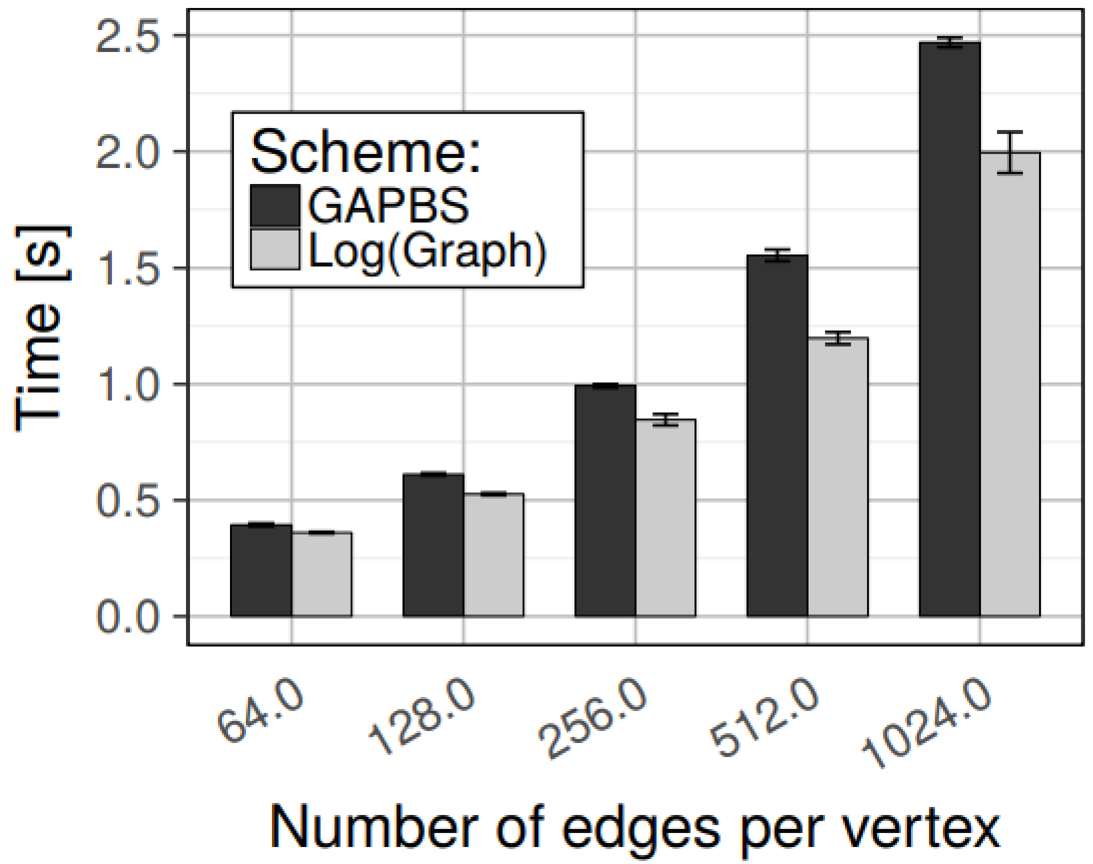


Kronecker graphs
Number of vertices: 4M

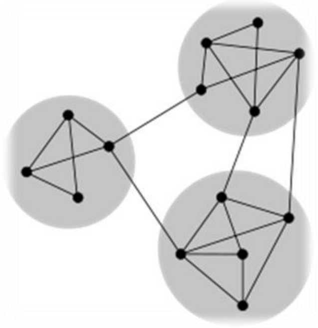
Log(Graph) consistently reduces storage overhead (by 20-35%)

SSSP

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



Log(Graph) **accelerates** GAPBS



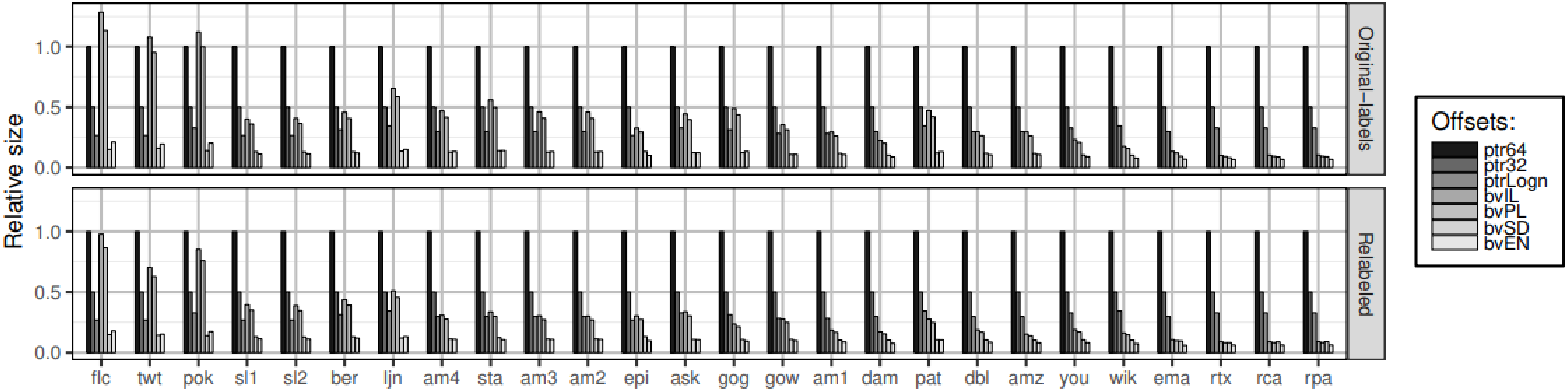
Kronecker graphs
Number of vertices: 4M

Both storage and performance are improved **simultaneously**

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

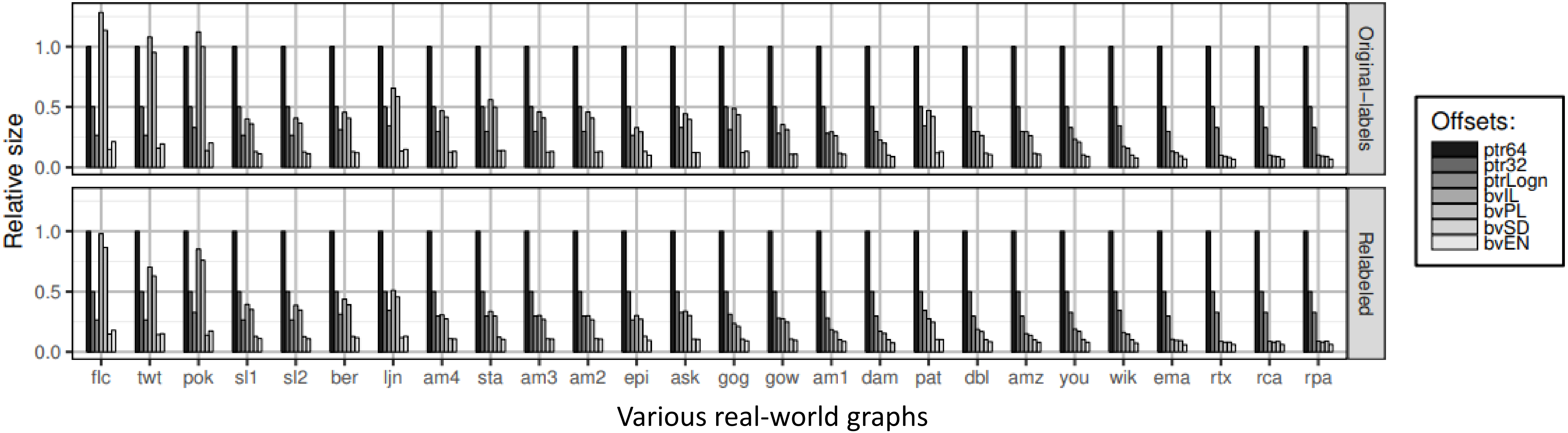
2 **Log (Offset structure) Storage**

2 Log (Offset structure) Storage



Various real-world graphs

2 Log (Offset structure) Storage



Lots of data ☺

Conclusions:

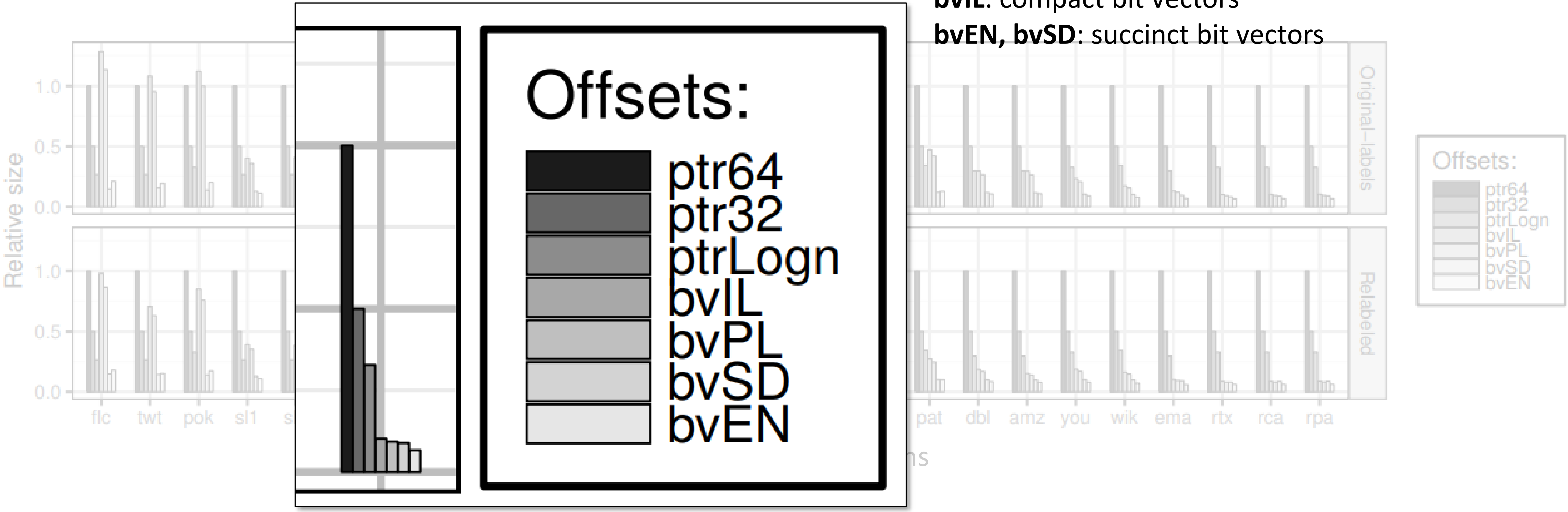
2 Log (Offset structure) Storage

Lots of data ☺

Conclusions:

2 **Log (** Offset structure **)** **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

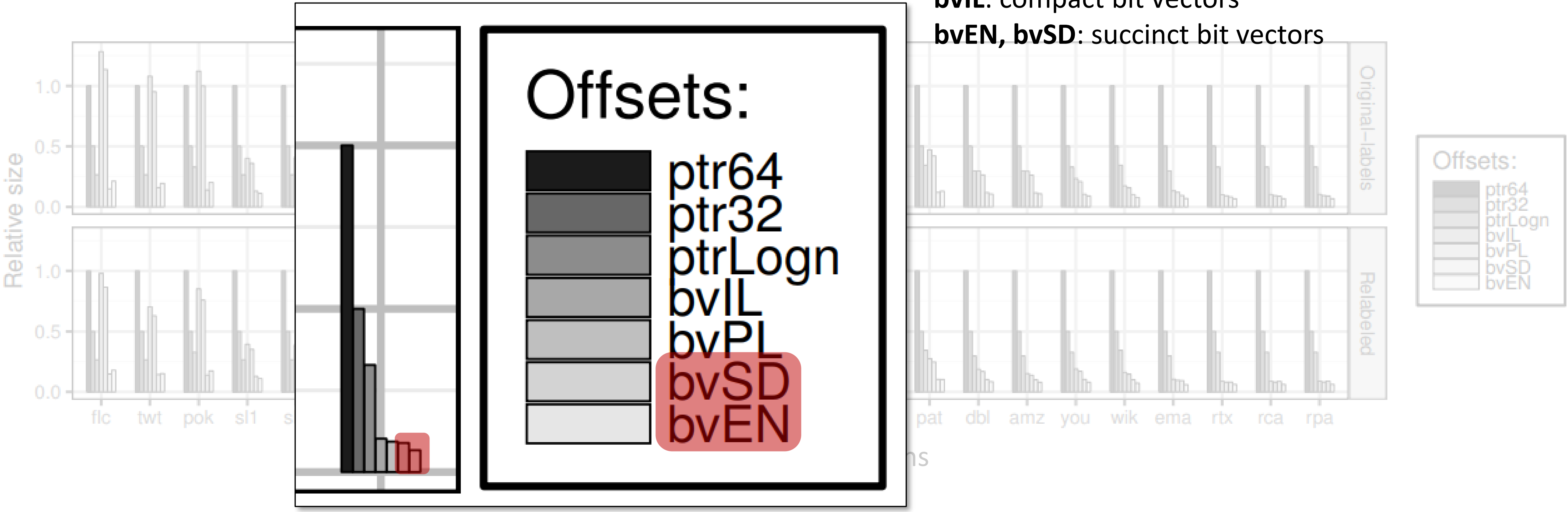


Lots of data ☺

Conclusions:

2 Log (Offset structure) 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

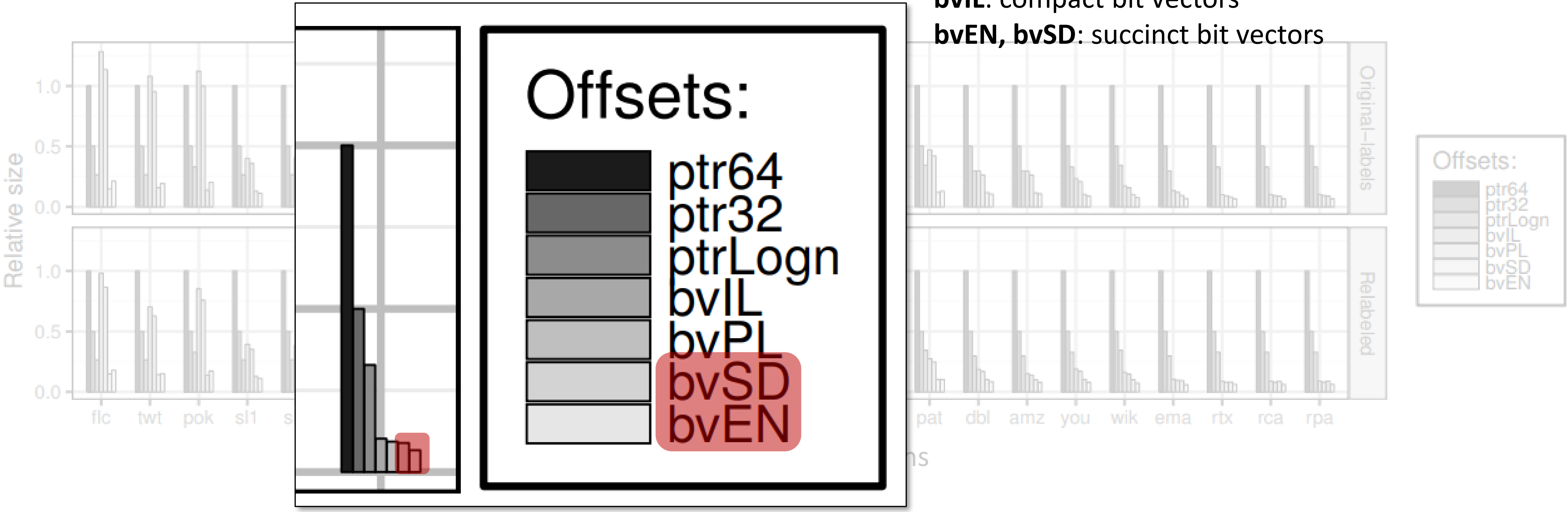


Lots of data ☺

Conclusions:

2 Log (Offset structure) 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



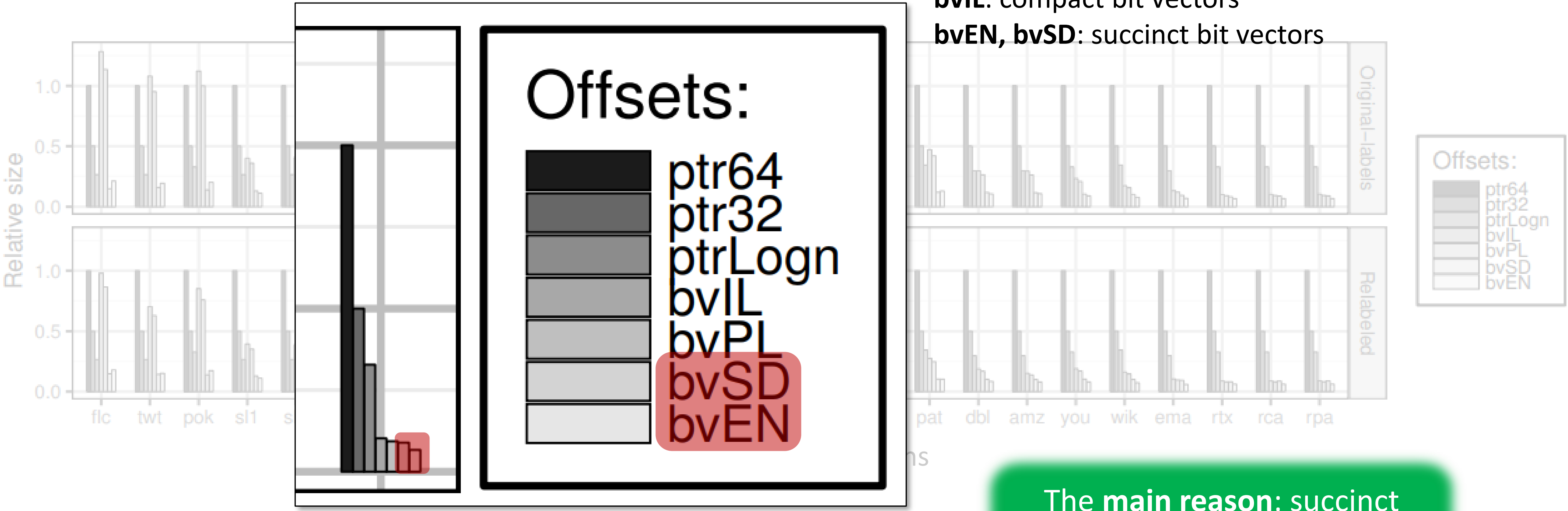
Lots of data ☺

Conclusions:

Succinct bit vectors consistently ensure best storage reductions

2 Log (Offset structure) 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 ☺

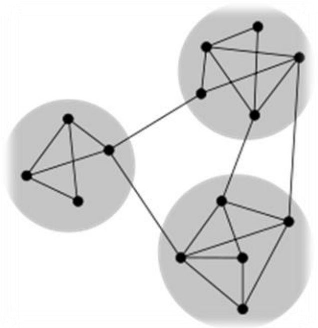
Conclusions:

Succinct bit vectors consistently ensure best storage reductions

The **main reason**: succinct designs work well for sparse bit vectors, and graphs „that matter" are sparse

2 Log (Offset structure) Performance

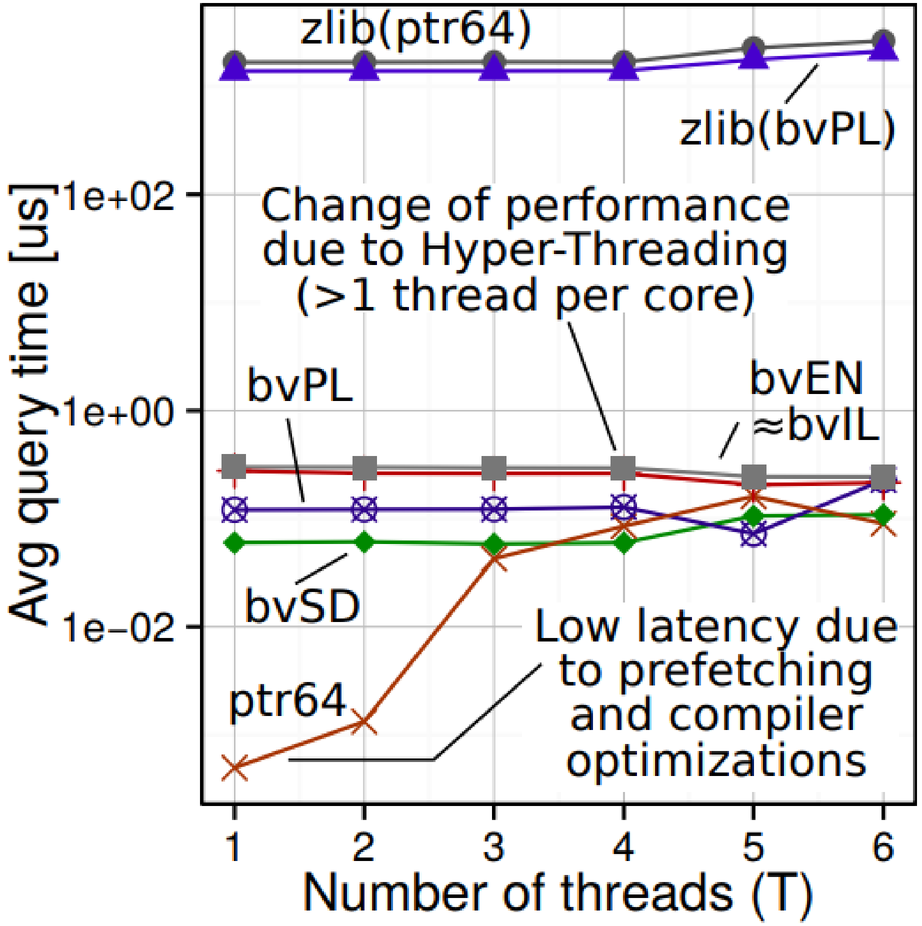
Accessing randomly selected neighbors

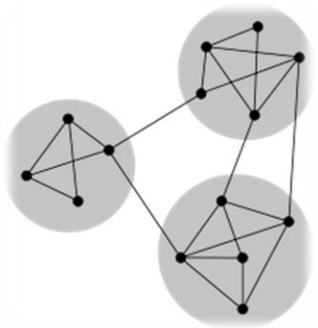


Kronecker graphs
Number of vertices: 4M

2 Log (Offset structure) Performance

Accessing randomly selected neighbors



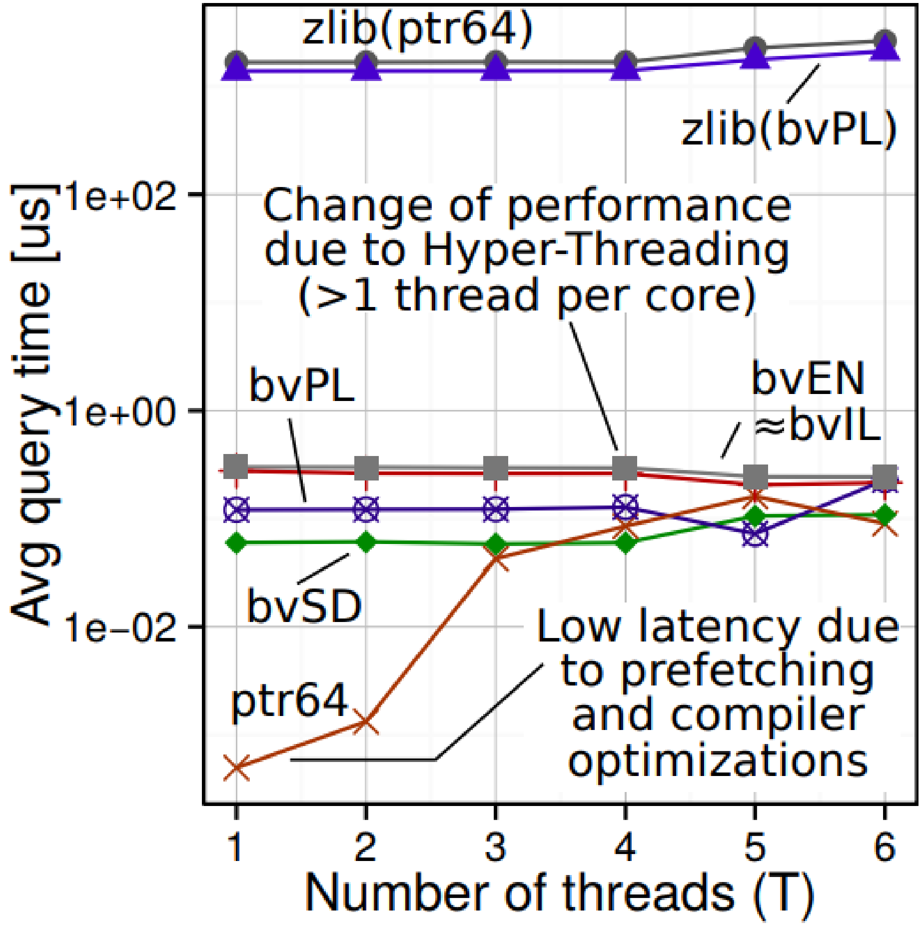


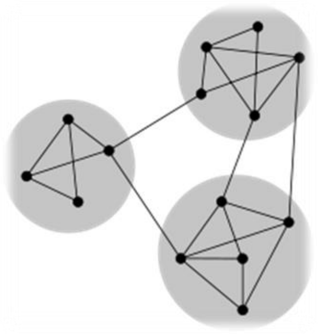
Kronecker graphs
Number of vertices: 4M

2 Log (Offset structure) Performance

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





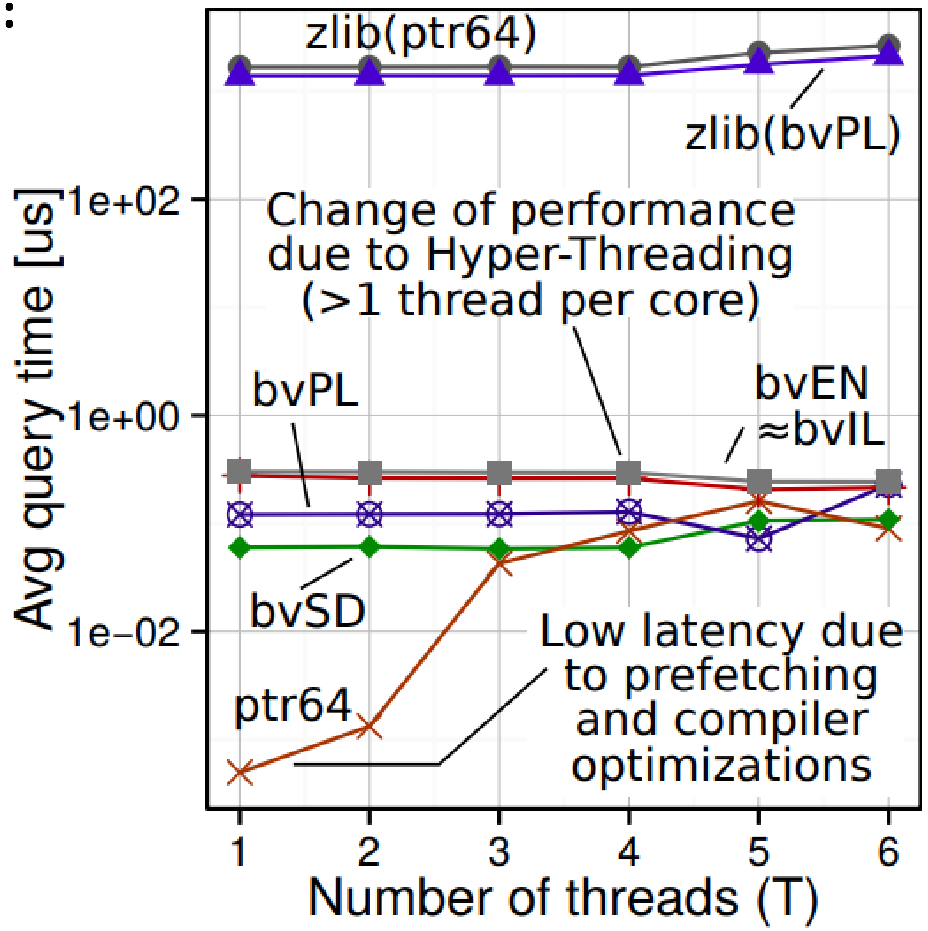
Kronecker graphs
Number of vertices: 4M

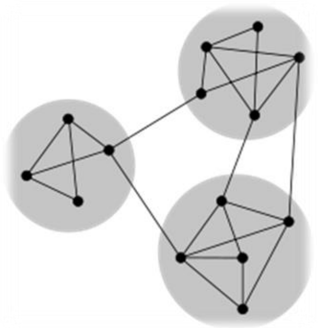
2 Log (Offset structure) Performance

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

Lots of data again ☺ Conclusions:





Kronecker graphs
Number of vertices: 4M

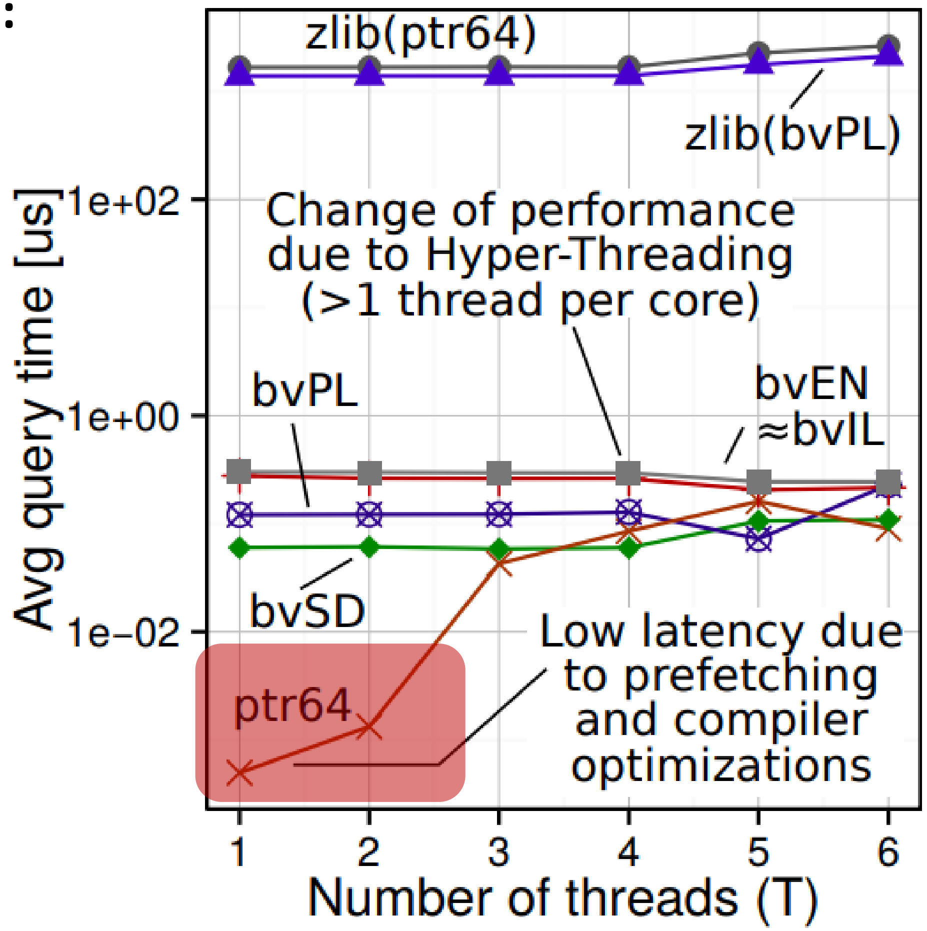
2 Log (Offset structure) Performance

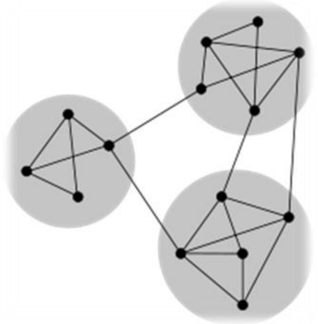
Accessing randomly selected neighbors

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

Lots of data again ☺ Conclusions:

In sequential settings (or settings with low parallelism), **simple offset arrays are the fastest**





Kronecker graphs
Number of vertices: 4M

2 Log (Offset structure) Performance

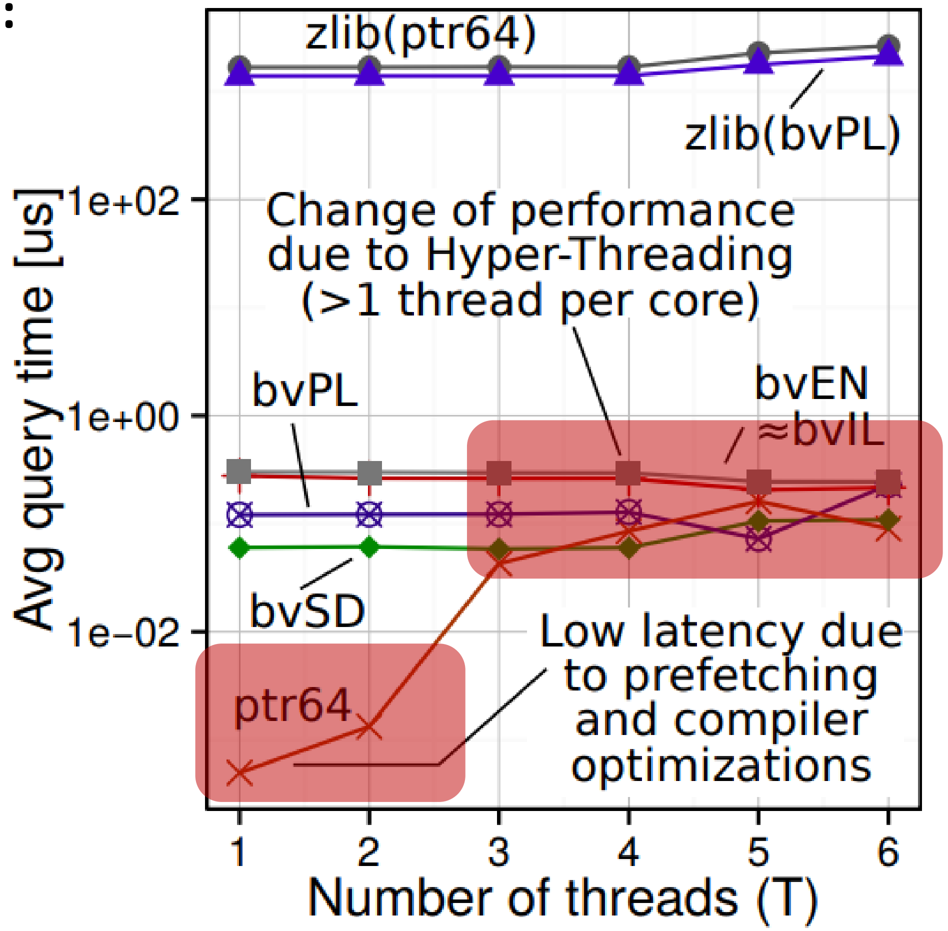
Accessing randomly selected neighbors

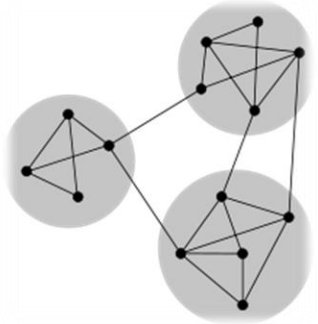
ptr64: traditional array of offsets
bvPL: plain bit vectors
bvIL: compact bit vectors
bvEN, bvSD: succinct bit vectors
zlib(.): zlib-compressed variants

Lots of data again ☺ Conclusions:

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**





Kronecker graphs
Number of vertices: 4M

2 Log (Offset structure) Performance

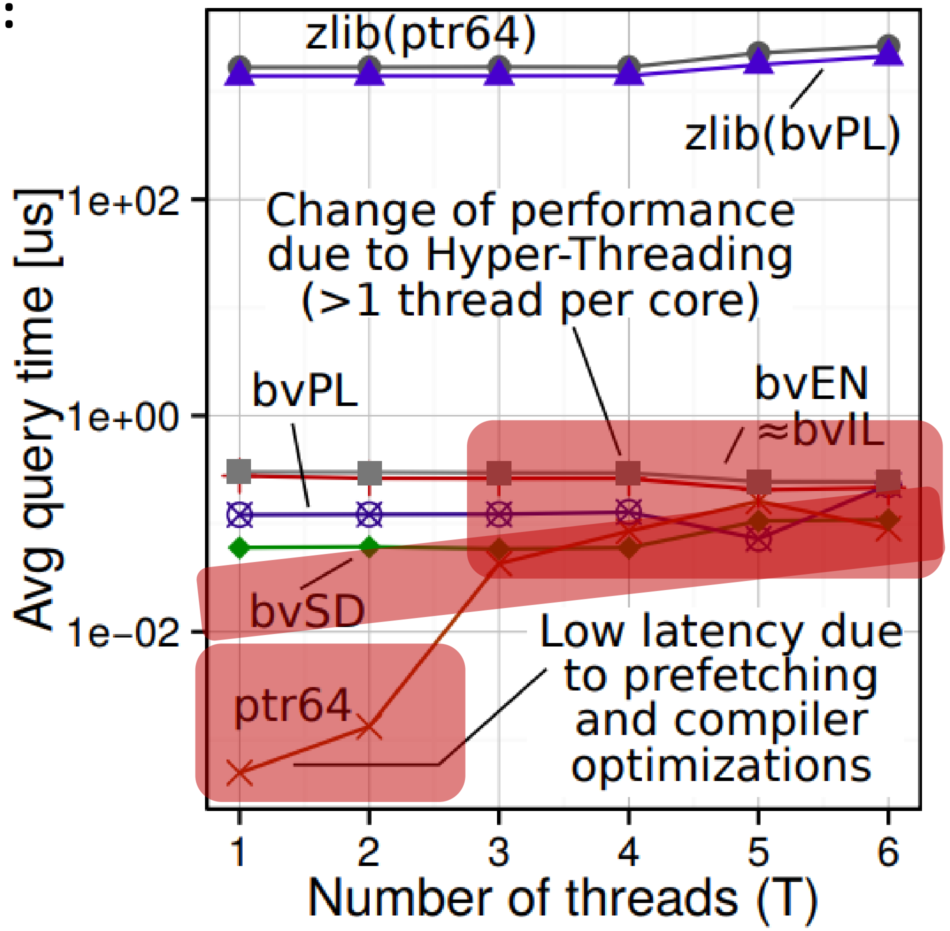
Accessing randomly selected neighbors

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

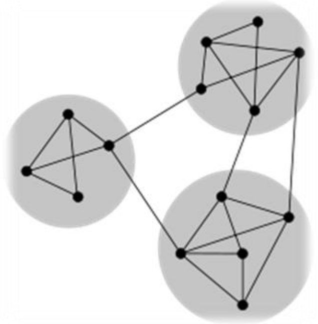
Lots of data again ☺ Conclusions:

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**



bvSD: the fastest and (usually) the smallest

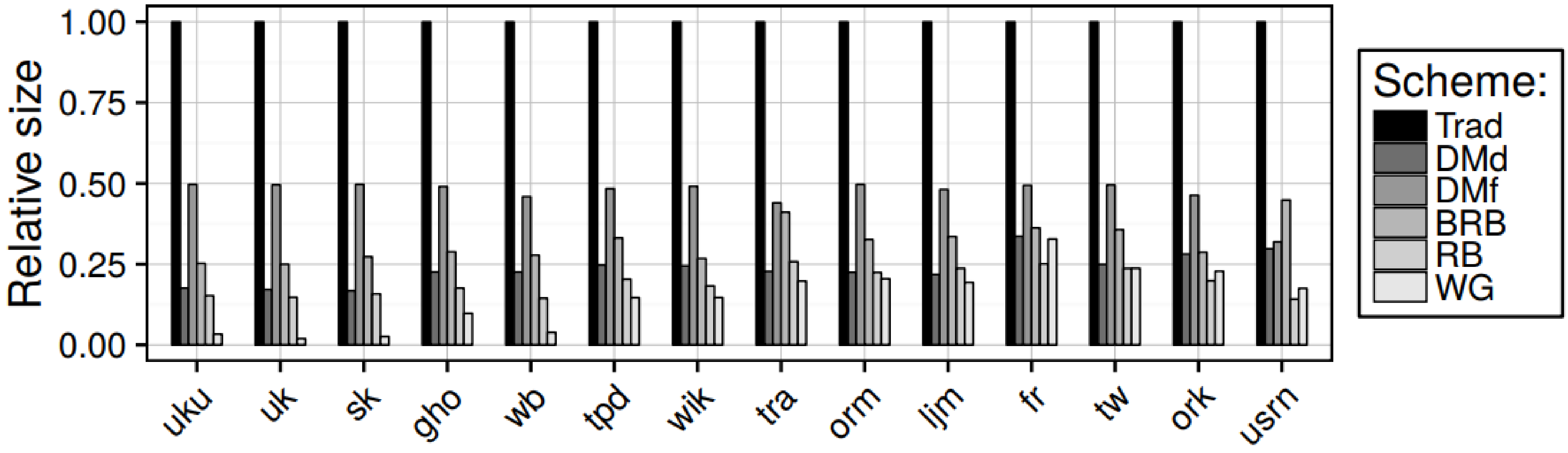


Kronecker graphs
Number of vertices: 4M

3 Log (Adjacency structure) Storage, Performane

3 **Log (** Adjacency structure **)** **Storage, Performane**

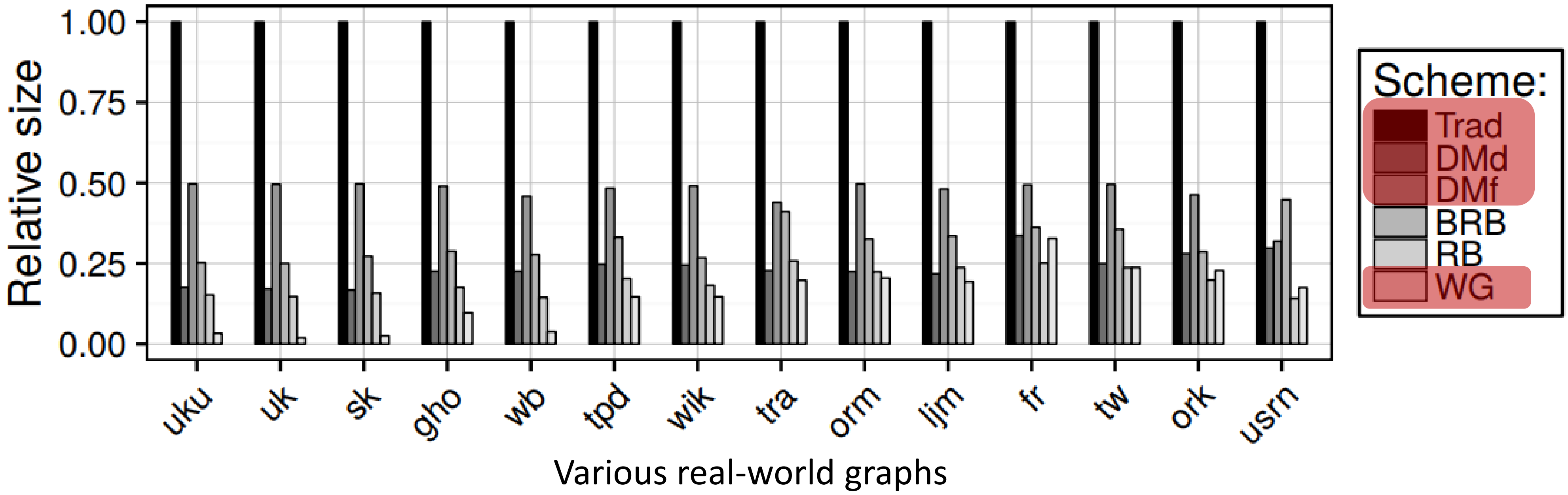
Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs



Various real-world graphs

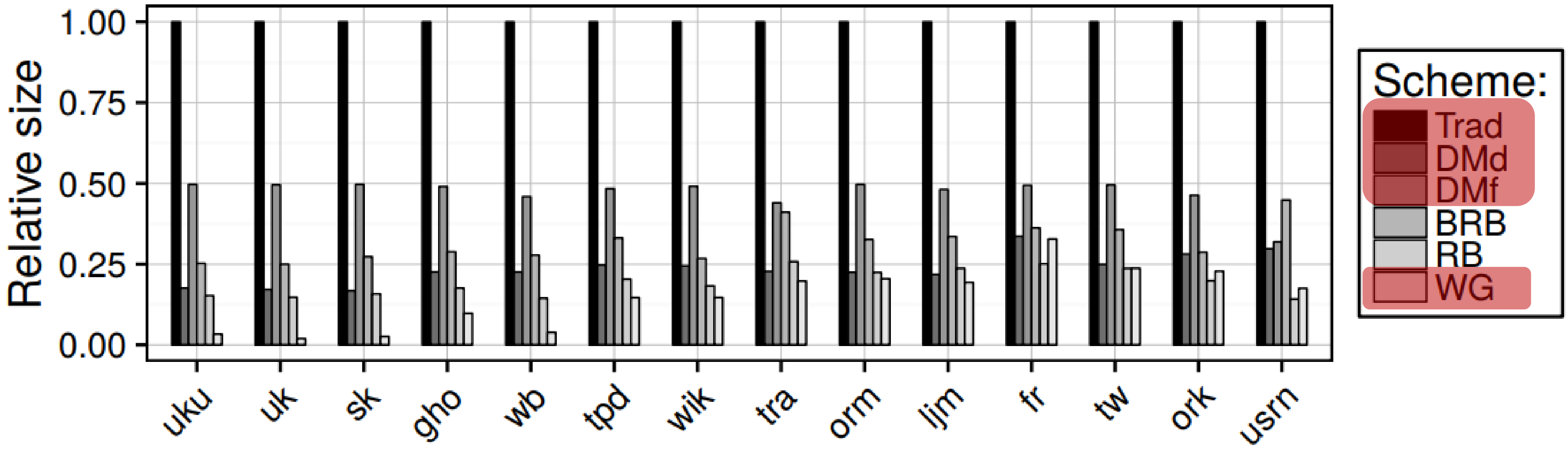
3 Log (Adjacency structure) Storage, Performane

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



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



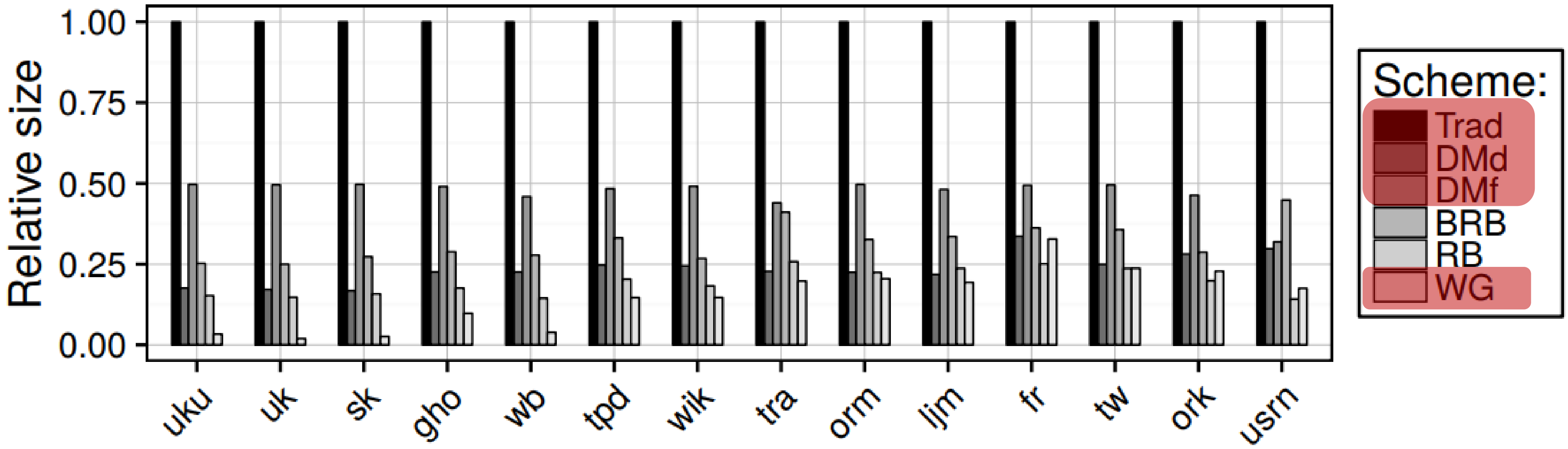
Various real-world graphs

Lots of data ☺

Conclusions:

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



Various real-world graphs

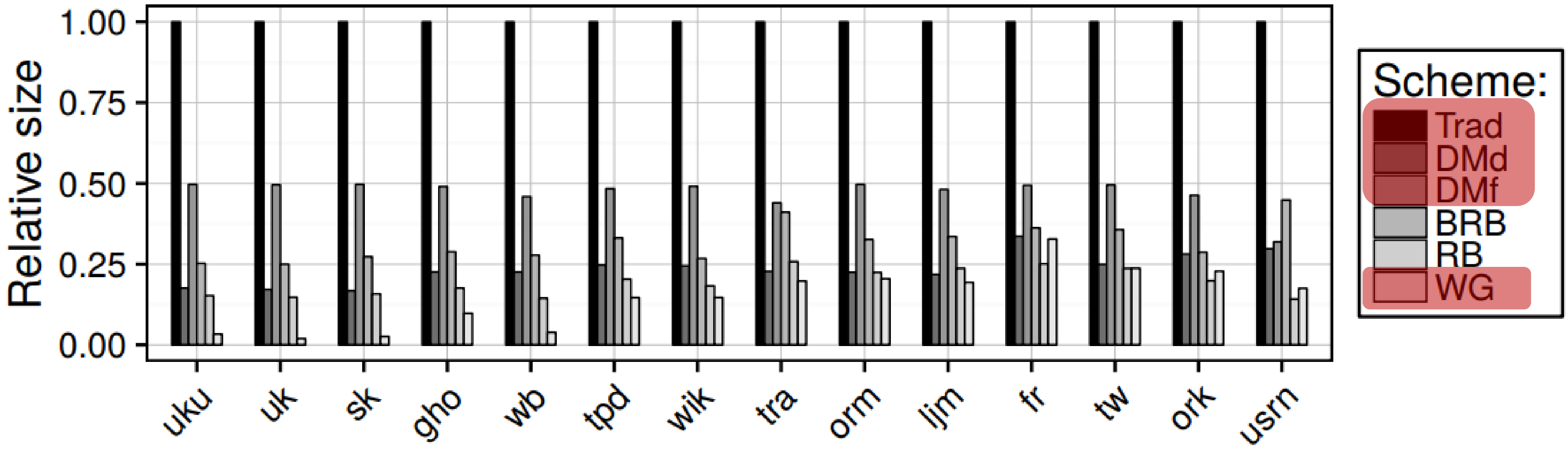
Lots of data ☺

Conclusions:

WebGraph **best for web 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



Various real-world graphs

Lots of data ☺

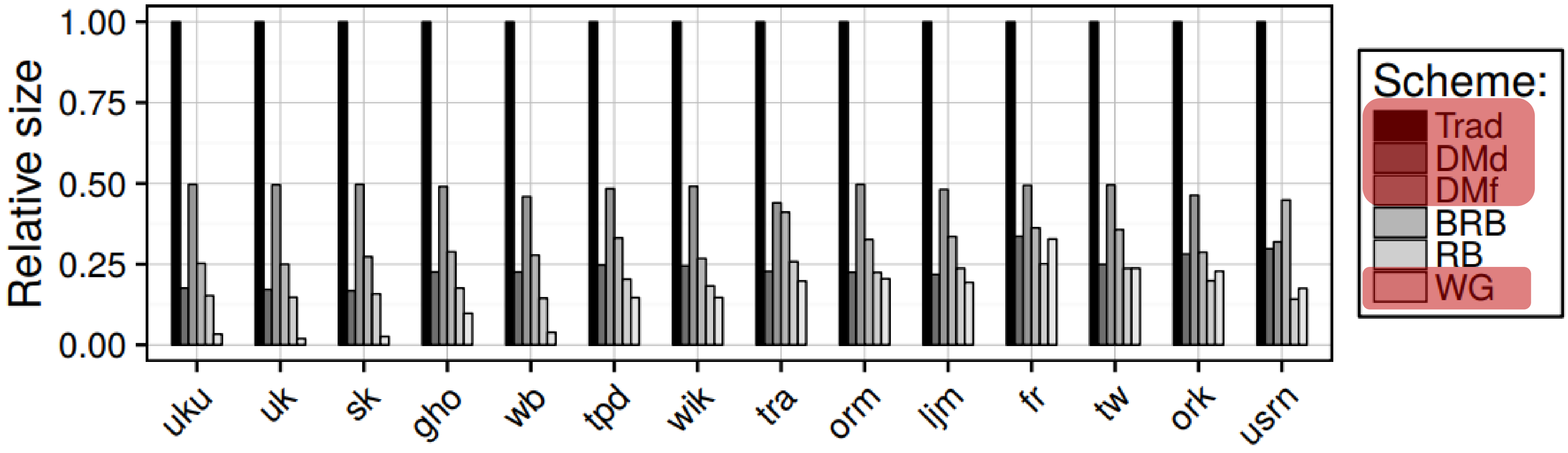
Conclusions:

WebGraph **best for web graphs ☺**

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

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



Various real-world graphs

Lots of data ☺

Conclusions:

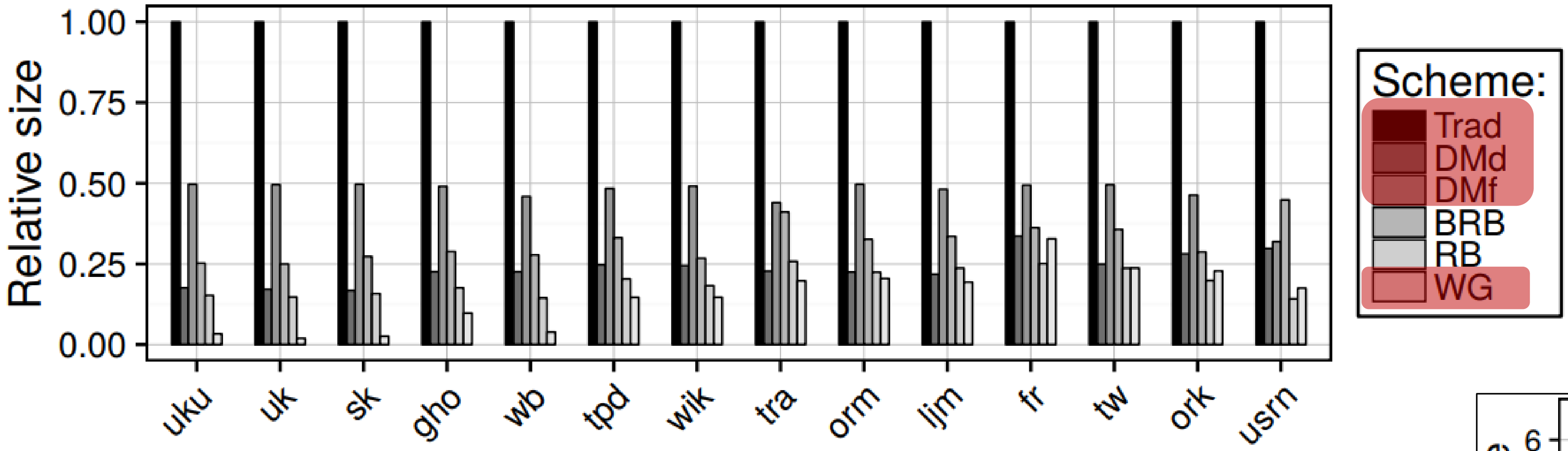
WebGraph **best for web graphs ☺**

DMd: much better than DMf, often comparable to WG

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

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



Various real-world graphs

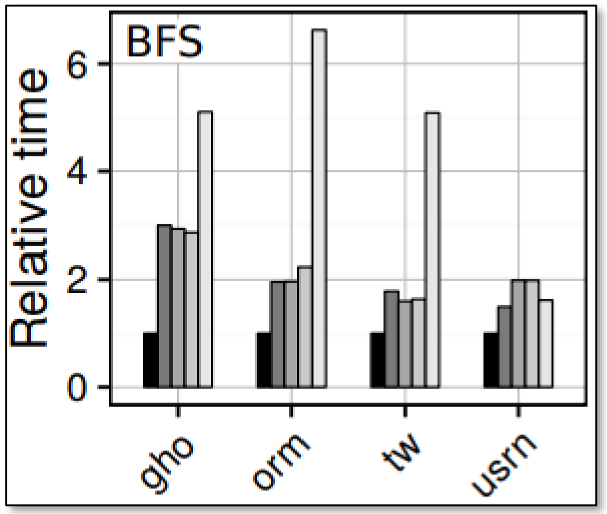
Lots of data ☺

Conclusions:

WebGraph **best for web graphs ☺**

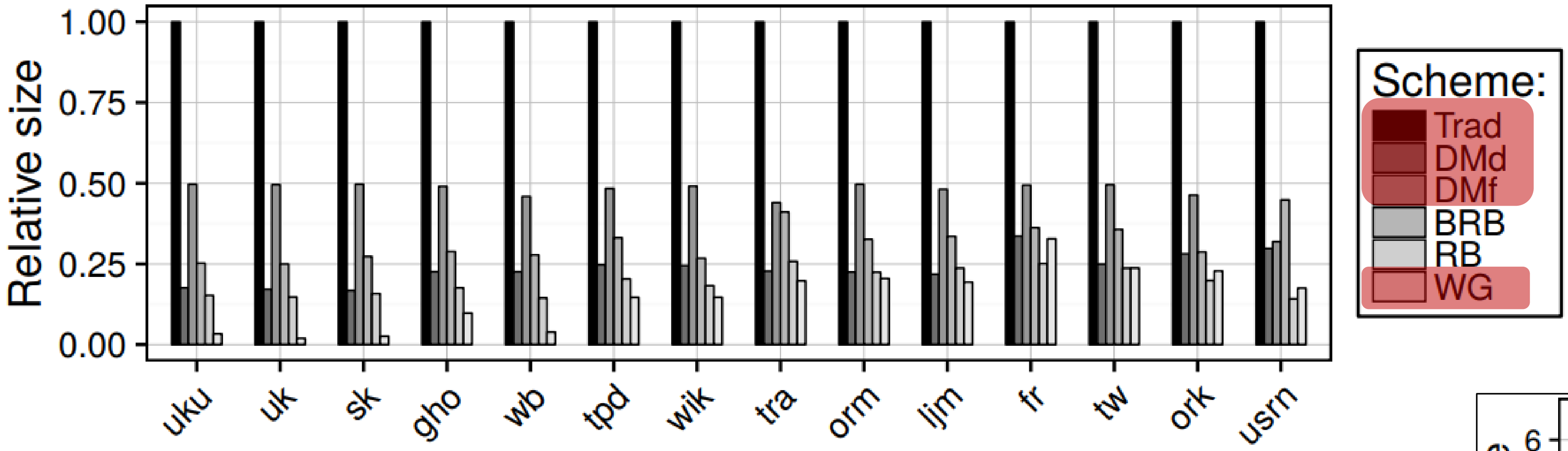
DMd: much better than DMf, often comparable to WG

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



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



Various real-world graphs

WebGraph is the slowest, DM somewhat slower than Trad

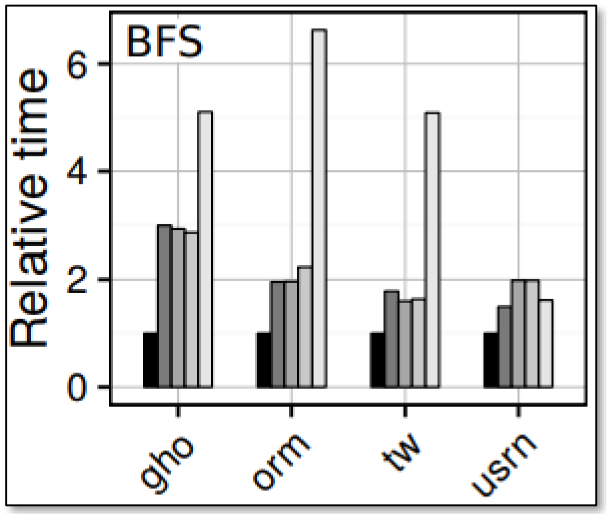
Lots of data ☺

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)



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

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

Key insight (offsets)

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

Key insight (vertex labels)

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

Key insight (adjacency data)

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

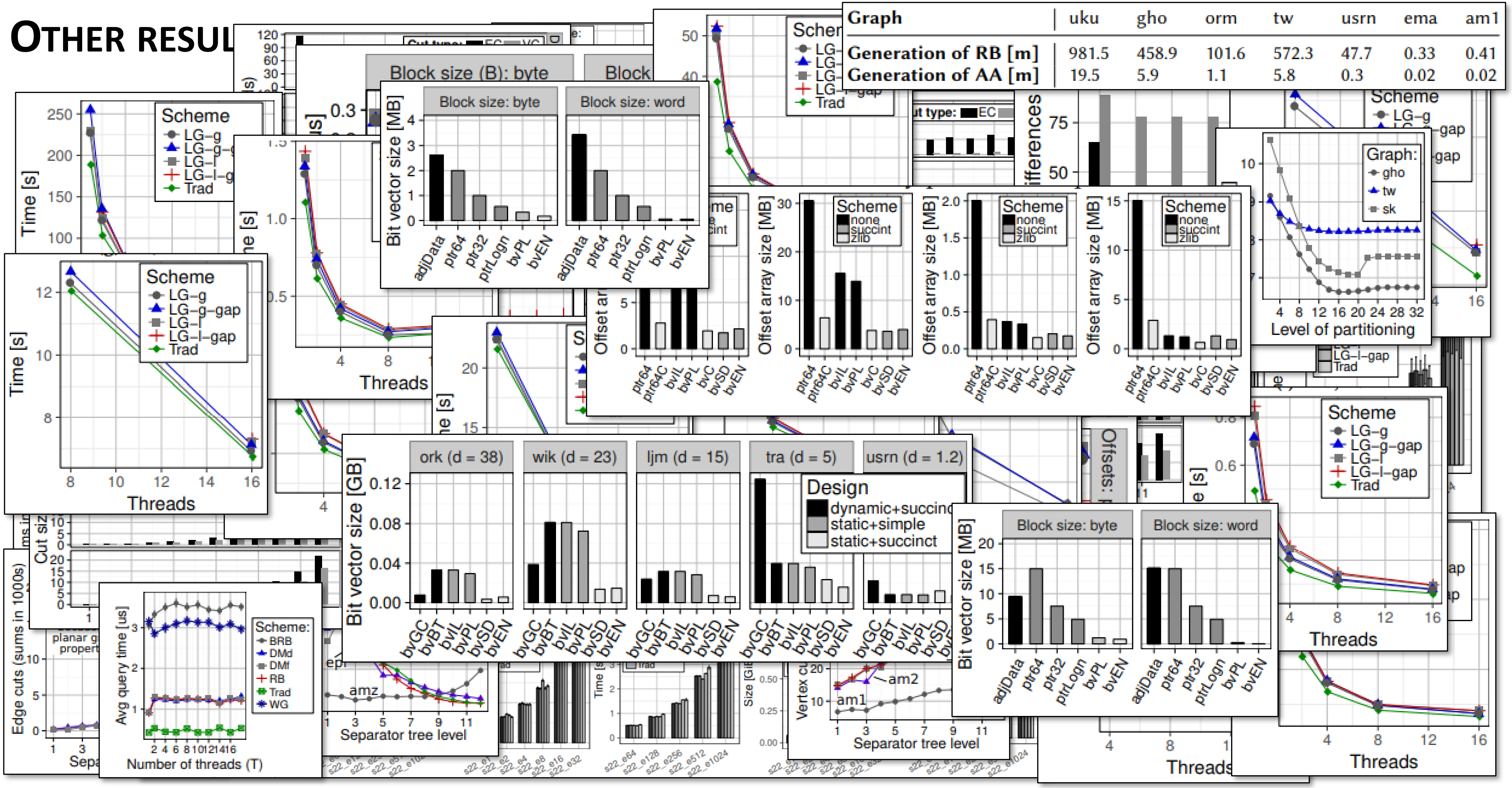
Takeaway (Results): Log(Graph) ensures Space-Performance sweetspot (tunable!)

Key insight (offsets)

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

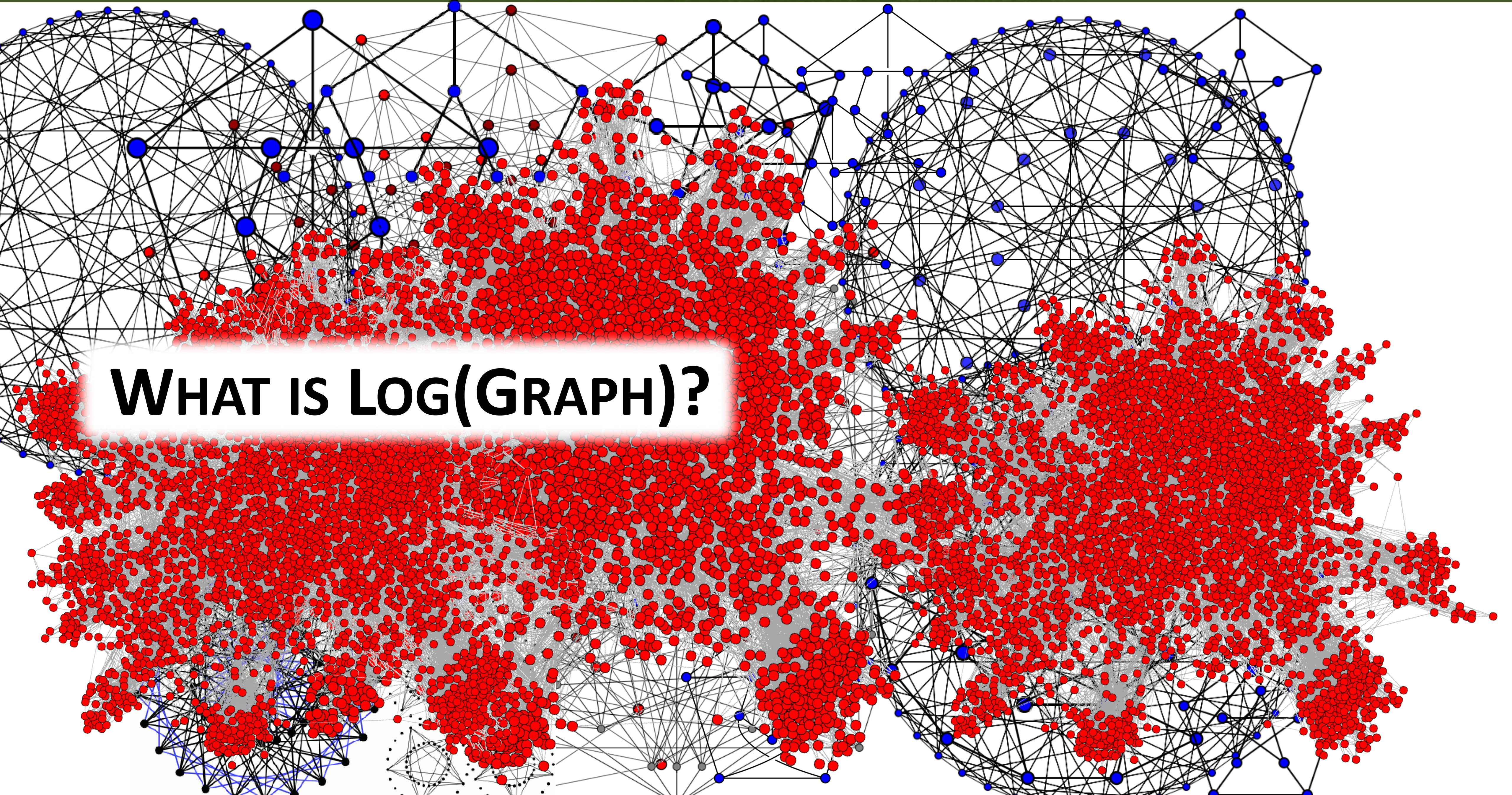
Other results

OTHER RESUL

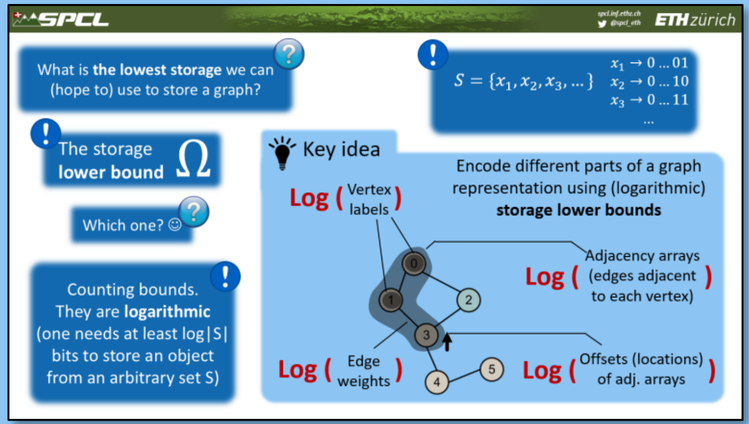


Graph	uku	gho	orm	tw	usrn	ema	am1
Generation of RB [m]	981.5	458.9	101.6	572.3	47.7	0.33	0.41
Generation of AA [m]	19.5	5.9	1.1	5.8	0.3	0.02	0.02

What is Log(Graph)?

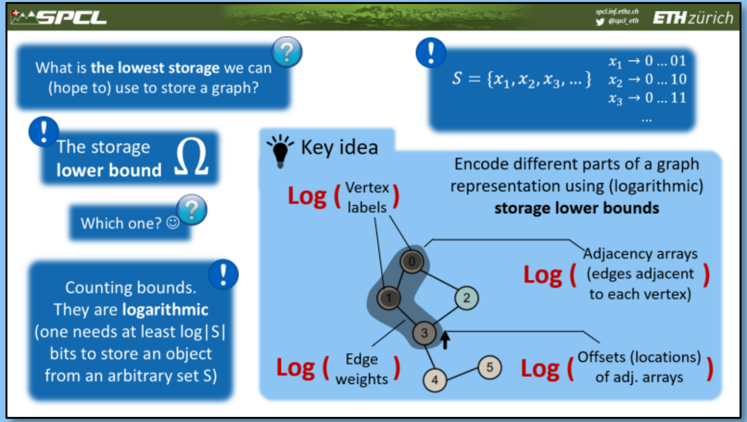


A Near-Optimal Graph Representation

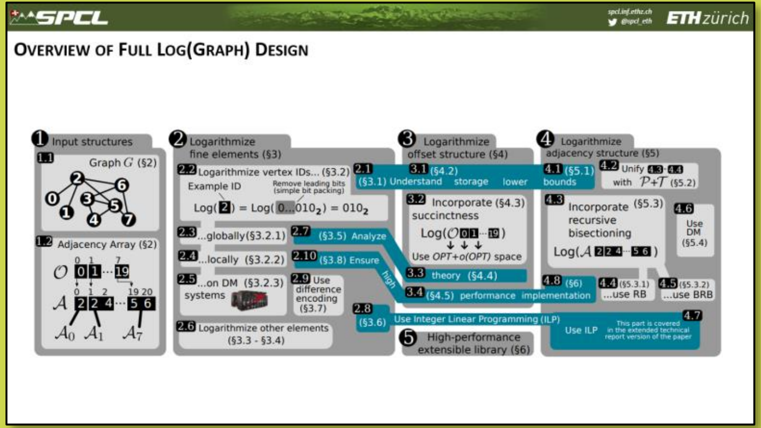


What is Log(Graph)?

A Near-Optimal Graph Representation

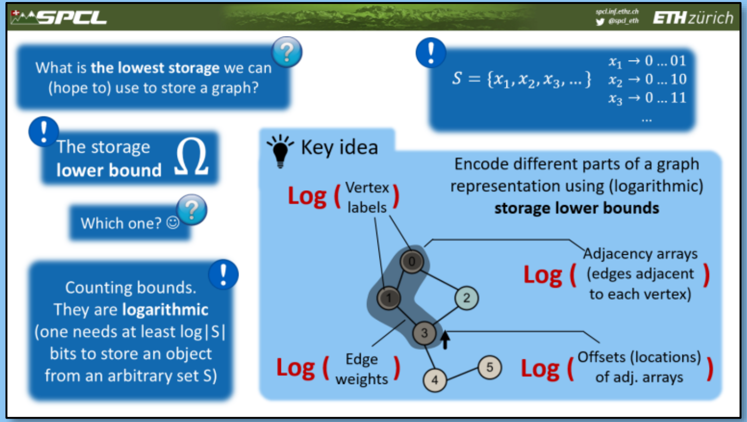


An Extensible Graph Representation

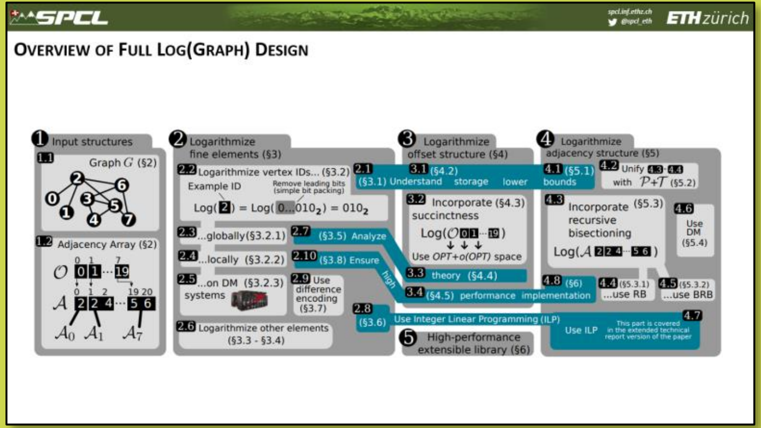


What is Log(Graph)?

A Near-Optimal Graph Representation

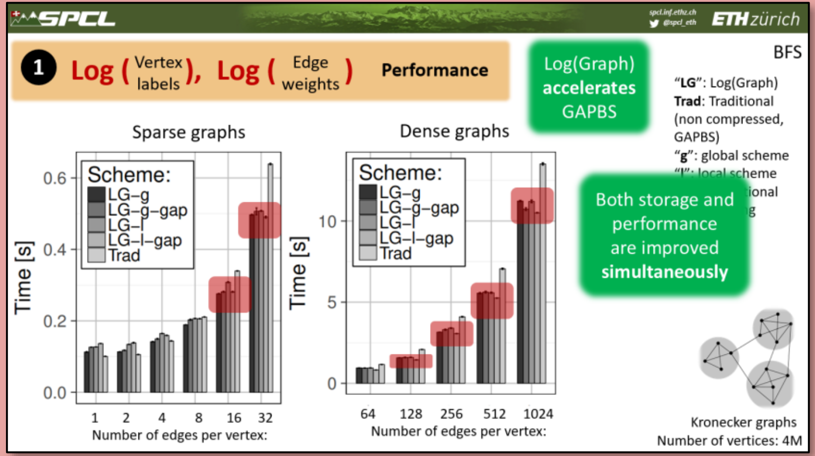


An Extensible Graph Representation

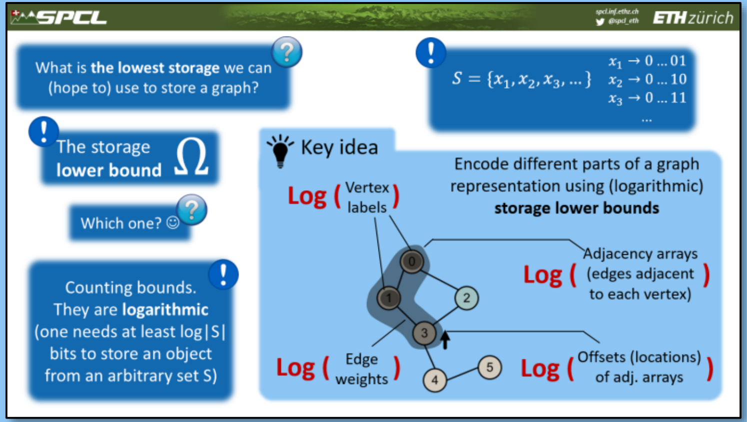


What is Log(Graph)?

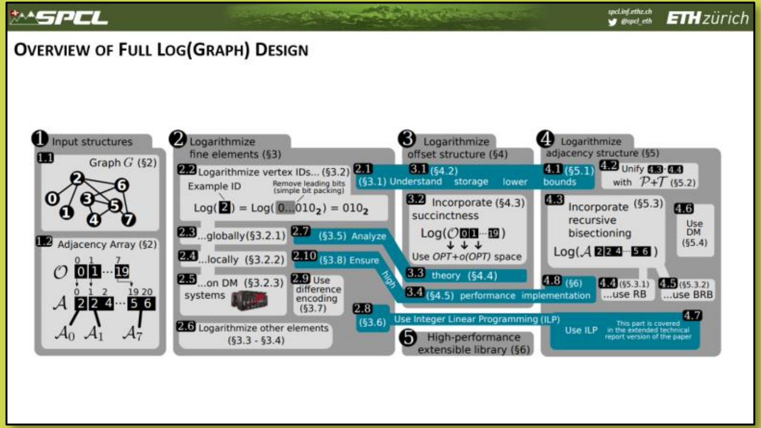
A High-Performance Graph Representation



A Near-Optimal Graph Representation

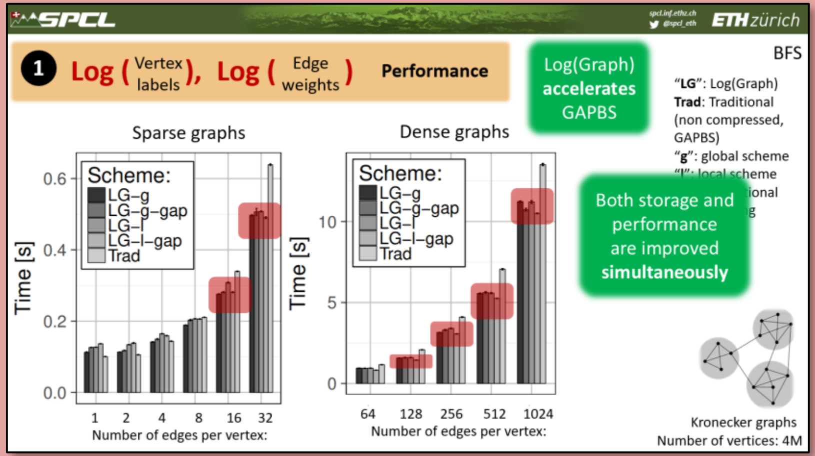


An Extensible Graph Representation

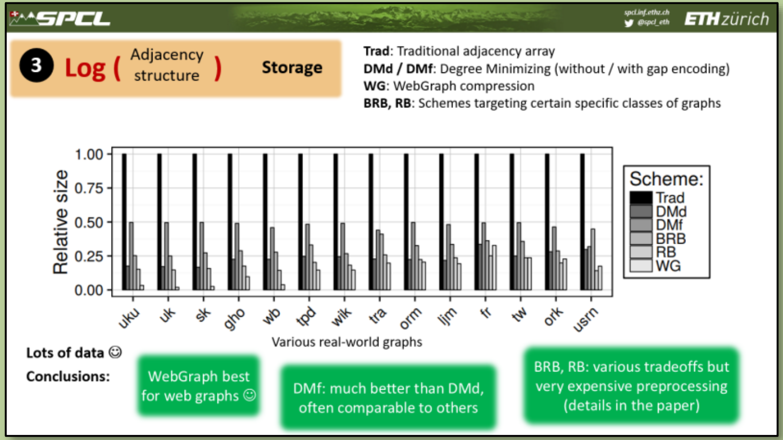


What is Log(Graph)?

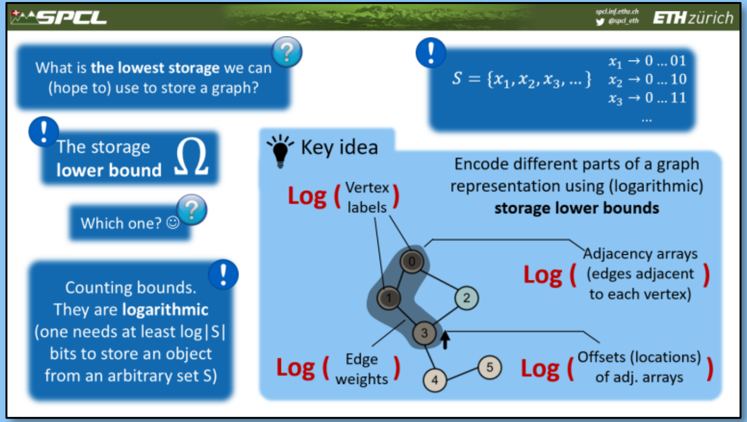
A High-Performance Graph Representation



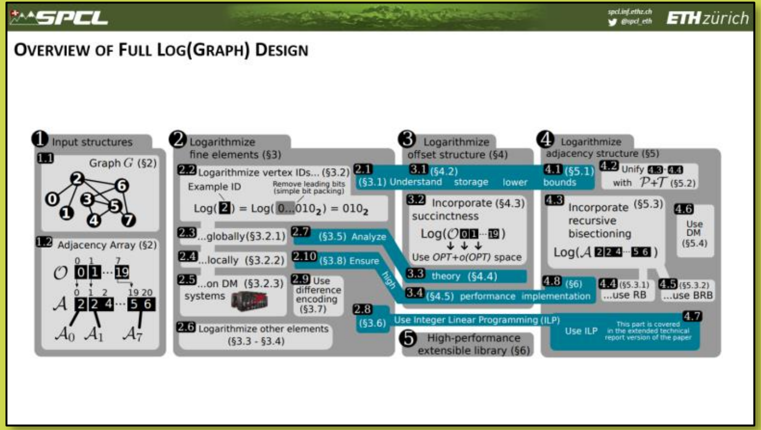
A Condensed Graph Representation



A Near-Optimal Graph Representation

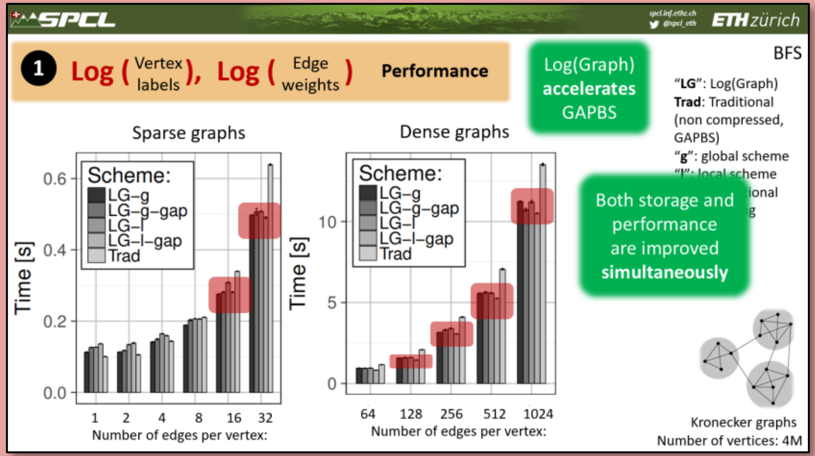


An Extensible Graph Representation

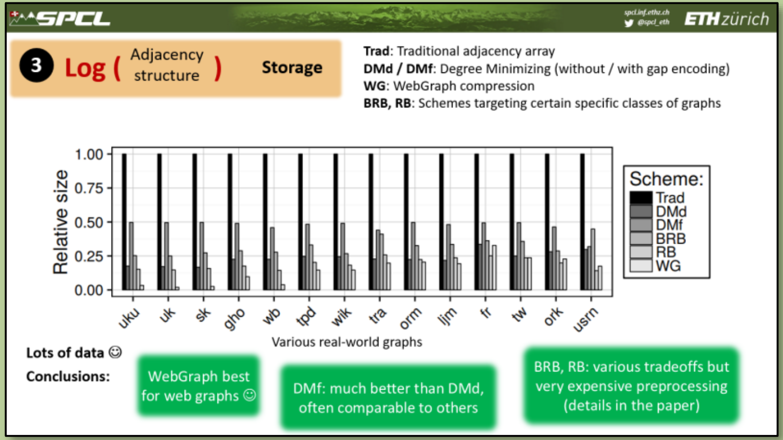


What is Log(Graph)?

A High-Performance Graph Representation



A Condensed Graph Representation

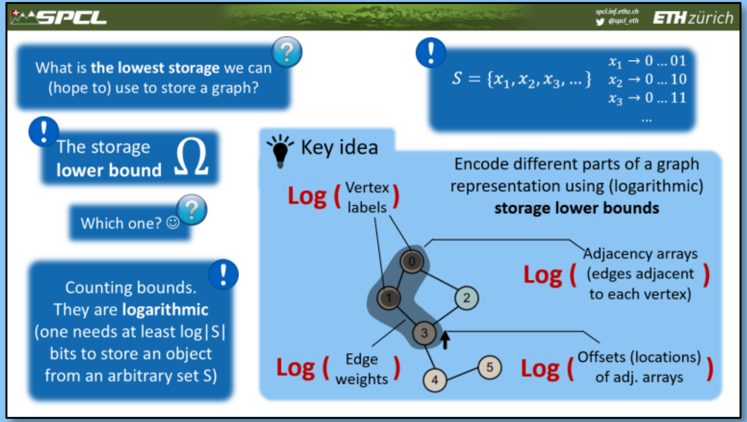


Website:

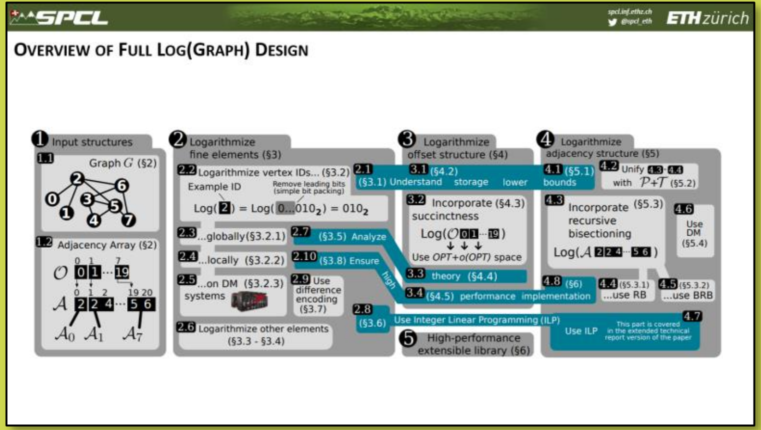
http://spcl.inf.ethz.ch/Research/Performance/LogGraph

What is Log(Graph)?

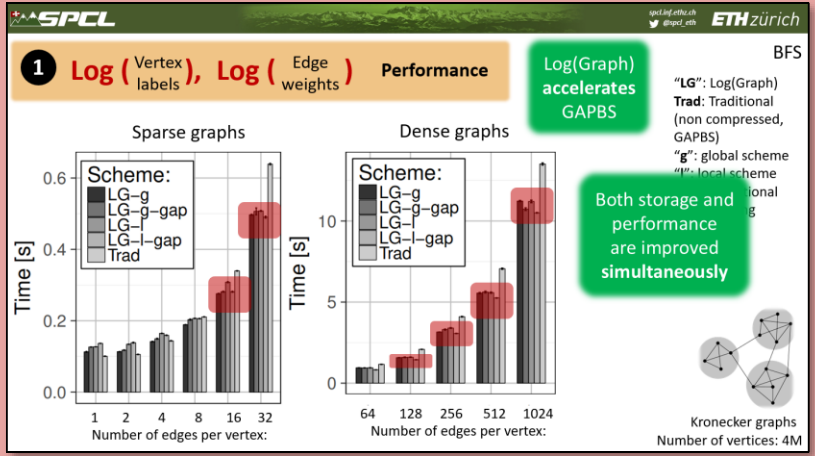
A Near-Optimal Graph Representation



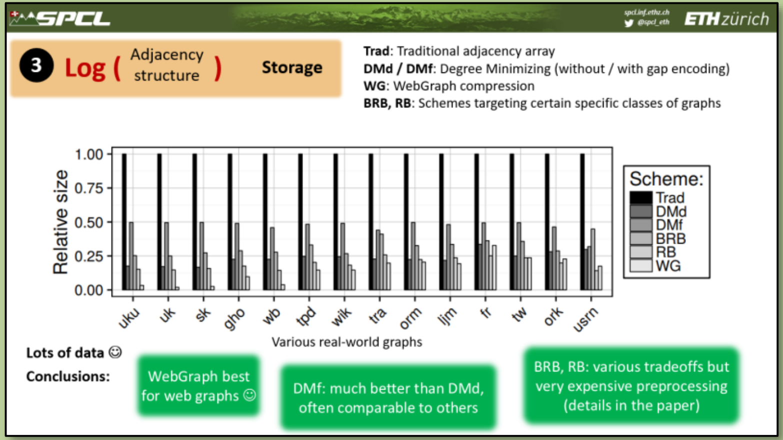
An Extensible Graph Representation



A High-Performance Graph Representation



A Condensed Graph Representation



COMING SOON

Website:

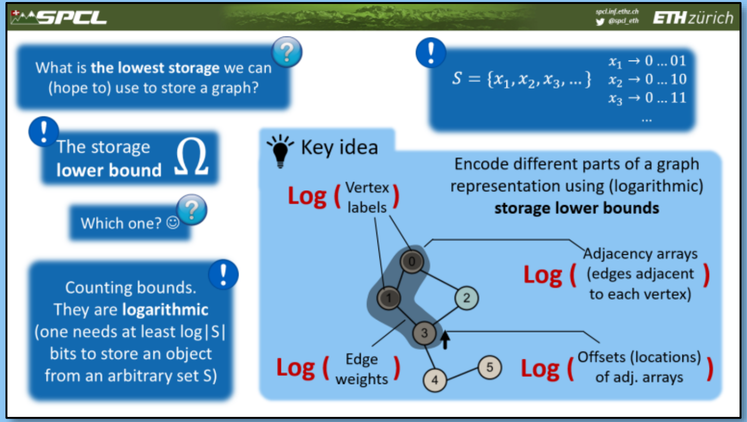
http://spcl.inf.ethz.ch/Research/Performance/LogGraph

SPCL

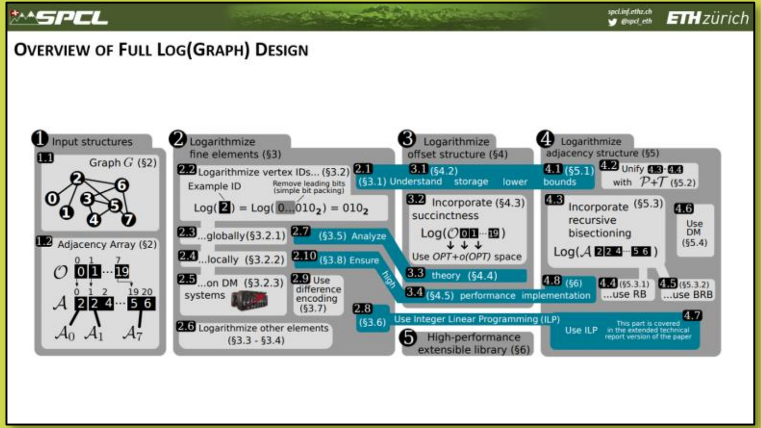
spcl.inf.ethz.ch
@spcl_eth

ETH zürich

A Near-Optimal Graph Representation



An Extensible Graph Representation



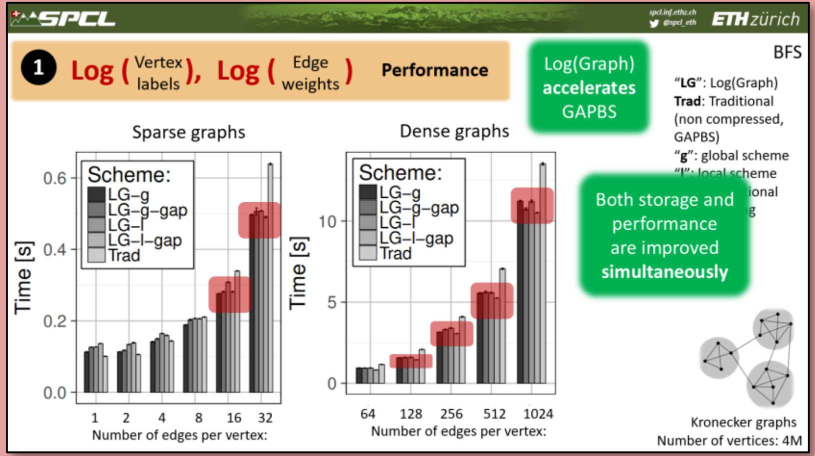
What is Log(Graph)?

Thank you for your attention

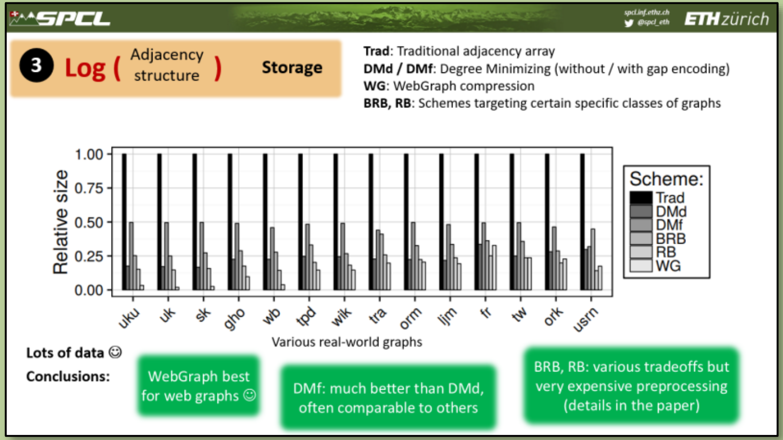
Website: COMING SOON

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

A High-Performance Graph Representation

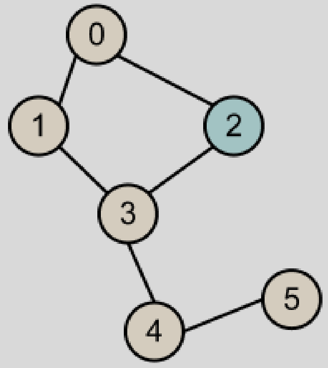


A Condensed Graph Representation

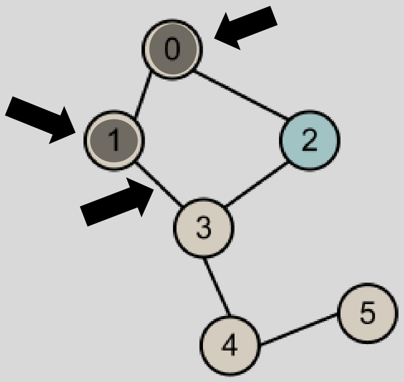


1 **Log (** Vertex labels **), Log (** Edge weights **)**

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



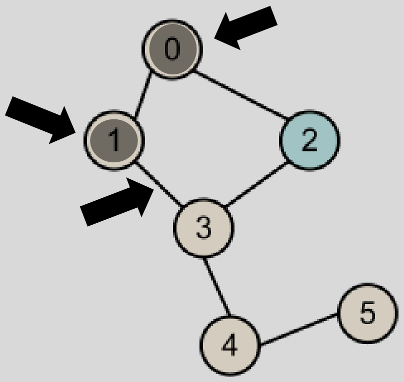
1 **Log (** Vertex labels **), Log (** Edge weights **)**



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

Symbols

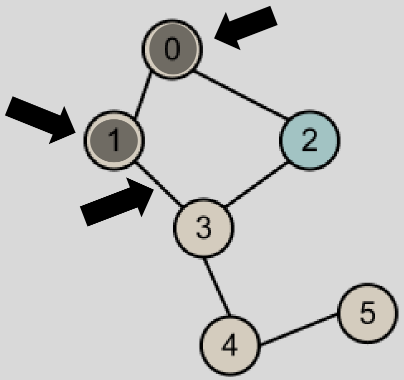
$\widehat{W}$: max edge weight,
n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v



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

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 (global)

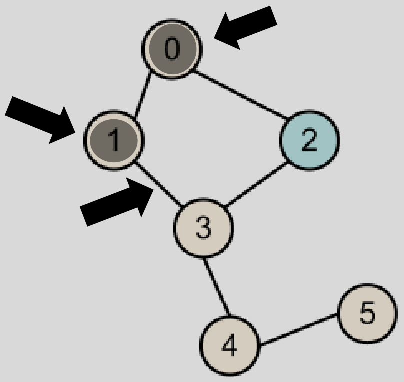
1 **Log (Vertex labels), Log (Edge weights)**

Lower bounds (global)

$\lceil \log n \rceil$

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



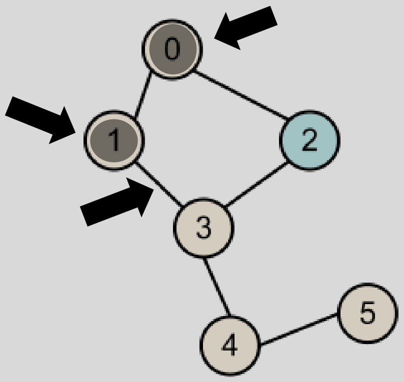
1 **Log (** Vertex labels **), Log (** Edge weights **)**

Lower bounds (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

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



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

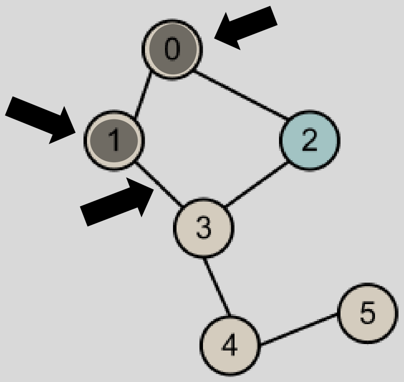
Lower bounds (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Symbols

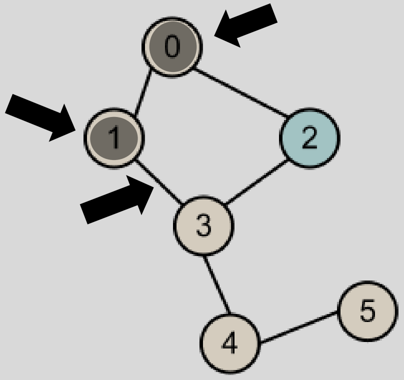
$\widehat{W}$: max edge weight,
n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v



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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

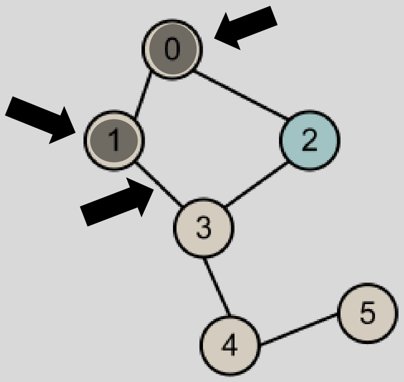
This is it?
Not really ☺

Lower bounds (local)

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

Symbols

$\hat{W}$: 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 (global)

$\lceil \log n \rceil$ $\lceil \log \hat{W} \rceil$

This is it?
Not really ☺

Lower bounds (local)

Assume:

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

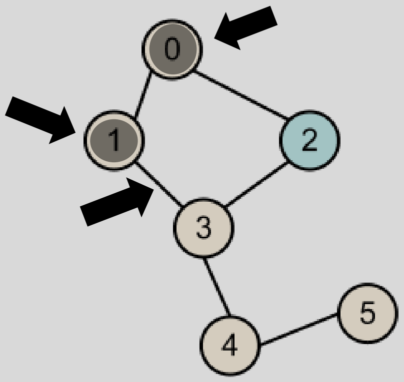
Lower bounds (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

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)

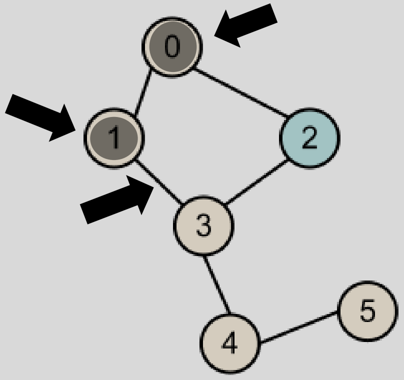
Assume:

- a graph, e.g., $V = \{1, \ldots, 2^{22}\}$

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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

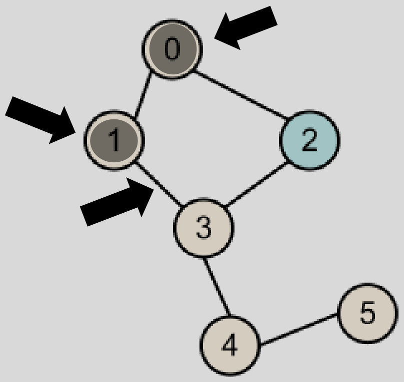
Lower bounds (local)

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

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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Lower bounds (local)

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

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

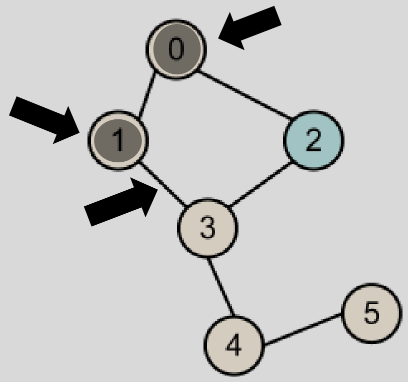
Lower bounds (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Symbols

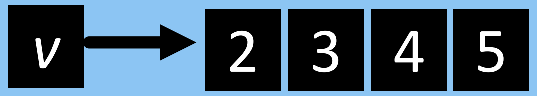
$\widehat{W}$: max edge weight,
n : #vertices,
m : #edges,
d_v : degree of vertex v,
N_v : neighbors (adj. array) of vertex v,
$\widehat{N_v}$: maximum among N_v



Lower bounds (local)

Assume:

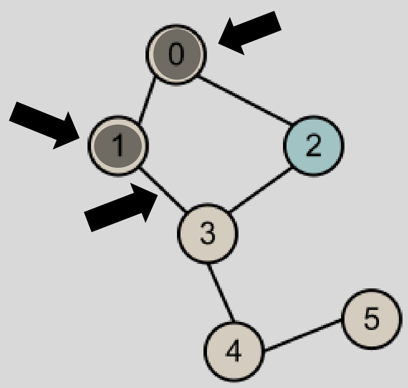
- a graph, e.g., $V = \{1, \ldots, 2^{22}\}$
- A vertex v with few neighbors: $d_v \ll n$
- ...all these neighbors have small labels: $\widehat{N_v} \ll n$



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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Lower bounds (local)

Assume:

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

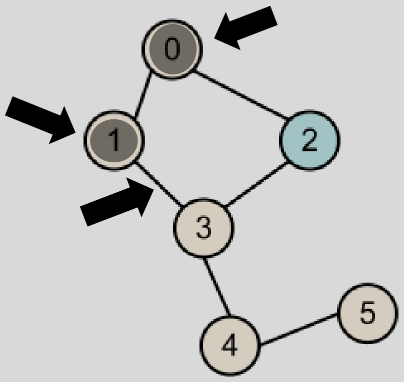
$$\lceil \log 2^{22} \rceil = 22$$

v → | 2 | 3 | 4 | 5 |

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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Lower bounds (local)

Assume:

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

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

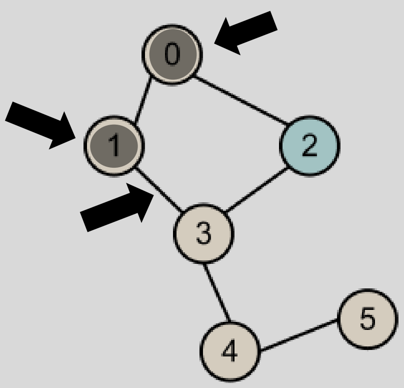
v → | 0...10 | 0...11 | 0...100 | 0...101 |

v → | 2 | 3 | 4 | 5 |

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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

This is it?
Not really ☺

Lower bounds (local)

Assume:

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

v → 2 | 3 | 4 | 5

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

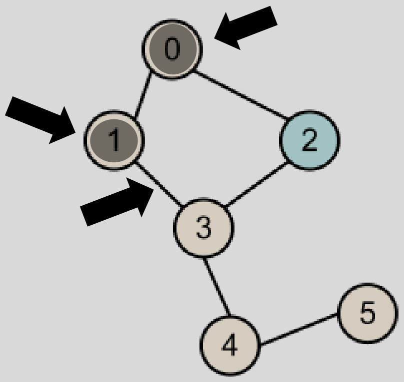
v → 0...10 | 0...11 | 0...100 | 0...101

19 zeros!

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

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 (global)

$\lceil \log n \rceil$ $\lceil \log \widehat{W} \rceil$

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: $\widehat{N_v} \ll n$

v → 2 | 3 | 4 | 5

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

v → 0...10 | 0...11 | 0...100 | 0...101

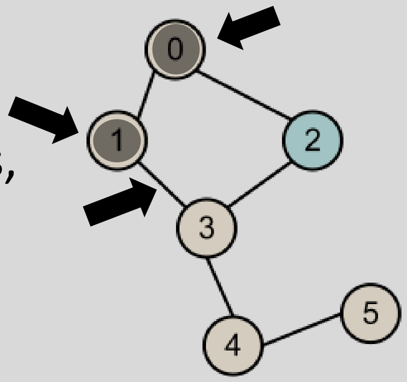
19 zeros!

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

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

This is it? Still not really ☺ ?

Symbols

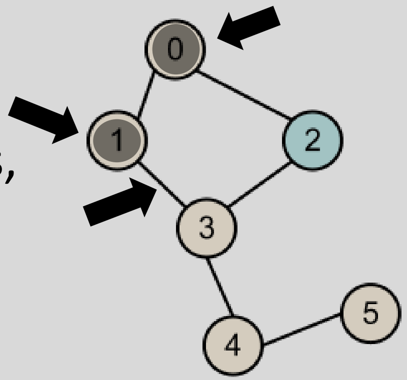
n : #vertices,

m : #edges,

H : number of compute nodes,

H_i : number of machine elements at level i,

N : number of machine levels

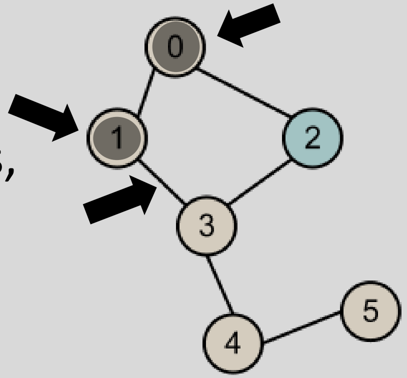


1 **Log (Vertex labels), Log (Edge weights)**

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



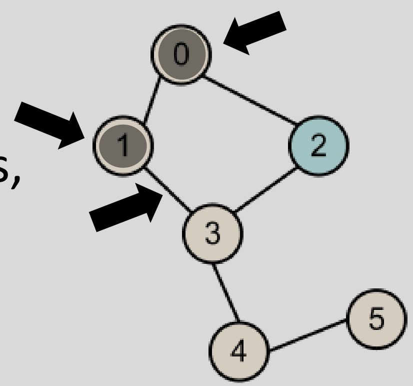
Lower bounds (local): distributed memories

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

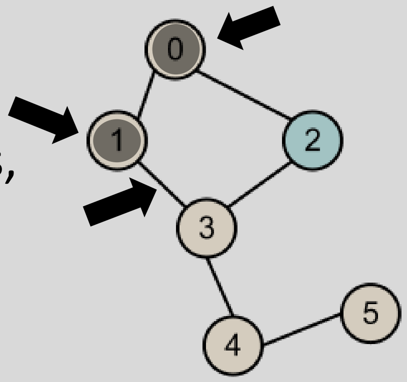
A Cray XE/XT supercomputer

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



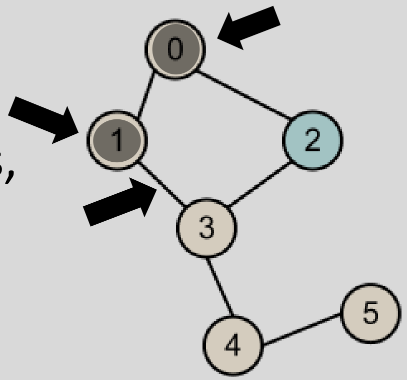
Lower bounds (local): distributed memories

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

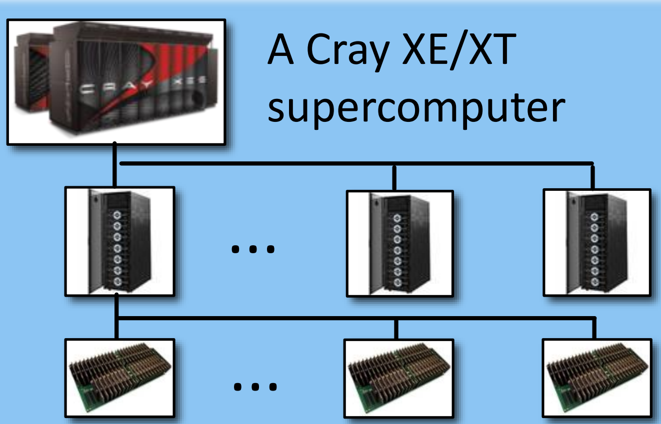
This is it? Still not really ☺

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



4 cabinets:

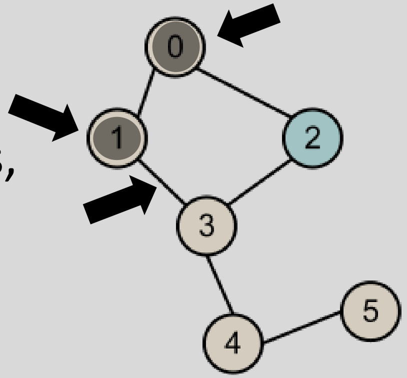
3 chassis:

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

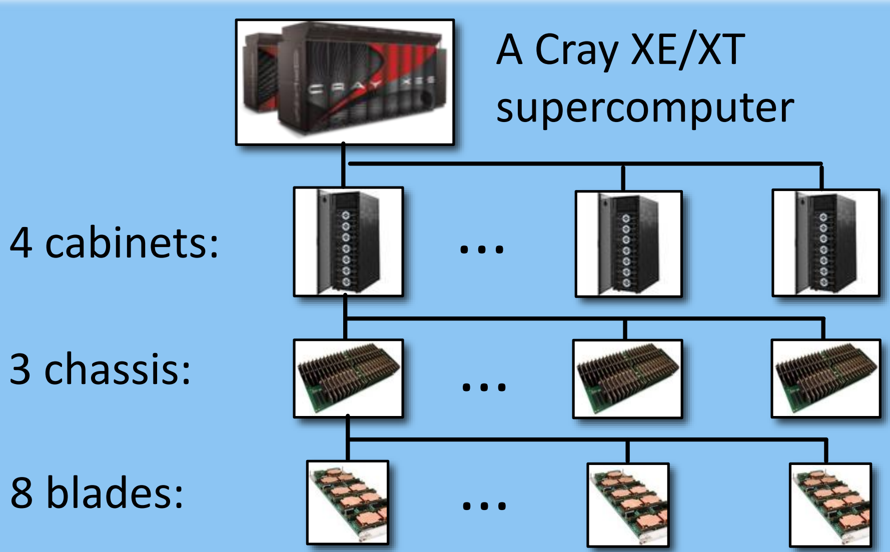
This is it? Still not really ☺

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

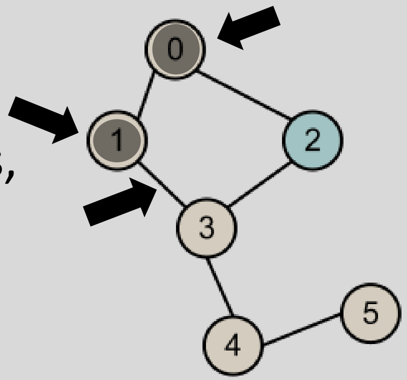


1 **Log (Vertex labels), Log (Edge weights)**

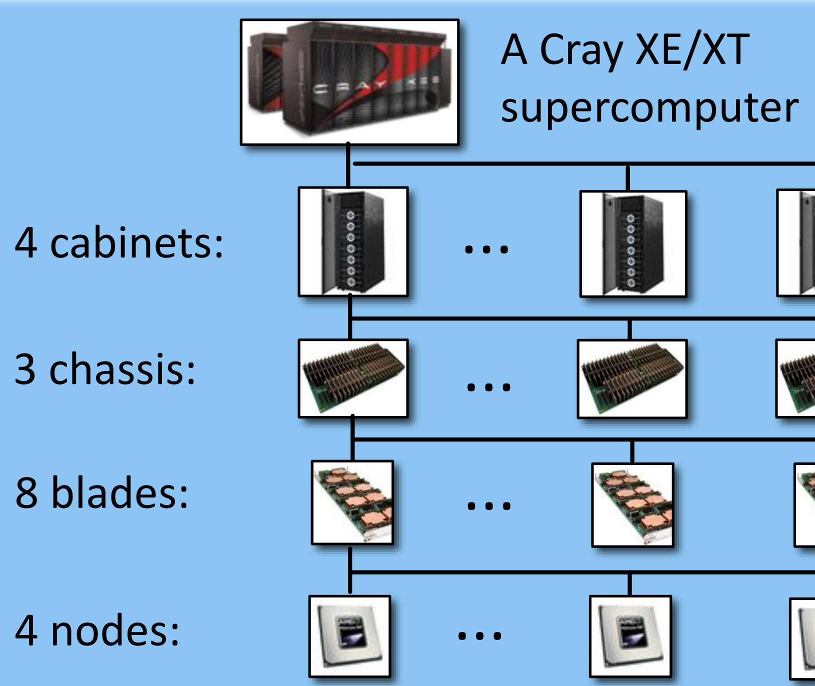
This is it? Still not really ☺

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

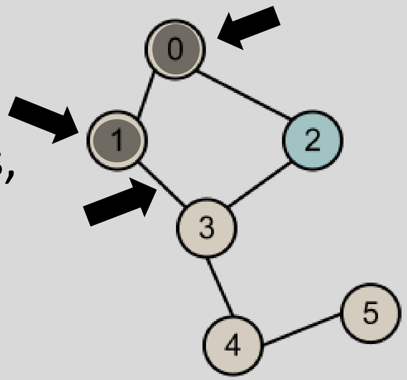


1 **Log (Vertex labels), Log (Edge weights)**

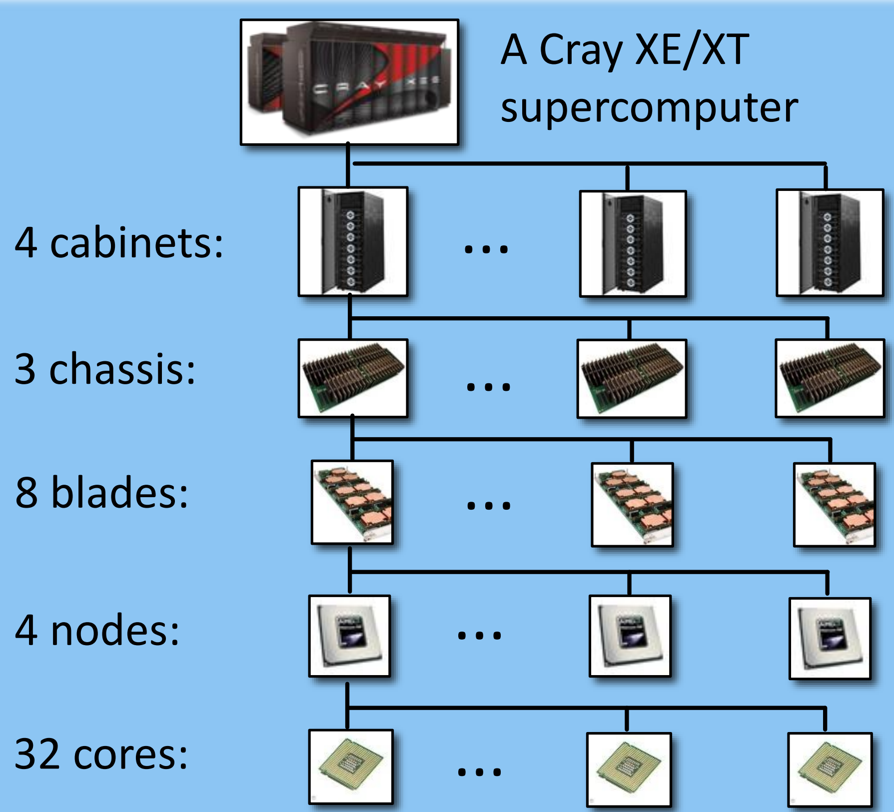
This is it? Still not really ☺

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

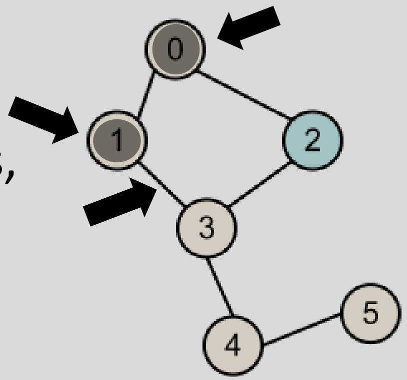


1 **Log (Vertex labels), Log (Edge weights)**

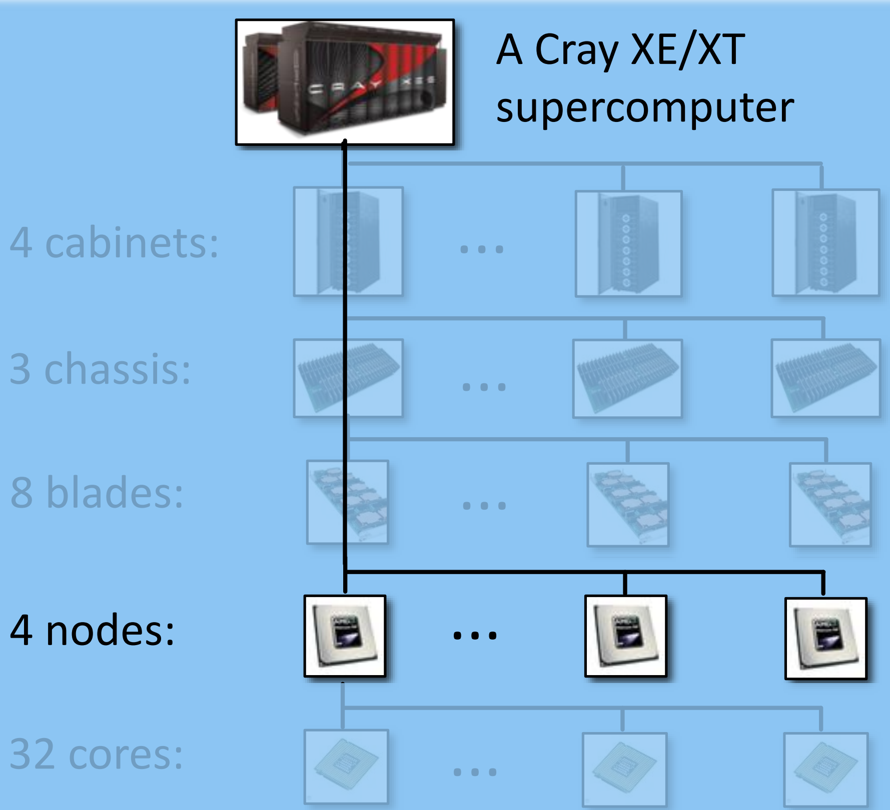
This is it? Still not really ☺

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



A Cray XE/XT supercomputer

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

This is it? Still not really ☺

Symbols

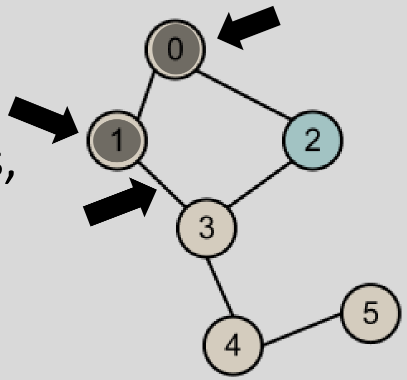
n : #vertices,

m : #edges,

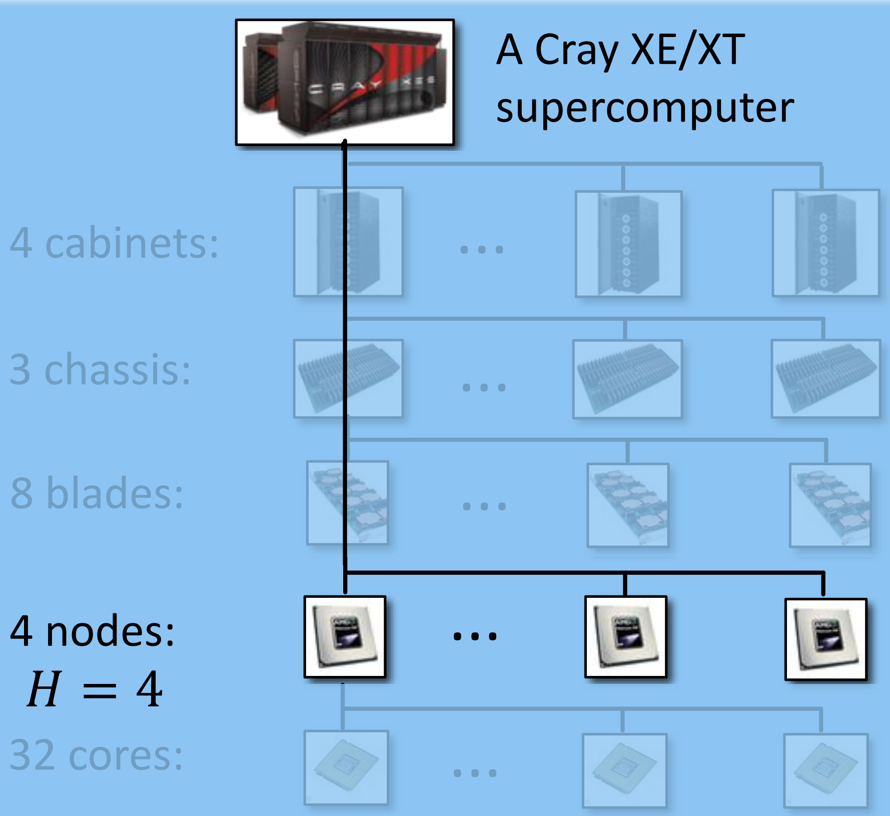
H : number of compute nodes,

H_i : number of machine elements at level i,

N : number of machine levels



Lower bounds (local): distributed memories

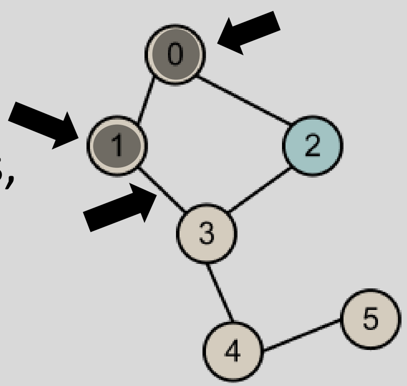


1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really ☺

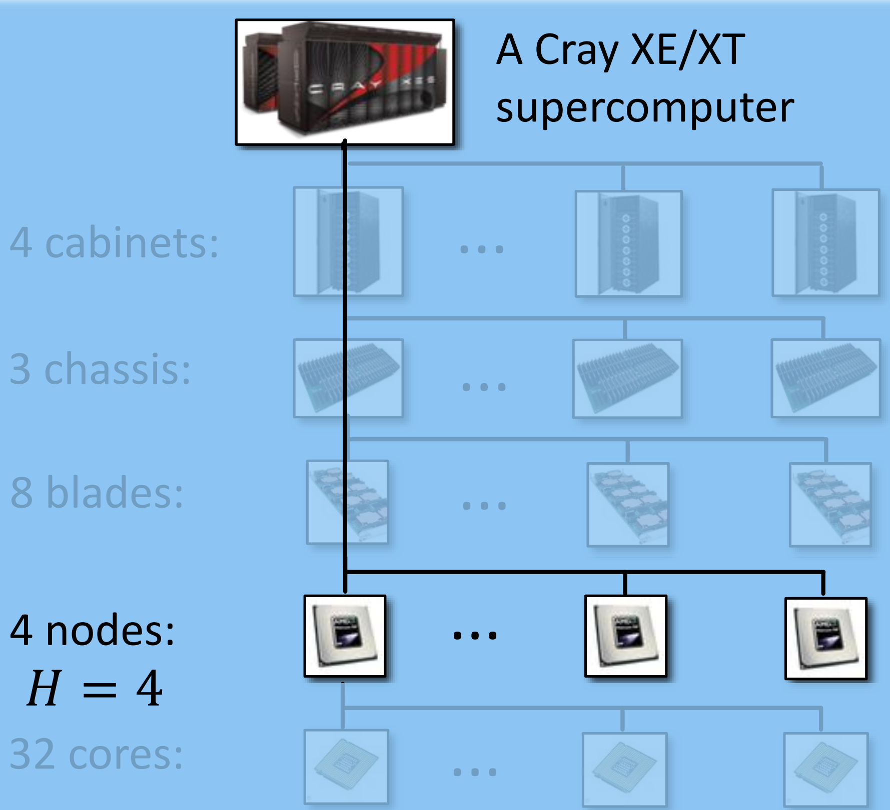
Symbols

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:

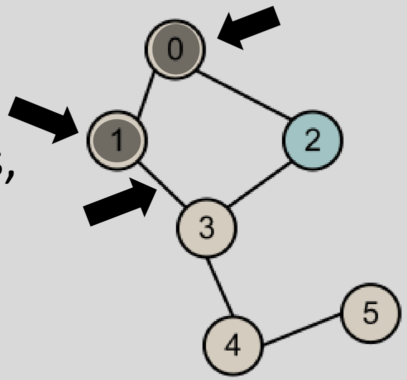


1 Log (Vertex labels), Log (Edge weights)

This is it? Still not really ☺

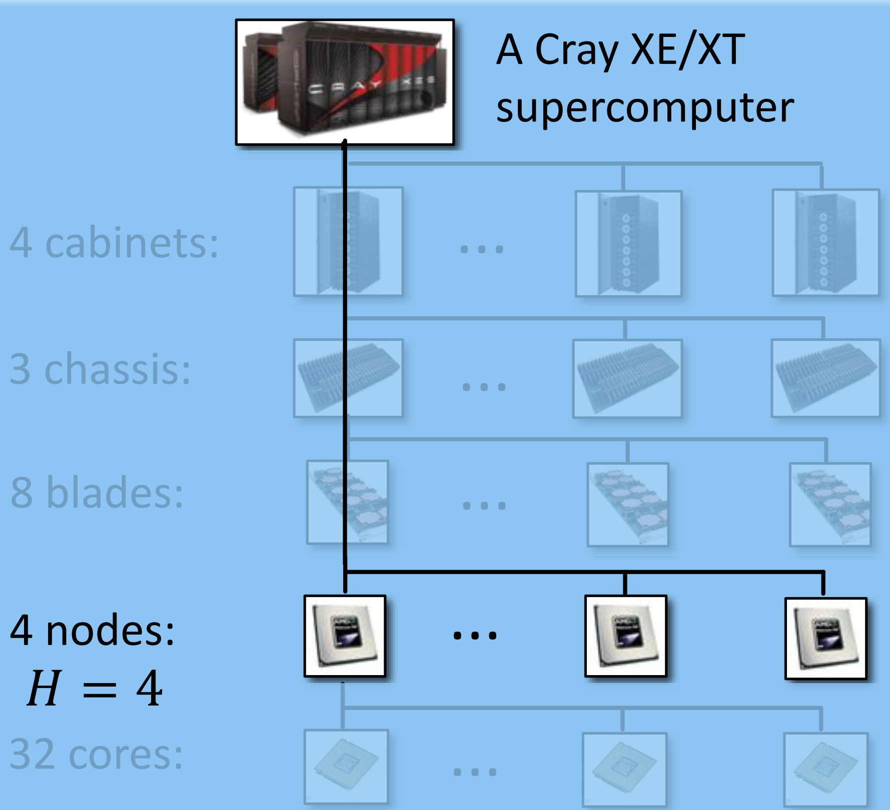
Symbols

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: $\frac{n}{H}$



A Cray XE/XT supercomputer

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

This is it? Still not really ☺

Symbols

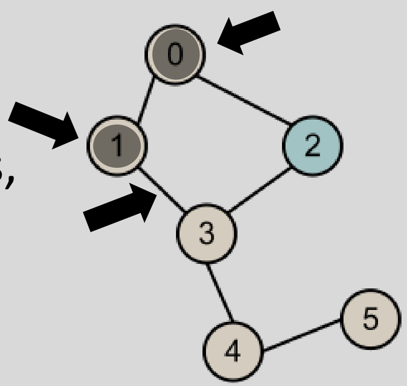
n : #vertices,

m : #edges,

H : number of compute nodes,

H_i : number of machine elements at level i,

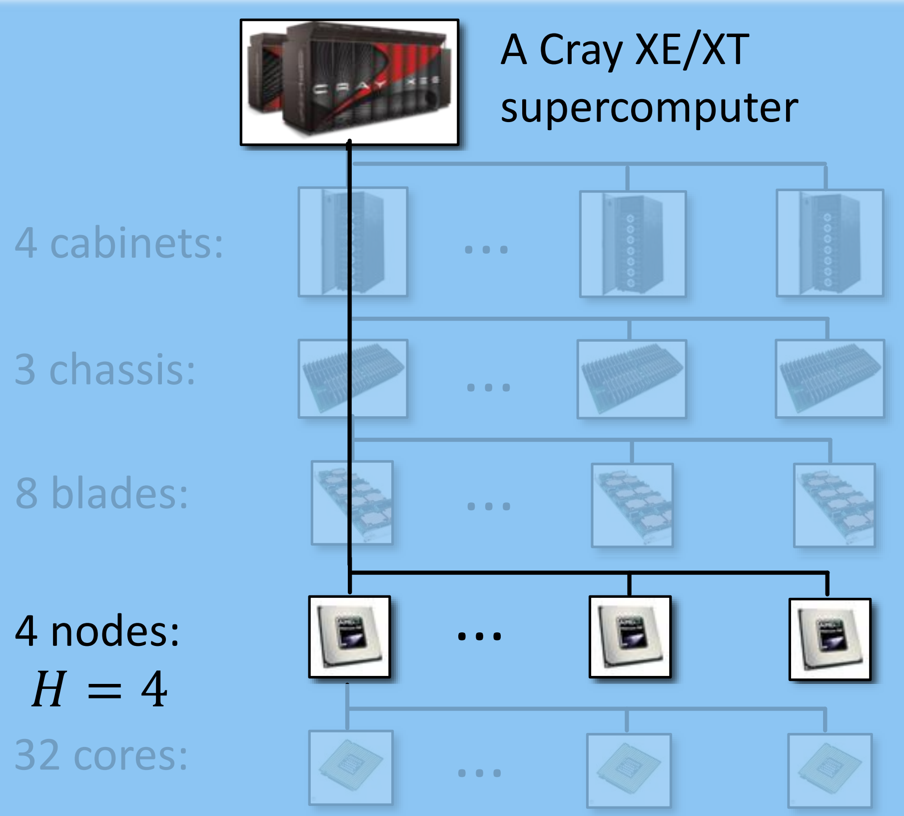
N : number of machine levels



Lower bounds (local): distributed memories

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

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

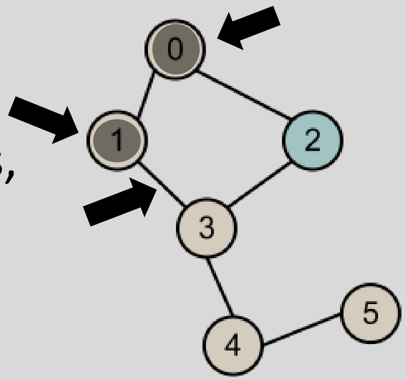


1 Log (Vertex labels), Log (Edge weights)

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

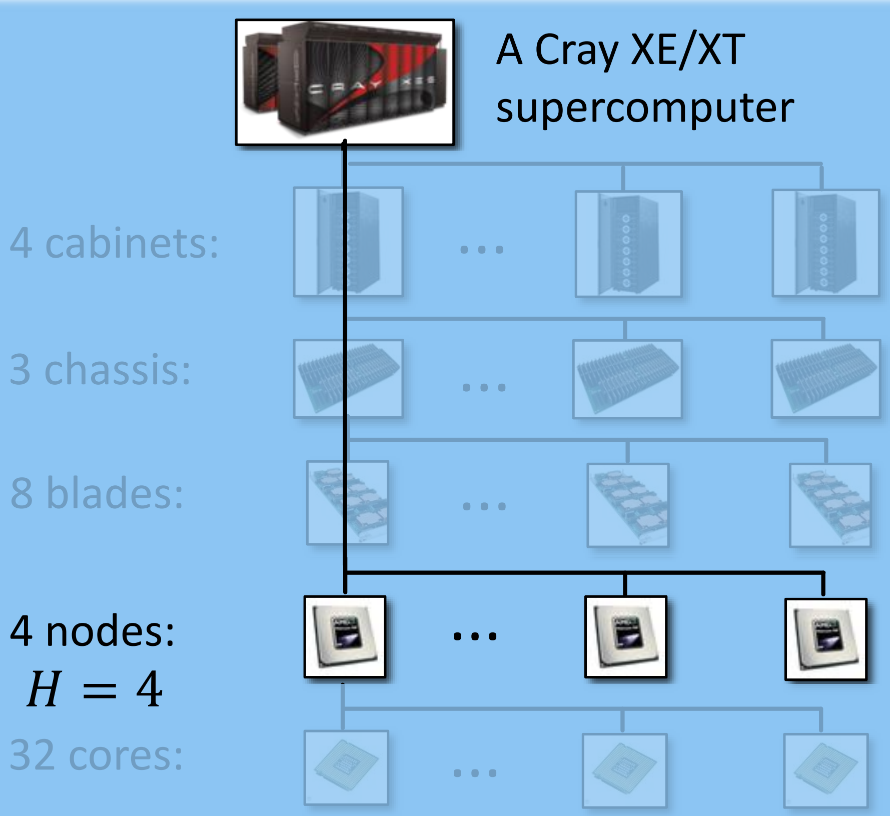


Lower bounds (local): distributed memories

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

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

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

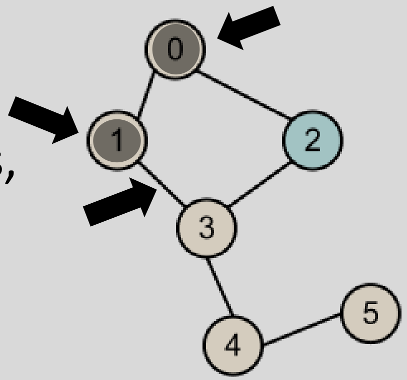


1 Log (Vertex labels), Log (Edge weights)

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

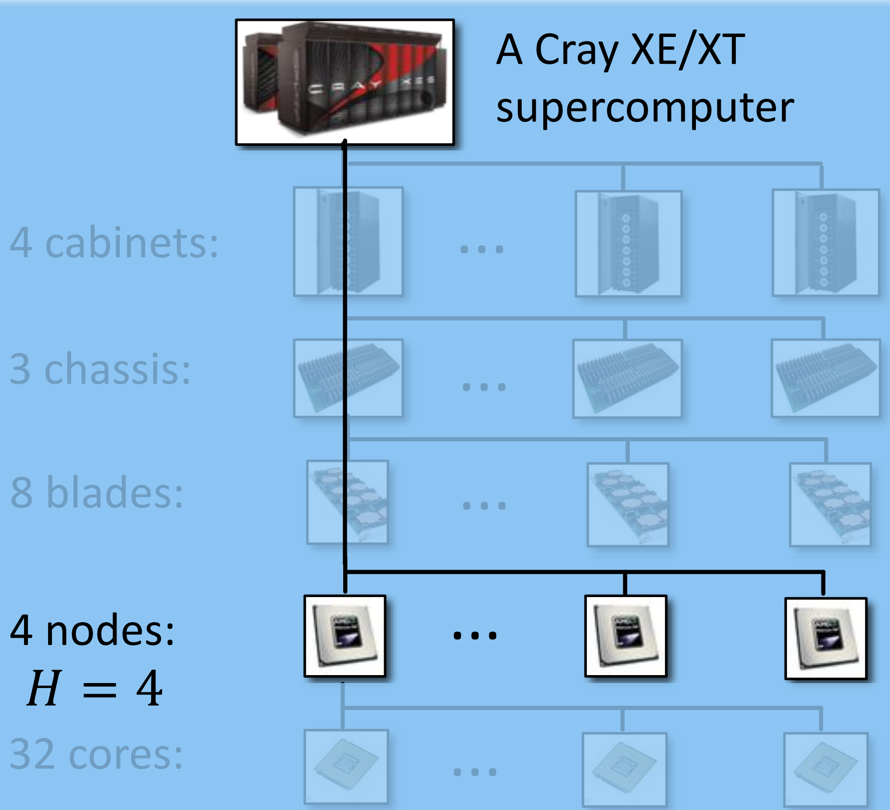


Lower bounds (local): distributed memories

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

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

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



A Cray XE/XT supercomputer

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

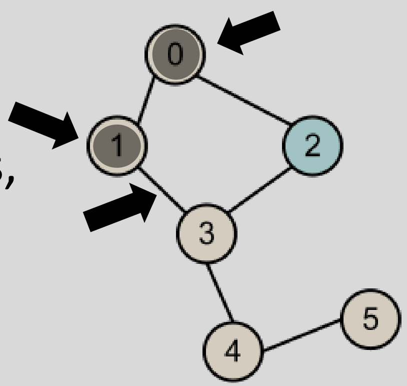
$$n \left\lceil \log \frac{n}{H} \right\rceil + H \lceil \log H \rceil$$

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

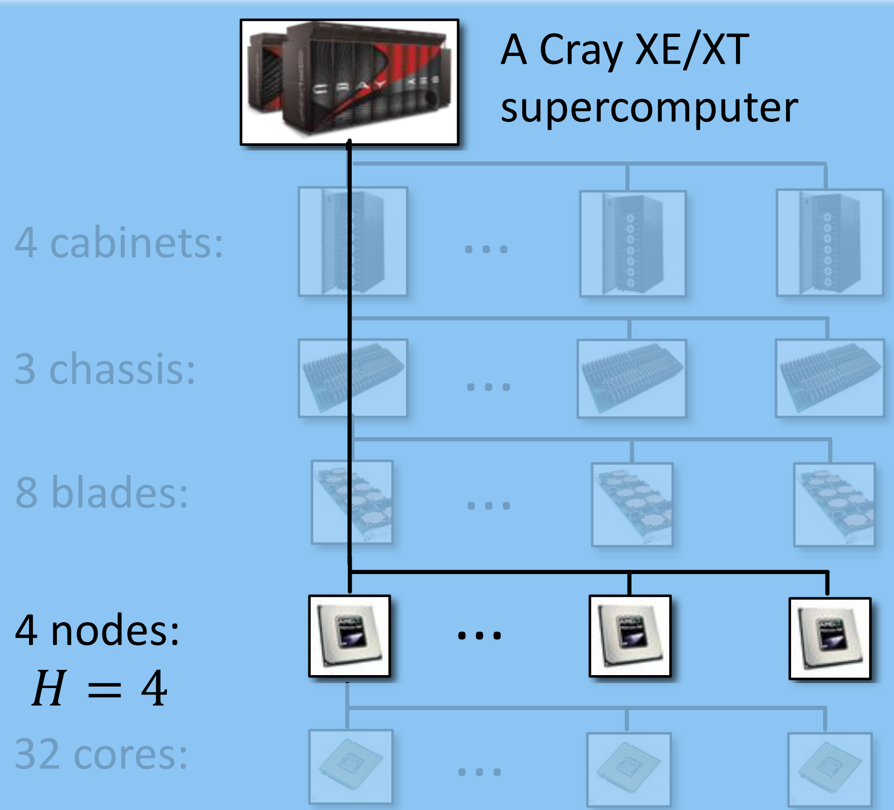


Lower bounds (local): distributed memories

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

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

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



A Cray XE/XT supercomputer

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

$$n \left\lceil \log \frac{n}{H} \right\rceil + H \lceil \log H \rceil$$

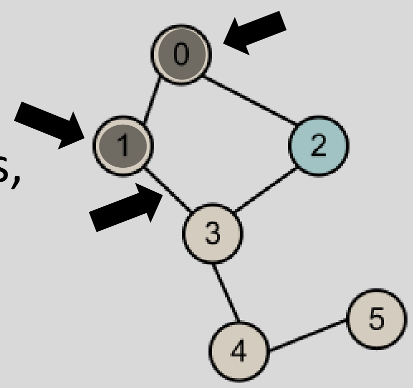
We also generalize this to arbitrarily many levels (details in the paper ☺) and derive the total size:

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



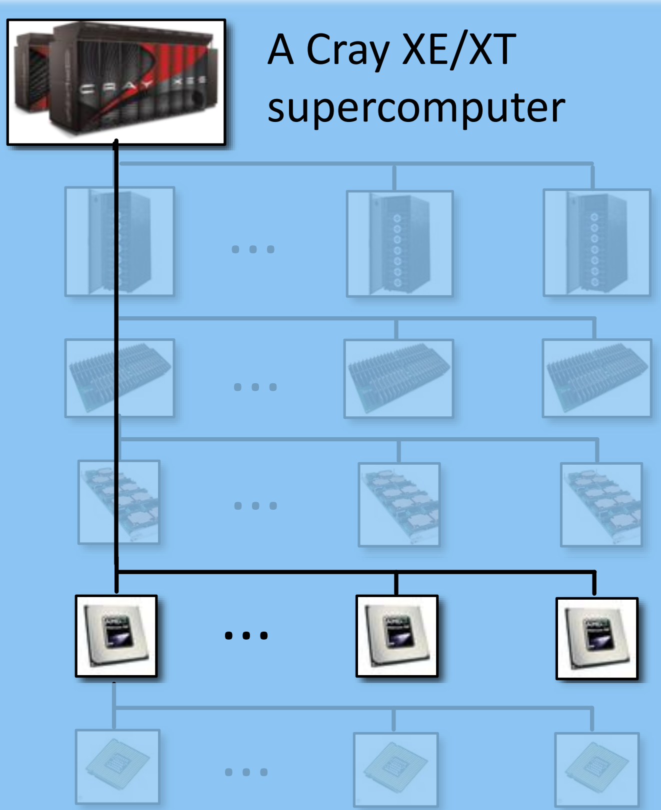
Lower bounds (local): distributed memories

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

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

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

A Cray XE/XT supercomputer



4 cabinets:

3 chassis:

8 blades:

4 nodes: $H = 4$

32 cores:

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

$$n \left\lceil \log \frac{n}{H} \right\rceil + H \lceil \log H \rceil$$

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

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

1 **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 ☺)

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

$$\forall_{v,u\in V}\,(u \in N_v) \Rightarrow \left[\mathcal{N}(u) \le \widehat{N}_v\right]$$

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

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

$$|\mathcal{A}| = n\left\lceil \log \frac{n}{\mathcal{H}}\right\rceil + \mathcal{H}\lceil \log \mathcal{H}\rceil$$

$$|\mathcal{A}| = 2m\left(\lceil \log n\rceil + \left\lceil \log \widehat{\mathcal{W}}\right\rceil\right)$$

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

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

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

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

Formal analyses: more (check the paper ☺)

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

$$\forall_{v,u\in V}\left(u \in N_v\right) \Rightarrow \left[\mathcal{N}(u) \le \widehat{N}_v\right]$$

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

$$|\mathcal{A}| = 2m\left(\lceil \log n\rceil + \left\lceil \log \widehat{\mathcal{W}}\right\rceil\right)$$

$$\widehat{N}_v\rceil + \left\lceil \log \widehat{\mathcal{W}}\right\rceil\right) + \left\lceil \log\log \widehat{N}_v\right\rceil + \left\lceil \log\log \widehat{\mathcal{W}}\right\rceil\right)$$

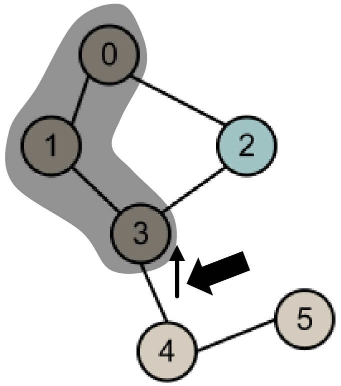
$$E[|\mathcal{A}|] = \left(\lceil \log n\rceil + \left\lceil \log \widehat{\mathcal{W}}\right\rceil\right) pn^2$$

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

2 Log (Offset structure)

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

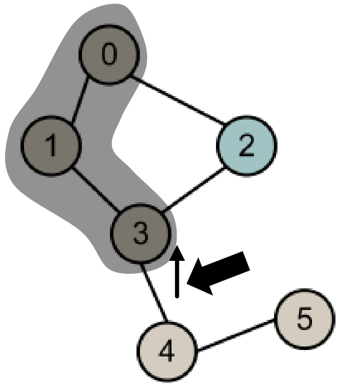
!



2 Log (Offset structure)

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

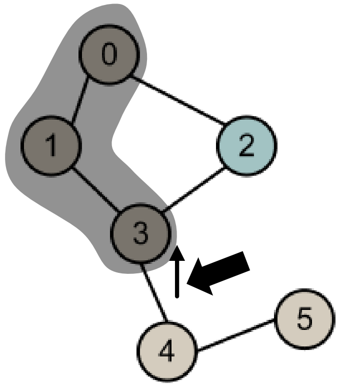
!



Formal analyses

2 Log (Offset structure)

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

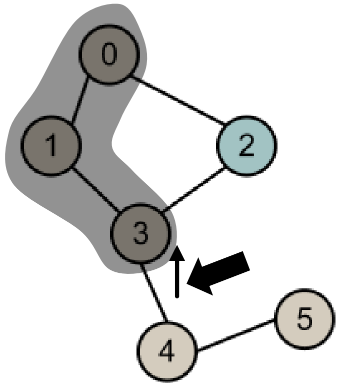


∰ Formal analyses

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

2 Log (Offset structure)

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



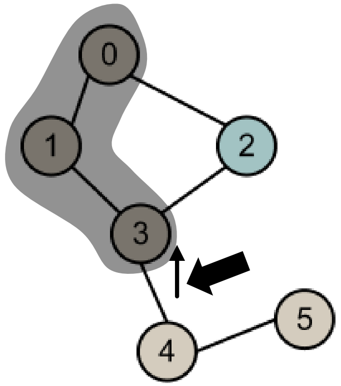
Formal analyses

Check the paper for details ☺

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

2 Log (Offset structure)

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



Formal analyses

Check the paper for details ☺

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

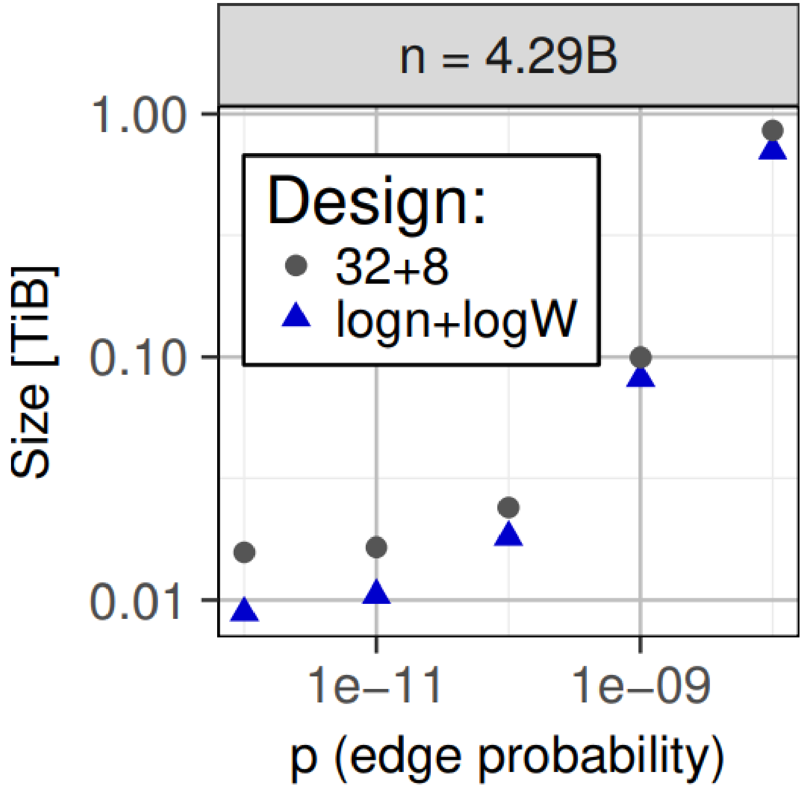
Key methods

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

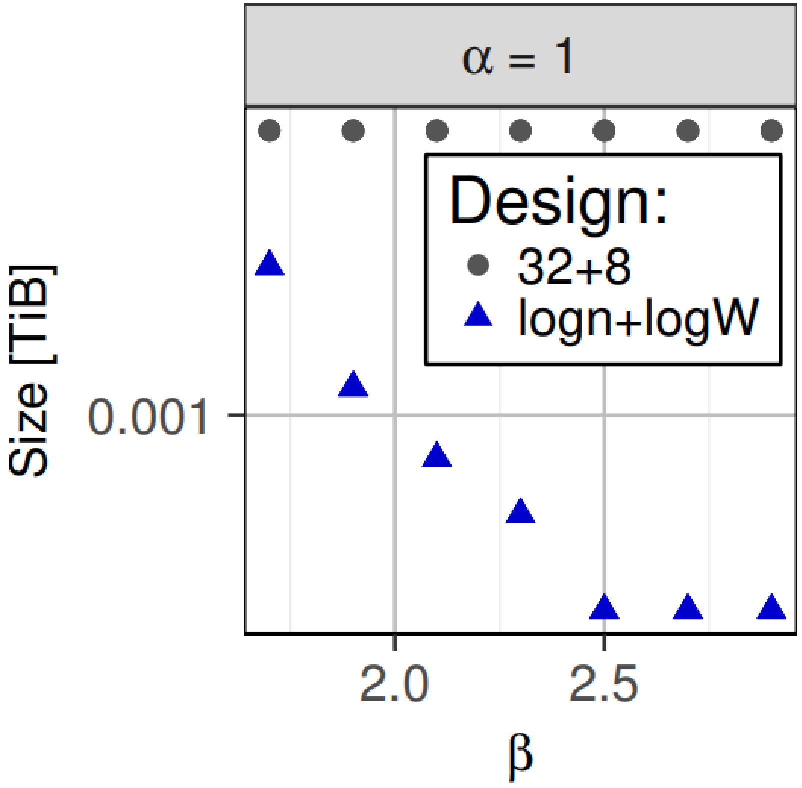
[1] S. Gog. SDSL-Lite Succinct Library. 2015.

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

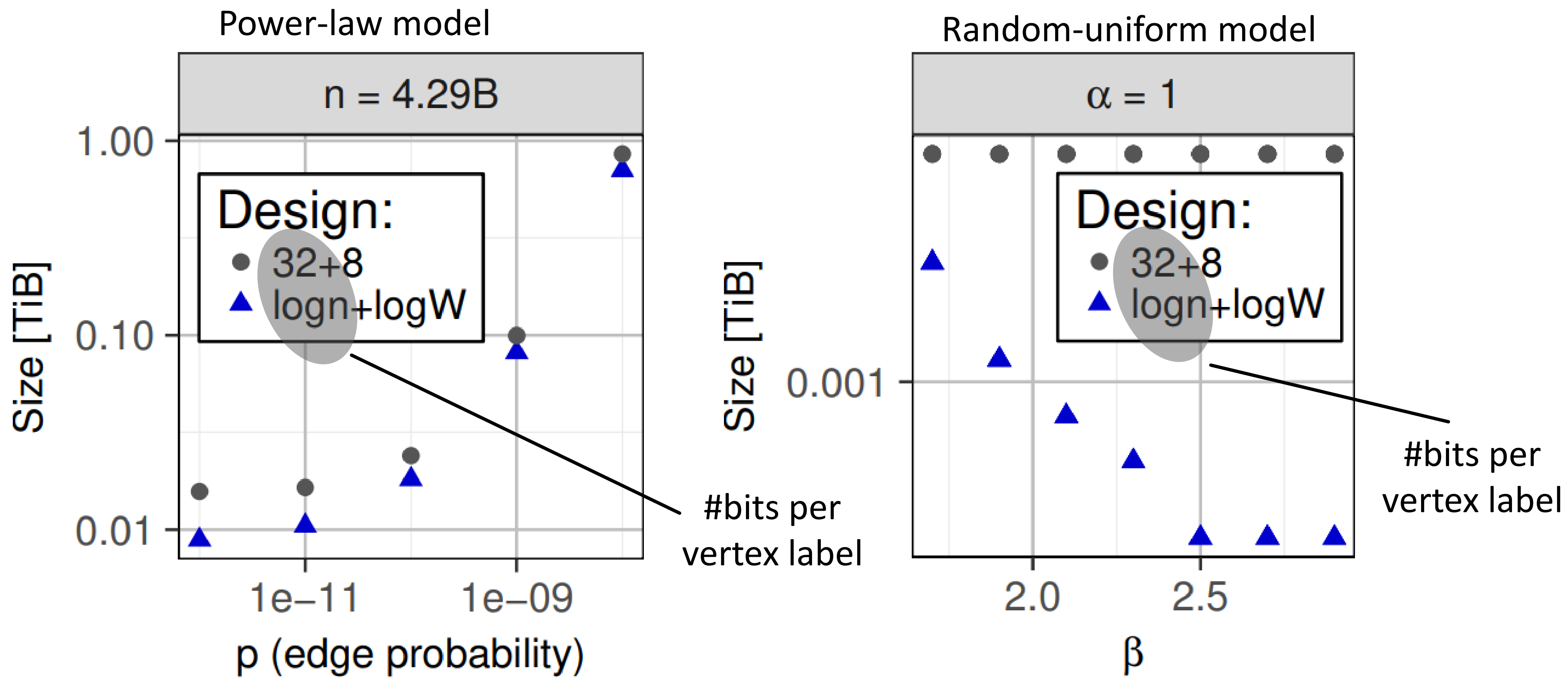
Power-law model



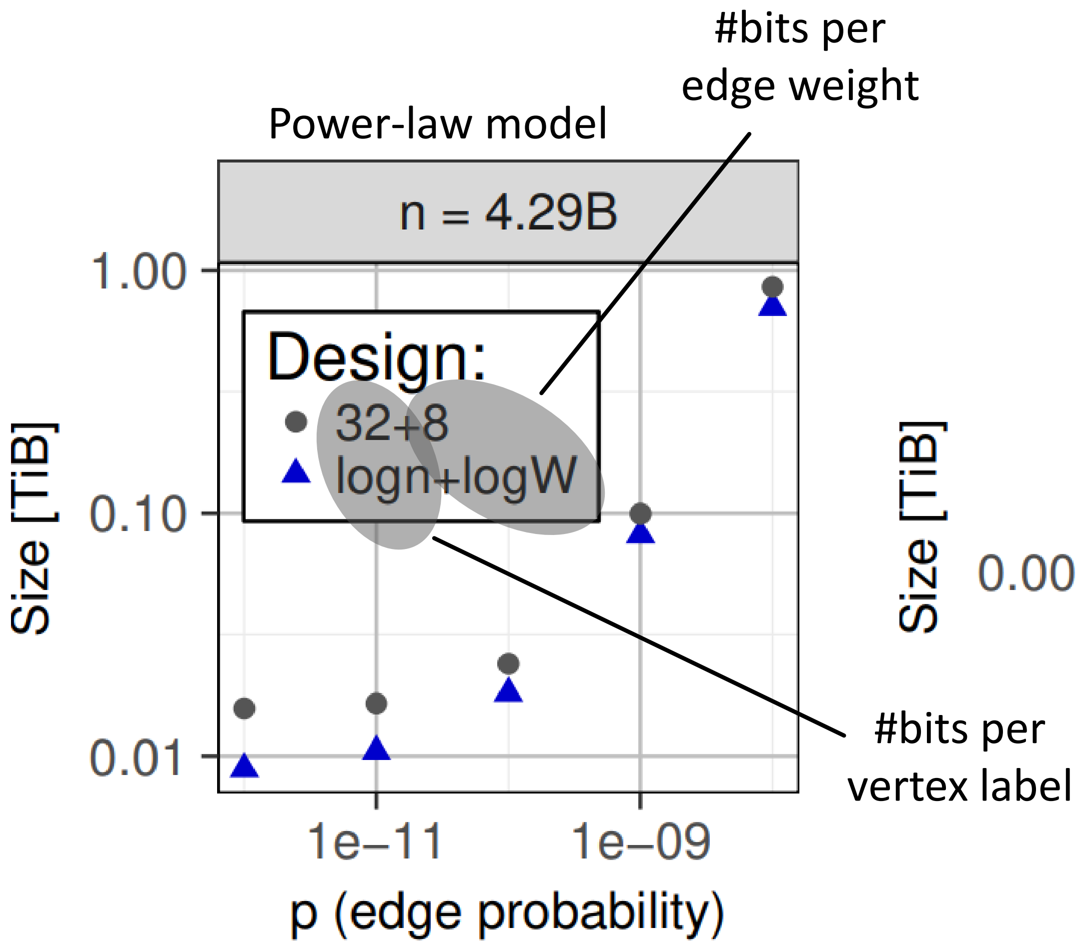
Random-uniform model

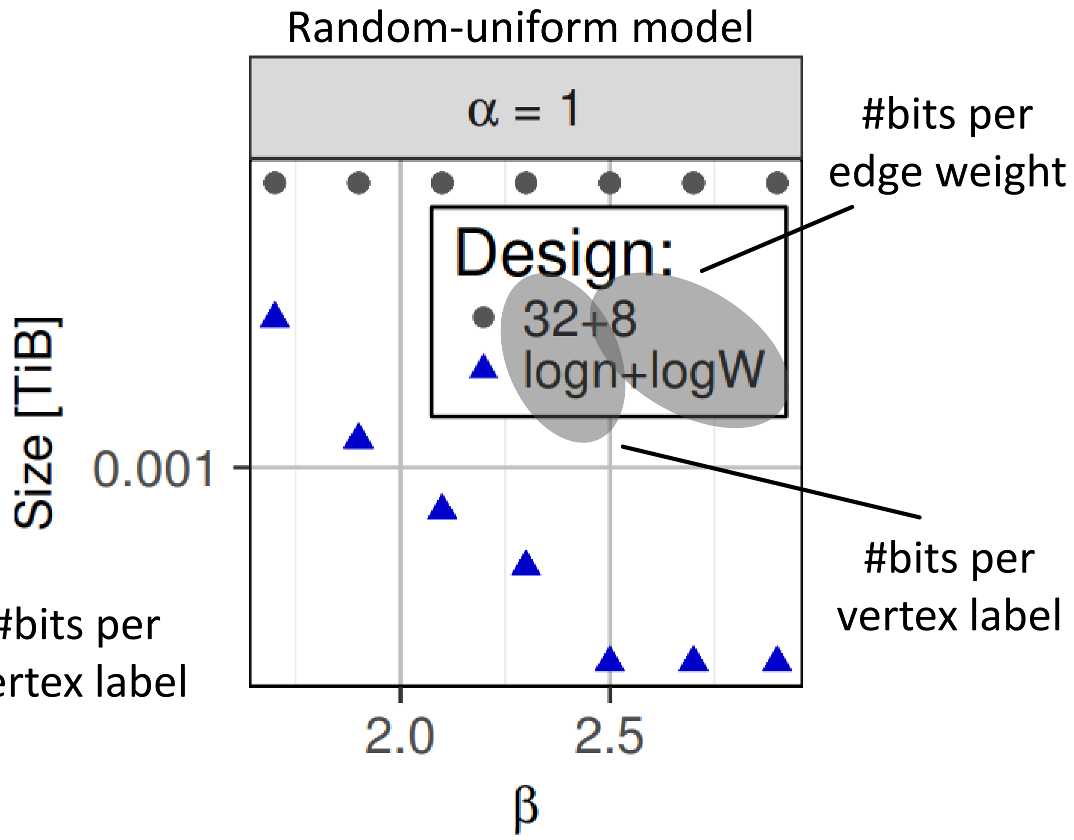


1 Log (Vertex labels), Log (Edge weights) Storage



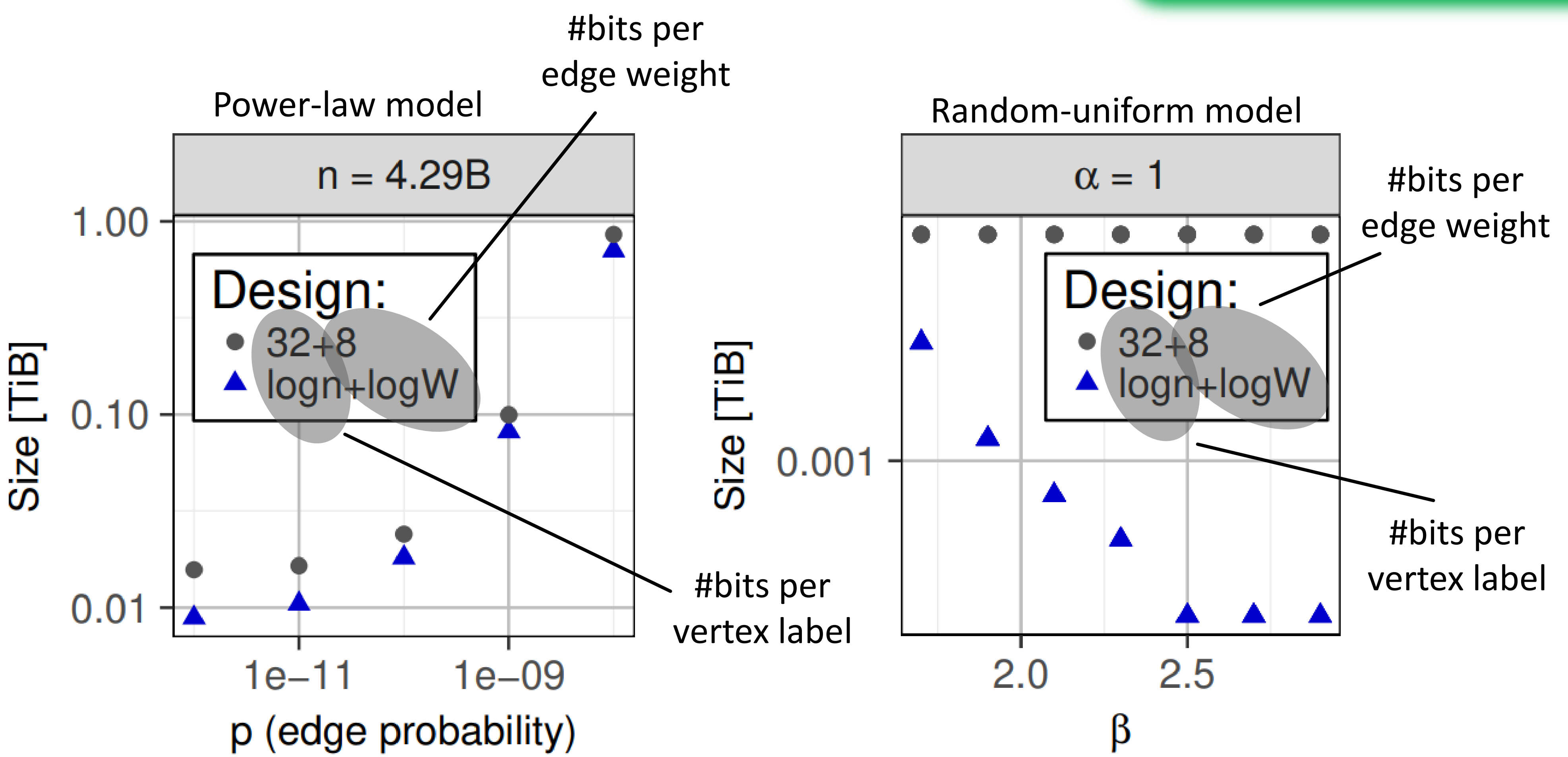
1 Log (Vertex labels), Log (Edge weights) Storage





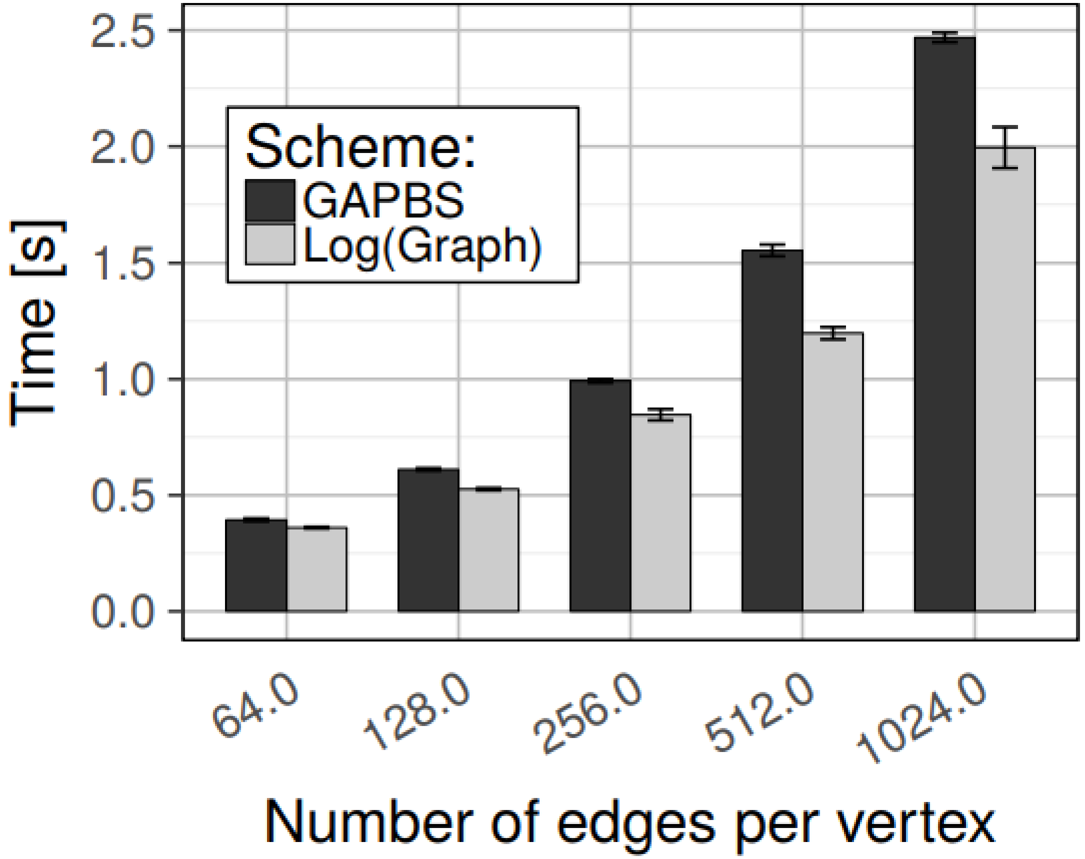
1 **Log (Vertex labels), Log (Edge weights) Storage**

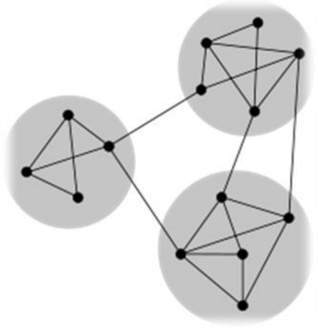
Log(Graph) consistently reduces storage overhead (by 20-35%)



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

SSSP

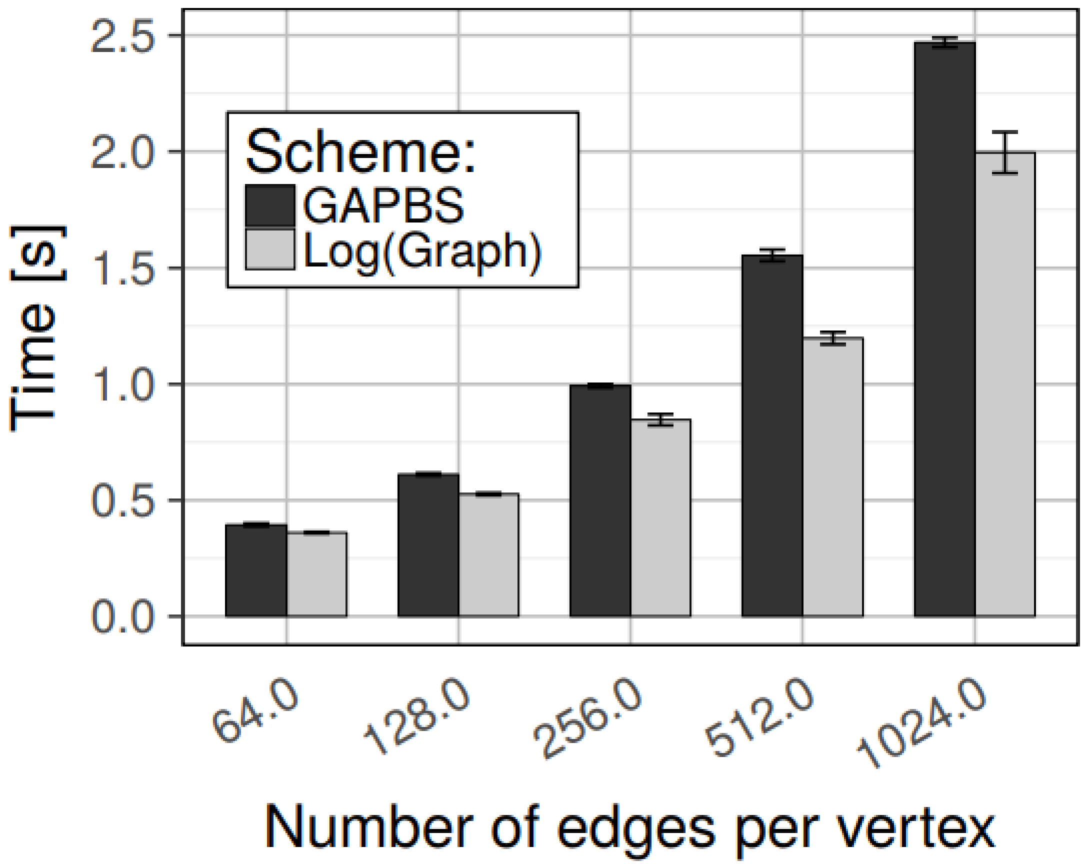




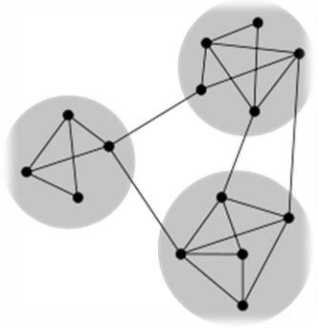
Kronecker graphs
Number of vertices: 4M

SSSP

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



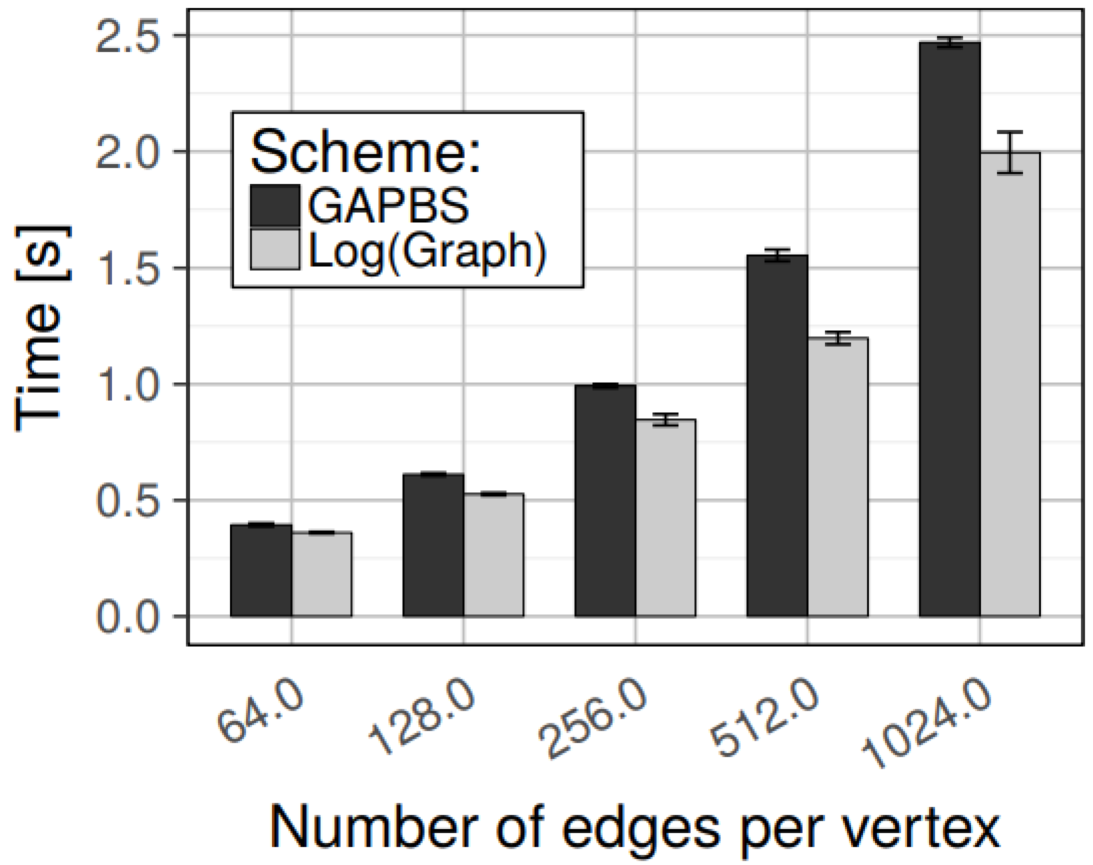
Log(Graph) **accelerates** GAPBS



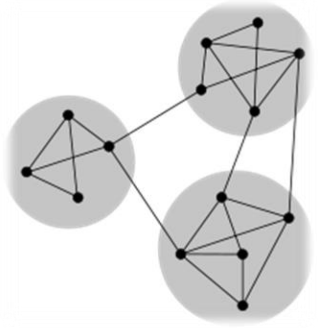
Kronecker graphs
Number of vertices: 4M

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

SSSP



Log(Graph) **accelerates** GAPBS



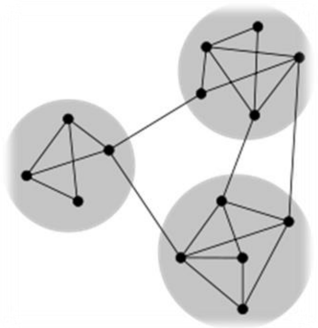
Kronecker graphs
Number of vertices: 4M

Both storage and performance are improved **simultaneously**

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



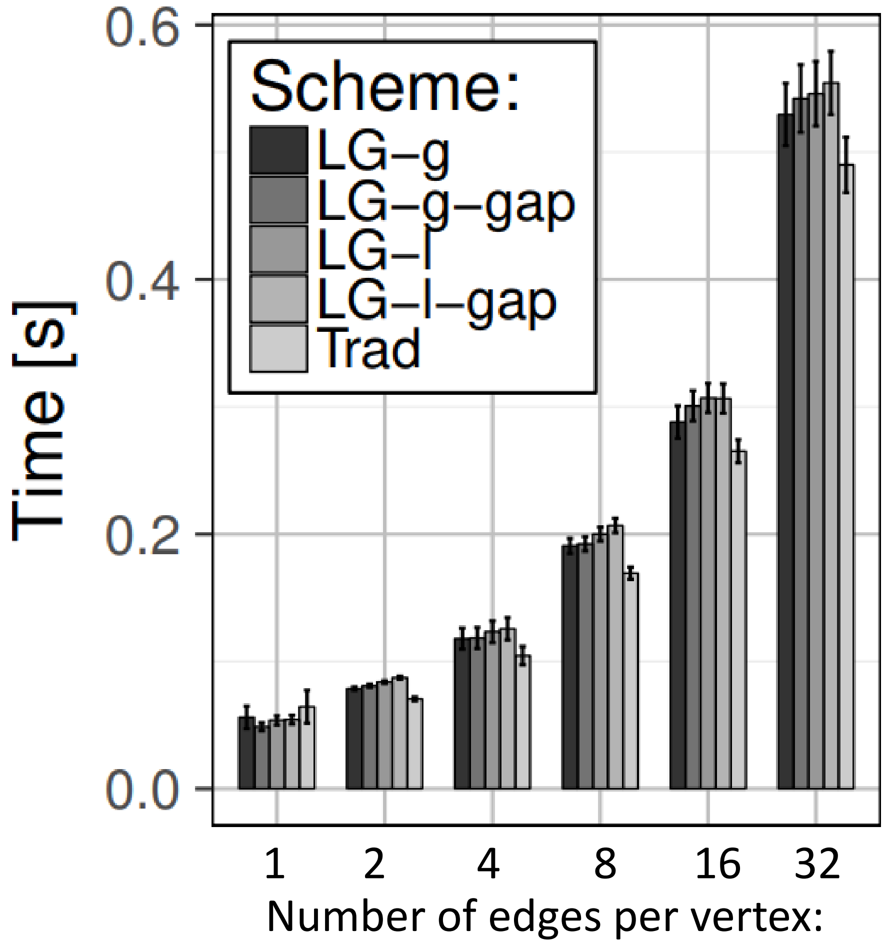
Kronecker graphs
Number of vertices: 4M

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

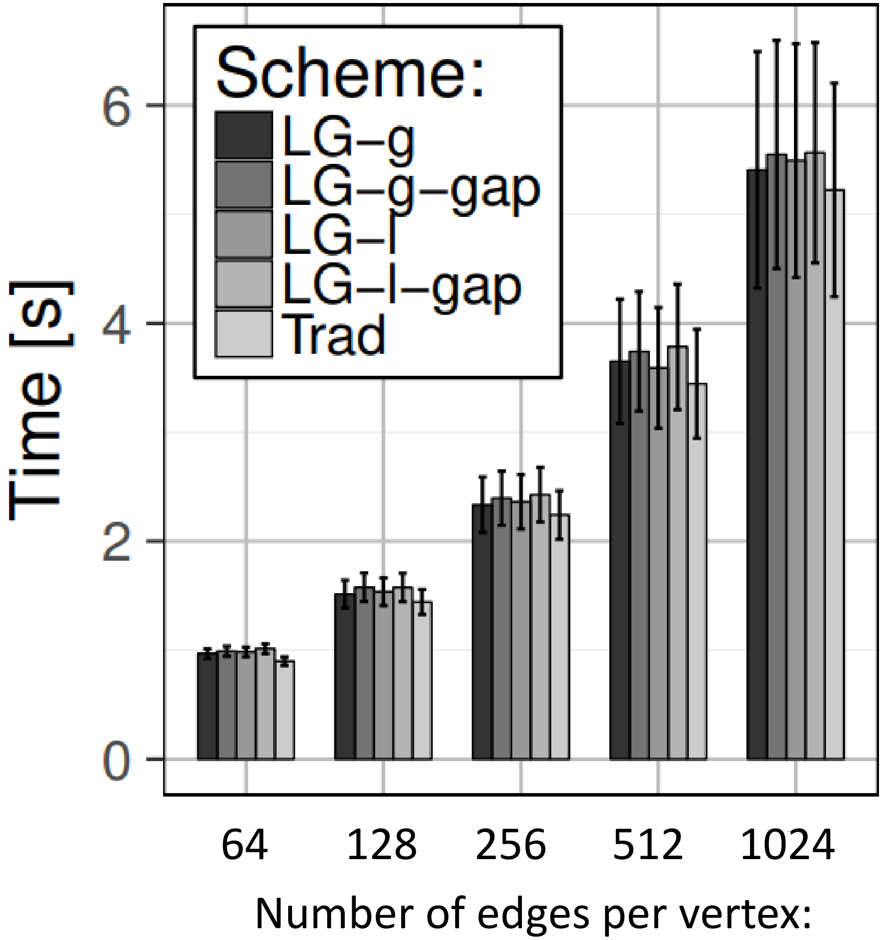
Betweenness Centrality

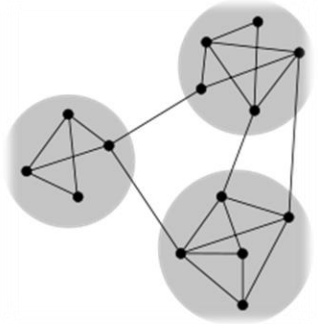
"**LG**": Log(Graph)
Trad: Traditional (non compressed, GAPBS)
"**g**": global scheme
"**l**": local scheme
"**gap**": additional gap encoding

Sparse graphs



Dense graphs





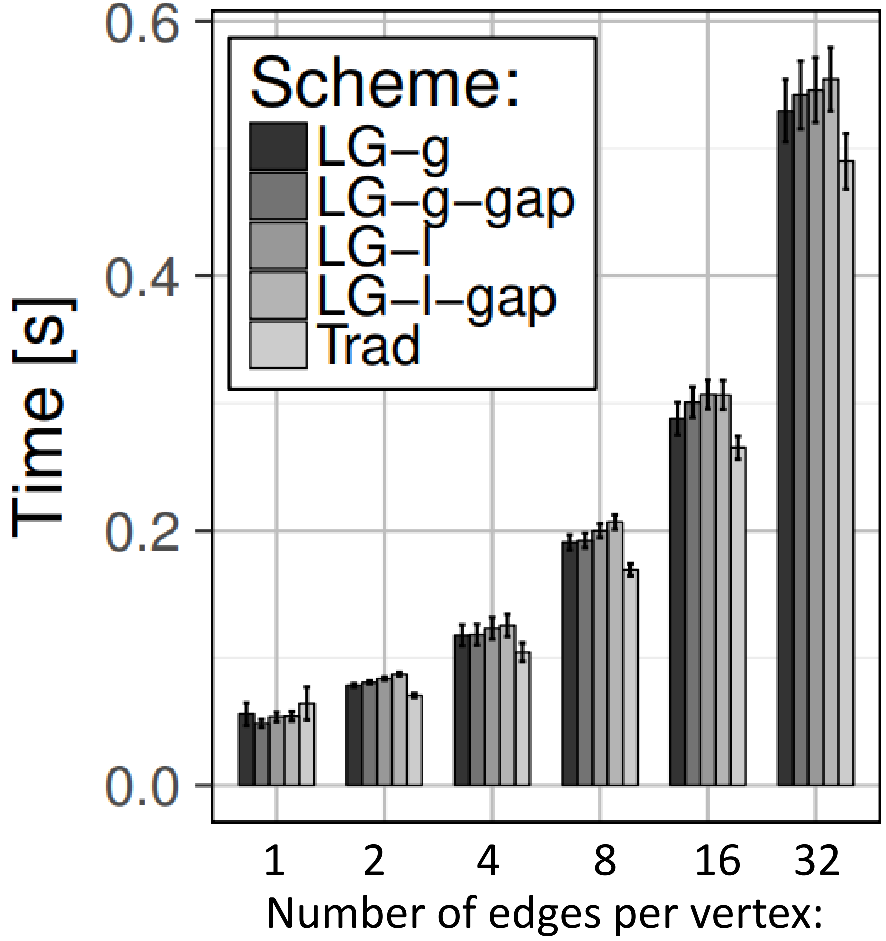
Kronecker graphs
Number of vertices: 4M

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

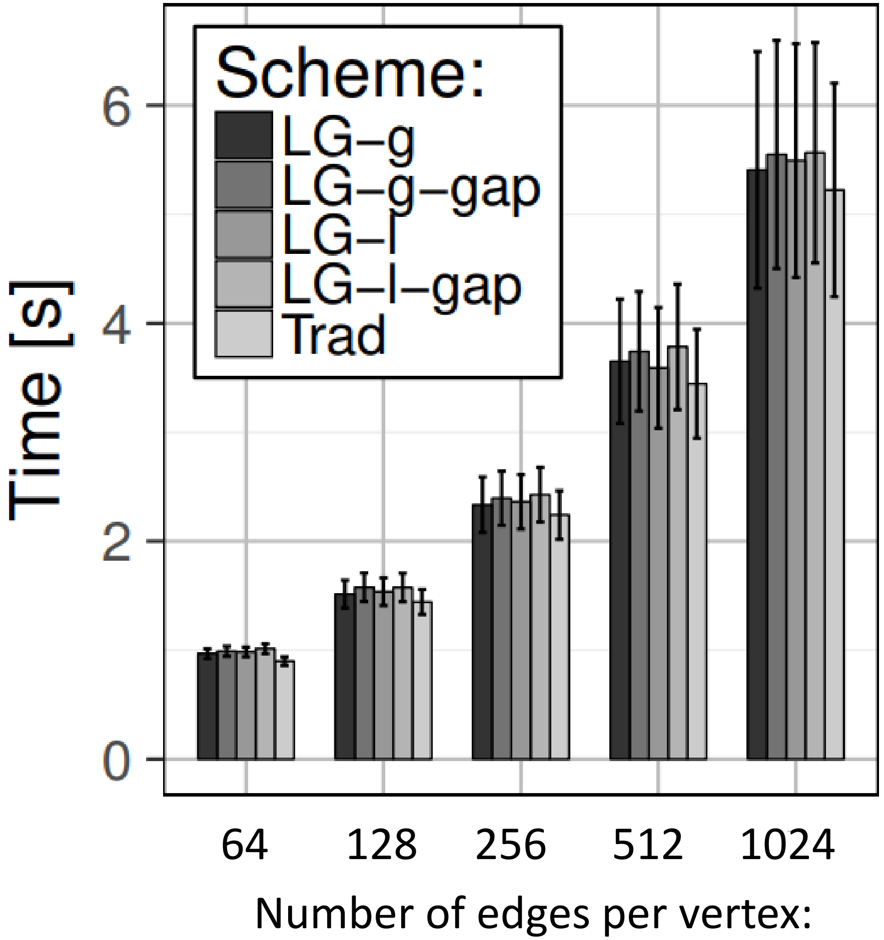
Betweenness Centrality

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

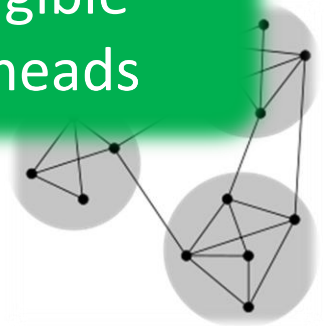
Sparse graphs



Dense graphs



Log(Graph) incurs negligible overheads

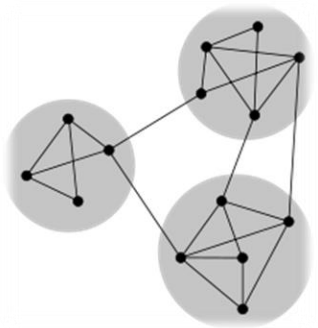


Kronecker graphs
Number of vertices: 4M

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



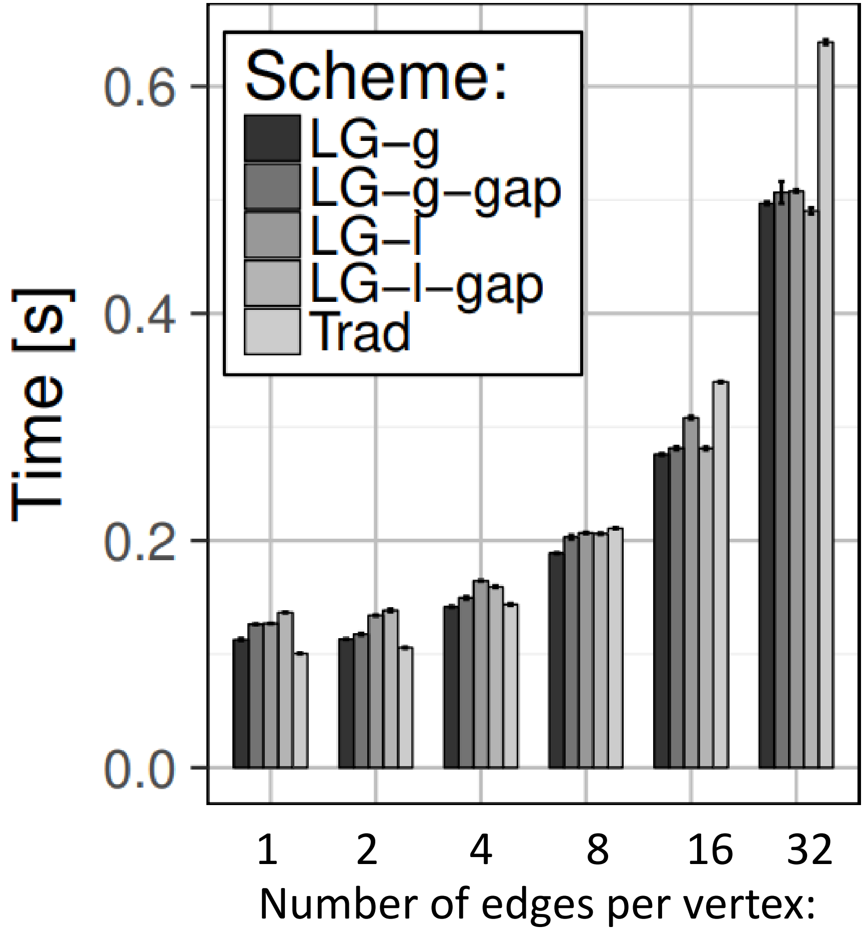
Kronecker graphs
Number of vertices: 4M

BFS

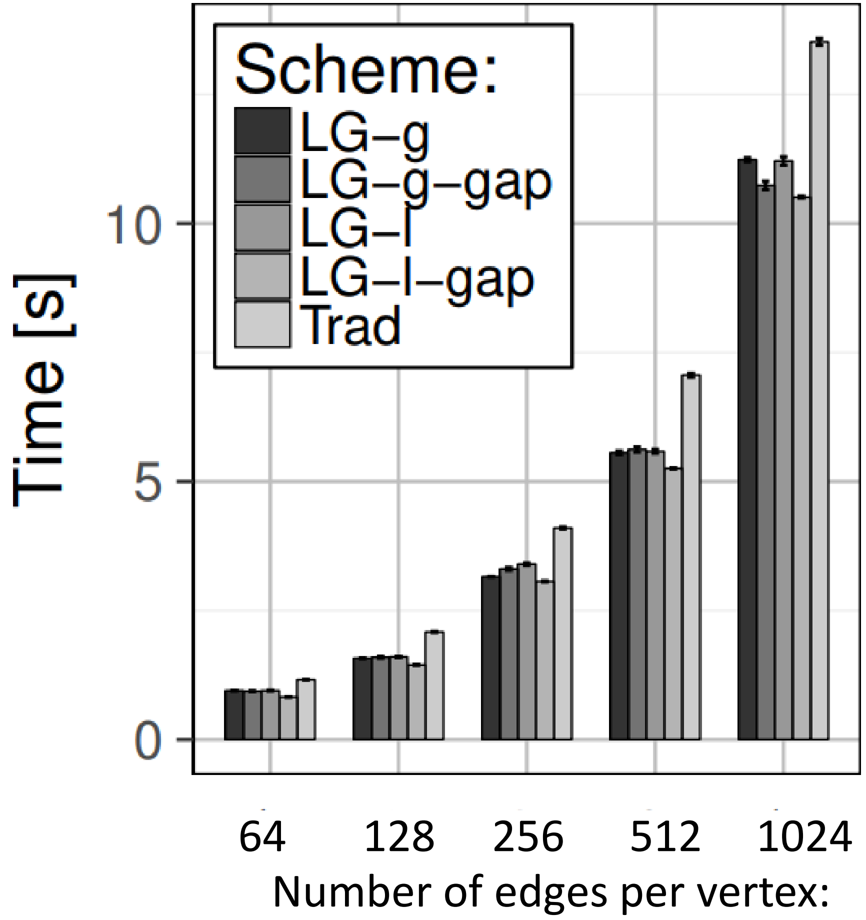
1 Log (Vertex labels), Log (Edge weights) Performance

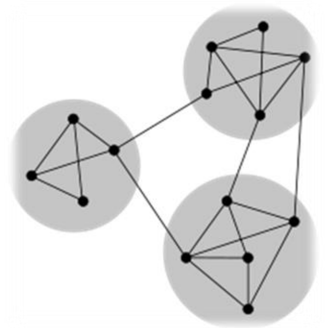
"**LG**": Log(Graph)
Trad: Traditional (non compressed, GAPBS)
"**g**": global scheme
"**l**": local scheme
"**gap**": additional gap encoding

Sparse graphs



Dense graphs





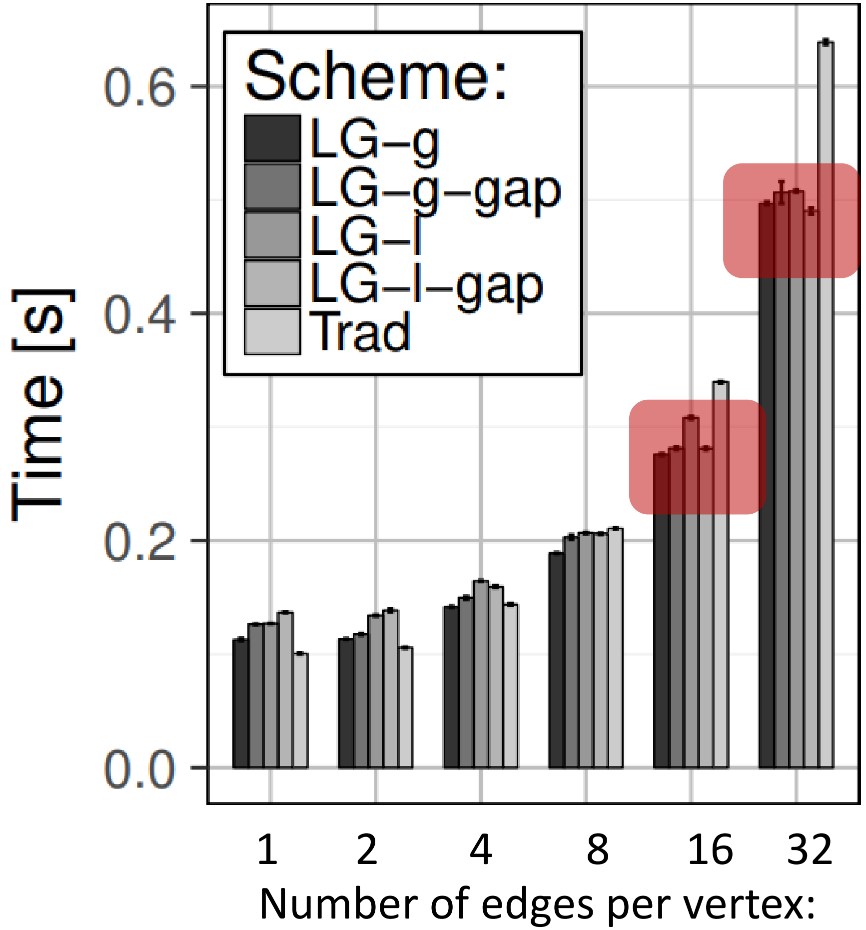
Kronecker graphs
Number of vertices: 4M

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

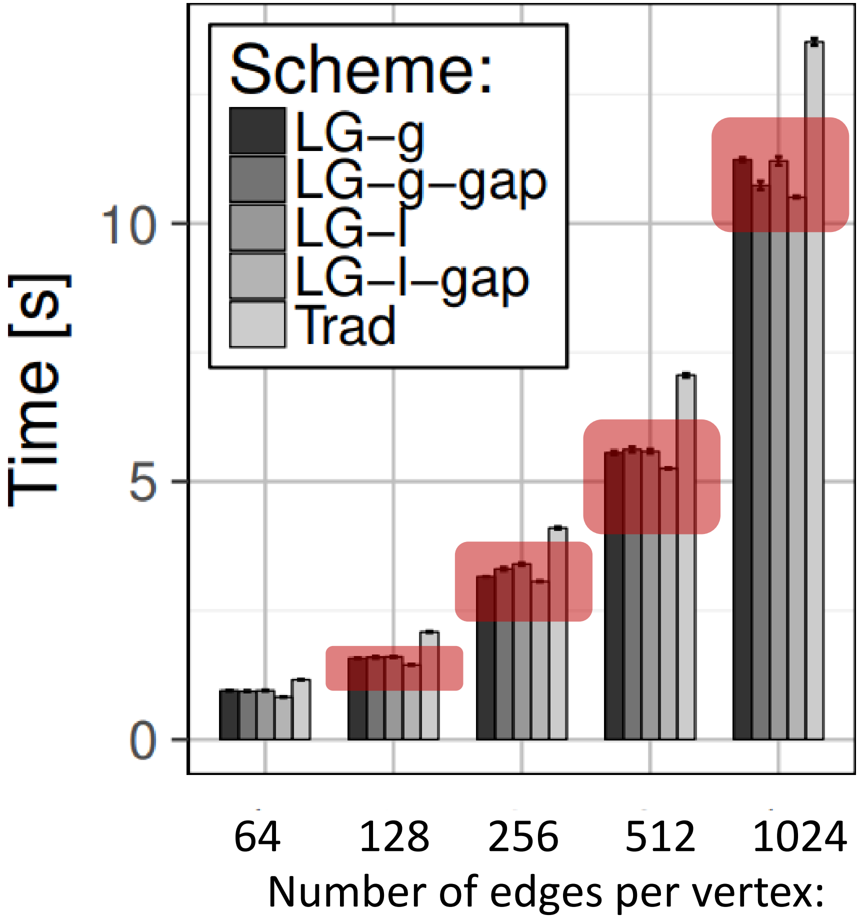
BFS

“**LG**”: Log(Graph)
Trad: Traditional (non compressed, GAPBS)
“**g**”: global scheme
“**l**”: local scheme
tional
ng

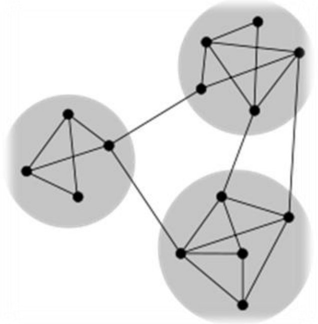
Sparse graphs



Dense graphs



Both storage and performance are improved **simultaneously**



Kronecker graphs
Number of vertices: 4M

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

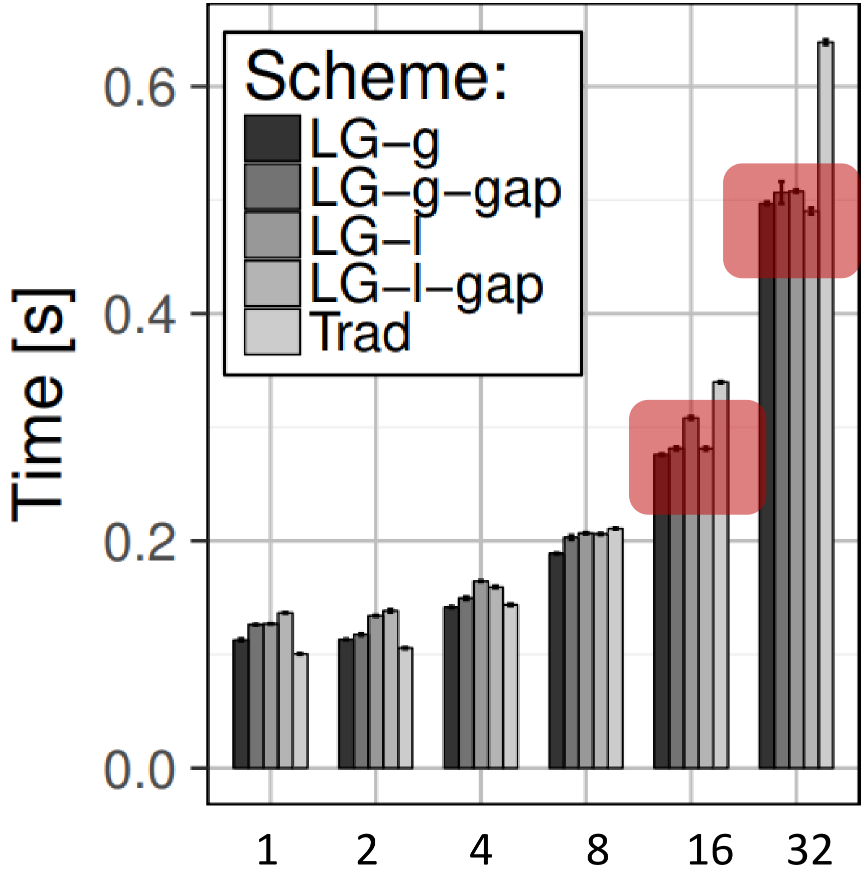
BFS

Log(Graph) **accelerates** GAPBS

"**LG**": Log(Graph)
Trad: Traditional (non compressed, GAPBS)
"**g**": global scheme
"**l**": local scheme

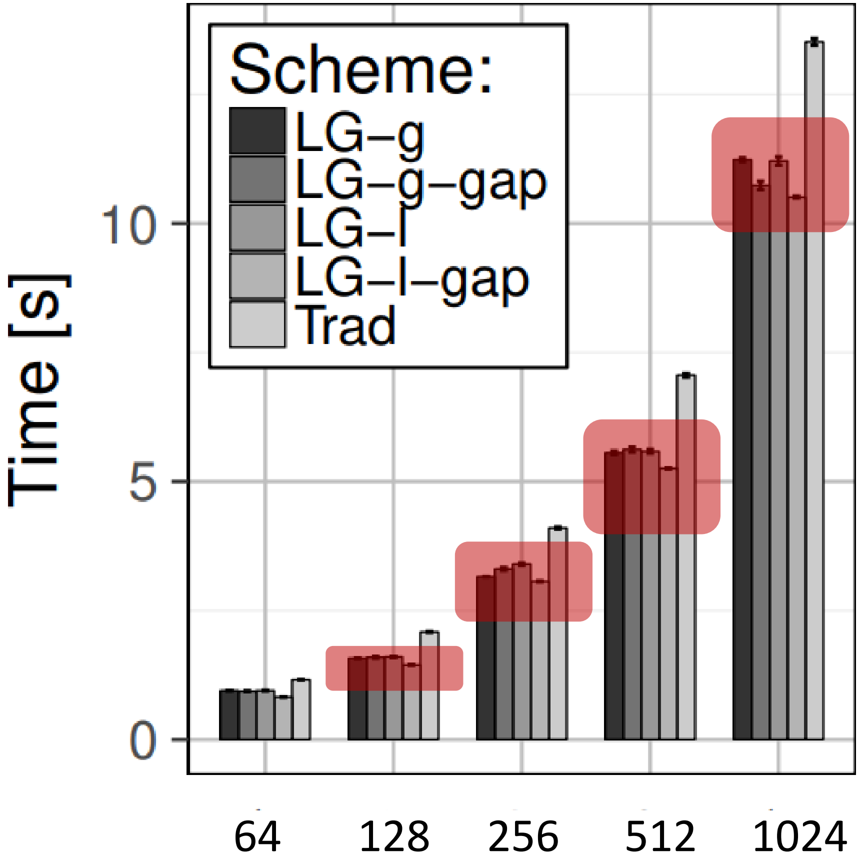
Both storage and performance are improved **simultaneously**

Sparse graphs

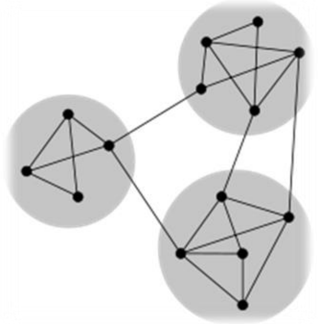


Number of edges per vertex:

Dense graphs



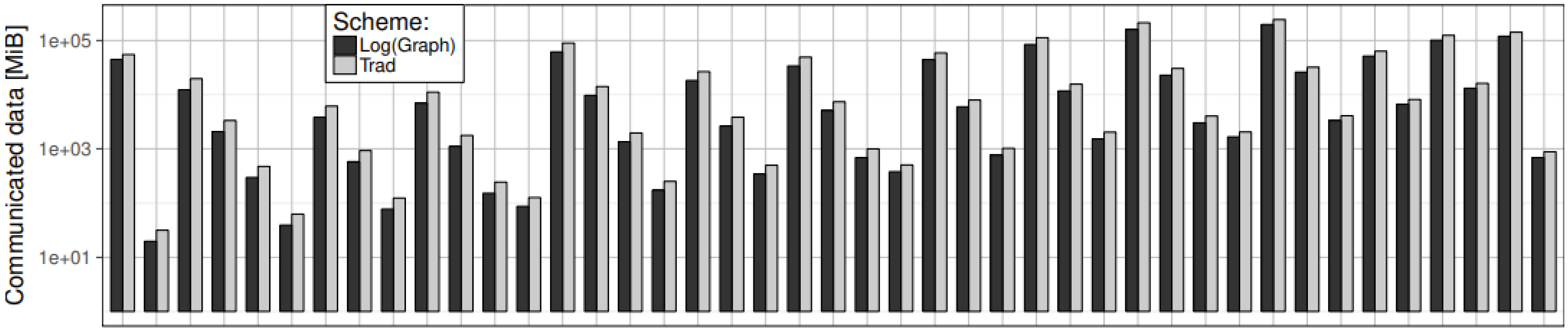
Number of edges per vertex:



Kronecker graphs
Number of vertices: 4M

1 **Log (Vertex labels), Log (Edge weights)** **Communicated data**

PageRank

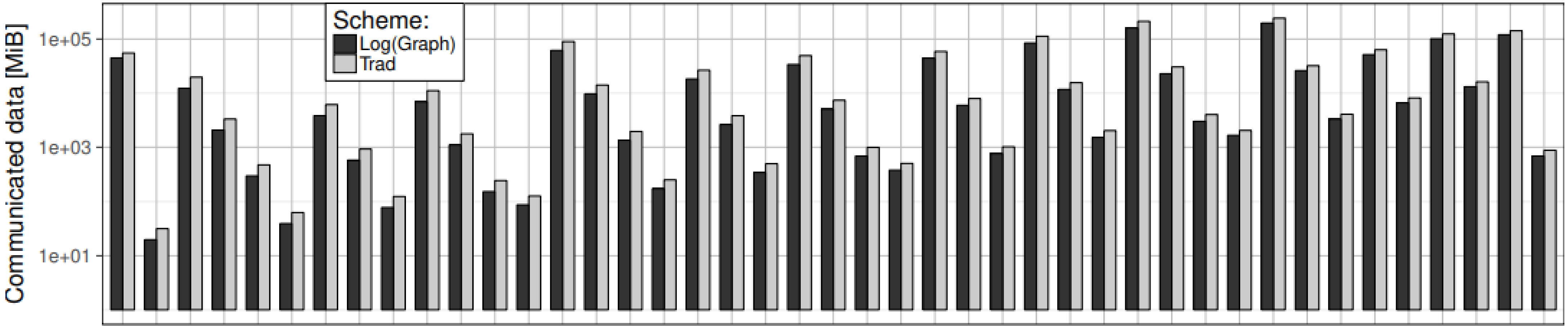


Various real-world and synthetic graphs

1024 compute nodes

PageRank

1 **Log (** Vertex labels **), Log (** Edge weights **)** **Communicated data**



Various real-world and synthetic graphs

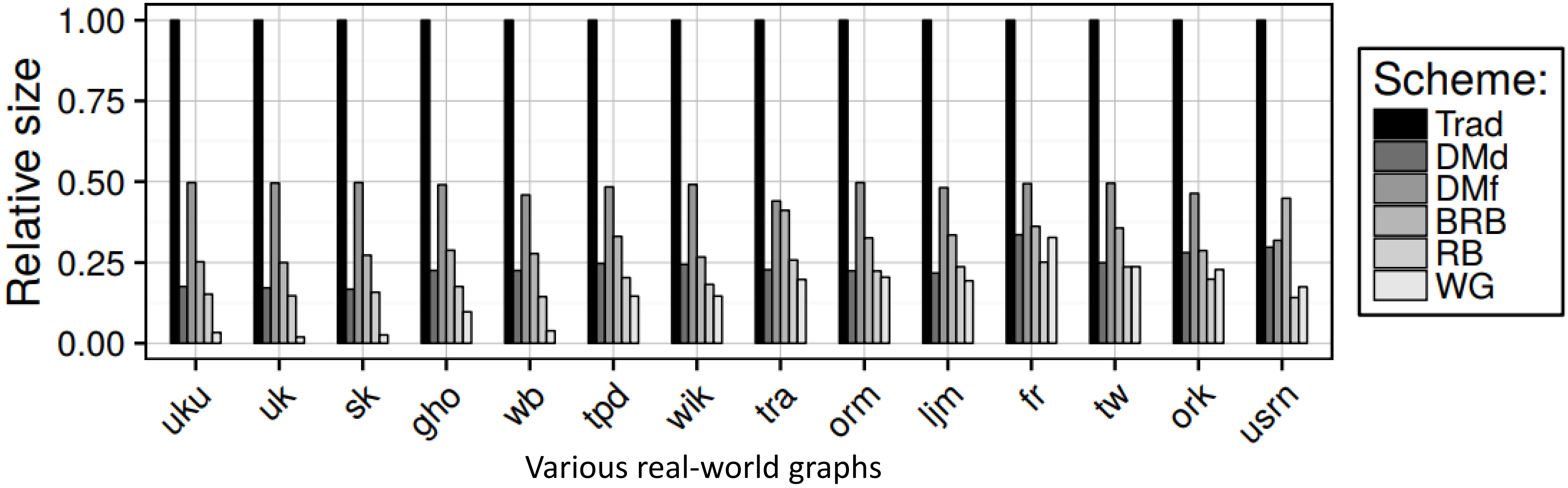
The amount of communicated data is consistently reduced by ~37%

1024 compute nodes

3 **Log (** Adjacency structure **)** **Storage**

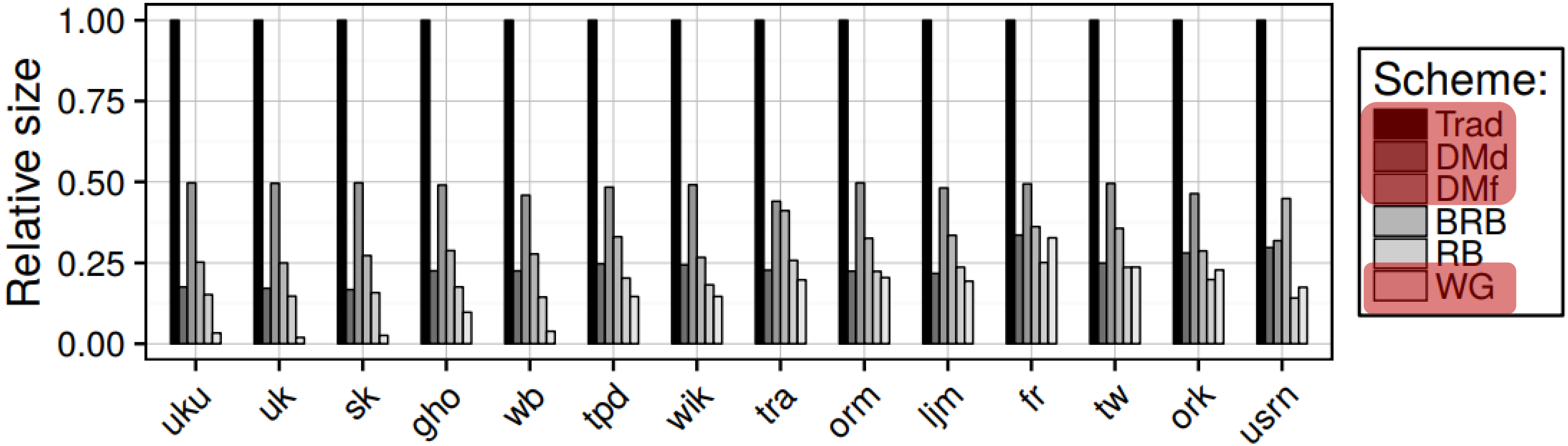
3 **Log (Adjacency structure) Storage**

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



3 Log (Adjacency structure) Storage

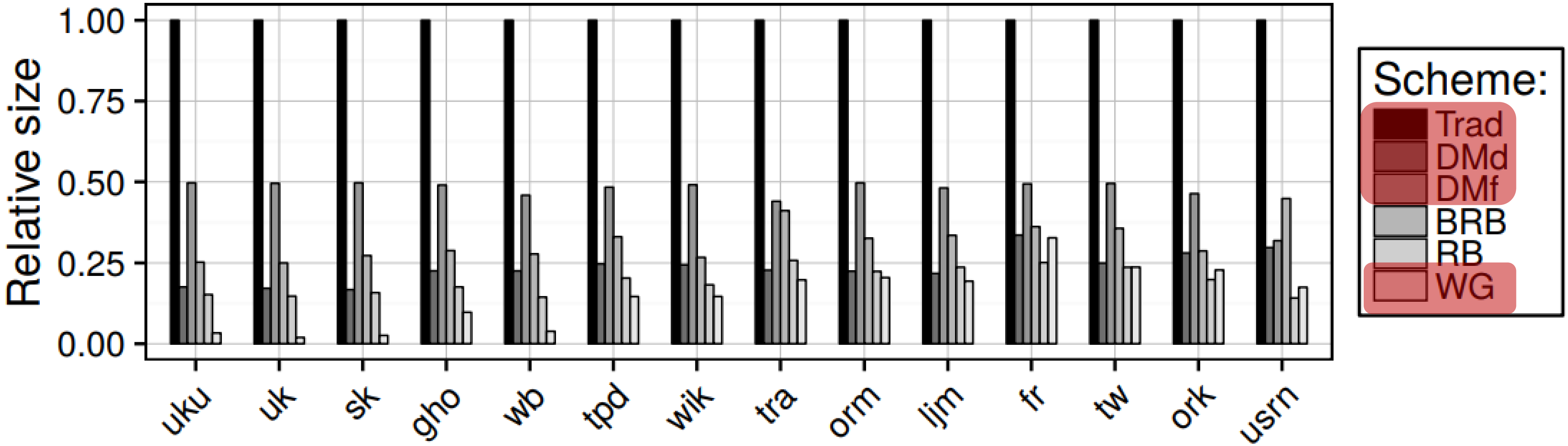
Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
BRB, RB: Schemes targeting certain specific classes of graphs



Various real-world graphs

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



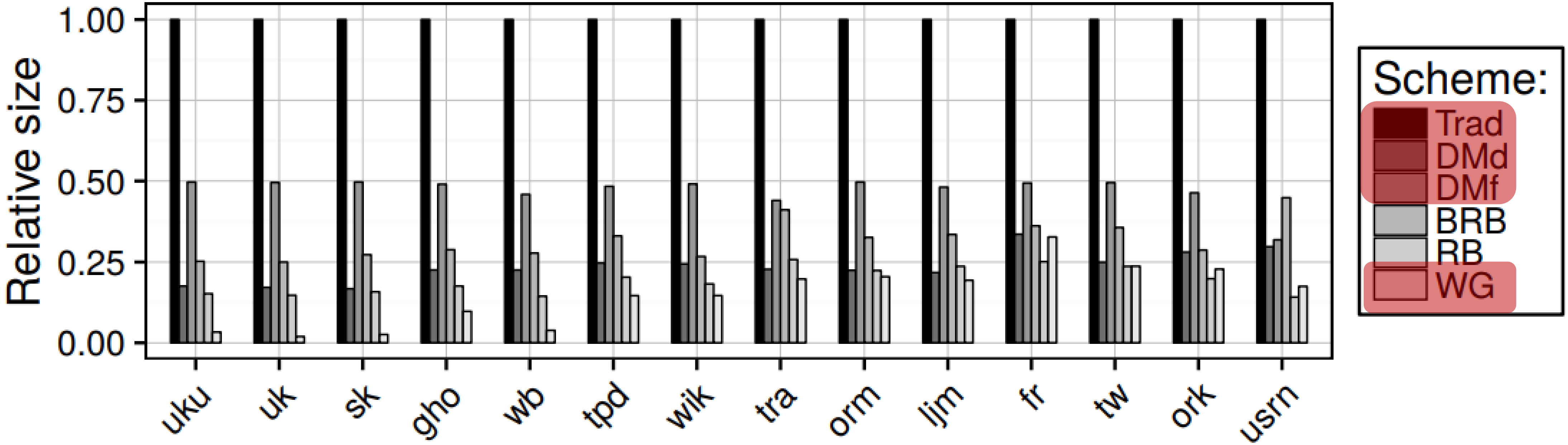
Various real-world graphs

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

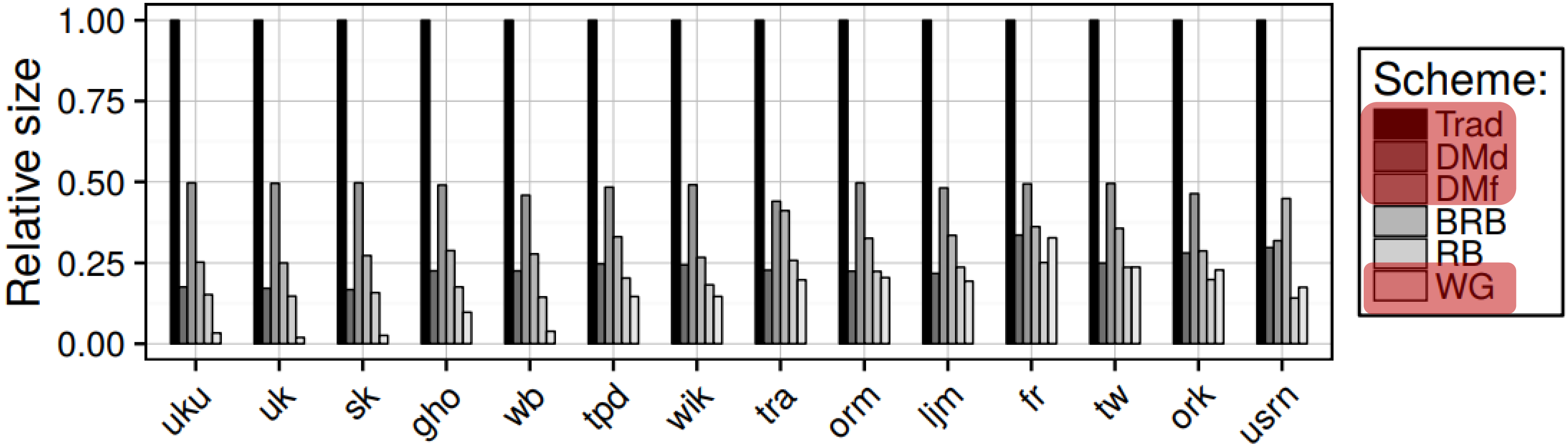


Lots of data ☺

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

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



Various real-world graphs

Lots of data ☺

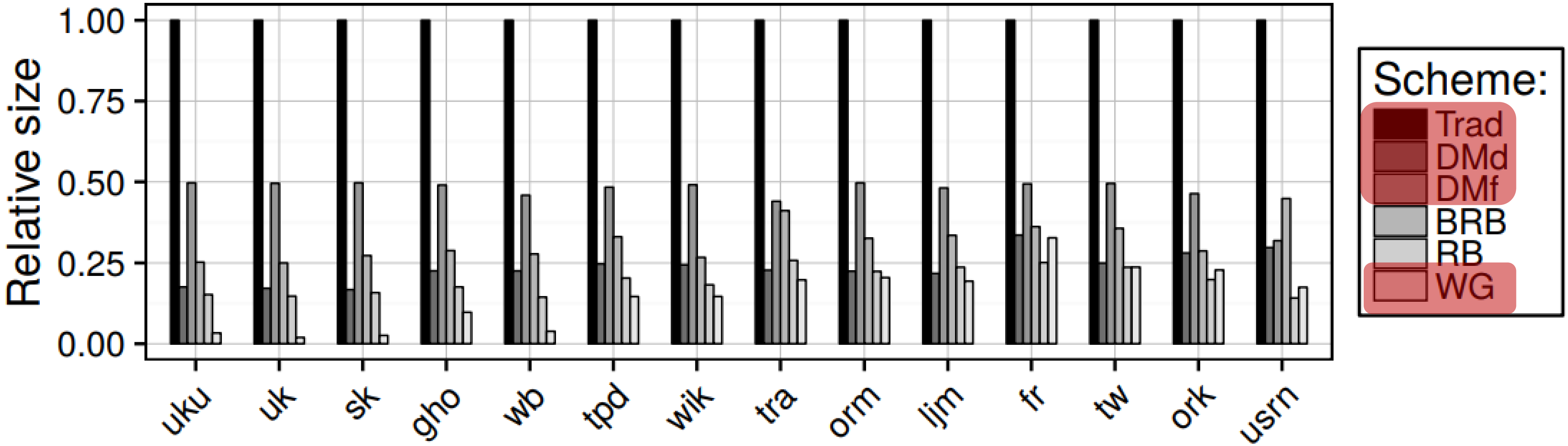
Conclusions:

WebGraph **best for web graphs** ☺

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

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



Various real-world graphs

Lots of data ☺

Conclusions:

WebGraph **best for web graphs ☺**

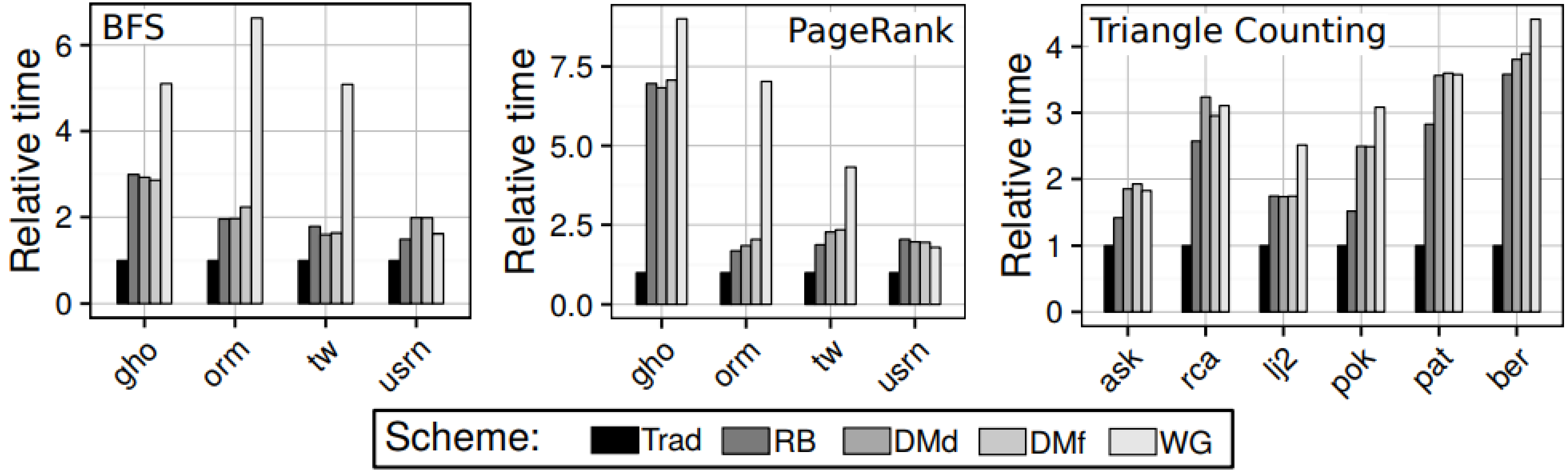
DMd: much better than DMf, often comparable to others

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

3 **Log (Adjacency structure) Performance**

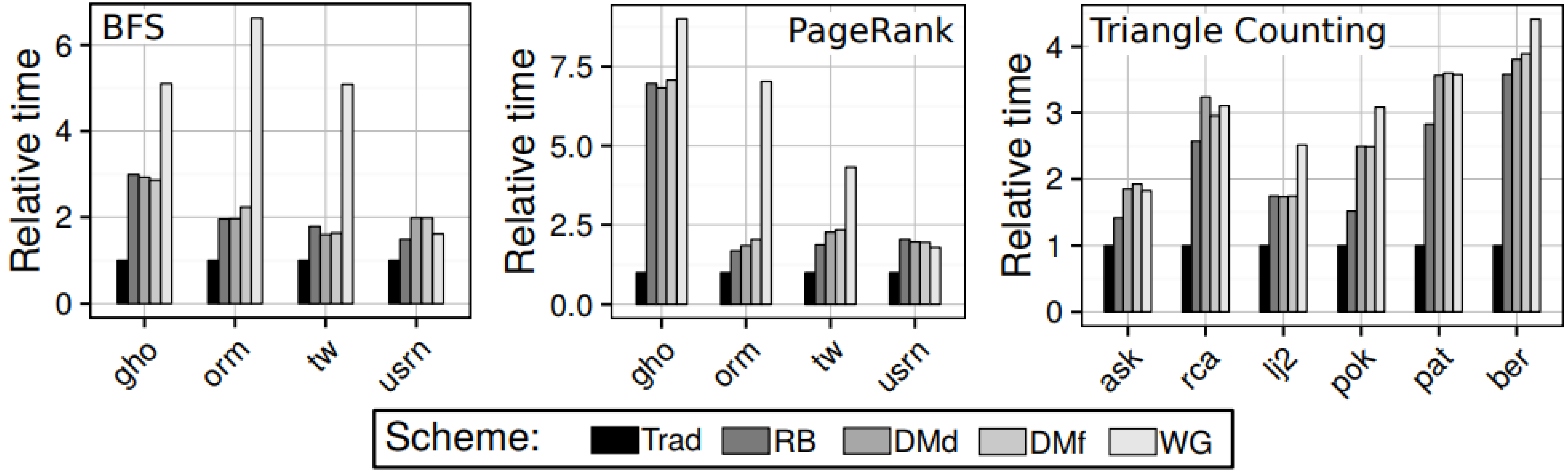
3 Log (Adjacency structure) Performance

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



3 Log (Adjacency structure) Performance

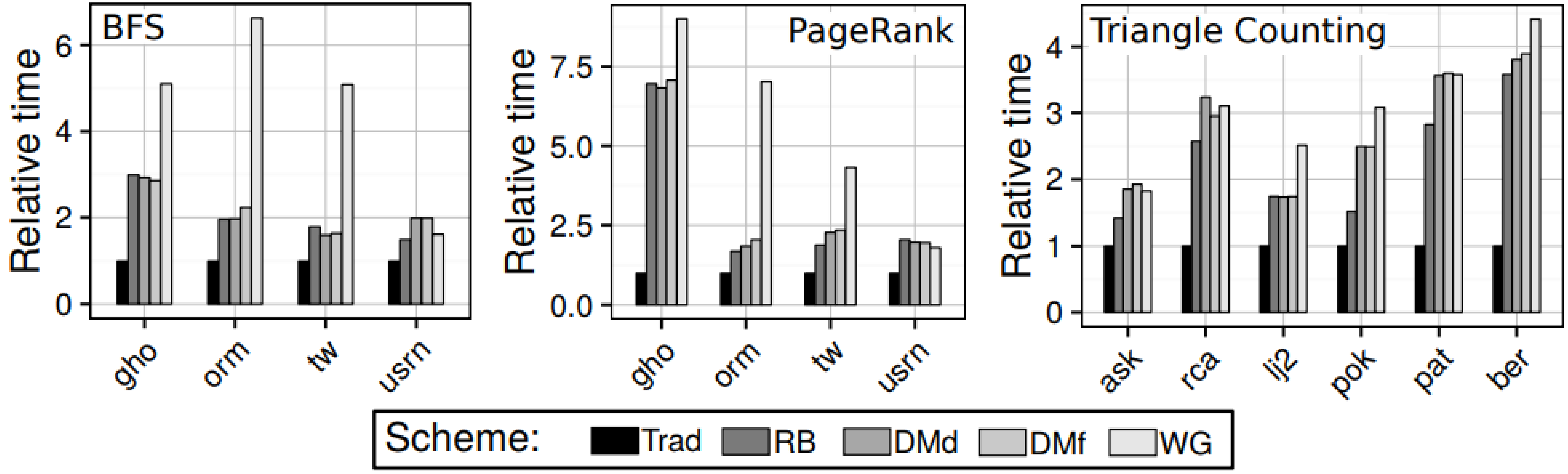
Trad: Traditional adjacency array
DMd / DMf: Degree Minimizing (without / with gap encoding)
WG: WebGraph compression
RB: Scheme targeting certain specific classes of graphs



WebGraph is the slowest

3 Log (Adjacency structure) Performance

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

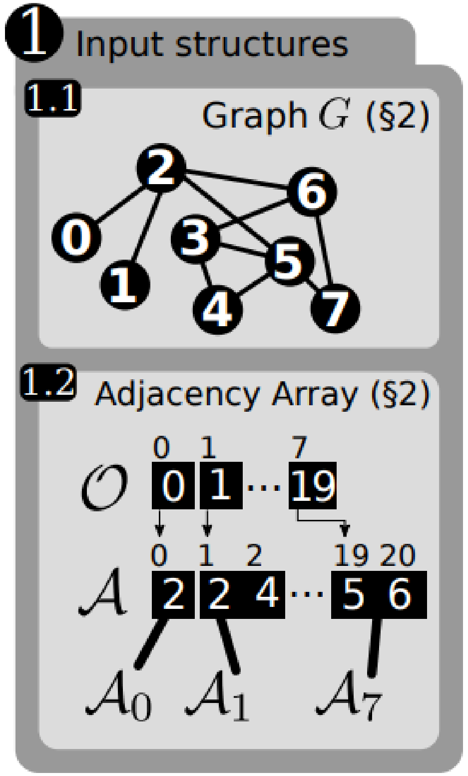


WebGraph is the slowest

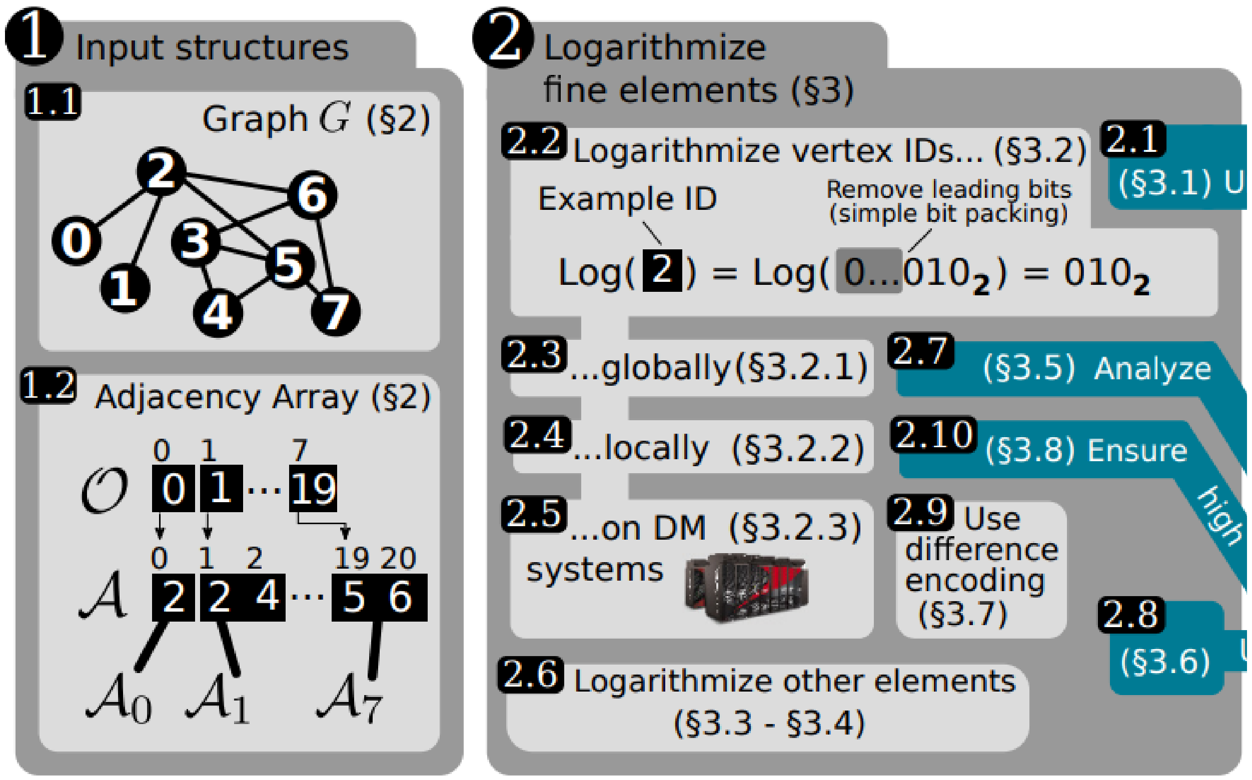
DM, RB: comparable

Log(Graph) full design…

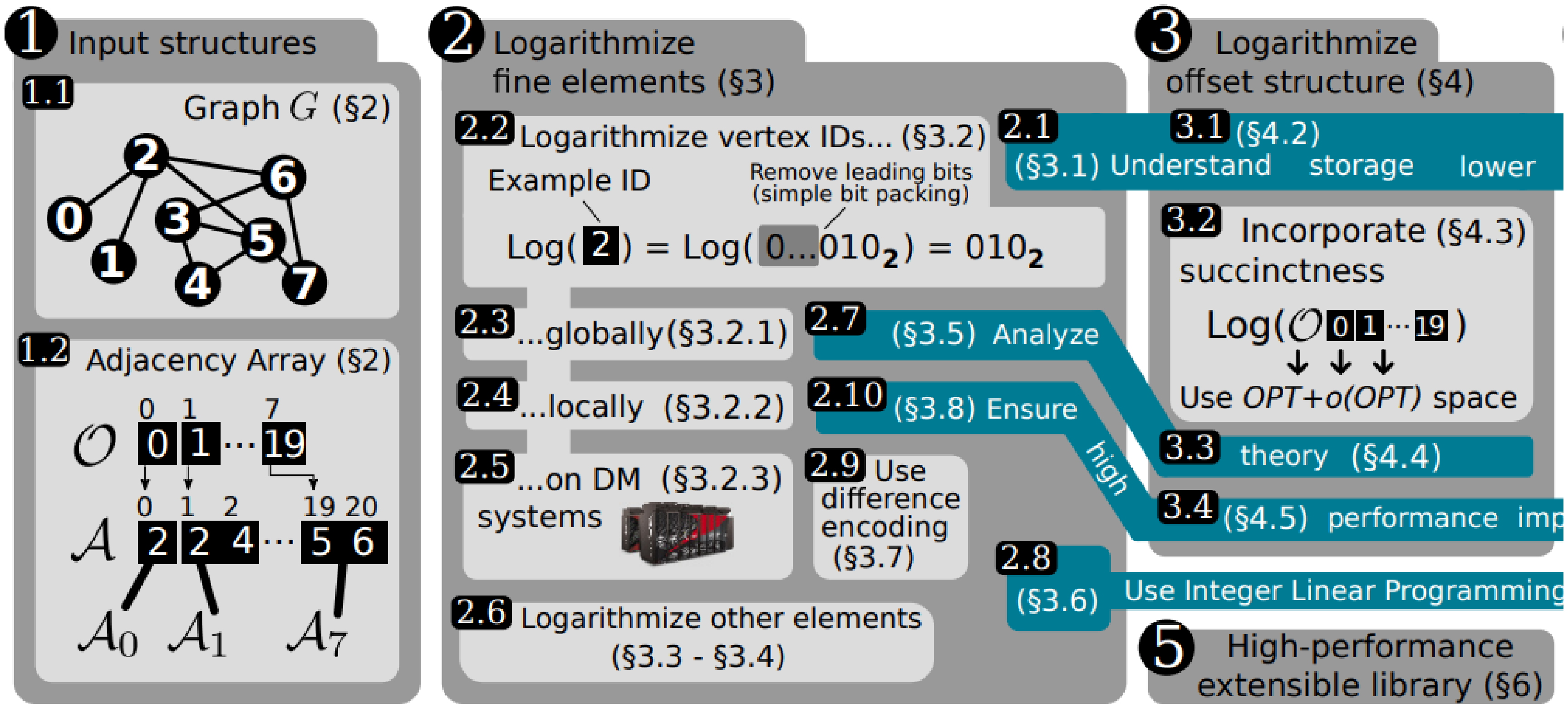
Log(Graph) full design…



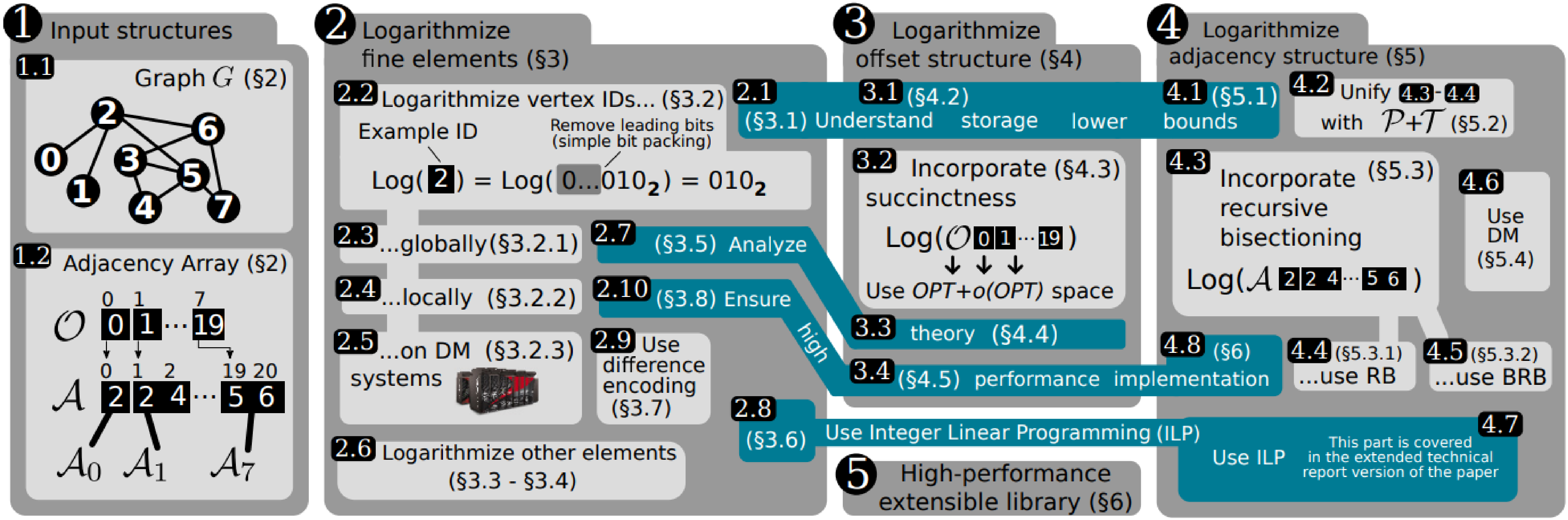
Log(Graph) full design...



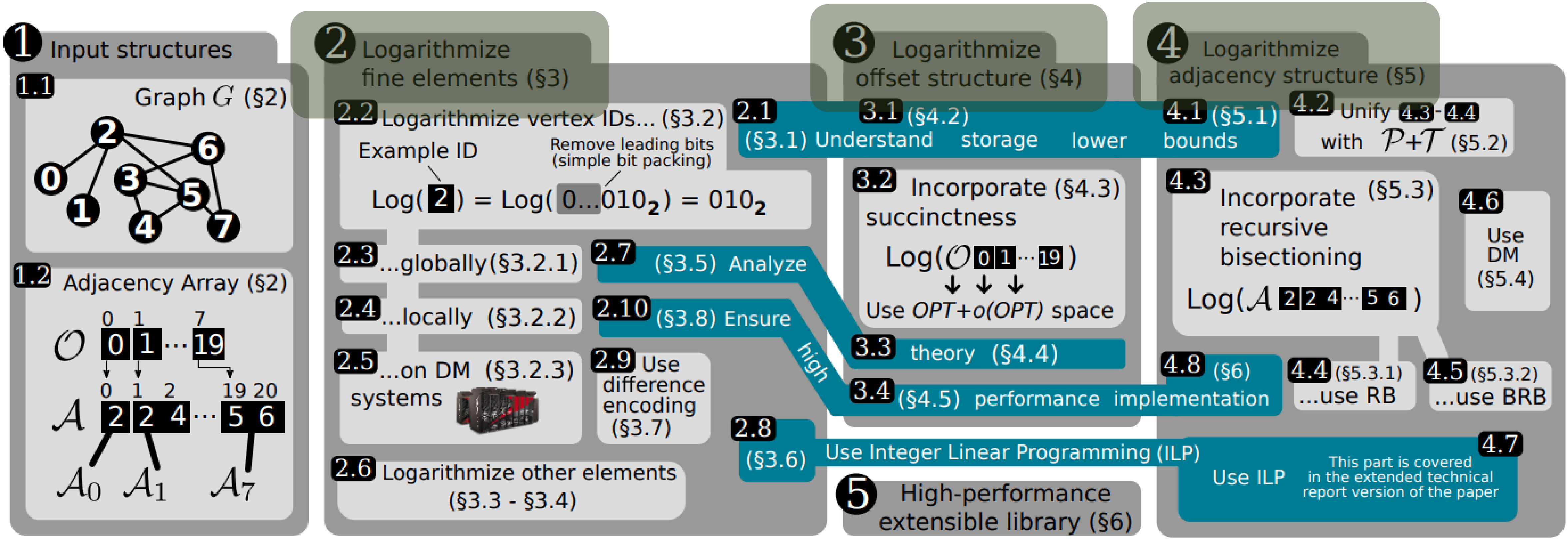
Log(Graph) full design…



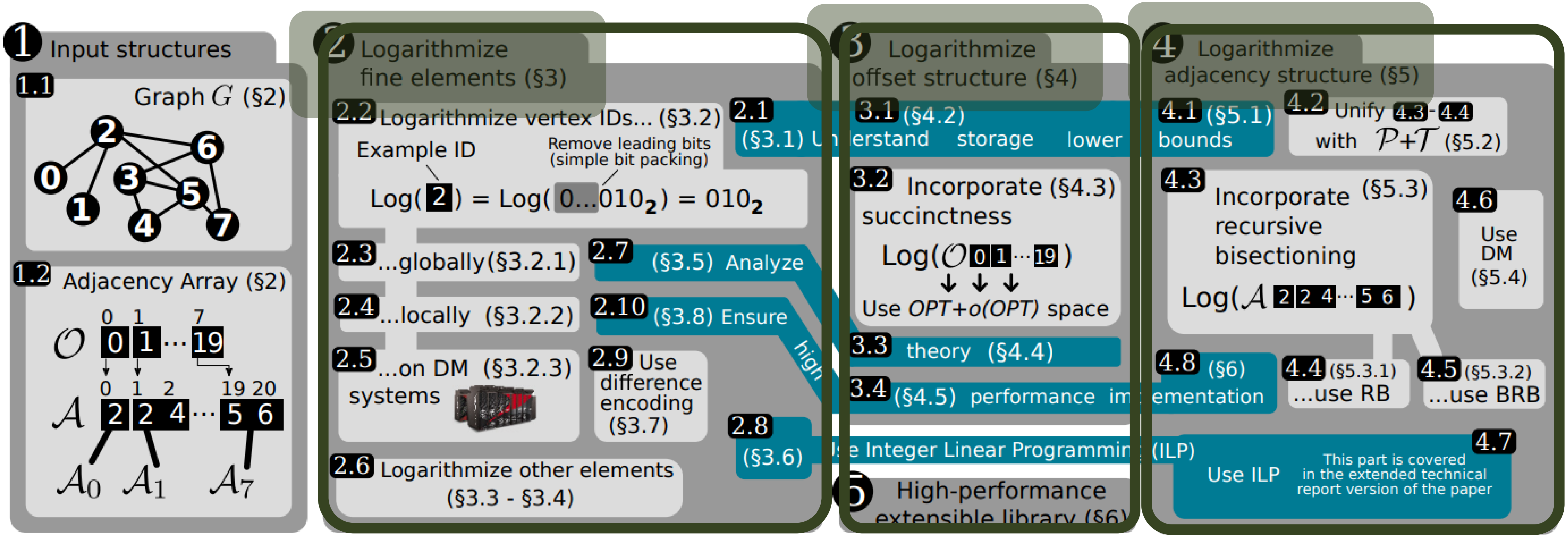
Log(Graph) full design…



Log(Graph) full design…

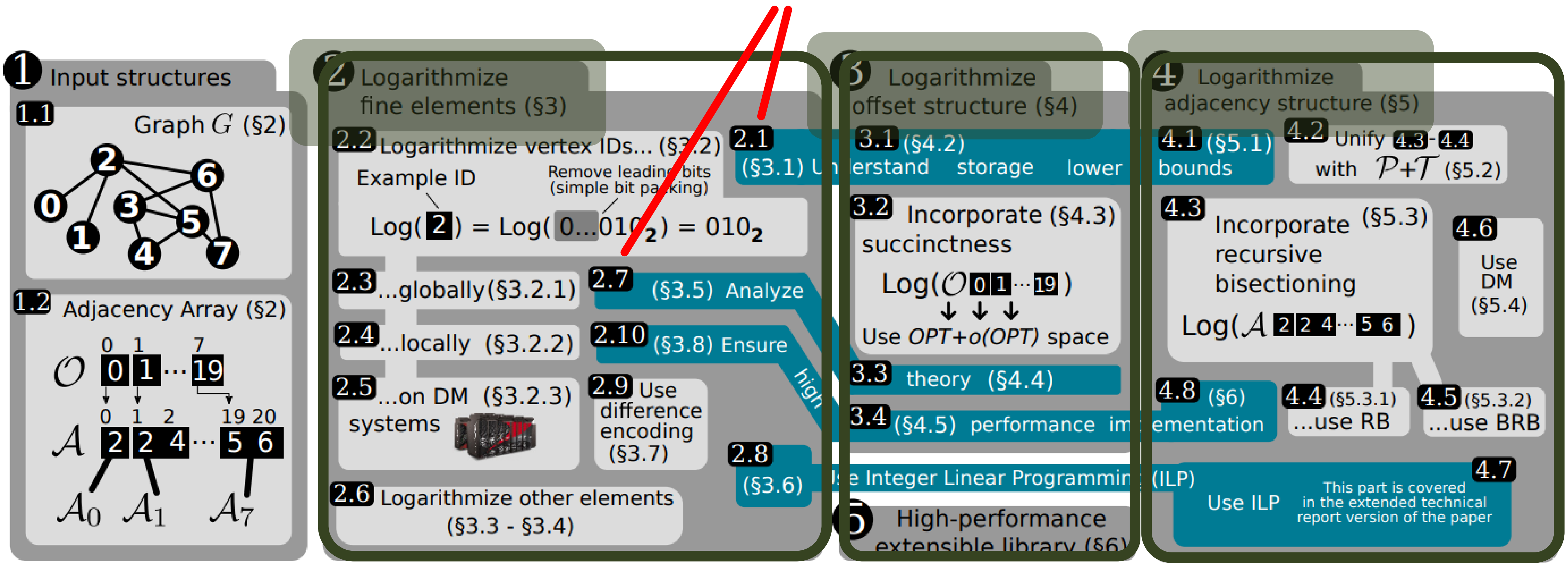


Log(Graph) full design…



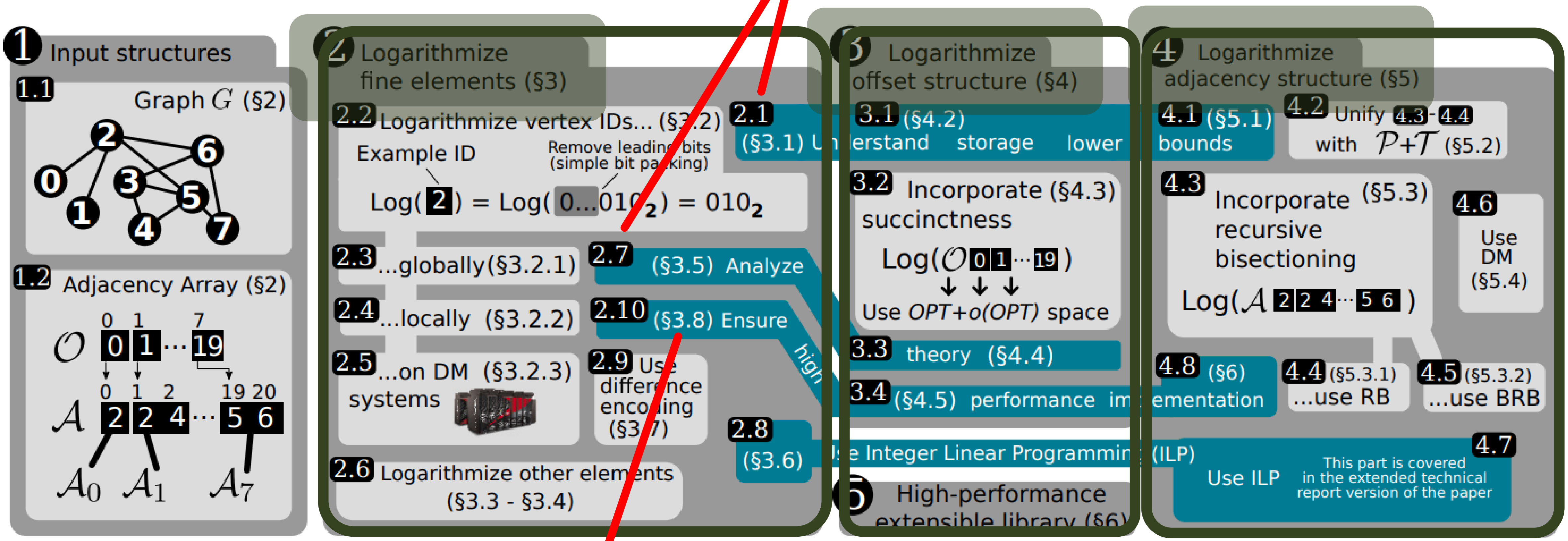
Log(Graph) full design…

Understand storage lower bounds and the theory



Log(Graph) full design…

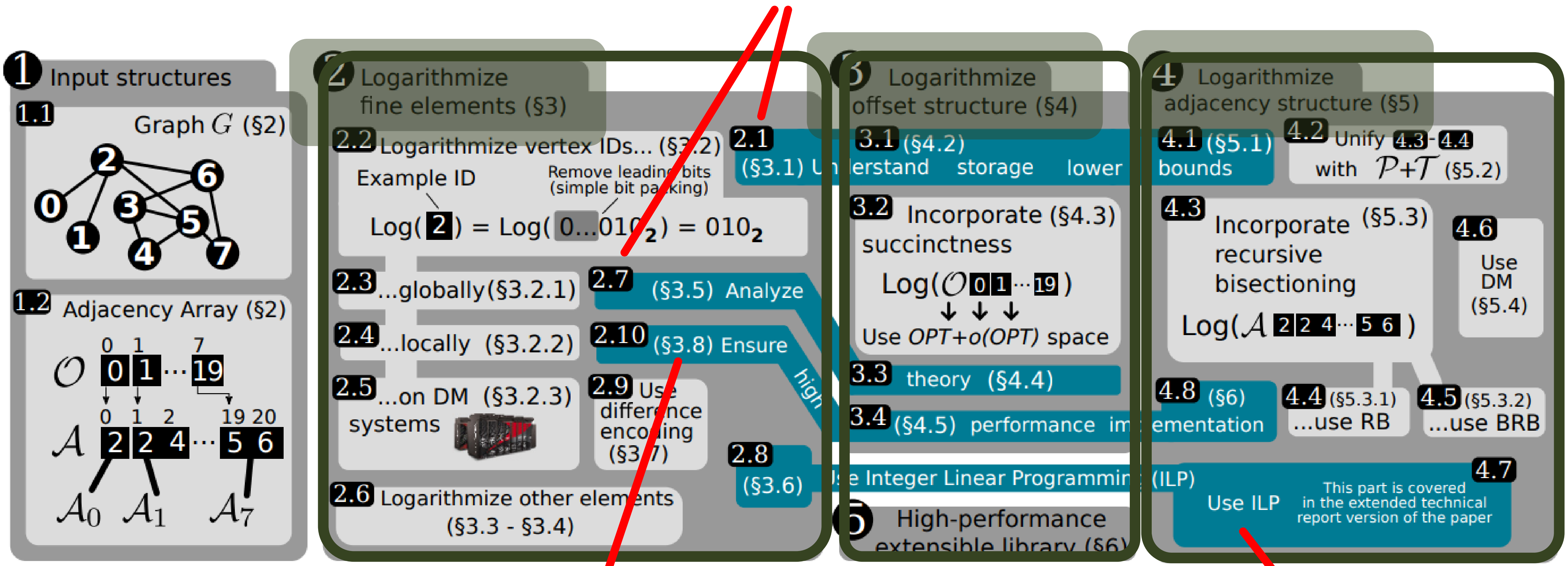
Understand storage lower bounds and the theory



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 $\lceil \log n \rceil$ 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 $\lceil \log n \rceil$ 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

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

spcl.inf.ethz.ch
@spcl_eth

Key method (vertex labels)

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

Modern bitwise operations

Key method (neighborhoods)

Key method (offsets)

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

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

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

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

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

C++ templates to reduce overheads in performance-critical kernels

Key method (offsets)

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

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