Memory-Optimized Distributed Graph Processing through Novel Compression Techniques*

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

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

Alex Delis
University of Athens
Athens, Greece
ad@di.uoa.gr

ABSTRACT

A multitude of contemporary applications now involve graph data whose size continuously grows and this trend shows no signs of subsiding. This has caused the emergence of many distributed graph processing systems including `Pregel` and `Apache Giraph`. However, the unprecedented scale now reached by real-world graphs hardens the task of graph processing even in distributed environments and the current memory usage patterns rapidly become a primary concern for such contemporary graph processing systems. We seek to address this challenge by exploiting empirically-observed properties demonstrated by graphs that are generated by human activity. In this paper, we propose three space-efficient adjacency list representations that can be applied to any distributed graph processing system. Our suggested compact representations reduce respective memory requirements for accommodating the graph elements up to 5 times if compared with state-of-the-art methods. At the same time, our memory-optimized methods retain the efficiency of uncompressed structures and enable the execution of algorithms for large scale graphs in settings where contemporary alternative structures fail due to memory errors.

Keywords

Graph compression; Pregel; distributed computing.

1. INTRODUCTION

The proliferation of WWW-based applications, the explosive growth of social networks, as well as the continually-expanding WWW-space, have collectively led to systems whose operations routinely deal with voluminous data that are modeled as graphs. The active users of Facebook are more than 1 billion [11] and Google indexes over 1 trillion unique URLs [2]. This ever-increasing requirement in terms of graph vertices has given rise to a number of distributed graph-processing systems [1, 18, 21, 22] that efficiently handle large-scale graphs using commodity hardware [13].

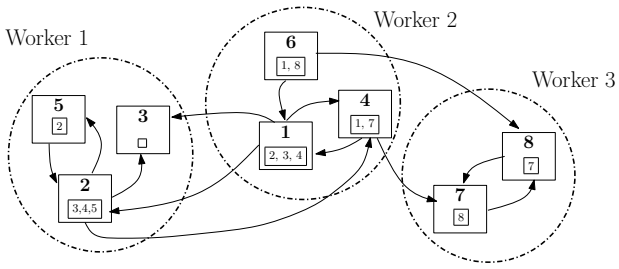

Figure 1: A graph partitioned on a vertex basis in a distributed environment. Each vertex maintains a list of its out-edges.

Most of these approaches parallelize the execution of algorithms by dividing graphs into partitions and assigning vertices to workers (i.e., machines) following the "*think like a vertex*" programming paradigm introduced with `Pregel` [19]. However, recent studies [9, 13] point out that the proposed frameworks fail to handle the unprecedented scale of real-world graphs due to ineffective memory usage [13]. Thus, memory optimization has become critical when dealing with graphs of such magnitude even in a distributed environment.

Figure 1 illustrates a graph partitioned over three workers. Every vertex is assigned to a *single* machine and maintains a list of its out-edges. This partitioning hardens the task of compression, as related research efforts have focused on a centralized machine environment and exploit similarities exhibited between vertices [3, 5, 8, 10, 16, 17]. However, this is infeasible when graphs are partitioned on a vertex basis. A major step towards memory optimization was contributed to `Apache Giraph` [1] by Facebook [11]. However, improvements focused entirely on a more careful implementation for the representation of the out-edges of a vertex; the redundancy exhibited in real-world graphs was not exploited.

In this paper, we follow the `Pregel` paradigm and partition the graph vertices among the nodes of a distributed computing environment. In this context, we present three novel techniques that 1) offer space efficient-representations of the out-edges of vertices, 2) allow fast mining (in-situ) of the graph elements without the need of decompression, 3) enable the execution of graph algorithms in memory-constrained settings, and 4) ease the task of memory management, thus allowing faster execution. The vertex placement policy that `Pregel`-like systems follow forces us to store the out-edges of each vertex independently as Figure 1

*This work has been partially supported by the University of Athens Special Account of Research Grants № 13233.

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI:

depicts. This preserves the *locality of reference* property, known to be exhibited in real-world graphs [4, 20], and enables us to exploit patterns that arise among the out-edges of a *single* vertex.

Our first technique, termed `BVEdges`, applies all methods of [5] that are appropriate for the vertex placement policy of `Pregel`. `BVEdges` primarily focuses on identifying intervals of consecutive out-edges of a vertex and employs universal codings to efficiently represent them. To facilitate access without imposing the significant computing overheads of `BVEdges`, we propose `IntervalResidualEdges`, which holds the corresponding values of intervals in a non-encoded format. Additionally, we propose `IndexedBitArrayEdges`, a novel technique that considers the out-edges of each vertex as a row in the adjacency matrix of the graph and indexes only the areas holding edges using byte sized bit-arrays.

Our experimental results show significant improvements on space-efficiency for all proposed techniques. We manage to reduce memory requirements up to 5 times in comparison with currently applied techniques. This eases the task of scaling to billions of vertices and allows us to load much larger graphs than was ever possible before. In settings where earlier approaches were also able to execute graph algorithms, we achieve significant performance improvements in terms of time of up to 31% due to memory optimization, as less time is spent for garbage collection. These findings establish our structures as the undisputed preferable option for web graphs, which offer compression-friendly orderings, or any other type of graph, after the application of a reordering that favors its compressibility.

2. RELATED WORK

Our work lies in the intersection of distributed graph processing and compressed graph representations. In this regard, we outline here pertinent aspects of these two areas.

Google's proprietary `Pregel` [19] is a graph processing system that enables scalable batch execution of iterative graph algorithms. As the source code of `Pregel` is not publicly available, a number of graph processing systems that follow the same data flow paradigm have emerged. `Apache Giraph` [1] is such an open-source `Java` implementation with contributions from Yahoo! and Facebook, that operates on top of `HDFS`. Our work focuses on `Pregel`-like systems and extends `Giraph`'s implementation. `GPS` [21] is a similar `Java` open-source system that proposes the large adjacency list partitioning technique for high-degree vertices. `Pregel+` [22] is implemented in `C++` and uses `MPI` processes as workers to achieve high efficiency. Unlike the aforementioned systems, `GraphLab` [18] does not follow the `Pregel` paradigm but rather supports asynchronous execution and adopts a data-pull model with a shared memory abstraction. Our work is orthogonal to these approaches as our compressed representations can be readily applied to all above systems.

The field of graph compression has yielded significant research results after the work presented in [20]. Randall et al. exploit the *locality of reference* as well as the *similarity property* that is unveiled in web graphs when their links are ordered lexicographically. The seminal work on web graph compression is that of Boldi and Vigna [5], who introduce a number of sophisticated techniques to further reduce the bits per link ratio. Following efforts present improved results with regard to space [3], study the compressibility characteristics of social graphs [10], and employ adjacency matrix representations to reduce access time of the graph's elements [8, 16, 17]. All the above approaches focus on providing a compact representation of a graph to be loaded in the memory of a *single* machine. Hence, they exploit the presence of all edges in a centralized computing node, which is not suitable for `Pregel`-like systems offering distributed execution.

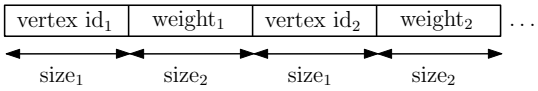

Figure 2: `Giraph`'s *ByteArrayEdges* representation.

To the best of our knowledge, `GBASE` [14] is the only approach that considers compressed graph representations in a distributed environment. Kang et al. consider the case of a graph stored over `HDFS`. `GBASE` uses block compression to efficiently store graphs by splitting the respective adjacency matrices into regions. The latter are compressed using several methods including *Gzip* and *Gap Elias'-γ* encoding. We should note, however, that `GBASE` does not follow the vertex-centric model we have adopted in this work.

3. BACKGROUND

In this section we outline `Pregel` and `Apache Giraph`, and offer definitions for the encodings that we use for our suggested compression techniques.

3.1 Pregel & Apache Giraph

`Pregel` [19] is a computational model suitable for large scale graph processing, inspired by the *Bulk Synchronous Parallel* programming model. `Pregel` encourages programmers to *"think like a vertex"* by following a vertex-centric approach. The input to a `Pregel` algorithm is a directed graph whose vertices, along with their respective out-edges, are distributed among the machines of a computing cluster. `Pregel` algorithms are executed as a sequence of iterations, termed *supersteps*. During a *superstep*, every vertex independently computes a user-defined list of actions and sends messages to vertices that are to be used during the following *superstep*. A synchronization barrier between *supersteps* is used to ensure that all messages are delivered at the beginning of the next *superstep*. A vertex may vote to halt at any *superstep* and will be reactivated upon receiving a message. The algorithm terminates when all vertices are halted and there are no messages in transit. `Pregel` loads the input graph and performs all associated computations in-memory. Thereby, `Pregel` only supports graphs whose edges entirely fit in main-memory.

`Apache Giraph` [1] is an open-source implementation of `Pregel`. The default `Giraph` structure for holding the out-edges of a vertex is that of `ByteArrayEdges` [11]. This representation is realized as a byte array, in which target vertex ids and their respective weights are held consecutively, as Figure 2 illustrates. The bytes required per out-edge are determined by the data type used for its id and weight; for integer numbers $4+4=8$ bytes are required. `ByteArrayEdges` are impractical for algorithms involving mutations as they deserialize all out-edges to perform a removal.

3.2 Codings for Graph Compression

Over the last two decades studies of real-world graphs have revealed the presence of common properties [5, 15, 20].

In this work we exploit two of these properties, namely the *heavy-tailed degrees' distribution* and the *locality of reference*, to achieve effective compression. The former property implies the sparsity of such graphs, while due to the latter, the majority of the edges of a graph link vertices that are close to each other in the order. The locality of reference is evident in web graphs whose vertices are ordered lexicographically, and can be surfaced in other types of graphs by applying the reordering algorithm of [4].

In order to compress the data in our structure, we can use various encoding approaches; below, we provide the definitions of *Elias' γ* and ζ codings that we employ in Section 4.1.1. We also furnish the definitions of unary and minimal binary coding that help define the first two codings. Let x denote a positive integer, b its binary representation and l its length. The aforementioned codings are defined as follows:

1. *Unary coding*: the unary coding of x consists of $x-1$ 0s followed by a 1.
2. *Minimal binary coding* over an interval: consider the interval $[0, z-1]$ and let $s = \lceil \log z \rceil$. If $x < 2^s - z$ then x is coded using the x-th binary word of length $s-1$ (in lexicographical order), otherwise, x is coded using the $(x - z + 2^s)$-th binary word of length s.
3. *Elias' γ coding* [12]: the γ coding of x consists of l in unary, followed by the last $l-1$ digits of b.
4. *ζ coding* with parameter k [6]: given a fixed positive integer k, if $x \in [2^{hk}, 2^{(h+1)k} - 1]$, its ζ_k-coding consists of $h+1$ in unary, followed by a minimal binary coding of $x - 2^{hk}$ in the interval $[0, 2^{(h+1)k} - 2^{hk} - 1]$.

In the context of graph compression, Elias' γ coding is preferred for the representation of rather small values of x, whereas ζ coding is more proper for potentially large values.

4. OVERVIEW OF OUR APPROACH

In this section we detail our compressed data structures for the representation of a vertex's neighbors in a graph.

4.1 Representations based on intervals

The *locality of reference* property is evident through the adjacency lists of the graphs of our dataset, all of which tend to have a lot of neighbors with *consecutive* ids. We can exploit this property by applying a technique similar to the one introduced in [5]. In particular, [5] distinguishes between the neighbors whose ids form some *interval* of consecutive ids, and the rest. To reconstruct all the edges of the *intervals* only the leftmost neighbor id and the length of the *interval* needs to be kept. This information is further compressed using gap *Elias' γ coding*. The rest of the edges, termed *residuals*, are compressed using *ζ coding*. We build on these ideas and introduce two vertex-centric representations that exploit *locality of reference* in a similar fashion.

4.1.1 BVEdges

Our first representation, namely `BVEdges`, focuses solely on compressing the neighbors of a vertex, at the cost of computing overheads. Therefore, we simply adjust the method of Boldi and Vigna [5] to the requirements imposed by `Pregel`'s model. The compressed data structure discussed in [5] considers the whole graph and exploits the current vertex's id during compression. However, this information is not available in the level where adjacency lists are kept in the `Pregel` model. To overcome this issue, we use the first neighbor id we store in our structure as a reference to proceed with gap encoding. We use *Elias' γ coding* for *intervals*, and *ζ coding* for *residuals*. *Elias' γ coding* is most preferable for *intervals* of at least 4 elements [5]; shorter *intervals* are more compactly stored as *residuals*.

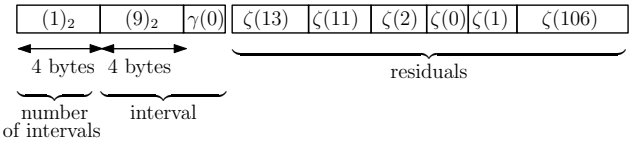

Figure 3: The storage of neighbors in `BVEdges`, detailed in Example 1. $\gamma(x)$ and $\zeta(x)$ denote the γ and ζ encodings of x respectively.

Definition 1 (`BVEdges`)**:** *Given a list l of a node's neighbors, `BVEdges` is a sequence of bits holding consecutively: the number of intervals in l of length at least 4; for each such interval, the smallest neighbor id in it and the γ-coded difference of the interval length minus 4; a ζ coding for each of the remaining neighbors, its argument being either the difference x between the current node's id and the previous node id which was encoded to be stored in the sequence minus 1, or, in case $x < 0$, the quantity $2|x| - 1$.*

Example 1: *Consider the following sequence of neighbors to be represented:* (2, 9, 10, 11, 12, 14, 17, 18, 20, 127)*. We employ `BVEdges` as illustrated in Figure 3. Here, there is only one* interval *of length at least equal to* 4*:* $[9\,..\,12]$*. We first store the number of* intervals *in unencoded binary form. Then, we store the leftmost id of the* interval*, i.e.,* 9*, again using its unencoded binary representation. We proceed with storing a representation of the length of the* interval *to enable the recovery of the remaining elements. In particular, we store the γ* coding *of the difference of the* interval *length minus the minimum* interval *length, which is $4 - 4 = 0$ in our case. Then, we append a representation for the* residual *neighbors. For each* residual*, we store the ζ* coding *of the difference of its id with the id of the last node stored, minus* 1 *(as each id appears at most once in the neighbors' list). The* residual *id 2 is smaller than the smallest id of the first* interval*, so we store the* residual *neighbor* 2 *as $\zeta(13)$, since $2|2 - 9| - 1 = 13$, and the* residual 14 *as $\zeta(11)$, since $14 - 2 - 1 = 11$. Similarly, we store* 17*,* 18*,* 20 *and* 127 *as $\zeta(2), \zeta(0), \zeta(1)$ and $\zeta(106)$, respectively.*

The respective values computed in each step are written using a bit stream. This, combined with the fact that values have to be encoded, renders the operation costly. We also investigated the idea of treating all neighbors as *residuals* to examine if the re-construction of *intervals* was more expensive. However, we experimentally found that the resulting larger bit stream offered worse access time. Accessing the out-edges of a vertex requires the following procedure: first, we decode the *intervals* and create an iterator using them. After this iterator returns all its elements, we create a second one that decodes the *residuals* one by one.

4.1.2 IntervalResidualEdges

Our second representation, namely `IntervalResidualEdges`, also incorporates the idea of using *intervals* and *residuals*. However, to avoid costly bit stream writes and reads, we propose a different structure. In particular, we opt to keep the value of the leftmost id of an *interval* unencoded, along with a byte that is able to index up to 256 consecutive neighbors. *Residuals* are then also kept unencoded. Clearly,

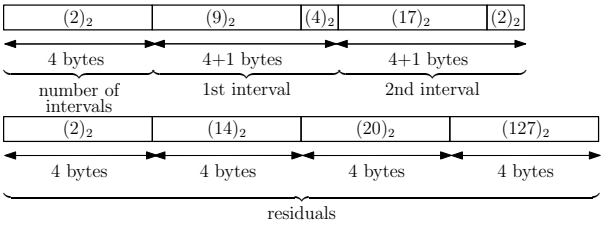

Figure 4: The storage of neighbors in `IntervalResidualEdges`, detailed in Example 2. $(x)_2$ is the binary representation of x.

any consecutive neighbors of length at least equal to 2 are represented more efficiently using an *interval* rather than two or more *residuals*. Therefore we set the minimum interval length with `IntervalResidualEdges` equal to 2. Due to the *locality of reference* property, this one byte allows us to compress the adjacency list significantly, while also avoiding the use of expensive encodings and bit streams. We note here that all ids are stored in binary form.

Definition 2 (`IntervalResidualEdges`)**:** *Given a list l of a node's neighbors, `IntervalResidualEdges` is a sequence of bits holding consecutively: the number of intervals in l; the smallest neighbor id and the length of each such interval; the id of each of the remaining neighbors.*

Example 2: *The representation of the aforementioned sequence of neighbors* (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) *using `IntervalResidualEdges` is illustrated in Figure 4. In this case there are two* intervals *of at least 2 consecutive neighbors, namely* $[9\,..\,12]$ *and* $[17, 18]$. *We first store the number of* intervals*, and then use one* 5*-byte element for each* interval*, consisting of a 4-byte representation of the smallest neighbor id in it (i.e.,* 9 *and* 17*), plus a byte holding the number of neighbors in this* interval *(*4 *and* 2 *respectively). Finally we append a* 4*-byte representation for each* residual *neighbor.*

This representation delivers its elements through the following procedure: while there are still unread *intervals*, the procedure reads 5-bytes, i.e., the leftmost element of the *interval* and its length, and produces one by one the elements of the *interval*. When all *intervals* are processed, the procedure reads in the *residuals* directly as integers.

4.2 IndexedBitArrayEdges

Our first two representations exploit the *consecutivity* exhibited among the neighbors of a vertex. Here we propose `IndexedBitArrayEdges`, which takes advantage of the *concentration* of edges in *specific areas* of the adjacency matrix, regardless of whether these edges are in fact consecutive. `IndexedBitArrayEdges` uses a single byte to depict eight possible out-edges. Using a byte array, we construct a data structure of 5-byte elements, one for each interval of neighbor ids having the same quotient by 8. The first 4 bytes of each element represent the quotient, while the last one serves as a set of 8 flags indicating whether each possible edge in this interval exists. As the neighbor ids of each node tend to concentrate within a few areas, the number of intervals is small and the compression achieved is exceptional.

Definition 3 (`IndexedBitArrayEdges`)**:** *Given a bit-array r representing a list of a node's neighbors, `IndexedBitArrayEdges` is a sequence of 5-byte elements, each one holding an octet of r that contains at least one 1: the first 4 bytes hold the distance in r of the first bit of the octet from the beginning of r; the last one holds the octet.*

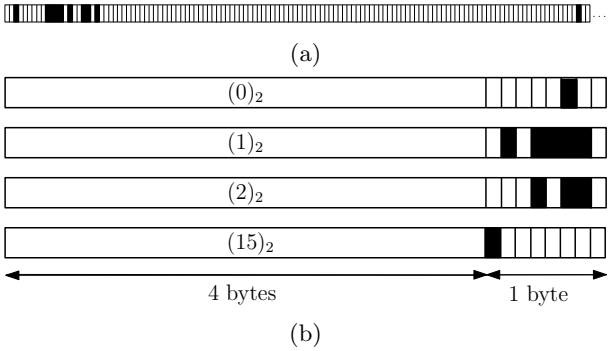

Figure 5: A bit-array representation of an adjacency list (a) and the storage of these neighbors in `IndexedBitArrayEdges (b), detailed in Example 3`. $(x)_2$ denotes the binary representation of x.

Example 3: *The representation of the aforementioned sequence of neighbors* (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) *using `IndexedBitArrayEdges` is illustrated in Figure 5. The bit-array representation of this adjacency list is shown in Figure 5(a). The quotient and remainder of each node id divided by* 8 *give us the approximate position (octet) and the exact position, respectively, of the node in the bit-array; hence, as depicted in Figure 5(b), the neighbors are grouped in four sets:* $\{2\}$, $\{9, 10, 11, 12, 14\}$, $\{17, 18, 20\}$, $\{127\}$. *All ids in each set share the same quotient when divided by* 8*, which will be referred as* index number *henceforth. For instance, the index number of the third set is* 2*, and is stored in binary form, denoted by* $(2)_2$*, in the first part of the third element. Moreover, the remainders of the ids* 17*,* 18 *and* 20 *divided by* 8 *are* 1*,* 2*, and* 4 *respectively, and so the* 2*nd,* 3*rd and* 5*th flags from the right side of the same element are set to* 1 *to depict these neighbors.*

Accessing the out-edges of a vertex requires the following procedure: first, we read a 5-byte element. Then, we recover out-edges from the flags of its last byte and reconstruct the neighbor ids using the first 4 bytes. After we examine all flags of the last byte, we proceed by reading the next 5-byte element and repeat until we retrieve all out-edges.

5. EXPERIMENTAL EVALUATION

We implemented[1] our techniques using `Java` and compared their performance against `ByteArrayEdges` using a number of well-studied web and social network graphs [4, 5], reaching up to 2 billion edges. We first present the dataset and detail the specifications of the machines used in our experiments. Then, we proceed with the evaluation of our out-edge representations by answering the following questions: (i) How much more space-efficient is each of our three compressed out-edge representations compared to `ByteArrayEdges`? (ii) Are our techniques competitive speed-wise when memory is not a concern? (iii) How much more efficient are our compressed representations when the available memory is constrained? (iv) Can we execute algorithms for large graphs in settings where it was not possible before?

[1] https://goo.gl/TSWcyO

graph	vertices	edges	ByteArrayEdges	BVEdges	IntervalResidualEdges	IndexedBitArrayEdges
uk-2007-05@100000	100,000	3,050,615	22.61MB	6.41MB	7.92MB	8.91MB
uk-2007-05@1000000	1,000,000	41,247,159	279.16MB	67.36MB	82.7MB	97.79MB
indochina-2004	7,414,866	194,109,311	1,511.67MB	442.34MB	646.03MB	554.23MB
hollywood-2011	2,180,759	228,985,632	1,381.91MB	287.53MB	613.52MB	676.88MB
uk-2002	18,520,486	298,113,762	2,733.6MB	1,092.82MB	1,224.07MB	1,255.67MB
arabic-2005	22,744,080	639,999,458	4,820.09MB	1,428.97MB	1,674.75MB	1,849.83MB
uk-2005	39,459,925	936,364,282	7,401.88MB	2,383.54MB	2,728.74MB	2,928.81MB
sk-2005	50,636,154	1,949,412,601	14,829.64MB	4,889.85MB	5,657.79MB	6,354.17MB

Table 1: Memory requirements for the small and large-scale graphs of our dataset.

5.1 Experimental Setting

Our dataset consists of 8 web and social network graphs, whose properties are detailed in Table 1. We run our experiments on a cluster that comprises 8 virtual machines with 13GB of virtual RAM each, running on a Dell PowerEdge R630 with an Intel® Xeon® E5-2630 v3, 2.40 GHz with 8 cores and 128GB of RAM. We set up Apache Hadoop 1.0.2 with 1 master and 8 slave nodes and a maximum per machine JVM heap size of 10GB. Lastly, we used `Giraph` 1.1.0.

5.2 Space Efficiency Comparison

We present here our results regarding space efficiency for the web and social network graphs of our dataset. We compare our methods involving compression with `Giraph`'s default representation, namely `ByteArrayEdges`.

Table 1 lists the memory required by all four representations in MB. We observe that our techniques require significantly less memory compared to `ByteArrayEdges`. `BVEdges` outperforms all representations, as expected. In particular, `BVEdges` always needs less than 40% of the requirements of `ByteArrayEdges`, and reaches much smaller figures in certain cases, e.g., 20.08% for *hollywood-2011*. However, we see that our novel `IntervalResidualEdges`, that does not impose any computing overheads, also manages to achieve equivalent space-efficiency, with its requirements ranging from 29% to 44% compared to those of `ByteArrayEdges`. Finally, the results of our `IndexedBitArrayEdges` are impressive as well, as its requirements are usually less than 40% and always below 50% of those of `ByteArrayEdges`.

5.3 Execution Time Comparison

In this section, we present results regarding the execution time of the PageRank [7] algorithm.[2] We expect that any `Pregel` algorithm not involving mutations would exhibit similar behavior for the different representations, as it would feature the same set of actions regarding out-edges, i.e., initialization and retrieval. Reported timings are averages of multiple executions. We do not consider the initialization time as it is negligible compared to the execution time.

5.3.1 Comparison using small-scale graphs

We begin by investigating the performance of the different out-edge representations in settings where memory is sufficient. Figure 6 depicts the total time needed for each of the four techniques when executing the PageRank algorithm on setups of 2, 4, and 8 workers for the graph *indochina-2004*.

We observe that `IndexedBitArrayEdges` and `IntervalResidualEdges` do not impose any latency in the process. In particular, using either of our two novel representations we achieve execution times that are slightly better than those

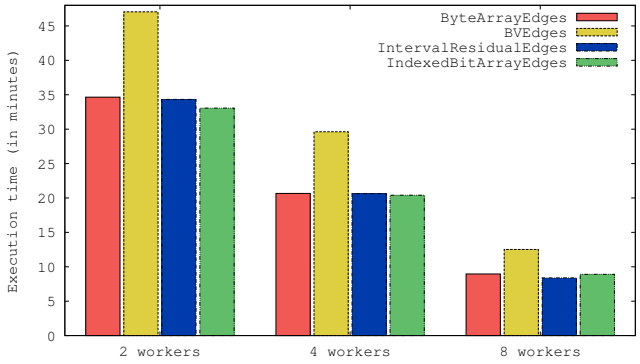

Figure 6: Execution time of PageRank algorithm for the graph *indochina-2004* using 2, 4, and 8 workers.

of `ByteArrayEdges` for all three setups. `BVEdges` is inferior speed-wise due to the computationally expensive access of the out-edges offered through this structure which requires decoding *Elias-*γ and ζ*-coding* values. Thus, the computing overheads imposed by the techniques of [5] are not negligible and simply adopting them proves to be inefficient.

For graphs which are equivalent to or smaller than *indochina-2004* the performance was similar. In particular, for all three setups `IndexedBitArrayEdges` and `IntervalResidualEdges` managed to execute the PageRank algorithm in slightly less time than that of `ByteArrayEdges`. On the contrary, `BVEdges` required more time for each *superstep*.

5.3.2 Comparison using large-scale graphs

We further examine the performance of our representations using setups where memory does not suffice for the needs of the execution of PageRank. Figure 7 depicts the time needed for each *superstep* of the execution of PageRank for the *uk-2005* graph for all four out-edge representations. We see that `BVEdges` requires significantly more time than our two other representations for every *superstep*, as was the case with small-scale graphs. We also see, however, that in this setup the execution with `ByteArrayEdges` tends to fluctuate in performance, and occasionally performs worse than `BVEdges`. The increased memory requirements of `Giraph`'s default implementation, result in an unstable pace during the execution of PageRank, as it needs to perform garbage collection very frequently to accommodate the memory objects required in every *superstep*. `ByteArrayEdges` still manages to outperform `BVEdges` requiring an average of 4.8 minutes per *superstep* as opposed to 5.08 minutes per *superstep*. However, both `IndexedBitArrayEdges` and `IntervalResidualEdges` greatly outperformed `ByteArrayEdges`, requiring 3.15 and 3.56 minutes per *superstep*, respectively.

[2] https://goo.gl/CEbvOX

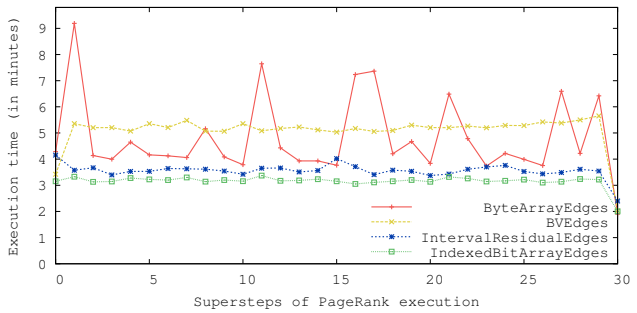

Figure 7: Execution time for each *superstep* of PageRank for the graph *uk-2005* using 5 workers.

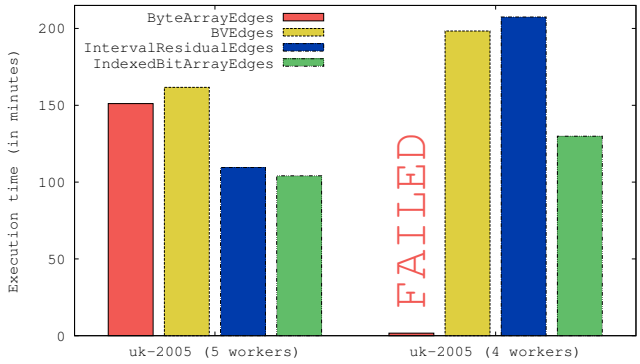

Figure 8: Execution time of the PageRank algorithm for the graph *uk-2005* using 5 and 4 workers.

The performance difference of the four representations with regard to the total execution time of PageRank is evident in Figure 8 (left). The executions using `IndexedBitArrayEdges` and `IntervalResidualEdges` are faster by 31.1% and 27.54% than the one with `ByteArrayEdges`, respectively.

We repeat the same experiment using 4 workers and illustrate the respective results in Figure 8 (right). The execution with `ByteArrayEdges` on this setup fails as the garbage collection overhead limit is exceeded, i.e., more than 98% of the total time is spent doing garbage collection. Our proposed implementations, however, are able to execute PageRank for the *uk-2005* graph despite the limited resources. We observe that under these settings `IntervalResidualEdges` performs worse than both `BVEdges` and `IndexedBitArrayEdges`. As we can see in Table 1, the latter requires more memory than `IntervalResidualEdges` to represent the out-edges of *uk-2005*. However, the retrieval of out-edges using `IndexedBitArrayEdges` is more memory-efficient than that of `IntervalResidualEdges`. Therefore, execution spends much less time performing garbage collection with `IndexedBitArrayEdges` which results in it being 37.42% faster than `IntervalResidualEdges` under these settings.

6. CONCLUSIONS

In this paper, we propose and implement three compressed out-edge representations for distributed graph processing, termed `BVEdges`, `IntervalResidualEdges`, and `IndexedBitArrayEdges`. We focus on the vertex-centric model that all `Pregel`-like graph processing systems follow and examine the efficiency of our structures by extending one such system, namely `Apache Giraph`. Our techniques build on empirically-observed properties of real-world graphs and offer significant memory optimizations that are applicable to any distributed graph compressing system that follows the `Pregel` paradigm. Our `IntervalResidualEdges` and `IndexedBitArrayEdges` representations outperform `Giraph`'s default representation, namely `ByteArrayEdges`, and are able to execute algorithms over large-scale graphs under very modest settings. Furthermore, our representations are clearly superior than `ByteArrayEdges` when memory is an issue, and are capable of successfully performing executions in settings where `Giraph` fails due to memory requirements.

7. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] We knew the web was big.... https://goo.gl/ugjWuI.

[3] A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

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

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

[6] P. Boldi and S. Vigna. The WebGraph Framework II: Codes For The World-Wide Web. In *DCC*, 2004.

[7] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[8] N. Brisaboa, S. Ladra, and G. Navarro. k2-Trees for Compact Web Graph Representation. In *SPIRE*. 2009.

[9] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. M. Jermaine. A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms. In *ACM SIGMOD*, 2014.

[10] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On Compressing Social Networks. In *KDD*, 2009.

[11] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. of the VLDB Endowment*, 8(12):1804–1815, 2015.

[12] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[13] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. of the VLDB Endowment*, 7(12):1047–1058, 2014.

[14] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. GBASE: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

[15] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD*, 2005.

[16] P. Liakos, K. Papakonstantinopoulou, and M. Sioutis. On the Effect of Locality in Compressing Social Networks. In *ECIR*, 2014.

[17] P. Liakos, K. Papakonstantinopoulou, and M. Sioutis. Pushing the Envelope in Graph Compression. In *CIKM*, 2014.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.

[19] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, 2010.

[20] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The Link Database: Fast Access to Graphs of the Web. In *DCC*, 2002.

[21] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[22] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *WWW*, 2015.