KOJAPH: Visual Definition and Exploration of Patterns in Graph Databases

Walter Didimo(✉), Francesco Giacchè, and Fabrizio Montecchiani

Università Degli Studi di Perugia, Perugia, Italy
{walter.didimo,fabrizio.montecchiani}@unipg.it, fgiacc@gmail.com

Abstract. We present KOJAPH, a new system for the visual definition and exploration of patterns in graph databases. It offers an expressive visual language integrated in a simple user interface, to define complex patterns as a combination of topological properties and node/edge attribute properties. Users can also interact with the query results and visually explore the graph incrementally, starting from such results. From the application perspective, KOJAPH has been designed to run on top of every desired graph database management system (GDBMS). As a proof of concept, we integrated it with NEO4J, the most popular GDBMS.

1 Introduction

Graph databases are of growing interest in the many application domains where data are conveniently modeled as graphs [1,5]. In contrast to relational databases, a graph database allows users to directly execute graph-like queries, such as finding pairs of nodes that are connected by a "short" path, finding the common neighbors of two specific nodes, finding cycles or cliques including a desired subset of nodes, and so on. On graph-structured data, this approach leads to a more efficient extraction process, which does not require the expensive join operations necessary on a relational database. For a survey on query languages for graph databases see, e.g., [10].

Besides the use of new paradigms for storing graph-structured data and retrieving information from them, a complementary line of research focuses on the design of visual languages and systems that allow users to easily define queries for extracting desired information. These tools are particularly valuable when the user wants to look for specific patterns in the data set without learning the native query language of the database management system. Within this research line, GRAPHITE is a system that allows users to visually construct graph patterns and to subsequently apply exact or approximate pattern matching algorithms to extract the results [4]. However, this system has several limitations: (*i*) it is not designed to directly work on top of widely used graph database technologies; (*ii*) its visual language is quite simple and does not allow for the creation of sophisticated patterns; (*iii*) the interaction of the

Research supported in part by the MIUR project AMANDA "Algorithmics for MAssive and Networked DAta", prot. 2012C4E3KT_001.

E. Di Giacomo and A. Lubiw (Eds.): GD 2015, LNCS 9411, pp. 272–278, 2015.
DOI: 10.1007/978-3-319-27261-0_23

user with the presented results is rather limited. Conversely, QGRAPH is a more complete visual query language for graphs [3]. It is used in the knowledge discovery system PROXIMITY (https://kdl.cs.umass.edu/display/public/Proximity), which allows users to easily understand and modify large relational data sets. However, PROXIMITY has it own data-structures to efficiently store and retrieve relational data, and it is not conceived to interact with modern and widely used GDBMS, like NEO4J, TITAN, and so on. There are also other technologies in this field, that are however not conceived for graph-structured data. Among them, the system POLARIS offers a visual query language for describing a wide range of table-based graphical presentations of data, extracted from multidimensional relational databases [9]. IMMENS is a system for real-time visual querying of big data [8]. System architectures and algorithms that are mainly designed to efficiently interleave visual query formulation and graph query processing are also described in the literature [2,7].

This paper presents KOJAPH, a new system for the visual definition and exploration of patterns in graph databases. KOJAPH has the following main features: (*a*) It offers an expressive visual language integrated in an intuitive user interface, to define complex patterns as a combination of topological properties and node/edge attribute properties. (*b*) Users can interact with the query results, which can be used as seeds for subsequent incremental explorations of the graph. (*c*) It is designed to work with any graph database management system (GDBMS) and the user can access it with a common Web browser.

Section 2 describes the KOJAPH visual language and user interface. Section 3 presents the system architecture and its integration with NEO4J (http://neo4j.com/). Future work is discussed in Sect. 4. A demo version of KOJAPH is available at: http://mozart.diei.unipg.it:8080/Kojaph/.

2 Visual Language and User Interface

Denote by G the entire graph stored in the graph database. The user can construct a desired pattern to be matched in G, using a graphical interface that integrates all the logical elements of the visual query language. At a high-level view, a pattern P consists of a pair $\langle G_P, R_P \rangle$ of specifications, where $G_P = (V_P, E_P)$ is a graph that defines the topological structure of P, and R_P is a set of rules on the nodes and the edges of G_P. An edge $e \in E_P$ does not necessarily correspond to a single edge of G, but it can also correspond to a path whose length is within a desired range. This correspondence can be established by a specific type of rules of R_P, which we call *path constraints*. The other types of rules in R_P are used to describe desired properties for node/edge attributes of G_P; these properties can then be combined with logical operators AND, OR, NOT to form a binary tree, called the *properties tree*.

Structure of the Interface. The first time the user accesses the graphical interface, the system automatically retrieves from the database all types of node and edge attributes. The interface is shown in Fig. 1. The left-side panel, called

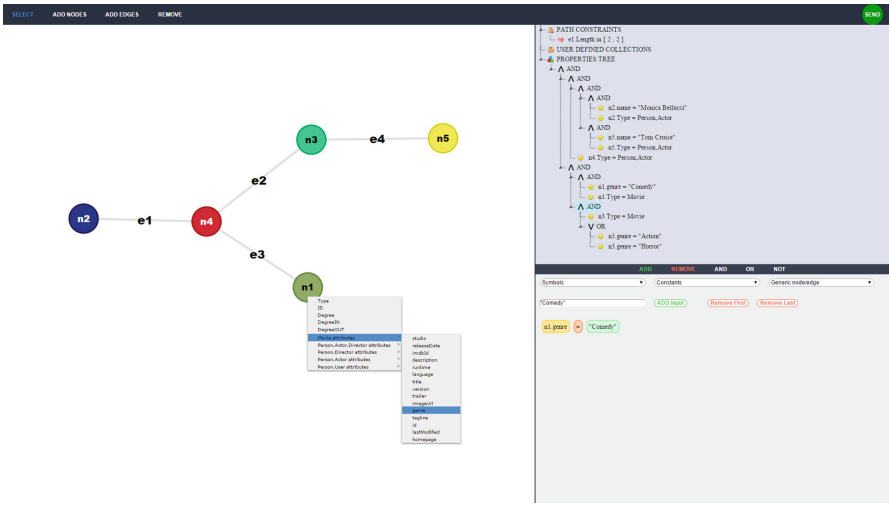

Fig. 1. The interface of KOJAPH for graph pattern definition.

graph-ed panel, is a canvas for editing G_P. Similarly to a common graph editor, it allows users to add, remove, select, or move nodes and edges; multiple edges and self-loops are allowed and are automatically drawn avoiding overlaps. A self-loop on a node v can be useful, for instance, to refer to a cycle that passes through v. Each time a new node is added, it is automatically assigned a unique label (identifier). The right-side of the interface is used to define the rules of R_P and it is further subdivided into a bottom panel and a top panel. The bottom panel, called the *prop-def panel*, is used to define desired properties of node/edge attributes; the top panel is used to group and combine them, so to form the properties tree above mentioned. The top panel also reports the path constraints and user-defined *collections* of attribute values, which can be used to construct properties.

Attributes. Each node or edge of G_P has an associated list of attributes. Generic attributes include the *type* of the element in the database (e.g., in a movie database, the type of a node can be "movie", "actor", "director", etc.), its *identifier* in the database, its *degree* (in/out-degree or total degree) if the element is a node, and its *direction* if the element is an edge. Specific attributes depend on the data modeled by the graph: for instance, in a movie database, a node of type "actor" might have attributes like "name","birthday", or "biography"; an edge connecting an actor to a movie might have an attribute "acts-in", whose value is the character interpreted by the actor in the movie. For any edge e of G_P, there are also additional attributes that are related to path constraints, which can be used when e corresponds to a path Π_e instead of a single edge. In particular, there are attributes whose values define the minimum/maximum length of Π_e and attributes that refer to the nodes and edges in Π_e. For example, the *sub-node properties* of e can be used to define rules on all nodes, any node, a

single node, or no nodes of Π_e. Analogously, the attribute *sub-edge properties* of e is used to define rules on all edges, any edge, a single edge, or no edges of Π_e.

Property Definition. To define a property the user must switch to the "select" mode on the graph-ed panel, so to avoid modifications of G_P during the definition process. The property is defined as follows: (*i*) The user can see the list of attributes for a node/edge of G_P by clicking with the mouse right-button on it. (*ii*) A selected attribute is added to the prop-def panel. (*iii*) Attributes can be correlated to specific user-input or constant values, or subset of values like the above mentioned collections; they can also be combined together to form complex expressions, using a variety of operators, accessible from the list of "symbols" in the graphical interface. These symbols consist of *comparison*, *inclusion*, and *mathematical* operators, including parenthesis to define association rules and arrays. It is also possible to associate an attribute value with a regular expression (using the $\sim=$ operator). Each time the user adds an element (attribute, value, or symbol) to the property, it is visually appended to the right of the previous ones; the user can freely reorder the elements by means of drag-and-drop operations.

Properties Tree. Once a property is defined, the user can add it to the properties tree. The system will just append a new property at the root level. However, in a valid tree, each property must be a leaf node. To this aim, the user can add to the tree a suitable number of internal *operator nodes*, each corresponding to a boolean operator, and then he/she can make a property as a child of an operator node, by means of a drag-and-drop operation. When the user sends the query to the GDBMS, the system first checks the validity of the tree and, in the positive case, it translates the query constructed with the visual language into a query defined with the native language of the GDBMS.

Example. Figure 1 shows a pattern $\langle G_P, R_P \rangle$ defined on a movie database where persons (actors, directors, etc.) are connected to movies. The pattern G_P consists of 5 nodes and 4 edges. The properties tree and the path constraints on the right-hand side describe the set of rules. G_P describes an actor, node n4, such that: (*i*) n4 acted in a movie together with "Monica Bellucci", the node n2; this is expressed by a path constraint that requires that e1 is a path of length 2. (*ii*) n4 acted in at least one "Comedy" (node n1) and at least one "Horror" or "Action" (node n3), together with "Tom Cruise", who is node n5. The rules of R_P are summarized by the following expression:

```
(e1.Length in [2,2]) AND
(n4.Type=Person.Actor AND
 (n2.Type=Person.Actor AND n2.name="Monica Bellucci") AND
 (n5.Type=Person.Actor AND n5.name="Tom Cruise"))
 AND
((n1.Type=Movie AND n1.genre="Comedy") AND
 (n3.Type=Movie AND (n3.genre="Horror" OR n3.genre="Action")))
```

Presentation and Exploration of the Results. The system shows the results of the query in a different window. Multiple results that match the pattern are

listed in a pop-up menú and the user can display them one by one. Each result is viewed as a graph isomorphic to G_P, with the same node/edge labels and colors as in G_P, so to preserve the user's mental map. Edges corresponding to paths are depicted as dashed segments. An important option is the possibility of displaying all the results as a unique graph: KOJAPH merges all of them without duplicating elements. Graphs are automatically drawn by the force-directed algorithm of the *D3.js* library (http://d3js.org/); a post-processing procedure is applied to represent multiple edges as non-overlapping curves. Figure 2 shows the results of the query for the pattern of Fig. 1, drawn as a unique graph.

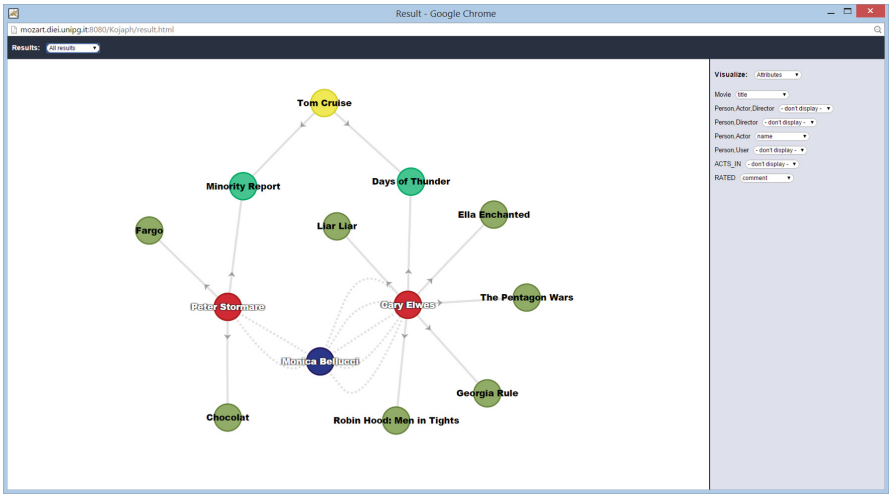

Fig. 2. The results of the query for the pattern of Fig. 1, shown as a unique graph.

The user can interact with the result in different ways. He/she can change at any time the kind of information displayed as node and edge labels. In the figure, actors are labeled with their names and movies with their titles, while edges are not labeled. A mouse-over interaction on an element will show all attribute values of that element. Zooming in/out (using the mouse wheel) and node adjustments are possible. More importantly, the user can explore the displayed result, incrementally enriching it with additional information. Double clicking on a node v, the system visualizes the neighbors of v not already in the drawing. Double clicking on a dashed edge e, the system replaces e with its associated path. To keep clear the original pattern throughout the exploration, new nodes that enter in the drawing have smaller size than the pattern nodes, and the new edges that enter in the drawing for a double click on a node v gets the color of v.

3 System Architecture and Integration with NEO4J

KOJAPH is a client-server Web application. The user interactive graphical environment is implemented using JavaScript, JQuery, and AJAX. From the server-side,

client requests are handled by a Java servlet. In order to integrate the KOJAPH server with any GDBMS, we defined a Java abstract class, named DBMSInterface, that describes few methods to interact with the GDBMS, including methods for automatically retrieving node/edge attributes and a method to translate any query constructed with the visual language of KOJAPH into a query defined with the language of the GDBMS. These methods exchange data (input parameters and output values) in the JSON lightweight data-interchange format. A specific implementation of DBMSInterface must be provided for each specific GDBMS to be used.

As a proof of concept, we integrated KOJAPH to the popular NEO4J GDBMS. We implemented the methods of class DBMSInterface using the NEO4J query language *Cypher*, and we exploited the REST API of NEO4J for sending the queries to the GDBMS. The robustness of the integrated system and the correctness of the query answers have been tested on three databases. One is a movie/actor network of 63,042 nodes and 106,651 edges (http://neo4j.com/developer/example-data/); another is a "food network" with approximately the same number of nodes but many more edges, namely 626,641 (http://blog.bruggen.com/2013/12/fascinating-food-networks-in-neo4j.html). The third database is a portion of the co-authorship network in Computer Science extracted from DBLP (http://dblp.uni-trier.de/); it consists of 198,830 nodes and 207,793 edges. The time required to translate a KOJAPH query to the corresponding Cypher query is negligible; hence the response time of the system to a visual query is mainly related to the performances of the graph pattern matching algorithms applied by the GDBMS. Recall that the graph pattern matching problem is NP-hard [6], but the specification of node/edge attribute properties in addition to topological requirements may strongly reduce the search space and consequently greatly improve the GDBMS performances. Furthermore, path constraints in KOJAPH are translated into a pre-processing filtering of the possible results, which often leads to a dramatic reduction of the response time. The query in the example of Fig. 2 took less than 1 s under an Ubuntu Linux OS, within a VMware virtual machine with a 4 vCPU processor and 16 GB RAM.

4 Future Work

In the near future we plan to: (*i*) equip KOJAPH with more functionalities to visualize and explore the results; (*ii*) evaluate the usability of KOJAPH versus similar systems (e.g., PROXIMITY); (*iii*) testing KOJAPH on different GDBMS, other than Neo4J.

References

1. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008)
2. Bhowmick, S.S., Choi, B., Zhou, S.: VOGUE: towards A visual interaction-aware graph query processing framework. In: CIDR 2013 (2013)

3. Blau, H., Immerman, N., Jensen, D.: A visual language for querying and updating graphs. Technical report UM-CS-2002-037, University of Massachusetts Amherst, Computer Science Department
4. Chau, D.H., Faloutsos, C., Tong, H., Hong, J.I., Gallagher, B., Eliassi-Rad, T.: GRAPHITE: a visual query system for large graphs. In: ICDM 2008, pp. 963–966. IEEE (2008)
5. Dominguez-Sal, D., et al.: Survey of graph database performance on the HPC scalable graph analysis benchmark. In: Shen, H.T., et al. (eds.) WAIM 2010. LNCS, vol. 6185, pp. 37–48. Springer, Heidelberg (2010)
6. Gallagher, B.: Matching structure and semantics: a survey on graph-based pattern matching. Artif. Intell. **6**, 45–53 (2006)
7. Hung, H.H., Bhowmick, S.S., Truong, B.Q., Choi, B., Zhou, S.: QUBLE: towards blending interactive visual subgraph search queries on large networks. VLDB J. **23**(3), 401–426 (2014)
8. Liu, Z., Jiang, B., Heer, J.: imMens: Real-time visual querying of big data. Comput. Graph. Forum **32**(3), 421–430 (2013)
9. Stolte, C., Tang, D., Hanrahan, P.: Polaris: a system for query, analysis, and visualization of multidimensional databases. Commun. ACM **51**(11), 75–84 (2008)
10. Wood, P.T.: Query languages for graph databases. SIGMOD Record **41**(1), 50–60 (2012)