A Simple Algorithm for Compressing Web-like Graphs Efficiently

Panagiotis Liakos[1], Katia Papakonstantinopoulou[1], and Michael Sioutis[2]

[1] University of Athens, Athens, Greece, {p.liakos,katia}@di.uoa.gr
[2] Pierre & Marie Curie University, Paris, France, michael.sioutis@lip6.fr

Abstract. We introduce an efficient compression algorithm for web-like graphs that exploits the graph's structure to achieve better compression rate. In particular, we make use of the locality of reference in the graph, the node similarity and the power law distribution of its nodes' degrees, three properties usually observed in large sparse graphs that model networks created by human activity. Furthermore, our approach focuses on navigating through both the incoming and outgoing edges of each node in linear time. First experimental evaluations of the proposed algorithm indicate promising results.

Keywords: Graph compression, web graphs, social graphs, citation graphs, locality of reference

1 Introduction

Real-world systems and phenomena that involve interactions among various entities are being modeled using graphs for decades now. The recent explosive growth of large-scale systems that are traditionally modeled as graphs, with the worldwide web, social networks, and citation networks being typical examples, has intensified the need for compact, yet efficient representations of graphs. In particular, we need compressed graph representations that allow for mining without decompressing the graph. In this way, classical algorithms can run in main memory over much larger graphs by compressing their plain representations.

The graphs we are interested in are huge, but the degrees of their nodes (indegrees and outdegrees) are power law distributed, so the graphs are rather sparse. Moreover they exhibit *locality of reference*: nodes tend to have successors that are 'close' to them in a sense that depends on the context of the network. For instance, web pages often contain links to pages of the same web site or domain, people in social networks are often friends with individuals from the same neighborhood, University or work, and scientific papers which are tightly relevant tend to cite each other. Furthermore, these graphs exhibit the *copy property* (or similarity), according to which nodes that occur close to each other tend to have many common successors.

In this paper we exploit both the locality of reference and the copy property to compress such graphs efficiently.

Related work In the last dozen of years graph compression has turned into a very active research area and many algorithms have been proposed. Most algorithms in this direction try to offer a good space/time tradeoff. The highest compression ratios are probably achieved by the algorithms of Boldi et al.: In [2] the authors focus on the compression of *social networks* and in its predecessor [4] they compress *web* graphs, exploiting their aforementioned statistical properties. However, the high compression has a negative impact on the access times on some of the graph's elements: the retrieval of the incoming edges of a specific node becomes involved. In another line of work, Brisaboa et al. [5] focus on the efficient storage of the adjacency matrix that represents the graph. They partition the matrix in boxes and store each box in a way that allows quick access of it. The compression they achieve is still high, but the data structure (index) required for the quick access is quite large and has to be present in the main memory. Buehrer and Chellapilla [6] attempt to identify communities in order to replace them with virtual nodes; however the compression they achieve is not better than in [4].

The locality of reference property as well as the node similarity property have been observed in the graphs we're interested in, and they are common in graphs that represent networks created by human activity. In [8] the authors took them into account for the case of the web and initiated research on web graph compression. The similarity of nodes in citation graphs has been studied in [7].

The most common graph representation is the adjacency matrix of the graph. The locality of reference property is reflected on the adjacency matrix in the following way: using a 'good' ordering of the nodes' labels, i.e., an ordering in which labels of densely connected nodes are close to each other, many edges fall close to the main diagonal of the adjacency matrix. Such orderings are preferred in practice, but finding the ordering that minimizes the distance of the edges from the main diagonal is computationally hard. Intuitively, if we have some good clustering of the graph, based solely on the link structure, and assign consecutive labels to the nodes in each cluster, the lexicographic ordering of the labels is rather good in the above sense.

Such an extrinsic ordering appears naturally in the case of world wide web. Web graph representations assume that each URL corresponds to some identifier. Moreover it is assumed that URLs are alphabetically sorted, and this naturally puts together the pages of the same domain. As a result, the locality of reference translates into closeness of page identifiers. However extrinsic orderings are not obvious for all graphs, so for social and citation graphs, finding a good ordering is a challenging issue. In [3] Boldi et al. test some known orderings of the nodes and propose some new ones, and study their effect on the compression of web and social graphs. Citation graphs share some features with the web graphs. In fact, the web graph is a citation graph of references from one web page to another in the world wide web. In both networks the individuals (scientific paper authors and web page authors respectively) have similar incentives: they link to

other papers/pages in order to increase the quality of the content of their own paper/page.

Short description of results We develop a compression algorithm for directed graphs that exploits the structural properties of the graph to avoid redundancy, and focuses on surfacing both the incoming and outgoing edges of each node in linear time. Our algorithm considers the adjacency matrix that represents the graph and stores its 'dense' part in a way that can be accessed in constant time (adjacency matrix-like storage), while from the rest of the matrix only the useful (i.e., non zero) parts are stored (adjacency list-like storage). Our experiments, demonstrated in Section 3, indicate promising results which encourage us to further pursue this approach.

The organization of this paper is as follows. In Section 2 our algorithm is presented and its complexity is discussed. In Section 3 we evaluate our approach experimentally. Finally, in Section 4 we conclude and give directions for future research.

2 Algorithms and Complexity

This section presents SiVaC[3], our algorithm for compressing directed graphs, and analyzes its asymptotic complexity.

SiVaC attempts to exploit the locality of reference and the copy properties along with the power law distribution of the nodes' degrees (indegrees and outdegrees) to provide a compression schema with very efficient navigation. Due to the above properties and assumptions, an edge is with high probability *close* to the main diagonal of the adjacency matrix representing the graph. Hence, given that these graphs are generally rather sparse, the area around the main diagonal is denser than the rest of the graph. We call this area the *diagonal stripe*, and define it as follows: an edge (i, j) is in the diagonal stripe iff $i - k \leq j \leq i + k$. In the citation graphs we examined experimentally [1], large number of edges tend to be in the diagonal stripe, meeting our expectations regarding the locality of reference property. This trend for $k = 3$ is illustrated in Figure 2. We choose this value for k since the resulting thickness of the diagonal stripe, shown in Figure 1(a), achieves the optimal tradeoff between number of edges in it and space required for its storage. Moreover, if $(k+1) \mod 4 = 0$, each box of the diagonal stripe can fit in a whole number of bytes, making the computation of its exact position in the file where we store the compressed graph less involved. The value of k was fixed by performing experiments for several values on different graphs of the dataset we experimented on [1] and carefully interpreting the results.

Figure 1(b) illustrates the way a SiVaC compressed file is organized. The first four bytes are used to hold the number of the nodes of the graph. The following $n + (n \mod 2)$ bytes (for $k = 3$) represent the diagonal stripe of the adjacency matrix and are thoroughly described in Section 2.1. The rest of the edges are compressed using the Vacuum Chamber Step explained in Section 2.2.

[3] SiVaC is an acronym for *Stairs In a Vacuum Chamber*.

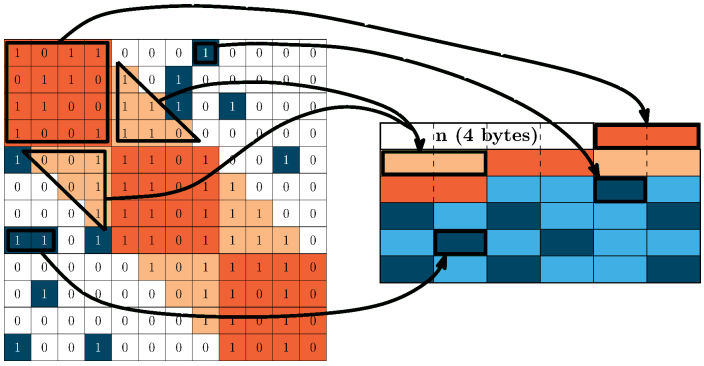


Fig. 1: Example of an adjacency matrix (a) and the corresponding compressed file format (b).

2.1 Stairs Step

In order to exploit the high concentration of edges in the diagonal stripe, and thus take advantage of the locality of reference property, we store it separately from the rest of the graph in the format of an adjacency matrix. In particular, we partition the diagonal stripe into boxes, assign them an ordering, and store them uncompressed in the beginning of the obtained (compressed) file. Thus, for every edge in the diagonal stripe of the adjacency matrix we are aware of the position of the bit that represents it and can access it in constant time.

For example, the boxes in Figure 1(a) would be ordered in the following way: the top left orange box would be box 0 and two bytes would be used in the compressed file to enclose it. The two light orange areas that share borders with it would be combined to form box 1 and would be stored in the following two bytes. This ordering would continue until the end of the diagonal stripe of the adjacency matrix, finally holding $n + (n \mod 2)$ bytes of the compressed file.

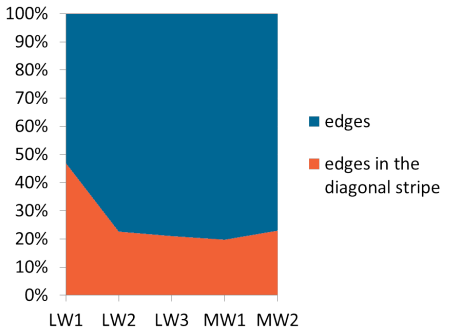
We name this step *Stairs* since the diagonal stripe resembles two stairs as they appear when one is viewed in front of the other.

2.2 Vacuum Chamber Step

The rest of the edges are represented using a structure that we call the *Vacuum Chamber*. The adjacency matrix of the given graphs tends to contain small series of 1s, each one indicating the existence of an edge, followed by a large number of 0s, indicating the absence of edges. We use two types of bytes, called α and β, the first to store edges and take advantage of edges close to each other and the second to represent large distances without edges in the matrix in a cheap and compact way. Both types are illustrated in Figure 4. An α byte is split in two and can hold two edges along with their symmetric ones. The first two bits represent the offset of the current edge from the previous edge in the matrix and

graph	*# edges*	*# edges in diagonal*
LW1	3,422	1,607
LW2	55,506	12,558
LW3	120,963	25,469
MW1	249,755	49,347
MW2	564,110	129,576

(a)



(b)

Fig. 2: The percentage of edges in the diagonal stripe indicates *locality of reference*

the other two bits the presence or absence of edges between two nodes, say i and j. In particular, '01', '10', and '11' denote the presence of edges (j, i) (incoming to i), (i, j) (outgoing from i), and both edges respectively, while '00' denotes the absence of an edge between these nodes. Information about the symmetric edge is held in order to allow navigation in both the incoming and outgoing edges of each node in linear time. In the case of a '00' in an α_2 part (see Figure 4 to locate it), the two offset bits are used in order to exploit the copy property. In particular, they are used to denote large offsets, i.e. an offset equal to the number of the nodes of the graph, that appear often due to the aforementioned property.

A β byte contains only information about the offset that stands between two α bytes. In case a β byte is followed by an α byte, we use the first two bits of the α byte to quadruplicate the range of offsets that can be represented. Moreover, we dedicate the most significant bit of a β byte to indicate whether it is followed by an α (0) or a β (1) byte. The next seven bits should be multiplied by 2^9 in the former case and 2^7 in the latter. Frequently, using these two extra bits saves us from employing one more β byte to represent the desired distance, thus reducing the use of many consecutive β bytes.

Algorithm 1 describes how an edge outside the diagonal stripe is added to the result file. We name this step Vacuum Chamber because it resembles the process of air being pumped out of a room.

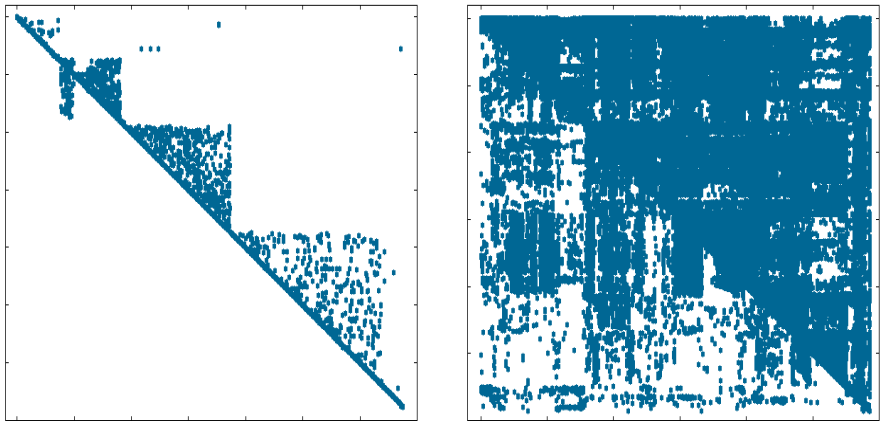


Fig. 3: Adjacency matrices of graphs LW1 and MW2 respectively.

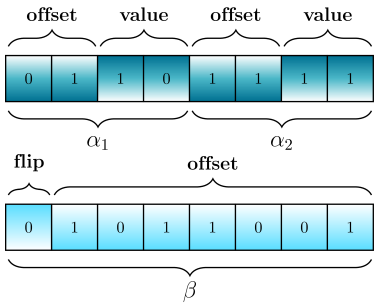


Fig. 4: α and β-type bytes

Algorithm 1 Pseudocode for edge storing with SiVaC

```
procedure storeEdgeInFile(offset,value,boxType)
    while not edgeInFile do
        if boxType is α1 then
            if offset< 4 then
                write offset and value to the 4 leftmost bits
                edgeInFile = True
            else
                write '0000' to the 4 leftmost bits
                boxType = α2
            end if
        else if boxType is α2 then
            if offset< 4 then
                write offset and value to the 4 rightmost bits
                edgeInFile = True
            else
                depending on the offset,
                write a special value ('XX00') to
                indicate compactly a common transition
            end if
        else if boxType is β then
            if offset< 2^9 then
                write '0 XXXXXXX' (offset-offset mod 4 bits)
                offset = offset - offset mod 4
                boxType = α1
            else if offset> 2^9 * (2^7 − 1) then
                write '1 1111111' (2^7−1 bits)
                offset = offset - 2^9 * (2^7 − 1)
            else
                write '1 XXXXXXX' (offset div 2^9 bits)
                offset = offset mod 2^9
            end if
        end if
    end while
end procedure
```

The rules for creating α or β-type boxes when *reading* the compressed file are summarized in Figure 5. The transitions are conditional on the last box created and the byte we just read. A byte is denoted by a string of 8 bits, where the values of the bits that do not matter in the current decision are set to 'X'. In accordance with Figure 4, α bytes are illustrated in light-colored frames whereas the β bytes in dark-colored ones. After a β box, another β box is created iff the most significant bit of the next byte is equal to 1. After an α box, another α box is created iff the value of the α_2 part is '00' and its offset is anything but '11'.

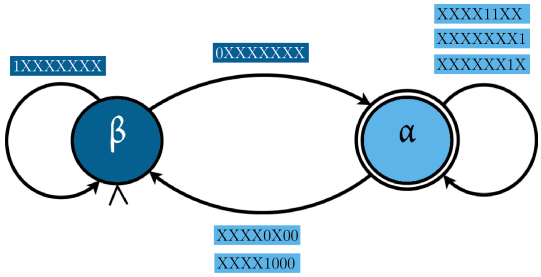


Fig. 5: Automaton deciding the type of box to be created

The format composed in this step introduces an impediment in its navigation which we surpass by indexing into memory the offset to which some specific bytes correspond. Indexing is performed by employing a sorted dictionary in conjunction with a bisection algorithm. Thus, the access times of the edges in the obtained file can be bounded using an index of a proper size. This structure is the only memory requirement of the algorithm after the compression has been performed. We examine the effect of the index size in Section 3.

2.3 Navigation

In order to examine whether a specific edge exists in the graph, we first clarify whether it should be in the diagonal stripe. In the case where the edge belongs in the diagonal stripe, we immediately calculate the corresponding byte of the compressed file, read it from the file, and return the value of the bit representing the edge. In the case where the edge doesn't belong in the diagonal stripe, we calculate the offset of the pair of nodes, i.e., the offset of the edge's intended position in the file, discover the closest access point to it using the memory index, and start reading α and β bytes from the file according to the automaton of Figure 5 to move forward. If we find the offset of the pair we return the value, otherwise we reach a bigger offset and so we infer that the edge does not exist.

The incoming and outgoing edges of a node are discovered in a similar way. We first check for all possible edges that would end up in the diagonal stripe bytes, using the method described earlier, and then search for the rest of the edges, starting from the offset which the edge with the first node would have. Due to the format of the compressed file, the incoming and outgoing edges of each node are grouped together, thus the cost of their retrieval is close to the cost of checking for the existence of a specific edge.

2.4 Complexity

We now bound the worst case time complexity of our algorithm, denoting by V the set of nodes of the graph. Let us denote with Δ the maximum among all degrees of the graph (in- and out-degrees).

Theorem 1. *The time complexity of SiVaC algorithm is $O(\Delta|V|)$. In the compressed graph, the incoming and outgoing edges of a specific node are retrieved in time $O(|V|)$ and the existence of an edge is verified in $O(\log|V|)$.*

Proof. SiVaC processes the edges of E sequentially and for each edge, if it belongs into the diagonal stripe it computes the box it should be stored in and writes it to the (compressed) file. In order to compress the rest of the edges we perform a preprocessing step that brings into memory the incoming and outgoing edges of each node in $O(|E|)$ time. Then, we iterate through this structure and for each edge we compute the offset from the previous one stored and place the necessary (β and) α-byte(s) in the file. It follows that the time complexity of SiVaC is $O(\Delta|V|)$.

Given a compressed graph and a specific node, we retrieve its incident edges that belong in the diagonal stripe in $O(1)$ time. For the rest of the edges we have to perform a sequential search in the file from the position of its first possible edge to the position of the last one. This is achieved in $O(|V|)$ time in the worst case.

In order to check whether a specific edge exists, we navigate to the closest position that is indexed in memory and search forward until we discover (success) or go past its place in the file (does not exist). Employing a proper index, this check is done in $O(\log|V|)$ time.

3 Implementation and Experimental Results

We implemented SiVaC algorithm in Python (version 2.7.3) as a proof of concept of our technique; our code is available for download in the following Python repository: `http://pypi.python.org/pypi/SiVaC/`.

The experiments were carried out on a computer with an Intel Core i7-3520M processor with a CPU frequency of 2.90GHz and a 4MB L2 cache, a total of 4GB DDR3 1600MHz RAM, a SATA3 Intel SSD hard disc of 80GB, and the Precise Pangolin (Ubuntu Linux 12.04 LTS) x86_64 OS. Only one of the CPU cores was used for the experiments.

We first applied our compression technique on the directed lightweight and middleweight graphs of the dataset provided by WSDM 2013 Data Challenge [1]. In all the obtained results, the file and index sizes are in Bytes, the compression times in seconds, and the access times in milliseconds.

Table 1 shows the sizes of the compressed graphs achieved by SiVaC, in comparison to their initial sizes and the sizes of the files produced by the well-known Boldi-Vigna (BV) technique [4], for the graphs tested. We observe that the obtained files are compressed with a ratio from 34.61% to 40.38%. The higher compression for the first lightweight graph (LW1) is due to the fact that a higher percentage of its edges are in the diagonal stripe, as opposed to rest of the graphs. However, even with the less intense locality of reference observed in those graphs, our technique manages to reduce the initial size significantly. The compression rate of our suggested algorithm is worse than the rate of BV,

but the latter only gives out-neighbors access. In order to have efficient access on both the in- and out-neighbors using BV, we have to keep two copies of the graph (the original one and its transpose), so the size of the compressed graph doubles. In Table 1 we see that the size of our compressed LW1 graph is 26.4% smaller than the double size of the graph compressed by BV. In Table 2 we can see that the time needed for creating the compressed files is negligible for both graphs.

graph	***# nodes***	***# edges***	***size***	***compressed size***	
				BV	***SiVaC***
LW1	3,382	3,422	31,506	7,410	10,905
LW2	31,017	55,506	600,686	115,715	249,788
LW3	63,311	120,963	1,361,464	260,426	564,270
MW1	124,538	249,755	2,924,640	563,892	1,243,613
MW2	293,154	564,110	7,369,002	1,254,182	2,975,910

Table 1: Compression results

Then, we searched for all outgoing or all incoming edges of a given node, as well as for both incoming and outgoing edges. We experimented on two of the graphs of Table 1, namely LW1 and MW1, and employed indices of different size to evaluate their effect on the performance of the above actions.

The size of the index is the only significant memory footprint of our approach and it is clear that its role is crucial. We observe that all access times are inversely proportional to the index size for both graphs. We also notice that the average times for mining the incoming, the outgoing, or both of the edges of a node are almost identical. This was expected since the first two actions perform exactly the same number of operations, while the third is remotely larger since it spends twice as much time searching in the diagonal stripe in comparison to the former two.

Finally, we tested for edge existence in the compressed graph. The tests were performed for random edges and the results indicate that this action is sufficiently faster than the other three.

graph	***LW1***		***MW1***	
Compression (s)	0.1286		10.7489	
Index size (#entries)	34	135	1,246	4,982
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

Table 2: Compression and access times

We must note that the timings obtained in Table 2, serve strictly as a measurement of efficiency of our algorithm with respect to the index size used. Due to our proof of concept implementation, our timings are not to be compared with state-of-the-art approaches, such as the very robust graph compression software of Boldi [4][4].

4 Future Work

Under minor modifications our algorithm works for undirected graphs as well, with comparable compression ratios and access times. In order to be able to handle graphs of arbitrary sizes we can extend our method by modifying the structure of the β bytes to exploit their bits so that they can cover even larger distances in the adjacency matrix. We reckon that by utilizing them in a more sophisticated way, we can apply our algorithm to the heavyweight graphs of [1], expecting similar compression ratios and access times.

Further, we will research on dynamically specifying an optimal value for the thickness of the diagonal stripe, which is now fixed (and dependent on the provided dataset) as mentioned in Section 2. This could be achieved by either a heuristic approach or an automated statical analysis per given graph prior to its compression.

Finally, we will perform major code refactoring and we will explore alternative Python environments, such as PyPy[5], utilizing trace-based just-in-time (JIT) compilation techniques.

References

1. Wsdm 2013 data challenge. `http://www.wsdm2013.org/index.php/authors/data-challenge`, 2013. [Online; accessed 22-April-2013].
2. P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*, 2011.
3. P. Boldi, M. Santini, and S. Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.
4. P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *WWW*, 2004.
5. N. R. Brisaboa, S. Ladra, and G. Navarro. k2-Trees for Compact Web Graph Representation. In *SPIRE*, 2009.
6. G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, pages 95–106, 2008.
7. W. Lu, J. Janssen, E. Milios, N. Japkowicz, and Y. Zhang. Node similarity in the citation graph. *Knowl. Inf. Syst.*, 11(1):105–129, 2006.
8. K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*, DCC '02, pages 122–, Washington, DC, USA, 2002. IEEE Computer Society.

[4] `http://webgraph.di.unimi.it/`

[5] `http://pypy.org`