

Realizing Memory-Optimized Distributed Graph Processing

Panagiotis Liakos, Katia Papakonstantinou, and Alex Delis

Abstract—A multitude of contemporary applications heavily involve graph data whose size appears to be ever-increasing. This trend shows no signs of subsiding and has caused the emergence of a number of distributed graph processing systems including Pregel, Apache Giraph and GraphX. However, the unprecedented scale now reached by real-world graphs hardens the task of graph processing due to excessive memory demands even for distributed environments. By and large, such contemporary graph processing systems employ ineffective in-memory representations of adjacency lists. Therefore, memory usage patterns emerge as a primary concern in distributed graph processing. We seek to address this challenge by exploiting empirically-observed properties demonstrated by graphs generated by human activity. In this paper, we propose 1) three compressed adjacency list representations that can be applied to any distributed graph processing system, 2) a variable-byte encoded representation of out-edge weights for space-efficient support of weighted graphs, and 3) a tree-based compact out-edge representation that allows for efficient mutations on the graph elements. We experiment with publicly-available graphs whose size reaches two-billion edges and report our findings in terms of both space-efficiency and execution time. Our suggested compact representations do 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.

Index Terms—Distributed graph processing, graph compression, Pregel.

1 INTRODUCTION

THE proliferation of web applications, the explosive growth of social networks, and the continually-expanding WWW-space have led to systems that routinely handle voluminous data modeled as graphs. Facebook has over 1 billion active users [1] and Google has long reported that it has indexed unique URLs whose number exceeds 1 trillion [2]. This ever-increasing requirement in terms of graph-vertices has led to the realization of a number of distributed graph-processing approaches and systems [3], [4], [5], [6]. Their key objective is to efficiently handle large-scale graphs using predominantly commodity hardware [7].

Most of these approaches parallelize the execution of algorithms by dividing graphs into partitions [8], [9] and assigning vertices to workers (i.e., machines) following the “think like a vertex” programming paradigm introduced with Pregel [10]. However, recent studies [7], [11] point out that the so-far proposed frameworks [3], [4], [5], [6] fail to handle the unprecedented scale of real-world graphs as a result of ineffective, if not right out poor, memory usage [7]. Thereby, the space requirements of real-world graphs have become a major memory bottleneck.

Deploying space-efficient graph representations in a vertex-centric distributed environment to attain memory optimization is critical when dealing with web-scale graphs and remains a challenge. Figure 1 illustrates a graph partitioned over three workers. Every vertex is assigned to a *single* node and maintains a list of its out-edges. For example, vertices 1, 4, and 6 are assigned to worker 1, and

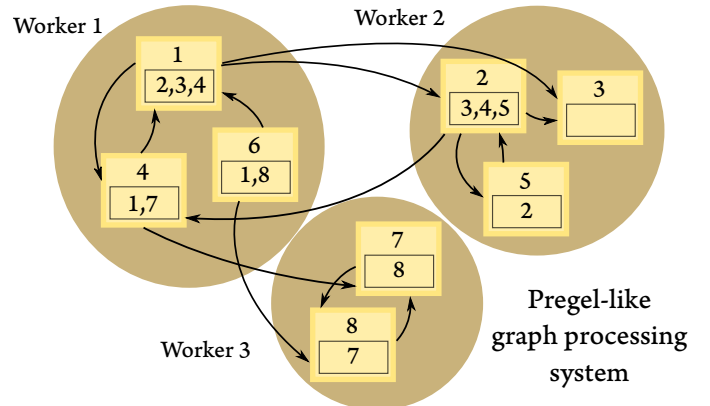


Fig. 1. A graph partitioned on a vertex basis in a distributed environment. Each vertex maintains a list of its out-edges.

vertex 1 maintains out-edges towards vertices 2, 3, and 4. This partitioning hardens the task of compression as vertices become unaware of the physical node their neighbors ultimately find themselves in. Related efforts have exclusively focused on providing a compact representation of a graph in a centralized machine environment [12], [13], [14], [15], [16]. In such single-machine settings, we can exploit the fact that vertices tend to exhibit similarities. However, this is infeasible when graphs are partitioned on a vertex basis, as each vertex must be processed independently of other vertices. Furthermore, to achieve memory optimization, we need representations that allow for mining of the graph’s elements *without decompression*; this decompression would unfortunately necessitate additional memory to accommodate the resulting unencoded representation.

• P. Liakos, K. Papakonstantinou, and A. Delis are with the University of Athens, Greece. A. Delis is also with the NYU Abu Dhabi, UAE. E-mail: {p.liakos, katia, ad}@di.uoa.gr.

A noteworthy step towards memory optimization was taken by *Facebook* when it adopted *Apache Giraph* [3] for its graph search service; the move yielded both improved performance and scalability [1]. However, *Facebook's* improvements regarding memory optimization entirely focused on a more careful implementation for the representation of the out-edges of a vertex [1]; the redundancy due to properties exhibited in real-world graphs was not exploited.

In this paper, we investigate approaches that help realize compact representations of out-edges in (weighted) graphs of web-scale while following the *Pregel* paradigm. The vertex placement policy that *Pregel*-like systems follow necessitates for storing the out-edges of each vertex independently as Figure 1 depicts. This policy preserves the *locality of reference* property, known to be exhibited in real-world graphs [17], [18], and enables us to exploit in this work, patterns that arise among the out-edges of a *single* vertex. We cannot however utilize similarities among out-edges of different vertices, for we are unaware of the partition each vertex is placed into.

Our first technique, termed *BVEdges*, applies all methods proposed in [12] that can effectively function with the vertex placement policy of *Pregel* in a distributed environment. *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. We facilitate support of weighted graphs with the use of a parallel array holding variable-byte encoded weights, termed *VariableByteArrayWeights*. Additionally, we propose *IndexedBitArrayEdges*, a novel technique that considers the out-edges of each vertex as a single row in the adjacency matrix of the graph and indexes only the areas holding edges using byte sized bit-arrays. Finally, we propose a fourth space-efficient tree-based data structure termed *RedBlackTreeEdges*, to improve the trade-off between memory overhead and performance of algorithms requiring mutations of out-edges.

Our experimental results with diverse datasets indicate significant improvements on space-efficiency for all our proposed techniques. We reduce memory requirements up-to 5 times in comparison with currently applied methods. This eases the task of scaling to *billions of vertices per machine* and so, it allows us to load much larger graphs than what has been feasible thus far. In settings where earlier approaches were also capable of executing graph algorithms, we achieve significant performance improvements in terms of time of up-to 41%. We attribute this to our introduced 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. Last but not least, we attain a significantly improved trade-off between space-efficiency and performance of algorithms requiring mutations through a representation that uses a tree structure and does not depend on node orderings.

In summary, our contributions in this paper are that we: I) offer space efficient-representations of the out-edges of

vertices and their respective weights, II) allow fast mining (in-situ) of the graph elements without the need of decompression, III) enable the execution of graph algorithms in memory-constrained settings, and IV) ease the task of memory management, thus allowing faster execution.

2 RELATED WORK

Our work lies in the intersection of distributed graph processing systems and compressed graph representations. In this regard, we outline here pertinent aspects of these two areas: i) well-established graph processing systems and the challenges they face when it comes to memory management, and ii) state-of-the-art space-conscious representation of real-world graphs.

Google's proprietary *Pregel* [10] 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* [3] 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. Therefore, we provide a short discussion on both *Pregel* and *Giraph* in Section 3.1. *GPS* [4] is a similar Java open-source system that introduces an optimization for high-degree vertices: as the degrees of graphs created by human activity are heavy-tail distributed, certain vertices have an "extreme number" of neighbors and stall the synchronization at every iteration. To overcome this deficiency, *GPS* proposes the large adjacency list partitioning (*LALP*) technique. *Pregel+* [6] is implemented in C++ and uses MPI processes as workers to achieve high efficiency. Moreover, *Pregel+* features two additional optimizations. The first is the mirroring of vertices, an idea similar to that of *LALP*. The second is a request-respond API which simplifies the process of a vertex requesting attributes from other vertices and merges all requests from a machine to the same vertex into a single request. Unlike the aforementioned distributed graph processing systems that follow *Pregel's* BSP execution model, some approaches employ asynchronous execution [5], [19], [20], [21]. *GraphLab* [5] is such an example that also adopts a shared memory abstraction. *PowerGraph* [19] is included in *GraphLab* and mitigates the problem of high-degree vertices by following an edge-centric model bundle. Han and Daudjee [20] extend *Giraph* with their Barrierless Asynchronous Parallel (BAP) computational model to reduce the frequency of global synchronization barriers and message staleness. *GraphX* [22] is an embedded graph processing system build on top of the very successful *Apache Spark* [23] distributed dataflow system. *GraphX* has received notable attention, partly due to the widespread adoption of the *Spark* framework. However, a recent comparison [24] against *Giraph* shows that the latter is able to handle 50x larger graphs than *GraphX* and is more efficient even on smaller graphs. Our work is orthogonal to these approaches as we introduce compressed adjacency list representations that can be readily applied to all above systems. Several *Facebook* optimizations contributed to *Giraph* are reported in [1]. Significant improvements are realized through a new representation of

out-edges which serializes the edges of every vertex into a byte-array. However, this representation does not entail any memory optimization through compression. `MOCGraph` [25] is a `Giraph` extension focused on improving scalability by reducing the memory footprint. This is achieved through the *message online computing* model according to which messages are digested on-the-fly. The `MOCGraph` approach is also orthogonal to our work, as `MOCGraph` focuses solely on the memory footprint of messages exchanged, whereas our focus is on representation of the graph. `Deca` [26] transparently decomposes and groups objects into byte-arrays to significantly reduce memory consumption and achieves impressive execution time speed ups. Our techniques achieve compression over `Giraph`'s graph representation that already uses byte-arrays and resides in memory for the entire execution of algorithms. However, `Deca` can offer additional benefits through the optimization of memory consumption with regard to objects other than the graph representation, such as the messages exchanged.

As the size of graphs continues to grow numerous efforts focus on shared-memory or secondary storage architectures. Shun et al. [27] consider compression techniques that can be applied on a shared-memory graph processing system and manage to halve space usage at the cost of slower execution when memory is not a bottleneck. `Gemini` [28] achieves surprising efficiency by using MPI and performing updates directly on shared-memory graph data, instead of passing messages between cores on the same socket. Our focus is on shared-nothing distributed computing architectures, in which certain techniques of [27] and [28] are inapplicable. `GraphChi` [29], `FlashGraph` [30], and `Graphene` [31] maintain graph data on disks and achieve reasonable performance, having very modest requirements. However, no effort is spent on compressing the graph data. Moreover, our approach does not impose any limitations on the execution time of in-memory distributed graph processing systems, or sacrifice the ease of programming and fault tolerance that go along with the `Pregel` paradigm.

The increasing number of proposed graph processing systems initiated research concerning their performance. Lu et al. [32] experiment with the number of vertices in a graph and report that `GPS` and `GraphLab` run out of memory in settings where `Giraph` and `Pregel+` manage to complete execution. In [11], Cai et al. find that both `Giraph` and `GraphLab` face significant memory-related issues. Han et al. [7] carry out a comparative performance study that includes among others, `Giraph`, `GraphLab` and `GPS`. The asynchronous mode of `GraphLab` is reported to have poor scalability and performance due to the overhead imposed by excessive locking. Moreover, the optimization of `GPS` for high degree vertices offers little performance benefit. These findings motivated us to use the implementation of `Giraph` as a basis for this work. [7] notes that `Giraph` is much improved compared with its initial release, yet, it still demonstrates noteworthy space deficiencies. We note that this is also the case in `Giraph`'s only subsequent release since, i.e., 1.2, as it does not introduce any additional out-edge representations providing improved space- or time-efficiency. Therefore, in this paper we investigate compact representations to further reduce `Giraph`'s space requirements. Lastly, [7] additionally reports that `Giraph`'s new

adjacency list representation is not suitable for algorithms featuring mutations (i.e., additions and/or deletions). To this effect, we have opted to investigate structures that do not necessarily favor mutations.

The field of graph compression has yielded significant research results after the work presented in [17]. Randall et al. exploit the *locality of reference* as well as the *similarity property* that is unveiled in web graphs when their links are sorted lexicographically. The seminal work on web graph compression is that of Boldi and Vigna [12], who introduce a number of sophisticated techniques as well as a new coding to further reduce the bits per link ratio. Several following efforts [13], [14], [15] managed to present improved results with regard to space but not access time of the graph's elements. Brisaboa et al. [15] introduce a graph compression approach that uses the adjacency matrix representation of the graph, instead of adjacency lists. A tree structure is used to hold the areas of the adjacency matrix that do actually represent edges. As real-world graphs are sparse, these areas are a very small part of the original matrix. However, there is also a cost in maintaining the in-memory tree structure. In [16], [33], this cost is amortized by representing a specific area around the diagonal of the adjacency matrix without the use of an index and the remaining elements of the graph through an adjacency list representation. All the above approaches focus on providing a compact representation of a graph that can be loaded in the memory of a *single* machine. Hence, the techniques used exploit the presence of all edges in a centralized computing node, which is not suitable for distributed graph processing systems. To the best of our knowledge, our approach is the first to consider compressed graph representations for `Pregel`-like systems offering distributed execution.

`GBASE` [34] is the only approach we are aware of that considers compressed graph representations in a distributed environment in general. `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 established by now "think like a vertex" model we have adopted in this work. In addition, `GBASE` aims at minimizing the storage and I/O cost and its techniques require full decompression of multiple blocks for the extraction of the out-edges of a single vertex. In contrast, we seek to minimize the overall memory requirements and, thus, we cannot apply the techniques used in `GBASE`; doing so would require at least equivalent amount of memory with non-compressed structures.

A preliminary version of our work appeared in [35]. Here, we propose new representations for weights of out-edges and algorithms requiring mutations. Further, we evaluate our techniques through the execution of additional `Pregel` algorithms and carry out the entire range of our experimentation using settings that suppress the overhead generated from system logging activity.

3 BACKGROUND

Key `Pregel` concepts and structures used for representing adjacency lists by the Apache `Giraph` make up the four-

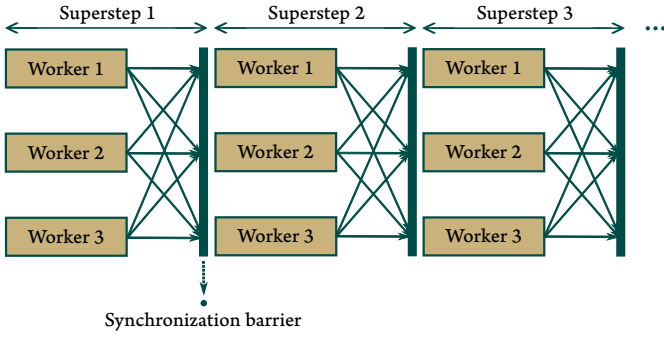


Fig. 2. The Pregel programming model: workers compute in parallel the vertices' actions at every *superstep* and messages between iterations are synchronized using a barrier before every *superstep* commences.

ation upon which we develop our proposed techniques. In this section, we outline both Pregel and Giraph, present empirically-observed properties of real-world graphs, and offer definitions for the encodings to be used by our suggested compression techniques.

3.1 Pregel

Pregel [10] is a computational model suitable for large scale graph processing, inspired by the *Bulk Synchronous Parallel (BSP)* 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 other vertices, to be used in the following *superstep*. Therefore, edges serve as communication channels for the transmission of results. A synchronization barrier between *supersteps* ensures 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. This programming model is illustrated in Figure 2.

Pregel loads the input graph and performs all associated computations in-memory. Thereby, Pregel only supports graphs whose edges entirely fit in main-memory. Regarding the management of out-edges, the basic operations provided by Pregel API are the initialization of adjacency lists, the retrieval of the out-edges, and mutations of out-edges, i.e., additions and removals. For example in Figure 1, vertex 1 maintains a list of its neighbors: 2, 3, and 4; Pregel algorithms need to be able to initialize such a list, retrieve its elements, and possibly add or remove elements.

3.2 Apache Giraph

The Apache Software Foundation has spear-headed the implementation of Giraph [3], an open-source implementation of Pregel that operates atop HDFS and uses *map-only* Hadoop jobs for its computations. The project has been in rapid development since Facebook released its own graph

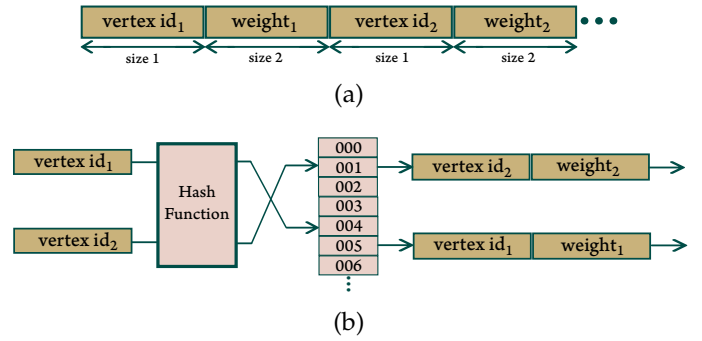


Fig. 3. Giraph's adjacency-list representations: *ByteArrayEdges* (a) and *HashMapEdges* (b).

search service based on an earlier Giraph release. A key Facebook contribution is related to the system's memory optimization. Giraph used separate Java objects for all data types that needed to be maintained, including the out-edge representation (*HashMapEdges*). A new representation, namely *ByteArrayEdges*, significantly reduced the memory usage as well as the number of objects being maintained by serializing edges as byte arrays instead of instantiating native Java objects. Below, we outline these two widely used Giraph data structures to highlight their difference when it comes to memory usage. We note that Giraph's configuration allows for specifying the representation of out-edges to be used and maintains an object of the respective class for each vertex of the graph. Extending Giraph with a new out-edge representation is as simple as writing your own class that implements the *OutEdges* interface.

- *ByteArrayEdges*: The default Giraph structure for holding the out-neighbors of a vertex is that of *ByteArrayEdges* [1]. This representation is realized as a byte array, in which target vertex ids and their respective weights are held consecutively, as Figure 3(a) illustrates. The bytes required per out-neighbor 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.

- *HashMapEdges*: An earlier and more “memory-hungry” representation for holding the out-neighbors of a vertex is *HashMapEdges*. This representation is backed by a hashtable that maps target vertex ids to their respective weights as Figure 3(b) illustrates. *HashMapEdges* offer constant time mutations to the adjacency list of a vertex but are very inefficient space-wise. In particular, the memory cost of maintaining out-edges is up to 10 times larger with *HashMapEdges* than it is with *ByteArrayEdges* [1].

3.3 Properties of Real-World Graphs

Over the last two decades, studies of real-world graphs have led to the identification of common properties that the graphs in question exhibit [12], [17], [36]. In this context, we list here four empirically-observed properties of real-world graphs that are frequently encountered and allow for effective compression. We begin with two such properties of

web graphs that occur when their vertices are ordered lexicographically by URL [12], [17]. We note that even though there is no equivalent way of ordering vertices of other types of graphs, the same properties arise when we apply appropriate reordering algorithms [18], [37], [38]. Then, we list two additional properties observed in realistic graphs from various domains, related to the distribution of node degrees and edge weights. More specifically, the following properties of real-world graphs may be exploited:

- *Locality of reference*: this property states that the majority of the edges of a graph link vertices that are close to each other in the order.
- *Similarity (or copy property)*: vertices that are close to each other in the order tend to have many common out-neighbors.
- *Heavy-tailed distributed degrees*: a constrained number of vertices demonstrate high-degree, whereas the majority of vertices exhibit low-degree. Consequently, graphs created by human activity are generally sparse.
- *Right-skewed weight distributions*: Statistical analysis of weighted graphs shows that the weights of edges are right-skewed distributed [39].

3.4 Codings for Graph Compression

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

- 1) *Unary coding*: the unary coding of x consists of $x - 1$ zeros followed by a 1, e.g., the unary coding of 2 is 01.
- 2) *Minimal binary coding* over an interval [40]: 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 . As an example, the minimal binary coding of 8 in $[0, 56 - 1]$ is 010000, as $8 = 2^{\lceil \log 56 \rceil} - 56$ and therefore we need the $8 - 56 + 2^6 = 16$ -th binary word of length 6.
- 3) *Elias' γ coding*: the γ coding of x consists of l in unary, followed by the last $l - 1$ digits of b , e.g., b of 2 is 10, thus l in unary is 01 and the γ coding of 2 is 010.
- 4) *ζ coding* with parameter k [40]: 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 $[0, 2^{(h+1)k} - 2^{hk} - 1]$. As an example, 16 is ζ_3 -coded to 01010000, as $16 \in [2^3, 2^6 - 1]$, thus $h = 1$ and the unary of $h + 1 = 2$ is 01, and the minimal binary coding of $16 - 2^3$ over the interval $[0, 2^6 - 2^3 - 1]$ is 010000, as shown above.

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. Handling zero is achieved by adding 1 before coding and subtracting 1 after decoding. In the following representations, when no coding is mentioned, the unencoded binary representation is being used.

4 OVERVIEW OF OUR APPROACH

In this section we describe in detail the space-efficient data structures we suggest for the representation of a vertex's neighbors in a graph. We first propose three compressed out-edge representations that enable the efficient execution of graph algorithms in modest settings. Then, we extend these representations to additionally support weighted graphs, by providing a structure to hold the weights of out-edges. Finally, we propose a compact tree-based out-edge representation that provides significant space and efficiency earnings, favors mutations and offers type-flexibility.

Some centralized graph compression methods, as [12], focus on the compression of the adjacency lists, while others, for example [15], are based on the compact representation of the adjacency matrices. In this work, as we follow a vertex-centric approach, we consider both of these approaches at a vertex level. In particular, we are unable to exploit certain properties that centralized graph compression methods use, such as the *similarity* property, as each vertex in `Pregel` is unaware of the information present in other vertices. However, we are able to take into account all the other properties described in Section 3.3.

4.1 Representations based on consecutive out-edges

A common property of graphs created by human activity is *locality of reference*: Vertices, adhering to the orderings mentioned in Section 3.3, tend to neighbor with vertices of similar ids. This 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 [12]. In particular, [12] 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* need to be kept. This information is further compressed using gap *Elias' γ coding*. The remaining out-edges, termed *residuals*, are compressed using ζ coding.

We build on these ideas and introduce two compressed representations that exploit *locality of reference* in a similar fashion but are applicable to `Pregel`-like systems. We consider that neighbors are sorted according to their id, as the case is in [12]. We also note that both of our structures based on consecutive out-edges do not favor mutations, as any addition or removal of an edge would require a complete reconstruction of the compact representation to discover the new set of intervals and residuals.

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 [12] to the restrictions imposed by `Pregel`. In particular, we use the ideas of distinguishing *intervals* and *residuals*, as well as applying appropriate codings on them. The compressed data structure discussed in [12] considers the whole graph and exploits the current vertex's id during compression. However, the vertex id 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

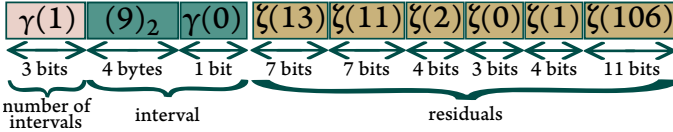


Fig. 4. The storage of neighbors in *BVEEdges*, detailed in Example 1. $\gamma(x)$ and $\zeta(x)$ denote the γ and ζ encodings of x respectively.

our structure as a reference to proceed with gap encoding. As the case is with [12], 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 [12]; shorter *intervals* are more compactly stored as *residuals*. We note here that [12] uses *copy lists* to exploit the *similarity property*. However, using *copy lists* in a vertex-centric distributed environment is infeasible.

Definition 1 (*BVEEdges*). Given a list l of a node's neighbors, *BVEEdges* is a sequence of bits holding consecutively: the γ -coded number of intervals in l of length at least 4; for the first such interval, the smallest neighbor id in it and the γ -coded difference of the interval length minus 4; for each of the rest of the intervals, the difference of the smallest neighbor id in it minus the largest neighbor id of the previous interval decreased by one; 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 *BVEEdges* as illustrated in Figure 4. Here, there is only one *interval* of length at least equal to 4: [9..12]. We first store the number of *intervals* using γ coding. Then, we store the leftmost id of the *interval*, i.e., 9, 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 read the number of *intervals*. For the

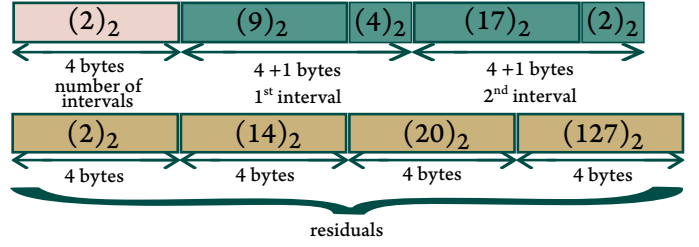


Fig. 5. The storage of neighbors in *IntervalResidualEdges*, detailed in Example 2. $(x)_2$ is the binary representation of x .

first interval, we read the id of the smallest neighbor in it and decode its length. For each of the rest of the intervals, we construct the smallest neighbor id by adding to the next γ -coded value the largest neighbor id of the previous interval incremented by one, and decode its length. After we process the specified number of *intervals*, we decode the *residuals* one by one.

4.1.2 *IntervalResidualEdges*

Our second compressed out-edge representation, namely *IntervalResidualEdges*, also incorporates the idea of using *intervals* and *residuals*. However, we propose a different structure to avoid costly bit stream I/O operations. In particular, we 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, 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 dedicated byte of each *interval* allows us to compress the adjacency list significantly, while also avoiding the use of expensive encodings and bit streams.

Definition 2 (*IntervalResidualEdges*). Given a list l of a node's neighbors, *IntervalResidualEdges* is a sequence of bytes 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 5. 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: we first read the number of *intervals*; while there are still unread *intervals*, we read 5-bytes, i.e., the leftmost element of the *interval* and its length, and recover one by one the elements of the *interval*. When all out-edges that are grouped into *intervals* are retrieved, we read in the *residuals* directly as integers.

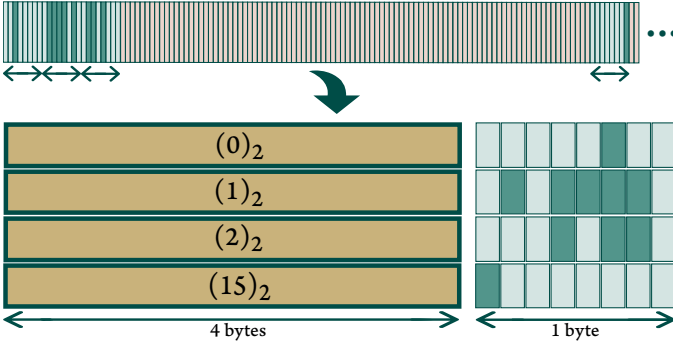


Fig. 6. A bit-array representation of an adjacency list and the storage of these neighbors in *IndexedBitArrayEdges*, detailed in Example 3. $(x)_2$ denotes the binary representation of x .

4.2 IndexedBitArrayEdges

Our first two representations are based on the presence of *consecutivity* among the neighbors of a vertex. Here we propose a representation termed *IndexedBitArrayEdges*, that takes advantage of the *concentration* of edges in *specific areas* of the adjacency matrix, regardless of whether these edges are in fact consecutive. With *IndexedBitArrayEdges* we use a single byte to depict eight possible out-neighbors. 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 really exists. As the neighbor ids of each node tend to concentrate within a few areas, the number of intervals we need to represent 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.

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 6. In the top part we see the bit-array r representation of this adjacency list. The quotient and remainder of each node id divided by 8 give us the approximate position (octet) and the exact position of the node in r , respectively; hence, as depicted in the bottom part of Figure 6, 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 the first part of the third element, denoted by $(2)_2$. Moreover, the remainders of the ids 17, 18 and 20 divided by 8 are 1, 2, and 4 respectively, and so the 2nd, 3rd and 5th flags from the right side of the same element are set to 1 to depict these neighbors.

Accessing the out-edges of a vertex is performed as follows: 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.

We note that *IndexedBitArrayEdges* is able to represent graphs that are up to 8 times larger than the maximum size achieved with *ByteArrayEdges* and 32-bit integers. Hence, we expect its space-efficiency against *ByteArrayEdges* will be even more evident when dealing with a graph of this size. In addition, this representation is clearly more suitable for supporting mutations as opposed to our other two suggested techniques. The addition of an edge in the graph requires us to search linearly the 5-byte elements to ascertain whether we have already indexed the corresponding byte. If that is the case, we merely have to change a single flag in that byte. Otherwise, we have to append a 5-byte element at the end of the structure with the new index number (4-bytes) plus one byte with one *specific* flag set to 1. Obviously, *IndexedBitArrayEdges* does not require that the out-edges are sorted by their id, an assumption that our two other compressed representations make. To remove an edge from the structure, we again have to search for the element with the corresponding index, and set a specific flag to 0. In the case of ending up with a completely empty byte, removing the 5-byte element would be costly. However, this cost is imposed only when elements are left completely empty. Hence, removals are more efficient than with *ByteArrayEdges*, in which the cost is imposed for every out-edge removal. Moreover, there is no inconsistency in keeping the element in our representation, only some memory loss which can be addressed via marking elements when they empty so that they be used in a subsequent neighbor addition.

4.3 VariableByteArrayWeights

Our proposed *BVEEdges* and *IntervalResidualEdges* consider ordered lists of neighbors. Thus, they can be easily modified to support weighted graphs through the use of an additional array, holding the respective weights of the neighbors. This array could simply adapt the format of *ByteArrayEdges* and maintain only the weight of each neighbor in its uncompressed binary format. However, statistical analysis of weighted graphs has shown that the weights of edges exhibit right-skewed distributions [39], [41]. Therefore, there is strong potential for memory optimization in using a compressed weight representation.

Variable-byte coding [42] uses a sequence of bytes to provide a compressed representation of integers. In particular, when compressing an integer n , the seven least significant bits of each byte are used to code n , whereas the most significant bit of each byte is set to 0 in the last byte of the sequence and to 1 if further bytes follow. Hence, variable-byte coding uses $\lfloor \log_{128}(n) \rfloor + 1$ bytes to represent an integer n . The advantage of this approach over the more compact bitwise compression schemes, such as Golomb-Rice, is the significantly faster decompression time it offers due to byte-alignment. In particular, Scholer et al. [43] show that when using the variable-byte coding scheme, queries are executed twice as fast as with bitwise codes, at a small loss of compression efficiency.

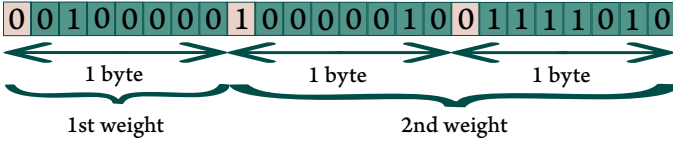


Fig. 7. The storage of edge weights using `VariableByteArrayWeights`. Weights of neighbors are held in variable-byte encoding. Two weights (32 and 378) are represented using only one and two bytes, respectively. `VariableByteArrayWeights` can extend `BVEEdges` and `IntervalResidualEdges` to support weighted graphs.

Definition 4 (VariableByteArrayWeights). Given a list l of a node’s edge weights sorted according to the id of their respective neighbor, `VariableByteArrayWeights` is a sequence of bytes holding consecutively the weights in variable-byte coding.

Example 4. Consider the following sequence of edge weights to be represented: (32, 378). Figure 7 provides an illustration of the parallel array using variable-byte coding that we extend `BVEEdges` and `IntervalResidualEdges` with, to support weighted graphs. The weight of the first neighbor is represented using only one byte, and thus the most significant bit of the latter is set to 0. In contrast, the second weight requires two bytes, the first of which has its most significant bit set to 1, to signify that the following byte is also part of the same weight.

Extracting the weight of a neighbor is as simple as reading a sequence of bytes until reaching one with the most significant bit set to 0, and using the 7 least significant bits of each byte in the sequence to decode the weight.

4.4 RedBlackTreeEdges

Compressed representations essentially limit the efficiency of performing mutations. Even the non-compressed `ByteArrayEdges` representation is impractical when executing algorithms involving mutations of edges [7]. This is due to the excessive time required to perform a removal, as all out-edges need to be deserialized. However, we can achieve mutation efficiency without the overwhelming memory overhead induced when using `HashMapEdges`. Java’s `HashMap` objects use a configurable number of buckets in their hash-table, which doubles once their entries exceed a percentage of their current capacity. Giraph sets the initial capacity of each `HashMap` to be equal to the number of out-edges of the corresponding adjacency list, thus ending up with significantly more buckets than what is needed at initialization. Furthermore, the iterator of out-edges for this representation requires additional $O(n)$ space.

The space wasted when using a `HashMap` due to empty hash-table buckets and additional memory requirements for iterating elements motivated us to implement a *red-black tree*-based representation¹ offering the same type flexibility provided by `HashMapEdges`. Even though the individual entries of the tree require more space, we noticed that the total memory used is reduced by more than 15% for graphs of our dataset, as our tree does not waste space for empty buckets. These savings can be significantly enhanced

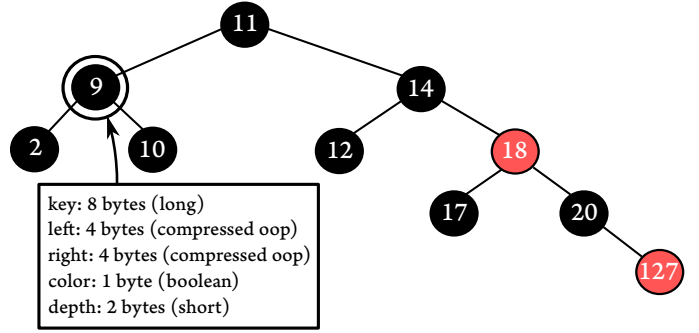


Fig. 8. The storage of neighbors in `RedBlackTreeEdges`. Neighbors’ ids are inserted as keys to a *red-black tree*. For weighted graphs each node would additionally maintain a variable to hold the weight.

through the use of primitive data types. Moreover, using Morris’ tree traversal algorithm [44], we can iterate through the out-edges without additional cost in space. Based on these observations, we developed `RedBlackTreeEdges`, a space-efficient representation that favors mutations and offers type-flexibility.

Definition 5 (RedBlackTreeEdges). Given a list l of a node’s neighbors, each one potentially associated with an edge weight, `RedBlackTreeEdges` is a *red-black tree* which uses the id of a neighbor as a key. The nodes of the tree comprise two references to their left and right child, a boolean for the color of the node, a short for its depth, and two variables using primitive data types for the id and the weight. The memory requirements of the key and the weight depend on the id’s respective primitive data type. The references on the left and right children require 4 bytes each—when the maximum heap size for each worker is less than 32GB and thus *compressed ordinary object pointers (oops)* can be used—or 8 bytes each otherwise.

Example 5. The representation of the aforementioned sequence of neighbors (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) using `RedBlackTreeEdges` is illustrated in Figure 8. We observe that the neighbors’ ids are inserted as keys to a *red-black tree*. For each id, a tree node is created and holds the id as a key, references to the left and right child of the node, the color of the node, and the depth of the node. In case of a weighted graph, each node additionally maintains a variable to hold the weight. In this example, we consider that keys are long integers, and thus require 8 bytes. In addition compressed oops can be used, so the references to the left and right child need 4 bytes each. The graph is unweighted so no bytes are required for the weights, and finally, for the color and the depth of the node, 1 and 2 bytes are needed, respectively, as is always the case.

The use of a *red-black tree* instead of a *hash-table* allows us to access the neighbors without inducing further costs space-wise, and to avoid resizing as neighbors are added or removed. This leads to significantly less memory requirements than with `HashMapEdges`, without forgoing the efficiency of performing mutations. The use of primitive data types instead of generic types necessitates defining suitable Java classes for the input graph; however, this

1. Java’s `TreeMap` uses an unnecessary parent reference.

graph	vertices	edges	type
uk-2007-05@100000	100,000	3,050,615	web
uk-2007-05@1000000	1,000,000	41,247,159	web
ljournal-2008	5,363,260	79,023,142	social
indochina-2004	7,414,866	194,109,311	web
hollywood-2011	2,180,759	228,985,632	social
uk-2002	18,520,486	298,113,762	web
arabic-2005	22,744,080	639,999,458	web
uk-2005	39,459,925	936,364,282	web
twitter-2010	41,652,230	1,468,365,182	social
sk-2005	50,636,154	1,949,412,601	web

TABLE 1

Dataset of our experimental setting with a total of ten publicly available web and social network graphs [12], [18].

is insignificant when compared to our space earnings. Instead of reducing the total memory requirements 15 percentage points we are able to achieve significantly higher savings, as we will show in our experimental evaluation. Besides, the majority of publicly available graphs use *integer* ids, and Facebook uses *long integers*, which are represented at the same cost with `RedBlackTreeEdges`, due to JVM alignment. We also note that the ordering of the vertices' labels does not impact the performance of this representation which is applicable to graphs with in-situ node labelings.

5 EXPERIMENTAL EVALUATION

We implemented our techniques using Java and compared their performance against Giraph's out-edge representations using a number of publicly available and well-studied web and social network graphs [12], [18], reaching up to 2 billion edges. Our implementation is available online.² 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:

- How much more space-efficient is each of our three compressed out-edge representations compared to Giraph's default representation?
- Are our techniques competitive speed-wise when memory is not a concern?
- How much more efficient are our compressed representations when the available memory is constrained?
- Can we execute algorithms for large graphs in settings where it was not possible before?
- Is our compressed weight representation able to induce additional gains?
- What are the benefits of using our tree-based out-edge representation instead of Giraph's fastest representation for algorithms involving mutations?

5.1 Experimental Setting

Our dataset consists of 10 web and social network graphs of different sizes. The properties of these graphs are detailed in Table 1. We ran our experiments on a Dell PowerEdge R630 server with an Intel[®]Xeon[®] E5-2630 v3, 2.40 GHz with 8 cores, 16 hardware threads and a total of 128GB of RAM. Our cluster comprises eight virtual machines running

2. <https://goo.gl/hJIG8H>

Function: computePageRank(vertex, messages)

```

1 begin
2   if superstep ≥ 1 then
3     sum ← 0;
4     foreach message ∈ messages do
5       sum ← sum + message;
6       vvertex ←  $\frac{1-\alpha}{|V|} + \alpha \times \text{sum}$ ;
7   if superstep < MAX_SUPERSTEPS then
8     dvertex ← degree(vertex);
9     sendMessageToAllOutEdges( $\frac{v_{\text{vertex}}}{d_{\text{vertex}}}$ );
10  else
11    voteToHalt();

```

Xubuntu 14.04.02 with Linux kernel 3.16.0-30-generic and 13GB of virtual RAM. On this cluster 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 release.

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 the one discussed in [1], viz. `ByteArrayEdges`, which is currently the default Giraph representation for out-edges. To measure the memory usage we loaded each graph using a fixed capacity Java array list to hold the adjacency lists, dumped the heap of the JVM and used the Eclipse Memory Analyzer³ to retrieve the total occupied memory.

Table 2 lists the memory required by the four representations examined here and the representation of [12] in MB. We observe that all our proposed compression techniques have significantly reduced memory requirements compared to `ByteArrayEdges`. As was expected, `BVEEdges`, which essentially also serves as a yardstick to measure the performance of our structures that focus on access-efficiency, outperforms all representations as far as space-efficiency is concerned. In particular, depending on the graph, its memory requirements are always less than 40% of the requirements of `ByteArrayEdges`, and reach much smaller figures in certain cases, e.g., 20.08% for *hollywood-2011*. However, we observe that our novel `IntervalResidualEdges` as well as the less restrictive `IndexedBitArrayEdges`, both of which do not impose any computing overheads, also manage to achieve impressive space-efficiency.

5.3 Execution Time Comparison

In this section, we present results regarding the execution times of Pregel algorithms using our compressed out-edge representations. Reported timings for all our results are averages of multiple executions.

5.3.1 PageRank Computation

PageRank is a popular algorithm employed by many applications that run on top of real world-networks, with (web page/social network users) *ranking* and *fake account detection* being typical examples.

3. <https://eclipse.org/mat/>

graph	ByteArrayEdges	BVEEdges (BV)	IntervalResidualEdges	IndexedBitArrayEdges
uk-2007-05@100000	22.61 MB	6.41 MB (0.96 MB)	7.92 MB	8.91 MB
uk-2007-05@1000000	279.16 MB	67.36 MB (10.54 MB)	82.7 MB	97.79 MB
ljournal-2008	866.36 MB	386.73 MB (117.68 MB)	497.52 MB	648.52 MB
indochina-2004	1,511.67 MB	442.34 MB (48.03 MB)	646.03 MB	554.23 MB
hollywood-2011	1,381.91 MB	287.53 MB (145.85 MB)	613.52 MB	676.88 MB
uk-2002	2,733.6 MB	1,092.82 MB (116.39 MB)	1,224.07 MB	1,255.67 MB
arabic-2005	4,820.09 MB	1,428.97 MB (187.58 MB)	1,674.75 MB	1,849.83 MB
uk-2005	7,401.88 MB	2,383.54 MB (279.45 MB)	2,728.74 MB	2,928.81 MB
twitter-2010	11,189.88 MB	4,628.48 MB (2,600.07 MB)	7,127.76 MB	8,888.50 MB
sk-2005	14,829.64 MB	4,889.85 MB (607.92 MB)	5,657.79 MB	6,354.17 MB

TABLE 2

Memory requirements of Giraph’s `ByteArrayEdges` and our three out-edge representations for the small and large-scale graphs of our dataset. Requirements of BV [12] in a centralized setting are also listed to provide an indication of the compressibility potential of each graph.

A Pregel implementation of PageRank is shown in Function `computePageRank`. In our experimental setting `MAX_SUPERSTEPS` is set to 30 and α is set to 0.85. Every vertex executes the function `computePageRank` at each *superstep*. The graph is initialized so that in *superstep* 0 all vertices have value equal to $\frac{1}{|V|}$. In each of the first 30 (i.e., 0 to 29) *supersteps*, each vertex sends along each out-edge its current PageRank value divided by the number of out-edges (line 9). From *superstep* 1 and on, each vertex computes its PageRank value v_{vertex} as shown in line 6. When *superstep* 30 is reached, no further messages are sent, each vertex votes to halt, and the algorithm terminates.

We expect that any Pregel algorithm not involving mutations would exhibit similar behavior for the different representations with the one reported here for PageRank, as it would also feature the same set of actions regarding out-edges, i.e., initialization and retrieval.

5.3.2 Shortest Paths Computation

Single-source Shortest Paths algorithms [45] focus on finding a shortest path between a single source vertex and every other vertex in the graph, a problem arising in numerous applications.

A Pregel implementation of Shortest Paths is shown in Function `computeShortestPaths`. Initially, the value associated with each vertex is initialized to infinity, or a constant larger than any feasible distance in the graph from the source vertex. Then, using the temporary variable `minDist` the function examines cases that may update this value. There are two such cases: *i*) if the vertex is the source vertex the distance is set to zero, and *ii*) if the vertex receives a message with a smaller value than the one it currently holds, the distance is updated accordingly. When a vertex updates its value it must also send a message to all its out-neighbors to notify them about the newly found path. Each message is set to the updated distance of the vertex plus the weight of the edge that connects the vertex with the respective neighbor. Finally, the vertex votes to halt and remains halted until a message reaches it. The algorithm terminates when all vertices are halted, at which time each vertex holds the value of the shortest path to the source vertex.

The Shortest Paths algorithm involves the same operations as the PageRank algorithm, but additionally serves the purpose of evaluating our techniques on weighted graphs.

5.3.3 Comparison using small-scale graphs

We begin our access time comparison by investigating the performance of our three compressed out-edge represen-

Function: `computeShortestPaths(vertex, messages)`

```

1 begin
2   if superstep == 0 then
3     | vertex.setValue(∞);
4     minDist ← isSource(vertex) ? 0 : ∞;
5     for message in messages do
6       | minDist ← min(minDist, message);
7     if minDist < vertex.getValue() then
8       | vertex.setValue(minDist);
9       for edge in vertex.getEdges() do
10        | sendMessage(edge, minDist + edge.getValue());
11    voteToHalt();

```

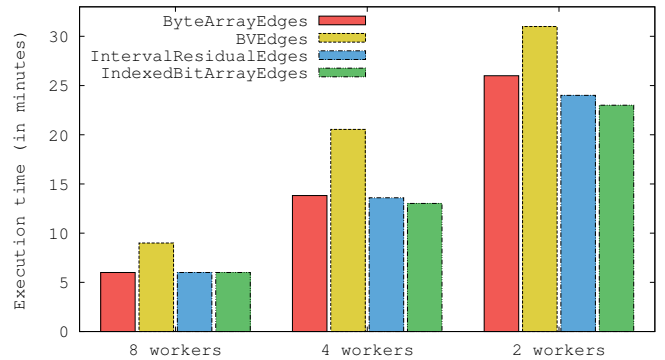


Fig. 9. Execution time (in minutes) of PageRank algorithm for the graph *indochina-2004* using a setup of 2, 4, and 8 workers.

tations, as well as that of Giraph’s `ByteArrayEdges`. Figure 9 depicts the results of all four techniques when executing the PageRank algorithm for the graph *indochina-2004*. We run experiments on setups of 2, 4, and 8 workers and present the results of the total time needed for each representation.

We observe that `IndexedBitArrayEdges` and `IntervalResidualEdges` do not impose any latency in the process. In fact, using either of our two novel representations we achieve execution times for all three setups that are better than those of `ByteArrayEdges`. The performance gain becomes more notable as we limit the number of available workers. `BVEEdges` 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. This indicates that the computing overheads imposed by the *state-of-the-art* techniques of [12] are not negligible and simply adopting them proves to be inefficient. We note that this graph is fairly small for all our

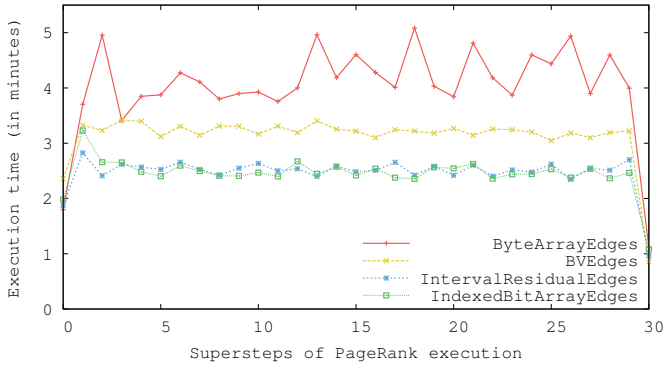


Fig. 10. Execution time (in minutes) for each *superstep* of the PageRank algorithm for the graph *uk-2005* using 5 workers. *ByteArrayEdges* performance fluctuates due to extensive garbage collection.

setups and its memory requirements are not a bottleneck for any of the representations we examine. However, the messages that are exchanged during the execution of the algorithm need in total more than 65GB of memory. Thus, garbage collection needs to take place in the setups of 2 and 4 workers.

For graphs which are equivalent to or smaller than *indochina-2004* the performance is similar. In particular, for all three setups *IndexedBitArrayEdges* and *IntervalResidualEdges* managed to execute the PageRank algorithm faster than *ByteArrayEdges* was able to. On the contrary, *BVEEdges* required more time for each *superstep*.

5.3.4 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. This forces the JVM to work too hard and results in wasting a significant proportion of the total processing time performing garbage collection. Hence, the overall performance degrades extremely. In particular, we examine the behavior of all four representations for the graph *uk-2005*, using a setup of 5 workers, i.e., the smallest possible setup that can handle the execution of PageRank using *ByteArrayEdges*.

The merits of memory optimization in the execution of Pregel algorithms for large scale graphs are evident in Figure 10. In particular, Figure 10 depicts the time needed for each *superstep* of the execution of PageRank for the *uk-2005* graph with each one of the four space-efficient out-edge representations. We observe that *BVEEdges* requires significantly more time than our other two representations for every *superstep*, as was the case with small-scale graphs. In particular, using *BVEEdges* most *supersteps* require more than 3 minutes each, whereas using our other two representations most *supersteps* need about 2.5 minutes each. We also see, however, that in this setup the execution with *ByteArrayEdges* tends to fluctuate in performance, and consequently performs worse than our slowest structure, i.e., *BVEEdges*. 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 col-

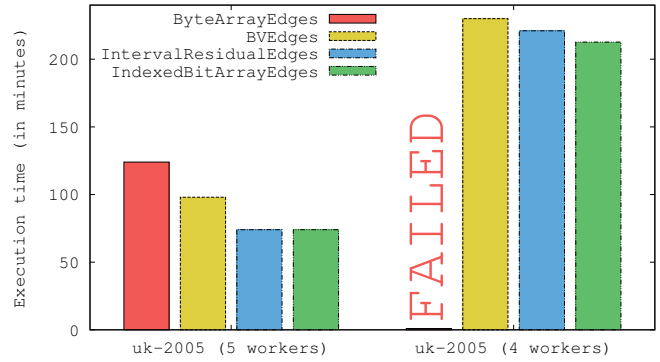


Fig. 11. Execution time (in minutes) of the PageRank algorithm for the graph *uk-2005* using 5 and 4 workers. *IntervalResidualEdges* and *IndexedBitArrayEdges* outperform *ByteArrayEdges* which fails to complete execution with 4 workers.

lection very frequently to accommodate the memory objects required in every *superstep*. *IndexedBitArrayEdges* and *IntervalResidualEdges* were able to handle every *superstep* at a steady pace and greatly outperformed *ByteArrayEdges*, requiring 2.45 and 2.46 minutes of execution per *superstep*, respectively, when in fact *ByteArrayEdges* needed 4.03. Our most compact structure, i.e., *BVEEdges* required 3.13 minutes per *superstep* to run the PageRank algorithm, which is also significantly faster than Giraph’s default representation.

The performance difference of the four representations with regard to the total execution time of PageRank for the graph *uk-2005* is even more evident in Figure 11. The executions using *IndexedBitArrayEdges* and *IntervalResidualEdges* are faster by 40.63% and 40.01% than the one with *ByteArrayEdges*, respectively.

We further evaluate the performance of the four representations by executing PageRank for the same graph using only 4 workers. As already mentioned, 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. The total time needed by our three representations is also illustrated in Figure 11. We observe that under these settings *IndexedBitArrayEdges*, *IntervalResidualEdges*, and *BVEEdges* need 212.65, 221.27, and 230.47 minutes, respectively. As we can see in Table 2, *IndexedBitArrayEdges* 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 using *IntervalResidualEdges*, which results in it being 4% faster under these settings.

We note that for the *uk-2005* graph, PageRank execution requires the exchange of messages that surpass 313GB of memory in total.

5.3.5 Initialization time comparison

Having measured the execution time of the PageRank algorithm using small- and large-scale graphs we

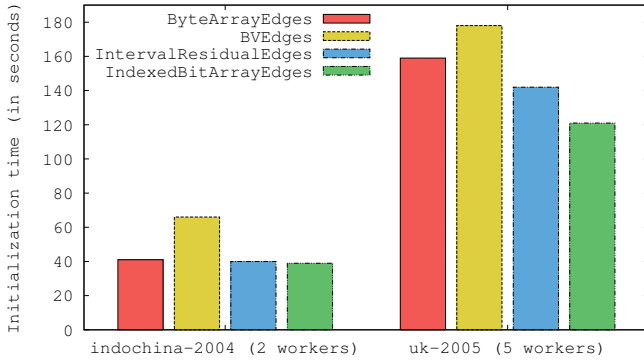


Fig. 12. Initialization time (in seconds) for graphs *indochina-2004* (using 2 workers) and *uk-2005* (using 5 workers). There is notable performance gain on large-scale graphs over *ByteArrayEdges* when using *IndexedBitArrayEdges* or *IntervalResidualEdges*. *BVEEdges* is the slowest of the representations examined.

now report the initialization time the different representations need. Figure 12 illustrates a comparison between our three space-efficient out-edge structures and *ByteArrayEdges* in two different setups. In particular, we first examine the loading time for the relatively small graph *indochina-2004* when using two workers. We observe that there are negligible differences between *ByteArrayEdges*, *IntervalResidualEdges*, and *IndexedBitArrayEdges*, with the former being the slowest and the latter being the fastest. In contrast, *BVEEdges* is significantly slower than all other representations. Furthermore, we see in Figure 12 that when loading a larger graph, i.e., *uk-2005*, the performance of the different structures varies considerably. Again, *IndexedBitArrayEdges* is the fastest approach, followed by *IntervalResidualEdges*, *ByteArrayEdges*, and *BVEEdges*, but in this setting there is obvious disparity in the initialization performance.

We note that the graph loading time is negligible compared to the execution time of the PageRank algorithm. For instance, for graph *uk-2005* using 5 worker nodes, *IndexedBitArrayEdges* requires 121.64 seconds to initialize the graph, whereas the execution time for this setting is over 70 minutes using any of the representations examined here. However, the significant performance gain induced when using *IntervalResidualEdges* and *IndexedBitArrayEdges* can have a notable impact in algorithms requiring less execution time.

5.3.6 Comparison when using weighted graphs

We continue our experimental evaluation by measuring the time needed for the execution of a Pregel algorithm that operates on weighted graphs. In particular, we present performance results for the different compact out-edge representations when executing the Shortest Paths algorithm, as described through Function `computeShortestPaths`. Being that all the graphs of our dataset are unweighted, we assign random weights exhibiting a Zipf distribution⁴ on the edges of graph *uk-2005*. Then, we proceed with the execution

4. We used the `numpy.random.zipf` function from NumPy’s random sampling library to generate weights for the graph using $\alpha = 2$.

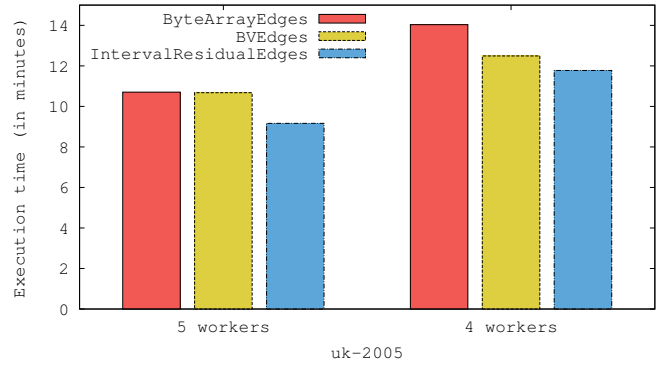


Fig. 13. Execution time (in minutes) of the ShortestPaths algorithm for a single vertex, on the graph *uk-2005*, using a setup of 5 and 4 workers.

of the algorithm using *ByteArrayEdges*, *BVEEdges* and *IntervalResidualEdges* in setups of 5 and 4 workers. For *BVEEdges* and *IntervalResidualEdges* we additionally use the *VariableByteArrayWeights* representation to hold the weights of edges.

Figure 13 illustrates a comparison of the results we obtain with our representations against Giraph’s *ByteArrayEdges*. We observe that using *IntervalResidualEdges* we are able to execute the Shortest Paths algorithm more than 1.5 minutes faster than using *ByteArrayEdges* in the setup of 5 workers. The significant savings in execution time are due to the limited memory usage of *IntervalResidualEdges* and *VariableByteArrayWeights*. Our *BVEEdges* behaves similarly to *ByteArrayEdges* as the computation overhead involved in accessing the edges counterbalances the merits of space-efficiency this representation offers. Furthermore, Figure 13 shows the respective results for the setup of 4 workers. We see that as we limit the available memory resources, the performance gains of our representations become more evident. In particular, *BVEEdges* is clearly also preferable than *ByteArrayEdges* in this setting being more than 1.5 minute faster. Moreover, *IntervalResidualEdges* is able to terminate 2.26 minutes faster than *ByteArrayEdges*.

We note that *VariableByteArrayWeights* requires additional 1,957.25MB of memory to hold the weights of out-edges, whereas *ByteArrayEdges* needs 5,074.36MB. Moreover, *IndexedBitArrayEdges* does not presume that the ids of out-edges are sorted, and thus, cannot support weighted graphs through *VariableByteArrayWeights*. For this reason we do not include *IndexedBitArrayEdges* in this experiment.

5.3.7 Comparison when performing mutations

All the aforementioned experiments focus on space-efficient structures that are applicable on algorithms that do not involve additions or removals of out-edges. However, often-times graph algorithms need to perform mutations on the vertices’ neighbors. To this end, we examine here the performance of our novel *RedBlackTreeEdges* representation, against Giraph’s *HashMapEdges*. Both structures provide type flexibility, support weighted graphs, and can operate on graphs with in-situ node labelings.

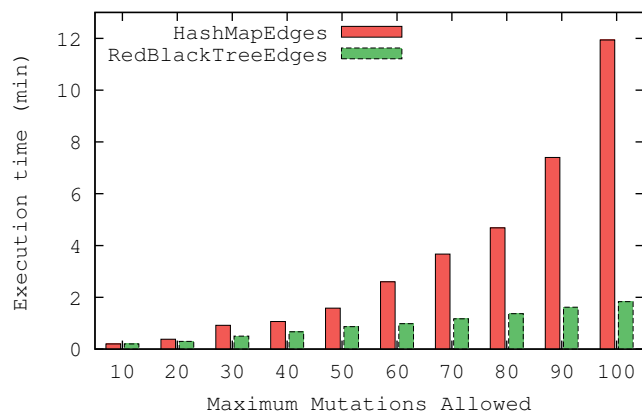


Fig. 14. Execution time (in minutes) of an algorithm performing a random number of mutations on the graph *hollywood-2011* using 5 workers, for a varying number of maximum mutations allowed.

We consider here an input graph which uses long integers for the ids of its nodes⁵ and the execution of a simple algorithm that performs additions and removals of out-edges on this graph. In particular, we executed over *hollywood-2011*—the largest graph we were able to load using `HashMapEdges`—an algorithm of two *supersteps*. The first one performs a random number of insertions of out-edges, and the second one removes them.

The initialization phase, in which the graph is loaded in memory, is faster using our novel tree-based structure. `RedBlackTreeEdges` needs 33.64 seconds to do so, whereas `HashMapEdges` requires 38.73 seconds. Moreover, Figure 14 depicts the execution time needed by the two representations when varying the number of maximum insertions/deletions allowed in our algorithm. We observe that when the number of mutations is low, the time spent using the two representations is equivalent. However, as the number of mutations grows and more memory is needed for the representation of the graph, the performance of `HashMapEdges` deteriorates significantly, and `RedBlackTreeEdges` proves to be clearly superior.

We note that `RedBlackTreeEdges` requires less than half of the space that `HashMapEdges` needs to load the graph *hollywood-2011* in memory. In particular, `RedBlackTreeEdges` uses 7,079.6MB of memory, whereas `HashMapEdges` uses 19,323.8MB.

6 CONCLUSION

In this paper, we propose and implement three compressed out-edge representations for distributed graph processing, termed `BEdges`, `IntervalResidualEdges`, and `IndexedBitArrayEdges`, a variable-byte encoded representation of out-edge weights, termed `VariableByteArrayWeights`, for compact support of weighted graphs, and a compact tree-based representation that favors mutations, termed `RedBlackTreeEdges`. We focus on the vertex-centric model that all `Pregel`-like graph processing systems follow and examine the efficiency of our structures

5. We examine the case of long integer ids as this is the data type used by *Facebook*, *Giraph*'s most significant contributor.

by extending one such system, namely *Apache Giraph*. Our techniques build on empirically-observed properties of real-world graphs that are exploitable in settings where graphs are partitioned on a vertex basis. In particular, we capitalize on the sparseness of such graphs, as well as the *locality of reference* property they exhibit. We cannot, however, exploit the *similarity* property as vertices are unaware of any information regarding other vertices.

All our representations offer significant memory optimizations that are applicable to any distributed graph compressing system that follows the `Pregel` paradigm. `BEdges`, which is based on *state-of-the-art* graph compression techniques, achieves the best compression but offers relatively slow access time to the graph's elements. Our `IntervalResidualEdges` and `IndexedBitArrayEdges` representations outperform *Giraph*'s most efficient 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. Our compressed out-edge representations are also shown to allow for efficient execution of weighted graph algorithms, through `VariableByteArrayWeights`, a variable-byte encoding based representation of out-edge weights. Finally, through our evaluation regarding algorithms involving mutations we show that the performance of `RedBlackTreeEdges` is equivalent to that of `HashMapEdges` when memory is sufficient, and shows significant improvements otherwise.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful remarks, as well as Prof. Yannis Smaragdakis and Michael Sioutis for fruitful discussions and valuable feedback. A preliminary version of this work appeared in [35].

REFERENCES

- [1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale," *Proc. of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.
- [2] "We knew the web was big. . . ." <http://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html>.
- [3] "Apache Giraph," <http://giraph.apache.org/>.
- [4] S. Salihoglu and J. Widom, "GPS: a graph processing system," in *Proc. of the 25th Int. Conf. on Scientific and Statistical Database Management, Baltimore, MD, USA, July 29 - 31, 2013*, 2013, pp. 22:1–22:12.
- [5] 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*, vol. 5, no. 8, pp. 716–727, 2012.
- [6] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation," in *Proc. of the 24th Int. Conf. on World Wide Web, Florence, Italy, May 18-22, 2015*, 2015, pp. 1307–1317.
- [7] 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*, vol. 7, no. 12, pp. 1047–1058, 2014.
- [8] J. Ugander and L. Backstrom, "Balanced Label Propagation for Partitioning Massive Graphs," in *Proc. of the 6th ACM Int. Conf. on Web Search and Data Mining, Rome, Italy, February 4-8, 2013*, pp. 507–516.

- [9] A. Zheng, A. Labrinidis, P. K. Chrysanthos, and J. Lange, "Argo: Architecture-aware graph partitioning," in *2016 IEEE Int. Conf. on Big Data, Washington DC, USA, December 5-8*, pp. 284–293.
- [10] 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 *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 135–146.
- [11] 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 *Proc. of the Int. Conf. on Management of Data, Snowbird, UT, USA, June 22-27, 2014*, pp. 1371–1382.
- [12] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *Proc. of the 13th Int. Conf. on World Wide Web, New York, NY, USA, May 17-20, 2004*, pp. 595–602.
- [13] A. Apostolico and G. Drovandi, "Graph compression by BFS," *Algorithms*, vol. 2, no. 3, pp. 1031–1044, 2009.
- [14] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. of the 15th Int. Conf. on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, pp. 219–228.
- [15] N. Brisaboa, S. Ladra, and G. Navarro, "k2-Trees for Compact Web Graph Representation," in *String Processing and Information Retrieval, 2009*, vol. 5721, pp. 18–30.
- [16] P. Liakos, K. Papakonstantinou, and M. Sioutis, "Pushing the Envelope in Graph Compression," in *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management, Shanghai, China, 2014*, pp. 1549–1558.
- [17] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe, "The link database: Fast access to graphs of the web," in *Proc. of the 2002 Data Compression Conference, 2-4 April, Snowbird, UT, USA, 2002*, pp. 122–131.
- [18] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks," in *Proc. of the 20th Int. Conf. on World Wide Web, Hyderabad, India, March 28 - April 1, 2011*, pp. 587–596.
- [19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, CA, USA, October 8-10, 2012*, pp. 17–30.
- [20] M. Han and K. Daudjee, "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 950–961, May 2015.
- [21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283.
- [22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA, 2014*, pp. 599–613.
- [23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Berkeley, CA, USA, 2010*, pp. 10–10.
- [24] M. Kabiljo, D. Logothetis, S. Edunov, and A. Ching, "A comparison of state-of-the-art graph processing systems," <https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/>.
- [25] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 377–388, Dec. 2014.
- [26] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," *PVLDB*, vol. 9, no. 12, pp. 936–947, 2016.
- [27] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pp. 403–412.
- [28] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 301–316.
- [29] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 31–46.
- [30] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 45–58.
- [31] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 285–300.
- [32] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 281–292, Nov. 2014.
- [33] P. Liakos, K. Papakonstantinou, and M. Sioutis, "On the effect of locality in compressing social networks," in *Proc. of the 36th Eur. Conf. on IR Research, Amsterdam, The Netherlands, April 13-16, 2014*, pp. 650–655.
- [34] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos, "GBASE: an efficient analysis platform for large graphs," *VLDB J.*, vol. 21, no. 5, pp. 637–650, 2012.
- [35] P. Liakos, K. Papakonstantinou, and A. Delis, "Memory-optimized distributed graph processing through novel compression techniques," in *Proc. of the 25th ACM Int. on Conf. on Information and Knowledge Management, Indianapolis, Indiana, USA, 2016*, pp. 2317–2322.
- [36] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proc. of the 11th Int. Conf. on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pp. 177–187.
- [37] P. Boldi, M. Santini, and S. Vigna, "Permuting web and social graphs," *Internet Mathematics*, vol. 6, no. 3, pp. 257–283, 2009.
- [38] O. Goonetilleke, D. Koutra, T. Sellis, and K. Liao, "Edge labeling schemes for graph data," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, ser. SSDBM '17*. New York, NY, USA: ACM, 2017, pp. 12:1–12:12. [Online]. Available: <http://doi.acm.org/10.1145/3085504.3085516>
- [39] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks," *Proc. of the National Academy of Sciences of the United States of America*, vol. 101, no. 11, pp. 3747–3752, 2004.
- [40] P. Boldi and S. Vigna, "The webgraph framework II: codes for the world-wide web," in *Proc. of the 2004 Data Compression Conference, March 23-25, Snowbird, UT, USA, 2004*, p. 528.
- [41] M. McGlohon, L. Akoglu, and C. Faloutsos, "Weighted graphs and disconnected components: patterns and a generator," in *Proc. of the 14th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pp. 524–532.
- [42] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *Comput. J.*, vol. 42, no. 3, pp. 193–201, 1999.
- [43] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, "Compression of inverted indexes for fast query evaluation," in *Proc. of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland*, pp. 222–229.
- [44] J. M. Morris, "Traversing binary trees simply and cheaply," *Information Processing Letters*, vol. 9, no. 5, pp. 197–200, 1979.
- [45] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, no. 2, pp. 129–174, 1996.

Panagiotis Liakos is a Ph.D. student at the University of Athens, from where he holds B.Sc. and M.Sc. degrees. His research interests are in Graph Mining and Information Retrieval.

Katia Papakonstantinou is a Research Associate at the University of Athens and an Adjunct Lecturer at the Athens University of Economics and Business. She holds B.Sc., M.Sc., and Ph.D. degrees from the University of Athens. Her research interests are in Algorithmic Game Theory and Social and Information Network Analysis.

Alex Delis is a Professor of Computer Science at the University of Athens and the New York University Abu Dhabi. His research interests are in Distributed and Virtualized Data Systems. He holds a Ph.D. in Computer Science from the University of Maryland at College Park.