Graph-based Exploration of Non-graph Datasets

Udayan Khurana
IBM Research
ukhurana@us.ibm.com

Srinivasan Parthasarathy
IBM Research
spartha@us.ibm.com

Deepak Turaga
IBM Research
turaga@us.ibm.com

ABSTRACT

Graphs or networks provide a powerful abstraction to view and analyze relationships among different entities present in a dataset. However, much of the data of interest to analysts and data scientists resides in non-graph forms such as relational databases, JSON, XML, CSV and text. The effort and skill required in identifying and extracting the relevant graph representation from data is often the prohibitive and limits a wider adoption of graph-based analysis of non-graph data. In this paper, we demonstrate our system called *GraphViewer*, for accelerated graph-based exploration and analysis. It automatically discovers relevant graphs implicit within a given non-graph dataset using a set of novel rule-based and data-driven techniques, and optimizes their extraction and storage. It computes several node and graph level metrics and detects anomalous entities in data. Finally, it summarizes the results to support interpretation by a human analyst. While the system automates the computationally intensive aspects of the process, it is engineered to leverage human domain expertise and instincts to fine tune the data exploration process.

1. INTRODUCTION

Graphs help capture interactions and relationships between entities. Phenomena such as financial transactions, social interactions, biological structures, movements of people and goods, are naturally represented as graphs. The analyses of such processes in a graph-theoretic manner provides us additional insights into the characteristic of the entities and dynamics of the network as a whole. For instance, centrality based metrics such PageRank, or Betweenness tell us the relative importance of different entities in the network. On the other hand, graph-level metrics such as density or conductivity help us reason about the characteristics of the graph as a whole. Techniques such as clustering or anomaly detection on graphs help with fraud detection, advertising suggestions, and retail targeting amongst others.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

In spite of the inherent strength of graph-based analysis of datasets, it is still not widely used. This is due to the reason that much of the data in the digital world resides in non-graph formats such as relational databases or XML or CSV or unstructured text. The burden of identifying and extracting a potentially meaningful graph from such data is placed on the analyst. Several graphs that are implicitly present are not explicitly visible. In a complex XML schema or a relational database with dozens of tables, for instance, this task becomes especially prohibitive. Moreover, extraction of graphs from such datasets requires the analyst to write complex code or queries. For instance, consider a simple schema of an author-publication database in Figure 4, which can yield graphs such as co-authorship, author-publication, authorship in the same conference, etc. The co-authorship graph extraction query in SQL can be seen Figure 3(b). With complex schemas, and complex graph definitions, such extraction queries become significantly more complex. In case of non-relational source formats, the querying mechanism requires more intricate programming. While there has been recent work on declarative specification for graph extraction [9] [6], the analyst is still required to overcome the burden of identifying and expressing a potentially meaningful graph. The work presented in this paper is motivated by the complexity of graph discovery in data analytics.

Figure 1 describes the typical methodology for performing graph-based exploration of a dataset, where an analyst would extract a graph, run a particular algorithm and interpret results. This is repeated in a recursive manner until she finds the right combination of a graph and algorithm for a desired outcome. This activity is time-consuming, intensive in query or code writing, and repetitive in nature. With the objective of saving analyst cycles and the need to write ETL code, we propose a system for graph discovery, extraction and analysis from non-graph data, called *GraphViewer*[1]. The cornerstone of GraphViewer is an engine called *GraphMapper* that enumerates and extracts several implicit graphs in a given dataset and relieves the analyst the burden of graph discovery and extraction. Further, the Metric Computation and Summarization (MCS) engine matches all the graphs obtained from the GraphMapper with compatible algorithms based on each graph's type and characteristic. It then runs the algorithms, collects and summarizes result for analyst's interpretation. While the methodology is based on automated enumeration, extraction and analysis of the underlying graphs, the system is

[1] GraphViewer at IBM Research: https://ibm.biz/Bd4AMr

receptive to an analyst's domain expertise and instinct. It tunes the execution plans as per the analyst's directions.

In this work, our focus is on enabling the analyst to perform graph based knowledge discovery conveniently and efficiently. Our key contributions are, (a) a novel methodology for performing graph-based exploration of non-graph datasets; and (b) an engine for graph enumeration, extraction, and graph-based exploration on a variety of non-graph datasets. Note that we do not focus on suggesting new metrics of graph analysis or anomaly detection, but utilize the existing work in that area to compliment our system. We refer the interested reader to explore those topics through Aggarwal [1, Chapter 17,19] and Akoglu et al. [2]. In the rest of this paper, we present an overview of the GraphViewer system and briefly describe it components, GraphMapper and MCS in Section 2. In Section 3, we present the user interface and describe the usability and demonstration plans, including some insights from our analysis of the 2016 US presidential candidate debate transcripts.

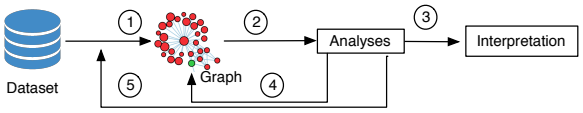


Figure 1: Illustration of graph analytics methodology: (1) graph extraction from the dataset, (2) running an analysis algorithm on the graph, (3) interpretation of results, (4) running a different algorithm on the same graph, (5) finding another graph from the dataset.

2. GRAPHVIEWER OVERVIEW

We begin with a brief overview of the components in GraphViewer, followed by a description of specific components, GraphMapper and MCS, respectively; we conclude with the system implementation details.

2.1 System Architecture

GraphViewer's high level architecture and data flow can be seen in Figure 2. GraphMapper generates a *Graph Spec* describing a list of enumerated graphs for the input dataset. The user may modify, add or remove graph descriptions from the specification. The specified graphs are then extracted and passed on to the MCS. MCS matches the graphs with a list of metric computation algorithms, constructing a *Run Spec*, which may be modified according the analyst's preference. The final specification is executed and the results are presented in a summarized manner, ready for interpretation.

2.2 Graph Discovery and Extraction

GraphMapper uses a principled approach for analyzing a given dataset and enumerating a set of graphs that are implicitly present in the dataset. The key steps in this approach are: (a) first reads (and mines) entity types and the direct relationships between the entity-types found in the data; (b) abstracts them in a network of entity-types and relationships; and (c) based on the connected components of the network, it discovers further (hidden) relationships, which in turn, form the basis for enumerating data graphs from the given dataset. These steps are explained below.

When possible, GraphMapper exploits the structure in the data to determine entities and relationships. For instance, in a normalized relational schema, it makes use of

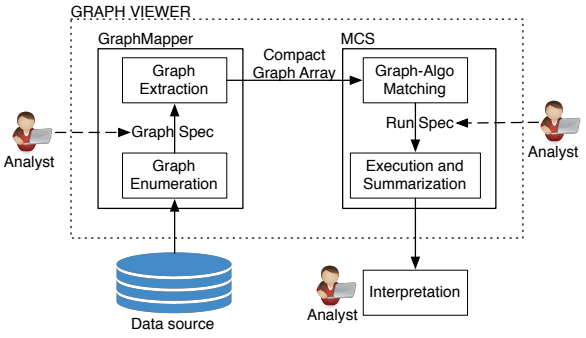


Figure 2: GraphViewer architecture. The dashed connections reflect the analyst's ability to modify a specification.

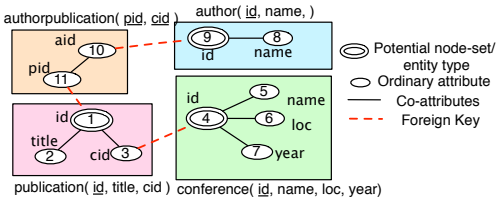
the knowledge of primary keys or uniqueness constraints to find potential *node-sets*; and, it uses the knowledge of foreign keys, co-attributes (in the same relation) or functional dependencies to describe relationships between various entity groups or node-sets. Additionally, in cases where such structure is absent in the data, it adopts a data-driven approach to locate entities and relationships. For instance, in textual data, it performs entity-resolution to extract different entities that are categorized by their types, such as "person", "organization", "location", forming different *node-sets*; it uses natural language processing techniques, and additionally, several heuristics such as proximity of occurrence or co-occurrence to determine the potential relationships. For semi-structured datasets such as JSON, XML, or tabular datasets such as CSV, GraphMapper tries to induce structure and relationships using a host of techniques. These include a survey on entity resolution by Brizan et al. [3], XML to relational schema conversion by Lee et al. [4], identification of knowledge graphs [7], and automatic schema matching for relational databases [8].

We now describe the graph enumeration setup in the context of a relational data source, that was briefly introduced in [9]. For a relational dataset $\mathcal{D}$ Let $\mathcal{N}$ be the set of potential node-set entity groups, for instance, the primary key attributes in a relational database. Let $\mathcal{N}'$ be the set of all remaining entity groups, for instance, the non-primary key attributes in the schema. We define a *schema graph*[2], $\mathcal{G}_S$ with $\mathcal{N} \cup \mathcal{N}'$ as its nodes. An edge in $\mathcal{G}_S$ indicates a direct relationship between two nodes, such as the one derived from a functional dependency between a primary key and another attribute in a relation, or a foreign key constraint. Note that the nodes of the schema-graph are not actual entities, but entity types, and edges are undirected. In reference to the schema shown in Figure 4(a), two such nodes of the schema graph are `Author.id` and `Conference.year`, but not actual values of author-ids or conference years, respectively. Figure 3(a) illustrates the corresponding schema graph.

The technicalities of schema graph construction depend on the type of data source. For instance, in text data, the entity groups and their inter-relations are derived using the NLP techniques described earlier. For simplicity and want of space, we restrict the explanation to relational datasets.

A data graph that may be derived from $\mathcal{D}$ is described by a path p from N_1 to N_2 in $\mathcal{G}_S$ such that $N_1, N_2 \in \mathcal{N}$. Note that

[2] The *schema graph* is a meta-graph on entity types in $\mathcal{D}$, not be confused with the *data graphs* extracted from $\mathcal{D}$.



(a) Illustration of a *schema graph* over the author-publication schema described in Figure 4(a). The co-authorship relationship is described by the path $9 \rightarrow 10 \rightarrow 11 \rightarrow 10 \rightarrow 9$.

```
Nodes: Select Author.id from Author;
Edges: Select A1.id, A2.id from Author A1,
Author A2, AuthPub AP1, AuthPub AP2 WHERE
AP1.aid = A1.aid AND AP2.aid = A2.aid AND
AP1.pid = AP2.pid;
```

(b) Co-authorship data graph extraction query.

Figure 3: Schema Graph and graph extraction query.

p may be a loop or traverse same intermediate or end nodes as well as edges more than once. For the same terminal node, i.e., $N_1 = N_2$, we obtain homogeneous graphs, for $N_1 \neq N_2$, bipartite graphs. Further extensions of this may involve finding a tree instead of a path between three terminal nodes instead of two, and so on. An enumeration of all such paths in $\mathcal{G}_S$ gives us the enumeration of all graphs in $\mathcal{D}$. Note that the number of graphs obtainable is unbounded in general. For instance, while multiple traversals of the same loop will yield a syntactically correct graph, it is unlikely represent a meaningful interpretation. We use certain heuristics to trim the number of enumerations, typically those restricted to multiple traversals of the same nodes and edges. A set of enumerations that are further extracted from the dataset is called a *graph specification*. Each graph description in the specification is translated to an extraction code or a query; Figure 3(b) shows the required query to generate a particular data graph from $\mathcal{D}$. Note that the GraphMapper auto-generates and deploys these queries, not the analyst.

Storage and Extraction Optimizations: The graph extraction process uses optimizations similar to multi-query optimization in relational databases. Specifically, it avoids redundant fetching of nodes and join sets. The set of extracted graphs are stored in a compact manner, utilizing overlaps of nodes and edges between different graphs. We store nodes, edges or attribute repeated across a set of graphs only once, along with bitmaps and additional tables for book-keeping, that signify component-graph correspondence.

2.3 Metric Computation and Summarization

Upon enumeration of a set of graphs $\mathcal{S}_G = \{G_1, G_2 \dots\}$, we match these with the set of available metric computation algorithms $\mathcal{A}_{AD} = \{A_1, A_2 \dots\}$. A subset of the cross-product of the two sets, $\mathcal{S}_G \times \mathcal{A}_{AD}$, is called the *run specification*. The described cross product between graph and algorithms is typically large and the number of candidates in a run specification is pruned by matching the suitability of the algorithm to the graph. Some of attributes used for pruning include the graph type, i.e., directness, weightedness, density, etc. After this matching, the algorithms are run and the results are summarized in manner similar to a data cube, i.e., using different aggregation and group-by criteria (further elaborated in Section 3), in order to facilitate easy and quick interpretation by the analyst.

Implementation: The core system is implemented in Java and uses the JUNG library[3] for some of its graph operations. Stanford coreNLP suite [5] is used for text processing tasks. The web front-end uses JavaScript and JQuery for its interactive components, along with Java Servlets processing webserver requests. We currently use node centrality based graph metrics, such as betweenness, degree, closeness, clustering coefficient, page rank, HITS, etc.

3. DEMONSTRATION

In this section, we briefly describe the interactive exploration process of GraphViewer, and discuss the demonstration plan. The user interface allows the analyst to (a) load a non-graph data source from one of the many formats, and view the structure or sample contents of the input; (b) auto-generate a set of graph enumerations, and add/remove to that list; (c) auto-synthesize a specification for specific metric computation or anomaly detection algorithms on various graphs, and edit that specification; (d) explore the set of metrics computed or anomalies detected using a *group-by* or *aggregation* by the graph, entity (node), entity type or the algorithm used; (e) explore specific entities in detail.

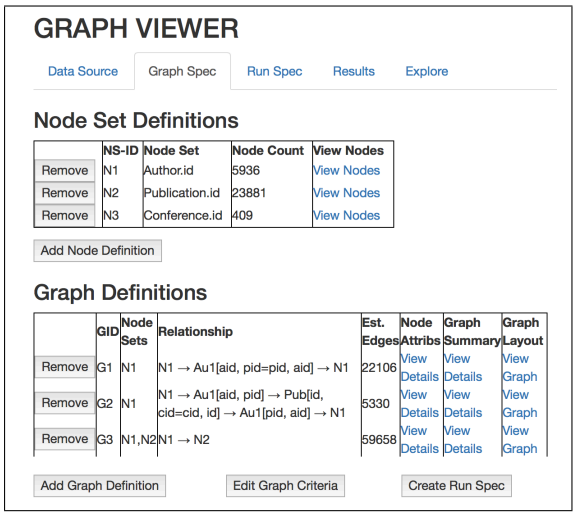
The `Data Source` tab lets a user specify a data source type and import details. Upon successful loading, an appropriate summary of the data structure (relational schema, XML hierarchy, etc.) and a sample of the data is displayed. Clicking `Generate Graph Spec`, moves the focus on to the `Graph Spec` tab, as shown in Figure 4(b). It displays the selection of node-sets and graph definitions, respectively, as computed by GraphMapper's default rules. The user can edit the rules as well as specific definitions of the node-sets or graphs. Additionally, the `View Graph` option lets the user browse through a physical layout of the graph in an interactive manner with filters and color controls to change the appearance of the graph layout. With the desired graph specification, a user clicks `Generate Run Spec` which extracts the specified graphs and switches context to the next tab.

Figure 4(b) shows a `Run Spec` which contains specification of anomaly detection algorithms to be run on specific graphs and the anomaly selection criteria[4]. The analyst can chose to modify, remove a suggested specification or add a new one. Upon clicking `Execute`, the algorithms are run as per the specifications and once complete, the `Results` tab displays all the computed anomalies or metrics. As shown in Figure 4(c), the user may choose to browse the anomalies by grouping them by the anomalous entity i.e., node, the entity type, graph from which the anomaly was discovered, or the algorithm used. The analyst may also summarize the results using one of these criteria. This helps the analyst to compare different entities in the dataset based on their overall anomalous behavior, or which are the graphs or algorithms that are more useful in finding certain anomalies. Finally, the `Explore` tab (not shown) facilitates the exploration of data in an entity-based manner, i.e., one may specify an entity of interest and only the graphs extraction and analyses around that entity will be performed. This tab also contains features `Find Similar Anomalies` and `Export Graphs`.

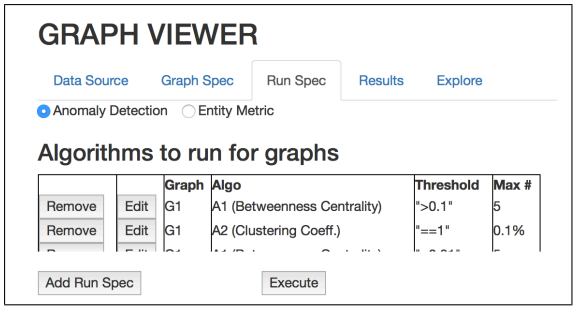
[3] http://jung.sourceforge.net

[4] `RunSpec` and `Results` tabs have separate modes for viewing *anomaly detection* and *metric computation*, respectively.

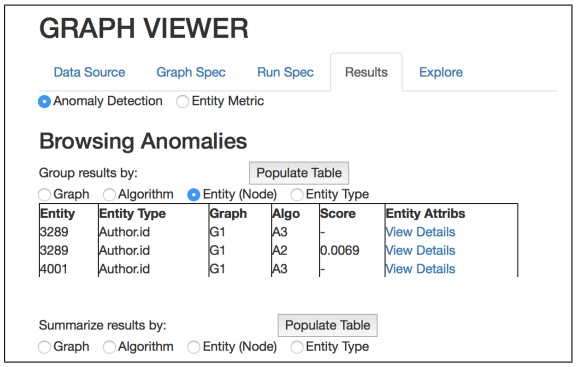
In a case of textual data exploration through Graph Viewer, we analyzed transcripts of the nine GOP and six Democratic Party presidential debates held between August 6, 2015 and February 13, 2016. GraphMapper was able to generate various graphs based on entity-sets labeled *Person*, *Location*, and *Organization*; it mined relationships between entities using several different criteria such as co-occurrence of two entities in the same sentence or same paragraph, a subject-predicate-object relationship between entities, etc. We ended up with dozens of different graphs for each debate dataset, such as a bi-partite weighted graph between persons and locations, a directed graph of who mentioned whom, a graph reflecting co-occurrence of all entities, to name a few. Curious to find a graph model that reflects the popularity of candidates (from polls) in the debate, we performed an entity-based exploration to conclude that a betweenness centrality and pagerank from a person co-mentions graph, are the most indicative factors. In the the bi-partite graph between persons and locations, we looked at the connectivity of different presidential candidates to a cluster of location entities such as "Syria", "Iran", "Iraq", etc. to gauge the candidates' relative reference to foreign policy topics. We also observed that the disparity between the two major Democratic candidates in terms of links with various entities, reduced from 0.6 to 0.39 (Jaccard) over 3 months.



(a) *Graph specification* describes graph enumerations which can be modified by a user.



(b) The *run spec* matches graphs with algorithms. It is generated automatically and a user can modify it.



(c) Browsing Anomalies.

Figure 4: `GraphMapper`

We believe that the underlying graph enumeration and the ability to slice and dice a multiplicity of metrics on each graph is what enabled us to gain these quick insights into these datasets. Also, the *GraphViewer* framework is easily extensible by plugging additional techniques for detecting entities and relationships from data, as well as performing a wider set of graph analytics. We continue to work on optimizing the graph retrieval, storage, and analysis aspects of the system. Additionally, we are working on extracting temporal graphs from timestamped datasets, as well as, implementing the system in a distributed, scaled-out fashion.

Demonstration Plan: During the conference, the attendees will be given a preview of the system through the demonstration of a prepared anomaly detection and a graph discovery scenario, each. Further, they will use the interface for "graphically" exploring the following datasets: (a) DBLP publications records[5] (database and XML); (b) New York city public records datasets on transportation, health and environment[6] (CSV); (c) Presidential debates for 2016 election[7] (text) (d) an anonymized DNS transactions log database (CSV); (e) the novel, A Tale of Two Cities by Charles Dickens (text); (f) Chicago's crime dataset[8] (JSON).

4. REFERENCES

[1] C. C. Aggarwal. *Data mining: The textbook*, 2015.

[2] L. Akoglu, et al. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 2014.

[3] D. G. Brizan, et al. A. survey of entity resolution and record linkage methodologies. *C. IIMA*, 2015.

[4] D. Lee, et al. Constraints-preserving transformation from XML document type definition to relational schema. In *Conceptual Modeling–ER*. 2000.

[5] C. D. Manning, et al. The Stanford CoreNLP natural language processing toolkit. In *ACL*, 2014.

[6] Y. Perez, et al. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD*, 2015.

[7] J. Pujara, et al. Knowledge graph identification. In *ISWC*, 2013.

[8] E. Rahm et al. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[9] K. Xirogiannopoulos, et al. Graphgen: Exploring interesting graphs in relational data. *VLDB*, 2015.

[5] http://dblp.uni-trier.de/xml/

[6] https://nycopendata.socrata.com

[7] http://www.presidency.ucsb.edu/debates.php

[8] https://catalog.data.gov/dataset/crimes-2001-to-present-398a4