Graph Theory and Algorithms

Ph.D. Course – Marco Viviani

Graph Compression and Summarization
(April 29, 2021 / 15:00-17:00)

TABLE OF CONTENTS

1

Scenario

Intro, Basic Notions, and
Open Issues

From: Graph Compression – Data compression (Lecture 5)

Graph sizes in 2018

Graph	IVI	IEI (symmetrized)
com-Orkut	3M	234M
Twitter	41M	1.46B
Friendster	124M	3.61B
Hyperlink2012-Host	101M	2.04B
Facebook (2011) [1]	721M	68.4B
Hyperlink2014 [2]	1.7B	124B
Hyperlink2012 [2]	3.5B	225B
Facebook (2018)	> 2B	> 300B
Google (2018)	?	?

Publicly available graphs
Private graphs

[1] The Anatomy of the Facebook Social Graph, Ugander et al. 2011
[2] http://webdatacommons.org/hyperlinkgraph/

From: Graph Compression – Data compression (Lecture 5)

Graph compression in industry

NetflixGraph Metadata Library: An Optimization Case Study

by Drew Koszewnik

Problem: running into memory issues when storing the movie property graph in memory

Solution: Compact Encoded Data Representation

We knew that we could hold the same data in a more memory-efficient way. We created a library to represent directed-graph data, which we could then overlay with the specific schema we needed.

Results

When we dropped this new data structure in the existing NetflixGraph library, our memory footprint was reduced by 90% A histogram of our test application from above, loading the exact same set of data, now looks like the following:

Source: Netflix Tech Blog

From: Graph Compression – Data compression (Lecture 5)

Graph compression in industry

Compressing Graphs and Indexes with Recursive Graph Bisection

Abstract

Graph reordering is a powerful technique to increase the locality of the representations of graphs, which can be helpful in several applications. We study how the technique can be used to improve compression of graphs and inverted indexes.

Our experiments show a significant improvement of the compression rate of graph and indexes over existing heuristics. The new method is relatively simple and allows efficient parallel and distributed implementations, which is demonstrated on graphs with billions of vertices and hundreds of billions of edges.

Source: Facebook Research

From: Graph Compression – Data compression (Lecture 5)

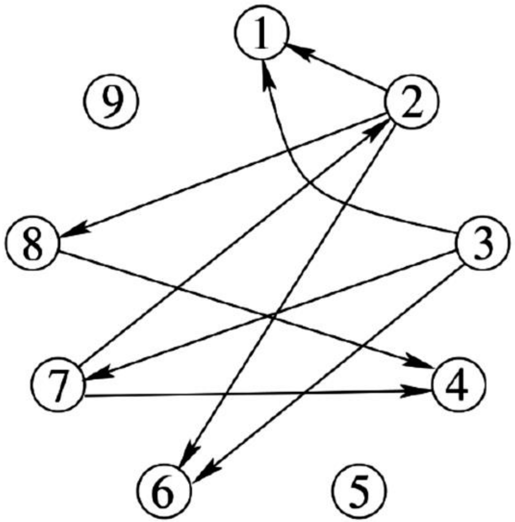
Operations on Graphs

- **Static graphs**:
  - Scanning the whole graph (i.e., the storage cost),
  - `get_neighbors(v)` (in/out neighbors for digraphs),
  - `is_edge(u,v)` (is the (u, v) edge present in G?).

- **Dynamic graphs**:
  - Insert/delete nodes/edges.

From: Compressed Sparse Row Format for Representing Graphs

Graph Representations



(a) example graph

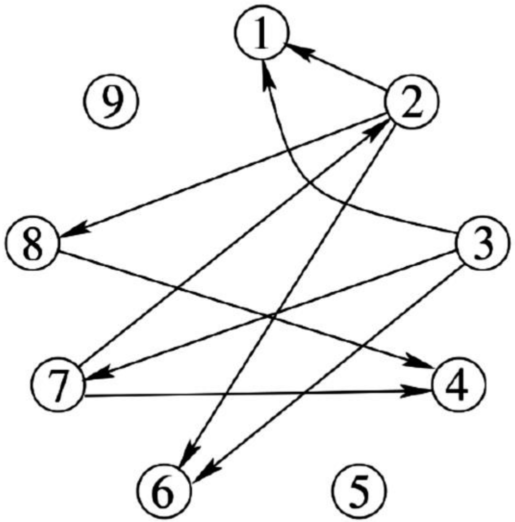
- **Edge List**

2	1
2	6
2	8
3	1
3	6
3	7
7	2
7	4
8	4

(b) edge list

From: Compressed Sparse Row Format for Representing Graphs

Graph Representations ... Cont'd



(a) example graph

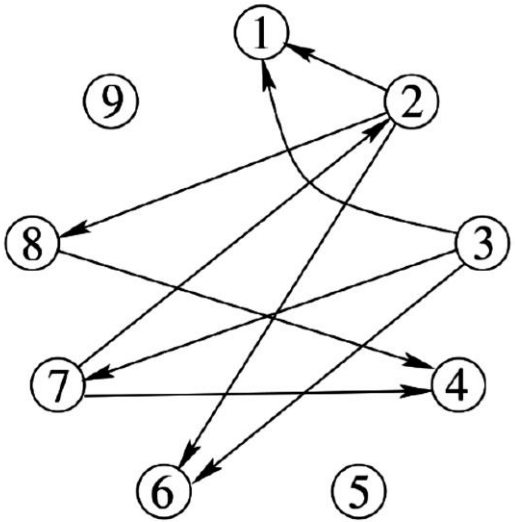
- **Adjacency Matrix**
  - Vertices labeled from 0 to $n-1$.
  - Entry of "1" if edge exists, "0" o.w. (or the weight on the edge).

	1	2	3	4	5	6	7	8	9
1									
2	1					1		1	
3	1					1	1		
4									
5									
6									
7		1		1					
8				1					
9									

(c) adjacency matrix

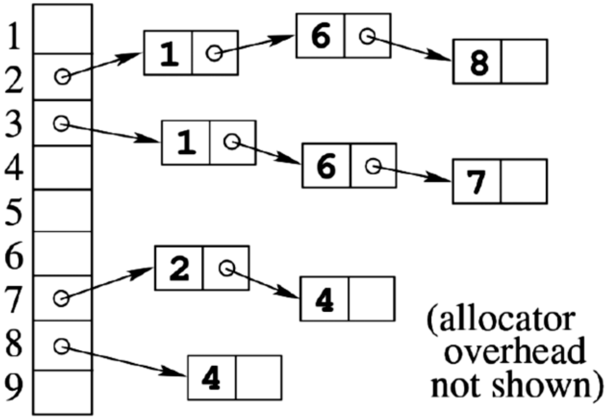
From: [Compressed Sparse Row Format for Representing Graphs]()

Graph Representations … Cont'd



(a) example graph

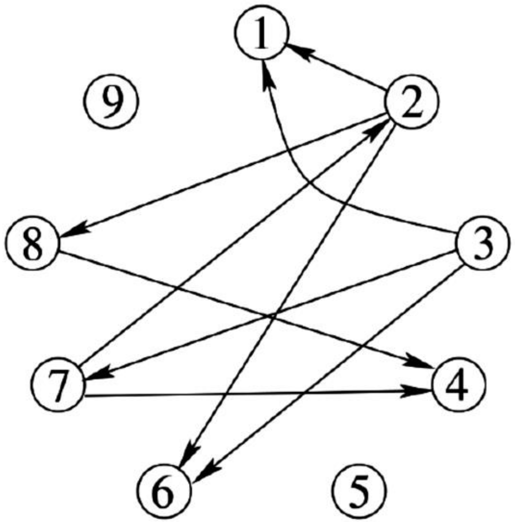
- **Adjacency List**
  - Array of pointers (one per vertex).
  - Each vertex points to a list of its neighbors.



(d) adjacency lists

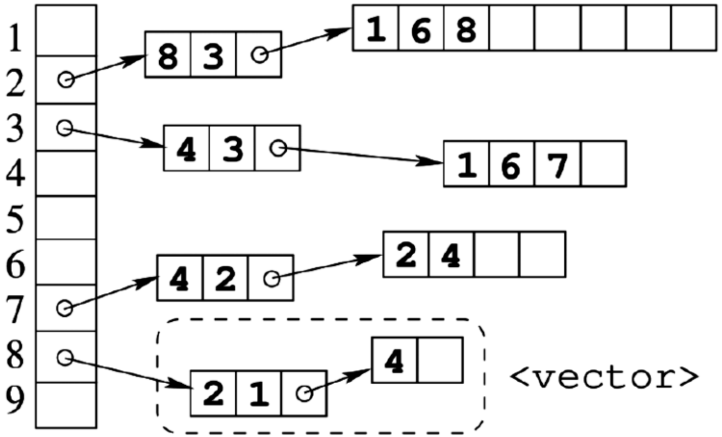
From: Compressed Sparse Row Format for Representing Graphs

Graph Representations … Cont'd



(a) example graph

- **Adjacency Vectors**
  - An array indexed by "from" vertexID contains entry points to <vector>s of "to" vertexIDs.



(e) adjacency vectors

From: Graph Compression – Data compression (Lecture 5)

Computational Costs – Time

Operation	Adjacency Matrix	Edge List	Adjacency List
scan_graph	$O(n^2)$	$O(m)$	$O(m+n)$
get_neighbors	$O(n)$	$O(m)$	$O(d)$
is_edge	$O(1)$	$O(m)$	$O(d)$
insert edge	$O(1)$	$O(1)$	$O(1)$ or $O(d)$
delete edge	$O(1)$	$O(m)$	$O(d)$

From: Graph Compression – Data compression (Lecture 5)

Computational Costs – Space

Hyperlink2012 Graph

- n = 3.6B, m = 225B (undirected edges)
- Vertex ids fit into 4 bytes
- > 900Gb to store in CSR format

We are going to detail it later

32Gb DRAM: about 300$*

So, about 9000$ of memory just to store the graph.
Doesn't include memory needed to run algorithms on it!

*Source: Hynix HMA84GR7MFR4N-UH 32GB DDR4-2400 ECC REG DIMM Server Memory

Compression VS Summarization

- **Graph compression** applies various encoding techniques so that the resultant graph needs lesser **storage space**.
- **Graph summarization** aggregates nodes having similar structural properties/patterns to represent a graph with reduced main **memory requirements**.

Seo, H., Park, K., Han, Y., Kim, H., Umair, M., Khan, K. U., & Lee, Y. K. (2018). An effective graph summarization and compression technique for a large-scaled graph. The Journal of Supercomputing, 1-15

2

Graph Compression

Intro and Some
Compression Models

Graph Compression

- Aim: **storage-efficient processing of large graphs**
  - This is becoming increasingly important w.r.t. Big Data Analysis
- Many different **areas of application**:
  - Web graphs
  - Biology networks
  - Social graphs
  - …
- Many of these approaches originated in the area of **data compression** and **high-performance scientific computing**.

The Compressed Sparse Row (CSR) Representation

- The **Compressed Sparse Row (CSR)** Representation originated in high-performance scientific computing as a way to represent **sparse matrices**, whose rows contain mostly zeros.

- "Old" representation
  - Appeared in the mid-60.

- The basic idea is to pack the column indices of non-zero entries into a dense array → How?

The Compressed Sparse Row (CSR) Representation … Cont'd

Advantage:

- CSR is **more compact and is laid out more contiguously in memory** than adjacency lists and adjacency `<vector>`s, eliminating nearly all space overheads and reducing random memory accesses compared with these other formats.

Disdvantage:

- The price we pay for CSR's advantages is **reduced flexibility**: adding new edges to a graph in CSR format is <u>inefficient</u>, so CSR is <u>suitable for graphs whose structure is fixed</u> and given all at once.

From: [Compressed Sparse Row Format for Representing Graphs]

The Compressed Sparse Row (CSR) Representation ... Cont'd

- The Compressed Sparse Row represents an mxn matrix M by **three (one-dimensional) arrays**, that respectively contain non-zero values, the row pointers, and column indices.

- The arrays V and C_{Index} are of length nnz, and contain the **non-zero values** and the **column indices** of those values respectively.
  - nnz denotes the number of nonzero entries in M.

- The array R_{Index} **encodes** the index in V and C_{Index} where the given row starts. Its length is $m + 1$.

The Compressed Sparse Row (CSR) Representation ... Cont'd

- The R_{Index} vector stores the **cumulative number of non-zero elements** upto (not including) the i-th row.

- It is defined by the **recursive relation**:
  - $R_{Index}[0] = 0$.
  - $R_{Index}[i] = R_{Index}[i-1]$ + number of non-zero elements in the $(i-1)$th row of the matrix.

- To find the number of non-zero elements in say row i, we perform:
  - $R_{Index}[i+1] - R_{Index}[i]$.

The Compressed Sparse Row (CSR) Representation – Example 1

For **example**, the matrix

0	0	0	0
5	8	0	0
0	0	3	0
0	4	0	0

is a 4×4 matrix with 4 nonzero elements, hence:

- $V = \quad (5 \;\; 8 \;\; 3 \;\; 4)$
- $C_{Index} = (0 \;\; 1 \;\; 2 \;\; 1)$

- What about the R_{Index}?

The Compressed Sparse Row (CSR) Representation – Example 1 ... Cont'd

For **example**, the matrix

0	0	0	0
5	8	0	0
0	0	3	0
0	4	0	0

- $R_{Index}[0] = 0$
- $R_{Index}[1] = R_{Index}[0]$ + n. of non-zero elements in row 0, i.e, 0 + 0 = 0.
- $R_{Index}[2] = R_{Index}[1] + 2 = 2$
- $R_{Index}[3] = R_{Index}[2] + 1 = 3$
- $R_{Index}[4] = R_{Index}[3] + 1 = 4$
- Therefore, $R_{Index} = (0\ 0\ 2\ 3\ 4)$

The Compressed Sparse Row (CSR) Representation – Example 1 … Cont'd

For **example**, the matrix

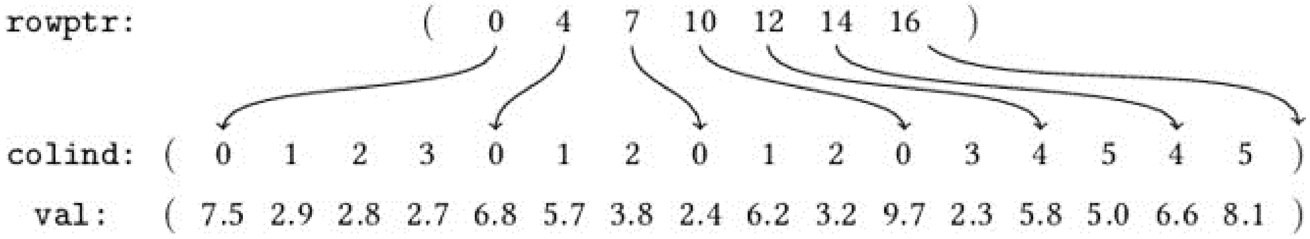
0	0	0	0
5	8	0	0
0	0	3	0
0	4	0	0

has the following **CSR representation**:

- $V = \quad (5 \ \ 8 \ \ 3 \ \ 4)$
- $C_{Index} = (0 \ \ 1 \ \ 2 \ \ 1)$
- $R_{Index} = (0 \ \ 0 \ \ 2 \ \ 3 \ \ 4)$

The Compressed Sparse Row (CSR) Representation – Example 2

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$



Elafrou, A., Goumas, G., & Koziris, N. (2015). A lightweight optimization selection method for Sparse Matrix-Vector Multiplication. arXiv preprint arXiv:1511.02494

The Compressed Sparse Row (CSR) Representation – Exercise

Given the following matrix

0	2	0	0	1	0	0	0
0	8	0	0	0	0	2	0
2	0	3	0	1	1	0	0
0	0	0	0	0	0	8	0

has the following **CSR representation**:

- $|V| = ?$
- $|C_{Index}| = ?$
- $|R_{Index}| = ?$

$V = ?$

$C_{Index} = ?$

$R_{Index} = ?$

The Compressed Sparse Column (CSC) Representation

- **Compressed Sparse Column (CSC)** is similar to CSR except that:
  - values are read first by column,
  - a row index is stored for each value,
  - column pointers are stored.

From: Graph Compression – Data compression (Lecture 5)

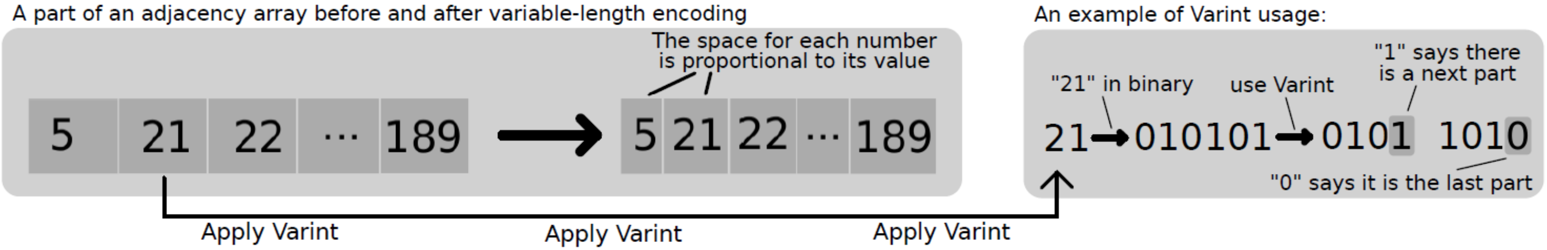
Computational Costs – Time

Operation	Adjacency Matrix	Edge List	Adjacency List	CSR/CSC
scan_graph	$O(n^2)$	$O(m)$	$O(m+n)$	$O(m+n)$
get_neighbors	$O(n)$	$O(m)$	$O(d)$	$O(d)$
is_edge	$O(1)$	$O(m)$	$O(d)$	$O(d)$
insert edge	$O(1)$	$O(1)$	$O(1)$ or $O(d)$	$O(m+n)$
delete edge	$O(1)$	$O(m)$	$O(d)$	$O(m+n)$

From: arXiv preprint arXiv:1806.01799

Variable-Length Encoding

- In **variable-length encoding**, vertex IDs stored in the adjacency array are encoded with one of the selected **variable-length codes***, such as **variable length integer** (varint) coding → details



*In coding theory, a **variable-length code** is a code which maps source symbols to a **variable** number of bits.

From: arXiv preprint arXiv:1806.01799

Huffman Degree Encoding

- The core idea in the **Huffman degree encoding** scheme is to use fewer bits to encode vertex IDs of higher degrees.
  - Vertex IDs that occur more often use fewer bits → saving space.

Huffman Degree Encoding ... Cont'd

standard ASCII table.

char	ASCII	bit pattern (binary)
h	104	01101000
a	97	01100001
p	112	01110000
y	121	01111001
i	105	01101001
o	111	01101111
space	32	00100000

The string **"happy hip hop"** would be encoded in ASCII as **104 97 112 112 121 32 104 105 112 32 104 111 112**. Although not easily readable by humans, it would be written as the following stream of bits (each byte is boxed to show the boundaries):

01101000	01100001	01110000	01110000	01111001	00100000	01101000
01101001	01110000	00100000	01101000	01101111	01110000	

Huffman Degree Encoding … Cont'd

optimal Huffman encoding for the string **"happy hip hop"**:

char	bit pattern
h	01
a	000
p	10
y	1111
i	001
o	1110
space	110

Each character has a unique bit pattern encoding, but not all characters use the same number of bits. The string **"happy hip hop"** encoded using the above variable-length code table is:

01	000	10	10	1111	110	01	001	10	110	01	1110	10

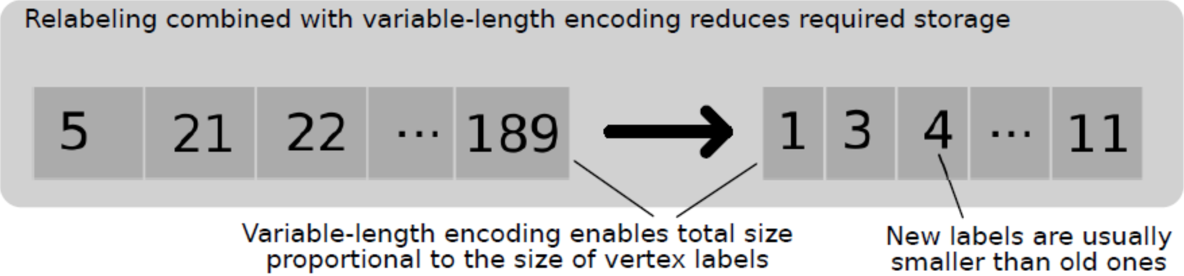
The encoded phrase requires a total of 34 bits, shaving a few more bits from the fixed-length version.

From: arXiv preprint arXiv:1806.01799

Vertex Relabeling

- In **vertex relabeling**, the main idea is to change the initial IDs of vertices so that the new IDs, when stored, use less space. We also use the name **vertex permutations** to refer to this technique.

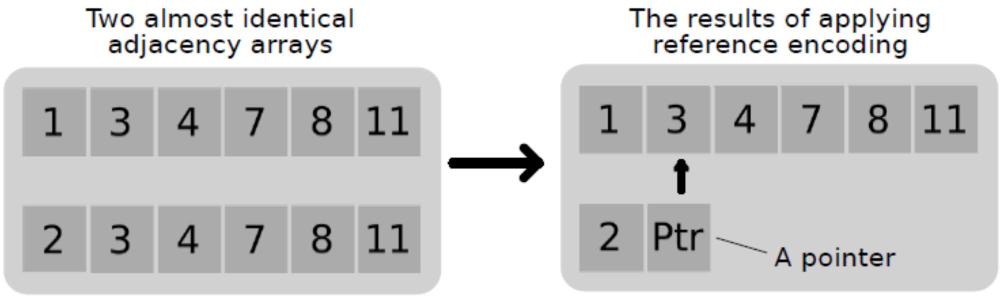
- This scheme is usually combined with variable-length encoding.



From: arXiv preprint arXiv:1806.01799

Reference Encoding

- In **reference encoding**, identical sequences of vertices in the adjacency arrays of different vertices are identified.
- Then, all such sequences (except for a selected one) are encoded with references.



From: arXiv preprint arXiv:1806.01799

Gap Encoding

- The **gap encoding** scheme preserves differences between vertex IDs rather than the IDs themselves.
  - The motivation is that, in most cases, differences occupy less space than IDs.

- **Several variants** can be used:
  - The most popular is storing differences between the IDs of the consecutive neighbors of each vertex v, for example:

$$N_1(v) - v, N_2(v) - N_1(v), \dots, N_{d_v-1}(v) - N_{d_v-2}(v), N_{d_v}(v) - N_{d_v-1}(v)$$

    (the first of the above differences is sometimes called an initial distance and each following: an increment).
  - Another variant stores the differences between v and each of its neighbors:

$$N_1(v) - v, N_2(v) - v, \dots, N_{d_v-1}(v) - v, N_{d_v}(v) - v$$

Gap Encoding … Cont'd

- **Original representation**:

1	→	3	5	8	44	88	120

- **Gap encoding**:

1	→	2	2	3	36	44	32

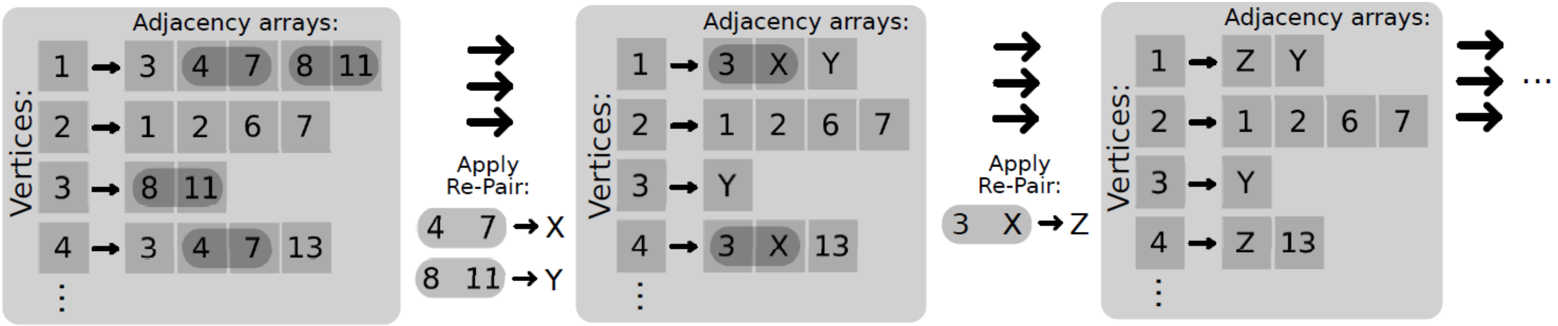
Re-Pair (Claude and Navarro, 2007)

- In the context of Web Graph compression, the authors propose a **text-based approach for graph compression**.
- A **phrase-based compression scheme** that enables fast decompression that is also local:
  - It does not always require accessing the whole graph.
- Re-Pair repeatedly **finds the most frequent pairs of symbols in a given graph representation** and replaces them with new symbols.
- This is repeated as long as storage is reduced.

Claude, F., & Navarro, G. (2007, October). A fast and compact Web graph representation. In International Symposium on String Processing and Information Retrieval (pp. 118-129). Springer, Berlin, Heidelberg

From: arXiv preprint arXiv:1806.01799

Re-Pair (Claude and Navarro, 2007) ... Cont'd



From: arXiv preprint arXiv:1806.01799

k^2 Trees (Brisaboa et al., 2014)

- A graph representation model where **a graph is modeled with a tree**.
- Initially, the graph is divided into k^2 submatrices of identical size (k is a parameter); these submatrices are recursively divided in the same way.
- Now, the key idea is to represent the graph as a k^2-ary tree (called a k^2 tree) that corresponds to the above recursive "partitioning" of the graph.
- At every partitioning level, if a given submatrix to be partitioned contains only 0s, the corresponding tree node contains 0. Otherwise, it contains a 1.

Brisaboa, N. R., Ladra, S., & Navarro, G. (2014). Compact representation of Web graphs with extended functionality. Information Systems, 39, 152-174

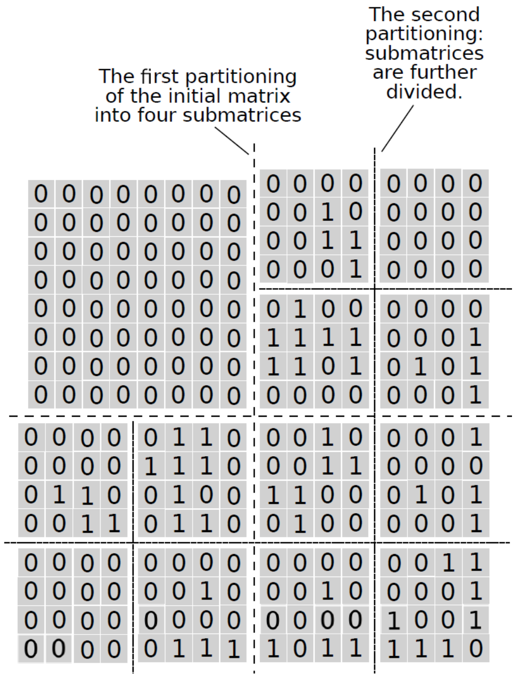
From: arXiv preprint arXiv:1806.01799

k^2 Trees (Brisaboa et al., 2014) ... Cont'd

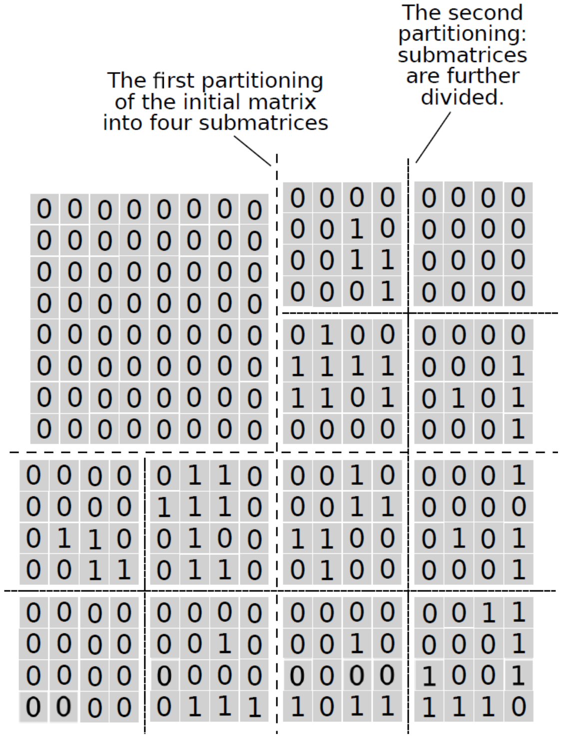
An adjacency matrix of some graph:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1
0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1
0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0
0	1	1	0	0	1	0	0	1	1	0	0	0	1	0	1
0	0	1	1	0	1	1	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0

Partition the matrix for k=2, using two partitoning levels (more levels could compress the matrix better; we use two for simplicity of the example)



k^2 Trees (Brisaboa et al., 2014) ... Cont'd



Generate the corresponding output tree, encoded with any space-efficient scheme