G

Graph Path Navigation

Marcelo Arenas[1], Pablo Barceló[2], and Leonid Libkin[3]
[1]Pontificia Universidad Católica de Chile, Santiago, Chile
[2]Universidad de Chile, Santiago, Chile
[3]School of Informatics, University of Edinburgh, Edinburgh, UK

Synonyms

Navigational queries; Path queries; Regular path queries

Definitions

Navigational query languages for graph databases allow to recursively traverse the edges of a graph while checking for the existence of a path that satisfies certain regular conditions. The basic building block of such languages is the class of *regular path queries* (RPQs), which are expressions that compute the pairs of nodes that are linked by a path whose label satisfies a regular expression. RPQs are often extended with features that turn them more flexible for practical applications, e.g., with the ability to traverse edges in the backward direction (RPQs with *inverses*) or to express arbitrary patterns over the data (*conjunctive* RPQs).

Overview

Graph Databases

Graph databases provide a natural encoding of many types of data where one needs to deal with objects and relationships between them. An object is represented as a node, and a relationship between two objects is represented as an edge, where labels are assigned to these edges to indicate what types of relationships they represent. This interpretation of data often arises in situations where one needs to *navigate* in the graph and reason about the *paths* in it. To talk about such navigation, we can abstract graph databases as edge-labeled graphs. Formally, assuming that $\mathbf{L}$ is a fixed infinite set of edge labels, we have the following formal definition of a graph database.

Definition 1 A graph database G is a pair (N, E), where:

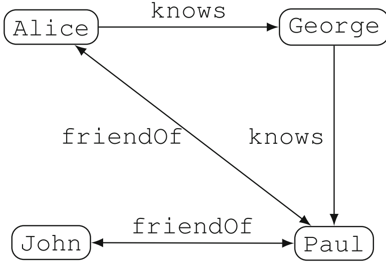
1. N is a finite set of *nodes*; and
2. E is a finite set of *labeled edges*, that is, a finite subset of $V \times \mathbf{L} \times V$. □

Example 1 Figure 1 shows a simple graph database $G = (N, E)$ storing information about people and their relationships. Here N has four nodes:

$$\{\texttt{Alice}, \texttt{George}, \texttt{John}, \texttt{Paul}\}$$

and E contains six labeled edges:

© Springer International Publishing AG 2018
S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_214-1



Graph Path Navigation, Fig. 1 A graph database

$\{$(Alice, knows, George),
(George, knows, Paul),
(Alice, friendOf, Paul),
(Paul, friendOf, Alice),
(John, friendOf, Paul),
(Paul, friendOf, John)$\}$. □

Path Queries

The simplest type of query on a graph database extracts nodes according to the relationship between them. We assume that $\mathbf{V}$ is an infinite set of variables that is disjoint with $\mathbf{L}$. Then given $x, y \in \mathbf{V}$ and $\ell \in \mathbf{L}$, the following is a basic query

$$x \xrightarrow{\ell} y$$

that asks for the pairs of nodes in a graph database connected by an edge-labeled ℓ. The semantics of a such a query Q on a graph database $G = (N, E)$, denoted by $[\![Q]\!]_G$, is the set of *matchings* $h : \{x, y\} \to N$ such that $(h(x), \ell, h(y)) \in E$.

Example 2 Evaluating the basic query $Q_1 = x \xrightarrow{\text{knows}} y$ over the graph database G in Fig. 1 produces as a result:

	x	y
h_1	Alice	George
h_2	George	Paul

As depicted in the table, we have that $[\![Q_1]\!]_G = \{h_1, h_2\}$, where $h_1(x) =$ Alice, $h_1(y) =$ George, $h_2(x) =$ George, and $h_2(y) =$ Paul. □

While basic queries like the one above allow for extracting valuable information from a graph database, it is often useful to provide more flexible querying mechanisms that allow to *navigate* the topology of the data. One example of such a query is to find all friends-of-a-friend of some person in a social network. We are not only interested in immediate acquaintances of a person but also in people he/she might know through other people, namely, his/her friends-of-a-friend, their friends, and so on. For instance, in the graph in Fig. 1, we may want to discover that Alice is connected with John through a common friend.

Queries such as the one above are called *path queries*, since they require navigation in a graph using paths of arbitrary lengths. Path queries have found applications in areas such as the Semantic Web (Alkhateeb et al. 2009; Pérez et al. 2010; Paths 2009), provenance (Holland et al. 2008), and route-finding systems (Barrett et al. 2000), among others. In this entry, we provide a brief overview of path queries and some of their extensions; for detailed surveys, the reader is referred to Barceló (2013) and Angles et al. (2017).

Regular Path Queries

A *path* p in a graph database G is a sequence e_1, e_2, ..., e_n of edges of the form $e_i = (a_i, \ell_i, b_i)$ such that $n \geq 0$ and $b_j = a_{j+1}$ for every $j \in \{1, \ldots, n-1\}$. That is, each edge in a path starts in the node where the previous edge ends. If $n = 0$ we refer to p as the empty path. We write *labels*(p) for the string of labels formed by the sequence of edges in p, that is, *labels*$(p) = \varepsilon$ if $n = 0$ and *labels*$(p) = \ell_1 \ell_2 \cdots \ell_n$ otherwise. Finally, we say that a_1 is the starting node of p and b_n is the ending node of p.

Example 3 $p =$ (Alice, knows, George), (George, knows, Paul) is a path in the graph database in Fig. 1. This path p is a path of length 2, its starting node is Alice, its ending node is Paul, and *labels*$(p) =$ knows knows. □

Definition 2 A *regular path query* (RPQ) Q is an expression of the form

$$x \xrightarrow{r} y,$$

where $x, y \in \mathbf{V}$ and r is a regular expression over $\mathbf{L}$. The evaluation of Q over a graph database $G = (N, E)$, denoted by $[\![Q]\!]_G$, is the set of matchings $h : \{x, y\} \rightarrow N$ for which there exists a path p in G whose starting node is $h(x)$, ending node is $h(y)$, and *labels*(p) is a string in the regular language defined by r. □

Thus, an RPQ $x \xrightarrow{r} y$ asks for the pair of nodes in a graph database that are the endpoints of a path whose labels conform to the regular expression r.

Example 4 Assume that Q_2 is the RPQ $x \xrightarrow{\texttt{knows}^+} y$, where $\texttt{knows}^+$ is a regular expression that defines the language of nonempty strings of the form knows knows ⋯ knows. The following is the result of evaluating Q_2 over the graph database G shown in Fig. 1:

	x	y
h_1	Alice	George
h_2	George	Paul
h_3	Alice	Paul

In this case, $h_3 \in [\![Q_2]\!]_G$ since for the path $p =$ (Alice, knows, George), (George, knows, Paul) in G, we have that the starting node of p is Alice, the ending node of p is Paul, *labels*$(p) =$ knows knows, and knows knows is a string in the regular language defined by $\texttt{knows}^+$. □

Notice that Definition 2 does not impose the restriction that x and y be distinct variables, so an RPQ of the form $x \xrightarrow{r} x$ is valid, and its semantics is well defined. Such an RPQ asks for cycles whose labels conform to the regular expression r.

Constant values are usually allowed in RPQs. For the sake of readability, we do not consider them separately, as the syntax and semantics of an RPQ with constants are defined exactly in the same way as in Definition 2. We only give an example of such a query to illustrate how constants are used in RPQs.

Example 5 Assume that Q_3 is the RPQ $\texttt{Alice} \xrightarrow{\texttt{knows}^+} z$, where z is a variable and Alice is a constant representing a node in a graph. The following is the result of evaluating Q_3 over the graph database shown in Fig. 1:

	z
h_4	George
h_5	Paul

□

Regular Path Queries with Inverse

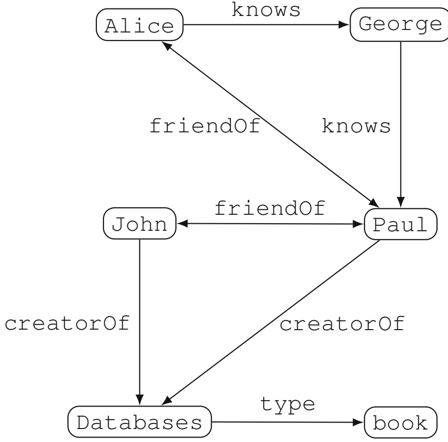
Let G be the graph database shown in Fig. 2, which is an extension of our running example with information about creations. Assume that we want to retrieve pairs of people that are co-creators of the same artifact. We can think of this query as a path in G: starting from a person, we traverse an edge with label creatorOf to reach an artifact, and then from there we traverse again an edge with label creatorOf to reach a person, who is then another creator of the artifact. But there is an issue with such a path as the second edge with label creatorOf has to be traversed in the opposite direction.

To overcome this problem, RPQs are often extended with the ability to traverse edges in both directions, which gives rise to the notion of RPQ with *inverses* (Calvanese et al. 2000). To define this, let $\mathbf{L}^{\pm}$ be the extension of $\mathbf{L}$ with the symbol ℓ^-, for each $\ell \in \mathbf{L}$. Moreover, for a graph database $G = (N, E)$, let $G^{\pm}$ be the extension of G by adding the edges (b, ℓ^-, a), for each $(a, \ell, b) \in E$.

Definition 3 A *regular path query with inverses* (2RPQ) Q is an expression of the form

$$x \xrightarrow{r} y,$$

where $x, y \in \mathbf{V}$ and r is a regular expression over $\mathbf{L}^{\pm}$. The evaluation of Q over a graph database G, denoted by $[\![Q]\!]_G$, is defined as $[\![Q]\!]_{G^{\pm}}$. □



Graph Path Navigation, Fig. 2 A graph database including information about creations

In this definition, $[\![Q]\!]_{G^\pm}$ is well defined as Q is an RPQ over $G^\pm$.

Example 6 The following 2RPQ Q_4 can be used to retrieve pairs of people that are co-creators of the same artifact in the graph database shown in Fig. 2:

$$x \xrightarrow{\texttt{creatorOf}\ \texttt{creatorOf}^-} y$$

In particular, if h_6 is a matching such that $h_6(x) = \texttt{John}$ and $h_6(y) = \texttt{Paul}$, then we have that $h_6 \in [\![Q]\!]_G$ as $p =$ (John, creatorOf, Databases), (Databases, creatorOf$^-$, Paul) is a path in $G^\pm$ such that *labels*(p) is a string that conforms to the regular expression creatorOf creatorOf$^-$.

Notice that we can retrieve pairs of people who are connected by a co-creation path of arbitrary length by using the following 2RPQ:

$$x \xrightarrow{(\texttt{creatorOf}\ \texttt{creatorOf}^-)^+} y$$

□

Conjunctive Regular Path Queries

A more expressive form of graph database queries is obtained by viewing RPQs as basic building blocks and then defining conjunctive queries over them. Recall that conjunctive queries, over relational databases, are obtained by closing relational atoms under conjunction and existential quantification (or, equivalently, under selection, projection, and join operations of relational algebra). When the same operations are applied to RPQs, they give rise to the notion of conjunctive RPQs.

Definition 4 A *conjunctive regular path query* (CRPQ) is an expression of the form

$$ans(u_1, \ldots, u_k) \text{ :- } \bigwedge_{i=1}^{n} v_i \xrightarrow{r_i} w_i \qquad (1)$$

where each $v_i \xrightarrow{r_i} w_i$, for $i \in \{1, \ldots, n\}$, is an RPQ that may include constants and $u_1, \ldots, u_k$ is a sequence of pairwise distinct variables among those that occur as v_is and w_is. □

The left-hand side of (1) is called the head of the query, and its right-hand side is called the body. Variables in the body of (1) need not be pairwise distinct; in fact, sharing variables allows joining results of the RPQs in the body of the query, which is a fundamental feature of CRPQs. As in the case of relational conjunctive queries, variables mentioned in the body of (1) but not in the head (i.e., those that are not in the answer) are assumed to be existentially quantified (in other words, projected away). Finally, if $k = 0$, then (1) is called a Boolean CRPQ, as its evaluation results in either a set containing a matching with empty domain or an empty set of matchings, which represent the values true and false, respectively.

The semantics of (1) is defined as follows. Assume that $G = (N, E)$ is a graph database, and let $x_1, \ldots, x_m$ be the variables occurring in the body of (1). Then a matching $h : \{x_1, \ldots, x_m\} \to N$ is said to satisfy the body of (1) if for every $i \in \{1, \ldots, n\}$:

$$h|_{\{v_i, w_i\} \cap \mathbf{V}} \in [\![v_i \xrightarrow{r_i} w_i]\!]_G,$$

where $h|_{\{v_i, w_i\} \cap \mathbf{V}}$ is the restriction of matching h to the domain $\{v_i, w_i\} \cap \mathbf{V}$. A matching $g : \{u_1, \ldots, u_k\} \to N$ is an *answer* to (1) over G if

there is an extension $h : \{x_1, \ldots, x_m\} \to N$ of g that satisfies the body of (1). The evaluation of a CRPQ Q over G, written $[\![Q]\!]_G$, is the set of answers to Q over G.

Example 7 Let $G = (N, E)$ be the graph database in Fig. 2. Then the following CRPQ Q_5 can be used to retrieve the pairs of people that are co-authors of the same book:

$$ans(x, y) \text{ :- } x \xrightarrow{\texttt{creatorOf}} z \wedge$$
$$y \xrightarrow{\texttt{creatorOf}} z \wedge$$
$$z \xrightarrow{\texttt{type}} \texttt{book}$$

Let $g : \{x, y\} \to N$ and $h : \{x, y, z\} \to N$ be matchings such that $g(x) = \texttt{John}$, $g(y) = \texttt{Paul}$, $h(x) = \texttt{John}$, $h(y) = \texttt{Paul}$, and $h(z) = \texttt{Databases}$. We have that $g \in [\![Q_5]\!]_G$ since $g = h|_{\{x,y\}}$, $h|_{\{x,z\}} \in [\![x \xrightarrow{\texttt{creatorOf}} z]\!]_G$, $h|_{\{y,z\}} \in [\![y \xrightarrow{\texttt{creatorOf}} z]\!]_G$, and $h|_{\{z\}} \in [\![z \xrightarrow{\texttt{type}} \texttt{book}]\!]_G$. □

Key Research Findings

The Complexity of Evaluating Path Queries

We now consider the cost of evaluating a query in a graph query language $\mathscr{L}$. More precisely, the complexity of the problem $\mathscr{L}$-EVAL is defined as follows: Given a graph database G, a query $Q \in \mathscr{L}$, and a matching h, does h belong to the evaluation of Q over G? (in symbols, is $h \in [\![Q]\!]_G$?). Notice that the input to $\mathscr{L}$-EVAL consists of a graph database G, a query Q and a matching h, and, thus, it measures the *combined complexity* of the evaluation problem. Since the size of h is bounded by the size of Q and the size of G, it suffices to measure the combined complexity in terms of $|Q|$ and $|G|$, the sizes of Q and G.

In practice, queries are usually much smaller than graph databases, so one is also interested in the *data complexity* of the evaluation problem, which is measured only in terms of the size of the graph database (i.e., the query is assumed to be fixed).

The combined and data complexities of RPQ-EVAL, 2RPQ-EVAL, and CRPQ-EVAL are shown in Fig. 3.

When we state that data complexity is NLOGSPACE-complete, we mean that for each fixed query Q in those language, the evaluation problem can be solved in NLOGSPACE, and there are some queries – in fact, RPQs – for which it is NLOGSPACE-hard.

We now outline the key ideas behind the $O(|G| \cdot |Q|)$ algorithm as well as NLOGSPACE-completeness and NP-completeness.

Given a database graph $G = (N, E)$, a 2RPQ $Q = x \xrightarrow{r} y$, and a matching $h : \{x, y\} \to N$, we show that 2RPQ-EVAL can be solved in time $O(|G| \cdot |Q|)$. The idea is from Mendelzon and Wood (1995). Assume that $h(x) = a$ and $h(y) = b$. First, we compute $G^{\pm}$ from G in time $O(|G|)$, and then we compute in time $O(|Q|)$, a nondeterministic finite automaton with ε-transitions (ε-NFA) A_r that defines the same regular language as r. Let $G^{\pm}(a, b)$ be the NFA obtained from $G^{\pm}$ by setting its initial and final states to be a and b, respectively. We know that $h \in [\![Q]\!]_G$ if and only if $h \in [\![Q]\!]_{G^{\pm}}$, and the latter is equivalent to checking whether there exists a word accepted by both $G^{\pm}(a, b)$ and A_r. In turn, this is equivalent to checking the product of $G^{\pm}(a, b)$ and A_r for nonemptiness, which can be done in linear time in the size of the product automaton, i.e., in time $O(|G^{\pm}| \cdot |A_r|)$. The whole process takes time $O(|G| + |Q| + |G^{\pm}| \cdot |A_r|)$, that is, $O(|G| \cdot |Q|)$.

If the query Q is fixed, the same algorithm can be performed in NLOGSPACE. Indeed, we just need to compute reachability in the product automaton, and reachability in graphs is computable in NLOGSPACE. Of course the whole product requires more than logarithmic space, but it need not be built, as reachability can be checked by using standard on the fly techniques. Moreover, reachability in graphs is known to be NLOGSPACE-complete, so even RPQ query evaluation can be NLOGSPACE-complete in data complexity.

Graph Path Navigation, Fig. 3 The complexity of the evaluation problem for path queries

	Combined complexity	Data complexity
RPQ-EVAL	$O(\vert G\vert \cdot \vert Q\vert)$	NLOGSPACE-complete
2RPQ-EVAL	$O(\vert G\vert \cdot \vert Q\vert)$	NLOGSPACE-complete
CRPQ-EVAL	NP-complete	NLOGSPACE-complete

Finally, to see that CRPQs can be evaluated in NP in combined complexity, it just suffices to guess the values of all the v_is and w_js and then check in polynomial time that all the RPQs from (1) hold. And since the evaluation problem for relational conjunctive queries over graphs is NP-hard in data complexity, so is the evaluation problem for the more expressive CRPQs.

Simple Path Semantics

The evaluation of RPQs as defined here is based on a semantics of arbitrary paths. It has been argued, on the other hand, that for some applications it is more reasonable to apply a semantics based on *simple paths* only, i.e., those without repeating nodes. Under such a semantics, an RPQ $x \xrightarrow{r} y$ computes the pairs of nodes that are linked by a *simple path* whose label satisfies r. However, evaluation of RPQs under the simple path semantics becomes NP-complete even in data complexity (Mendelzon and Wood 1995).

While this casts doubt on its practical applicability, a closely related semantics was implemented in a practical graph DBMS, namely, Neo4j (Robinson et al. 2013). There, paths in graph patterns can only be matched by paths that do not have repeated *edges*. While still NP-complete in data complexity in the worst case, in practice this semantics could behave well, as worst-case scenarios do not frequently occur in real life.

Extensions

(C)RPQs have been extended with several features. One extension is a *nesting* operator that allows one to perform existential tests over the nodes of a path, similarly to what is done in the XML navigational language XPath (Pérez et al. 2010). Another extension is the ability to compare paths (Barceló et al. 2012). The idea is that paths are named in an RPQ, like $x \xrightarrow{\pi:r} y$, and then the name π can be used in the query. Without defining them formally, we give an example:

$$ans(x, y) \text{ :- } x \xrightarrow{p_1:a^*} z \wedge z \xrightarrow{p_2:b^*} y \wedge \text{equal_length}(p_1, p_2)$$

says that there is a path p_1 from x to z labeled with as and a path p_2 from z to y labeled with bs and these paths have equal lengths (expressed by the predicate equal_length(p_1, p_2)). The equal length predicate belongs to the well-known class of regular relations, but the above query says that there is a path from x to y whose label is of the form $a^n b^n$ for some n. This of course is not a regular property, even though we only used regular languages and relations in the query. Such extended CRPQs still behave well: their data complexity remains in NLOGSPACE, and their combined complexity goes up to PSPACE, which is exactly the combined complexity of relational algebra. However such additions must be handled with care: if we add common predicates on paths such as subsequence or subword, query evaluation becomes completely infeasible or even undecidable. We refer to Barceló et al. (2012) and Barceló (2013) for further discussions.

The last extension is constituted by adding mechanisms for checking regular properties over an extended data model in which each node carries data values from an infinite domain. This models graph databases as they occur in real life, namely, *property graphs*, where nodes and edges can carry tuples of key-value pairs. Extending RPQs and CRPQs requires choosing a proper analog of regular expressions. There are many proposals, but with few exceptions they lead to very high complexity of query evaluation. The one that maintains good complexity is given by *register automata*. Such extensions still have NLOGSPACE data complexity and PSPACE combined complexity and come with different syntactic ways of describing navigation that mixes data

and topology of the graph. We refer the reader to Libkin et al. (2016) for details and the surveys by Barceló (2013) and Angles et al. (2017) for a thorough review of the literature on extensions of CRPQs.

Cross-References

- ▶ Graph Data Management Systems
- ▶ Graph Pattern Matching
- ▶ Graph Query Languages
- ▶ Graph Query Processing

References

Alkhateeb F, Baget J, Euzenat J (2009) Extending SPARQL with regular expression patterns (for querying RDF). J Web Sem 7(2):57–73

Angles R, Arenas M, Barceló P, Hogan A, Reutter JL, Vrgoc D (2017) Foundations of Modern Graph Query Languages. ACM Computing Surveys 50(5)

Barceló P (2013) Querying graph databases. In: Proceedings of the 32nd ACM Symposium on Principles of Database Systems, PODS 2013, pp 175–188

Barceló P, Libkin L, Lin AW, Wood PT (2012) Expressive languages for path queries over graph-structured data. ACM Trans Database Syst 37(4):31

Barrett CL, Jacob R, Marathe MV (2000) Formal-language-constrained path problems. SIAM Journal on Computing 30(3):809–837

Calvanese D, De Giacomo G, Lenzerini M, Vardi MY (2000) Containment of conjunctive regular path queries with inverse. In: Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning, KR 2000, pp 176–185

Holland DA, Braun UJ, Maclean D, Muniswamy-Reddy KK, Seltzer MI (2008) Choosing a data model and query language for provenance. In: 2nd International Provenance and Annotation Workshop (IPAW)

Libkin L, Martens W, Vrgoč D (2016) Querying graphs with data. J ACM 63(2):14

Mendelzon AO, Wood PT (1995) Finding regular simple paths in graph databases. SIAM J Comput 24(6):1235–1258

Paths TFP (2009) Use cases in property paths task force. http://www.w3.org/2009/sparql/wiki/TaskForce:PropertyPaths#Use_Cases

Pérez J, Arenas M, Gutierrez C (2010) nSPARQL: A navigational language for RDF. J Web Sem 8(4): 255–270

Robinson I, Webber J, Eifrem E (2013) Graph Databases, 1st edn. O'Reilly Media