Enabling Graph Mining in RDF Triplestores using SPARQL for Holistic In-situ Graph Analysis

Sangkeun Leea, Sreenivas R. Sukumarb, Seokyong Hongb, Seung-Hwan Lima

aComputational Sciences and Engineering Division
Oak Ridge National Laboratory, TN, USA
bDepartment of Computer Science
North Carolina State University

Abstract

Graph analysis is now considered as a promising technique to discover useful knowledge in data with a new perspective. We envision that there are two dimensions of graph analysis: OnLine Graph Analytic Processing (OLGAP) and Graph Mining (GM) where each respectively focuses on subgraph pattern matching and automatic knowledge discovery in graph. As these two dimensions aim to complementarily solve complex problems, holistic in-situ graph analysis which covers both OLGAP and GM in a single system is critical for minimizing the burdens of operating multiple graph systems and transferring intermediate result-sets between those systems. Nevertheless, most existing graph analysis systems are only capable of one dimension of graph analysis. In this work, we take an approach to enabling GM capabilities (e.g., PageRank, connected-component analysis, node eccentricity, etc.) in RDF triplestores, which are originally developed to store RDF datasets and provide OLGAP capability. More specifically, to achieve our goal, we implemented six representative graph mining algorithms using SPARQL. The approach allows a wide range of available RDF data sets directly applicable for holistic graph analysis within a system. For validation of our approach, we evaluate performance of our implementations with nine real-world datasets and three different computing environments - a laptop computer, an Amazon EC2 instance, and a shared-memory Cray XMT2 URIKA-GD graph-processing appliance. The experimental results show that our implementation can provide promising and scalable performance for real world graph analysis in all tested environments. The developed software is publicly available in an open-source project that we initiated.

Keywords: graph, mining, analysis, RDF, SPARQL, triplestore, Semantic Web

Acknowledgement

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

The authors acknowledge Tyler C. Brown who contributed significantly to the software development efforts during his internship at the Oak Ridge National Laboratory.

URL: lees4@ornl.gov (Sangkeun Lee), sukumarsr@ornl.gov (Sreenivas R. Sukumar), shong3@ncsu.edu (Seokyong Hong), lims1@ornl.gov (Seung-Hwan Lim)
1. Introduction

The growing interests in discovering relationships across disparate datasets in various domains (e.g., social network, life science, government, healthcare, etc.) has increasingly highlighted graph analysis as a tool for data-driven discovery. Graph analysis is a general term describing the process of unveiling useful knowledge implied in graphs. Major operations in graph analysis fall into two broad categories: (1) online graph query processing such as online data retrieval, aggregation of entities (nodes, edges, paths, and subgraphs) in a graph, and graph pattern matching; and (2) automatic discovery of patterns and prediction of graph properties such as counting triangles (Tsourakakis, 2008), calculating degree distribution, finding eccentricity (Hage and Harary, 1995), finding connected components, computing PageRank/Personalized PageRank (Page et al., 1999; Tong et al., 2006). In this paper, the first category of analysis activities is referred to as Online Graph Analytic Processing (OLGAP) and the second category as Graph Mining (GM).

As with traditional On-Line Analytic Processing (OLAP) (Chaudhuri and Dayal, 1997) that performs real-time and online statistical analysis of historical data using a multi-dimensional cube in traditional data warehouse environments, OLGAP can be considered as its counterpart. It aims to provide real-time and online data analysis over a graph data model, instead of a multi-dimensional cube. OLGAP encloses graph pattern matching as well as the statistical analysis. For example, let us assume that we have a data graph $g$ that was constructed by integrating datasets from two different hospitals A and B. OLGAP answers questions like “Who are the patients having the treatment graph pattern $p$ in the healthcare data graph $g$?”, where $p$ is usually defined by a subject-matter-expert based on his or her domain knowledge. The expert may perform statistical analysis on the retrieved graph instances, or he can refine the pattern and repeat the process.

On the other hand, instead of utilizing user-defined graph queries, to discover patterns or predict unknown graph properties, GM involves a predefined set of procedures such as degree distribution, triangle count (Tsourakakis, 2008), node eccentricity (Hage and Harary, 1995), connected components, PageRank/Personalized PageRank (Page et al., 1999; Tong et al., 2006), etc. One chooses graph mining algorithms based on the mathematical expression of saliency to answer questions like “Who is likely to be the most influential person in the social network?” or “Are there communities in the literature citation network?”. GM is a sub-category of traditional data mining. Figure 1 illustrates the relationship between two major categories of graph analysis.

It is very important to understand that OLGAP and GM complement each other and typically form a cyclic workflow in graph analysis. For instance, a user may want to compute the importance of nodes in a social network using the PageRank algorithm, then they may want to understand structural friendship pattern among the users with high PageRank scores. Thus, supporting both analysis capabilities in a single system, so-called the capability of holistic in-situ graph analysis, is critical for data-driven knowledge discovery. The holistic in-situ graph analysis will allow users not to worry about managing multiple systems, dealing with consistency of datasets staged in different systems, and data transformation cost caused by transferring intermediate datasets. However, among a wide range of technologies for graph analysis, it is not easy to find a system that provides both OLGAP and GM capabilities. For instance, Graph
databases (e.g., Neo4j (Robinson et al., 2013), DEX (Martínez-Bazan et al., 2007), Titan1, etc.) and RDF triplestores (e.g., Jena SDB (Schmidt et al., 2008), Sesame (Broekstra et al., 2002), Virtuoso (Erling and Mikhailov, 2009), etc. ) provide OLGAP capability, but they lack built-in GM capabilities. In contrast to graph databases, graph processing systems such as Pegasus (Deelman et al., 2005), Giraph (Avery, 2011), and GraphX (Xin et al., 2013), focus on scaling up graph mining algorithms for large-scale graphs. However, their online pattern matching capability, which is essential for OLGAP, is very limited, as they do not provide high-level graph query processing capability.

In this paper, to address such limitations, we aim to enable holistic in-situ graph analysis, which is to perform both OLGAP and GM within a single system. Particularly, we take an approach to enabling GM capabilities in RDF triplestores, which are originally developed to only provide OLGAP capability. More specifically, we implement representative graph mining algorithms using the standard query language of RDF triplestores, called SPARQL.

Our choice is based on the following reasons. First, there is a number of readily available RDF datasets disseminated by data scientists or organizations. For instance, the Linked Data Open Community Project in RDF consists of over 31 billion triples. The open community effort contains a wide range of information about places, people, organisms, diseases, genes, medicines, and vast bibliographic data about books, music, television and movies. Second, thanks to the standardization, various RDF triplestores (e.g., Jena (Schmidt et al., 2008), Sesame (Broekstra et al., 2002), RDFSuite(Alexaki et al., 2001), SPARQLVerse (Liu et al., 2014), etc.) can take the advantage of our implementations. Note that not only conventional triplestores but also RDF/SPARQL-based graph processing appliance such as Urika-GD (Sukumar and Bond, 2013) can achieve the capability of performing GM. Third, graph datasets that are not in RDF format can be easily transformed in a straightforward way into RDF datasets, as RDF naturally represents a graph. We believe that enabling GM capability in RDF triplestores will bring a huge impact, as it allows a number of data scientists who have their RDF datasets in various RDF triplestore to perform GM without requiring additional data transformation or placement.

Unfortunately, it is not trivial to implement graph mining algorithms using SPARQL. The complexity arises from the fact that most graph-theoretic algorithms have a linear-algebra formulation and assume adjacency-matrices as the default data model. Matrix and array structures are not straightforward to realize using the SPARQL query algebra. One has to redesign algorithms that can handle the triple representation and algorithms have to be simplified for graph operations supported by the SPARQL-query algebra. Also, most graph mining techniques are iterative algorithms and SPARQL does not support iterative querying.

In our previous workshop paper (Lee et al., 2015b), we introduced our initial concepts and challenges of enabling graph mining using SPARQL in triplestores. We presented three iterative graph mining algorithms and evaluated their performance on a standalone machine. As an extension of our work, we propose more optimized algorithms along with three additional algorithms in triplestores which are critical to graph analysis. The proposed algorithms are extensively evaluated and analyzed. The main contributions of efforts are summarized as follow:

1. To achieve the capability of holistic in-situ graph analysis in RDF triplestores, we present how to implement graph mining algorithms using SPARQL queries. For further development, we suggest guidelines that will help data scientists to design efficient implementation of graph mining algorithms in SPARQL.

2. The work described in this paper has been released as publicly available open-source graph mining algorithm implementations for RDF triplestores (https://github.com/ssrangan/gm-sparql). The implemented graph algorithms include; degree distribution, triangle count, eccentricity, connected component, and PageRank/Personalized PageRank. To the best of our knowledge, there was no publicly available graph mining library for RDF triplestores.

3. We evaluate the performance of our implementations on RDF triplestores over several computing environments including a laptop computer, cloud computing instance (Amazon EC2), and a specialized RDF processing supercomputer (Urika). The experimental results confirm that graph mining algorithms implemented using SPARQL show promising performance and scalability with properly configured triplestores.

The remainder of this paper flows as follows. Section 2 reviews related work and representative graph mining algorithms that we focus on in this paper. In section 3, beginning with introducing common conceptual software design of our implementation, we explain rules and guidelines for implementing graph algorithms using SPARQL.
queries. Then, we explain our implementation of each algorithm in detail. In Section 4, we show performance evaluation results of our implementations using real world datasets with RDF triplestores hosted by three different computing environments - a laptop computer, an Amazon EC2 cloud computing instance, and a shared-memory Cray XMT2 Urika graph-processing appliance. Finally, in Section 5, we conclude our work and present future work.

2. Research Background

2.1. RDF, SPARQL, and Triplestores

The World Wide Web Consortium (W3C) published a specification of RDF in 1999. RDF is an acronym for Resource Description Framework, and it is a standard data model originally designed to represent resources on the Web for data exchange. An RDF collection consists of a set of triples in the form of \(<subject> <predicate> <object>\), where \(subject\) denotes a globally unique resource, \(object\) denotes either a unique resource or a literal (i.e., a string or a number), and \(predicate\) denotes a relationship between the subject and object. Triples can be understood as statements about resources. For example, the notion “Matt is the father of John” can be represented in RDF as a triple: a subject denoting “Matt”, a predicate denoting “is the father of”, and an object denoting “John”.

RDF can facilitate data integration of heterogeneous datasets even if their underlying schema vary. Each subject, predicate, and object is identified using a URI (Unique Resource Identifier). Only object can be a literal along with the denoting “Matt”, a predicate denoting “is the father of”, and an object denoting “John”.

A number of researches related to knowledge discovery for RDF datasets have been conducted so far. Many earlier studies focused on domain-specific problems or development of algorithms or tools. For instance, Erétéo et al. (2008, 2009) focused on social network analysis problem using SPARQL to compute quantitative measures such as degree centrality and betweenness centrality, (Henschel et al., 2014; Gosal et al., 2011) presented how to utilize RDF/SPARQL for healthcare and medical science. (Heim et al., 2009, 2010) presented an algorithm for path finding in RDF dataset. (Bikakis et al., 2015) presented a large-scale RDF data visualization technique, and (Huang et al., 2011; Sun and Jin, 2010) studied how to scale up pattern matching in triplestores using parallel data processing. However, how to perform general graph mining with RDF datasets staged in a triplestore, which is our objective, has not been evaluated results of our implementations using real world datasets with RDF triplestores hosted by three different computing environments - a laptop computer, an Amazon EC2 cloud computing instance, and a shared-memory Cray XMT2 Urika graph-processing appliance. Finally, in Section 5, we conclude our work and present future work.

2. Research Background

2.1. RDF, SPARQL, and Triplestores

The World Wide Web Consortium (W3C) published a specification of RDF in 1999. RDF is an acronym for Resource Description Framework, and it is a standard data model originally designed to represent resources on the Web for data exchange. An RDF collection consists of a set of triples in the form of \(<subject> <predicate> <object>\), where \(subject\) denotes a globally unique resource, \(object\) denotes either a unique resource or a literal (i.e., a string or a number), and \(predicate\) denotes a relationship between the subject and object. Triples can be understood as statements about resources. For example, the notion “Matt is the father of John” can be represented in RDF as a triple: a subject denoting “Matt”, a predicate denoting “is the father of”, and an object denoting “John”.

RDF can facilitate data integration of heterogeneous datasets even if their underlying schema vary. Each subject, predicate, and object is identified using a URI (Unique Resource Identifier). Only object can be a literal along with the denoting “Matt”, a predicate denoting “is the father of”, and an object denoting “John”.

A number of researches related to knowledge discovery for RDF datasets have been conducted so far. Many earlier studies focused on domain-specific problems or development of algorithms or tools. For instance, Erétéo et al. (2008, 2009) focused on social network analysis problem using SPARQL to compute quantitative measures such as degree centrality and betweenness centrality, (Henschel et al., 2014; Gosal et al., 2011) presented how to utilize RDF/SPARQL for healthcare and medical science. (Heim et al., 2009, 2010) presented an algorithm for path finding in RDF dataset. (Bikakis et al., 2015) presented a large-scale RDF data visualization technique, and (Huang et al., 2011; Sun and Jin, 2010) studied how to scale up pattern matching in triplestores using parallel data processing. However, how to perform general graph mining with RDF datasets staged in a triplestore, which is our objective, has not been
much explored. In our previous work (Lee et al., 2015b), we validated our first attempt to implement three different iterative and non-iterative graph mining algorithms using SPARQL. (Techentin et al., 2014) took a similar approach, but they only covered few iterative algorithms. (Abdelaziz et al., 2015) presented an approach to present an analytic framework for graph mining algorithm processing in triplestores. This approach aims to achieve the same objective as ours, but the authors did not use the standard SPARQL to achieve their goal, so their approach is only applicable to certain modified triplestores. Also, the implemented system is not open to public.

2.2. Other Existing Systems for Graph Analysis

Graph databases are designed to optimally store, retrieve and query graph data, other than tuples in relational databases. In the previous section, we explained about triplestores, which can be considered as a specific kind of graph databases. There are many other graph databases which are not triplestores, such as Neo4j (Robinson et al., 2013), DEX (Martínez-Bazan et al., 2007), Titan, etc.). While triplestores focus on the storage and retrieval of RDF triples, these graph databases are versatile with the kinds of stored graphs (e.g., undirected/directed graphs, weighted graphs, property graphs, etc.) and query languages (e.g., Cypher (Robinson et al., 2013), Gremlin (Holzschuher and Peinl, 2013), SNQL (San Martin et al., 2011), GraphLog (Consens and Mendelzon, 1990), etc.).

There have been several efforts towards software packages for graph research such as NetworkX (Hagberg et al., 2004), iGraph (Csardi and Nepusz, 2006), and Gephi (Bastian et al., 2009). They provide a rich set of APIs for creating, manipulating, visualizing, and mining graphs; however, they have scalability issue when dealing with large-scale graphs, as they are stand-alone systems. To address the limitation, several systems focusing on processing large-scale graphs have been presented. Pegasus (Kang et al., 2009) is a graph mining system that runs in parallel manner on Hadoop (Shvachko et al., 2010), and it provides a set of built-in graph mining algorithms. Graph mining algorithms are written as a series of MapReduce tasks, which requires overhead to pass the entire state of the graph from one step to its next step. In order to overcome the challenges, Google’s Pregel (Malewicz et al., 2010) and its open-source counterpart Giraph (Avery, 2011) provide a vertex centric computational model, called Bulk Synchronous Parallel (BSP), which is suitable for large scalable graph mining tasks. They also come with a set of graph mining algorithms. GraphX (Xin et al., 2013) is a Pregel runtime for Spark (Zaharia et al., 2010), and it also carries several graph mining algorithms.

Table 1 summarizes the comparison of representative graph analysis systems we introduced. We can notice that none of the systems are fully capable of both OLGAP and GM. NetworkX (Hagberg et al., 2004), iGraph (Csardi and Nepusz, 2006), and Gephi (Bastian et al., 2009) provide a certain level of OLGAP capability; however, users need to write their own programs to perform graph pattern matching, as the systems do not provide high-level declarative graph query languages such as SPARQL or Cypher. Neo4j (Robinson et al., 2013) provides limited mining capability as its query language, Cypher, can identify shortest paths between nodes, but it does not support a comprehensive set of native graph mining capabilities. In this work, we enable GM capability in triplestores and RDF/SPARQL-based graph appliance by implementing graph mining algorithms to enable holistic in-situ graph analysis.

2.3. Graph mining algorithms

In this section, we explain six graph mining algorithms that we selected for our implementation. Note that they are widely used representative algorithms that are commonly included in the other state-of-the-art graph mining systems such as GraphX (Xin et al., 2013), Pegasus (Kang et al., 2009), or Pregel (Malewicz et al., 2010). In addition, we clarify our decisions we made for the implementation, if there are variations of algorithms.

Degree Distribution (Barabási and Albert, 1999; Faloutsos et al., 1999): Degree distribution is the probability distribution of the degrees over the whole graph, where the degree of a node in a graph is the count of the number of edges associated with that node. It is a common and fundamental statistical metric of a graph (Barabási and Albert, 1999; Faloutsos et al., 1999) that explains the characteristics of the graph. For example, earlier researchers identified that many real-world graphs such as World Wide Web, social networks, and protein-protein interaction networks approximately follow a power law degree distribution, in other words, they include few number of hub nodes connected to a huge number of other nodes with small degrees. Such graphs are called scale-free networks and they share common characteristics (Barabási et al., 2000). There have been many existing researches regarding graphs with different degree distributions and their properties. Degree distribution is simple but very useful tool for understanding properties of graphs (Callaway et al., 2000) (Barabasi and Oltvai, 2004).
Table 1. Comparison of representative graph analysis systems

<table>
<thead>
<tr>
<th>Category</th>
<th>System</th>
<th>Architecture Style / Back-end System</th>
<th>In-memory processing</th>
<th>Query Language</th>
<th>API</th>
<th>Online Graph Analytic Processing (O.GAP)</th>
<th>Graph Mining (CM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph processing library</td>
<td>NetworkX, GraphiX, iGraph</td>
<td>Standalone</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>Large-scale graph processing system</td>
<td>Pegasus</td>
<td>Distributed/ Backend</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Graph database</td>
<td>Neo4j</td>
<td>Standalone</td>
<td>X</td>
<td>Cyber</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>Titan</td>
<td>Distributed/ Cassandra, HBase</td>
<td>X</td>
<td>Gremlin</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Tripstore</td>
<td>Jena, Sesame, RDFSuite, Allegrograph</td>
<td>Standalone</td>
<td>Δ (optional)</td>
<td>SPARQL</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Graph appliance</td>
<td>Urinky-GD</td>
<td>Shared-memory</td>
<td>O</td>
<td>SPARQL</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Triangle Count (Bar-Yossef et al., 2002): As the most basic subgraph in graphs, triangles play an important role in graph analysis (Tsourakakis et al., 2009). For example, in social network graphs, counting triangles can be exploited to analyze homophily (people tend to be friends who are similar to themselves) and transitivity (people tend to be friends with friends’ friends). According to Eckmann et al. (Eckmann and Moses, 2002), counting triangles can be used to understand the hidden thematic structure of graphs. Depending on context, the way of identifying a triangle in a graph can be defined in different ways. For example, depending on purposes, we may want to consider triangles composed of the same set of nodes as identical regardless of their edge directions or order. On the other hand, we may want to consider that triangles are not the same if the edge direction between any of their node pairs is different. In this work, we count triangles by considering all edges as undirected by following the decision of SNAP (Stanford Network Analysis Project)\(^2\) so that we could confirm the correctness of our implementation by matching our results with the statistics provided by the SNAP.

Node Eccentricity (Hage and Harary, 1995): The eccentricity \(e(v)\) of a node \(v\) in a graph \(G\) is the longest distance between the node and any other node of the graph \(G\). The capability of computing eccentricity is a very important feature, as the measure can be extended to other measures such as diameter and radius (Hage and Harary, 1995; Takes and Kosters, 2013). More specifically, the maximum eccentricity of all nodes in graph \(G\) is called the diameter and the minimum eccentricity of all nodes in graph \(G\) is the radius of \(G\). Accordingly, the nodes that have the lowest eccentricity, the same value as radius, are center nodes of the graph. By definition, the eccentricity of a node in a disconnected graph is infinite; however, in order to simplify the implementation, we assume that the given graph is a fully-connected graph. When a disconnected graph \(G\) is given, eccentricity \(e(v)\) of a node \(v\) will be computed considering the largest fully-connected subgraph \(G'\) of \(G\), which includes the node \(v\) as the entire graph.

Connected Component (Qin et al., 2014): A connected component of a graph is a maximal set of nodes that can reach each other through paths in the graph. Finding all connected components in a graph is a fundamental operation that can be used for various applications such as understanding community structure of a social network, predicting trends in academic research, ranking of web pages, etc. (Varahesh and Akbari, 2014). Connected component is defined for undirected graphs. In this paper, when presented with the case of computing connected component for a directed graph, we have ignored the direction of edges.

PageRank (Page et al., 1999), Personalized PageRank (Tong et al., 2006): PageRank is one of the most widely used link analysis algorithms, and it is designed to measure the importance of nodes in a graph. The algorithm is originally designed to compute rankings of web pages that are connected with hyperlinks, but it is now generally exploited in many other graph analysis. The intuition of the algorithm is that the more a node has incoming links from important nodes, the ones with higher PageRank scores, the higher PageRank score it has. Thus, the PageRank of a

\(^2\)http://snap.stanford.edu/index.html
node is recursively defined, and it is dependent on the number and PageRank scores of all nodes from its incoming links. Computing PageRank can be explained as a random walk model. Starting from a node, the random walker keeps following a random edge. The walker periodically jumps to a random node with uniform distribution over the graph. PageRank score of a node can be interpreted as the average portion of time that the random walker spent at each node after enough numbers of iterations. Formally, PageRank is computed by, \( \vec{r} = \alpha P^T \vec{r} + (1 - \alpha) \frac{1}{N} \vec{e} \), where \( N \) is the number of vertices in the graph, \( P \) is the transition matrix for the graph, \( r_v \) is the PageRank value for node \( v \), \( \vec{e} = (1, 1, \ldots, 1)^T \), and \( \alpha \) is a damping factor, usually 0.85. Personalized PageRank is a personalized variant of the original algorithm. Computing Personalized PageRank can be explained as a random walk model just as the original PageRank. However, during the random walk process, the random walker periodically jumps to a node chosen based on the biased distribution vector \( \vec{r}' \) instead of jumping to a uniformly chosen node. Thus, Personalized PageRank is calculated as follows: \( \vec{r}' = c P^T \vec{r}' + (1 - c) \vec{t} \) by replacing the term \( \frac{1}{N} \vec{e} \) in the original equation with \( \vec{r}' \).

3. Implementing Graph Mining Algorithms using SPARQL

3.1. Software Design

Figure 2 shows the conceptual software design for implementing graph mining algorithms on an RDF triplestore. In most cases, data are already ready in triplestores since they are used as knowledge-bases in different domains, which does not necessitate data loading phase. In case that data do not exist in triplestores, the first task for graph analysis is loading the data into systems. We rely on the functionality of triplestores to load various RDF serialization formats (e.g., N-triple, Turtle, RDF/XML, etc.). RDF data sets do not need data translation phase, but in case we want to process non-RDF formatted graph datasets, it is required to transform into RDF format before data loading. We discuss the detail in Section 3.2.

RDF graph of interest is first staged into a triplestore as the default Graph. We then break down the algorithm of interest into a sequence of SPARQL-friendly processing steps. As illustrated in Figure 2, our approach uses both Python and SPARQL to process the graph data. The SPARQL queries enable processing within the triplestore, while the Python scripts interact with SPARQL endpoints and control the logical flow of the algorithm expressed as a series of SPARQL queries. We used the SPARQLWrapper library for interactions between the Python script and the triplestore.

We exploit the named-graphs feature supported by triplestores to maintain a copy of intermediate results as we process the input graph. A named-graph within a triplestore is a collection of RDF triples uniquely tagged using an identifier. A triplestore hosting a dataset can contain many named-graphs involving the same vertices and edges as the default graph. The named graphs feature is particularly important for iterative algorithms such as PageRank and connected component analysis. We will be explaining the details of each algorithm expressed as a sequence of SPARQL queries wrapped within Python scripts in the paragraphs below.

We notice that implementation of a graph mining algorithm can be either iterative or non-iterative. When it issues SPARQL queries only once, we call it a non-iterative implementation. Degree distribution algorithm can be implemented in this way. On the other hand, several algorithms issue SPARQL queries more than once for iterative processing. For example, PageRank and Connected Component need to perform iterative updates on intermediate results by repeating a sequence of SPARQL queries. Those algorithms are called iterative algorithms.

3.2. Rules and Guidelines

In this section, we share the knowledge we learned from implementing graph mining algorithms using SPARQL. We believe that this section can be referenced as rules-of-thumb for practitioners who want to use SPARQL for developing other graph mining algorithms that are not currently included in our open-source implementation.

**Interpretation of RDF data as a graph:** As we discussed in Section 2.1, an RDF dataset naturally forms a directed labeled graph, but there are several concerns to be addressed. (1) **Interpretation of nodes:** We consider subjects and objects with URIs as nodes, but we do not interpret literals as nodes. It is because literals are not identified by their values and they do not represent entities in most cases. Note that a literal is always connected to a single URI, so they

---

3http://rdflib.github.io/sparqlwrapper/
do not much affect the result of most algorithms exploiting connections of nodes. (2) **Interpretation of edges:** Directions of edges should be also carefully considered. An RDF dataset forms a directed labeled graph. So, it is natural that every triple \(<s><p><o>\) is interpreted as an edge \(<s>\rightarrow<o>\), where \(s\), \(p\), and \(o\) are URLs. However, in practice, sometimes we also need to consider a RDF data set as a undirected graph. To do that, there are two approaches. One approach is to additionally store an additional triple \(<o><p><s>\) for every original triple \(<s><p><o>\), and they are interpreted as an undirected edge \(<s>\rightarrow<o>\). This approach is simple and straightforward, but it requires additional storage space. So, we chose the other option that is to decide whether a single triple \(<s><p><o>\) is interpreted as directed or undirected edge when performing traversals in algorithm implementations. For instance, when performing a 1-hop traversal from a node identified URI \(<a>\), both triple \(<a><p><b>\) and \(<c><p><a>\) should be considered. With this approach, we can save storage space compared to the first approach. However, graph algorithms should be more carefully designed for its traversals.

**Dealing with non-RDF datasets:** In this study, we focus on using RDF datasets as the input data, since we utilize an RDF triplestore as our underlying system. Our choice of RDF is based on its standardization and prevalence. But still, there are a wide range of graph data serialization formats such as edge-list, GEXF(Group et al., 2007), GraphML(Brandes et al., 2010), Property Graph/JSON(Lee et al., 2015a). Translations of datasets in such non-RDF formats into RDF will allow our implementations to analyze them. Translating one graph data serialization format into RDF is straightforward. The following summarizes how various representations of a very simple graph which contains two nodes 0 and 1, and an edge connecting them can be translated into a RDF representation.

# edge-list
0 1

# GEXF
<gex xmlns="http://www.gexf.net/1.2draft" version="1.2">
  <graph mode="static" defaultedgetype="directed">
    <nodes>
      <node id="0" label="" />
      <node id="1" label="" />
    </nodes>
    <edges>
      <edge id="0" source="0" target="1" />
    </edges>
  </graph>
</gex>

# GraphML
Exploiting a temporary named graph to store node labels: Node labeling is a useful technique for implementing many graph mining algorithms. In the case of implementing the PageRank algorithm, for example, we can use node labeling to keep each node’s current PageRank score. We take care of node labeling by adding triples in the triplestore. Let us assume that an RDF dataset contains a node whose URI is `<node>` and a node labeled as “exampleLabel” as well. In several cases, different types of labels may be necessary. In such cases, URIs of predicates are used to distinguish the types of labels. For example, two triples `<node> <label : pageRank > “0.15”, <node> <label : numOfDegree > “5”` represent that the node `<node>` has two labels identified by URIs `<label : pageRank >` and `<label : numOfDegree >`, where the labels have different semantics and values. Node labels are stored separately in a temporary named graph (Carroll et al., 2005). Without specifying a named graph’s URI, triples are stored in the default graph. Figure 3(a) shows an example of storing in the default graph and its node labels in a separate named graph. Note that triples in the default graph and a named graph can be accessed by a SPARQL query at the same time. Figure 3(b) shows an example of querying a graph pattern composed of triples over two separate graphs.

Maintaining intermediate data as small as possible: triples move between operations in a query plan after each operation completes its own computation. Hence, any amount of such intermediate data can not only waste memory space but also make a huge impact on the performance. In order to reduce the amount of intermediate data, SPARQL queries should leverage the virtue of several types of filters in an efficient manner. For example, SPARQL provides...
a way to embed filters for patterns (FILTER, NOT EXISTS) and for aggregated results in queries (HAVING). In addition, the presences of constants in triple patterns play an important role for pruning unnecessary intermediate triples. When designing SPARQL queries, we try to actively and efficiently use those filters for making intermediate triples as small as possible.

**Avoiding frequently processing simple queries:** SPARQL queries are submitted to an SPARQL endpoint via SPARQLWrapper over HTTP protocol. Hence, if an algorithm issues a lot of simple queries, then I/O overhead can degrade the query processing performance. For example, one way of implementing the PageRank algorithm is using the Monte-Carlo method (Avrachenkov et al., 2007). Instead of deterministically computing each node’s PageRank score; Monte-Carlo method exploits a large number of probabilistically generated random walk sequences to approximate PageRank scores. Unfortunately, generating large number of random walk sequences requires sending a number of single step traverses over the graph, and it leads to issuing a lot of simple SPARQL queries. As a result, although Monte-Carlo method approach is known to be advantageous in that it has natural parallel implementation, we found that it is not suitable to be implemented using SPARQL. In order to avoid excessive submissions of queries, we choose an algorithmic design in a way that we can reduce the number of SPARQL queries, where each SPARQL query handles multiple traverses at the same time. For example, to reduce the number of SPARQL queries, we perform a node label update using aggregation SPARQL query at each iteration.

### 3.3. Detailed implementations of each algorithm

In the section, we present and explain our implementation of each selected graph mining algorithm in detail.

#### 3.3.1. Degree Distribution

Computing degree distribution using SPARQL can be done in a fairly straightforward manner. In our implementation of degree distribution, it can be computed by issuing a single query, as standard SPARQL natively supports the aggregate function, COUNT. The pseudo code of our degree distribution algorithm and the result generated from the example graph in Figure 3(a) are presented in Listing 1.

```
Pseudo Python Code with SPARQL Query:

```counting_degree_distribution```

```python
def execute_query(query):
    # Execute SPARQL query

rs = execute_query(""
SELECT ?degree (COUNT (?degree) AS ?count )
WHERE {
  SELECT (COUNT (?n) AS ?degree)
  WHERE {
    (?n out_edge ?out)
    UNION
    (?in in_edge ?n)
  }
  GROUP BY ?n
}
GROUP BY ?degree"")
print_result(rs)
```

**Example Result:**

<table>
<thead>
<tr>
<th>degree</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

The SPARQL query for degree distribution uses a nested query which computes the degrees of all the vertices by counting the number of edges connected to each vertex. In order to count the number of edges for a vertex without consideration on their directions, the nested query unions triples which represent incoming edges to the vertex and...
outgoing edges from the vertex. Then, the enclosing query performs a group by operation on the result from the nested query so that it produces the frequencies of each degree. In summary, the proposed degree distribution algorithm is simple in that it requires a single SPARQL query to be submitted only once. Hence, this algorithm is classified into a non-iterative algorithm.

3.3.2. Triangle Count

Listing 2 shows the pseudo codes for our triangle counting algorithm. The algorithm consists of two separate SPARQL queries. The first query (Query 1) is responsible for retrieving edges from the default graph and storing them into a temporary named graph <ng:temp>. As mentioned in Section 2, our implementation counts the number of triangles without consideration on edge directions. Hence, the query unions in-coming edges and out-going edges for each vertex. It is notable that the query contains a filter which enforces an lexicographical order between the labels of two vertices. The advantages of the ordering are twofold. First, the ordering prevents several isomorphic triangles from being counted. There are \( 2^3 = 8 \) possible triangle patterns as each of three edges of a triangle can have two directions, so removing the duplicated triangles is a key challenge. However, the order enforcement enables only one representative edge between any two vertices to be stored. Second, by pruning unnecessary edges not in order, the ordering can reduce the intermediate data a lot. The second query (Query 2) retrieves triples from the temporary graph which form triangles and counts the number of resulting triangles. Since the SPARQL queries are issued only once, our triangle counting algorithm is considered as a non-iterative algorithm.

```
Listing 2. Counting triangles in a graph
Pseudo Python Code with SPARQL Query:

''' Creating a temporary graph for removing duplicated triangles - Query 1 '''
execute_update('"
DROP GRAPH <ng:temp>;

INSERT { GRAPH <ng:temp> { ?vertex <label:connect> ?neighbor. }}
WHERE {
  SELECT ?vertex ?neighbor
  WHERE {
    (?vertex ?p ?neighbor . )
    UNION {
      FILTER (STR(?vertex) < STR(?neighbor))
    }
  }
}"

''' Counting unique triangles - Query 2 '''
rs = execute_query('"
SELECT (COUNT(DISTINCT *) AS ?count)
WHERE {
  GRAPH <ng:temp>
  }
}
"
print_result(rs)
Example Result:
-----------------
<table>
<thead>
<tr>
<th>numOfTriangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
-----------------
```

3.3.3. Node Eccentricity
Listing 3. Computing eccentricity of a node with the URI <node:8>.

Pseudo Python Code with SPARQL Query:

```python
input_node = "<node:8>"

''' Attach a label to input node '''
execute_update(""
DROP GRAPH <ng:labels>; CREATE GRAPH <ng:labels>;
INSERT DATA{
GRAPH <ng:labels> {""input_node="" <label:flag> 1} ;
}"
)

iter_no = 0
saturated = False

''' Spread the label of each node to its adjacent nodes - Query 2 '''
while not saturated:
    execute_update(""
    INSERT { GRAPH <ng:labels>
    { ?neighbor <label:flag> {""+str(iter_no+1)+""} . }
    WHERE
    {
        SELECT ?neighbor ?label
        WHERE
        {
            { GRAPH <ng:labels>
                { ?node <label:flag> ""+str(iter_no)+"" . } }
            { ?node ?edge ?neighbor. }
            UNION
            { ?neighbor ?edge ?node. }
        }
        NOT EXISTS
        { GRAPH <ng:labels>
            { ?neighbor <label:flag> ?any. }
        }
    }
    }
}
    iter_no+=1
    if the number of triples in the graph <ng:labels> did not increase:
        saturated = True
print " - The eccentricity of given node: " + node+ " is : " +str(iter_no - 1)
```

Example Result:
- The eccentricity of given node: <node:8> is : 3

Our algorithm for computing node eccentricity is an iterative algorithm and based on the idea of label spreading over a graph. Listing 3 shows the pseudo code of the algorithm which computes the eccentricity for a given input vertex. The operation of the algorithm can be explained as follows. Initially, a temporary named graph, <ng:labels>, is created. As the starting vertex, a triple of (<node:8>, <label:flag>, 1) is stored in the temporary graph. The object literal 1 means that the vertex should be retrieved at the first iteration of the following loop. The loop retrieves all the vertices from the temporary graph which correspond to the current iteration number, i, finds all the neighbors to those vertices, and stores the neighbors as triples of (neighbor, <label:flag>, i + 1) into the temporary graph. This process is performed until the number of triples in the temporary graph remains same. By labeling vertices with their iteration numbers, the algorithm can retrieve only corresponding vertices, reducing computation overhead. In addition, the use of NOT EXISTS operation ensures that only vertices that have not been explored until the current iteration can be inserted into the temporary graph. This can increase the overall performance by avoiding unnecessary insert operations.

3.3.4. Connected Component
Listing 4. Finding connected components in a graph

Pseudo Python Code with SPARQL Query:

```python
''' Initialize a named graph for storing labels - Query 1 '''
execute_update(""
    DROP GRAPH <ng:labels>; CREATE GRAPH <ng:labels>;
    INSERT { GRAPH <ng:labels> { ?s <label:comp_id> ?s.}}
    WHERE
    { (?s ?p ?o.)
      UNION
      { ?o ?p ?s.}
    }"
);

''' Iteratively update label of each node referring to the labels of the node's adjacent nodes - Query 2'''
converged = False
while not converged:
    execute_update(""
    DELETE { GRAPH <ng:labels>
        { ?node <label:comp_id> ?original.}}
    INSERT { GRAPH <ng:labels>
        { ?node <label:comp_id> ?update.}}
    WHERE
    {
        SELECT ?node (MIN(?label) AS ?update)
        WHERE
        {
            { GRAPH <ng:labels>
                { ?node <label:comp_id> ?minimum.}
            }
            { ?node ?edge ?neighbor.}
            UNION
            { ?neighbor ?edge ?node.}
            { GRAPH <ng:labels>
                { ?neighbor <label:comp_id> ?label.}
            }
            FILTER (STR(?minimum) > STR(?label))
        }
        UNION
        { GRAPH <ng:labels>
            { ?node <label:comp_id> ?label.}
        }
        GROUP BY ?node
    }
    { GRAPH <ng:labels>
        { ?node <label:comp_id> ?original.}
    }
    FILTER (?original != ?update)
    }"
    if there is no more update in <ng:labels>:
        converged = True

''' Retrieve the results - Query 3 '''
rs = execute_query(""
SELECT (?s as ?node) (?o as ?comp_id)
WHERE { GRAPH <ng:labels> {?s <label:comp_id> ?o};}
"
)
print_result(rs)

Example Result:
--------------
<table>
<thead>
<tr>
<th>node</th>
<th>comp_id</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="">node:4</a></td>
<td>&quot;node:1&quot;</td>
</tr>
<tr>
<td><a href="">node:6</a></td>
<td>&quot;node:1&quot;</td>
</tr>
</tbody>
</table>
```

13
The pseudo codes of our connected component algorithm are shown in Listing 4. This algorithm is iterative since a query is repeatedly processed for finding connected vertices through edges in a graph. The algorithm consists of three SPARQL queries. Initially, (Query 1) creates a temporary named graph, \(<ng:labels>\), and iterates over the entire triples in the input graph, producing per-vertex state triples of \((n_i, <\text{label}:\text{min}, l_{n_i})\) in the temporary graph. Each vertex, \(n_i\), has a state, \(l_{n_i}\), linked with \(<\text{label}:\text{min}>\). At the initial time, \(l_{n_i}\) is set to the corresponding vertex’s label. At the iteration step, (Query 2) is iteratively executed so that \(l_{n_i}\) is set to the minimum label in the \(n_i\)’s egocentric network. That is, for each node in the \(G\), the associated label in the temporary graph is updated as the minimum label among all of labels of the node’s adjacent nodes in \(G\) including its own corresponding label. For example, if \(<\text{node:6}>\) is the label of a node whose adjacent nodes’ labels are \(<\text{node:4}>\), \(<\text{node:5}>\), and \(<\text{node:9}>\), the label of the node is updated to be \(<\text{node:4}>\) since \(<\text{node:4}>\) is the lexicographical minimum among the four labels. (Query 2) is repeatedly processed until there is no state modification in the temporary graph.

As a way to improve the query processing performance, we exploit two filter operations. First, the ‘FILTER (STR(?minimum) > STR(?label))’ is to pre-filter the labels of adjacent nodes which will be eventually pruned by MIN(?label) from being propagating through the group by operation. Hence, the filter operation can reduce the intermediate data, reducing computational overhead and saving memory space. Second, the ‘FILTER (?original !\(\not\) ?update)’ reduces the number of updates by avoiding the situation that labels in the temporary graph are updated with the same values. The reduced number of updates can improve the query processing performance. At the end of computation, (Query 3) retrieves \(<\text{label:comp_id}>\) label’s value of each node. The value of \(\text{comp_id}\) identifies in which component each node is included. After computation, all nodes in the same connected component will have a same label, and the number of distinct \(<\text{label:comp_id}>\) labels will be the number of graph components. Listing 4 shows the example result of finding connected component computation for the graph in Figure 3(a). The result shows that all of nodes in the example graph are fully connected and included in the same component identified by the value “node:1”.

3.3.5. PageRank/Personalized PageRank

Listing 5. Computing PageRank for a directed graph

Pseudo Python Code with SPARQL Query:

```python
''' Get number of nodes in the graph '''
numOfNodes = get_node_num()

''' Parameters '''
dampingFactor = 0.85
addTerm = (1-dampingFactor)/numOfNodes
top_k = 10
convergenceThreshold = 1e-6

''' Initialize labels - Query 1a '''
execute_update(""
DROP SILENT GRAPH <ng:labels>; 
INSERT { GRAPH <ng:labels>
  { ?s <label:outdegree> ?outdegree. } }
WHERE {
  SELECT ?s (COUNT(*) AS ?outdegree)
  { ?s ?p ?o. }
  GROUP BY ?s
}"); 

```
```sql
''' Set initial PageRank scores (1/numOfNodes) - Query 2a '''
execute_update(''
 INSERT { GRAPH <ng:labels>
  { ?node <label:PR> """+str(1/numOfNodes)+"""".}}
 WHERE {
  SELECT DISTINCT ?node
  WHERE {
    ?node ?edge [].
    UNION
    { [] ?edge ?node.}
  }
}
'''

''' Power iteration computation - Query 3a '''
while not converged:
  execute_update(''
 DELETE { GRAPH <ng:labels>
  { ?node <label:diff> ?prevDiff.}}
 WHERE {
  GRAPH <ng:labels> ( ?node <label:diff> ?prevDiff.)
 }
 DELETE { GRAPH <ng:labels> { ?node <label:PR> ?prevPR.}}
 INSERT {
  GRAPH <ng:labels>
  { ?node <label:PR> ?prevPR ; <label:diff> ?diff.}}
 WHERE {
  { 
    SELECT ?node ?prevPR
    (SUM(?neighborPR/?degree)*""""+str(dampingFactor)+""""+str(addTerm)
    WHERE {
      { 
        GRAPH <ng:labels>
        { ?node <label:PR> ?prevPR.}
      }
      { 
      }
      GRAPH <ng:labels>
      { 
        ?neighbor <label:outdegree> ?degree ;
        <label:PR> ?neighborPR.
      }
    }
  GROUP BY ?node ?prevPR
  HAVING (?diff >="""
     + str(convergenceThreshold) +"")
  }
  ;
  '''}

''' Checking if converged - Query 4 '''
rs = execute_query(''
 SELECT (COUNT(*) AS ?changed)
 WHERE { GRAPH <ng:labels>
  { ?node <label:diff> ?diff.}}
'''
changed = rs["changed"][0]
if changed == 0:
  converged = True

''' Retrieve the results - Query 5 '''
rs = execute_query(''
 SELECT ?node ?pagerank
 WHERE { GRAPH <ng:labels>
  { 
'''
```

// Expert Systems with Applications 00 (2015) 1–25
15
As eccentricity and connected component algorithms, our SPARQL-based PageRank algorithm is classified into iterative algorithms. The PageRank algorithm is shown in Listing 5. At the initial step, the PageRank algorithm requires to calculate outdegrees and initial PageRank scores \(1/N\) for each vertex and store them into the temporary named graph \(<\text{ng:labels}>\) where \(N\) is the number of nodes in the input graph. (Query 1a) and (Query 2a) are executed for computing outdegrees and initial PageRank scores, respectively. Since (Query 1a) focuses only on outdegree computation, it contains just one triple pattern, \(?s\ ?p\ ?o\). Per-vertex outdegree forms a triple of (\(\text{vertexlabel}, <\text{label:outdegree}, \text{outdegree}>\)) and per-vertex PageRank score forms a triple of (\(\text{vertexlabel}, <\text{label:PR}, \text{score}>\)) in the temporary graph.

The algorithm can be slightly modified to support Personalized PageRank computation. For Personalized PageRank, instead of setting uniform initial scores, biased scores can be set as shown in (Query 2b) Listing 6.

Listing 6. Example of setting biased initial scores for computing Personalized PageRank

Pseudo Python Code with SPARQL Query:

```python
def set_biased_initial_scores():
    # Query 2b
    execute_update(""
    INSERT DATA {
        GRAPH <ng:labels> {
            <node:1> <label:prevPR> 0.5.
            <node:3> <label:prevPR> 0.4.
            <node:5> <label:prevPR> 0.1.
        };
    }
    ");
```

At the iterative computation step, (Query 3a) is repeatedly executed so that a new PageRank score for each vertex is computed by combining the PageRank scores of its neighbors. At the same time, the difference between the previous and the current PageRank scores is calculated. By combining the new PageRank and the difference computations into a single SPARQL query, our algorithm can reduce the number of queries that are iteratively issued and processed. In addition, the algorithm reduces the number of PageRank updates by using a HAVING filter. By using the filter, newly calculated PageRank values are updated only when the differences are greater than or equal to the convergence threshold. In the end of each iteration, (Query 4) checks whether the algorithm converges. As a convergence condition, it checks the number of triples whose property is \(<\text{label:diff}>\). If there have been any such triples inserted at the current iteration, the algorithm starts next iteration. Otherwise, the algorithm stops iterations. At the end of the computation, (Query 5) is executed so that the top-k highest PageRank scores are printed by retrieving the values of \(<\text{label:PR}>\) in \(<\text{ng:labels}>\).

Our implementation considers that a triplestore forms a directed graph and compute PageRank for the directed graph. However, in some cases, we may want to consider a triplestore as an undirected graph. In this case, we need to consider a triple \(<s><p><o>\) as two directed edges \(<s><o>\) and \(<o><s>\). In this case, (Query 1b) and (Query 3b) shown in Listing 7 should be used instead of (Query 1a) and (Query 3a), respectively.

Listing 7. Queries for computing PageRank for undirected graphs
Pseudo Python Code with SPARQL Query:

```python
... ''' Initialize labels - Query 1b '''
execute_update(""");
DROP SILENT GRAPH <ng:labels>;
INSERT { GRAPH <ng:labels>
    { ?x <label:outdegree> ?degree } }
WHERE
    { SELECT ?x (COUNT(*) as ?degree)
    { (?x ?p ?o)
      UNION
      (?x ?p ?x)
    }
    GROUP BY ?x
};
```

... ''' Power iteration computation - Query 3b '''

while not converged:
    execute_update(""");
    DELETE { GRAPH <ng:labels>
        { ?node <label:diff> ?prevDiff.}
    WHERE
        { GRAPH <ng:labels>
        { ?node <label:diff> ?prevDiff. }
    }
    DELETE { GRAPH <ng:labels>
        { ?node <label:PR> ?prevPR.}
    INSERT { GRAPH <ng:labels>
        { ?node <label:PR> ?newPR ;<label:diff> ?diff.}
    WHERE
        { SELECT ?node ?prevPR
            (SUM(?neighborPR/?degree)*"" +str(dampingFactor)"" +str(addTerm)
    WHERE { { GRAPH <ng:labels>
            {{ ?node <label:PR> ?prevPR.}}
        } { ?neighbor ?edge ?node. }
        { GRAPH <ng:labels>
        { ?neighbor <label:outdegree> ?degree ;
            <label:PR> ?neighborPR .}
        }
    }
    UNION
    { { GRAPH <ng:labels>
            {{ ?node <label:PR> ?prevPR.}}
        } { ?node ?edge ?neighbor. }
        { GRAPH <ng:labels>
        { ?neighbor <label:outdegree> ?degree ;
            <label:PR> ?neighborPR .}
        }
    }
    GROUP BY ?node ?prevPR
    HAVING (?newPR != ?prevPR)
});
```
4. Performance Evaluation

4.1. Experimental Setup

Comprehensive and fair comparison of graph analysis systems involves with many challenges due to heterogeneity of systems including system designs, controllable parameters, the way of operating the system, and etc., as we discussed in our previous studies (Lim et al., 2015; Hong et al., 2015). Thus, in order to provide a general reference for relative performance difference between systems, we chose 6 real world datasets (Leskovec and Krevl, 2014) released by Stanford Network Analysis Project (SNAP) (Leskovec and Sosič, 2014), which are one of the most widely used data sets for graph analysis system performance benchmarking. Since SNAP data sets are released in edge-list format, and not in RDF format, we needed to translate the data sets into RDF format as discussed in Section 3.2.

Performance evaluation of our implementation was conducted in three different computing environments. (1) A laptop computer with a 2.3 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory where OS X Yosemite 10.10.1 is installed. (2) An Amazon Web Service (AWS) r3.2xlarge instance which is equipped with 8 Intel Xeon E5-2670 v2 processors, 61 GB memory, and a 320 GB SSD where Ubuntu Server 14.04 LTS runs. (3) A Cray’s supercomputer-based graph processing appliance, named Urika at Oak Ridge National Laboratory. For (1) and (2), we installed Apache Jena Fuseki SPARQL server 1.0.2 which comes with Jena TDB triplestore and Java runtime 1.8.0 with configuring 80% of each system’s memory as heap space for JVM. (3) contains 2 TB of memory and is designed to provide massively parallel, multi-threaded, shared-memory computing environments specialized for processing SPARQL queries over large-scale RDF data sets. It contains Apache Jena as its SPARQL engine. Table 1 shows the summary of data sets we used for our evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th># of edges</th>
<th># of vertices</th>
<th># of triangles</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-Astro(AS)</td>
<td>396,160</td>
<td>18,772</td>
<td>1,351,441</td>
<td>Collaboration network of Arxiv Astro Physics, undirected</td>
</tr>
<tr>
<td>soc-Epinion1(EP)</td>
<td>508,837</td>
<td>75,879</td>
<td>1,624,481</td>
<td>Who-trusts-whom network of Epinions.com, directed</td>
</tr>
<tr>
<td>com-DBLP(DB)</td>
<td>1,049,866</td>
<td>317,080</td>
<td>2,224,385</td>
<td>DBLP collaboration network, undirected</td>
</tr>
<tr>
<td>com-Youtube(YU)</td>
<td>2,987,624</td>
<td>1,134,890</td>
<td>3,056,386</td>
<td>Youtube online social network, undirected</td>
</tr>
<tr>
<td>web-Google(GO)</td>
<td>5,105,039</td>
<td>875,713</td>
<td>13,391,903</td>
<td>Web graph from Google, directed</td>
</tr>
<tr>
<td>Wiki-talk(WI)</td>
<td>5,021,410</td>
<td>2,394,385</td>
<td>9,203,519</td>
<td>Wikipedia talk (communication) network, directed</td>
</tr>
<tr>
<td>soc-Pokec(PO)</td>
<td>30,622,564</td>
<td>1,632,803</td>
<td>32,557,458</td>
<td>Poke online social network, directed</td>
</tr>
<tr>
<td>soc-LiveJournal1(LJ)</td>
<td>68,993,773</td>
<td>4,847,571</td>
<td>285,730,264</td>
<td>LiveJournal online social network, directed</td>
</tr>
<tr>
<td>com-Orkut(OR)</td>
<td>117,185,083</td>
<td>3,072,441</td>
<td>627,584,181</td>
<td>Orkut online social network, undirected</td>
</tr>
</tbody>
</table>

In this evaluation, we assume that each input dataset is already loaded into triplestores and exclude data loading times from execution times. When the execution of any algorithm is not completed in 24 hours or interrupted due to errors, the execution times for such algorithms are not depicted in graphs. Because com-Orkut dataset could not be loaded into the triplestore on (1) due to the lack of enough memory, we omitted the result from the graphs. In the following sections, we explain the parameters that we used for the execution of each algorithm and discuss the experimental results and related issues. We use Mac, AWS, and Urika abbreviation respectively for each computing environment.

4.2. Performance comparison of non-iterative algorithms

As shown in Section 2.3, our degree distribution and triangle count algorithms are not iterative in that they do not repeatedly issue SPARQL queries. In addition, both algorithms do not take any input parameters. For those non-iterative algorithms, we measured their execution times with the datasets listed in Table 2 in the three different computing environments.

**Degree distribution**: Figure 4(a) shows the execution times of our degree distribution algorithm on the three different machines. The degree distribution algorithm is relatively simple compared to the other algorithms. For
every dataset, the execution times of the algorithms were less than 300 seconds. On *Urika*, the algorithm was able to complete its computation in less than one minute even with large datasets. However, *Urika* showed a little worse performance than the other machines for a few small datasets due to its initialization overhead. In cases of *Mac* and *AWS*, the execution times increase exponentially as the input graph dataset becomes large. In general, *Mac* showed slightly better performance than *AWS* until its memory capacity allowed intermediate data of the algorithm to be efficiently managed. However, with large datasets (e.g., *soc-LiveJournal1*), *Mac* showed relatively high performance degradation due to the limited memory capacity. Overall, the exceptional processors and memory capacities enable *Urika* to outperform the other two machines even with large data sets.

**Triangle count:** Figure 4(b) depicts how many seconds our triangle counting algorithm took in the three different environments. *Mac* and *AWS* raised out-of-memory errors with *soc-pokec* and *soc-LiveJournal1* and *soc-LiveJournal1* and *com-Orkut*, respectively. In both environments, the algorithm showed rapid performance degradation from *Youtube* due to the limitation on available memory. In case of *Urika*, the algorithm took less than 3000 seconds for all the datasets without errors. The performance of our triangle counting algorithm degraded much faster than that of the degree distribution did as the size of datasets increased. This is because the triangle counting algorithm is more complex than the degree distribution algorithm, producing more intermediate data. In addition, compared to the degree distribution, the triangle counting algorithm showed no strong relation between dataset size and execution time.

### 4.3. Performance comparison of iterative algorithms

**Graph eccentricity:** Graph eccentricity takes a node as its parameter for its computation, and the performance of eccentricity depends on the given node. For performance evaluation, we ran the algorithm with a randomly selected node in the given graph for five times. Figure 5(a) shows the comparison of average performance for graph eccentricity computation. It is noticeable that the execution times significantly increase for *Mac* and *AWS* but not for *Urika*. As results, with *Wiki-talk* dataset, *Urika’s* performance is 705.130 times better than *Mac*. Overall, *Urika* shows the promising performance, and it finishes computing eccentricity for the largest dataset *com-Orkut*, composed of over hundred million edges, in just 124.268 seconds. The eccentricity algorithm was not able to complete for one day with the three largest datasets both on *Mac* and *AWS*. 

---

Figure 4. Performance evaluation of non-iterative algorithm implementations

Figure 5. Performance evaluation of iterative algorithm implementations
Connected components: In Figure 5(b), comparison of execution times for connected component with different datasets is depicted. It takes 2135.886 seconds to compute connected components of Wiki-talk dataset on Mac; however, for three larger datasets soc-pokec, soc-LiveJournal1, and com-Orkut, it is not able to compute connected components on Mac due to not enough memory. In the case of AWS, it takes 6679.507 seconds to compute connected components of soc-pokec dataset, but still it is not able to finish the larger datasets in the given time frame (1 day). Urika shows excellent performance again, but note that it takes longer to process smaller dataset ca-AstroPh on Urika. We reckon that it is because that overhead of the initialization, inserting triples for labeling for each node, is larger in comparison with the other computing environments.

PageRank: For PageRank computation, we need to select several parameter values. For the damping factor, we used the typical value 0.85. Convergence threshold was selected as $\frac{1}{N} \times 0.1$, where $N$ is the number of nodes in the given graph. Additionally, we set the maximum number of iterations to 50. It should be noticed that three datasets ca-AstroPh, com-Dblp, and com-YouTube are undirected graphs so the PageRank computation should be handled differently with the other datasets. In Figure 6, it shows the execution time of each iteration for computing PageRank of soc-Epinion1 dataset. What we can notice is that the processing time for each iteration decreases as iteration proceeds. It is because, when updating PageRank scores, we filter out the nodes whose difference between current and previous scores already became less than the convergence threshold. Figure 6 confirms that our optimization is effective in practice. Figure 5(c) shows the performance comparison of PageRank computation. Mac and AWS can finish PageRank computation of soc-pokec dataset around 10 hours; however, computing PageRank for larger dataset is not feasible. However, on Urika, even with the largest dataset, PageRank computation is completed less than one hour.

For both iterative and non-iterative graph mining algorithms, our results are very comparable to results from linear-algebra based implementations of the same algorithms. A comparison of our implementation with different linear-algebra based methods and map-reduce based methods for graph processing is presented in (Lim et al., 2015). The results in (Lim et al., 2015) along with the results in this paper clearly confirm that SPARQL-based implementation of graph mining algorithms can be unleashed on graph-data represented and hosted using Semantic Web standards and technologies.

5. Conclusion and Future Work

In this work, we presented our efforts to realize holistic in-situ graph analysis which is to perform OnLine Graph Analytic Processing (OLGAP) and Graph Mining (GM) in a single system. To achieve our goal, we take an approach to enable GM capability in triplestores that are originally developed for efficient processing OLGAP tasks, as we can take the advantages of RDF data model and SPARQL query language. We implemented six fundamental graph mining algorithms - degree distribution, triangle count (Tsourakakis, 2008), graph eccentricity (Hage and Harary, 1995), connected component, PageRank/Personalized PageRank (Page et al., 1999; Tong et al., 2006).

Triplestores have been prevalent in many domains accumulating an amount of domain-specific data. Our implementation can bring both OLGAP and GM capabilities to those existing triplestores. This is especially valuable in that users can perform “in-situ” graph analysis without considerations of adapting any other graph analysis systems and it prevents expensive data exporting-transformation-importing step. We expect that our guidelines and reference
implementations of graph analysis algorithms help other researchers to develop their own extended graph mining algorithms on triplestores. We initiated an open source project on github and it is now publicly available 4.

We tested our performance of graph mining algorithms with triplestores hosted in different computing environments. A number of million-scale real world graphs can be practically processed with a general computing environment such as laptop or a cloud instance. However, in order to process graph mining for larger graphs, we notice that the memory size is the key property of computing environment to achieve scalability of graph mining in triplestores. Since AWS has larger memory in comparison with Mac, it could deal with some cases that was not handled by Mac. As expected, graph processing super computer with 2TB of shared-memory, Urika outperforms Mac and AWS for most cases with very few exceptional cases. Among the implemented algorithms, PageRank requires the longest processing time. But, with Urika, we were able to get the PageRank computation results for the largest tested data set com-Orkut (117 million edges) less than a hour. It shows the feasibility of Urika system as a graph mining system that is powered by our implementations.

Most researches conducted on graph analysis focused on achieving specific analytic operations (McColl et al., 2014; Lim et al., 2015; Shao et al., 2012); however, we envision that many applications in graph analysis require processing cyclic work-flows of disparate analytic operations. Realizing a holistic in-situ graph analysis system is a new challenge we face, and it has not been much explored so far. We believe that our approach is the first step to bridge the gap between two different dimensions of graph analysis that used to be processed in multiple systems.

For future work, we plan to keep extending our graph mining algorithm set to cover more algorithms (e.g., Single source Shortest Path, Radius, Peer-pressure clustering etc.). In addition, there are remaining interesting subjects that we are planning to tackle in the future work as follow:

**Graph mining algorithm performance tuning:** From our experience, we notice that the performance of a graph mining algorithm implemented using SPARQL is closely related to the size of intermediate data and the query execution plan of SPARQL query processing engine. We plan to continue optimizing the performance of currently implemented algorithms and establish the common guidelines for implementing optimized graph mining algorithms.

**Developing building-blocks of graph mining algorithms:** While implementing algorithms using SPARQL queries, we identified several common aspects of graph mining algorithms such as node labeling, node-label propagation, and neighborhood-based node aggregation. We will identify the necessary yet fundamental building blocks for graph mining algorithms and develop a set of application programming interfaces (APIs) to assist user-derived custom graph mining algorithms. These APIs will enable programmers to implement graph mining algorithms on triplestores with minimal understanding of SPARQL-algebra - analogous to how business intelligence tools abstract Structured Query Language (SQL) algebra in traditional databases for the end user.

References


4https://github.com/ssrangan/gm-sparql


Group, G. W. et al. (2007). Gexf file format. Gephi Community project.


distributed database system: A case study for managing energy data on mobile devices. International Journal of

system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference
on Management of data, pages 135–146. ACM.

sixteenth ACM conference on Conference on information and knowledge management, pages 573–582. ACM.

source graph databases. In Proceedings of the first workshop on Parallel programming for analytics applications,
pages 11–18. ACM.


Robinson, I., Webber, J., and Eifrem, E. (2013). Graph databases. ” O’Reilly Media, Inc.”.

cities, 5:r5.

management approaches in a SPARQL benchmark scenario. Springer.

ACM.

Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pages 1–10. IEEE.

Sukumar, S. R. and Bond, N. (2013). Mining large heterogeneous graphs using cray’s urika. 2013 ORNL Compu-
tional Data Analytics Workshop.

Sun, J. and Jin, Q. (2010). Scalable rdf store based on hbase and mapreduce. In Advanced Computer Theory and
Engineering (ICACTE), 2010 3rd International Conference on, volume 1, pages V1–633. IEEE.

118.

Techentin, R. W., Gilbert, B. K., Lugowski, A., Dewese, K., Gilbert, J. R., Dull, E., Hinckley, M., and Reinhardt,


In Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on, pages 608–617. IEEE.

with a coin. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and
data mining, pages 837–846. ACM.

