The shortest path is not always a straight line

Leveraging semi-metricity in graph analysis

Vasiliki Kalavri
KTH Royal Institute of Technology
Stockholm, Sweden
kalavri@kth.se

Tiago Simas
Telefonica Research
Barcelona, Spain
tiago.simas@telefonica.com

Dionysios Logothetis
Facebook
Menlo Park, CA, USA
dionysios@fb.com

ABSTRACT

In this paper, we leverage the concept of the *metric backbone* to improve the efficiency of large-scale graph analytics. The metric backbone is the minimum subgraph that preserves the shortest paths of a weighted graph. We use the metric backbone in place of the original graph to compute various graph metrics exactly or with good approximation. By computing on a smaller graph, we improve the performance of graph analytics applications on two different systems, a batch graph processing system and a graph database.

Further, we provide an algorithm for the computation of the metric backbone on large graphs. While one can compute the metric backbone by solving the all-pairs-shortest-paths problem, this approach incurs prohibitive time and space complexity for big graphs. Instead, we propose a heuristic that makes computing the metric backbone practical even for large graphs. Additionally, we analyze several real datasets of different sizes and domains and we show that we can approximate the metric backbone by removing only first-order *semi-metric* edges; edges for which a shorter two-hop path exists.

We provide a distributed implementation of our algorithm and apply it in large scale scenarios. We evaluate our algorithm using a variety of real graphs, including a Facebook social network subgraph of ∼50 billion edges. We measure the impact of using the metric backbone on runtime performance in two graph management systems. We achieve query speedups of up to 6.7x in the Neo4j commercial graph database and job speedups of up to 6x in the Giraph graph processing system.

1. INTRODUCTION

Graph analysis is an invaluable tool in several domains, such as Online Social Network (OSN) analysis and web analytics. Many analytical applications encode metrics of distance or similarity in the edge weights [9]. Consider, for instance, the graph in Figure 1 that represents a social network with the link weights representing the social proximity of the relationship. Scientists may then analyze these weighted relationships and calculate metrics, such as node centrality [14], to infer influential users, find optimal paths to propagate information, discover communities [34] or detect fraud. Weighted graphs are also important in recommendations, where weights represent similarity between users or user-item preferences.

As graphs grow bigger, these analytical tasks become challenging. Even with the use of specialized graph management systems [37, 36, 27], several graph metrics are hard to compute. For instance, betweenness centrality requires the computation of all-pairs shortest distances, an operation that incurs high overhead with regard to both computation and storage. Previous work has attempted to address the challenges of computational overhead with approximation [13, 31] and reduce the storage overhead with compression techniques [12, 16, 23].

This paper proposes to leverage the concepts of *semi-metricity* [45] and the *metric backbone*, introduced in [20], for efficient large-scale graph analysis. In a weighted graph, semi-metric edges are direct links for which there exists an indirect shorter path. In the example of Figure 1, the dashed lines represent such semi-metric edges. The metric backbone is the subgraph of a weighted graph that includes no semi-metric edges. In Figure 1, the solid lines represent the metric backbone of the depicted social graph. Effectively, the metric backbone is a reduced representation of a graph, that preserves information about shortest paths. This property has been used to improve recommendation algorithms [46, 44, 48] and more recently, to improve the modularity in community detection [49].

In this work, we further explore how the performance of various large-scale graph analysis tasks can benefit from the concept of the metric backbone. In particular, we apply the metric backbone concept in the context of large-scale graph analysis systems, such as graph databases [3] and batch processing systems [1, 37]. First, we show that, for applications that depend explicitly on the calculation of shortest paths, we can get exact answers, but significantly improve performance by computing on the reduced metric backbone instead. Second, even when shortest paths are not explicitly used, such as when performing reachability queries, the metric backbone can still yield correct answers faster, by reducing the amount of paths that must be explored. Third, we study algorithms, for which the metric backbone does not yield exact answers, but an approximation. Here, we consider PageRank and show that executing the algorithm on the metric backbone produces a good approximation, while

Proceedings of the VLDB Endowment, Vol. 9, No. 9

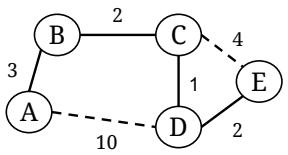

Figure 1: An example graph representing a social network. The edge weights represent the proximity of the social tie. Dashed lines represent semi-metric edges, indicating there is a shorter, indirect path between two nodes.

considerably improving efficiency.

Despite its usefulness, the metric backbone has not been used in large-scale graph analysis yet. This is mainly due to the fact that the calculation of the backbone itself, on big graphs, is challenging. Computing the metric backbone requires solving the APSP (All-Pairs-Shortest-Paths) problem and storing $O(N^2)$ paths, which is prohibitive for large graphs.

In this paper, we address this challenge in two ways. First, we provide an algorithm for the calculation of the backbone, that does not require the computation of APSP. Our algorithm starts by detecting and removing only those semi-metric edges that violate the triangle inequality. Subsequently, it iteratively labels metric edges by performing short breadth-first searches. Second, we show that even an approximation of the metric backbone, where only the semi-metric triangles have been removed, still reduces the size of the original graph significantly, improving performance.

This paper makes the following contributions:

- We analyze a variety of real graphs and show that they exhibit high semi-metricity, even at the first level of transitivity (Section 2).
- We categorize the types of applications which benefit from the metric backbone (Section 2).
- We propose an algorithm for calculating the metric backbone, which does not require the solution of APSP (Section 3).
- We show that the computation of the backbone is practical for large-scale scenarios and provide an open source implementation of the algorithm, on top of the Giraph graph processing system (Section 4).
- We apply the approach inside two graph management systems and evaluate their efficiency with real datasets. First, we show that integrating the metric backbone inside the the Neo4j graph database can speed up typical graph queries by up to 6.7x. Second, we improve the performance of several graph applications in Giraph, reducing execution time by up to 6x and communication by up to 70% (Section 5).

The rest of the paper is structured as follows. In Section 2 we give an overview of our approach and motivate it with a list of real-world examples. Section 3 presents the algorithm for the computation of the metric backbone, while in Section 4 we describe a scalable implementation. In Section 5, we perform an extensive evaluation on algorithm scalability and performance impact. Section 6 outlines the related work, and in Section 7 we conclude.

2. BACKGROUND AND MOTIVATION

We start by describing the concept of the metric backbone. To motivate its use in graph analysis, we first discuss how different graph analysis tasks may benefit from the concept at the algorithm level. Then, we analyze a variety of real data sets, to validate that graphs exhibit high degree of semi-metricity, making this approach effective in real scenarios. Further, we outline how we can apply the concept in graph management systems.

2.1 The metric backbone

The metric backbone is introduced in [20], primarily to improve the accuracy of community detection algorithms [49]. To describe it more formally, let us first introduce the necessary notation. Let $G = (V, E)$ be a graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges, with $(u, v) \in E$ denoting an edge from u to v. We use $d(u, v)$ to denote the weight of an edge (u, v), representing a distance. The weight may represent any application-defined distance metric imposed on the graph, such as communication latency or euclidean distance. Finally, given an acyclic path p, $d(p)$ denotes the total distance of the path.

The utility of the metric backbone is based on the role of *semi-metric* edges in a graph. In a weighted graph, we say that a direct edge between two nodes is semi-metric, if there exists an indirect path between these two nodes, with a shorter distance.

Definition 1. *An edge (u, v) is n^{th}-order* semi-metric *if there exists an alternative path, $u, x_1, ..., x_n, v$, with $n + 1$ edges $(u, x_1), ..., (x_n, v) \in E$, such that $d(u, v) > d(u, x_1) + ... + d(x_n, v)$.*

For instance, in Figure 1, edge CE is 1^{st}-order semi-metric and edge AD is 2^{nd}-order semi-metric. The metric backbone is essentially a subgraph of the original graph that contains no semi-metric edges.

Definition 2. *Given a weighted graph, where weights represent non-negative distance, the metric backbone is the minimum subgraph that preserves the shortest paths of the original graph.*

For instance, in Figure 1 the solid lines represent the metric backbone of the depicted social graph.

The utility of the backbone is not limited to weighted graphs where weights represent distances. Often, instead of a distance metric, relations in graphs are naturally characterized by a similarity metric [49, 15]. For instance, the Jaccard index [30] is a popular similarity metric, that takes into consideration the number of common neighbors between two nodes in a graph. Alternatively, in the context of an OSN, similarity is sometimes related to the amount of interaction between two users, like the number of messages exchanged. In such cases, we can transform similarity to distance, through appropriate functions [49, 15], and still take advantage of the metric backbone for analysis[1].

Once we have computed the metric backbone, we can use it to calculate metrics based on shortest distances, since it maintains this information. Additionally, recent work has shown that removing semi-metric edges from a graph also

[1]While there are various functions available, a simple and commonly used function to convert a similarity metric x to distance is $\varphi(x) = \frac{1}{x} - 1$.

Graph	$\|V\|$	$\|E\|$	metric	%
Facebook[2]	190M	49.9B	Custom	26.5%
Twitter [32]	40M	1.5B	Jaccard	39%
Tuenti [4]	12M	685M	Jaccard	59%
LiveJournal [8]	4.8M	34M	Jaccard	40%
NotreDame [7]	0.3M	1.5M	Jaccard Adamic	45% 29%
DBLP [52]	318K	1M	Jaccard Adamic	23% 9%
Twitter-ego [38]	81K	1.7M	Jaccard Adamic	57% 39%
Movielens [2]	1.6K	1.9M	Jaccard	88%
Facebook [41]	1K	143K	#messages message size	78% 77%
US-Airports [19]	0.5K	6K	#passengers	72%
C-Elegans [50]	0.3K	2.3K	#connections	17%

Table 1: The percentage of semimetric edges on various real graphs under different similarity metrics. In the small Facebook dataset, we use the number of messages or size of messages exchanged between users to measure similarity. In the US-Airports graph we measure similarity as the number of passengers that travel between cities.

allows us to perform community detection [49] or recommendations [44] with improved accuracy.

2.2 Semi-metricity in real graphs

The utility of the metric backbone is based on the observation that many real-world graphs exhibit high degree of semi-metricity. As it has been shown in various contexts, especially in social networks, indirect connections are often stronger than direct ones. For instance, OSN interactions between users, a common metric of social proximity [28], are often more frequent between users who are not directly connected [11, 51]. In fact, this principle has been used, for example, to predict information propagation paths [54], to improve link prediction in OSNs, to provide better recommendations and even to design more efficient storage systems that back OSNs [11].

In general, different real-world graphs present different degrees of semi-metricity. Semi-metricity also varies with the distance metric imposed on the graph. To study the practicality of our approach, we analyze a variety of real-world graph datasets, measuring the degree of semi-metricity. We present results for graphs in several domains, such as OSN, web, authorship, air traffic and biological graphs. We measure semi-metricity under various commonly used metrics, such as the Jaccard [30] and the Adamic-Adar [5] metrics.

In Table 1, we describe the datasets we analyzed and summarize the results. The degree of semi-metricity ranges from 9% to 88% depending on the dataset and metric. The Movielens [2] movie preference graph exhibits the highest semi-metricity, while among the analyzed OSN graphs, Tuenti [4] has the highest semi-metricity of 59%. Further, we see that the same graph may exhibit different semi-metricity for different distance metrics. For instance, the Jaccard similarity metric [30] typically results in more semi-metric edges than the Adamic-Adar metric [5].

Further, in Figure 2 we plot the percentage of semi-metric edges for different orders of semi-metricity, for some of the networks of Table 1. Notice that, in most of the graphs, the vast majority of the semi-metric edges are 1^{st}-order semi-metric. In other words, there are *few* indirect paths with three or more edges that are shorter than any direct edges. Previous work has also identified that the strength of indirect social connections decreases with the length [25, 17].

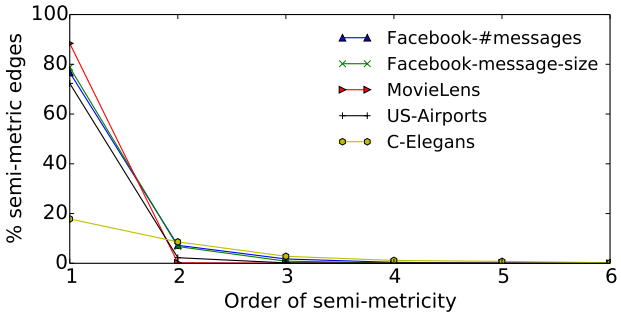

Figure 2: Percentage of semi-metric edges over their order of semi-metricity. For most of the graphs, the majority of edges are 1st-order semi-metric edges.

This analysis reveals an opportunity. First, based on Table 1, we see that in practice, we can run a variety of analytical tasks on a graph that is significantly reduced in size, in some cases, with more than half the edges removed. In Section 5, when we apply the concept in large-scale graph management systems, even seemingly modest degrees of semi-metricity can have a significant impact on application performance. Second, the analysis of Figure 2 shows that we can compute a good approximation of the metric backbone by removing only the 1^{st}-order semi-metric edges. We use this intuition in Section 3, to guide the design of the algorithm for the computation of the metric backbone. This approximation is not the optimal subgraph that gives us the shortest paths, but it is very close to the optimal. Note that the approximate metric backbone has the same properties as the exact metric backbone and it generates the same shortest paths distribution.

2.3 Algorithm classification

The calculation of graph metrics may benefit from the metric backbone in different ways. Here, we divide graph metrics in two classes, depending on how we can exploit the metric backbone for their calculation, and give examples for each class. Note that here, we consider the metrics abstractly; the classification does not depend on the framework or model in which the corresponding algorithms may be programmed and executed. We discuss framework-specific impact in Section 2.4.

We summarize the classification in Table 2. Class A consists of algorithms that we can run unmodified on the metric backbone and get exactly the same answer, as on the original graph. Examples of such algorithms include the calculation of shortest distances or algorithms that depend on computing shortest distances, like betweenness centrality. The metric backbone maintains also the connectivity of the

[2]A subgraph of the Facebook social network representing a geographic area. The graph is weighed with a custom similarity score that integrates a number of user features.

Class	Description	Examples
A	Algorithms we can run unmodified and produce exact result.	Shortest weighted paths, betweeness centrality, closeness centrality, connected components, radius, reachability queries.
B	Algorithms we can run unmodified and produce an approximation.	PageRank, eigenvector centrality, random walks, community detection, clustering.

Table 2: A classification of graph algorithms and metrics that may be computed on the metric backbone. For each class we provide a list of example algorithms.

graph, therefore, we can compute algorithms like connected components and reachability queries.

In Class B, we include algorithms that we can run unmodified on the metric backbone, but may return an approximation of the metric they are intended to calculate. Examples in this category include PageRank and various community detection algorithms [49, 24]. To give an intuition why we can calculate fairly accurate approximations for such algorithms, let us consider PageRank. PageRank is essentially a diffusion process in a network with a damping factor. The damping factor guarantees, regardless of the network topology, that the process will always converge. High damping means that the process will converge in few hops, and low damping means that it will converge in a more wide hop process. Since the metric backbone contains only edges where the flow is high, working as the way of shortest (stronger) communication, we can assume that the main diffusion process ultimately goes through the metric backbone. We empirically validate the accuracy of PageRank approximation when using the metric backbone in Section 5.

The metric backbone reduces the information and naturally, there are graph metrics for which the metric backbone may yield highly inaccurate answers. For instance, we cannot use the metric backbone to calculate the unweighted shortest distances, as it will overestimate the distance for all the pairs of nodes connected by semi-metric edges.

Finally, note that this is not meant to be an exhaustive list of the algorithms we can or cannot use with the metric backbone. We believe that the concept of the metric backbone opens up an opportunity to define more metrics and characterize how to benefit from it.

2.4 Applications

While the benefit of the metric backbone is not specific to a computation model or framework, it manifests in distinct ways when used in the context of graph management systems. In this paper, we use the metric backbone to improve the performance of two types of systems, graph databases and distributed batch processing systems.

Graph databases. Graph databases are used to store and query large graphs. They are optimized for traversals, reachability and pattern matching queries [3]. For example, users can query for paths that satisfy criteria, such as length or the properties of the nodes. For several queries, the metric backbone preserves the semantics. At the same time, executing a query on the metric backbone only, reduces the path search space and may provide significant query speedups. While, in this paper, we apply this technique manually, by re-writing queries to use the metric backbone, we envision that a closer integration with automatic query re-writing will allow for more optimizations and a more user-friendly interface. We evaluate the impact in Section 5.3.

Batch processing systems. Several of these graph metrics are often programmed on top of large-scale graph processing systems, such as Pregel [37, 1] and Graphlab [36]. In such systems, graph algorithms are implemented as parallel per-vertex computations and typically, vertices communicate by exchanging messages. This communication usually occurs along the edges of the graph. The CPU and memory requirements, in such systems, depend on the number of messages that have to be processed, which is typically proportional to the number of the edges of the graph. In Section 5.4, we validate that by reducing the edges of a graph, we reduce communication overhead and resource requirements, eventually improving runtime performance.

Graph compression The metric backbone can also be used as a lossy compression mechanism, as the amount of semi-metric edges directly translates to storage reduction. The last column of Table 1 corresponds to size reduction, when the backbone is used in the place of the original graph.

2.5 Discussion

For algorithms that do not depend on the edge weights of a graph, it might not be clear whether they could benefit from the metric backbone. Actually, iterative applications that use the graph structure to propagate information, might require more iterations to converge, when run on top of the metric backbone.

For example, let us consider the Connected Components problem. In its typical distributed implementation, in every iteration, a node receives the ids of its neighbors, adopts the minimum of these ids and, if its value has changed since the previous iteration, it propagates the new value to its neighbors. Computation stops when none of the nodes changes value. This computation does not utilize the edge weights of the graph, but only the graph structure. The number of iterations necessary for convergence is equal to the maximum graph diameter + 1. When removing edges to generate the metric backbone, the *absolute* paths between some nodes become longer, i.e. the edges removed might be *shortcuts* in the unweighted graph. Thus, by removing them, we increase the graph diameter and consequently, the number of iterations required for convergence.

However, as we show in Section 5, such applications can still benefit from the metric backbone. Even if the algorithm does not depend on the edge weights, when removing a large amount of edges, we notably decrease the communication required, thus, speeding up execution.

3. COMPUTING THE BACKBONE

In this section, we describe the algorithm for computing the metric backbone. We show how to implement it in an efficient and scalable manner in Section 4. The algorithm we propose in this paper assumes a static graph that does not change over time. We believe that incremental maintenance

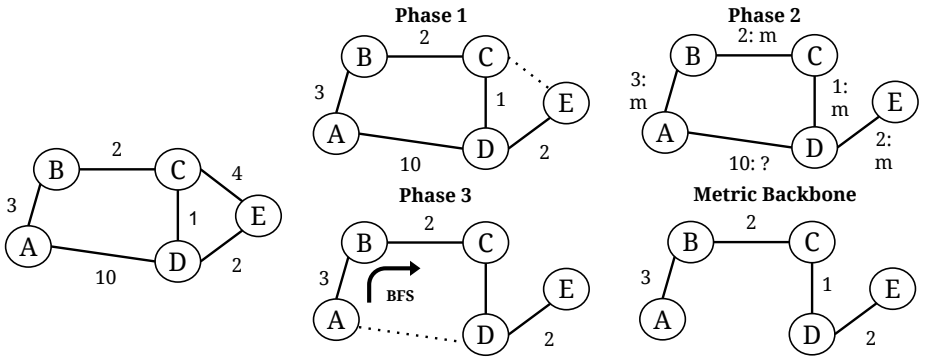

Figure 3: The three phases of the backbone calculation. In the first phase, the algorithms removes 1st-order semi-metric edges, in this case edge CE, marked with a dotted line. In the second phase, the algorithm identifies metric edges, within the two-hop neighborhood of each node. Here, edges AB, BC, CD and DE are identified as metric, while edge AD remains unlabeled. In the third phase, the algorithm discovers all remaining higher-order semi-metric edges, by running a BFS for each unlabeled edge (in this case AD).

of the backbone is an important topic and we we plan to address it in future work. In the Appendix, we provide a brief description of an incremental algorithm and explain how changes in the original graph affect the backbone.

In the rest of this paper, we consider undirected graphs with symmetric relations. We focus on undirected graphs as they appear naturally in the scenarios we examined. For instance, commonly used similarity metrics, like Adamic-Adar, are symmetric. Note, however, that the concept of the metric backbone applies to directed graphs with asymmetric relations too. For a detailed description of the conditions under which an edge in a directed graph is semi-metric, we refer the reader to [49].

3.1 Naive algorithm

The most straight-forward approach is to identify semi-metric edges through multiple breadth-first searches: for each edge (u, v) that we want to test for semi-metricity, we start a breadth first search from node u and we accumulate path weights, while visiting new nodes. During the BFS, if vertex v is visited, we check whether the weight of the newly discovered path is lower than $d(u, v)$. If yes, then (u, v) is semi-metric. Otherwise, we stop exploring towards this direction. If the BFS finishes without encountering vertex v, then there is no alternative path from u to v, and thus, (u, v) is metric. This process is essentially equivalent to solving the APSP problem.

This approach incurs high overhead and does not scale to large graphs. Even if we start several BFSs in parallel, a lot of communication and substantial storage is required to keep track of the visited paths and their weights.

Next, we present a three-phase algorithm, that uses optimizations and empirical heuristics to considerably speed up the computation of the metric backbone. We show that, by using simple scalable steps, we can identify the majority of the semi-metric edges and significantly prune the paths that ultimately need to be explored by a BFS.

3.2 Core algorithm

We divide the algorithm in three phases. In the first phase, we discover and remove all the 1st-order semi-metric edges. In the second phase, we take the induced subgraph and identify metric edges within the two-hop neighborhood of each node. Finally, we discover remaining semi-metric edges with breadth-first searches and remove them to produce the metric backbone. Figure 3 illustrates the algorithm phases using the network example of Section 2.

We divide the algorithm in these three phases for different reasons. First, as we will show, we can easily parallelize and scale the detection of 1st-order semi-metric edges, by detecting triangles. Specifically, in Section 4.1.1, we show how to implement this phase on top of a distributed graph processing system. Second, as we already saw in the analysis results of Section 2.2, the largest fraction of semi-metric edges are typically *1st-order* semi-metric. This allows us to significantly reduce the size of the graph early in the process and provide a fair and practical approximation of the metric backbone, after having executed just the first phase. Third, the second phase can exploit the knowledge that there are no 1st-order semi-metric edges to efficiently discover metric edges. Next, we describe the three phases in detail.

In the first phase, for every triangle in the graph, we test whether one of its edges violates the triangle inequality. Such edges are by definition semi-metric, and we can remove them from the graph. Triangle enumeration is a well-studied problem in graph theory and several algorithms have been proposed for its solution [29, 47, 18]. Here, we use a variation of the node-iterator algorithm to demonstrate the removal of first-order semi-metric edges. Algorithm 1 shows the pseudocode for this phase.

Algorithm 1 Detect semi-metric edges in triangles.

```
1: Input: the set of vertices, V and the set of edges, E
2: for all v ∈ V                          ▷ Iterate over all vertices
3:     for all x, y ∈ neighbors(v)
4:         if x, y ∈ E                    ▷ Check if there exists a triangle
5:             if d(x, y) + d(y, v) < d(x, v)
6:                 remove (x, v)
7:                 remove (v, x)
```

In the second phase, we reverse the logic of the algorithm and aim to identify metric edges. Each node exploits information in its two-hop neighborhood to reason about the semi-metricity of its edges. The initialization of this phase is based on the following proposition:

Proposition 1. *The lowest weight edge of every vertex in a graph, belongs to the metric backbone.*

Let v be a vertex in G and $(v, u_1), (v, u_2), ..., (v, u_k)$ be $v's$ edges, in increasing weight order. If the edge with the lowest weight, (v, u_1), is semi-metric, then there exists a path $p = (v, u_x, ..., u_1)$, such that $d(p) < d(v, u_1)$. This cannot be true, since $d(v, u_1) \leq d(v, u_x), \forall x \neq 1$.

For example, consider node C of the network in Figure 1. We can easily see that its lowest-weight edge, CD, belongs to the metric backbone: any indirect path between C and D would contain either edge CB or edge CE and thus, have a larger weight than the direct edge.

After each node has marked its lowest-weight edges as metric, it checks whether it can reason about the semi-metricity of the rest of its edges, by comparing their weights to the minimum weights of its two-hop paths, which contain metric edges. This process is shown in Figure 4. Node u decides whether edge e_1 is metric, by checking the weights of the two-hop paths along its metric edges, m_1 and m_2. Any alternative path would include edges e_2 or e_3, which already have a larger weight than e_1. If u discovers that the minimum two-hop paths containing its metric edges have longer distance than the direct edge e_1, then e_1 is metric.

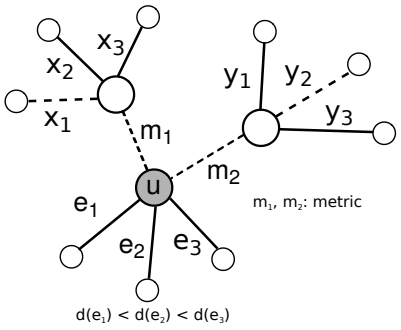

Figure 4: The second phase of the algorithm. Edge indices are ordered by increasing weight value. u has already discovered that m_1 and m_2 are metric. The minimum paths containing its metric edges are shown with dashed lines. If edge e_1 has a lower weight than both these paths, then e_1 is metric.

Proposition 2. *Given a node with metric edges $m_1, m_2, ..., m_k$ and unlabeled edges $e_1, e_2, ..., e_l$, in increasing weight order, edge e_1 is metric if its weight is lower than all the weights of the node's two-hop paths, which contain edges $m_1, m_2, ..., m_k$.*

The proof for Proposition 2 is given in the Appendix.

Algorithm 2 shows the pseudocode for the second phase. We assume a partial ordering on the edge weights and that a node's access to its edges respects this order. We represent the edges of a node v as an ordered set, U_v, with two additional methods, $first$ and $remove$. If the set is not empty, a call to $first$ will return the edge in the set with the minimum weight. If the set contains more than one edge with the minimum weight, $first$ will return all of them. A call to $remove$ will return the same edge(s) as a call to $first$, while also removing them from the set.

When no further local metric edges can be found, we proceed to the third phase, where we characterize the remaining unlabeled edges, by performing breadth first search. For each unlabeled edge (u, v), we start a BFS from node u. If the BFS discovers an indirect path from u to v with a lower weight than the weight of the direct edge, then (u, v) is semi-metric. Otherwise, if the BFS finishes without finding a shorter indirect path, then (u, v) is metric. We present the pseudocode for the third phase in Algorithm 3. The method $bfs(u, v)$ returns the set of all indirect paths, starting from node u and ending in node v.

Algorithm 2 Identify local metric edges.

```
1: Input: the set of vertices, V and the set of edges, E
2: M ← ∅                                   ▷ Metric edges found so far
3: for all v ∈ V                           ▷ Iterate over all vertices
4:     U_v ← E_v                           ▷ All edges are initially unlabeled
5:     W ← ∅                               ▷ Set of weights for comparison
6:     metric ← TRUE
7:     M ← M ∪ (U_v.remove)                ▷ See Proposition 1
8:     while U_v ≠ ∅
9:         e ← U_v.remove
10:        for all m ∈ M
11:            x ← m.target                ▷ The target node of m
12:            w_x = d(v, x) + d(U_x.first)  ▷ The min 2-hop
13:                                        ▷ path weight, that includes m
14:            W ← W ∪ w_x
15:        for all w ∈ W
16:            if d(e) > w
17:                metric ← FALSE
18:                break
19:        if metric                       ▷ All 2-step paths were larger
20:            M ← M ∪ e                   ▷ e is metric
21:            W ← ∅
22:        else
23:            return M                    ▷ Cannot label further edges
```

Based on the results of Figure 2, we expect the majority of metric edges to be identified during the first phase. Moreover, since most of the semi-metric edges are discovered during the first phase, we also expect the BFSs to finish early. Indeed, in the same figure, we observe that this is true for all the networks we analyze. In the worst case, a total of 6 hops is required to label all the edges of the graph.

Algorithm 3 Characterize remaining unlabeled edges

```
1: Input: the set of unlabeled edges, U
2: for all (u, v) ∈ U
3:     label ← METRIC
4:     P_u ← bfs(u, v)                     ▷ Indirect paths from u to v
5:     for all p ∈ P
6:         if d(p) < d(u, v)               ▷ Shorter indirect path found
7:             label ← SEMIMETRIC
8:             break
9:     if label = SEMIMETRIC
10:        remove(u, v)
11:    else
12:        label ← METRIC
```

3.3 Complexity analysis discussion

While our algorithm does not lower the worst-case complexity of the naive algorithm, our heuristic makes the computation practical for large-scale graphs. Here, we analyze the conditions under which it is faster than solving APSP.

Let U be the set of vertices which are sources of unlabeled edges, after removing semi-metric triangles. U is the upper bound of the number of BFSs we have to run, after semi-metric triangle removal. The worst-case complexity of the metric-backbone algorithm is *complexity of semi-metric triangles + complexity of computing shortest paths (or BFSs) on U*. The worst-case complexity of basic triangle listing algorithms is $\Theta(E*d_{max})$, where d_{max} is the maximum vertex degree [33]. This leads to $\Theta(E*V)$ in the worst case. Thus, computing the metric backbone has a worst-case complexity of $\Theta(E*V)+O(U^3)$. If the graph is highly metric, $U \approx V$, and the worst case is equivalent to running APSP, i.e. $O(V^3)$. If the graph is highly semi-metric (and most of the semi-metric edges are discovered in the first step), then $U \ll V$ and the practical run time is much lower.

4. IMPLEMENTATION

To be usable in real scenarios, the computation of the metric backbone must be practical for large graphs. We have implemented the computation of the backbone on top of the Pregel programming model [37], and specifically the Apache Giraph system [1]. Pregel-like platforms [1, 37, 36, 27] are widely adopted and are common components of data centers We have made the implementation of the algorithm available as open source [3].

4.1 Vertex-centric implementation

Here, we describe our vertex-centric implementation in Giraph. It consists of three phases: (i) detection of the 1^{st}-order semi-metric edges (ii) iterative labeling of local metric edges and (iii) labeling of all remaining metric edges.

4.1.1 Phase 1: Detect semi-metric triangles

This phase is based on the BSP-model algorithm for triangle detection, as described in [22] for a weighted, undirected graph with total ordering on the vertex IDs. The algorithm consists of four supersteps and discovers each triangle exactly once. In the first superstep, each vertex propagates its ID to neighbors with higher IDs. For example, if vertex 5 is a neighbor of vertices 1 and 6, it only propagates its ID to vertex 6. In the second superstep, each vertex iterates over received messages and augments each one with (1) its own ID and (2) the edge weight connecting this vertex with the message sender. It then propagates the augmented messages to all neighbors with higher IDs. In the third superstep, each vertex checks whether each of the received messages forms a triangle. If a triangle is found, the vertex compares the edge distances to discover whether there exists a semi-metric edge. If a semi-metric edge is found, it is marked for removal. In the final superstep, all marked edges are removed.

4.1.2 Phase 2: Identify local metric edges

This phase consists of three supersteps. The first superstep is executed once, while steps two and three are executed in an alternate fashion, until no further metric edges can be discovered.

1. Mark the lowest-weight local edges as metric: Each vertex marks its lowest-weight edges as metric (according to Proposition 1). Then, it sends a message with its ID and the edge weight, along the identified metric edges.

[3]Implementation available at http://grafos.ml

Graph	% of unlabeled edges
Tuenti [4]	1.17
LiveJournal [8]	4.36
NotreDame-web [7]	9.09
DBLP [52]	8.08
Twitter egonet [38]	1.15

Table 3: Percentage of unlabeled edges, after the second phase of the mertric backbone algorithm.

2. Send lowest alternative path distance to metric edges: Vertices which have received a message are endpoints of metric edges found in the previous superstep. Thus, these vertices mark opposite-direction edges as metric. Then, every vertex sends one message along all its metric edges. For metric edge (u, v), the message contains the distance of the shortest two-hop path, that passes through u and contains (u, v). This distance is computed by adding the weight of (u, v) and the smallest weights of the rest of u's edges.

3. Check lowest-weight unlabeled edge In the third superstep, each vertex checks whether it can reason about the semi-metricity of its smallest-weight unlabeled edge. If all of the weights in the received messages are larger than the weight of this edge, then both this edge and the opposite-direction edge can be safely marked as metric.

4.1.3 Phase 3: Label remaining metric edges

In order to characterize the remaining unlabeled edges, we initiate parallel breadth-first searches. For every unlabeled edge (u, v), u propagates a message to its neighbors to explore paths that have weight lower than the weight of (u, v). After one initialization superstep the computation iteratively runs custom breadth-first searches, until no unlabeled edges remain. During the initialization superstep, each vertex gathers its unlabeled edges. For each unlabeled edge, it creates a message and propagates it along all edges that have weight lower than the unlabeled edge weight. In the next supersteps, each vertex performs the following computation until convergence. Upon receiving a message, it checks whether it is the target of the message edge. If it is, then this edge is labeled as semi-metric. Otherwise, it propagates the message to neighbors that could produce shorter paths, making sure not to forward the message back to its source. Note that, the percentage of unlabeled edges for which we need to execute a BFS is usually very small. For the networks we analyze, the percentage of unlabeled edges, after executing the second step of the algorithm, is under 10% in all cases, and as low as 1% for the Twitter and Tuenti graphs. We present these results in Table 3. Also, since most of the semi-metric edges have already been discovered, the majority of the BFSs terminate after only a few steps.

4.2 Spreading the communication overhead

The first phase of the metric backbone algorithm has computational complexity equivalent to the one of triangle enumeration in undirected graphs. Even though there exist several heuristics that significantly reduce the practical computation time [10, 43, 22], the memory requirements in a message-passing system like Giraph, might still be fairly high. In order to avoid memory problems, we apply a simple, yet practically effective optimization in this phase.

We make the observation that the computations for detecting semi-metric edges can be performed completely independently. None of the parts of the algorithm require any message aggregation or combining. Thus, in order to reduce the communication load, we spread the algorithm execution into several identical superstep-groups, which we call *megasteps*. Each megastep contains the three supersteps of the first phase of the algorithm, as described in 4.1.1. Throughout the program execution, we keep all vertices active. However, during each megastep, only some of the vertices execute the computation, while the rest remain idle. Since all computations are independent from each other, there is no need to maintain or transfer any state across supersteps. When all vertices have executed their computation, we have encountered all first-order semi-metric edges. Note that for this optimization to work, we do not remove any edges before all vertices have completed their computation phase. Instead, when we encounter a semi-metric edge, we simply put a mark on it. We then remove semi-metric edges during a single finalization superstep.

In our implementation, we decide which vertices to activate in which megastep, based on their numeric vertex IDs. More sophisticated load balancing methods might yield better performance. In our evaluation, we varied the number of megasteps for each the of the experiments. Intuitively, the number of megasteps should increase with the graph size, but it also depends on the amount of available memory. For example, we found that, for our experimental setup, 10 megasteps result in a fairly fast execution for finding first-order semi-metric edges in the Livejournal dataset, while we used 100 megasteps for the Tuenti network.

5. EVALUATION

We evaluate our approach in different respects. First, we measure the performance or our algorithm and show that it is orders of magnitude faster than APSP. Second, we show that even when computing the exact backbone incurs high overhead, we can compute a backbone approximation by removing only 1^{st}-order semi-metric edges, to scale to large graphs. Third, we measure the impact on the performance on different graph management systems. We evaluate this using a variety of real-world data sets and graph queries.

5.1 Comparing to APSP

Here, we compare the overhead of our algorithm with that of APSP. Computing APSP on a distributed platform like Giraph is a challenging task. To perform this computation, every vertex must compute and store $|V|$ distances, resulting in excessive communication and memory overhead. Instead, we implement APSP in two possible ways. The first approach computes Single-Source Shortest Paths (SSSP) for each vertex individually, in successive jobs. The second approach runs multiple instances of Multi-Source Shortest Paths (MSSP) [4]. MSSP batches a configurable number of simultaneous SSSPs in the same physical job, improving efficiency.

First, we run 100 instances of SSSP from different sources and average the execution time. Second, we run MSSP using 1% of the vertices as sources. Because of the time it takes to run the entire APSP computation, we estimate the total time by projecting the measured time to the entire graph. We compare the projected execution times with the total execution time of our algorithm for the Twitter-egonet and DBLP graphs. Running APSP for larger graphs was impossible with our available computing resources.

[4]Our implementation of MSSP is available as open source.

Graph	SSSP	MSSP
DBLP [52]	120	11
Twitter-ego [38]	177	14

Table 4: Ratio of the projected execution time of (1) SSSP from all vertices and (2) MSSP for all vertices of the input graphs over the total execution time of our algorithm for computing the metric backbone.

We show the results in Table 4. The projected execution times for SSSP are in the order of months, while the projected execution times for MSSP are in the order of days. Instead, our algorithm was able to compute the metric backbone in 8 hours for the DBLP graph and 2 hours for the Twitter graph.

5.2 Scalability

Even though computing the exact backbone can incur high overhead, we can still compute a backbone approximation in a scalable manner by removing only 1^{st}-order semi-metric edges. As we show in Section 2, the first phase of our algorithm removes the majority of semi-metric edges, practically providing a good approximation for many applications. Here, we show that this phase affords a scalable implementation on top of distributed graph processing frameworks, such as Giraph. We measure execution time as the size of the graph increases. In the experiments following, we use this backbone approximation to measure speedup of graph analysis applications.

We apply our algorithm on synthetic graphs constructed with the Watts-Strogatz model [50], using Giraph's built-in graph generator. The generator produces unweighted graphs with high clustering coefficient and low average path length. Using synthetic graphs for these experiments allows us to gradually increase the number of vertices and edges in a controlled manner, still working with a graph that resembles a real-world social network or web-graph characterized by small-world properties. Even though the synthetic graphs do not have edge weights, the execution time of this phase depends only on the size of the graph and the number of triangles. Edge weights impact only the the number of edges removed at the end (see Section 4.1.1). Edge removals have a constant overhead that does not affect scalability. We configure the generator so that vertices have 30 edges on average and set the model rewiring probability to 0.3. We performed this experiment on an AWS cluster consisting of 32 r3.4xlarge instances (16 vCPUs, 122GB memory).

We show the result in Figure 5. As the graph size increases from 240 million to 3.8 billion edges, the running time increases almost linearly. On the largest graph, which has a size of 123GB and 4.6 billion triangles, the computation finishes in 14 minutes. This is the largest graph we could process on the 32 compute nodes due to the message overhead. Further, in Section 5.4, we provide results on a $\sim$50 billion-edge subgraph of the Facebook social network, demonstrating that our algorithm is practical for even larger graphs. In Table 5, we also provide the execution time of semi-metric triangle detection for the real-world graph

Graph	|E|	Size (GB)	Time (s)
Twitter [32]	1.5B	72	3792
Tuenti [4]	685M	33	1305
LiveJournal [8]	34M	1.6	62
NotreDame-web [7]	1.5M	0.07	25
Twitter egonet [38]	1.7M	0.08	32
DBLP [52]	1M	0.05	20

Table 5: Execution runtime of semi-metric triangle detection and removal for real-world graphs.

Graph	% of size reduction
Tuenti [4]	59.14
LiveJournal [8]	39.66
NotreDame-web [7]	16.67
DBLP [52]	22.62
Twitter egonet [38]	57.14

Table 6: The Neo4j relationship store size reduction when 1st-order semi-metric edges have been removed from the graph.

datasets we consider in this paper. The table shows the number of edges and size in GB of each graph. This experiment was performed on an Amazon EC2 cluster of 16 r3.2xlarge instances (8 vCPUs, 61GB memory).

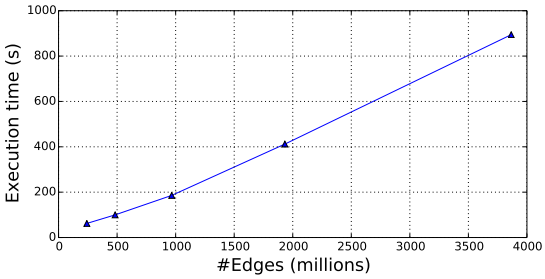

Figure 5: Scalability evaluation of the backbone algorithm. The figure shows execution time of the first phase for synthetic graphs of increasing size.

5.3 Graph databases

Here, we want to measure the impact on query latency and storage reduction by using the backbone transparently in a graph database. We load the original graph in the Neo4j database and then run two different queries: (i) we run a shortest path query for 1000 randomly selected pairs of nodes in the graph, (ii) we run a connected components query 10 times. For both queries we measure the average latency. We perform this measurement for different graphs. Subsequently, we start another instance of the database, where we load the graph after removing 1^{st}-order semi-metric edges, and repeat the same experiment. We run this experiment on an Intel Xeon E5530 2.40GHz server with 128GB of RAM, running Ubuntu 2.6.38.

Table 6 shows the size reduction of the Neo4j database when we use the approximate backbone for query evaluation. We see that for highly semi-metric graphs, the database files have close to 60% less storage requirements.

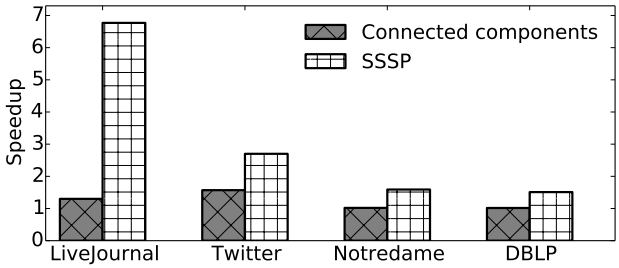

Figure 6: Query speedup on the Neo4j graph database. Graphs in the x-axis are ordered from larger to smaller.

Graph	Spearman Coefficient
Facebook	0.98
Tuenti [4]	0.98
LiveJournal [8]	0.95
NotreDame-web [7]	0.76
DBLP [52]	0.98
Twitter egonet [38]	0.97

Table 7: The Spearman correlation coefficient between (a) the ranking computed by PageRank on the original weighted graph and (b) the ranking computed by PageRank on the same graph, after removing first-order semimetric edges.

Figure 6 shows the speedup when we execute the queries on the approximate metric backbone compared to the original graph for all the workloads. First, we observe the highest speedups occur for the shortest path query. Second, the larger the graph, the higher the speedup. For the three smaller graphs, the speedup ranges from 1.51 to 2.7, while for the LiveJournal graph the speedup is 6.7.

Notice that the connected components query does not consider the edge weights, connectivity. For smaller graphs the speedup is 1.01. However, for larger graphs we still measured significant performance improvement, with the speedup ranging from 1.30 to 1.5.

In general, even in a highly optimized system such as a commercial graph database, we still derive a significant speedup by applying the backbone approach transparently. We believe that integrating the approach inside the system can yield even higher speedups.

5.4 Distributed graph analytics

Reducing the edges of a graph impacts directly the performance of algorithms developed on top of distributed graph processing systems, such as Pregel and Graphlab. In such systems, programs are typically communication intensive and communication coincides with the edges of the graph. Further, reducing the size of a distributed graph also reduces the runtime memory requirements of such systems, which may indirectly affect performance as well.

In this section, we measure how removing semi-metric edges from a graph improves the performance of different applications, developed on the Apache Giraph graph processing system. We run three applications on the original graph and on the graph with no 1^{st}-order semi-metric edges; Connected Components (CC), Single-Source-Shortest-Paths

(SSSP), and weighted PageRank (15 iterations). For each application, we measure (a) the runtime speedup, and (b) the total amount of messages sent. For PageRank, we also compute the Spearman correlation coefficient of the resulting rankings, shown in Table 7. We perform this experiment on an Amazon EC2 cluster with 16 r3.2xlarge instances[5].

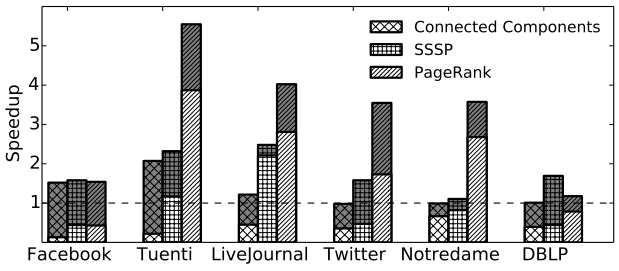

Figure 7: Runtime speedup on Apache Giraph. Graphs in the x-axis are ordered from larger to smaller. Bars are overlayed, not stacked.

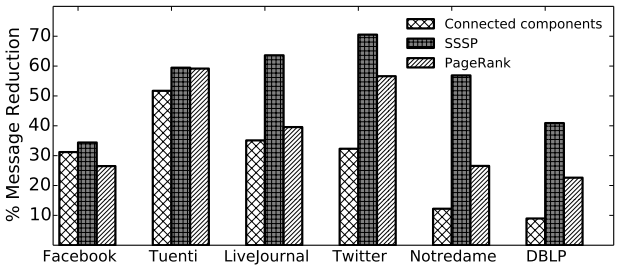

Figure 8: Communication reduction on Apache Giraph. Graphs in the x-axis are ordered from larger to smaller.

In Figure 7, we plot the runtime speedup for different applications. Bars are overlayed, not stacked. The white bars show the speedup $S_1 = T_o/(T_b + T)$, where T_o is the time to run the analysis on the original graph, T_b is the time to calculate the backbone approximation, that is, remove the 1^{st}-order semi-metric edges, and T is the time to run the analysis on the backbone approximation. The gray bars show the speedup $S_2 = T_o/T$ that considers only the time to run the analysis on the two graphs. This is the speedup after the overhead of our algorithms gets amortized.

For more compute-intensive applications, such as PageRank, the total runtime, including the removal of semi-metric edges, is typically lower than running the application on the original graph. For SSSP, this is also true for the Tuenti and Livejournal datasets. In some cases, though, if we consider only one run of the analysis, the overhead of our approach exceeds the analysis runtime on the original graph. This is true for the Facebook and DBLP datasets, and all instances of the Connected Components experiment.

Note, however, that computing the backbone is intended to be run once and re-used multiple times across applications, so S_2 represents the practical speedup we see in the applications. For example, approximating betweenness centrality in a graph requires running *multiple* SSSP instances with different source vertices. For such an application, the overhead of removing semi-metric edges gets amortized after only the first two runs of the analysis in the worst case.

With regard to graph semi-metricity, as expected, we notice larger gains for highly semi-metric graphs. Tuenti, Livejournal, Twitter, and Notredame (40-60% semi-metric) give better speedups than Facebook and DBLP (23-27% semi-metric). With regard to application complexity, we observe a higher benefit for more compute-intensive applications. We see the highest speedup for PageRank, which runs close to six times faster for the Tuenti graph. We believe that this is because, in contrast to the SSSP and CC applications, in PageRank, all the nodes communicate with all their neighbors, in every iteration (in SSSP and CC nodes do not send outgoing messages to their neighbors, if their value does not change). The shortest paths application also benefits significantly, running about two times faster on average. Not surprisingly, connected components experiences the lowest speedup, since it does not make use of the edge weights. Even so, for a highly semi-metric graph, such as Tuenti, the benefit is substantial.

Figure 8 shows the communication reduction, when running the same three applications, for the different networks. For SSSP and PageRank, we observe tremendous communication reduction, ranging from 30% to 70%, in terms of messages exchanged, throughout the application execution. Even for connected components, the reduction is remarkable, ranging from 10% to 50%.

6. RELATED WORK

Complex network analysis. The concept of semi-metricity in weighted graphs has been first used in complex network analysis. Semi-metricity was introduced in weighted graphs by Rocha [45, 46], showing that semi-metric edges in a weighted graph encode some latent information between a pair of nodes, which may be useful for information discovery [45, 46, 48, 44, 49]. Simas et al. introduce in [49] a new mathematical framework to the study of networks in general and specifically semi-metric networks. In [49] Simas et al. introduce the concept of distance backbones, a generalization of the metric backbone. In this work they present a few examples of how distance backbones, including the metric backbone, can be useful in network analysis, such as improving modularity in community detection.

Graph compression and query optimization. In [39, 6], the authors examine the problem of producing minimal graph representations, while preserving the graph's reachability properties. Their goal is to reduce the memory used for storing the graph and potentially improve the efficiency of certain algorithms that contain reachability queries. More recent works on graph compression, [12], [16], look into specific techniques for compressing web graphs and social networks. These provide methods that preserve the information of the original graph. In these cases, graph decompression is necessary for query evaluation. On the other hand, [23], proposes a query-preserving graph compression method, relative to reachability and graph pattern queries. Similar to our work, the query can be directly issued on the compressed

[5]The analysis on the Facebook graph was run on an experimental cluster with 50 machines, each with 16 cores and 10Gbs Ethernet.

graph representation. The authors also provide a method for incrementally maintaining the compressed graph structure, for dynamic graphs. The proposed algorithms summarize nodes, based on equivalence relations for each query class. Compression algorithms are also studied in [40, 35]. These works explore ways to minimize graph representation overhead and offer several algorithms, both sequential and distributed, for lossless and lossy graph compression. However, their main goal is reducing the graph sizes, while providing guarantees on decompression accuracy.

Graph compression is also closely related to graph query optimization. [53] provides a graph indexing technique that speeds up search in large graphs. It leverages shortest-paths information, but does not generate a reduced graph representation. The indexes encode neighborhood shortest-paths information, which is used to prune the graph search space. The technique is mostly targeted to graph isomorphism queries.

If we view the metric backbone as a reduced graph representation, there are two fundamental differences between our work and aforementioned works. First, our method does not collapse graph vertices; the metric backbone preserves all the vertices of the original graph. Second, none of these works take into account the weights when compressing the graph. Regarding the motivation of our work, an important difference is that we are mainly concerned with the challenge of scalability and we provide a distributed implementation of the metric backbone algorithm. Moreover, existing works focus on optimizing reachability and graph matching queries. In contrast, we go a step further and evaluate the metric backbone in the context of graph analytics and iterative graph processing.

Spanners and Sparsifiers. Spanners [42, 21] and sparsifiers [26] are approximate graph structures that are used to approximate graph distances and shortest paths. They have been mostly used in streaming algorithms as sketches or graph summaries. In this context, the metric backbone can be considered equivalent to the minimum 1-Spanner. A spanner guarantees that distances can be approximated with a certain maximum multiplicative error. On the contrary, the metric backbone preserves all shortest paths and gives the *exact* shortest distances. Thus, it can be used in applications that cannot tolerate errors on the graph distances.

7. CONCLUSION

Several real-world weighted graphs exhibit high degree of semi-metricity; direct edges between nodes are not always the shortest path. The metric backbone captures this property, allowing us to compute several graph metrics more efficiently. When used in the context of graph management systems, even modestly semi-metric graphs reduce the size of the original graph enough to allow for runtime speedups of up to 6 times.

Traditionally, the computation of the metric backbone required the solution of the APSP problem. Here, we have proposed an algorithm that can compute the metric backbone, avoiding the computation of APSP. Further, we have showed that we can approximate the metric backbone by just removing 1st-order semi-metric edges. This allows us to scale the computation of the metric backbone, making its use practical in large scale scenarios.

Finally, while in this paper we have used the metric backbone to improve the performance of graph algorithms transparently, one can use the concept to re-design certain algorithms explicitly on top of the backbone. For instance, we can compute shortest paths more efficiently by intelligently choosing which nodes to traverse. We believe that the metric backbone and distance backbones, in general, offer a framework for the design of such algorithms and we plan to investigate this in the future.

8. REFERENCES

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

[2] The movielens dataset. http://grouplens.org/datasets/movielens.

[3] Neo4j Graph Database. http://neo4j.com.

[4] The Tuenti Social Network. http://www.tuenti.com.

[5] L. A. Adamic et al. Friends and neighbors on the Web. *Social Networks*, 25(3):211–230, July 2003.

[6] A. V. Aho, et al. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[7] R. Albert, et al. Internet: Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, Sept. 1999.

[8] L. Backstrom, et al. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD'06*, Aug. 2006.

[9] A. Barrat, et al. The architecture of complex weighted networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(11):3747–52, Mar. 2004.

[10] J. W. Berry, et al. Why do simple algorithms for triangle enumeration work in the real world? In *Conference on Innovations in Theoretical Computer Science*, pages 225–234. ACM, 2014.

[11] J. Blackburn, et al. The power of indirect ties in friend-to-friend storage systems. *IEEE International Conference on Peer-to-Peer Computing*, Sept. 2014.

[12] P. Boldi, et al. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *ACM WWW'11*, pages 587–596, 2011.

[13] P. Boldi et al. In-Core Computation of Geometric Centralities with HyperBall: A Hundred Billion Nodes and Beyond. In *IEEE International Conference on Data Mining Workshops*, pages 621–628. IEEE, Dec. 2013.

[14] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 2001.

[15] S. Chen, et al. On the similarity metric and the distance metric. *Theoretical Computer Science*, 410(24-25):2365–2376, May 2009.

[16] F. Chierichetti, et al. On compressing social networks. In *ACM SIGKDD'09*, pages 219–228. ACM, 2009.

[17] N. A. Christakis et al. *Connected: The Surprising Power of Our Social Networks and how They Shape Our Lives*. 2009.

[18] S. Chu et al. Triangle listing in massive networks and its applications. In *ACM SIGKDD'11*, pages 672–680, 2011.

[19] V. Colizza, et al. Reaction-diffusion processes and metapopulation models in heterogeneous networks. *Nature Physics*, 2007.

[20] T. M. L. M. De Simas. *Stochastic Models and Transitivity in Complex Networks*. PhD thesis, Indiana University, 2012.

[21] F. F. Dragan, et al. Spanners in sparse graphs. *Journal of Computer and System Sciences*, 77(6):1108 – 1119, 2011.

[22] D. Ediger et al. Investigating graph algorithms in the bsp model on the cray xmt. In *IPDPS Workshops*, pages 1638–1645. IEEE, 2013.

[23] W. Fan, et al. Query preserving graph compression. In *ACM SIGMOD'12*, pages 157–168, 2012.

[24] S. Fortunato et al. Random walks on directed networks: the case of pagerank. *International Journal of Bifurcation and Chaos*, 17(07):2343–2353, 2007.

[25] N. Friedkin. Horizons of observability and limits of informal control in organizations. *Social Forces*, 62(1):54 – 77, 1983.
[26] W. S. Fung, et al. A general framework for graph sparsification. In *ACM Symposium on Theory of Computing*, pages 71–80, New York, NY, USA, 2011. ACM.
[27] J. E. Gonzalez, et al. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.
[28] M. Granovetter. The strenght of weak ties. *American Journal of Sociology*, 78(6):1360–1380, May 1973.
[29] A. Itai et al. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.
[30] P. Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37 – 50, 1912.
[31] U. Kang, et al. Centralities in Large Networks: Algorithms and Observations. *SIAM International Conference on Data Mining*.
[32] H. Kwak, et al. What is Twitter, a social network or a news media? In *International Conference on World Wide Web*. ACM Press, Apr. 2010.
[33] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, Nov. 2008.
[34] I. Leung, et al. Towards real-time community detection in large networks. *Physical Review E*, 79(6):066107, June 2009.
[35] X. Liu, et al. Distributed graph summarization. In *ACM CIKM'14*, pages 799–808, 2014.
[36] Y. Low, et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *The VLDB Endowment*, Aug. 2012.
[37] G. Malewicz, et al. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*, 2010.
[38] J. McAuley et al. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems*, 2012.
[39] D. M. Moyles et al. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM (JACM)*, 16(3):455–460, 1969.
[40] S. Navlakha, et al. Graph summarization with bounded error. In *ACM SIGMOD International Conference on Management of Data*, pages 419–432, 2008.
[41] T. Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks*, 35(2):159–167, May 2013.
[42] D. Peleg et al. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.
[43] L. Quick, et al. Using pregel-like large scale graph processing frameworks for social network analysis. In *International Conference on Advances in Social Networks Analysis and Mining*, pages 457–463, Washington, DC, USA, 2012.
[44] L. Rocha, et al. MyLibrary@LANL: Proximity and Semi-metric Networks for a Collaborative and Recommender Web Service. In *ACM International Conference on Web Intelligence*, 2005.
[45] L. M. Rocha. Proximity and semi-metric analysis of social networks. In *Internal Report of Advanced Knowledge Integration In Assessing Terrorist Threats LDRD-DR Network Analysis Component. LAUR*, pages 02–6557, 2002.
[46] L. M. Rocha. Semi-metric behavior in document networks and its application to recommendation systems. *Soft Computing Agents: A New Perspective for Dynamic Information Systems*, 83:137, 2002.
[47] T. Schank et al. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.
[48] T. Simas et al. Semi-metric Networks for Recommender Systems. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2012.
[49] T. Simas et al. Distance closures on complex networks. *Network Science*, 3:227–268, 6 2015.
[50] D. Watts et al. Collective dynamics of 'small-world' networks. *Nature*, 1998.
[51] C. Wilson, et al. User interactions in social networks and their implications. In *ACM European Conference on Computer Systems*, page 205, Apr. 2009.
[52] J. Yang et al. Defining and Evaluating Network Communities based on Ground-truth. In *IEEE International Conference on Data Mining*, May 2012.
[53] P. Zhao et al. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, Sept. 2010.
[54] X. Zuo, et al. The Influence of Indirect Ties on Social Network Dynamics. In *International Conference on Social Informatics*, Nov. 2014.

APPENDIX

Proofs. We prove Proposition 2, using the notation of Section 2.1.

Proof. Let edge e_1 of Figure 4 be the edge in consideration and let v be the label of its target node. According to the proposition,

$$d(e_1) < d(m_1) + d(x_1) \tag{1}$$

and

$$d(e_1) < d(m_2) + d(y_1) \tag{2}$$

We will assume that e_1 is semi-metric and prove that this cannot be true. If e_1 is semi-metric, then there exists a path p, from v to u, which does not contain e_1, has length at least two and its weight is lower than the weight of e_1, i.e.

$$d(p) < d(e_1). \tag{3}$$

Obviously, this path cannot contain e_2 and e_3, since $d(e_1) < d(e_2) < d(e_3)$. Thus, p passes through m_1 or m_2. If p passes through m_1, then its lowest weight possible would be $d(m_1) + d(x_1)$. According to 1, e_1 has a weight smaller than the lowest possible weight of a path passing through m_1, thus, equation 3 cannot be true. We arrive at the same contradiction assuming that p passes through m_2. Therefore, e_1 is metric. □

Incremental maintenance of the metric backbone. We only consider single edge removals and edge additions. Extending these to edge weight changes and node additions or removals, is straight-forward. Note that the edges that change from semi-metric to metric (or vice-versa), in one of the above cases, do not cause further changes in the metric backbone, since all weights remain unchanged.

Edge Removal: If the edge to be removed is semi-metric, the metric backbone does not change. The edge can be simply removed from the original graph. If the edge to be removed is metric, some of the semi-metric edges might now become metric. Note that the metric edges do not get affected. Let (u, v) be the metric edge to be removed. A semi-metric edge (x, y) is potentially affected by the removal of (u, v) if there is a path $p = (x, ..., u, v, ...y)$ in G, such that $d(p) < d(x, y)$. Thus, the only edges that might be affected are the semi-metric edges in indirect paths from u to v. We check these edges with the following steps. First, remove edge (u, v) from G. Then, for every semi-metric edge in all remaining paths from u to v, execute the backbone algorithm, starting from phase 2.

Edge Addition: When adding an edge (u, v) to the original graph, we first check whether this edge is semi-metric. In order to do so, we can easily find the current shortest path from u to v, using the metric backbone. If the weight of the new edge is larger than the current shortest path, then this edge is semi-metric and the metric backbone does not change. If the edge weight of the new edge is lower than the current shortest path from u to v, then the edge is metric. In this case, some of the metric edges of the backbone might become semi-metric, if a shorter path is introduced through the new edge. Again, the only edges that we need to consider are the ones on the indirect paths from node u to node v.