THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Adding regular expressions to graph reachability and pattern queries

Citation for published version:
Fan, W, Li, J, Ma, S, Tang, N & Wu, Y 2011, Adding regular expressions to graph reachability and pattern queries. in *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany.* Institute of Electrical and Electronics Engineers (IEEE), pp. 39-50. https://doi.org/10.1109/ICDE.2011.5767858

Digital Object Identifier (DOI):
10.1109/ICDE.2011.5767858

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 01. Dec. 2020

Adding Regular Expressions to Graph Reachability and Pattern Queries

Wenfei Fan [1,2], Jianzhong Li [2], Shuai Ma [1], Nan Tang [1], Yinghui Wu [1]

[1] *University of Edinburgh, UK*

{wenfei@inf., shuai.ma@, ntang@inf.,y.wu-18@sms.}ed.ac.uk

[2] *Harbin Institute of Technology, China*

lijzh@hit.edu.cn

***Abstract*—It is increasingly common to find graphs in which edges bear different types, indicating a variety of relationships. For such graphs we propose a class of reachability queries and a class of graph patterns, in which an edge is specified with a regular expression of a certain form, expressing the connectivity in a data graph via edges of various types. In addition, we define graph pattern matching based on a revised notion of graph simulation. On graphs in emerging applications such as social networks, we show that these queries are capable of finding more sensible information than their traditional counterparts. Better still, their increased expressive power does not come with extra complexity. Indeed, (1) we investigate their containment and minimization problems, and show that these fundamental problems are in quadratic time for reachability queries and are in cubic time for pattern queries. (2) We develop an algorithm for answering reachability queries, in quadratic time as for their traditional counterpart. (3) We provide two cubic-time algorithms for evaluating graph pattern queries based on extended graph simulation, as opposed to the NP-completeness of graph pattern matching via subgraph isomorphism. (4) The effectiveness, efficiency and scalability of these algorithms are experimentally verified using real-life data and synthetic data.**

I. Introduction

It is increasingly common to find data modeled as graphs in a variety of areas, *e.g.,* computer vision, knowledge discovery, biology, cheminformatics, network traffic, social networks, semantic Web and intelligence analysis. To query data graphs, two classes of queries are being widely used:

(a) Reachability queries, asking whether there exists a path from one node to another [12], [21], [22], [33].

(b) Graph pattern queries, to find all subgraphs of a data graph that are isomorphic to a pattern graph [31], [37] (see [17] for a survey).

In emerging applications such as social networks, edges in a graph are typically "typed", denoting various relationships such as marriage, friendship, co-work, advice, support, exchange and co-membership [23]. In practice one often wants to query the connectivity of a pair of nodes in such a graph via edges of particular types, or to identify graph patterns with edges of certain types, as illustrated by the following real-life example taken from [6].

Example 1: Consider an *Essembly* network service [6], where users post and vote on controversial issues and topics. Each person has attributes such as userid, job, contact information, as well as a list of issues they support or disapprove, denoted by "sp" and "dsp", respectively. There are four types of relationships between a pair of persons: (1) *friends-allies (fa)*, connecting one user to a friend, if she shares the same views on most (more than half) topics her friend votes for; (2) *friends-nemeses (fn)*, from one user to a friend if she disagrees with her friend on most topics; (3) *strangers-allies (sa)*, relates a user to a stranger she agrees with on most topics they vote; and (4) *strangers-nemeses (sn)*, from a user to a stranger with whom she disagrees on most topics they both vote.

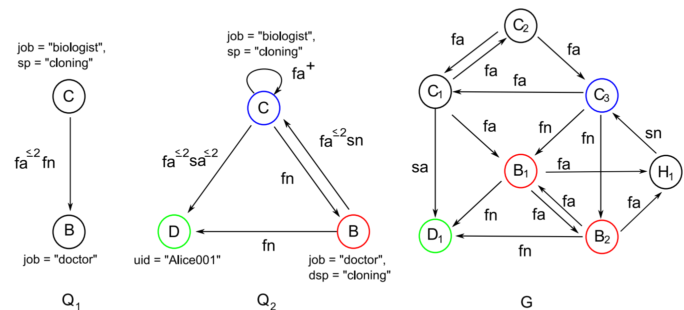

Fig. 1. Querying Essembly Network

Figure 1 depicts a part of the network as graph G that involves a debate on cloning research. In G each node denotes a person, and each edge has a type in $\{\mathsf{fa}, \mathsf{fn}, \mathsf{sa}, \mathsf{sn}\}$. Consider two queries Q_1 and Q_2 on G, also shown in Fig. 1.

(1) Query Q_1 is a reachability query, which is to find all biologists (nodes C) who support "cloning", along with those doctors (nodes B) who are friends-nemeses (via fn) of some users supported by C within 2 hops (via $\mathsf{fa}^{\leq 2}$).

(2) Query Q_2 is a pattern query, issued by a person D identified by id "Alice001" who supports "cloning". The person would like to find all her friends-nemeses (via fn) who are doctors, and are against "cloning". She also wants to know if there are people such that (a) they are biologists (nodes C), support "cloning research", and are connected within 2 hops to someone via fa relationships, who is in turn within 2 hops to person D via sa (edge (C, D)); (b) they are in a scientist group with friends all sharing the same view (edge (C, C)); and moreover, (c) these biologists are against those doctor friends of her, and vice versa, via paths of certain patterns (edges (C, B) and (B, C)).

Observe the following. (1) The graph G has multiple edge types $(\mathsf{fa}, \mathsf{fn}, \mathsf{sa}, \mathsf{sn})$ indicating different relationships, which are an important part of the semantics of the data. (2) Traditional reachability queries are not capable of expressing Q_1. Indeed, they characterize connectivity by the existence of a path of *arbitrary length*, with edges of *arbitrary types*. In contrast, Q_1 aims to identify connectivity via a path

(a) containing edges of particular *types* and *patterns*, and
(b) with a *bound* on its *lengths* (hops).

Here Q_1 bears richer semantics than its conventional counterparts. (3) Traditional graph pattern queries cannot express Q_2 for the two reasons given above; moreover, to find sensible information for person D, it should logically allow:

(c) its node to map to *multiple* nodes in G, *e.g.,* from B in Q_2 to both B_1 and B_2 in G, and
(d) its edges map to paths composing of edges with certain types, *e.g.,* from (C, D) in Q_2 to $C_3 \xrightarrow{\text{fa}} C_1 \xrightarrow{\text{sa}} D_1$ in G.

That is, traditional pattern queries defined *w.r.t.* subgraph isomorphism are no longer sufficient for expressing Q_2. □

As suggested by the example, emerging applications highlight the need for revising the traditional reachability and graph pattern queries to incorporate edge types and bounds on the number of hops. In addition, it is necessary to revise graph pattern matching to accommodate the semantics of data in new applications, and moreover, to reduce its complexity. Indeed, the NP-completeness of subgraph isomorphism makes it infeasible to match large data graphs.

Contributions & Roadmap. We propose a class of reachability queries and a class of graph pattern queries, defined in terms of a subclass F of regular expressions.

(1) We introduce reachability queries (RQs) and graph pattern queries (PQs) in Section II. In such a query, each node specifies search conditions on the content of the graph nodes, and each edge is associated with a regular expression in F, specifying the connectivity via a path of certain edge types and of a possibly bounded length. Moreover, we define pattern matching by extending graph simulation [19], instead of using subgraph isomorphism. For instance, Q_1 and Q_2 in Fig. 1 can be expressed as an RQ and a PQ, respectively.

(2) We study fundamental problems for these queries: containment, equivalence and minimization (Section III), along the same lines as for XML tree pattern queries [24], [36]. We show that these problems are in $O(n^2)$ time and $O(n^3)$ time for RQs and PQs, respectively, where n is the size of the queries. Contrast these low polynomial time (PTIME) bounds with their counterparts for general regular expressions, which are PSPACE-complete [25]. As an immediate application of these analyses, we develop an $O(n^3)$ algorithm to minimize PQs, and experimentally show its effectiveness.

(3) We develop two algorithms to answer RQs (Section IV). One employs a matrix of shortest distances between nodes. It is in *quadratic time*, the same as its traditional counterpart [33]. That is, the increased expressive power of RQs does *not* imply extra complexity. The other adopts bi-directional search with an auxiliary cache (using hashmap as indices) to keep track of frequently asked items. It is used when it is too costly to maintain all shortest distances for large graphs.

(4) We provide two algorithms for evaluating PQs (Section V), both in *cubic time* if a matrix of shortest distances between nodes is used. One follows a join-based approach, while the other adopts a split-based approach commonly used in labeled transition systems. Contrast this with the intractability of graph pattern matching based on subgraph isomorphism. These tell us that the revised notion of graph pattern matching allows us to efficiently find sensible patterns in emerging applications.

(5) Using both real-life data (*YouTube* and *Global Terrorism Database* [1]) and synthetic data, we conduct an experimental study (Section VI). We find that our evaluation algorithms for RQs and PQs scale well with large data graphs, and are able to identify sensible matches that their traditional counterparts fail to find. We also find that the minimization algorithm of PQs is effective in improving performance.

Related work. The idea of using regular expressions to query graphs has been adopted by query languages for semistructured data such as UnQL [7] and Lorel [3]. There has also been theoretical work on conjunctive regular path queries (CRPQs, *e.g.,* [16]) and extended CRPQs (ECRPQs) [5], which also define graph queries using regular expressions. However, these languages are defined with general regular expressions. As a result, the problem for evaluating CRPQs is already NP-complete, and it is PSPACE-complete for ECRPQs [5]. For those queries the containment and minimization analyses are also PSPACE-hard. We are not aware of any existing efficient algorithms for answering graph pattern queries defined with regular expressions. In contrast, this work defines graph queries in terms of a subclass of regular expressions, and revises the notion of pattern matching based on an extension of graph simulation. It aims to strike a balance between the expressive power needed to deal with common graph queries in emerging applications, and the increased complexity incurred. This allows us to conduct the static analyses (containment and minimization) and evaluate queries efficiently, in low PTIME.

There have also been recent graph query languages that support limited regular expressions, *e.g.,* GQ [18], SoQL [27] and SPARQL [29]. GQ supports arbitrary attributes on nodes, edges and graphs. SoQL is a SQL-like language that allows users to retrieve paths satisfying various conditions. SPARQL [29] is a query language tailored for RDF graphs coded as a set of triples (subject, predicate and object). It is based on subgraph isomorphism (NP-complete) for graph pattern search, which differs from this work (in PTIME).

A number of algorithms have been developed for evaluating reachability queries [12], [33]. These algorithms typically associate certain coding with graph nodes, and detect connectivity by inspecting the coding of relevant nodes. The coding, however, tells us neither the distance between nodes nor the types of edge on the shortest path. Distance queries [12], [34] compute the distance between a pair of nodes, but do not consider edge types. Recently, a class of label-constraint reachability queries was proposed in [21], which asks whether one node reaches another via a path whose edge labels are in a set of labels. However, none of these can express reachability characterized by regular expressions, such as Q_1 in Fig. 1.

Graph pattern matching is typically defined in terms of subgraph isomorphism [31], [37] (see [17], [28] for surveys).

Extensions of subgraph isomorphism are studied in [13], [15], [37], which extend mappings from edge-to-edge to edge-to-path. However, the problem remains NP-complete. Closer to this work is bounded simulation studied in [14], which is an extension of graph simulation [8], [19] for graph pattern matching. Graph simulation has proved useful in *e.g.,* process calculus [19] and Web site classification [11]. Bounded simulation [14] imposes bounds on the number of hops, and makes graph pattern matching a PTIME problem. This work further extends [14] by incorporating regular expressions as edge constraints, and for these more expressive graph queries, it develops efficient evaluation algorithms and settles their fundamental problems (containment, equivalence and minimization). No previous work has studied these.

The containment and minimization problems are classical problems for any query language (see, *e.g.,* [2]). These problems have been well studied for XPath (*e.g.,* [9], [24], [36]). However, we are not aware of previous work on these problems for graph pattern queries.

II. Graph Reachability and Pattern Queries

We start with data graphs, on which we then introduce reachability queries (RQs) and graph pattern queries (PQs).

Data graphs. A *data graph* is a directed graph $G = (V, E, f_A, f_C)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a finite set of edges, in which (v, v') denotes an edge from node v to v'; (3) f_A is a function defined on V such that for each node v in V, $f_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where $A_i = a_i$ ($i \in [1, n]$), representing that the node v has a constant value a_i for the attribute A_i, and denoted as $v.A_i = a_i$; and (4) f_C is a function defined on E such that for each edge e in E, $f_C(e)$ is a *color* symbol in a finite alphabet Σ.

Intuitively, the function f_A carries node properties, *e.g.,* labels, keywords, blogs, comments and ratings [17]; the function f_C specifies edge types, *i.e.,* relationships; and the alphabet Σ denotes all possible edge types, *e.g.,* marriage, friendship, work, advice, support, exchange and co-membership [23].

Example 2: Figure 1 shows a data graph $G = (V, E, f_A, f_C)$, where (1) each edge e in E carries a color $f_C(e)$ in $\{\mathsf{fa}, \mathsf{fn}, \mathsf{sa}, \mathsf{sn}\}$; and (2) each node v in V has a tuple $f_A(v)$, such that (a) for each $i \in [1, 2]$, $f_A(B_i)$ = (job = “doctor”, dsp = “cloning”), (b) for each $j \in [1, 3]$, $f_A(C_j)$ = (job = “biologist”, sp = “cloning”), (c) $f_A(D_1)$ = (uid = “Alice001”), and (d) $f_A(H_1)$ = (job = “physician”). □

We shall use the following notations for data graphs G.

(1) A *path* ρ in G is denoted as $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots v_{n-1} \xrightarrow{e_n} v_n$, where (a) $v_i \in V$ for each $i \in [0, n]$, and (b) $e_j = (v_{j-1}, v_j)$ in E for each $j \in [1, n]$. The *length* $|\rho|$ of ρ is n, *i.e.,* the number of edges in ρ. We say a path ρ is *nonempty* if $|\rho| \geq 1$.

(2) Abusing notations for trees, we refer to a node v_2 as a *child* of a node v_1 (or v_1 as a *parent* of v_2) if there exists an edge (v_1, v_2) in E, and refer to a node v_2 as a *descendant* of a node v_1 (or v_1 as an *ancestor* of v_2) if there exists a nonempty path from node v_1 to node v_2 in G.

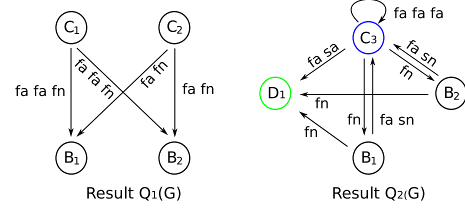

Fig. 2. Results of the queries Q_1 and Q_2 on G

Reachability queries. A *reachability query* (RQ) is defined as $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, f_e)$, where (1) u_1 and u_2 are two nodes; (2) f_{u_i} ($i \in [1, 2]$) is a predicate defined as a conjunction of atomic formulas of the form of ‘A op a’ such that A denotes an attribute of the node u_i, a is a constant value, and op is a comparison operator in the set $\{<, \leq, =, \neq, >, \geq\}$; and (3) f_e is a regular expression drawn from the subclass:

$$F ::= \ c \mid c^{\leq k} \mid c^+ \mid FF.$$

Here (1) c is either a color symbol in Σ or a wildcard _, where the wildcard _ is a variable standing for any color symbol in Σ; it can be expressed as a regular expression $c_1 \cup \ldots \cup c_m$, when $\Sigma = \{c_i \mid i \in [1, m]\}$; (2) k is a positive integer, and $c^{\leq k}$ denotes the regular expression $c^1 \cup c^2 \cup \ldots \cup c^k$, where c^j ($j \in [1, k]$) denotes j occurrences of c; and (3) c^+ denotes one or more occurrences of c.

We use $L(f_e)$ to denote the regular language defined by f_e, *i.e.,* the set of strings that can be parsed by f_e.

Semantics. Consider an RQ $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, f_e)$ posed on a data graph $G = (V, E, f_A, f_C)$.

We say that a node v in G *matches* the node u_1 in G_r, denoted as $v \sim u_1$, if for each atomic formula ‘A op a’ in f_{u_1}, there exists an attribute A in $f_A(v)$ such that $v.A$ op a; similarly for $v \sim u_2$. Intuitively, the predicates f_{u_1} and f_{u_2} specify search conditions on the content of nodes.

We say that a pair (v_1, v_2) of nodes in G *matches* the regular expression f_e, denoted as $(v_1, v_2) \approx f_e$, if there exists a nonempty path $\rho = v_1 \xrightarrow{e_1} v_1' \xrightarrow{e_2} v_2' \ldots v_{n-1}' \xrightarrow{e_n} v_2$ in G such that the string $f_C(e_1) \ldots f_C(e_n)$ is in $L(f_e)$.

The *result* $Q_r(G)$ of Q_r on G is the set of node pairs (v_1, v_2) in G such that $v_1 \sim u_1$, $v_2 \sim u_2$, and $(v_1, v_2) \approx f_e$.

That is, v_1 (resp. v_2) satisfies the conditions specified by u_1 (resp. u_2); and moreover, there exists a nonempty path from v_1 to v_2 in G such that the edge colors on the path match the pattern specified by the regular expression f_e.

Example 3: The query Q_1 shown in Fig. 1 is an RQ, in which f_e = $\mathsf{fa}^{\leq 2}\mathsf{fn}$. The node C has the predicate sp = “cloning” **and** job = “biologist”, and similar for the node B.

When Q_1 is posed on the data graph G shown in Fig. 1 and described in Example 2, the answer $Q_1(G)$ is shown in Fig. 2. Indeed, $B_i \sim B$ ($i \in [1, 2]$) and $C_j \sim C$ ($j \in [1, 3]$). In addition, $(C_2, B_1) \approx f_e$ since there exists a path $C_2 \xrightarrow{\mathsf{fa}} C_3 \xrightarrow{\mathsf{fn}} B_1$ in G, and the string fa fn matches the regular expression $\mathsf{fa}^{\leq 2}\mathsf{fn}$. Similarly, $(C_1, B_1) \approx f_e$, $(C_1, B_2) \approx f_e$, and $(C_2, B_2) \approx f_e$. Hence the query result $Q_1(G) = \{(C_1, B_1), (C_1, B_2), (C_2, B_1), (C_2, B_2)\}$. □

Remark. (1) Observe that a single edge in query Q_r is mapped to a nonempty path in the data graph G; moreover, the edge colors on the path have to match the regular expression f_e.

(2) RQs are more expressive than traditional reachability queries studied in *e.g.,* [21], [34], by capturing edge relationships with regular expressions.

Graph pattern queries. Using RQs as building blocks, we next define graph pattern queries.

A *graph pattern query* (PQ) is a directed graph $Q_p = (V_p, E_p, f_v, f_e)$, where (1) V_p is a finite set of nodes; (2) $E_p \subseteq V_p \times V_p$ is a finite set of edges, where (u, u') denotes an edge from node u to u'; and (3) the functions f_v and f_e are defined on V_p and E_p, respectively, such that for each edge $e=(u, u') \in E_p$, $Q_r = (u, u', f_v(u), f_v(u'), f_e(e))$ is an RQ.

Semantics. When the graph pattern query Q_p is evaluated on a data graph $G = (V, E, f_A, f_C)$, the result $Q_p(G)$ is the maximum set $\{(e, S_e) \mid e \in E_p\}$ that satisfies the following:

(1) for all edges $e = (u_1, u_2)$ in Q_p, $S_e \subseteq Q_e(G)$, where $Q_e = (u_1, u_2, f_v(u_1),\ f_v(u_2), f_e(e))$ is an RQ;

(2) for each edge $e = (u_1, u_2)$ in Q_p, if a pair (v_1, v_2) of nodes in G is in S_e, then (a) for each edge $e_1 = (u_1, u_3)$ in Q_p, there exists a node v_3 in G such that $(v_1, v_3) \in S_{e_1}$; and (b) for each edge $e_2 = (u_2, u_4)$ in Q_p, there exists a node v_4 in G such that $(v_2, v_4) \in S_{e_2}$; and

(3) there exists no edge e in Q_p such that S_e is empty.

Intuitively, $Q_P(G)$ defines a relation $R \subseteq V_p \times V$. To see this, for each edge $e = (u_1, u_2)$ in Q_p, denote by $Q_e = (u_1, u_2, f_v(u_1),\ f_v(u_2), f_e(e))$ its associated RQ embedded in G_p. Then for a node $u_1 \in V_p$ and a node $v_1 \in V$, (u_1, v_1) is in R if for each edge $e = (u_1, u_2)$ emanating from u_1 in G_p, there exists a nonempty path ρ from v_1 to v_2 in G such that (1) the node v_1 satisfies the search conditions specified by $f_v(u_1)$ in the RQ Q_e; (2) the path ρ is constrained by the regular expression $f_e(e)$; and (3) (u_2, v_2) is also in R. In addition, R covers all the nodes in V_p and is maximum. The result $Q_p(G)$ is simply R grouped by edges in E_p.

From this one can see that PQs are defined in terms of an extension of graph simulation [19], by (a) imposing search conditions on the contents of nodes; (b) mapping an edge in a pattern to a nonempty path in a data graph (*i.e.,* the child u_2 of u_1 is mapped to a descendant of v_2 of v_1); and (c) constraining the edges on the path with a regular expression. This also differs from the traditional notion of graph pattern matching defined in terms of subgraph isomorphism [17].

Example 4: The query Q_2 given in Fig. 1 is a PQ. In Q_2 each node carries search conditions, and each edge has an associated regular expression, as shown in Fig. 1.

When Q_2 is posed on the data graph G of Fig. 1, the result $Q_2(G)$ is depicted in Fig. 2 and is shown in the table below:

edge	matches	edge	matches
(B, C)	$\{(B_1, C_3), (B_2, C_3)\}$	(C, C)	$\{(C_3, C_3)\}$
(B, D)	$\{(B_1, D_1), (B_2, D_1)\}$	(C, D)	$\{(C_3, D_1)\}$
(C, B)	$\{(C_3, B_1), (C_3, B_2)\}$		

Indeed, one can verify that $B_i \sim B$ ($i \in [1, 2]$), $C_j \sim C$ ($j \in [1, 3]$) and $D_1 \sim D$. In addition, the edge from C to D (labeled with $\mathsf{fa}^{\leq 2}\mathsf{sa}^{\leq 2}$) in Q_2 is mapped to a path $C_3 \xrightarrow{\mathsf{fa}} C_1 \xrightarrow{\mathsf{sa}} D_1$ in G; similarly for other edges in Q_2.

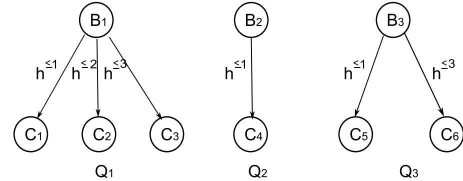

Fig. 3. Example for containment and equivalence

Observe that the node pair (C_1, B_1) in G is not a match of the edge (C, B) in Q_2, since there exists no path in G from C_1 to B_1 that satisfies fn. In light of this, (C_1, D_1) in G is not a match of the edge (C, D) in Q_2, although there exists a path $C_1 \xrightarrow{\mathsf{fa}} C_2 \xrightarrow{\mathsf{fa}} C_1 \xrightarrow{\mathsf{sa}} D_1$ in G that satisfies $\mathsf{fa}^{\leq 2}\mathsf{sa}^{\leq 2}$. □

Remark. (1) RQs are a special case of PQs, which consist of two nodes and a single edge.

(2) Bounded simulation [14] is a special case of PQs, by only allowing patterns in which (a) there is a single symbol c in Σ, *i.e.,* only a single edge type is allowed, and (b) all edges are labeled with either $c^{\leq k}$ or c^{+}, where k is a positive integer.

One can readily verify the following, which confirms that the semantics of PQs is well defined.

Proposition 1: *For any data graph G and any graph pattern query Q_p, there is a unique result $Q_p(G)$.* □

III. Fundamental Problems for Graph Queries

We next investigate containment, equivalence and minimization of graph queries. As remarked earlier, these problems are important for any query language [2]. We focus on graph pattern queries (PQs), but state the relevant results for reachability queries (RQs), a special case of PQs.

A. Containment and Equivalence

We first study containment and equivalence of PQs.

Containment. Given two PQs $Q_1 = (V_p^1,\ E_p^1, f_v^1, f_e^1)$ and $Q_2 = (V_p^2,\ E_p^2, f_v^2, f_e^2)$, we say that Q_1 is *contained* in Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if there exists a *mapping* λ from E_p^1 to E_p^2 such that for any data graph G and any edge e in Q_1, $S_e \subseteq S_{\lambda(e)}$, where $(e, S_e) \in Q_1(G)$, $(\lambda(e), S_{\lambda(e)}) \in Q_2(G)$, and $Q_1(G), Q_2(G)$ are the results of Q_1, Q_2 on G, respectively.

Intuitively, λ serves as a renaming function such that $Q_1(G)$ is mapped to $Q_2(G)$ after the renaming. For an edge $e = (u_1, u_2)$ in Q_1, let $\lambda(e) = (w_1, w_2)$. Then $Q_1 \sqsubseteq Q_2$ as long as for any data graph G and any node v in G, (1) if $v \sim u_1$, then $v \sim w_1$, denoted as $u_1 \vdash w_1$; and (2) $u_2 \vdash w_2$. Moreover, (3) $L(f_e^1(e)) \subseteq L(f_e^2(\lambda(e)))$, denoted as $e \models \lambda(e)$.

Example 5: Consider three PQs given in Fig. 3, in which all B_i's ($i \in [1, 3]$) carry the same predicates; similarly for all C_j's ($j \in [1, 6]$). Denote by $\lambda_{i,j}$ a mapping from Q_i to Q_j.

(1) $Q_2 \sqsubseteq Q_1$: there exists a mapping $\lambda_{2,1}$, where $\lambda_{2,1}$ ((B_2, C_4)) = (B_1, C_1). Note that the mapping is not unique, *e.g.,* both $\lambda_{2,1}((B_2, C_4)) = (B_1, C_2)$ and $\lambda_{2,1}((B_2, C_4)) = (B_1, C_3)$ are valid mappings.

(2) $Q_2 \sqsubseteq Q_3$, by letting $\lambda_{2,3}((B_2, C_4)) = (B_3, C_5)$.

(3) $Q_3 \sqsubseteq Q_1$, Indeed, one can define $\lambda_{3,1}((B_3, C_5))$ = (B_1, C_1) and $\lambda_{3,1}((B_3, C_6))$ = (B_1, C_3).

(4) $Q_1 \sqsubseteq Q_3$, by letting $\lambda_{1,3}((B_1, C_1)) = (B_3, C_5)$, $\lambda_{1,3}$ $((B_1, C_2)) = (B_3, C_5)$ and $\lambda_{1,3}$ $((B_1, C_3)) = (B_3, C_6)$. □

Equivalence. For PQs Q_1 and Q_2, we say that Q_1 and Q_2 are *equivalent*, denoted by $Q_1 \equiv Q_2$, if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.

For instance, for Q_1 and Q_3 of Fig. 3, we have that $Q_1 \equiv Q_3$, since $Q_1 \sqsubseteq Q_3$ and $Q_3 \sqsubseteq Q_1$ by Example 5.

Observe that $Q_1 \equiv Q_2$ does not necessarily imply that $Q_1(G) = Q_2(G)$ for a data graph G. Nevertheless, there exist mappings $\lambda_{1,2}$ and $\lambda_{2,1}$ such that $\lambda_{1,2}(Q_1(G)) \subseteq Q_2(G)$ and $\lambda_{2,1}(Q_2(G)) \subseteq Q_1(G)$, where $\lambda(Q(G))$ stands for $\{(\lambda(e), S_{\lambda(e)}) \mid (e, S_e) \in Q(G)\}$. That is, $Q_1(G)$ and $Q_2(G)$ are mapped to each other after the renaming by $\lambda_{1,2}$ and $\lambda_{2,1}$.

Complexity bounds. We next establish the complexity bounds of the containment and equivalence problems for PQs. To do this we first present a revision of similarity [19].

Consider two PQs $Q_1 = (V_p^1, E_p^1, f_v^1, f_e^1)$ and $Q_2 = (V_p^2, E_p^2, f_v^2, f_e^2)$. We say that Q_2 is *similar to* Q_1, denoted by $Q_1 \trianglelefteq Q_2$, if there exists a binary relation $S \subseteq V_p^1 \times V_p^2$ such that
(1) for any $(u_1, w_1) \in S$, (a) $w_1 \vdash u_1$, and (b) for each edge $e = (u_1, u_2) \in E_p^1$, there exists an edge $e' = (w_1, w_2) \in E_p^2$ such that $(u_2, w_2) \in S$ and $e' \models e$; and
(2) for each edge $e' = (w, w') \in E_p^2$, there exists an edge $e = (u, u') \in E_p^1$ such that (a) $(u, w), (u', w') \in S$ and (b) $e' \models e$.

Example 6: Recall PQs Q_1 and Q_2 from Example 5. One can verify that $Q_1 \trianglelefteq Q_2$. Indeed, there exists a binary relation $S = \{(B_1, B_2), (C_1, C_4), (C_2, C_4), (C_3, C_4)\}$, which satisfies the conditions of the revised similarity given above:
(1) for each $(u, w) \in S$, $w \vdash u$ (the condition (1)(a) above);
(2) for each edge e in Q_1 (*i.e.,* (B_1, C_1), (B_1, C_2) and (B_1, C_3)), there exists an edge e' in Q_2 (*i.e.,* (B_2, C_4)) such that $e' \models e$, since $L(h^{\leq 1})$ is contained in $L(h^{\leq 1})$, $L(h^{\leq 2})$ and $L(h^{\leq 3})$ (the condition (1)(b) above); and
(3) for the edge $e' = (B_2, C_4)$ in Q_2, there is an edge $e' = (B_1, C_1)$ in Q_1 such that $e' \models e$ (the condition (2) above). □

The relationship between the revised graph similarity and the containment of PQs is shown below.

Lemma 2: *For* PQs Q_1 *and* Q_2, $Q_1 \sqsubseteq Q_2$ *iff* $Q_2 \trianglelefteq Q_1$. □

It is known that graph similarity is solvable in quadratic time [19]. Extending the techniques of [19] by leveraging Lemma 2, one can verify the following:

Theorem 3: *Given two* PQs Q_1 *and* Q_2*, it is in cubic time to determine whether* $Q_1 \sqsubseteq Q_2$ *and whether* $Q_1 \equiv Q_2$. □

As a special case of PQs, the containment problem and the equivalence problem for RQs are much easier.

Proposition 4: *Given two* RQs Q_1 *and* Q_2*, it is in quadratic time to check whether* $Q_1 \sqsubseteq Q_2$ *or whether* $Q_1 \equiv Q_2$. □

Contrast this with the PSPACE-completeness of the containment problem for general regular expressions [20]. The gap between the two complexity bounds justifies the choice of the subclass F of regular expressions for RQs and PQs: those regular expressions have sufficient expressive power to specify edge relationships commonly found in practice, and moreover, allow efficient static analyses of fundamental properties.

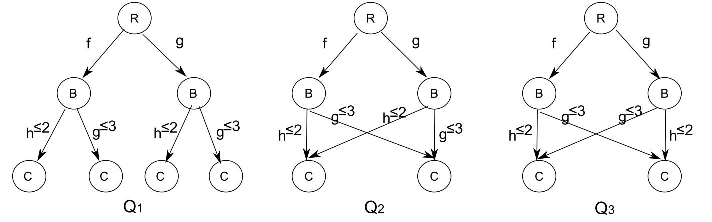

Fig. 4. Non-isomorphic equivalent minimum PQs

Algorithm minPQs
Input: PQ $Q = (V_p, E_p, f_v, f_e)$.
Output: a minimum equivalent PQ Q_m of Q.

1. compute the maximum revised graph similarity S over Q;
2. compute the node equivalent classes EQ based on S;
3. determine the edges for equivalent class pairs in EQ;
4. determine the number of copies for equivalent classes in EQ;
5. construct an equivalent query Q_m;
6. remove redundant edges in Q_m;
7. remove isolated nodes in Q_m;
8. **return** Q_m.

Fig. 5. Algorithm minPQs

B. Minimizing Graph Pattern Queries

A problem closely related to query equivalence is query minimization, which often yields an effective optimization strategy. It has been studied for, *e.g.,* relational conjunctive queries [2] and XML tree pattern queries [9], [24], [36]. For all the reasons that query minimization is important for relational queries and XML queries, we also need to study minimization of graph queries.

For a PQ $Q = (V_p, E_p)$, we define its size $|Q| = |V_p| + |E_p|$, a metric commonly used for pattern queries [9].

Minimization. Given a PQ $Q = (V_p, E_p, f_v, f_e)$, the *minimization* problem is to find another PQ $Q_m = (V_p^m, E_p^m, f_v^m, f_e^m)$ such that (1) $Q_m \equiv Q$, (2) $|Q_m| \leq |Q|$, and (3) there exists no other such Q' with $|Q'| < |Q_m|$. We refer to Q_m as a *minimum equivalent* PQ of Q.

Remark. (1) A PQ may have multiple minimum equivalent PQs of the same size that are not isomorphic to each other. As shown in Fig. 4, both Q_2 and Q_3 are minimum equivalent PQs of Q_1 with the same size, but they are not isomorphic.
(2) We ignore regular expressions in the minimization analysis since for those in the particular subclass F used in RQs and PQs, it takes linear time to minimize them. In addition, as will be seen from our algorithms in Section V, minimizing RQs has little impact on their complexity. This would be, however, no longer the case if general regular expressions were adopted. This further justifies the choice of F in the definition of PQs.

Below we focus on minimization of PQs since the case for RQs is trivial. The last main result of the section is as follows.

Theorem 5: *Given any* PQ Q*, a minimum equivalent* PQ Q_m *of* Q *can be computed in cubic time.* □

To show Theorem 5, we develop an algorithm that, given a PQ Q, finds a minimum equivalent PQ of Q in cubic time.

The algorithm, referred to as minPQs, is outlined in Fig. 5. Due to space constraint we illustrate how the algorithm works with an example.

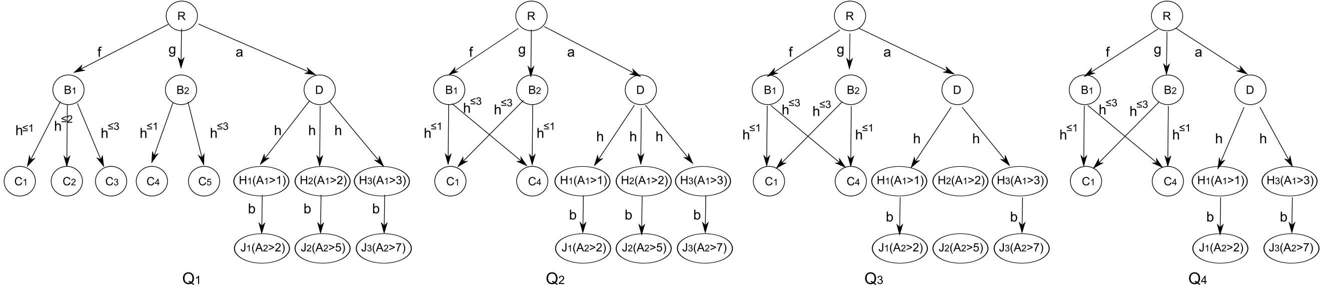

Fig. 6. Example for minimizing graph pattern queries

Example 7: Consider the PQ Q_1 shown in Fig. 6, where (a) nodes B_1 and B_2 have the same predicate, (b) all nodes labeled with C (C_i, $i \in [1, 5]$) have the same predicate, and (c) all nodes with distinct labels (ignoring subscripts) have different predicates. For clarity, we only explicitly annotate the predicates of the nodes labeled with H and J. The query Q_4 given in Fig. 6 is a minimum equivalent PQ of Q_1. Below we show how algorithm minPQs finds Q_4 step by step.

(1) The maximum similarity S on a PQ $Q(V_p, E_p)$ is the maximum relation $S \subseteq V_p \times V_p$ that satisfies the conditions of the revised similarity. One can verify that there exists a unique maximum one, along the same lines as [8].

The maximum similarity S on Q_1 is $\{(R, R)$, (B_{i_1}, B_{j_1}), $(C_{i_2}, C_{j_2}), (D, D), (H_{i_3}, H_{j_3}), (J_{i_4}, J_{j_4})\}$, where $1 \leq i_1, j_1 \leq 2$, $1 \leq i_2, j_2 \leq 5$, $1 \leq i_3 \leq j_3 \leq 3$, and $1 \leq i_4 \leq j_4 \leq 3$.

(2) An equivalent relation EQ is derived from the similarity relation S. More specifically, two nodes u, w in Q_1 are in the same equivalence class of EQ if $(u, w) \in S$ and $(w, u) \in S$.

For Q_1, EQ consists of $\mathsf{eq}_0 = \{R\}$, $\mathsf{eq}_1 = \{B_1, B_2\}$, $\mathsf{eq}_2 = \{C_1, C_2, C_3, C_4, C_5\}$, $\mathsf{eq}_3 = \{D\}$, $\mathsf{eq}_4 = \{H_1\}$, $\mathsf{eq}_5 = \{H_2\}$, $\mathsf{eq}_6 = \{H_3\}$, $\mathsf{eq}_7 = \{J_1\}$, $\mathsf{eq}_8 = \{J_2\}$, and $\mathsf{eq}_9 = \{J_3\}$.

(3) Consider two equivalent classes eq_1 and eq_2 in EQ, and let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of edges in Q_1 from the nodes in eq_1 to the nodes in eq_2. An edge e in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ is *redundant* if there exist two edges e_1, e_2 in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $e_1 \neq e$, $e_2 \neq e$ and $L(e_1) \subseteq L(e) \subseteq L(e_2)$.

(4) The number $N(\mathsf{eq})$ of the copies of an equivalent eq in EQ is determined by the *maximum* number of non-redundant edges in $E(\mathsf{eq}', \mathsf{eq})$ for all $\mathsf{eq}' \in \mathsf{EQ}$.

(5) After the non-redundant edges and the number of copies for equivalent classes in EQ are determined, an equivalent query Q_2 for Q_1 is constructed, shown in Fig. 6, by connecting (copies of) equivalent classes with non-redundant edges.

(6) To remove the redundant edges from Q_2, we first compute the maximum similarity S' on Q_2. An edge $e = (u, u')$ in Q_2 is *redundant* if there exist two edges $e_1 = (u_1, u_1')$ and $e_2 = (u_2, u_2')$ in Q_2 such that (a) $e_1 \neq e$, $e_2 \neq e$, (b) (u, u_1), (u', u_1'), (u_2, u), $(u_2', u') \in S'$, and (c) $e_1 \models e$ and $e \models e_2$.

After redundant edges are removed, Q_2 becomes the query Q_3 shown in Fig. 6.

(7) A node u in Q_3 is *isolated* if it does not have any edge.

After all isolated nodes are removed, the query Q_3 becomes Q_4 shown in Fig. 6. The algorithm then returns Q_4 as a minimum equivalent query of the query Q_1. □

Correctness & complexity. To show that algorithm minPQs indeed finds a minimum equivalent PQ Q_m of Q, (1) we first show that $Q_m \equiv Q$, by proving that the operations in the algorithm preserve query equivalence; and (2) then show that Q_m is a smallest equivalent query, by contradiction.

Algorithm minPQs runs in cubic time since each step in the algorithm can be done in cubic time in the size of the query.

From the correctness and complexity analysis of algorithm minPQs, Theorem 5 immediately follows.

IV. Evaluating Reachability Queries

We show that the increased expressive power of RQs with regular expressions does not incur extra complexity, by developing two simple methods to answer RQs. One employs a matrix of shortest distances between nodes. It is in *quadratic time*, the same as its counterpart for traditional reachability queries [33]. The other adopts bi-directional breadth-first search (BFS), and utilizes an auxiliary cache to maintain the most frequently asked items. It is used when maintaining a distance matrix is infeasible for large data graphs.

Consider an RQ $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, f_e)$ and a data graph G. For nodes v_1, v_2 of G, we want to determine whether v_i matches u_i ($i \in [1, 2]$) and moreover, whether there exists a path from v_1 to v_2 that matches f_e (see Section II). Below we start with a special case when f_e carries a single edge color, and then consider the general case with multiple colors.

Matrix-based method. We use a 3-dimensional *matrix* M, where 2 dimensions range over data graph nodes and 1 dimension is for edge colors. For two nodes v_1, v_2 in graph G, $M[v_1][v_2][c]$ (resp. $M[v_1][v_2][_]$) records the length of the shortest path from v_1 to v_2 via edges of color c (resp. arbitrary colors). Capitalizing on M one can detect in constant time whether v_1 reaches v_2 via a path satisfying f_e.

Assume that there are m distinct edge colors in G. The matrix can be built in $O((m+1)|V|^2 + |V|(|V| + |E|))$ time by using BFS [4], where m is typically much smaller than $|V|$. This matrix is pre-computed and shared by all queries.

Leveraging the matrix M, Q_r can be answered in $O(|V|^2)$ time by inspecting those nodes that satisfy the search conditions specified by u_1 and u_2, using a nested loop.

Bi-directional search. The space overhead $O((m+1)|V|^2)$ of the distance matrix, however, may hinder its applicability. To cope with large graphs, we propose to maintain a distance cache using hashmap as indices, which records the most frequently asked items. If an entry for a node pair (v_1, v_2) and a color c is not cached, it is computed at runtime and the cache is updated with the least recently used (LRU) replacement

strategy. To do this we adopt a bi-directional BFS at runtime as follows. Two sets are maintained for v_1 and v_2, respectively. Each set records the nodes that are reachable from (resp. to) v_1 (resp. v_2) only via edges of color c. We expand the smaller set at a time until either the two sets intersect (*i.e.,* the distance is the number of total expansions), or they cannot be further expanded (*i.e.,* unreachable). This procedure runs in $O(|V| + |E|)$. A similar technique is used in [10], but it does not consider edge colors.

Compared with traditional BFS, the bi-directional search strategy can significantly reduce the search space, especially when edge colors are considered. For instance, in data graph G at Fig. 1, if a user asks whether there exists a path from C_2 to D_1 satisfying the constraint fa^+, we can immediately answer *no* since no incoming edge to D_1 is colored with fa.

Next we extend the two methods to evaluate a general RQ Q_r. Assume that the number of edge colors in f_e is h.

Matrix-based method. We decompose Q_r into h RQs: $Q_{r_i} = (x_i, y_i, f_{x_i}, f_{y_i}, f_{e_i})$ ($i \in [1, h]$), where $x_1 = u_1$, $y_k = u_2$, and we add $y_j = x_{j+1}$ ($j \in [1, h-1]$) as dummy nodes between u_1 and u_2. Here each f_{e_i} ($i \in [1, h]$) carries a single edge color, and a dummy node d bears no condition, *i.e.,* for any node v in G, v matches d. Using the procedure for answering single-colored RQs, we evaluate Q_{r_i} from h to 1; we then compose these partial results to derive $Q_r(G)$. This is in $O(h|V|^2)$ time, where h is typically small and omitted.

Example 8: Recall the RQ Q_1 from Fig. 1 with edge constraint f_e = $\mathsf{fa}^{\leq 2}\mathsf{fn}$. The query Q_1 can be decomposed into $Q_{1,1}$ and $Q_{1,2}$ by inserting a dummy node d between C and B, where $Q_{1,1}$ (resp. $Q_{1,2}$) has an edge (C, d) (resp. (d, B)) with edge constraint $\mathsf{fa}^{\leq 2}$ (resp. fn).

When evaluating $Q_{1,2}$ on the data graph G of Fig. 1, we get $Q_{1,2}(G) = \{(C_3, B_1), (C_3, B_2)\}$, since $M[C_3][B_1][\mathsf{fn}]=1$ and $M[C_3][B_2][\mathsf{fn}] = 1$. Similarly, by $C_3 \sim d$ derived from $Q_{1,2}(G)$, we get $Q_{1,1}(G) = \{(C_1, C_3), (C_2, C_3)\}$. Combining $Q_{1,1}(G)$ and $Q_{1,2}(G)$, we find $Q_1(G)$. □

Bi-directional search. When a distance matrix is not available, runtime search is used instead, for evaluating an RQ $Q_r = (u_1, u_2, f_{u_1}, f_{u_2}, f_e)$. The bi-directional search method can handle the regular expression f_e, without decomposing it. Intuitively, this can be done by evaluating f_e by iteratively expanding from (resp. to) the nodes that may match u_1 (resp. u_2). In each iteration, the candidate match set with a smaller size will be expanded, and f_e is partially evaluated. When f_e is fully evaluated, we examine the intersection of the two sets to derive the result. This takes, however, $O(h|V|^2(|V|+|E|))$ time. Nonetheless, as will be seen in Section VI, this method is able to process queries on large data graphs, when maintaining a distance matrix for those graphs is beyond reach in practice.

These tell us that despite the increased expressiveness, RQs have the same complexity as their traditional counterparts [33]. Also note that although existing (index-based) solutions for traditional reachability queries cannot answer RQs directly, they can be used as filters, *i.e.,* only those connected nodes (possibly constrained by a set of labels) identified by those techniques are further checked by our algoithms.

V. Algorithms for Graph Pattern Queries

We next provide two algorithms to evaluate PQs. Given a data graph $G = (V, E, f_A, f_C)$ (simply written as (V, E)) and a PQ $Q_p = (V_p, E_p, f_v, f_e)$ (written as (V_p, E_p)), the two algorithms compute the result $Q_p(G)$ of Q_p on G, in *cubic time* in the size of G. One algorithm is based on join operations. The other is based on split, an operation commonly used in labeled transition system (LTS) verification [26].

A. Join-based Algorithm

We start with the join-based algorithm. It first computes, for each node u in the PQ Q_p, an initial set of (possible) matches, *i.e.,* nodes that satisfy the search conditions specified by u. It then computes $Q_p(G)$ as follows. (1) If Q_p is a *directed acyclic graph* (DAG), the query result is derived by a reversed topological order (bottom-up) process, which refines the match set of each query node by joining with the match sets of all its children, and by enforcing the constraints imposed by the corresponding query edges. (2) If Q_p is not a DAG, we first compute the *strongly connected components (*SCC*)* graph of Q_p, a DAG in which each node represents an SCC in Q_p. Then for all the query nodes within each SCC, their match sets are repeatedly refined with the join operations as above, until the *fixpoint* of the match set for each query node is reached. The algorithm utilizes the reverse topological join orders and nontrivial tricks to achieve the cubic-time complexity.

Algorithm. The algorithm, referred to as JoinMatch, is shown in Fig. 7. Besides Q_p and G, it also takes a boolean flag as input, indicating whether one opts to use a distance matrix. Based on flag, the algorithm decides to use which method given in Section IV to evaluate RQs embedded in Q_p.

The algorithm uses the following notations. We use u, v to denote nodes in the query Q_p, and x, y, z for nodes in the data graph G. (1) For each node u in Q_p, we initialize *its match set* $\mathsf{mat}(u)$ = $\{x \mid x \in V$ **and** $x \sim u\}$ (recall '$\sim$' from Section II). (2) For each edge $e = (u', u)$ in Q_p, we use a set $\mathsf{rmv}(e)$ to record the nodes in G that cannot match u' *w.r.t.* e. (3) An SCC graph of $Q_p = (V_p, E_p)$ is denoted as $Q_s = (V_s, E_s)$, where $C_s \in V_s$ presents an SCC in Q_p, and $(C'_s, C_s) \in E_s$ if there exists $v' \in C'_s$, $v \in C_s$ such that $(v', v) \in E_p$.

Algorithm JoinMatch first checks flag. If one wants to use a distance matrix M but it is not yet available, M is computed and Q_p is *normalized* as Q'_p (line 2), by decomposing each RQ of Q_p into simple RQs (*i.e.,* each edge only carries one color) via inserting dummy nodes. Otherwise no normalization is performed (line 1). The sets mat() and rmv() are then initialized (lines 3-4). The SCC graph Q_s of Q'_p is then computed, by using Tarjan's algorithm [30] (line 5).

In a reverse topological order, JoinMatch processes each node C_s of Q_s as follows: the match set of each query node in C_s is recursively refined until the *fixpoint* is reached (lines 7-14). For each node u in C_s and each edge $e = (u', u)$ (line 8), it computes the nodes in $\mathsf{mat}(u')$ that fail to satisfy the constraints of e, by invoking a procedure Join. The nodes

Input: a query $Q_p = (V_p, E_p)$, a data graph $G = (V, E)$ and flag.
Output: the result $Q_p(G)$.

1. **if** !flag **then** $Q'_p(V'_p, E'_p) := Q_p$;
2. **else** $Q'_p :=$ *Normalize*(Q_p); compute the distance matrix M;
   /* if the matrix is not yet available */
3. **for each** $u \in V'_p$ **do** $\mathsf{mat}(u) := \{x \mid x \in V,\ x \sim u\}$;
4. **for each** $e \in E'_p$ **do** $\mathsf{rmv}(e) := \emptyset$;
5. $Q_s := \mathsf{Sccgraph}(Q'_p)$;
6. **for each** C_s of Q_s in a reverse topological order **do**
7.   **do**
8.   **for each** edge $e = (u', u) \in E'_p$ where $u \in C_s$ **do**
9. $\mathsf{rmv}(e) := \mathsf{Join}(e, \mathsf{mat}(u'), \mathsf{mat}(u))$;
10. $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \mathsf{rmv}(e)$;
11.   **if** $\mathsf{mat}(u') = \emptyset$ **return** $\emptyset$;
12.   **for each** $e' = (u'', u') \in E'_p$ **do**
13. $\mathsf{rmv}(e') := \mathsf{rmv}(e') \cup \mathsf{Join}(e', \mathsf{mat}(u''), \mathsf{mat}(u'))$;
14.   **while** there exists $e = (u', u) \in E'_p$ *s.t.* $u \in C_s$ **and** $\mathsf{rmv}(e) \neq \emptyset$;
15.   **for each** edge $e = (u', u) \in E_p$ *s.t.* $u \in C_s$ **do**
16. $S_e := \{(x', x) \mid x' \in \mathsf{mat}(u'),\ x \in \mathsf{mat}(u)$ **and** $(x', x) \approx f_e(e)\}$;
17. **return** $Q_p(G) := \{(e, S_e) \mid e \in E_p\}$.

Procedure Join
Input: edge $e = (u', u) \in E_p$, $\mathsf{mat}(u')$, $\mathsf{mat}(u)$.
Output: $\mathsf{premv}(e)$ (a set of nodes that cannot match u').

1. $\mathsf{premv}(e) := \emptyset$;
2. **for each** $x' \in \mathsf{mat}(u')$ **do**
3.   **if** there does not exist $x \in \mathsf{mat}(u)$ *s.t.* $(x', x) \approx f_e(e)$ **do**
4. $\mathsf{premv}(e) := \mathsf{premv}(e) \cup \{x'\}$;
5. **return** $\mathsf{premv}(e)$;

Fig. 7. Algorithm JoinMatch

returned by Join are maintained in $\mathsf{rmv}(e)$ (line 9), which is then used to refine $\mathsf{mat}(u')$ (line 10). If the match set of any query node is empty, an empty result is returned (line 11) and the algorithm terminates. Otherwise, the $\mathsf{rmv}()$ sets of edges (u'', u') are checked for possible expansion due to nodes that cannot match u' (lines 12-13). The query result is finally collected (lines 15-16) and returned (line 17).

Procedure Join identifies nodes in $\mathsf{mat}(u')$ that do not satisfy the edge constraint imposed by $e = (u', u)$ or the match set $\mathsf{mat}(u)$. It examines each node x' in $\mathsf{mat}(u')$ (line 2). If there exists no node x in $\mathsf{mat}(u)$ such that (x', x) matches the regular expression $f_e(u', u)$ (line 3), x' is pruned from $\mathsf{mat}(u')$ and is recorded in $\mathsf{premv}(e)$ (line 4). The algorithm returns $\mathsf{premv}(e)$ (line 5). Note that if a distance matrix is used (when flag is true), one can check $(x', x) \approx f_e(e)$ (line 3) in constant time, for any edge color and wildcard. Otherwise we use bi-directional search to check the condition (Section IV).

Note that we provide the following options to handle regular expressions. (1) If a distance matrix M is available, a regular expression is decomposed into a set of simpler regular expressions, each containing a single color, to leverage M. (2) Otherwise, the regular expressions are evaluated straightforwardly using *bi-directional search* (see Section IV).

Example 9: Recall Q_2 and the data graph G from Fig. 1. We show how JoinMatch evaluates Q_2 on G. For each node u in Q_2, the initial and final match sets are as follows:

node	initial mat()	final mat()
B	$\{B_1, B_2\}$	$\{B_1, B_2\}$
C	$\{C_1, C_2, C_3\}$	$\{C_3\}$
D	$\{D_1\}$	$\{D_1\}$

In a reversed topological order (lines 6-14), JoinMatch repeatedly removes from $\mathsf{mat}()$ those nodes that do not make a match, by using $\mathsf{premv}()$ from procedure Join. There are two SCCs: SCC_1 and SCC_2, consisting of nodes $\{D\}$ and $\{B, C\}$, respectively. JoinMatch starts from node D and processes edge (C, D). The node C_1 is removed from $\mathsf{mat}(C)$, since it cannot reach D_1 within two hops colored fa followed by edges within two hops colored sa. When processing the edge (B, D), no nodes in $\mathsf{mat}(B)$ can be pruned. In SCC_2, the match sets $\mathsf{mat}(B)$ and $\mathsf{mat}(C)$ are refined by recursively using the edges (B, C), (C, B) and (C, C), and C_2 is removed from $\mathsf{mat}(C)$ as C_2 cannot reach any node in $\mathsf{mat}(B)$ with 1 hop colored fn. The result $Q_2(G)$ found is the same as in Example 4. □

Correctness & complexity. The algorithm returns $Q_p(G)$. Indeed, one can verify that for any query edge e, after the **for** loops (lines 6-16), each node pair in S_e is a match of e, and the result (e, S_e) is complete. The algorithm takes $O(m|V||E| + |E'_p||V|^2)$ time when a distance matrix is used, where m is the number of distinct edge colors and is typically small in practice. When the distance matrix is not available, it can be computed in $O((m+1)|V|^2 + |V|(|V| + |E|))$ time (line 2). Putting these together, the algorithm runs in $O(|V|^3)$ time.

B. Split-based Algorithm

We next present the split-based algorithm. It treats query nodes and data graph nodes uniformly, grouped into "blocks", such that each block B contains a set of nodes in $V \cup V_p$ from a data graph $G = (V, E)$ and a PQ $Q_p = (V_p, E_p)$. The algorithm creates a block for each query node u, denoted as $\mathsf{B}(u)$, initialized with all nodes $x \in V$ such that $x \sim u_i$. It then computes a *partition-relation pair* $\langle \mathsf{par}, \mathsf{rel} \rangle$, where par is set of blocks and rel a partial order on par. The pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ is recursively refined by *splitting* the blocks in par and rel using the constraints imposed by query edges. The process proceeds until a fixpoint is reached. The result of Q_p is then collected from the corresponding blocks of query nodes in V_p and the partial order on the blocks in rel.

The idea of split was first explored in LTS verification [26]. Our algorithm extends the algorithm of [26] in the following: (1) in contrast to [26] that works on edge-edge matching in one graph, our algorithm finds *edge-path* matching specified with *regular expressions*, *across two graphs* (a pattern graph and a data graph) with different models; and (2) it also develops nontrivial tricks to achieve the cubic-time complexity.

Algorithm. The algorithm SplitMatch is shown in Fig. 8. It also needs $\mathsf{mat}()$ and $\mathsf{rmv}()$ used by JoinMatch.

The algorithm first checks flag, and accordingly normalizes the query Q_p and computes the distance matrix if needed (lines 1-3), along the same lines as JoinMatch. It then initializes the match set and block set of each query node (line 5). In addition, it constructs the partition-relation pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ (line 6); it also initializes $\mathsf{rmv}()$ for each query edge (line 7), a step similar to its counterpart in JoinMatch. It then iteratively selects and processes those query edges with a nonempty remove set, *i.e.,* edges for which the match set can be refined (lines 8-14). The set of blocks par is split based on $\mathsf{rmv}(e)$

Input: a PQ $Q_p = (V_p, E_p)$, a data graph $G = (V, E)$ and flag.
Output: the result $Q_p(G)$.

1. par := $\emptyset$; rel := $\emptyset$;
2. **if** !flag **then** $Q'_p(V'_p, E'_p) := Q_p$;
3. **else** $Q'_p :=$ *Normalize*(Q_p); compute the distance matrix M; /* if the matrix is not yet available */
4. **for each** $u \in V'_p$ **do**
5. mat(u) := $\{x \mid x \in V$ **and** $x \sim u\}$; B(u) := $\{u\} \cup$ mat(u);
6. par := par $\cup$ B(u); rel := rel $\cup \{($B(u), B(u)$)\}$;
7. **for each** $e = (u', u) \in E'_p$ **do** compute rmv(e);
8. **while** there exists $e = (u', u)$ where rmv(e) $\neq \emptyset$ **do**
9. rmv := rmv(e); rmv(e) := $\emptyset$;
10. Split(e, ⟨par, rel⟩, rmv);
11. **for each** B $\subseteq$ rmv **do** rel(B(u')) = rel(B(u')) $\setminus$ B;
12. **for each** $e' = (u'', u')$ and **each** B $\subseteq$ rmv **do**
13. **for each** $x'' \in$ B(u'') *s.t.* no $x' \in$ B(u'), $(x'', x') \approx f_e(e')$ **do**
14. rmv(e') = rmv(e') $\cup \{x''\}$;
15. **for each** $e = (u', u) \in E_p$ **do**
16. $S_e := \{(x', x) \mid x' \in V, x \in V$, B($x$) $\in$ rel(B(u)), B(x') $\in$ rel(B(u')) **and** $(x', x) \approx f_e(e)\}$;
17. **if** $S_e = \emptyset$ **then return** $\emptyset$;
18. **return** $Q_p(G) := \{(e, S_e) \mid e \in E_p\}$.

Procedure Split
Input: edge $e = (u', u) \in E'_p$, pair ⟨par, rel⟩, a node set SpltN $\subseteq V$.
Output: updated pair ⟨par, rel⟩.

1. **for each** B $\in$ par **do**
2. B$_1$:= B $\cap$ SpltN; B$_2$:= B $\setminus$ SpltN;
3. par := par $\cup$ {B$_1$} $\cup$ {B$_2$}; par := par $\setminus$ {B};
4. rel(B$_1$) := rel(B$_2$) := {B$_1$, B$_2$};
5. **return** ⟨par, rel⟩;

Fig. 8. Algorithm SplitMatch

in procedure Split, and rel is updated accordingly (line 10). SplitMatch further extends the remove sets of edges $e'(u'', u')$ by checking if any node in mat(u'') has no descendants satisfying the constraints of e' (lines 12-14). If extended, the rmv(e') will be used to refine par.

The process (lines 8-14) iterates until par can no longer be split. The result is collected (line 16) and returned (line 18). SplitMatch terminates and returns an empty set, if the match set of any query edge is empty (line 17).

Procedure Split refines pair ⟨par, rel⟩ when given a set of nodes SpltN $\subseteq V$. Each block B $\in$ par is replaced by two blocks B$_1$ = B $\cap$ SpltN and B$_2$ = B $\setminus$ SpltN (line 2). Since B is split and new blocks are generated, par and rel are updated (lines 3-4), and the refined pair ⟨par, rel⟩ is returned (line 5).

Example 10: We show how SplitMatch evaluates the PQ Q_2 on the graph G of Fig. 1. For each node u in Q_2, SplitMatch initializes par, the set of blocks (Blks) as shown in the table below, together with the relation rel on the blocks. We also show the rmv() set of each edge, with empty rmv() omitted.

initial par	**initial** rel	**edge**	rmv() **sets**
Blk$_1$: $\{B, B_1, B_2\}$	{Blk$_1$, Blk$_1$}	(C, B)	$\{C_1, C_2\}$
Blk$_2$: $\{C, C_1, C_2, C_3\}$	{Blk$_2$, Blk$_2$}		
Blk$_3$: $\{D, D_1\}$	{Blk$_3$, Blk$_3$}		

After the process of SplitMatch, the final par and rel are shown in the following table. All the rmv() sets for query edges are $\emptyset$. One can verify that during the **while** loop (lines 8-14), the block set of node C is refined by using rmv(C, B), getting a new block set from which nodes C_1 and C_2 are absent. The other blocks are refined similarly.

final par	**final** rel
Blk$_1$: $\{B, B_1, B_2\}$	{Blk$_1$, Blk$_1$}
Blk$_2$: $\{C, C_3\}$	{Blk$_2$, Blk$_2$}
Blk$_4$: $\{C_1, C_2\}$	{Blk$_4$, Blk$_2$}, {Blk$_4$, Blk$_4$}
Blk$_3$: $\{D, D_1\}$	{Blk$_3$, Blk$_3$}

It finds the same result as reported in Example 4. □

Correctness & complexity. The algorithm returns $Q_p(G)$, since (1) all blocks are initialized with query nodes and all their possible matches; (2) the loop (lines 8-14) only drops those nodes that fail to match query nodes constrained by the query edges; (3) each graph node remaining in a block is a match to the corresponding query node, and (4) the size of each block decreases monotonically.

The algorithm takes $O(|\text{par}_{out}||V|^2)$ time in the worst case, when the distance matrix is used. Indeed, SplitMatch consists of three phases: pre-processing (lines 1-7), match computation (lines 8-14), and result collection (lines 15-18), which are in time $O((m+1)|V|^2 + |V|(|V| + |E|) + |V'_p||V| + |E'_p||V|^2)$, $O(|\text{par}_{out}||V|^2)$ and $O(|E'_p||V|^2))$, respectively. Observe that $|\text{par}_{out}|$ is bounded by $O(|V||V'_p|)$ and $|V'_p| \ll |V|$ in practice. Hence SplitMatch is in $O(|V|^3)$ time.

VI. Experimental Evaluation

We next present an experimental study using both real-life and synthetic data. Five sets of experiments were conducted, to evaluate: (1) the effectiveness of PQs, compared with a subgraph isomorphism algorithm SubIso [32] and a simulation based pattern matching algorithm Match [14]; (2) the effectiveness of minimization as an optimization strategy; (3) the efficiency of RQ evaluation; (4) the efficiency of algorithms JoinMatch and SplitMatch, employing distance matrix and distance cache as indices; and (5) the scalability of algorithms JoinMatch and SplitMatch.

Experimental setting. We used real-life data to evaluate the performance of our methods in real world, and synthetic data to vary graph characteristics, for an in-depth analysis.

(1) Real-life data. We used two sets of real-life data as follows: (a) *YouTube* dataset with 8350 nodes and 30391 edges, where each node denotes a video with attributes such as uploader (uid), category (cat), length (len), comment number (com) and age (the number of days since uploaded); edges between videos represent relationships such as friends recommendation fc (resp. reference fr) from earlier (resp. later) videos to later (resp. earlier) related ones, while their uploaders are friends; edge relationships also include strangers recommendation sc and reference sr defined similarly; (b) a terrorist organization collaboration network, from 81800 worldwide terrorist attack events in the last 40 years recorded in *Global Terrorism Database* [1], where each node represents a terrorist organization (*TOs*) with attributes such as name (gn), country, target type (tt), and attack type (at); and edges bear relationships, *e.g.,* international (resp. domestic) collaborations ic (resp. dc), from organizations to the ones they assisted or collaborated in the same country (resp. different countries). The network has 818 nodes and 1600 edges.

(a) Real-life result of PQs: *Youtube* and *Terrorist Organization*

(b) Effectiveness comparison

(c) Efficiency comparison

(d) Query minimization

(e) RQs over *Youtube*

(f) Varying $|V_p|$ on *YouTube*

(g) Varying $|E_p|$ on *YouTube*

(h) Varying $|\mathsf{pred}|$ on *YouTube*

(i) Varying b on *YouTube*

(j) Synthetic $G(|V|, 20K)$

(k) Synthetic $G(8K, |E|)$

(l) Varying b on Synthetic Data

(m) Varying $|V|$ $(\times 10^5)$

(n) Varying α

(o) Varying cr $(\times 10^{-3})$

Fig. 9. Performance Evaluation

(2) Query generator. We designed a query generator to produce meaningful PQs. The generator has five parameters: $|V_p|$ for the number of pattern nodes, $|E_p|$ the number of pattern edges, $|\mathsf{pred}|$ the number of predicates each pattern node carries, and bounds b and c such that each edge is constrained by a regular expression $e_1^{\leq b} \dots e_k^{\leq b}$, with $1 \leq k \leq c$. An RQ is a special case of a PQ as remarked earlier.

(3) Synthetic data. We implemented a generator to produce data graphs, controlled by 4 parameters: the number of nodes $|V|$, the number of edges $|E|$, the average number of attributes associated with a node, and a set Σ of edge colors that an edge may carry. The size of synthetic graphs scales from $1K$ nodes, $20K$ edges to $1M$ nodes, $4M$ edges.

(4) Implementation. We have implemented the following in Java: (a) the bi-directional search based method (BI-BFS) for RQs, with a distance cache employing hashmap to index frequently asked items; (b) algorithms JoinMatch and SplitMatch with distance matrix as indices, denoted as $\mathsf{JoinMatch_M}$ and $\mathsf{SplitMatch_M}$, respectively; (c) algorithms JoinMatch and SplitMatch using distance cache, denoted as $\mathsf{JoinMatch_C}$ and $\mathsf{SplitMatch_C}$, respectively; (d) SubIso, a subgraph isomorphism algorithm [32]; and (e) Match, a simulation based pattern matching algorithm [14].

All experiments were run on a machine with an AMD Athlon 64×2 Dual Core 2.30GHz CPU and 4GB of memory, using Scientific Linux. For each experiment, 20 patterns were generated and tested. The average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness of PQs**.** In this set of experiments, we evaluated the effectiveness of PQs. In contrast with SubIso and Match, we show that PQs can identify meaningful matches in real-life data. For quantitative comparison, the F-Measure [35] is adopted, which is defined as follows:

$$\mathsf{F\text{-}Measure} = 2 \cdot (\mathsf{recall} \cdot \mathsf{precision}) \;/\; (\mathsf{recall} + \mathsf{precision})$$
$$\mathsf{recall} = \#\mathsf{true_matches_found} \;/\; \#\mathsf{true_matches}$$
$$\mathsf{precision} = \#\mathsf{true_matches_found} \;/\; \#\mathsf{matches}$$

Here $\#\mathsf{matches}$ is the number of distinct node pairs (u, v), where u is a query node and v is a graph node that matches u; and $\#\mathsf{true_matches}$ is the number of meaningful results, *i.e.,* matches satisfying constraints on nodes and edges.

Figure 9(a) depicts two real-life PQs Q_1 and Q_2. Query Q_1 finds the videos A in the category "Film & Animation", having more than 20 comments and being uploaded more than 300 days. Videos A are related to videos B uploaded by "Davedays" via friends references (fr) or friends recommendations (fc), which in turn are related to videos C via constraint $\mathsf{sr}^{\leq 6}\mathsf{fr}$. Moreover, B and C both reference videos D, which are viewed over $160K$ times having less than 300 comments. Similarly, query Q_2 poses a request on a terrorist network searching for *TOs* related with a specified *TO* "Hamas" via various relations *e.g.,* $\mathsf{ic}^{\leq 2}\mathsf{dc}^{+}\mathsf{ic}^{\leq 2}$.

Partial results of Q_1 and Q_2 are drawn in Fig. 9(a). Interestingly, the result of Q_2 reflects some (indirect) connections from different *TOs* to the *Hamas TO* in the middle east. Existing approaches *e.g.,* SubIso and Match, are not sufficient

to express such queries. For a fair comparison, we allow different edge colors in a data graph but restrict the color constrained by a query edge of 1, to favor SubIso and Match.

Figure 9(b) shows the F-Measure values of different approaches for various such queries. The pair $(|V_p|, |E_p|)$ in the x-axis denotes the number of nodes $|V_p|$ and edges $|E_p|$ in a query. The y-axis represents the F-Measure values. The number of predicates at each query node is 2 or 3. The result shows that (1) PQs can always find meaningful matches, as expected; (2) SubIso has low F-Measure, *e.g.,* SubIso found 33 true matches among 245 when the x-value is $(3, 3)$. This is mainly due to its low recalls. For the other queries, SubIso cannot find any match. Its precision is always 1 if some matches can be identified. (3) The F-Measure of Match is better than that of SubIso. This is because its recall is high, *i.e.,* it can identify all true matches. However, its precision is relatively low, *e.g.,* among the 374 matches found by Match when the x-value is $(3, 3)$, only 245 are true matches.

Figure 9(c) reports the elapsed time of all the algorithms, using *Terrorism* data. The matrix-based methods were employed, *i.e.,* $\mathsf{SplitMatch_M}$, $\mathsf{JoinMatch_M}$ and $\mathsf{Match_M}$. It shows that $\mathsf{JoinMatch_M}$ and $\mathsf{SplitMatch_M}$ outperform $\mathsf{Match_M}$, and are much faster than SubIso.

These results us tell that PQs are not only more effective, but also more efficient than its conventional counterparts.

Exp-2: The effectiveness of PQ **minimization**. We evaluated the effectiveness of the minimization algorithm minPQs (Section III), using *YouTube* data. The queries were generated by varying $|V_p|$ and $|E_p|$. The average number of predicates |pred| is 3. Here c is between 2 and 4, and $b = 5$, *i.e.,* each edge is constrained by the expression $c_1^{\leq 5} \ldots c_k^{\leq 5}$, where $2 \leq k \leq 4$.

The results are reported in Fig. 9(d). In Fig. 9(d), the x-axis is the same as its counterparts in Fig. 9(b), and the y-axis represents the elapsed time for query evaluation. For space limitation, we only show the results of using the algorithm $\mathsf{JoinMatch_M}$, the others reflect similar trend and are thus omitted. The minimization process was performed instantly. The results tell us the following: (1) minPQs can reduce the size of queries and thus speed up the query evaluation; and (2) generally, the larger the queries are, the better the performance can be improved. This is because larger queries have a higher probability to contain redundant nodes and edges. Indeed, it took 18 seconds to handle queries with 12 nodes and 18 edges, while the running time was cut by over a half for the minimized queries, which have 7 nodes and 9 edges in average.

This set of experiments verified that the minimization algorithm can effectively optimize PQs. In the rest of experiments, all tested queries were minimized.

Exp-3: Efficiency of RQs**.** In this set of experiment, we tested the efficiency of the two algorithms presented in Section IV for evaluating RQs. Fixing the bound b at 5 and the cardinality of node predicates at 3, we varied the number of colors c from 1 to 4 per edge. More specifically, the tested regular expressions have the form $c_1{}^{\leq b} \ldots c_i{}^{\leq b}$ $(i \in [1, 4])$.

Figure 9(e) shows the average elapsed time of evaluating RQs on *YouTube* data. The x-axis represents the number of distinct colors and y-axis the elapsed time. The term DM means the method employing distance matrix. The results tell us the following. (1) The method based on distance matrix is most efficient, and BI-BFS is more efficient than BFS, as expected. (2) BI-BFS scales better than BFS with the number of colors c, since by searching from two directions, BI-BFS produces less intermediate nodes than BFS. The trend of the curves of BI-BFS and BFS indicates that BI-BFS works better for more complex regular expressions. (3) As maintaining distance matrix is costly for large graphs, BI-BFS makes a rational solution by balancing the tradeoff between time and space.

Exp-4: Efficiency of PQs **on *YouTube***. This set of experiments varied the parameters $|V_p|$, $|E_p|$, |pred|, c and b, whose default values are 6, 8, 3, 4 and 5, respectively.

Figures 9(f), 9(g), 9(h) and 9(i) depict the elapsed time when varying one of the parameters: $|V_p|$, $|E_p|$, |pred| and b, respectively. See Fig. 9(e) for the tests for varying c. The M-index represents the time of computing a distance matrix, which is shared by all patterns and thus is not counted in the algorithms $\mathsf{JoinMatch_M}$ and $\mathsf{SplitMatch_M}$.

Observe the following about these experimental results:

(1) Figure 9(f) shows that the matrix-based algorithms $\mathsf{JoinMatch_M}$ and $\mathsf{SplitMatch_M}$ outperforms the distance-cache based $\mathsf{JoinMatch_C}$ and $\mathsf{SplitMatch_C}$, respectively, since the former answers node distance in constant time, while the latter needs to compute it from scratch if the result is not cached.

(2) The join-based methods outperform the split-based methods. As shown in the figures with various parameters, in most cases $\mathsf{JoinMatch_M}$ is the fastest, followed by $\mathsf{SplitMatch_M}$; and $\mathsf{JoinMatch_C}$ outperforms $\mathsf{SplitMatch_C}$. This indicates that the computational cost of the join-based method is reduced by adopting the reverse topological order (see Section V).

(3) The elapsed time is more sensitive to the number of pattern edges (see Fig. 9(g)) than pattern nodes (see Fig. 9(f)), since the number of edges dominates the number of joins or splits to be conducted. Moreover, the elapsed time is sensitive to the number of predicates (see Fig. 9(h)) since predicates impose a strong constraint in initializing the match set. The more the predicates, the less graph nodes will satisfy them, resulting in smaller candidate matches and faster evaluation. The time is sensitive to the bound (see Fig. 9(i)) since the number of matches gets larger when b is increased.

(4) From these figures, we can expect that all algorithms have good scalability and they will work well when the numbers of $|V_p|, |E_p|$ |pred| and b become much larger.

(5) The M-index can be computed efficiently, and it improves the performance, when the dataset is relatively small.

Exp-5: Scalability of PQs **on synthetic data**. In the last set of experiments, we evaluated the scalability of both algorithms over (large) synthetic data. The default values of $|V_p|$, $|E_p|$, c, |pred| and b are 6, 8, 4, 3 and 5, respectively.

(1) We first tested both distance-cache based and matrix-based algorithms *w.r.t.* $|V|$ and $|E|$ of data graphs with default values

$8K$ and $20K$, respectively. Figures 9(j) and 9(k) show that all algorithms scale well with $|V|$ and $|E|$, respectively.
(2) We then tested the distance-cache based algorithms on large data graphs since the matrix-based algorithms do not work due to their high space overhead. Two additional parameters are used: (a) candidate rate (cr) such that the number of matches of a pattern node is bounded by $|V| \times \mathsf{cr}$, and (b) the density α of data graphs such that $|E| = |V|^\alpha$. The default values of $|V|$, cr and α are $50K$, 0.01 and 1.1, respectively.

Figures 9(l), 9(m), 9(n) and 9(o) show that (a) the distance-cache based algorithms scale well with b, $|V|$, α and cr, respectively; (b) they are sensitive to all these parameters; and (c) JoinMatch consistently outperforms SplitMatch.

Summary. We have the following findings. (1) PQs are able to identify far more sensible matches in emerging application than the conventional approaches can find. (2) The minimization algorithm can effectively identify and remove redundant nodes and edges, and thus can improve performance for query answering. (3) With distance matrix as indices, the evaluation of RQs is very efficient. Moreover, BI-BFS is rational when working on large graphs. (4) PQs can be efficiently evaluated, and the distance-cache based algorithms scale well even with large graphs with $1M$ nodes and $4M$ edges.

VII. Conclusion

We have proposed extensions of reachability queries (RQs) and graph pattern queries (PQs), by incorporating a subclass of regular expressions to capture edge relationships commonly found in emerging applications. We have revised graph pattern matching by introducing an extension of graph simulation. We have also settled fundamental problems (containment, equivalence, minimization) for these queries, all in low PTIME. In addition, we have shown that the increased expressive power does not incur higher evaluation complexity. Indeed, we have provided two algorithms for evaluating RQs, one in *quadratic time*, the same as their traditional counterparts [21]. We have also developed two *cubic-time* algorithms for evaluating PQs, as opposed to the intractability of graph pattern matching via subgraph isomorphism. We have verified experimentally that these queries are able to find more sensible information than their traditional counterparts, and that the algorithms are efficient when evaluating RQs and PQs on large graphs.

One topic for future work is to extend RQs and PQs by supporting general regular expressions. Nevertheless, with this comes increased complexity. Indeed, the containment and minimization problems become PSPACE-complete even for RQs. Another topic is to identify application domains in which simulation-based PQs are most effective. A third topic is to study incremental algorithms for evaluating RQs and PQs.

Acknowledgment

Fan, Ma and and Wu are supported in part by EPSRC E029213/1. Li is supported in part by the Key Program of NSFC 61033015.

References

[1] Terrorist organization network. *http://www.start.umd.edu/ gtd.*
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
[4] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.
[5] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, 2010.
[6] M. J. Brzozowski, T. Hogg, and G. Szabó. Friends and foes: ideological social networking. In *CHI*, 2008.
[7] P. Buneman, M. F. Fernandez, and D. Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.
[8] D. Bustan and O. Grumberg. Simulation-based minimization. *TOCL*, 4(2):181–206, 2003.
[9] D. Chen and C. Y. Chan. Minimization of tree pattern queries with constraints. In *SIGMOD*, 2008.
[10] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, 2009.
[11] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated Web collections. *SIGMOD Rec.*, 29(2), 2000.
[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5):1338–1355, 2003.
[13] W. Fan and P. Bohannon. Information preserving XML schema embedding. *TODS*, 33(1), 2008.
[14] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.
[15] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. In *PVLDB*, 2010.
[16] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, 1998.
[17] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.
[18] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2009.
[19] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[20] T. Jiang and B. Ravikumar. Minimal NFA Problems are Hard. *SICOMP*, 22(6):1117–1141, 1993.
[21] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, 2010.
[22] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.
[23] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27:415–444, 2001.
[24] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.
[25] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
[26] F. Ranzato and F. Tapparo. A new efficient simulation equivalence algorithm. In *LICS*, 2007.
[27] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, 2009.
[28] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
[29] SPARQL. *http://www.w3.org/TR/rdf-sparql-query/.*
[30] R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.
[31] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
[32] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.
[33] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.
[34] F. Wei. TEDI: Efficient shortest path query answering on graphs. In *SIGMOD*, 2010.
[35] Wikipedia. F-measure. *http://en.wikipedia.org/wiki/F-measure.*
[36] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.
[37] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. In *PVLDB*, 2009.