Front. Comput. Sci. China, 201X, X(X)
DOI

RESEARCH ARTICLE

Adding Regular Expressions to Graph Reachability and Pattern Queries

Wenfei, Fan[1,2] and Jianzhong, Li[2] and Shuai, Ma[3] and Nan, Tang[4] and Yinghui, Wu[1]

1 University of Edinburgh, Edinburgh, UK
2 Harbin Institute of Technology, Harbin, China
3 NLSDE Lab, Beihang University, Beijing, China
4 Qatar Computing Research Institute, Qatar Foundation, Qatar

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, as opposed to the NP-completeness of graph pattern matching via subgraph isomorphism. (4) The effectiveness and efficiency of these algorithms are experimentally verified using real-life data and synthetic data.

1 Introduction

It is increasingly common to find data modeled as graphs in a variety of areas, *e.g.,* computer vision, knowledge discovery, biology, chem-informatics, dynamic 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 [4, 17, 28–30, 44].
(b) Graph pattern queries, to find all subgraphs of a data graph that are isomorphic to a pattern graph [8, 14, 16, 42, 48] (see [24] for a survey).

In emerging applications such as social networks, edges in a graph are typically "typed", denoting various relationships such as marriage, friendship, work, advice, support, exchange, co-membership, etc [33]. In practice one often wants to query the connectivity of a pair of nodes 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 [9].

Example 1.1: Consider an *Essembly* network service [9], 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 that 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 whom she agrees with on most topics they vote; and (4) *strangers-nemeses (sn)*,

Fig. 1 Querying Essembly Network

Received month dd.yyyy; accepted month dd.yyyy

E-mail: {wenfei@inf., y.wu-18@sms.}ed.ac.uk
lijzh@hit.edu.cn, contact author
mashuai@act.buaa.edu.cn, contact author
ntang@qf.org.qa

from a user to a stranger with whom she disagrees on most topics they both vote.

Figure 1 depicts a part of the network as a graph G that involves a debate on cloning research. In the graph 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, which are 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 towards cloning (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 various 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) with edges of particular *types* and *patterns*, and
(b) with a *bound* on its *length* (hops).

In other words, 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 the edge (C, D) in Q_2 to the path $C_3 \xrightarrow{\mathsf{fa}} C_1 \xrightarrow{\mathsf{sa}} D_1$ in G.

That is, traditional pattern queries defined in terms of subgraph isomorphism are insufficient to express Q_2. □

As suggested by the example, emerging applications highlight the need for revising the traditional reachability queries and graph pattern queries to incorporate edge types and bounds on the number of hops. In addition, it is necessary to revise the notion of 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 find matches in large data graphs.

Contributions & Roadmap. To this end we propose a class of reachability queries, as well as 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 2. In such a query, each node specifies search conditions on the content of the graph nodes, and an 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. In addition, we define pattern matching by extending graph simulation [26], instead of using subgraph isomorphism. For instance, queries 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 3), along the same lines as for XML tree pattern queries [35, 47]. 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 [36]. As an immediate application, we develop an algorithm in $O(n^3)$ time to minimize PQs, which yields an effective optimization strategy.

(3) We develop two algorithms to answer RQs (Section 4). One employs a matrix of shortest distances between nodes. It is in *quadratic time*, the same as its traditional counterpart [44]. That is, the increased expressive power of RQs does not incur 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 5), 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 6). 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. This work extends [19] by including detailed proofs of the fundamental problems in connection with (1) the uniqueness of graph pattern query answers (Section 2), *i.e.,* graph pattern queries are well defined; and (2) the containment, equivalence and minimization problems of graph reachability queries and graph pattern queries (Section 3). (3) A detailed algorithm for pattern query minimization is also included (Section 3).

The idea of using regular expressions to query graphs is not new: it has been adopted by query languages for semistructured data such as UnQL [10] and Lorel [3]. There has also been theoretical work on conjunctive regular path queries CRPQs (*e.g.,* [23]) and recently on extended CRPQs (ECRPQs) [7], 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 [7]. 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 [25], SoQL [38] and SPARQL [40]. 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 [40] is a query language tailored for RDF graphs coded as a set of triples (subject, predicate and object). Queries on graphs with labeled directed or undirected edges and label or unlabeled nodes have also been studied [32]. These languages adopt subgraph isomorphism for graph pattern search, which differs from this work, among other things.

A number of algorithms have been developed for evaluating reachability queries [17, 29, 44]. 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, 17, 45] 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 [28], which ask 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.

Graph pattern matching is typically defined in terms of subgraph isomorphism [8, 14, 16, 42, 48] (see [24, 39] for surveys). Extensions of subgraph isomorphism are studied in [18,21,48], which extend mappings from edge-to-edge to edge-to-path. Nevertheless, the problem remains NP-complete. Closer to this work is the notion of bounded simulation studied in [20], which extends graph simulation [11, 26] for graph pattern matching by allowing bounds on the number of hops, and makes graph pattern matching a PTIME problem. This work further extends [20] by incorporating regular expressions as edge constraints, and for these more expressive graph queries, it develops efficient evaluation algorithms and settles their fundamental problems for containment, equivalence and minimization, which are important for query optimizations. 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.,* [13,35,47]). However, we are not aware of previous work on these problems for graph pattern queries.

There has also been a host of work on structural indices [31, 34] for evaluating regular expression queries. Unfortunately, the indexing structures are developed for tree-structured data (XML) in which there is a unique path between two nodes; they cannot be directly used when processing general graphs.

2 Graph Reachability and Pattern Queries

In this section, we start with data graphs, and then introduce reachability queries (RQs) and graph pattern queries (PQs) on data graphs.

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, ratings [5]; 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 [33].

Example 2.1: 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 $f_A(v)$, where (a) $f_A(B_i) = (\mathsf{job} =$ "doctor", $\mathsf{dsp} =$ "cloning") for $i \in [1, 2]$, (b) $f_A(C_j) = (\mathsf{job} =$ "biologist", $\mathsf{sp} =$ "cloning") for $j \in [1, 3]$, (c) $f_A(D_1) = (\mathsf{uid} =$ "Alice001"), and (d) $f_A(H_1) = (\mathsf{job} =$ "physician"). □

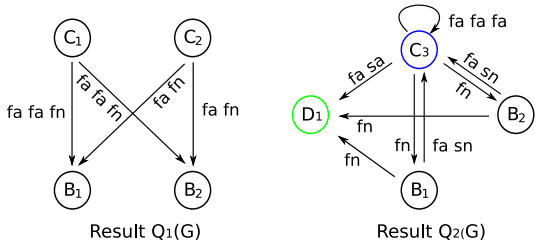


Fig. 2 Results of the queries Q_1 and Q_2 on G

We shall use the following notations.
(1) A *path* ρ in G is denoted as $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots v_{n-1} \xrightarrow{e_n} v_n$, where (a) $v_i \in V$ for each $i \in [0, n]$, and (b) $e_j = (v_j, v_{j+1})$ is 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 v_1 to v_2 in 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 $c_1 \cup \dots \cup c_m$, when $\Sigma = \{c_i \mid i \in [1, m]\}$; (2) k is a positive integer, and $c^{\leq k}$ denotes the regular expression $c^1 \cup c^2 \cup \dots \cup c^k$, where c^j ($j \in [1, k]$) denotes j occurrences of c; and (3) c^+ denotes one or more occurrences of c.

We shall use $L(f_e)$ to denote the regular language defined by f_e, *i.e.,* the set of all strings that can be parsed by the grammar 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 for query 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 $\rho = v_1 \xrightarrow{e_1} v_1' \xrightarrow{e_2} v_2' \dots v_{n-1}' \xrightarrow{e_n} v_2$ in G such that the string $f_C(e_1) \dots f_C(e_n)$ 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) such that $v_1 \sim u_1$, $v_2 \sim u_2$, and $(v_1, v_2) \approx f_e$.

Intuitively, (v_1, v_2) is in $Q_r(G)$ if v_1 and v_2 satisfy the conditions specified by u_1 and u_2, respectively, 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. We say v_1 (resp. v_2) is a *match* of u_1 (resp. u_2).

Example 2.2: 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 the node B has the predicate job = "doctor".

When Q_1 is posed on the data graph G shown in Fig. 1 and described in Example 2.1, 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.,* [28, 29, 45], 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, in which (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 $e = (u, u') \in E_p$, $Q_r = (u, u', f_v(u), f_v(u'), f_e)$ is an RQ. In the rest part of this paper, we shall simply use f_e to represent the regular expression assigned by the function f_e to an edge e unless specified otherwise.

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

(1) for all edges $e = (u_1, u_2)$ in Q_p, $S_e \subseteq Q_e(G)$, where $Q_e = (u_1, u_2, f_v(u_1), f_v(u_2), f_e)$ 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. In other words, $Q_p(G) = \emptyset$ if for some e in Q_p, S_e is empty.

We say v_1 (resp. v_2) is a *match* of u_1 (resp. u_2). Here

the size of $Q_p(G)$ is defined as $\sum_{e\in E_p}|S_e|$, where $|S_e|$ is the number of elements in S_e.

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 $Q_e = (u_1, u_2, f_v(u_1), f_v(u_2), f_e)$ 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; and (3) (u_2, v_2) is also in R. In addition, R covers all the nodes in V_p and moreover, it is maximum, *i.e.,* for all such relation R', $R' \subseteq R$. The result $Q_p(G)$ is simply R grouped by edges in E_p. In particular, if condition (3) above is not satisfied, $Q_p(G)$ is empty.

From this one can see that PQs are defined in terms of an extension of graph simulation [26], 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 differs from the traditional notion of graph pattern matching defined in terms of subgraph isomorphism [24] and graph simulation [26].

Example 2.3: 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 the query Q_2 is posed on the data graph G of Fig. 1, the query 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.

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 a similar reason, (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 [20] is a special case of PQs, by only allowing patterns in which (a) there is only 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 2.1: *For any data graph G and any graph pattern query Q_p, there is a unique result $Q_p(G)$.* □

Proof: (i) We first show that there exists a query result. We consider all possible sets of $\{(e, S_e) \mid S_e$ is a set of node pairs in G for each edge e in $Q_p\}$, which satisfy conditions (1) and (2) of the semantics of PQs. Note that those sets are not necessarily maximum, and the number of such possible sets is finite.

We define the query result to be a set with the maximum size, which, as will be seen shortly, is unique. If there exists an edge e such that $S_e = \emptyset$ in the set, the query result is $\emptyset$ by condition (3) of the semantics of PQs.

(ii) We then show the uniqueness by contradiction. Assume that there exist two distinct maximum query results $Q_p^1(G)$ and $Q_p^2(G)$. We then show that there exists a result larger than both $Q_p^1(G)$ and $Q_p^2(G)$. Given two such sets $S^1 = \{(e, S_e^1) \mid e$ is an edge in $Q_p\}$ and $S^2 = \{(e, S_e^2) \mid e$ is an edge in $Q_p\}$, we define the union of S^1 and S^2 as $\{(e, S_e^1 \cup S_e^2) \mid e$ is an edge in $Q_p\}$, denoted by $S^1 \cup S^2$. Observe that Q_p^i is possibly empty when S_e^i is empty for some e, where $i \in [1, 2]$. Let $Q_p(G) = Q_p^1(G) \cup Q_p^2(G)$. By the definition of PQs, one can readily verify that $Q_p(G)$ is a query result larger than both $Q_p^1(G)$ and $Q_p^2(G)$. This contradicts the assumption that both $Q_p^1(G)$ and $Q_p^2(G)$ are maximum.

By (i) and (ii) above, we have the conclusion. □

3 Fundamental Graph Queries Problems

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).

3.1 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 query results of Q_1, Q_2 on G, respectively.

Intuitively, the mapping 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 $v \sim w_1$,

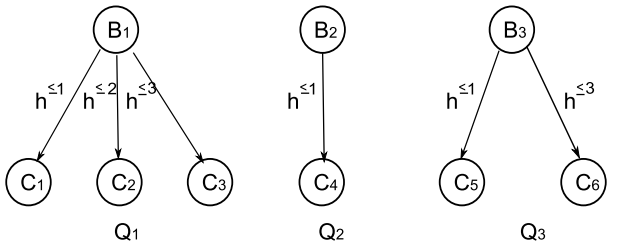


Fig. 3 Example for containment and equivalence

denoted as $u_1 \vdash w_1$; and (2) $u_2 \vdash w_2$. Moreover, (3) $L(f_e) \subseteq L(f_\lambda(e))$, denoted as $e \models \lambda(e)$.

Example 3.1: 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. Given two graph pattern queries 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 3.1.

For any PQs Q_1 and Q_2, 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 $\lambda_{2,1}$, respectively.

Complexity bounds. We next establish the complexity bounds of the containment and equivalence problems for PQs. We first present a revision of similarity [26].

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 $S_r \subseteq V_p^1 \times V_p^2$ such that

(1) for any $(u_1, w_1) \in S_r$, (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 $(u_2, w_2) \in S_r$ 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) $(u, w) \in S_r$, $(u', w') \in S_r$, and (b) $e' \models e$.

Example 3.2: Recall PQs Q_1 and Q_2 from Example 3.1. One can verify that $Q_1 \trianglelefteq Q_2$. Indeed, there exists a binary relation $S_r = \{(B_1, B_2), (C_1, C_4), (C_2, C_4), (C_3, C_4)\}$, which satisfies the conditions of the revised similarity given above:

(1) for each $(u, w) \in S_r$, $w \vdash u$ (the condition (1)(a));

(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)); 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)). □

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

Lemma 3.1: *Given two* PQs Q_1 *and* Q_2, $Q_1 \sqsubseteq Q_2$ *if and only if* Q_1 *is similar to* Q_2 (i.e., $Q_2 \trianglelefteq Q_1$). □

Proof: (1) Assume first $Q_1 \sqsubseteq Q_2$. We next show $Q_2 \trianglelefteq Q_1$ by proof by contradiction. Suppose that $Q_2 \not\trianglelefteq Q_1$, we construct a data graph G from Q_1 such that $Q_1 \not\sqsubseteq Q_2$, which contradicts the assumption.

Assume *w.l.o.g.* that $Q_1 = (V_p^1, E_p^1, f_v^1, f_e^1)$. The data graph $G(V, E, f_A, f_C)$ is constructed from Q_1 as follows. (a) For each node $u \in V_p^1$, create a node $u' \in V$ such that $f_A(u')$ satisfies $f_v^1(u)$, and (b) for each edge $(u_1, u_2) \in E_p^1$, create a path from nodes u_1' to u_2' in G, passing through only a set of dummy nodes satisfying $f_e^1(u_1, u_2)$. If $Q_2 \not\trianglelefteq Q_1$, either condition (1) or condition (2) of the revised similarity is violated. No matter which condition is violated, we can easily refine the data graph G given above so that $Q_2 \not\sqsubseteq Q_1$ when evaluated on G.

(2) Conversely assume that $Q_2 \trianglelefteq Q_1$. We next show $Q_1 \sqsubseteq Q_2$. Since $Q_2 \trianglelefteq Q_1$, there exists a similarity relation S_r from Q_2 to Q_1. By the definition of the revised similarity, we can readily construct a mapping λ from the edges in Q_1 to the edges in Q_2 based on S_r, and we then prove that the λ is indeed what we need.

Consider a data graph G. For any edge $e = (w_1, w_2)$ in Q_1 with $\lambda(e') = (u_1, u_2)$ in Q_2, we have the following: (a) for any graph node v, if $v \sim w_1$, then $v \sim u_1$ since $w_1 \vdash u_1$, and if $v \sim w_2$, then $v \sim u_2$ since $w_2 \vdash u_2$; and (b) by the semantics of PQs, for any $(v_1, v_2) \in S_e$, we can easily show that $(v_1, v_2) \in S_{\lambda(e')}$, where $(e, S_e) \in Q_1(G)$ and $(\lambda(e'), S_{\lambda(e')}) \in Q_2(G)$, *i.e.,* $S_e \subseteq S_{\lambda(e)}$. From this $Q_1 \sqsubseteq Q_2$ immediately follows.

From (1) and (2) above Lemma 3.1 follows. □

It is known that graph similarity is solvable in quadratic time [26]. Extending the techniques of [26] by leveraging Lemma 3.1, one can verify the following.

Theorem 3.2: *For* PQs Q_1 *and* Q_2, *it is in cubic time to determine whether* $Q_1 \sqsubseteq Q_2$ *and* $Q_1 \equiv Q_2$. □

To prove this, we first show the following for RQs,

which are a special case of PQs.

Proposition 3.3: *For* RQs Q_1 *and* Q_2*, it is in quadratic time to check whether* $Q_1 \sqsubseteq Q_2$ *and* $Q_1 \equiv Q_2$. □

Proof: Consider two RQs $Q_1 = (u_1, u_2, f_{u_1}, f_{u_2}, f_{e1})$ and $Q_2 = (w_1, w_2, f_{w_1}, f_{w_2}, f_{e2})$, where f_{u_1}, f_{u_2}, f_{w_1}, and f_{w_2} are *satisfiable*. It is easy to verify that $Q_1 \sqsubseteq Q_2$ if and only if $u_1 \vdash w_1$, $u_2 \vdash w_2$, and $L(f_{e1}) \subseteq L(f_{e2})$. Hence, it suffices to show the following. (1) testing $u_1 \vdash w_1$ can be done in $O(|f_{u_1}||f_{w_1}|)$ time; (2) testing $u_2 \vdash w_2$ can be done in $O(|f_{u_2}||f_{w_2}|)$ time; and (3) testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time. For if these hold, then one can check whether $Q_1 \sqsubseteq Q_2$ in quadratic time. Moreover, one can decide whether $Q_1 \equiv Q_2$ by inspecting whether $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$, both in quadratic time.

We next verify these one by one.

(1) We first show that testing $u_1 \vdash w_1$ can be done in $O(|f_{u_1}||f_{w_1}|)$ time.

Observe that $u_1 \vdash w_1$ if and only if each sub-formula A op a in f_{u_1} is implied by f_{w_1}. There are in total four cases to consider, based on the type of op.

Case (a). When op is $=$. We first find (i) the smallest value $a_<$ in f_{w_1} associated with the attribute A and the operator $<$; (ii) the smallest value $a_\leq$ in f_{w_1} associated with the attribute A and the operator $\leq$; (iii) the largest value $a_>$ in f_{w_1} associated with the attribute A and the operator $>$; and (iv) the largest value $a_\geq$ in f_{w_1} associated with the attribute A and the operator $\geq$.

If $a_\geq = a_\leq$, then A op a is implied by f_{w_1}. If not, it further checks whether $A = a$ appears in f_{w_1}. If 'yes', then $A = a$ is implied by f_{w_1}.

Case (b). When op is $\leq$. Again, it suffices to find the values $a_<$, $a_\leq$, $a_>$, $a_\geq$ and $a_=$. Then A op a is implied by f_{w_1} iff $a_< \leq a$, $a_\leq \leq a$ and $a_= \leq a$.

Case (c). When op is $<$, $\geq$ or $>$, it is similar to case (b).

Case (d). When op is $\neq$. Again, we find the values $a_<$, $a_\leq$, $a_>$, $a_\geq$ and $a_=$. Then A op a is implied by f_{w_1} iff $a_< > a$ and $a_\leq > a$, $a_> < a$ and $a_\geq < a$, $a_= \neq a$, or $A \neq a$ appears in f_{w_1}.

The checking takes $O(|f_{w_1}|)$ time in all these cases.

(2) Similar to (1), we can show that testing $u_2 \vdash w_2$ can be done in $O(|f_{u_2}||f_{w_2}|)$ time.

(3) Finally, we show that testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time. Note that we use a restricted form of regular expressions, as defined in Section 2. In such a regular expression F, we define the length of an atomic component c, $c^{\leq k}$ or c^+ to be 1. Hence, the length of F, denoted by $|F|$, is simply the number of its atomic components.

To determine whether $L(f_{e1}) \subseteq L(f_{e2})$, we sequentially scan f_{e1} and f_{e2} once. It is easy to verify that for any two regular expressions F_1 and F_2, if $L(F_1) \subseteq L(F_2)$, then $|F_1| = |F_2|$. It suffices to consider the following cases in the sequential scanning process:

Case (a). $L(c^{k_1}c^{k_2}\ldots c^{k_n}) \subseteq L(c^{k'_1}c^{k'_2}\ldots c^{k'_n})$, where $(k_1 + \ldots + k_n) \leq (k'_1 + \ldots + k'_n)$.

Case (b). $L(c_1^{k_1}c_2^{k_2}\ldots c_n^{k_n}) \subseteq L(c_1^{k'_1}c_2^{k'_2}\ldots c_n^{k'_n})$, where $(k_1 + \ldots + k_n) \leq (k'_1 + \ldots + k'_n)$, and, moreover, c_i is either c_i or _ for each $i \in [1, n]$.

Case (c). The $+$ operator is treated as an integer, but is larger than any positive integer k.

For each case above, it can be tested in linear time. Putting all these together, we conclude that testing $L(f_{e1}) \subseteq L(f_{e2})$ can be done in linear time. □

By using Proposition 3.3 and extending the algorithm for computing standard graph simulations [26], we are now ready to prove Theorem 3.2.

Proof of Theorem 3.2. It suffices to show that checking whether $Q_1 \sqsubseteq Q_2$ is in cubic time. We next develop an algorithm to test whether $Q_1 \sqsubseteq Q_2$, by testing whether Q_1 is similar to Q_2 (*i.e.,* $Q_2 \trianglelefteq Q_1$) based on Lemma 3.1. It consists of the following steps:

(i) First, determine whether $u \vdash w$ for all nodes u in Q_1 and all nodes w in Q_2. This is doable in quadratic time, as verified in the proof of Proposition 3.3.

(ii) Second, determine whether $e \models e'$ for all edges e in Q_1 and all edges e' in Q_2. This runs in quadratic time, as shown in the proof of Proposition 3.3.

(iii) Third, employ the algorithm for graph simulation in [26] to compute the maximum relation S_r from Q_2 to Q_1. The algorithm [26] runs in quadratic time.

(iv) Finally, test whether the relation S_r satisfies the *condition (2)* of the revised graph similarity. This can be done in cubic time, following from (ii) above.

The correctness of the above algorithm is guaranteed by Lemma 3.1, and in total it runs in cubic time. □

Remark. The equivalence problem for standard regular expressions is PSPACE-complete [27]. However, for the restricted regular expressions defined in Section 2, their equivalence problem is much simpler – it is in linear time. 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 analysis of fundamental properties.

3.2 Minimizing Graph Pattern Queries

A problem closely related to query equivalence is query minimization. As remarked earlier, query minimization often yields an effective optimization strategy. It has been studied for, *e.g.,* relational conjunctive queries [2] and XML tree pattern queries [13, 35, 47]. For all the reasons that query minimization is important for relational queries and XML queries, we also need to study

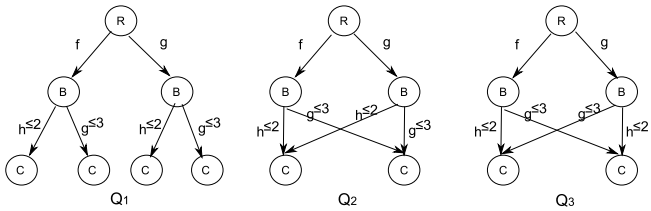


Fig. 4 Non-isomorphic equivalent minimum PQs

the 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 [13]. To simplify the discussion, we assume Q is *connected*.

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) no other such Q' has size $|Q'| < |Q_m|$. We refer to Q_m as a *minimum equivalent* PQ of Q.

Remark. (1) A PQ may have multiple minimum equivalent PQs. Moreover, these PQs may not be isomorphic to each other, although they have the same size. Figure 4 shows such an example, where both Q_2 and Q_3 are minimum equivalent PQs of Q_1 and $|Q_2| = |Q_3|$, 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 5, 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.

The minimization problem for RQs is trivial for the reason stated above. Below we focus on minimization of PQs. The last main result of the section is as follows.

Theorem 3.4: *Given any* PQ Q*, a minimum equivalent* PQ *of* Q *can be computed in cubic time.* □

To prove Theorem 3.4, we develop an algorithm that, given a pattern query Q as input, finds a minimum equivalent PQ Q_m of Q in cubic time.

To present the algorithm, we first introduce several notions that the algorithm uses. Recall the revised graph similarity relation S_r defined in Section 3.1. We say that two nodes u, w in Q are *simulation equivalent* if and only if $(u, w) \in S_r$ and $(w, u) \in S_r$. The equivalence relation S_{eq} consists of all the node pairs that are simulation equivalent. We denote the set of equivalence classes induced by S_{eq} as EQ, where each equivalence class in EQ is a set of nodes that are pairwise simulation equivalent.

Note that if u and v are simulation equivalent, then $u \vdash w$ and $w \vdash u$. Intuitively, this suggests that any two nodes in the same equivalence class should be treated as a single node for *any* queries. Based on this, the idea of minPQs is to (1) identify these equivalent nodes, (2) construct an equivalent query by "merging" these nodes into a single node and (3) remove redundant nodes and edges to construct a minimum equivalent query.

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_r over Q;
2. compute the node equivalence classes EQ based on S_r;
   /* construct an equivalent query */
3. construct V_p^m and E_p^m for Q_m;
4. refine V_p^m and E_p^m;
5. construct an equivalent query Q_m;
   /* construct a minimum equivalent query */
6. remove redundant edges in Q_m;
7. remove isolated nodes in Q_m;
8. **return** Q_m.

Fig. 6 Algorithm minPQs

The algorithm, referred to as minPQs, is outlined in Fig. 6. It has the following three steps. Given a PQ $Q(V_p, E_p)$, (1) minPQs first preprocesses Q by computing the maximum revised graph similarity S_r as well as the node equivalence classes EQ based on S_r; (2) by treating each equivalence class in EQ as a single node, it determines the edges for all these nodes, and constructs an equivalent, yet not necessarily minimum query Q_m for Q; (3) minPQs then identifies and removes redundant edges and nodes from Q_m, and returns it as a minimum equivalent query. We next illustrate each step as follows.

Step 1: Computing S_r *and* EQ (lines 1-2). As a preprocessing step, minPQs first determines whether $u \vdash w$ for all node pairs u, w in Q, and then determines whether $e \models e'$ for all edge pairs e, e' in Q. After that, the algorithm computes the maximum revised similarity relation S_r by employing an algorithm for standard graph simulations, *e.g.,* [26]. It next identifies the nodes that are simulation equivalent, and computes EQ accordingly.

Example 3.3: Consider the PQ Q_1 shown in Fig. 5, where (a) nodes B_1 and B_2 have the same predicate, (b) all those nodes labeled with C (C_i, $i \in [1, 5]$) have the same predicate, and (c) all those nodes with distinct labels (ignoring subscripts) have different predicates. For briefty, we only explicitly annotate the predicates of the nodes labeled with H and J. Given these, algorithm minPQs works as follows.

(1) It first computes the maximum similarity S_r on Q_1, which 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) The set EQ of equivalence classes is derived from the similarity relation S_r. For Q_1, EQ consists of $\mathsf{eq}_0 = \{R\}$,

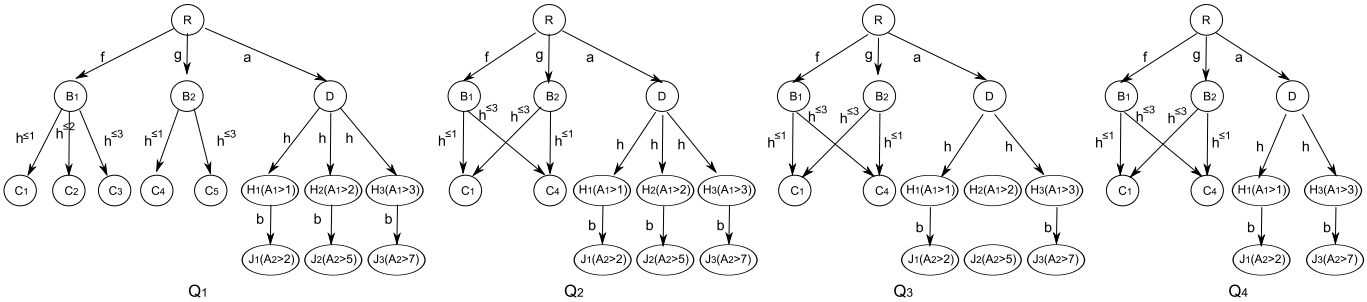


Fig. 5 Example for minimizing graph pattern queries

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

Step 2: Constructing an equivalent query Q_m (lines 3-5). Algorithm minPQs first constructs the nodes and edges of Q_m (line 3). For each equivalence class $Eq \in \mathsf{EQ}$, minPQs initializes a corresponding query node eq and constructs the node set V_p^m for Q_m. It then determines the edge set E_p^m of Q_m as follows. For any two equivalence classes eq_1 and eq_2 in EQ, let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of edges from the nodes in eq_1 to the nodes in eq_2, *i.e.,* $E(\mathsf{eq}_1, \mathsf{eq}_2) = \{e \mid e = (u, w) \in E_p,\ u \in \mathsf{eq}_1,\ w \in \mathsf{eq}_2\}$. If $E(\mathsf{eq}_1, \mathsf{eq}_2)$ is nonempty, minPQs adds an edge $(\mathsf{eq}_1, \mathsf{eq}_2)$ to E_p^m.

Algorithm minPQs then refines the pattern query Q_m by (a) removing redundant edges, and (b) making "copies" of nodes in V_p^m to transform Q_m from multigraph to a simple graph (line 4). More specifically,

- Q_m may contain redundant edges. We say an edge e is *redundant* in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ if (1) there exists another edge e' in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $L(f_e) = L(f_e')$, or (2) there exist two other edges e_1 and e_2 in $E(\mathsf{eq}_1, \mathsf{eq}_2)$ such that $L(f_{e_1}) \subseteq L(f_e) \subseteq L(f_{e_2})$. For each pair eq_1 and eq_2, minPQs removes redundant edges from $E(\mathsf{eq}_1, \mathsf{eq}_2)$, and updates E_p^m accordingly.
- Moreover, Q_m may be a *multigraph* [6], *i.e.,* there may exist multiple edges (with different labels) between two nodes in Q_m. To construct Q_m as a simple graph in which each pair of nodes are connected by at most a single edge, minPQs determines the number of *copies* $N(\mathsf{eq})$ for each nodes, which is defined to be the *maximum* number of non-redundant edges in $E(\mathsf{eq}', \mathsf{eq})$ for all $\mathsf{eq}' \in \mathsf{EQ}$. Here eq' and eq may refer to the same edge. It then extends V_p^m by making $N(\mathsf{eq})$ copies of node eq.

After both V_p^m and E_p^m are refined, algorithm minPQs proceeds to construct an equivalent query $Q_m(V_p^m, E_p^m, f_v^m, f_e^m)$ as follows (line 5).

1. For each eq in EQ, it includes into V_p^m a set $C(\mathsf{eq}) = \{\mathsf{eq}^1, \ldots, \mathsf{eq}^{N(\mathsf{eq})}\}$ of $N(\mathsf{eq})$ nodes. For all nodes u in $C(\mathsf{eq})$ ($\mathsf{eq} \in \mathsf{EQ}$), it sets $f_v^m(u) = f_v(w)$, where $w \in \mathsf{eq}$.
2. Let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of non-redundant edges from eq_1 to eq_2 in EQ. For each eq_1^i ($i \in [1, N(\mathsf{eq}_1)]$) in $C(\mathsf{eq}_1)$, it randomly chooses $|E(\mathsf{eq}_1, \mathsf{eq}_2)|$ nodes from $C(\mathsf{eq}_2)$, and includes in E_p^m a set of $|E(\mathsf{eq}', \mathsf{eq})|$ edges from eq_1^i to those nodes. For each new edge e_{new}, it randomly chooses a distinct edge e in $E(\mathsf{eq}', \mathsf{eq})$, and sets $f_{e_{new}} = f_e$.

Example 3.4: Recall PQ Q_1 in Fig. 5. Consider two equivalence classes $\mathsf{eq}_1 = \{B_1, B_2\}$ and $\mathsf{eq}_2 = \{C_1, C_2, C_3, C_4, C_5\}$ in EQ, and let $E(\mathsf{eq}_1, \mathsf{eq}_2)$ be the set of edges from the nodes in eq_1 to the nodes in eq_2.

(1) There are totally five edges in $E(\mathsf{eq}_1, \mathsf{eq}_2)$, among which edge $e = (B_1, C_2)$ with $f_e = h^{\leq 2}$ is a redundant edge. To see this, observe that there are two edges $e_1 = (B_1, C_1)$ and $e_2 = (B_1, C_3)$, where $f_{e_1} = h^{\leq 1}$, $f_{e_2} = h^{\leq 3}$, and thus, $L(f_{e_1}) \subseteq L(f_e) \subseteq L(f_{e_2})$ (see Section 3.1). Algorithm minPQs thus removes e from Q_m. Similarly, edge (B_1, C_1) and (B_2, C_3) are removed, and eq_1 and eq_2 are connected by two edges (B_1, C_3) and (B_2, C_4).

(2) The number $N(\mathsf{eq}_1)$ of the copies of node eq_1 in EQ is determined by the *maximum* number of non-redundant edges in $E(\mathsf{eq}_1, \mathsf{eq}_2)$, which is 2. Similarly, $N(\mathsf{eq}_2)$ is 2.

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

Step 3: Constructing a minimum Q_m (lines 6-8). Algorithm minPQs further removes redundant nodes and edges for query Q_m in this phase. (a) It first re-computes the maximum revised graph similarity relation S_{rq} on Q_m. (b) It then removes redundant edges. We say that an edge $e = (u, u')$ in Q_m is *redundant* if there exist two edges $e_1 = (u_1, u_1')$ and $e_2 = (u_2, u_2')$ in Q_m such that $(u, u_1) \in S_{rq}$, $(u_2, u) \in S_{rq}$, $(u', u_1') \in S_{rq}$, $(u_2', u') \in S_{rq}$, $e_1 \models e$, and $e \models e_2$. Note that here we use a different notion to identify redundant edges from the one in step 2. All such redundant edges in Q_m are removed at this step (line 6). (b) We say node u in Q_m is *isolated* if there are no edges starting from or ending with the node

u in Q_m. All isolated nodes in Q_m are removed at this step (line 7). Algorithm minPQs then returns Q_m as a minimum equivalent query of Q (line 8).

Example 3.5: Recall that query Q_2 of Fig. 5 is an equivalent query for Q_1. To remove redundant edges from Q_2, algorithm minPQs first computes the maximum revised similarity S'_r on Q_2. It then identifies edge (D, H_2) and (H_2, J_2) as redundant edges. After these edges are removed, Q_2 is updated to be Q_3 as shown in Fig. 5.

Algorithm minPQs then identifies isolated nodes in the updated query Q_3, which are nodes H_2 and J_2. These nodes are then removed from Q_3. After all the isolated nodes are removed, the query Q_3 becomes Q_4 as shown in Fig. 5. The algorithm then returns Q_4 as a minimum equivalent query of the query Q_1. □

To complete the proof of Theorem 3.4, we next show the correctness and complexity of algorithm minPQs.

Correctness. It suffices to show that (I) $Q_m \equiv Q$, and (II) that Q_m is minimum in size, *i.e.,* there is no other equivalent query Q'_m smaller than Q_m.

(I) We first show that $Q_m \equiv Q$, by proving that operations in the algorithm preserve the query equivalence.

(1) $Q_m \equiv Q$ after step 2 of minPQs (line 5). To see this, we only need to show that $Q \trianglelefteq Q_m$ and $Q_m \trianglelefteq Q$.

(a) We construct a relation S'_r from V_p of Q to V_p^m of Q_m as follows. Recall that each node $v_m \in V_p^m$ corresponds to a set $C(\mathsf{eq}) = \{\mathsf{eq}^1, \ldots, \mathsf{eq}^{N(\mathsf{eq})}\}$ of $N(\mathsf{eq})$ copies of an equivalence class eq in EQ. For each node $u \in Q$, $S'_r = \{u, \mathsf{eq}_{ui}\}$ for each $\mathsf{eq}_{ui} \in C(\mathsf{eq})$, where $u \in \mathsf{eq}$.

We show that S'_r is a revised similarity relation from Q to Q_m. Indeed, for any $(u, \mathsf{eq}_{ui}) \in S'_r$, (i) $u \vdash \mathsf{eq}_{ui}$, since eq_{ui} is an equivalence class such that for each $v \in \mathsf{eq}_{ui}$, $u \vdash v$ and $v \vdash u$. (ii) For each edge $e = (u, w) \in E_p$, there is an edge $e' = (\mathsf{eq}_{ui}, \mathsf{eq}_{wi}) \in E_p^m$, where $(w, \mathsf{eq}_{wi}) \in S'_r$ and $e' \models e$. To see (ii), suppose that there exists an edge e for which no other edge e' satisfies the condition given in (ii). If such an edge e' originally exists for e, but is removed from E_p^m as a redundant edge, then there must exist at least e_1 and e_2 such that $L(f_{e_1}) \subseteq L(f'_e) \subseteq L(f_{e_2})$, where e_1 serves as an edge that satisfies the condition of (ii). This indicates that the removal of redundant edges only reduce edge numbers, and preserves the equivalence of the query. Given this, e' does not exist before the removal of redundant edges. Thus, there must exist a child w of u which does not belong to any equivalence class eq_{wi}, the child of all the equivalence classes u belongs to. As a consequence, S_r is not the correct maximum revised similar revision, which contradicts the correctness of the standard graph simulation algorithm [26]. (iii) Along the same lines, one can verify that S'_r guarantees the condition (2) of the revised similarity relation. Thus, S'_r is indeed a revised similarity relation from Q to Q_m, and $Q \lhd Q_m$.

(b) We construct $S'^{-1}_r = \{\mathsf{eq}_{ui}, u\}$ for each $(u, \mathsf{eq}_{ui}) \in S'_r$. As argued above, we can show that $Q_m \trianglelefteq Q$ with S'^{-1}_r as the maximum revised similarity relation.

From (a) and (b) it follows that $Q_m \equiv Q$.

(2) $Q_m \equiv Q$ after step 3 of algorithm minPQs (line 8). Starting from an equivalent query Q_m, minPQs only removes redundant edges and isolated nodes, while preserving query equivalence. To see this, recall S'_r constructed in (1) above. Let $S''_r = S'_r \setminus \{u, \mathsf{eq}_u\}$, where eq_u is a node removed as an isolated node. We show that $Q \trianglelefteq Q_m$ with S''_r as the revised similarity relation. Observe that the removal of redundant edges and isolated nodes still preserves query equivalence. To see this, recall that algorithm minPQs recomputes a revised similarity relation S_{rq} over Q_m. Suppose that a redundant edge $e = (\mathsf{eq}_u, \mathsf{eq}_{u'})$ is removed from Q_m. Then there exist two edges $e_1 = (u_1, u'_1)$ and $e_2 = (u_2, u'_2)$ in Q_m such that $(u, u_1) \in S_{rq}$, $(u_2, u) \in S_{rq}$, $(u', u'_1) \in S_{rq}$, $(u'_2, u') \in S_{rq}$, $e_1 \models e$, and $e \models e_2$. This indicates that for any node $u_q \in Q$, where $(u_q, u) \in S''_r$, there must exist a node u_2 such that $(u_q, u_2) \in S''_r$ if u becomes an isolated node that can no longer match u_q. Moreover, S_{rq} is correctly computed via a standard graph simulation algorithm [26]. Thus, $Q \trianglelefteq Q_m$.

We construct $S''^{-1}_r = \{\mathsf{eq}_{ui}, u\}$ for each $(u, \mathsf{eq}_{ui}) \in S''_r$, which can be shown as the revised similarity relation from Q_m to Q. Thus $Q_m \trianglelefteq Q$. This shows that $Q_m \equiv Q$.

From (1) and (2) it follows that $Q \equiv Q_m$ after minPQs terminates. This completes the proof of (I).

(II) We now show that Q_m is a minimum equivalent query of Q. Consider a PQ $Q = (V_p, E_p)$ and the equivalent query Q_m returned by algorithm minPQs.

Assume that there exists a PQ Q' such that $Q' \equiv Q_m$ and $|Q'| < |Q_m|$. We show that $|Q'| = |Q_m|$, a contradiction. Let EQ_m and EQ' be the equivalence classes for Q_m and Q', computed by algorithm minPQs, respectively. It suffices to show the following, which indicates $|Q_m| = |Q'|$: (1) Q_m and Q' have the same number of nodes, *i.e.,* $|\mathsf{EQ}_m| = |\mathsf{EQ}'|$, and (2) Q_m and Q' have the same number of edges. To prove this, we only need to show that for each pair of equivalence classes eq_1 and eq_2 in EQ_m, $|E_m(\mathsf{eq}_1, \mathsf{eq}_2)| = |E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))|$, where $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$ is the set of edges from the nodes in eq_1 to the nodes in eq_2 in Q_m; similarly for $E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))$.

(1) We first show $|\mathsf{EQ}_m| = |\mathsf{EQ}'|$ by giving a *bijective* mapping f from EQ_m to EQ'. Since $Q_m \equiv Q'$, we have that $Q_m \trianglelefteq Q'$ and $Q' \trianglelefteq Q_m$ by Lemma 3.1. Let $S_r(Q_m, Q')$ and $S_r(Q', Q_m)$ be the maximum revised graph simulation relations for $Q_m \trianglelefteq Q'$ and $Q' \trianglelefteq Q_m$, respectively. We define the mapping $f \subseteq \mathsf{EQ}_m \times \mathsf{EQ}'$ such that $(\mathsf{eq}, \mathsf{eq}') \in f$ if and only if there exist $u \in \mathsf{eq}$ and $u' \in \mathsf{eq}'$ such that $(u, u') \in S_r(Q_m, Q')$ and $(u', u) \in S_r(Q', Q_m)$. We show that f is a bijection as follows.

(a) We first show that f is a function from EQ_m to EQ'. Assume by contradiction that there is an equivalence

class eq in EQ_m and two equivalence classes eq'_1 and eq'_2 in EQ' such that $(\mathsf{eq}, \mathsf{eq}'_1) \in f$ and $(\mathsf{eq}, \mathsf{eq}'_2) \in f$. One can see that $\mathsf{eq}'_1 = \mathsf{eq}'_2$ as follows.

- Since $(\mathsf{eq}, \mathsf{eq}'_1) \in f$, there exist $u_1 \in \mathsf{eq}$ and $w_1 \in \mathsf{eq}'_1$ such that $(u_1, w_1) \in S_r(Q_m, Q')$ and $(w_1, u_1) \in S_r(Q', Q_m)$.
- From $(\mathsf{eq}, \mathsf{eq}'_2) \in f$ it follows that there exist $u_2 \in \mathsf{eq}$ and $w_2 \in \mathsf{eq}'_2$ such that $(u_2, w_2) \in S_r(Q_m, Q')$ and $(w_2, u_2) \in S_r(Q', Q_m)$.
- By $u_1, u_2 \in \mathsf{eq}$, we have that $(u_2, w_1) \in S_r(Q_m, Q')$ and $(u_1, w_2) \in S_r(Q_m, Q')$.
- In light of $(w_1, u_1) \in S_r(Q', Q_m)$ and $(u_1, w_2) \in S_r(Q_m, Q')$, we have that $w_1 \vdash w_2$.
- From $(w_2, u_2) \in S_r(Q', Q_m)$ and $(u_2, w_2) \in S_r(Q_m, Q')$ it follows that $w_2 \vdash w_1$.

From these we can derive that $\mathsf{eq}'_1 = \mathsf{eq}'_2$ since w_1 and w_2 are simulation equivalent. Hence f is a function.

(b) The function f is a bijection. Indeed, f is total since it is induced by the revised similarity relation, which is total. We next show that f is injective, *i.e.,* for any two different nodes eq_1 and $\mathsf{eq}_2 \in \mathsf{EQ}_m$, $f(\mathsf{eq}_1) \neq f(\mathsf{eq}_2)$. Suppose that there are two nodes eq_1 and eq_2 such that $f(\mathsf{eq}_1) = f(\mathsf{eq}_2) = \mathsf{eq}'_u$ in Q'. (i) For every child eq'_1 of eq in Q_m, there exist three edges $e_1 = (\mathsf{eq}_1, \mathsf{eq}'_1)$, $e_2 = (\mathsf{eq}_2, \mathsf{eq}'_2)$ in Q_m and $e = (\mathsf{eq}_u, \mathsf{eq}_v)$ in Q', such that $(\mathsf{eq}_1, \mathsf{eq}_u), (\mathsf{eq}'_1, \mathsf{eq}_v) \in S_r(Q_m, Q')$, $e_1 \models e$, and $(\mathsf{eq}_2, \mathsf{eq}_u), (\mathsf{eq}'_2, \mathsf{eq}_v) \in S_r(Q', Q_m)$, $e \models e_2$, by $Q_m \equiv Q'$. Thus, $(\mathsf{eq}_1, \mathsf{eq}_2) \in S_r(Q_m, Q')$. (ii) Similarly, we can show that $(\mathsf{eq}_2, \mathsf{eq}_1) \in S_r(Q_m, Q')$. This tells us that eq_1 and eq_2 are simulation equivalent. Since algorithm minPQs computes the maximum revised similar relation over Q, eq_1 and eq_2 should be in the same equivalence class. This contradicts the assumption that $\mathsf{eq}_1 \neq \mathsf{eq}_2$. Thus, the function f is an injective function.

(c) We finally show that the mapping f is surjective. This can be verified by proving that f^- is a total and injective function, via a similar argument as (b).

Putting (a), (b) and (c) together, we have that f is a total, surjective and injective function. That is, f is a bijection from the nodes of Q_m to the nodes of Q'. Therefore, Q' and Q_m have the same number of nodes.

(2) We show that for each pair of equivalence classes eq_1 and eq_2 in EQ_m, $|E_m(\mathsf{eq}_1, \mathsf{eq}_2)| = |E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))|$. Consider a pair $(\mathsf{eq}_1, \mathsf{eq}_2)$ in EQ_m. Along the same lines as above, we can construct a bijective mapping g from the edges in $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$ to the edges in $E'(f(\mathsf{eq}_1), f(\mathsf{eq}_2))$ such that $L(f_e) = L(f_{g(e)})$ for each edge in $E_m(\mathsf{eq}_1, \mathsf{eq}_2)$.

From (1) and (2) above it follows that $|Q'| = |Q_m|$. This completes the prove of (II). The correctness of algorithm minPQs follows from (I) and (II). □

Remark. The proof is inspired by the proof for minimizing Kripke structures based on graph simulations [11]. It is shown there that all minimum Kripke structures are isomorphic. For graph pattern queries, however, two minimum queries may not be isomorphic, as remarked earlier in Figure 4. This makes the techniques used in this proof different from those used in the proof of [11].

Complexity. We next show that algorithm minPQs indeed runs in cubic time, by showing that each of its three steps, *i.e.,* preprocessing (lines 1-2), equivalent query construction (lines 3-5), and minimum equivalent query construction (lines 6-8), can be done in cubic time.

Preprocessing (lines 1-2). The computation of the maximum revised similarity S_r is in cubic time, *i.e.,* $O(|V_p||E_p||L|)$ time (line 1), via the graph simulation algorithm [26]. Here L is the maximum length of the regular expression over query edges. The node equivalence classes EQ can be computed in $O(|V_p|^2)$ time [11]. More specifically, the computation of S_r and EQ requires checking (1) whether $u \vdash v$ for two query nodes u and v, which is in quadratic time; and whether $L(f_{e_1}) \subseteq L(f_{e_2})$ for two query edges e_1 and e_2, which is in $O(L)$ time. The total time of preprocessing phase is thus in $O(|Q|^3)$.

Equivalent query construction (lines 3-5). The construction of V_p^m and E_p^m is in $O(|Q|)$ time (line 3). The refinement of V_p^m and E_p^m is in cubic time (line 4). The construction of Q_m is in $O(|Q|)$ time, as $|Q_m| \leq |Q|$ (line 5). More specifically, checking redundant edges is in $O(|E_p^m|^2|L|)$ time, and determining the copy number of nodes in Q_m is in $O(|E_p^m|)$ time. Thus, the total time for constructing Q_m is in $O(|Q|^3)$ time.

Minimum equivalent query construction (lines 6-8). It takes $O(|V_p||E_p||L|)$ time to re-compute the revised similarity relation S_{rq}. Removing redundant edges in Q_m take $O(|E_p^m|^2|L|)$ time. It takes $O(|V_p|)$ time to remove isolated nodes. Thus, this phase is in $O(|Q|^3)$ time.

Putting these together, minPQs is in $O(|Q|^3)$ time. □

From the correctness and complexity analyses of algorithm minPQs, Theorem 3.4 immediately follows.

Observe that the complexity bounds of minimization, containment and equivalence are all in the sizes of queries, which are typically *much smaller than* the sizes of data graphs in practice.

4 Evaluating Reachability Queries

In this section, we present two 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 [44]. 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 $G = (V, E, f_A, f_C)$. For two nodes v_1, v_2 in V, 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 2).

RQ with a single edge color. Below we start with a special case when f_e carries a single edge color, and then consider the general case.

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 the constraint 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 [6]. Note that m is typically much smaller than $|V|$. The 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 in a query, 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 $O(|V| + |E|)$ time. A similar technique is used in [15], 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 the data graph G of 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 labeled with color fa.

RQ with multiple colors. We next extend the two methods to evaluate a general RQ Q_r. Assume 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 can be omitted.

Example 4.1: 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 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 6, this method is able to process queries on large data graphs, when maintaining a distance matrix for those graphs is beyond reach in practice.

It should be remarked that although existing (index-based) solutions for traditional reachability queries cannot answer RQs studied in this paper, they can be leveraged as filters, *i.e.,* we invoke our methods only after those techniques decide that two nodes are connected (possibly constrained by a set of labels).

5 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 on G, in cubic time in the size of G. The first algorithm is based on join operations. The other is based on split, an operation commonly used in verifications of labeled transition systems (LTS, see, *e.g.,* [37]).

5.1 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.

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. Depending on flag, the algorithm decides to use which method given in Section 4 to evaluate the 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 2). (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.* edge 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 [41] (line 5).

In a reversed 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 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 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

---

Input: query $Q_p = (V_p, E_p)$, 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. $Q_s :=$ Sccgraph(Q'_p);
6. **for each** C_s of Q_s in a reversed 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 is $e = (u', u) \in E'_p$ *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. $S_e := \{(x', x) \mid x' \in \mathsf{mat}(u'), x \in \mathsf{mat}(u), (x', x) \approx f_e(e)\}$;
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

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 4).

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 make use of matrix M. (2) Otherwise, the regular expressions are evaluated straightforwardly using *bi-directional search* (see Section 4).

Example 5.1: Recall the pattern query 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 mat() those nodes that do not make a match, by using premv() from procedure Join.

There are two SCC's: 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 same result $Q_2(G)$ is found as illustrated in Example 2.3. □

We show the correctness and complexity analysis for the algorithm JoinMatch as follows.

Correctness. We show that the algorithm JoinMatch correctly returns $Q_p(G)$. (1) It always terminates. Indeed, for each node u' in Q_p, the set $\mathsf{mat}(u')$ decreases monotonically. (2) We show that after the **for** loop (lines 6-14), each node recorded in $\mathsf{mat}(u')$ is a match of node u'. Denote the set of matches of u as $\mathsf{mat}_t(u')$. We only need to show that for each node u', $\mathsf{mat}(u') = \mathsf{mat}_t(u')$ after the **for** loop.

We first show that JoinMatch preserves the invariant that at any iteration of the **for** loop, for any node u', $\mathsf{mat}_t(u') \subseteq \mathsf{mat}(u')$. We show this by induction on the iteration of the loop. (a) $\mathsf{mat}_t(u') \subseteq \mathsf{mat}(u')$ at the beginning of the loop. (b) Assume that $\mathsf{mat}_t(u') \subseteq \mathsf{mat}(u')_i$ at iteration i of the loop. At iteration $i+1$, the set $\mathsf{rmv}(e)$ is computed (line 9), where for each node $v' \in \mathsf{rmv}(e)$, there is no path satisfying the constraints of e, *i.e.,* it is not a match of u'. The match set $\mathsf{mat}(u')_i$ is refined to $\mathsf{mat}(u')_{i+1}$ by removing all these nodes that cannot match u'. Thus, $\mathsf{mat}_t(u') \subseteq \mathsf{mat}(u')_{i+1}$.

The argument above shows that JoinMatch only removes nodes that cannot match u' from $\mathsf{mat}(u')$. We next show that after the loop, $\mathsf{mat}(u') = \mathsf{mat}_t(u')$. Suppose that there exists a node $v' \in \mathsf{mat}(u')$ that cannot match u' after the loop. That is, there is an edge $e = (u', u)$ such that v' cannot satisfy the constraints of e. Observe that $\mathsf{rmv}(e)$ contains at least one such node v' after procedure Join (line 9 and line 13). This violates the termination condition of the loop (line 14), and v is to be removed from $\mathsf{mat}(u')$ at some iteration (line 10). Thus, $\mathsf{mat}(u') = \mathsf{mat}_t(u')$ after the loop, for each node u' in Q_p. Putting these together, one can verify that JoinMatch correctly computes the matches $Q_p(G)$.

Complexity. We analyze the complexity based on the case that the distance matrix is used. The algorithm consists of two phases: pre-processing (lines 1-5) and match computation (lines 6-17).

Preprocessing (lines 1-5). This step takes $O((m+1)|V|^2 + |V|(|V| + |E|))$ time to normalize Q_p and compute the distance matrix, each for a single color, where m is the number of distinct edge colors, typically a small number in real-life applications (lines 1-2). The initialization of $\mathsf{mat}(u)$ and $\mathsf{rmv}(e)$ for each node u and edge e in Q'_p takes in total $O(|V||V'_p| + |E'_p|)$ time (lines 3-4). It takes linear time $O(|V'_p| + |E'_p|)$ to compute Q_s (line 5) [41]. Thus, the preprocessing phase takes $O((m+1)|V|^2 + |V|(|V| + |E|) + |V||V'_p| + (|V'_p| + |E'_p|))$ time in total.

Match computation (lines 6-17). The **for** loop (lines 6-16) is repeated $O(|E'_p|)$ times. For each edge e in E'_p, procedure Join takes $O(|V|^2)$ time (line 9). It takes $O(|V|)$ time to update $\mathsf{mat}(u)$ (line 10), $\mathsf{rmv}(e)$ (line 9) and $\mathsf{rmv}(e')$ (line 13), respectively. Putting these together, the **for** loop is in $O(|E'_p||V|^2)$ time. It takes $O(|E'_p||V|)$ time to collect the result $Q_p(G)$. Thus, the match computation is in total $O(|E'_p||V|^2)$ time.

Putting these together, the algorithm is in $O(|V||E| + |E'_p||V|^2)$ time. Notably, $|E'_p|$ and $|V'_p|$ are bounded by $O(m|E_p|)$ and $O(V_p + (m-1)|E_p|)$, respectively.

Remark. Observe the following. (1) The distance matrix can be computed in $O((m+1)|V|^2 + |V|(|V| + |E|))$ time (line 2). The initialization of $\mathsf{mat}(u)$ is in $O(|V||V'_p|)$ time. The normalization and SCC graph are both bounded by $O(|V'_p| + |E'_p|)$. (2) Clearly, if Q_p is a DAG, the loop takes a single bottom-up sweep for each node in Q_p, which naturally takes $O(|E'_p||V|^2)$ time. Otherwise, an auxiliary structure is maintained for each node, recording its descendants removed from possible matches, to avoid redundant check in the iterations of the loop (lines 7-14). In this way, the loop is bounded by $O(|E'_p||V|^2)$ for PQs that are general graphs.

5.2 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 is a partial order over par. The pair $\langle \mathsf{par}, \mathsf{rel} \rangle$ is recursively refined by *splitting* the blocks in par and rel based on the constraints imposed by query edges. The process proceeds until a fixpoint is reached, and then the result of Q_p is collected from the corresponding blocks of query nodes in V_p, and the partial order over the blocks in rel.

The idea of split was first explored in LTS verification [37], which deals with a single graph. Our algorithm extends the idea to handle two graphs.

Algorithm. The algorithm, referred to as SplitMatch, is shown in Fig. 8. It also needs the procedures mat() and 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.

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 := ∅; rel := ∅;
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) := ∅;
10. Split(e, ⟨par, rel⟩, rmv);
11. **for each** B $\subseteq$ rmv **do** rel(B(u')) = rel(B(u')) \ 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** ∅;
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 \ SpltN;
3. par := par $\cup \{$B$_1\} \cup \{$B$_2\}$; par := par \ {B};
4. rel(B$_1$) := rel(B$_2$) := {B$_1$, B$_2$};
5. **return** ⟨par, rel⟩;

Fig. 8 Algorithm SplitMatch

It then initializes the match set and block set of each query node (line 5). In addition, it constructs the partition-relation pair ⟨par, rel⟩ (line 6); it also initializes 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 rmv() set, *i.e.,* edges for which the match set can be refined (lines 8-14). The set of blocks par is split based on rmv(e) in procedure Split, and rel is updated accordingly (line 10). SplitMatch further extends the rmv() 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). The extended rmv(e') is used to further 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 the 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 correspondingly (lines 3-4), and the refined pair ⟨par, rel⟩ is returned (line 5).

Example 5.2: 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 ∅. One can verify that during the **while** loop (lines 8-14), the block set of node C is refined by making use of rmv(C, B), resulting in a new block set from which nodes C_1 and C_2 are absent; similarly for the other blocks.

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$_3$}

Algorithm SplitMatch identifies the same result as reported in Example 2.3. □

We next give the correctness and complexity analyses for algorithm SplitMatch as follows.

Correctness. 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, *i.e.,* satisfying all the edge constraints; and (4) each block decreases monotonically. We provide details below.

We first introduce notations we shall use in the analysis. (a) Given a set of blocks Bs and a query edge e with $f_e(e) = c^k$, we define prev(Bs) as the set of blocks in G, such that for each block $B \in$ prev(Bs) and each node $u \in B$, there exists a node v in a block of Bs, where (i) there is a shortest path from u to v with all edges e' in the path satisfying $f_C(e') = c$, and (ii) the path has length bounded by k. (b) We say that partition-relation pair ⟨par, rel⟩ *over-approximates* S if for any edge $e = (u', u) \in E_q$ with $f_e(e) = c^k$, rel(B(u')) $\subseteq \cup$prev(e, rel(B(u))).

Using these notations, we next show that SplitMatch maintains an invariant, namely, at any time, ⟨par, rel⟩ *over-approximates* S. We verify this by induction on the iteration of the **while** loop (lines 8-14) as follows. (1) The invariant is preserved when ⟨par, rel⟩ is initialized

(line 6). (2) Suppose that at iteration i the invariant is maintained by $\langle \mathsf{par}_i, \mathsf{rel}_i \rangle$. At iteration $i+1$, (a) par_i is *split* based on a non-empty set $\mathsf{rmv}(e)$ for edge $e \in E_p$, and rel_i is updated according to newly generated blocks from par_i (line 11). Recall that for an edge $e = (u, v)$, where $f_e(e) = c^k$, $\mathsf{rmv}(e)$ represents the set of nodes which fail to satisfy the constraints of e. SplitMatch *only* removes such nodes as a block from an existing block in par_i, which preserves the invariant. Thus, SplitMatch maintains the invariant.

We finally show that the induced relation S (lines 15-16) is the query result when the **while** loop terminates (lines 8-14). Observe that when SplitMatch terminates, for *every* edge $e = (u, v)$, the set $\mathsf{rel}(\mathsf{B}(u))$ contains the desirable blocks, each of them (a) contains a set of nodes satisfying the constraint of edge e, guaranteed by the invariant, and (b) can no longer be further partitioned by $\mathsf{rmv}(e')$ of any other edge e', *i.e.,* it contains no node that is not a match, since all $\mathsf{rmv}(e')$ is empty for any edge e (line 8). The union of these blocks is thus exactly the match set of u. From these it follows that SplitMatch correctly computes the query result.

Complexity. The complexity analysis below is based on the assumption that SplitMatch uses the distance matrix as index. The algorithm consists of three phases: pre-processing (lines 1-7), match computation (lines 8-14), and result collection (lines 15-18). We give their complexity bounds as follows.

Pre-processing (lines 1-7). The pre-processing phase is in $O((m+1)|V|^2 + |V|(|V| + |E|) + |V'_p||V| + |E'_p||V|^2)$ time, similarly to its counterpart in JoinMatch, where m is the number of distinct edge colors.

Match computation (lines 8-14). We denote the initial par at line 6 as par_{in}, and the final refined par as par_{out}. For match computation process (lines 8-14), observe that (1) at each iteration i, each par_i is a refinement of par_{i-1} at iteration $i-1$, (2) $\mathsf{rmv}(e)_i$ and $\mathsf{rmv}(e)_{i-1}$ are disjoint, and (3) the total number of newly generated blocks at line 10 is $2(|\mathsf{par}_{out}| - |\mathsf{par}_{in}|)$. As a result, the overall time complexity of the code at line 10 is $O(|E_p||\mathsf{par}_{out}|)$. The time complexity for the inner **for** loop at line 11 is $O(|\mathsf{par}_{out}||V|^2)$, with the maintenance of a 2-D matrix along the same line in algorithm JoinMatch for each edge $e(u', u) \in E_p$ and $\mathsf{mat}(u')$. The Split procedure is in $O(|V|)$ time, thus the total time at line 8 is $O(|\mathsf{par}_{out}||V|)$. Putting these together, the total time in the second phase (lines 8-14) is in $O(|\mathsf{par}_{out}||V|^2)$.

Result collection (lines 15-18). There are totally $|E_p|$ edges, and for each edge $e = (u, v)$, there are at most $|V|$ matches for u and v, respectively. Thus, the result collection is in $O(|E_p||V|)$ time.

Remark. The set par_{out} represents the finally refined par, which is bounded by $O(|V||V'_p|)$. A closer observation of the complexity of SplitMatch tells us that $|\mathsf{par}_{out}|$ is between $|V'_p|$ and $|V'_p||V|$, *i.e.,* the algorithm is in $O(|V'_p||V|^3)$ time. However, suppose that a block $\mathsf{B}(u)$ is split (line 8) into B_1 (contains u) and B_2 (without u). It is unnecessary to find matches for B_2. Thus, one can verify that SplitMatch has a comparable worst case complexity to $|E'_p||V^2|$, measured with input size. Moreover, the same auxiliary structure used in algorithm JoinMatch is adopted here, to ensure that the loop (lines 6-14) runs in $O(|\mathsf{par}_{out}||V|^2)$ time for a cyclic query.

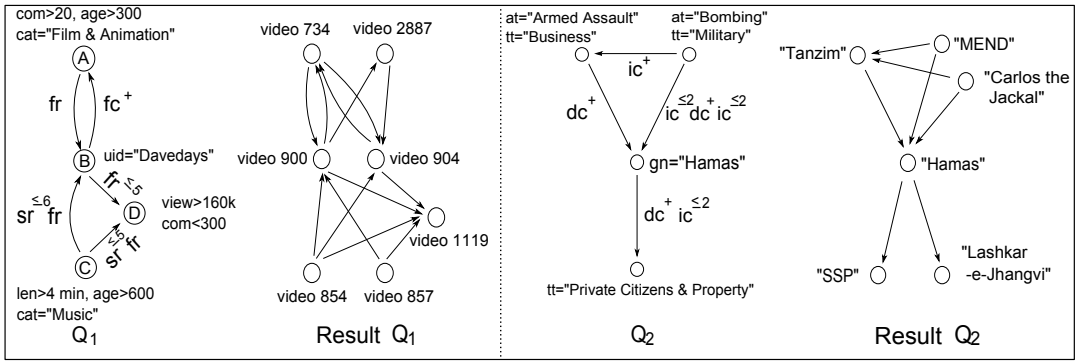
6 Experimental Evaluation

In this section we present an experimental study using both real-life data and synthetic data. Four sets of experiments were conducted, to evaluate: (1) the effectiveness of PQs, compared with a subgraph isomorphism algorithm SubIso [43] and a simulation based pattern matching algorithm Match [20]; (2) the effectiveness of minimization as an optimization strategy; (3) the efficiency of RQ evaluation; and (4) the efficiency and scalability of algorithms JoinMatch and SplitMatch, employing distance matrix and distance cache as indices.

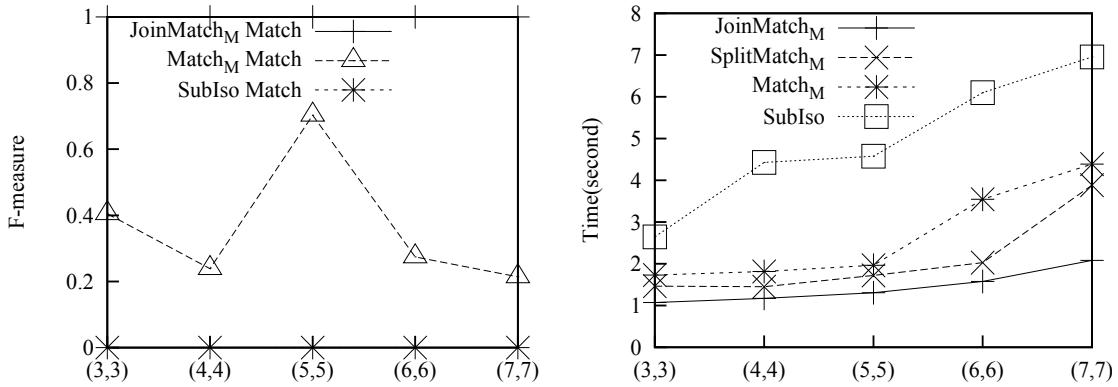
Experimental setting. We used real-life data to evaluate the performance of our methods in the real world applications, and synthetic data to vary graph characteristics, for an in-depth analysis.

(1) Real-life data. We used two sets of real-life data. (a) We used *YouTube* dataset, in which each node denotes a video with attributes such as uploader id (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. The dataset has 8350 nodes and 30391 edges. (b) We generated 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.

(2) Query generator. We designed a query generator to produce meaningful PQs. The generator has five parameters: $|V_p|$ denotes the number of pattern nodes, $|E_p|$ is the number of pattern edges, $|\mathsf{pred}|$ denotes the number of predicates each pattern node carries, and bounds b and c are used 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 with two nodes and one edge.



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



(b) Effectiveness comparison (c) Efficiency comparison

Fig. 9 Exp-1: Effectiveness of PQs

(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 of a node, and a set of edge colors that an edge may carry.

(4) Implementation. We have implemented the following, all in Java: (a) the bi-directional search based method (biBFS) 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 [43]; and (e) Match, a simulation based pattern matching algorithm developed in [20].

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

Experimental results. We next present our findings.

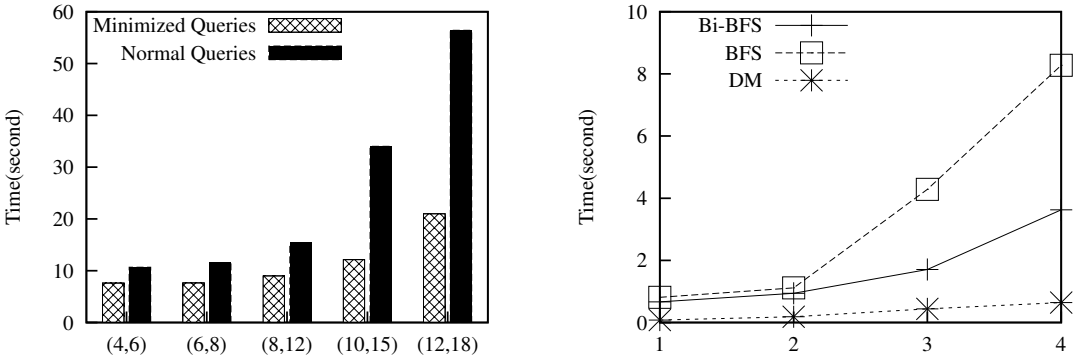
Exp-1: Effectiveness of PQs. In the first set of experiments, we evaluated the effectiveness of PQs. In contrast to SubIso and Match, we show that PQs can identify meaningful matches in real-life data. For quantitative comparison, the F-Measure [46] is adopted, which is defined as follows:

$$\mathsf{F\text{-}Measure} = 2 \cdot (\mathsf{recall} \cdot \mathsf{precision}) \,/\, (\mathsf{recall} + \mathsf{precision})$$
$$\mathsf{recall} = \#\mathsf{true\_matches\_found} \,/\, \#\mathsf{true\_matches}$$
$$\mathsf{precision} = \#\mathsf{true\_matches\_found} \,/\, \#\mathsf{matches}$$

Here #matches is defined as the number of distinct node pairs (u, v), where u is a query node and v is a graph node that matches u. The #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 is to find the videos A in the category "Film & Animation", which have more than 20 comments and were uploaded at least 300 days ago. 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 and have 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



(a) Query minimization

(b) RQs over *Youtube*

Fig. 10 Exp-2: PQs minimization and Exp-3: efficiency of RQ

middle east. Existing approaches *e.g.,* SubIso and Match, are not expressive enough to specify 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 the following, (1) PQs consistently 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, since 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.

The results above tell us that PQs are not only more effective, but are also more efficient than its conventional counterparts, *i.e.,* SubIso and Match.

Exp-2: The effectiveness of PQ minimization. We evaluated the effectiveness of the minimization algorithm minPQs (Section 3), using *YouTube* data. The queries were generated by varying $|V_p|$ and $|E_p|$. The average number of predicates $|\mathsf{pred}|$ is 3. The bound c is between 2 and 4, and b is 5, *i.e.,* each edge is constrained by a regular expression $c_1^{\leq 5} \ldots c_k^{\leq 5}$, where $2 \leq k \leq 4$. The results are reported in Fig. 10(a).

In Fig. 10(a), the x-axis is the same as its counterparts in Fig. 9(b), and the y-axis represents the elapsed time for query evaluation. We only show the results of using algorithm $\mathsf{JoinMatch_M}$, since 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 by removing redundant nodes and edges from a query, and thus speed up the query evaluation; and (2) in general, 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 after minimization, the running time was cut by over a half since the minimized queries have 7 nodes and 9 edges in average.

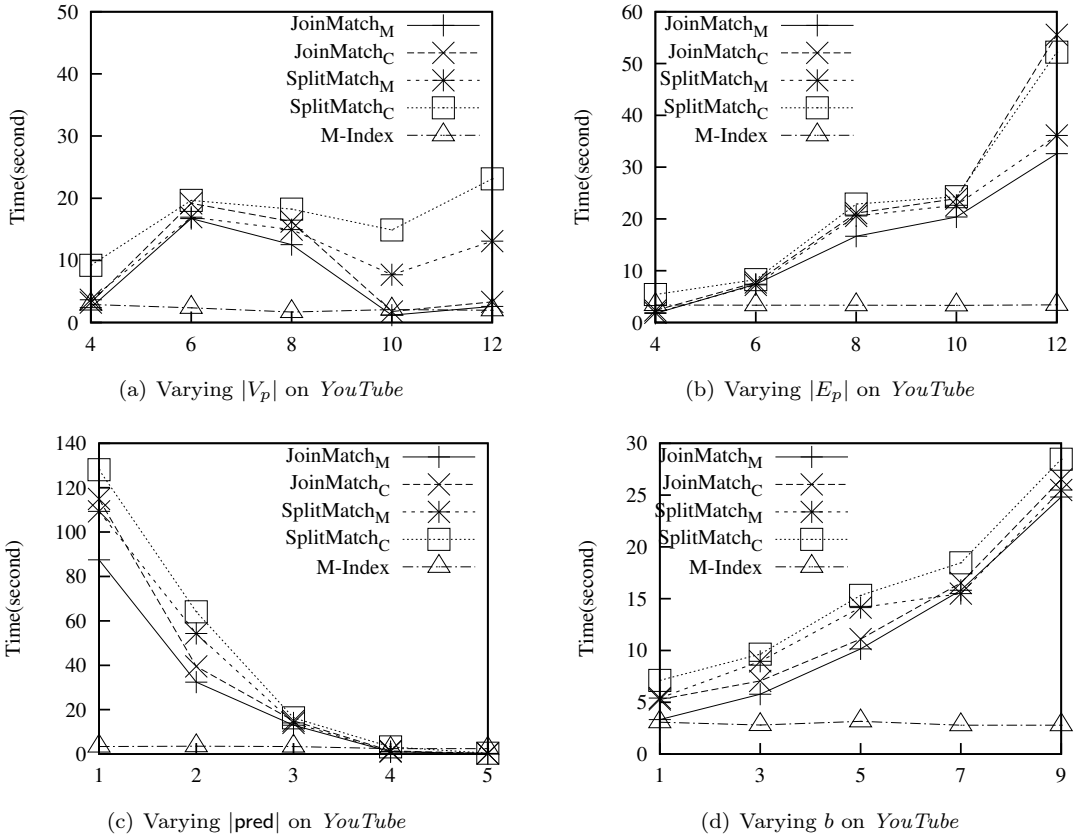
This set of experiments verified that the minimization algorithm can effectively optimize PQs. In the rest of experiments, all tested queries were minimized.

Exp-3: Efficiency of RQs. In this set of experiment, we tested the efficiency of the two algorithms presented in Section 4 for evaluating reachability queries 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 $c_1^{\leq b} \ldots c_i^{\leq b}$ for $i \in [1, 4]$.

Figure 10(b) shows the average elapsed time of evaluating RQs on *YouTube* data. The x-axis represents the number of distinct colors and y-axis indicates 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 biBFS is more efficient than BFS, as expected.

(2) Algorithm biBFS scales better than BFS with the number of colors, since by searching from two directions, biBFS produces less intermediate nodes than BFS. The trend of the curves of biBFS and BFS indicates that biBFS works better for more complex regular expressions.



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

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

(c) Varying |pred| on *YouTube*

(d) Varying b on *YouTube*

Fig. 11 Exp-4: Efficiency of PQs (Youtube)

(3) As will be seen shortly, maintaining distance matrix is expensive for large graphs. Hence biBFS makes a rational solution on large graphs, by striking the balance between time and space.

Exp-4: Efficiency of PQs on *YouTube*. In this set of experiments we evaluated the performance of JoinMatch and SplitMatch over synthetic and real life graphs.

Figures 11(a), 11(b), 11(c) and 11(d) depict the elapsed time when varying one of the parameters: $|V_p|$, $|E_p|$, |pred| and b, respectively. See Fig. 10(b) 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 JoinMatch$_M$ and SplitMatch$_M$. The result tells us the following.

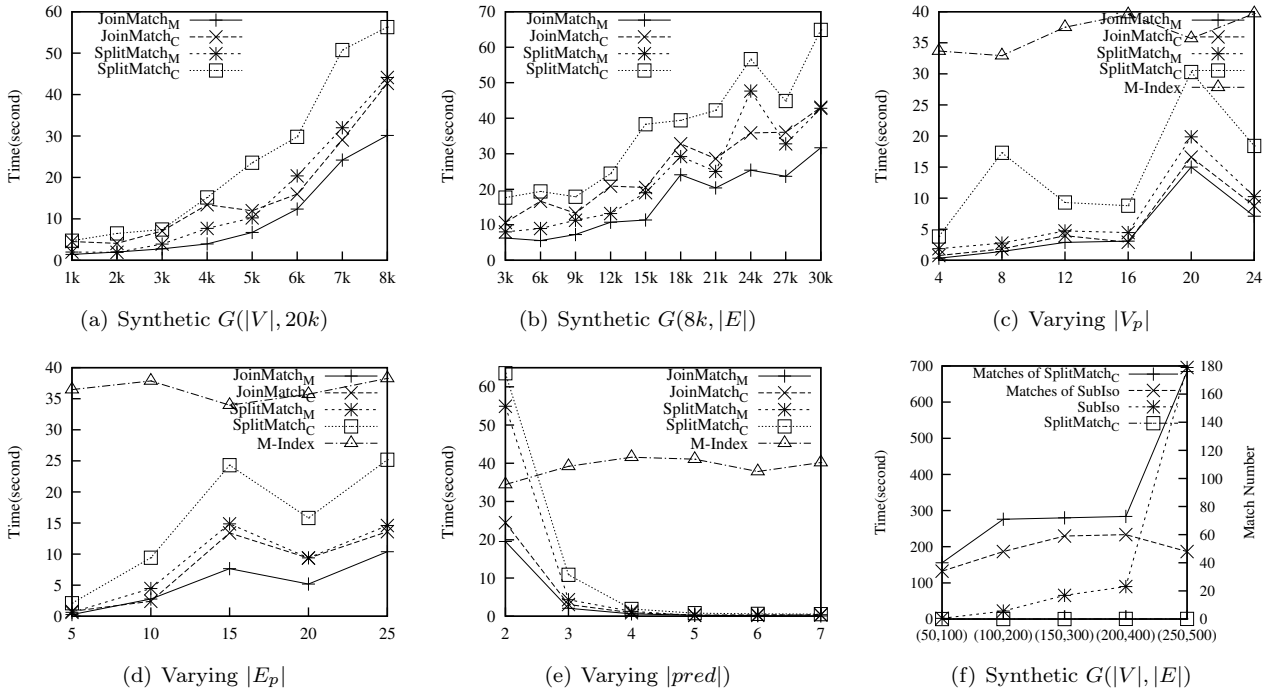
(1) Figure 11(a) shows that the matrix-based algorithm JoinMatch$_M$ (resp. SplitMatch$_M$) outperforms the distance-cache based JoinMatch$_C$ (resp. SplitMatch$_C$). This is because JoinMatch$_M$ and SplitMatch$_M$ use distance matrix as index, which answers node distance in constant time, while JoinMatch$_C$ and SplitMatch$_C$ are based on distance cache: if the distance of two nodes is not cached, it needs to be recomputed from scratch.

(2) The join-based methods outperform the split-based methods. As shown in the figures with various parameters, in most cases JoinMatch$_M$ is the fastest, followed by SplitMatch$_M$; and JoinMatch$_C$ outperforms SplitMatch$_C$. This indicates that the computational cost of the join-based method is reduced by adopting the reversed topological order (see Section 5).

(3) The elapsed time is more sensitive to the number of pattern edges (see Fig. 11(b)) than pattern nodes (see Fig. 11(a)), since the number of pattern edges dominates the number of joins or splits to be conducted. Moreover, the elapsed time is sensitive to the number of predicates (see Fig. 11(c)) since predicates impose a strong constraint on initializing the match set. The more the predicates, the less graph nodes satisfy them, resulting in smaller candidate matches and faster evaluation. The time is sensitive to the bound (see Fig. 11(d)) since the number of matches gets larger when b is increased.

(4) These results demonstrate that all algorithms have good scalability and they work well when the numbers of $|V_p|, |E_p|$ |pred| and b become much larger.

(5) We can see that M-index can be computed efficiently, and it may significantly improve the performance, when the dataset is relatively small.



(a) Synthetic $G(|V|, 20k)$ (b) Synthetic $G(8k, |E|)$ (c) Varying $|V_p|$

(d) Varying $|E_p|$ (e) Varying $|pred|$ (f) Synthetic $G(|V|, |E|)$

Fig. 12 Exp-4: Efficiency of PQs (synthetic graphs)

As a supplement, we verified the proposed algorithms using synthetic data. We first varied both the number of graph nodes and edges using synthetic data, in order to test the scalability. The results are shown in Figures 12(a) and 12(b), respectively. The five parameters: $|V_p|$, $|E_p|$, c, $|\mathsf{pred}|$ and b are 6, 8, 4, 3 and 5, respectively. We find that all algorithms scale well with the increasing number of graph nodes (Figure 12(a)), and the number of graph edges (Figure 12(b)). Furthermore, for synthetic data graphs with $8K$ nodes and 4 distinct colors, the distance matrix consumes $512MB$ memory, when using an *unsigned short integer* to store a matrix cell; for a data graph with $16K$ nodes and 4 distinct colors, it takes $2GB$ memory. This shows that a matrix is too large to be applicable to large graphs, and runtime techniques should be employed in such cases.

Figures 12(c), 12(d) and 12(e) confirm the previous observations in their real life counterparts (Figures 11(a), 11(b), 11(c), respectively). The results tell us the following. (1) All algorithms are not very sensitive to the number of query nodes (Figure 12(c)). (2) All algorithms scale well with the increasing number of query edges (see Figure 12(d)). (3) All algorithms are sensitive to the increasing number of predicates over queries (Figure 12(e)). Note that the results show that it takes longer time to compute the distance matrix, which hinders its applicability to larger data graphs.

In addition, we compared the efficiency and scalability of SubIso and $\mathsf{SplitMatch}_C$ by using a set of small data graphs. We generated queries with 8 nodes and 15 edges, where each node has 3 predicates, and each edge is associated with a regular expression in the form of $c_1^5 c_2^5 c_3^5 c_4^5$. We then tested SubIso and $\mathsf{SplitMatch}_C$ over the patterns by varying the number of data graph nodes and edges. To favor SubIso, we counted the number of matches as the number of distinct node pairs (u, x), where u is a query node, and x is a match of u in the data graph. The result tells us that while the matches found by SubIso is far less than those found by $\mathsf{SplitMatch}_C$, SubIso spent around 700 seconds, even for data graph of 200 nodes and 250 edges. In contrast, it took $\mathsf{SplitMatch}_C$ less than 1 second to identify all the meaningful matches. Moreover, SubIso is more sensitive to the change of the size of data graph than $\mathsf{SplitMatch}_C$.

Summary. From the experimental results we find the following. (1) Graph pattern queries (PQs) are able to identify far more sensible matches in emerging application than those found by the conventional approaches. (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, algorithm biBFS works reasonably well when working on large graphs. (4) PQs can be efficiently evaluated, and their evaluation algorithms scale well with large graphs and complex patterns.

7 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 also revised graph pattern matching by introducing an extension of the classical notion of graph simulation. Moreover, we have 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 [28]. 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, using real-life data and synthetic data.

Several extensions are targeted for future work. One topic 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. In practice data graphs are frequently modified, and it is too costly to re-evaluate PQs in cubic-time (or RQs in quadratic-time) on large data graphs every time the graphs are updated. This suggests that we evaluate the queries once, and incrementally compute query answers in response to changes to the graphs. It is, however, nontrivial to find incremental algorithms that guarantee to minimize unnecessary recomputation. While incremental graph pattern matching has recently been investigated [20, 22], it poses new challenges when graph patterns are defined in terms of regular expressions, hence, deserves a full treatment.

Acknowledgments. Fan is supported in part by the RSE-NSFC Joint Project Scheme and an IBM scalable data analytics for a smarter planet innovation award. Fan and Li are also supported in part by the National Basic Research Program of China (973 Program) 2012CB316200 and NSFC 61133002. Ma is supported in part by NGFR 973 grant 2011CB302602 and NSFC grants 90818028 and 60903149, and the Young Faculty Program of MSRA.

References

1. National Consortium for the Study of Terrorism and Responses to Terrorism (START). *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. R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, 1989.
5. S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.
6. J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.
7. P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, 2010.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.
9. M. J. Brzozowski, T. Hogg, and G. Szabó. Friends and foes: ideological social networking. In *CHI*, 2008.
10. 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.
11. D. Bustan and O. Grumberg. Simulation-based minimization. *TOCL*, 4(2):181–206, 2003.
12. E. P. Chan and H. Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343 – 369, 2007.
13. D. Chen and C. Y. Chan. Minimization of tree pattern queries with constraints. In *SIGMOD*, 2008.
14. L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, 2005.
15. Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, 2009.
16. J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.
17. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5):1338–1355, 2003.
18. W. Fan and P. Bohannon. Information preserving XML schema embedding. *TODS*, 33(1), 2008.
19. W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.
20. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. In *PVLDB*, 2010.
21. W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. In *PVLDB*, 2010.
22. W. Fan, J. Li, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.
23. D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, 1998.
24. B. Gallagher. Matching structure and semantics: A sur-

vey on graph-based pattern matching. *AAAI FS.*, 2006.
25. H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2009.
26. M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
27. T. Jiang and B. Ravikumar. Minimal NFA Problems are Hard. *SICOMP*, 22(6):1117–1141, 1993.
28. R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, 2010.
29. R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.
30. R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.
31. R. Kauhik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.
32. F. Mandreoli, R. Martoglia, and W. Penzo. Flexible query answering on graph-modeled data. In *EDBT*, 2000.
33. 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.
34. T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
35. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.
36. C. H. Papadimitriou. *Computational Complexity*. Addison -Wesley, 1994.
37. F. Ranzato and F. Tapparo. A new efficient simulation equivalence algorithm. In *LICS*, 2007.
38. R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, 2009.
39. D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
40. SPARQL. *http://www.w3.org/TR/rdf-sparql-query/*.
41. R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.
42. H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
43. J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.
44. 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.
45. F. Wei. TEDI: Efficient shortest path query answering on graphs. In *SIGMOD*, 2010.
46. Wikipedia. F-measure. *http://en.wikipedia.org/wiki/F-measure*.
47. P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.
48. L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. In *PVLDB*, 2009.