SiVaC* : An Efficient Graph Compression Algorithm

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

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

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

ABSTRACT

This paper introduces a new efficient graph compression algorithm for large-scale 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 and the power law distribution of its nodes' degrees, two 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.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks; E.4 [**Coding and Information Theory**]: Data compaction and compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Performance

Keywords

Citation graph, web graph, graph compression

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.

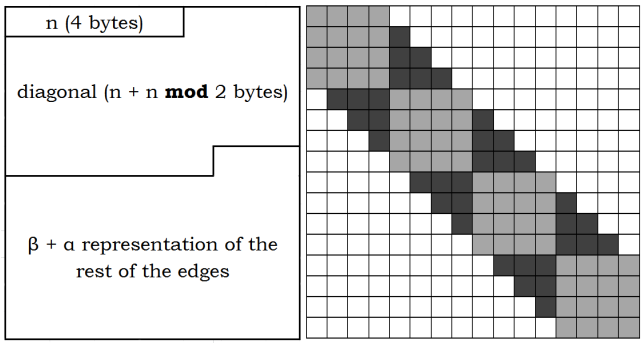
*SiVaC is an acronym for *Stairs In a Vacuum Chamber*.

The graphs we are interested in are huge, but the degrees of their nodes (indegrees and outdegrees) are power law distributed, thus, the graphs are rather sparse. These graphs exhibit the *locality of reference*: nodes tend to have successors that are close to them. Furthermore, they exhibit the *copy property* (or similarity) since nodes that occur close to each other tend to have many common successors. In this paper we concentrate on the locality of reference property.

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 [1] the authors focus on the compression of *social networks* and in its predecessor [2] 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, Claude and Ladra [3] 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.

Short description of results. We develop a graph 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, while from the rest of the matrix only the useful (i.e., non zero) parts are stored. 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.



(a) (b)

Figure 1: Compressed file format (a) and the diagonal of the adjacency matrix (b)

2. ALGORITHMS AND COMPLEXITY

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

SiVaC attempts to exploit the locality of reference property along with the power law distribution of the nodes' degrees (indegrees and outdegrees) to provide a compression schema with very efficient navigation. 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. Since the scientific paper authors have similar incentives with the web page authors, and the citation graphs' structure and properties resemble those of the web graph, it is reasonable to assume that the labeling of the nodes of the citation graphs is performed in a similar fashion with the web pages described above (instead of pages of the same domain we have papers of the same scientific area).

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*, and define it as follows: an edge (i, j) is in the diagonal iff $i - k \leq j \leq i + k$. In the citation graphs we examined experimentally, large number of edges tend to be in the diagonal, meeting our expectations regarding the locality of reference property. This trend for $k = 3$ is illustrated in Table 1. We choose this value for k since the resulting thickness of the diagonal, shown in Figure 1b, achieves the optimal tradeoff between number of edges in it and space required for its storage. The value of k was fixed by performing experiments for several values on different graphs and carefully interpreting the results.

Figure 1a 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 represent the diagonal 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.

graph	*# edges*	*# edges in diagonal*	*percentage*
LW1	3,422	1,607	46.96%
LW2	55,506	12,558	22.62%
LW3	120,963	25,469	21.06%
MW1	249,755	49,347	19.76%
MW2	564,110	129,576	22.97%
MW3	1,215,119	158,505	13.04%
MW4	4,041,859	370,370	9.16%

Table 1: The percentage of edges in the diagonal indicates *locality of reference*

2.1 Stairs Step

In order to exploit the high concentration of edges in the diagonal, we store it separately from the rest of the graph, in the format of an adjacency matrix. In particular, we partition the diagonal 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 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 1b would be ordered in the following way: the top left light gray box would be box 0 and two bytes would be used to enclose it. The two dark gray 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 of the adjacency matrix, finally holding $n+(n \mod 2)$ bytes of the compressed file.

We name this step *Stairs* since the diagonal 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 a 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 take advantage of edges close to each other and the second to represent large empty distances in a cheap and compact way. Both types are illustrated in Figure 2. An α byte is split in two and can hold two edges along with their symmetric ones. The first two bits represent the offset from the previous edge and 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. A β byte contains only information about the offset that stands between two α bytes. In order to avoid the use of many consecutive β bytes we dedicate the most significant bit of one to indicate whether the next seven should be multiplied by 2^9 or not.

We name this step Vacuum Chamber because it resembles

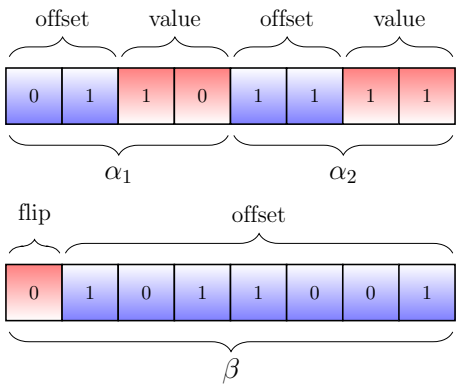


Figure 2: α and β-type bytes

the process of air being pumped out of a room.

Given a set of edges E, $SiVaC$ algorithm computes its representation in the compressed format. Algorithm 1 describes how an edge outside the diagonal is added to the result file.

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' (XXXXXXX: offset-offset mod 4)
                offset = offset - offset mod 4
                boxType = α1
            else if offset>2^9 * (2^7 − 1) then
                write '1 1111111' (1111111: 2^7−1)
                offset = offset - 2^9 * (2^7 − 1)
            else
                write '1 XXXXXXX' (XXXXXXX: offset div 2^9)
                offset = offset mod 2^9
            end if
        end if
    end while
end procedure
```

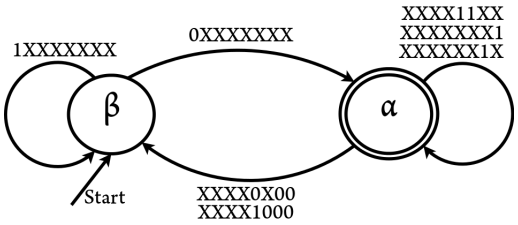


Figure 3: Automaton deciding the type of box to be created

The rules for creating α or β-type boxes when reading the compressed file are summarized in Figure 3. The transitions are conditional on 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'.

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 occurred. 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. In the case where the edge belongs in the diagonal, 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, we calculate the offset of the pair, discover the closest access point using the memory index, and start reading α and β bytes from the file to move forward. If we find the offset of the pair we return the value, and if we reach a bigger offset we infer that the edge does not exist.

The incoming and outgoing edges of a node are discovered as follows. We check for all possible edges that would end up in the diagonal 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.

Theorem 1. *The time complexity of SiVaC algorithm is $O(|V||E|)$. 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

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

Table 2: Compression results

for each edge, if it belongs into the diagonal 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 $O(|V||E|)$ 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 $O(|V||E|)$.

Given a compressed graph and a specific node, we retrieve its incident edges that belong in the diagonal 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); 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 given dataset. 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 2 shows the sizes of the compressed graphs achieved by SiVaC in comparison to their initial sizes, for one lightweight and one middleweight graph. We observe that the obtained files are compressed to 34.61% and 42.52% of the original files respectively. The higher compression for the lightweight graph is due to the fact that a higher percentage of its edges was in the diagonal. In Table 3 we can see that the time needed for creating the compressed files is negligible for both graphs.

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 the same two graphs and employed indices of different size to evaluate their effect on the performance of the above actions.

graph	*LW1*		*MW1*	
Compression (s)	0.1286		10.7489	
Index size (B)	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

Table 3: Compression and access times

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 is 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 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.

4. FUTURE WORK

Under minor modifications our algorithm works for undirected graphs as well, with comparable compression ratio and access times. We reckon that by utilizing the β-boxes in a more sophisticated way to cover larger distances in the adjacency matrix, we can apply our algorithm to heavyweight graphs, expecting similar compression ratio and access times.

Further, we will research on dynamically specifying an optimal value for the thickness of the diagonal, which is done statically 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[1], utilizing trace-based just-in-time (JIT) compilation techniques.

5. REFERENCES

[1] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*, 2011.

[2] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *WWW*, 2004.

[3] F. Claude and S. Ladra. Practical Representations for Web and Social Graphs. In *CIKM*, 2011.

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