SiVaC*: An Efficient Graph Compression Algorithm

Panagiotis Liakos
National and Kapodistrian University of Athens, Greece
p.liakos@di.uoa.gr

Katia Papakonstantinopoulou
National and Kapodistrian University of Athens, Greece
katia@di.uoa.gr

Michael Sioutis
Pierre and Marie Curie University, France
michael.sioutis@lip6.fr

Introduction

• Many critical applications today run on web-like graphs (web itself, social networks, citation graphs, etc.).

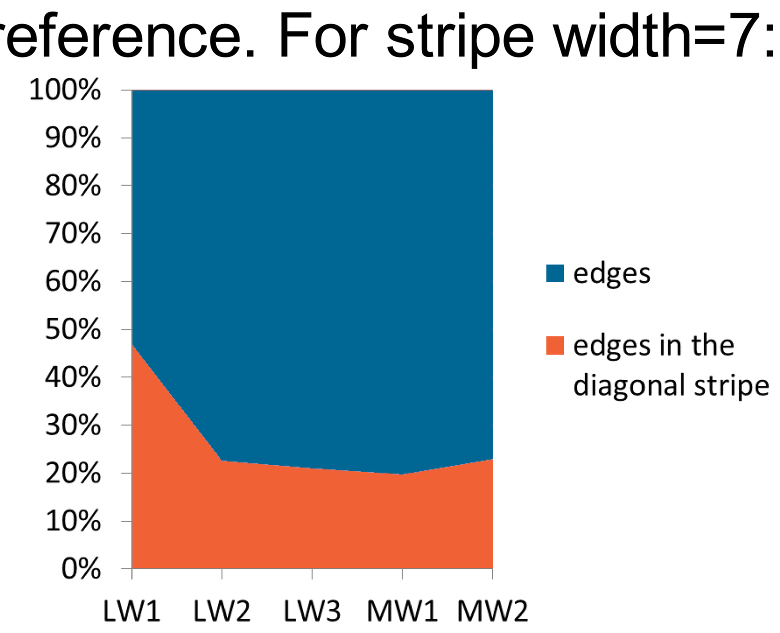
• These graphs are *huge*, *sparse*, and exhibit the *locality of reference* property (i.e., edges often connect nodes with lexicographically close labels).

• Can we compress them in a way that allows efficient mining of their elements?

Compression

Many edges of the citation graphs tested belong *in a stripe around the diagonal* of the adjacency matrix, exhibiting locality of reference. For stripe width=7:

Graph: #edges
LW1: 3,422
LW2: 55,506
LW3: 120,963
MW1: 249,755
MW2: 564,110



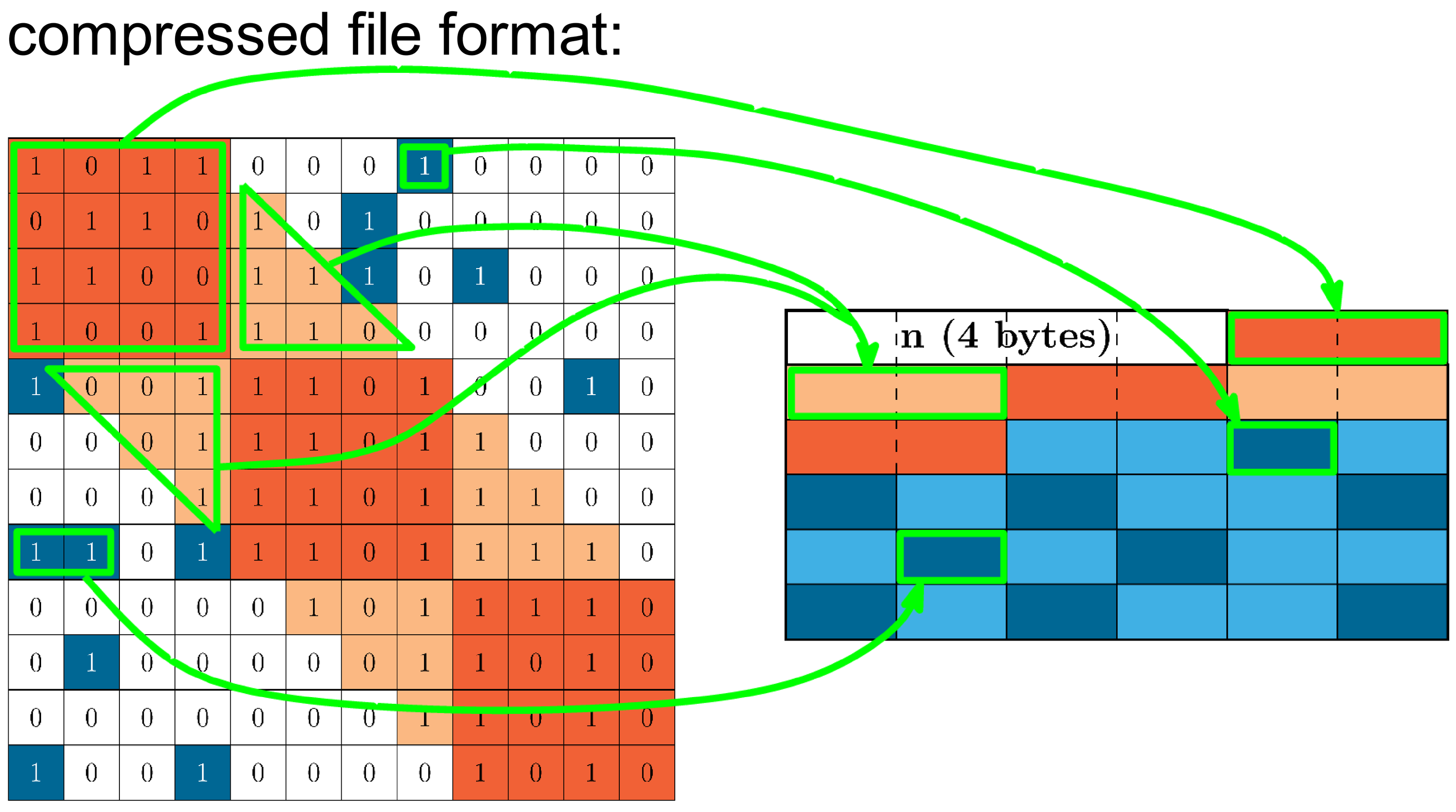
so this diagonal stripe is *denser* than the rest of the matrix.

The Compression is performed in 2 stages:

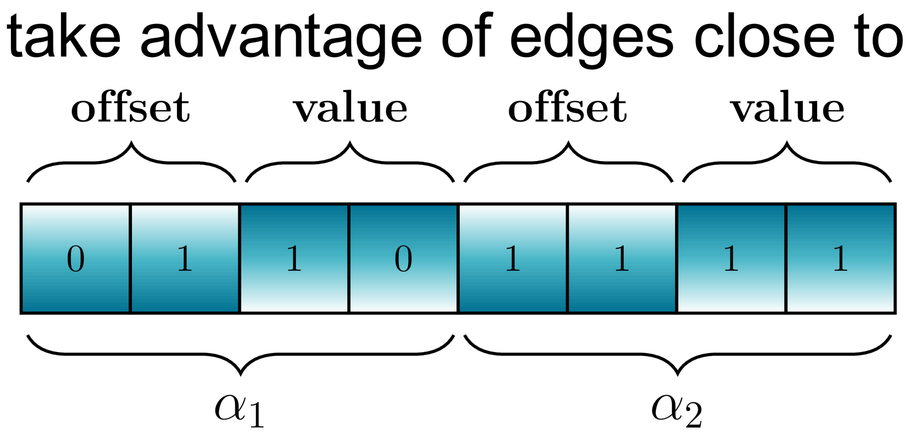
• adjacency matrix-like storage of the diagonal stripe,

• adjacency list-like storage of the remaining edges.

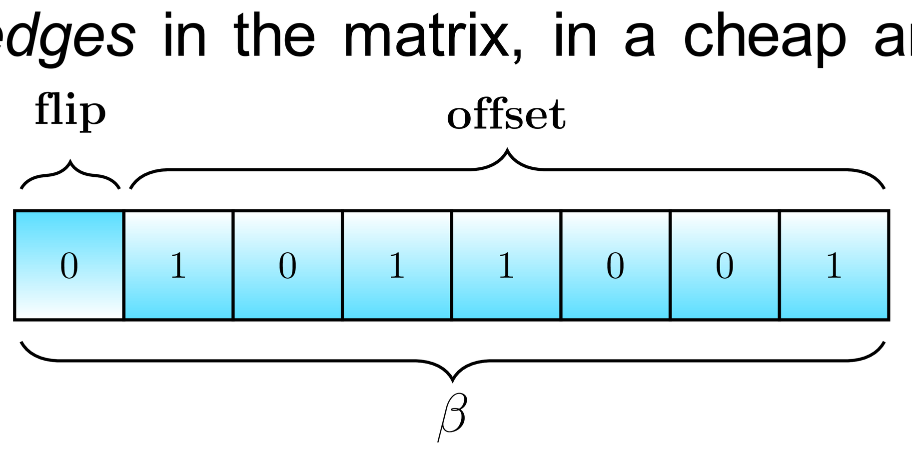
Adjacency matrix example and the corresponding compressed file format:



In the file we use α-type bytes to represent *edges* (along with their offset from previous edge in the matrix and a mark of whether the opposite direction edge exists as well) and take advantage of edges close to each other:



and β-type (if needed) to represent *large distances without edges* in the matrix, in a cheap and compact way:



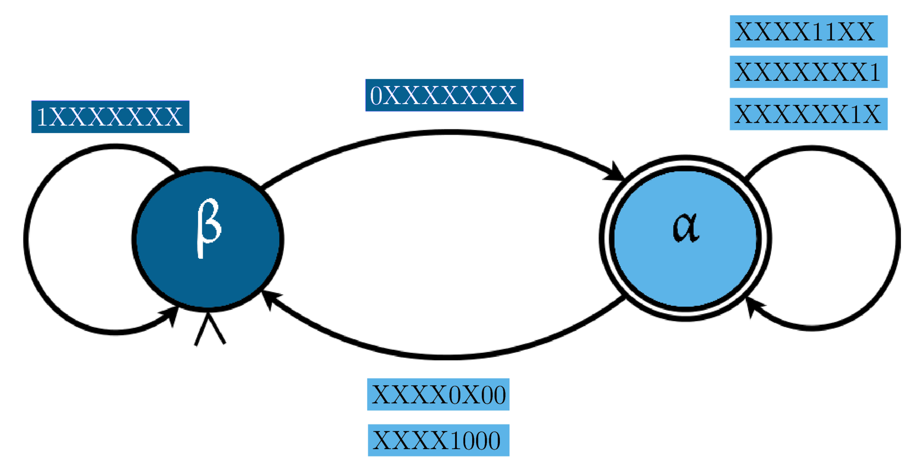
*SiVaC is an acronym for Stairs in a Vacuum Chamber.

Navigation in the Compressed Graph

To tell whether some edge ***exists*** in the graph, we first check whether it should belong in the diagonal stripe.

• If so, we access directly the bit representing the edge

• otherwise we calculate the offset of its intended position in file, approach it using a memory index and move forward reading α and β bytes according to the automaton:



until we find it (success!) or surpass it (edge does not exist).

A node's ***incoming and outgoing*** edges are retrieved in a similar way: we first get those in the diagonal and then access the rest, starting from the offset the first such edge outside the diagonal should have.

How Fast?

Consider a graph G=(V, E).

• Employing a proper index, we check whether a specific edge *exists* in O(log|V|) time.

• A node's *incoming and outgoing* edges are retrieved within the above time plus a term linear in the number of its neighbours.

Experimental Results

• Implemented in Python. Source code is available at http://pypi.python.org/pypi/SiVaC/

• Tested on Intel processor 2.9 GHz, 4MB cache, 4GB RAM, 80 GB SSD, for light and middle weight graphs.

Indicative Compression rates:

graph	*# nodes*	*# edges*	*size (bytes)*	*compressed size (bytes)*
LW1	3,382	3,422	31,506	10,905
MW1	124,538	249,755	2,924,640	1,243,613

Indicative Compression and Access times:

operation	*LW1*		*MW1*	
Compression (s)	0.1286		10.7489	
Index size (bytes)	3,352	12,568	49,432	196,888
Outgoing (ms)	1.2441	0.4145	4.5131	1.3140
Incoming (ms)	1.2427	0.4116	4.6040	1.2929
Both (ms)	1.3145	0.4635	4.4793	1.3417
Exists Edge (ms)	1.1698	0.3225	4.4374	1.0927

In Brief

We exploited the *graph structure* to design a simple yet efficient compression algorithm for graphs exhibiting the locality of reference property, e.g., those modelling networks created by human activity.

We achieved compression rates up to 65.4%. The check for edge existence takes time logarithmic in the number of nodes and the retrieval of a node's in/out-going edges slightly more.