Adding Counting Quantifiers to Graph Patterns

Wenfei Fan\textsuperscript{1,2} Yinghui Wu\textsuperscript{3} Jingbo Xu\textsuperscript{1,2}
\textsuperscript{1}University of Edinburgh \textsuperscript{2}Beihang University \textsuperscript{3}Washington State University
\{wenfei@inf, jingbo.xu@\}.ed.ac.uk, yinghui@eecs.wsu.edu

ABSTRACT
This paper proposes quantified graph patterns (QGPs), an extension of graph patterns by supporting simple counting quantifiers on edges. We show that QGPs naturally express universal and existential quantification, numeric and ratio aggregates, as well as negation. Better still, the increased expressivity does not come with a much higher price. We show that quantified matching, \textit{i.e.}, graph pattern matching with QGPs, remains \textit{NP}-complete in the absence of negation, and is \textit{DP}-complete for general QGPs. We show how quantified matching can be conducted by incorporating quantifier checking into conventional subgraph isomorphism methods. We also develop parallel scalable algorithms for quantified matching. As an application of QGPs, we introduce quantified graph association rules defined with QGPs, to identify potential customers in social media marketing. Using real-life and synthetic graphs, we experimentally verify the effectiveness of QGPs and the scalability of our algorithms.

CCS Concepts
• Theory of computation → Pattern matching;

Keywords
Quantified graph patterns; graph association rules

1. INTRODUCTION
Given a graph pattern $Q(x_o)$ and a graph $G$, graph pattern matching is to find $Q(x_o,G)$, the set of matches of $x_o$ in subgraphs of $G$ that are isomorphic to $Q$. Here “query focus” $x_o$ is a designated node of $Q$ denoting search intent \cite{9}. Traditionally, pattern $Q$ is modeled as a (small) graph in the same form as $G$. This notion of patterns is used in social group detection and transportation network analysis.

However, in applications such as social media marketing, knowledge discovery and cyber security, more expressive patterns are needed, notably ones with counting quantifiers.

Example 1: (1) Consider an association rule for specifying regularities between entities in social graphs:

$\circ$ If (a) person $x_o$ is in a music club, and (b) among the people whom $x_o$ follows, at least 80\% of them like an album $y$, then the chances are that $x_o$ will buy $y$.

Its antecedent specifies conditions (a) and (b). If these two conditions hold, then we can recommend album $y$ to $x_o$. This is an example of social media marketing, which is predicted to trump traditional marketing. Indeed, empirical studies suggest that “90\% of customers trust peer recommendations versus 14\% who trust advertising” \cite{1}, and “the peer influence from one’s friends causes more than 50\% increases in odds of buying products” \cite{8}.

The antecedent is specified as a quantified graph pattern (QGP) $Q_1(x_o)$ shown in Fig. 1, where $x_o$ is its query focus, indicating potential customers. Here edge follow($x_o,z$) carries a counting quantifier “$\geq 80\%$”, for condition (b) above.

In a social graph $G$, a node $v_x$ matches $x_o$ in $Q_1$, \textit{i.e.}, $v_x \in Q_1(x_o,G)$, if (a) there exists an isomorphism $h$ from $Q_1$ to a subgraph $G'$ of $G$ such that $h(x_o) = v_x$, \textit{i.e.}, $G'$ satisfies the topological constraints of $Q_1$, and (b) among all the people whom $v_x$ follows, 80\% of them account for matches of $z$ in $Q_1(G)$, satisfying the counting quantifier.

The following association rules are also found useful in social media marketing, with various counting quantifiers:

$\circ$ If for all the people $z$ whom $x_o$ follows, $z$ recommends Redmi 2A (cell phone), then $x_o$ may buy a Redmi 2A.

$\circ$ If among the people followed by $x_o$, (a) at least $p$ of them recommend Redmi 2A, and (b) no one gives Redmi 2A a bad rating, then $x_o$ may buy Redmi 2A.

The antecedents of these rules are depicted in Fig. 1 as QGPs $Q_2(x_o)$ and $Q_3(x_o)$, respectively. Here $Q_2$ uses a universal quantification ($= 100\%$), while $Q_3$ carries numeric aggregate ($\geq p$) and negation ($= 0$). In particular, a node $v_z$ in $G$ matches $x_o$ in $Q_3$ only if there exists no node $v_w$ in $G$ such that follow($v_x,v_w$) is an edge in $G$ and there exists an edge from $v_w$ to Redmi 2A labeled “bad rating”. That is, counting quantifier “$= 0$” on edge follow($x_o, z_2$) enforces negation.

(2) Quantified graph patterns are also useful in knowledge discovery. For example, QGP $Q_4(x_o)$ of Fig. 1 is to find

$\circ$ all people who (a) are professors in the UK, (b) do not have a PhD degree, and (c) have at least $p$ former PhD students who are professors in the UK.

It carries negation ($= 0$) and numeric aggregate ($\geq p$).

These counting quantifiers are not expressible in traditional graph patterns. Several questions about QGPs are open. (1) How should QGPs be defined, to balance their expressive power and complexity? (2) Can we efficiently conduct graph pattern matching with QGPs in real-life graphs,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

\textcopyright{} 2016 ACM, ISBN 978-1-4503-3531-7/16/06...
\$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882937
which may have trillions of nodes and edges [20]? (3) How can we make use of QGPs in emerging applications? The need for studying these is highlighted in, e.g., social marketing, knowledge discovery and cyber security.

Contributions. This paper aims to answer these questions.

(1) We propose QGPs (Section 2). Using simple counting quantifiers, QGPs uniformly support numeric and ratio aggregates, universal and existential quantification, and negation. We formalize quantified matching, i.e., graph pattern matching with QGPs, by revising the traditional semantics of pattern matching to incorporate counting quantifiers.

(2) We establish the complexity of quantified matching (Section 3). We show that despite their increased expressivity, QGPs do not make our lives much harder: quantified matching is NP-complete in the absence of negation, the same as subgraph isomorphism; and it is DP-complete otherwise.

(3) We provide a quantified matching algorithm (Section 4). The algorithm unifies conventional pattern matching and quantifier verification in a generic search process, and handles negation by novel incremental evaluation IncQMatch. As opposed to conventional incremental settings, IncQMatch acts in response to changes in patterns, not in graphs, and is optimal by performing only necessary verification.

(4) We develop parallel algorithms for quantified matching (Section 5). We identify a practical condition under which quantified matching is parallel scalable, i.e., guaranteeing provable reduction in sequential running time with the increase of processors. Under the condition, we develop graph partition and QGP matching algorithms, both parallel scalable, by exploring inter and intra-fragment parallelism.

(5) As an application of QGPs, we introduce quantified graph association rules (QGARs; Section 6). QGARs help us identify potential customers in social graphs, and (positive and negative) correlations in knowledge graphs. We propose support and confidence metrics for QGARs, a departure from their conventional counterparts. We also show that the (parallel) quantified matching algorithms can be readily extended to identify interesting entities with QGARs.

(6) Using real-life and synthetic graphs, we experimentally verify the effectiveness of QGPs and the scalability of our algorithms (Section 7). We find the following. (a) Quantified matching is feasible on large graphs. It takes 125 seconds on graphs of 150 millions nodes and edges by using 4 processors, and 42.3 seconds with 20 processors. (b) Our matching (resp. partition) algorithm is parallel scalable: it is on average 2.8 (resp. 3.5) and 3.2 (resp. 2.5) times faster on real-life social and knowledge graphs, respectively, when the number of processors increases from 4 to 20. (c) QGARs capture behavior patterns in social and knowledge graphs that cannot be expressed with conventional graph patterns.

We contend that QGPs and QGARs are useful in emerging applications such as social marketing and knowledge discovery. Despite the increased expressivity, they yield practical tools over large real-world graphs, which can be built upon existing (parallel) graph analytic systems.

Proofs and optimization strategies are given in Appendix.

Related work. We categorize the related work as follows.

Quantified graph querying. The need for counting in graph queries has long been recognized. SPARQLog [28] extends SPARQL with first-order logic (FO) rules, including existential and universal quantification over node variables. Rules for social recommendation are studied in [30], using support count as constraints. QGRAPH [10] annotates nodes and edges with a counting range (count 0 as negated edge) to specify the number of matches that must exist in a database. Set regular path queries (SRPQ) [31] extends regular path queries with quantification for group selection, to restrict the nodes in one set connected to the nodes of another. For social networks, SocialScope [5] and SNQL [32] are algebraic languages with numeric aggregates on node and edge sets.

The study of QGPs is to strike a balance between the expressivity and the complexity. It differs from the prior work in the following. (1) Using a uniform form of counting quantifiers, QGPs support numeric and ratio aggregates (e.g., at least p friends and 80% of friends), and universal (100%) and existential quantification (≥ 1). In contrast, previous proposals do not allow at least one of these. (2) We focus on graph pattern queries, which are widely used in social media marketing and knowledge discovery; they are beyond set regular expressions [31] and rules of [30]. (3) Quantified matching with QGPs is DP-complete at worst, slightly higher than conventional matching (NP-complete) in the polynomial hierarchy [33]. In contrast, SPARQL and SPARQLog are PSPACE-hard [28], and SRPQ takes EXPTIME [31]; while the complexity bounds for QGRAPH [10], SocialScope [5] and SNQL [32] are unknown, they are either more expensive than QGPs (e.g., QGRAPH is a fragment of FO(count)), or cannot express numeric and ratio quantifiers [5,32]. (4) No prior work has studied parallel scalable algorithms for its queries.

Parallel pattern matching. A number of (parallel) matching algorithms have been developed for subgraph isomorphism [22, 27, 35]. None of these addresses quantifiers. In contrast, (1) in the same general framework [27] used by these methods, our sequential quantified matching algorithms cope with quantifiers and negated edges without incurring considerable cost; and (2) our parallel scalable algorithms exploit both inter and intra-fragment parallelism for effective quantifier verification in QGP evaluation.

Various strategies have been studied for graph partition [6, 11, 23]. This work differs from the prior work in the following. (1) We propose a d-hop preserving partition scheme such that the d-hop neighbor of each node is contained in a fragment, and that all fragments have an even size, with an approximation bound. Closest to ours is the n hop-guarantee partition [22]. However, [22] provides no approxi-
mation bound to ensure both d-hop preserving and balanced fragment sizes, especially for nodes with a high degree. (2) We propose a partition algorithm that is parallel scalable, a property that is not guaranteed by the prior strategies.

**Quantified association rules.** Association rules [3] are traditionally defined on relations of transaction data. Over relations, quantified association rules [38] and ratio rules [25] impose value ranges or ratios (e.g., the aggregated ratio of two attribute values) as constraints on attribute values. There has also been recent work on extending association rules to social networks [30, 36] and RDF knowledge bases, which resorts to mining conventional rules and Horn rules (as conjunctive binary predicates) [17] over tuples with extracted attributes from social graphs, instead of exploiting graph patterns. Closer to this work is [16], which defines association rules directly with patterns without quantifiers.

Our work on QGARs differs from the previous work in the following. (1) As opposed to [3, 25, 38], QGARs extend association rules from relations to graphs. They call for topological support and confidence metrics, since the conventional support metric is not anti-monotonic in graphs. (2) QGARs allow simple yet powerful counting quantifiers to be imposed on matches of graph patterns, beyond attribute values. In particular, rules of [25, 38] cannot express universal quantification and negation. When it comes to graphs, (3) the rules of [16] cannot express counting quantifiers, and limits their consequent to be a single edge, and (4) applying QGPs and QGARs becomes an intractable problem, as opposed to PTIME for conventional rules in relations.

2. **QUANTIFIED GRAPH PATTERNS**

We next introduce quantified graph patterns QGPs. To define QGPs, we first review conventional graph patterns.

2.1 **Conventional Graph Pattern Matching**

We consider labeled, directed graphs, defined as \( G = (V, E, L) \), where (1) \( V \) is a finite set of nodes; (2) \( E \subseteq V \times V \) is a set of edges, in which \((v, v')\) denotes an edge from node \( v \) to \( v' \); and (3) each node \( v \) in \( V \) (resp. edge \( e \) in \( E \)) carries \( L(v) \) (resp. \( L(e) \)), indicating its label or content as commonly found in social networks and property graphs.

Two example graphs are depicted in Fig. 2.

We review two notions of subgraphs. (1) A graph \( G' = (V', E', L') \) is a subgraph of \( G = (V, E, L) \), denoted by \( G' \subseteq G \), if \( V' \subseteq V \), \( E' \subseteq E \), and for each edge \( e \in E' \) (resp. node \( v \in V' \)), \( L'(e) = L(e) \) (resp. \( L'(v) = L(v) \)). (2) We say that \( G' \) is a subgraph induced by a set \( V' \) of nodes if \( G' \subseteq G \) and \( E' \) consists of all the edges in \( G \) whose endpoints are in \( V' \).

**Patterns.** A graph pattern is traditionally defined as a graph \( Q(x_o) = (V_Q, E_Q, L_Q) \), where (1) \( V_Q \) (resp. \( E_Q \)) is a set of pattern nodes (resp. edges), (2) \( L_Q \) is a function that assigns a node label \( L_Q(u) \) (resp. edge label \( L_Q(e) \)) to each pattern node \( u \in V_Q \) (resp. edge \( e \in E_Q \)), and (3) \( x_o \) is a node in \( V_Q \), referred to as the query focus of \( Q \), for search intent.

**Pattern matching.** A match of pattern \( Q \) in graph \( G \) is a bijective function \( h \) from nodes of \( Q \) to nodes of a subgraph \( G' \) of \( G \), such that (a) for each node \( u \in V_Q \), \( L_Q(u) = L(h(u)) \), and (b) \((u, u')\) is an edge in \( Q \) if and only if \((h(u), h(u'))\) is an edge in \( G' \), and \( L_Q(u, u') = L(h(u), h(u')) \). From \( h \), subgraph \( G' \) can be readily deduced.

We denote by \( Q(G) \) the set of matches of \( Q \) in \( G \), i.e., the set of bijective functions \( h \) that induce a match of \( Q \) in \( G \).

Query answer is the set of all matches of \( x_o \) in \( Q(G) \).

Given \( Q(x_o) \) and \( G \), graph pattern matching is to compute \( Q(x_o, G) \), i.e., all matches of query focus \( x_o \) in \( G \) via \( Q \).

2.2 **Quantified Graph Patterns**

We next define QGPs, by extending conventional graph patterns to express quantified search conditions.

**Syntax.** A quantified graph pattern (QGP) \( Q(x_o) \) is defined as \((V_Q, E_Q, L_Q, f)\), where \( V_Q, E_Q, L_Q \) and \( x_o \) are the same as their traditional counterparts, and \( f \) is a function such that for each edge \( e \in E_Q \), \( f(e) \) is a predicate of

- a positive form \( \sigma(e) \circ p\% \) for a real number \( p \in (0, 100) \),
- or \( \sigma(e) \circ p \) for a positive integer \( p \), or
- or \( \sigma(e) = 0 \), where \( e \) is referred to as a negated edge.

Here \( \circ \) is either \( = \) or \( \geq \), and \( \sigma(e) \) will be elaborated shortly.

We refer to \( f(e) \) as the counting quantifier of \( e \), and \( p\% \) and \( p \) as ratio and numeric aggregate, respectively.

Counting quantifiers express logical quantifiers as follows:

- negation when \( f(e) = \sigma(e) = 0 \) (e.g., \( Q_1 \) in Example 1);
- existential qualification if \( f(e) = \sigma(e) \geq 1 \); and
- universal quantifier if \( f(e) = \sigma(e) = 100\% \) (e.g., \( Q_2 \)).

A conventional pattern \( Q \) is a special case of QGP when \( f(e) = \sigma(e) \geq 1 \) for all edges \( e \) in \( Q \), i.e., if \( Q \) has existential quantification only. We leave out \( f(e) \) if it is \( \sigma(e) \geq 1 \).

We call a QGP \( Q \) positive if it contains no negated edges (i.e., edges \( e \) with \( \sigma(e) = 0 \)), and negative otherwise.

**Example 2:** Graph patterns \( Q_1, Q_2 \) given in Example 1 are QGPs with various counting quantifiers, e.g., (1) edge \((x_o, z)\) in \( Q_1 \) has a quantifier \( \sigma(x_o, z) \geq 80\% \); (2) \( Q_2 \) has a universal quantifier \( \sigma(x_o, z) = 100\% \) on edge \((x_o, z)\), and an existential quantifier for edge \((z, \text{Redmi} 2A)\); and (3) \( Q_3 \) has a negated edge \((x_o, z_2)\) with \( \sigma(x_o, z_2) = 0 \). Among the QGPs, \( Q_1 \) and \( Q_2 \) are positive, while \( Q_3 \) and \( Q_1 \) are negative.

**Remark.** To strike a balance between the expressive power and the complexity of pattern matching with QGPs in large-scale graphs, we assume a predefined constant \( l \) such that on any simple path (i.e., a path that contains no cycle) in \( Q(x_o) \), (a) there exist at most \( l \) quantifiers that are not existential, and (b) there exist no more than one negated edge, i.e., we exclude “double negation” from QGPs.

The reason for imposing the restriction is twofold. (1) Without the restriction, quantified patterns would be able to express first-order logic (FO) on graphs. Indeed, FO sentences \( P_1 X_1 \ldots P_l X_l \) can be encoded in such a pattern, where \( P_i \) is either \( \lor \) or \( \exists \), \( \lor \) is a logic formula, and \( l \) is unbounded. Such patterns inherit the complexity of FO [29], in addition to #P complication. Then even the problem for deciding whether there exists a graph that matches such a pattern is beyond reach in practice. As will be seen shortly, the restriction makes QGPs discovery and evaluation feasible in
large-scale graphs. (2) Moreover, we find that QGPs with the restrictions suffice to express quantified patterns commonly needed in real-life applications, for small $l$. Indeed, empirical study suggests that $l$ is at most 2, and “double negation” is rare, as “99% of real-world queries are star-like” [18].

One can extend $f(e)$ to support $>$, $\neq$ and $\leq$ as $\circ$ and conjunctions of predicates. To simplify the discussion, we focus on QGPs $Q(x_o)$ in the simple form given above.

Semantics. We next give the semantics of QGPs. We consider positive QGPs first, and then QGPs with negation.

Positive QGPs. We use the following notations. Stripping all quantifiers $f(e)$ off from a QGP $Q(x_o)$, we obtain a conventional pattern, referred to as the stratified pattern of $Q(x_o)$, and denoted by $Q_s(x_o)$. Consider an edge $e = (u, v')$ in $Q(x_o)$, a graph $G$ and nodes $u$ and $v$ in $G$. When $x_o$ is mapped to $v$, we define the set of children of $v$ via $e$ and $Q$, denoted by $M_e(u,v,Q)$ when $G$ is clear from the context:

$$\{v' \mid h \in Q_s(u), h(x_o) = v, h(e) = (v,v')\}$$

i.e., the set of children of $v$ that match $u'$ when $u$ is mapped to $v$, subject to the constraints of $Q_s$. Abusing the notion of isomorphic mapping, $h(e) = (v,v')$ denotes $h = h(v)$, $h(u') = h'(v')$, $(v,v') \in G$ and $Q(x_o, u', u) = L(v,v')$.

We define $M_e(u,v,Q) = \{v' \mid (v,v') \in G, L(v,v') = L_Q(e)\}$, the set of the children of $v$ connected by an edge $e$.

For a positive QGP $Q(x_o)$, a match $h_0 \in Q(G)$ satisfies the following conditions: for each node $u$ in $Q$ and each edge $e = (u, u')$ in $Q$,

- if $f(e)$ is $\sigma(e) \circ p\%$, then $|M_e(h_0(u),h_0(u'),Q)| \circ p\%$, in terms of the ratio of the number of children of $v$ via $e$ and $Q$ to the total number of children of $v$ via $e$; and
- if $f(e)$ is $\sigma(e) \circ p$, then $|M_e(h_0(u),h_0(u'),Q)| \circ p$, in terms of the number of children of $v$ via $e$ and $Q$.

That is, $\sigma(e)$ is defined as the ratio $|M_e(h_0(u),h_0(u'),Q)|/|\text{cardinality of } M_e(h_0(u),h_0(u'),Q)|$ or cardinality $|M_e(h_0(u),h_0(u'),Q)|$, for $p\%$ or $p$, respectively. Intuitively, $\sigma(e)$ requires that at least $p\%$ of nodes or $p$ nodes in $M_e(h_0(v))$ are matches for $u'$ when $v$ is mapped to $u$. A match in $Q(G)$ must satisfy the topological constraints of $Q_s$ and moreover, the counting quantifiers of $Q$. Note that the counting quantifier on edge $e = (u, u')$ is applied at each match $h_0(u)$ of $u$, to enforce the semantics of counting.

We denote by $Q(u,G)$ the set of matches of a pattern node $u$, i.e., nodes $v = h(u)$ induced by all matches $h$ of $Q$ in $G$. The query answer of $Q(x_o)$ in $G$ is defined as $Q(x_o,G)$.

Example 3: For graph $G_1$ in Fig. 2 and QGP $Q_2$ of Example 1, $Q_2(x_o,G_1) = \{x_1, x_2\}$. Indeed, 100% of the friends of $x_1$ and $x_2$ recommend Redmi 2A. More specifically, for pattern edge $e = \text{follow}(x_o, z)$, when $x_o$ is mapped to $x_1$ via $h_0$, $M_e(h_0(x_1), x_1, Q) = \{v_0\}$, which is the set $M_e(x_1)$ of all people whom $x_1$ follows; similarly when $x_o$ is mapped to $x_2$. In contrast, while $x_3$ matches $x_o$ via the stratified pattern of $Q_2$, $x_3 \not\in Q_2(x_o,G_1)$ since at least one user whom $x_3$ follows (i.e., $v_1$) has no edge to Redmi 2A.

Table 1: Notations used in the paper

Negative QGPs. To cope with QGP $Q(x_o)$ with negated edges, we define the following: (1) $\Pi(Q)$: the QGP induced by those nodes in $Q(x_o)$ that are connected to $x_o$ (via a path from or to $x_o$) with non-negated edges in $Q(x_o)$, i.e., $\Pi(Q)$ excludes all those nodes connected via at least one negated edge; (2) $Q^{\sim e}$, obtained by “positifying” a negated edge $e$ in $Q$, i.e., by changing $f(e)$ from $\sigma(e) = 0$ to $\sigma(e) \geq 1$; and (3) $E_0$, the set of all negated edges in $Q$.

Then in a graph $G$, query answer to $Q(x_o)$ is defined as $Q(x_o,G) = \Pi(Q)(x_o,G) \setminus (\bigcup_{e \in E_0} \Pi(Q^{\sim e})(x_o,G))$.

That is, we enforce negation via set difference. One can verify that for each node $u$ in $Q$ and each negated edge $e = (u, u')$ in $Q$, $|M_e(h_0(x_o),h_0(u),Q)| = 0$.

Example 4: Consider $G_1$ and $Q_3$ of Example 1 with $p=2$. Pattern $\Pi(Q_3)$, which excludes the negated edge $e_1 = (x_2, z_2)$ in $Q_2$ and $\Pi(Q_3^{\sim e_2})$, which “positifies” $e_2$ in $Q_2$, are shown in Fig. 3. One can verify the following: (1) $\Pi(Q_3)(x_2, z_2)$ is $\{x_2, z_2\}$; note that $x_1$ is not a match since only 1 user whom $x_1$ follows recommends Redmi 2A, and hence violates the counting quantifier $\geq p$; and (2) $\Pi(Q_3^{\sim e_2})$ is $\{x_3\}$, which is a “negative” instance for $Q_3$. Hence, $\Pi(Q_3)(x_1, z_1) = \{x_2, x_3\}$, where $x_3$ is excluded since he/x follows $v_3$, who gave a bad rating on Redmi 2A, i.e., violating the negation $\sigma(e) = 0$.

Similarly, for QGP $Q_1$ and graph $G_2$ of Fig. 2, when $p=2$, $Q_1(x_2, z_2)$ is $\{x_2, x_2\}$. Note that $x_2$ matches the stratified pattern of $Q_2$, but it violates the negation on $x_2$ (PhD), which requires that matches of $x_2$ must not be a PhD.

As another example, consider $Q_2(x_3)$ with two negated edges $e_1 = (\text{Prof}, UK)$ and $e_2 = (z, PhD)$. It is to find non-UK professors who supervised students who are professors but have no PhD degree. As shown in Fig. 3, $\Pi(Q_2)$ finds professors who supervised students who are professors. In contrast, $\Pi(Q_2^{\sim e_2})$ finds such professors in the UK, and $\Pi(Q_2^{\sim e_2})$ (not shown) retrieves professors with students who are professors and have a PhD. In a graph $G$, $Q(x_o,G) = \Pi(Q_3)(x_o,G) \setminus (\Pi(Q_3^{\sim e_2})(x_o,G) \cup \Pi(Q_3^{\sim e_3})(x_o,G))$.

The notations of this paper are summarized in Table 1.

3. THE COMPLEXITY OF QUANTIFIED MATCHING

In the next three sections, we study quantified matching:

- **Input:** A QGP $Q(x_o)$ and a graph $G$.
- **Output:** $Q(x_o,G)$,

to compute the set of all matches of query focus $x_o$ in $G$. We start with its complexity in this section.

Decision problem. Its decision problem, referred to as the quantified matching problem, is stated as follows.

- **Input:** A QGP $Q(x_o)$, a graph $G$ and a node $v$ in $G$.
Lemma 4: encoded as numeric aggregates by transforming ratio aggregates, for instance, a brute-force approach invokes an NP algorithm that calls a #P oracle to check the ratio aggregate. We show that while the increased expressive power of QGPs comes with a price, their complexity bound does not get much higher. In particular, #P is not necessary.

Theorem 1: The quantified matching problem remains NP-complete for positive QGPs, and it becomes DP-complete for (possibly negative) QGPs.

Here DP is the class of languages recognized by oracle machines that make a call to an NP oracle and a call to a coNP oracle. That is, L is in DP if there exist languages $L_1 \in NP$ and $L_2 \in coNP$ such that $L = L_1 \cap L_2$.

Is in the polynomial hierarchy $[33]$. The upper bounds are verified by the next two lemmas. The lower bounds follow from the stronger results below, which are in turn verified by reductions from SUBGRAPH ISOMORPHISM and EXACT-CLIQUE, which are NP-complete and DP-complete, respectively (cf. [33]).

Lemma 2: For QGPs with numeric aggregates only, the quantified matching problem is NP-hard for positive QGPs, and DP-hard for (possibly negative) QGPs.

The upper bounds are verified by the next two lemmas. In particular, Lemma 4 shows that ratio aggregates can be encoded as numeric aggregates by transforming both query $Q$ and graph $G$, in PTIME. This explains why positive QGPs with ratio aggregates retain the same complexity as conventional patterns, despite their increased expressivity.

Lemma 3: For QGPs with numeric aggregates only, the quantified matching problem is in NP for positive QGPs, and is in DP for (possibly negative) QGPs.

Lemma 4: Any QGP $Q(x_o)$ and graph $G$ can be transformed in PTIME to QGP $Q_d(x_o)$ with numeric aggregates only and graph $G_d$, respectively, such that $Q(x_o, G) = Q_d(x_o, G_d)$.

Remark. As QGPs with quantifiers bearing $\leq, \neq$ and $<$ subsume the case when $\sigma(e) = 0$, quantified matching is DP-hard for such QGPs. Due to the space limit, we focus on $\geq, =, >$ and leave a full treatment to future work.

4. ALGORITHMS FOR QUANTIFIED MATCHING

We next provide an algorithm, denoted by QMatch, for quantified matching. It takes a QGP $Q(x_o)$ and a graph $G$ as input, and computes $Q(x_o, G)$ as output. It extends existing algorithms $T$ for conventional subgraph isomorphism, to incorporate quantifier checking and process negated edges.

Generic graph pattern matching. We start by reviewing a generic procedure for subgraph isomorphism, denoted by Match and shown in Fig. 4, slightly adapted from [27] to output $Q(x_o, G)$ for query focus $x_o$. As observed in [27], state-of-the-art graph pattern matching algorithms $T$ typically adopt Match, and differ only in how to optimize key functions (e.g., SelectNext, IsExtend; see below). Given a traditional pattern $Q(x_o)$ and a graph $G$, Match initializes $Q(x_o, G)$, as well as a partial match $M$ as a set of node pairs (line 1). Each pair $(u, v)$ in $M$ denotes that a node from $G$ matches a pattern node $u$ in $Q$. It identifies a candidate match set $C(u)$ for each pattern node $u$ in $Q$ (lines 3-4) (FilterCandidate). If there exists a pattern node $u$ with no candidate, it returns $\emptyset$ (line 4). Otherwise, it invokes SubMatch to compute all matches (isomorphic mappings) $Q(G)$ (lines 5). It then computes and returns query answer $Q(x_o, G)$ from mappings $h \in Q(G)$ (lines 6-8).

Figure 4: Generic search procedure Match

Procedure SubMatch terminates with steps (1) and (2) (line 7), and restores it when SubMatch backtracks (line 8). If $M$ is a valid isomorphism (Verify, line 1), it adds $M$ to $Q(G)$ (line 2). This continues until $Q(G)$ is completed.

4.1 Quantified Graph Pattern Matching

Algorithm QMatch revises the generic Match to process quantifiers. (1) It first adopts a dynamic selection and pruning strategy to compute $\Pi(Q)(x_o, G)$. The dynamic search picks top $p$ promising neighbors based on a potential score, with $p$ adapted to the corresponding quantifiers. (2) It then employs optimal incremental evaluation to process negated edges, which maximally reuses cached matches for $\Pi(Q)$ when processing $Q^+ e$ for positified $e$, instead of recomputing $Q^+(G)$ starting from scratch. The strategies are supported by optimized data structures and key functions from Match.

Auxiliary structures. QMatch maintains auxiliary structures for each node $v$ in $C(u)$ as follows: (1) A Boolean variable $X(u, v)$ indicating whether $v$ is a match of $u$ via isomorphism from $Q(G)$ to $G$, and (2) a vector $T$, where entry $T(v, e)$ for an edge $e=(u, u')$ in $Q$ is a pair $(c(e, v), U(v, e))$, in which $c$ (resp. $U$, initialized as $M_u(v)$) records the current size (resp. an estimate upper bound for $|M_u(v, G)|$).
Algorithm QMatch

Input: a QGP $Q(x_o)$, graph $G$
Output: the answer set $Q(x_o, G)$
1. $Q(x_o, G):=\emptyset$; $M:=\emptyset$; $\Pi(Q)(x_o, G):=\emptyset$; $\Pi(Q)(G):=\emptyset$;
2. for each $u$ of $Q$ do
3. initialize $C'(u)$ and auxiliary structures;
4. $\Pi(Q)(x_o, G):=\operatorname{DMatch}(\Pi(Q)(G), M, \Pi(Q)(G))$;
5. for each negative edge $e$ in $E_Q$ do
6. $Q^+e:=\operatorname{IncQMatch}((\Pi(Q)(x_o, G), Q^+e))$;
7. $Q(x_o, G):=\Pi(Q)(x_o, G) \setminus \bigcup\in E_Q Q^+e(x_o, G)$;
8. return $Q(x_o, G)$;

Figure 5: Algorithm QMatch

Algorithm. Algorithm QMatch (outlined in Fig. 5) revises Match to process QGP $Q(x_o)$ in three steps. (1) It first initializes the candidate set and auxiliary structures with a revised FilterCandidate (lines 1-3). For each pattern node $u$ in $Q(x_o)$, it initializes (a) $C'(u)$ with nodes $v$ of the same label, and (b) $X(u, v) = \bot$, $c(v, e) = 0$ and $U(v, e) = |M_e|$, for each $e = (u, v)$ in $Q$. It removes $v$ from $C'(u)$ if $U(v, e)$ does not satisfy the quantifier of $e$. (2) It then invokes a procedure DMatch revised from SubMatch in Fig. 4 to compute $\Pi(Q)(x_o, G)$ (line 4). (3) It then processes each negated edge $e$ by constructing its positified pattern $Q^+$, and computes $Q^{++}(x_o, G)$ with an incremental procedure IncQMatch (lines 5-6). (4) It computes $Q(x_o, G)$ by definition (line 7). We now present DMatch, and defer IncQMatch to Section 4.2.

Example 5: Given $Q_3$ with $p=2$ (Fig. 1) and $G_1$ (Fig. 2), QMatch first computes $\Pi(Q_3)(x_o, G_1)$ (Fig. 3). It initializes variables for nodes in $G_1$, partially shown below ($i \in [0, 4]$).

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$e$</th>
<th>$c$</th>
<th>$c(x_i, z_1)=0$</th>
<th>$e(x_i, z_1)=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td></td>
<td></td>
<td>$U(x_1, (x_1, z_1))=1$</td>
<td>$U(x_1, (x_1, z_1))=1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td></td>
<td></td>
<td>$U(x_2, (x_2, z_2))=2$</td>
<td>$U(x_2, (x_2, z_2))=2$</td>
</tr>
<tr>
<td>$x_3$</td>
<td></td>
<td></td>
<td>$U(x_3, (x_3, z_3))=3$</td>
<td>$U(x_3, (x_3, z_3))=3$</td>
</tr>
<tr>
<td>$x_4$</td>
<td></td>
<td></td>
<td>$U(x_4, (x_4, z_4))=1$</td>
<td>$U(x_4, (x_4, z_4))=1$</td>
</tr>
</tbody>
</table>

At this stage, since $U(x_1, (x_1, z_1))=1 \leq 2$, $x_1$ fails the quantifier of $(x_1, z_1)$, and is removed from $C(x_o)$.

Procedure DMatch. Given positive QGP $\Pi(Q)$, DMatch revises SubMatch (Fig. 4) by adopting dynamic search. To simplify the discussion, we consider numeric $\sigma(e) \odot p$ first. (1) Given a selected pattern node $u'$ (line 3 of SubMatch), a candidate $v \in C'(u)$, and an edge $e=(u', u')$ with quantifier $\sigma(e) \odot p$, DMatch dynamically finds $p$ best nodes (recorded in a heap $SP(u')$) from $C(u')$ that are children of $v$ (lines 4-5 of SubMatch, IsExtend), using selection and pruning rules (see Appendix B). Denote as $P(u')$ the parent set of $u'$ in $G$, the potential of a match $v' \in C'(u')$ is defined as:

$$\left(1 + \frac{|P(u') \cap C(u)|}{|C(u)|}\right) \cdot \sum_{e=(u', u')} \frac{U(v, e)}{p_e},$$

where $p_e$ is the number in $\sigma$ or $p_e$ for edge $e=(u', u')$. It favors those candidates that (a) benefit the verification of more candidates during future backtracking, and (b) have higher upper bounds $w.r.t.$ $p$ (hence more likely to be a match itself). We select candidates with the highest scores.

DMatch then updates $M$ by including $(u, v)$, and recursively conducts the next level of search by forking $p$ verifications in the order of the selected $p$ candidates (line 7, SubMatch). It keeps a record of $M$ and a cursor to memorize the candidates in $SP$ for backtracking, using a stack.

(2) When backtracking to a candidate $v \in SP(u)$ from a child $v'$ of $v$, DMatch restores $M$ and the cursor (Restore, line 8 of SubMatch). It then dynamically updates $SP(u)$. (a) If $X(u', v')=\bot$, it reduces $U(v, e)$ by $1$. (b) It applies the selection and pruning rules to $C(u)$ using the updated potentials $w.r.t.$ the changes in (a). If the upper bound $U(v, e)$ fails the quantifier of $e$, $v$ is removed from $C(u)$ and $SP(u)$ without further verifying its other children. Otherwise, it picks a new set $SP(u)$ of candidates with top potentials.

(3) When $M$ is complete, i.e., each node $u$ in $\Pi(Q)$ has a match in $M$, DMatch checks whether $M$ is an isomorphic mapping. If so, it updates $X(u, v)=\bot$ for each pair $(u, v) \in M$, and increases the counter $c(v, e)$. It then checks whether the counters satisfy the quantifiers of $\Pi(Q)$. If so, it adds $v_u$ to $Q(x_o, G)$, Otherwise, it proceeds. DMatch terminates when all the candidates of $x_o$ are checked.

Example 6: Continuing with Example 5, given $Q(x_o) = \{x_2, x_3\}$, DMatch selects $x_2$, and extends $M$ with $(x_o, x_2)$. In contrast to Fig. 4, DMatch picks top 2 best candidates $SP(z_1) = \{v_2, v_3\}$ from $C(z_1)$ following edge $e=(x, z_1)$. This adds $(z_1, v_2)$ to $M$, and (Redmi 2A, Redmi 2A) for the next round. At verification, it finds $M$ a complete isomorphism, and updates $X(v_o, x_2)=\bot$ and $c(x_2, e)=1$. As $x_2$ cannot be verified as a match via $\Pi(Q_3)$ yet, DMatch next verifies $v_1$, and sets $c(x_2, e)=2$. As $x_2$ is a match and has a counter satisfying the quantifier, it is added to $\Pi(Q_3)(x_o, G_1)$. The updated variables for candidates of $C(x_o)$ are as follows.

<table>
<thead>
<tr>
<th>$x_o$</th>
<th>$x(x_o, x_2)=true$</th>
<th>$x(x_o, x_3)=true$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2(x_o, x_2)=1$</td>
<td>$x_2(x_o, x_3)=2$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$x_3(x_o, x_3)=2$</td>
<td>$x_3(x_o, x_3)=3$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$x_4(x_o, x_4)=1$</td>
<td>$x_4(x_o, x_4)=1$</td>
</tr>
</tbody>
</table>

DMatch next verifies $x_3$. It starts by selecting top 2 candidates $SP(z_3)=\{v_2, v_3\}$. Once $v_3$ is processed, it finds that $x_3$ is in an isomorphism with $c(x_3, e)=2$, and hence is a match. It returns $\{x_2, x_3\}$ as $\Pi(Q_3)(x_o, G_1)$. □

One can readily verify the following (see Appendix A).

Lemma 5: DMatch computes $\Pi(Q)(x_o, G)$ by (a) verifying no more candidates than any Match-based subgraph isomorphism algorithm $T$, and (b) with space cost $O(p_o|Q|+|V|)$, where $p_o$ is the largest constant in all quantifiers of $Q$. □

That is, quantified matching can be evaluated following conventional $T$ without incurring significant extra time and space cost. The performance of DMatch is further improved by selection and pruning rules presented in Appendix B.

Ratio aggregates. DMatch can be readily extended to process ratio aggregates. Indeed, for each pattern $e=(u', u')$ with $\sigma(e) \odot p$ and at a candidate $u$ of $u$, DMatch “transforms” the quantifier to its equivalent numeric counterpart $\sigma(e) \odot p$, as follows. (a) DMatch computes $|M_e(u)|$ by definition. (2) It sets $p' = |M_e(u)| \cdot p$. The transformation for $e$ preserves all the exact matches for ratio quantifiers by definition, and takes a linear scan of $G$ (in $O(|G|)$ time). In addition, QMatch easily extends to QGPs with quantifiers $\sigma(e) \odot p$, by replacing it with $\sigma(e) \geq p + 1$.

4.2 Incremental Quantified Matching

If $\Pi(Q)(x_o, G)$ is nonempty, QMatch proceeds to compute $\Pi(Q')(x_o, G)$ for each negated edge $e \in E_Q$ (lines 5-6, Fig. 5). Observe the following: (1) $\Pi(Q')=\Pi(Q) \cup \Delta E$, i.e., $\Pi(Q')$ “expands” $\Pi(Q)$ with a set $\Delta E$ of positive edges; and (2) for any node $u$ in $\Pi(Q)$, $\Pi(Q')(u, G) \subseteq \Pi(Q)(u, G)$, since $\Pi(Q')$ adds more constraints to $\Pi(Q)$.

This observation motivates us to study a novel incremental quantified matching problem. Given a graph $G$, a QGP $Q$,
computed matches \(Q(u, G)\) for each \(u\) in \(Q\), and a new QGP \(Q' = Q \oplus \Delta E\), it is to compute \(Q'(x_o, G) = Q(x_o, G) \oplus \Delta O\), i.e., to find changes \(\Delta O\) in the output. It aims to make maximum use of cached results \(Q(u, G)\), instead of computing \(Q'(x_o, G)\) from scratch. As opposed to conventional incremental problems [15, 34], we compute \(\Delta O\) in response to changes in \(Q\), rather than to changes in graph \(G\).

As observed in [34], the complexity of incremental graph problems should be measured in the size of affected area, which indicates the amount of work that is necessarily performed by any algorithm for the incremental problem. For pattern matching via subgraph isomorphism, the number of verifications is typically the major bottleneck. Below we identify affected area for quantified matching, to characterize the optimality of incremental quantified matching.

**Optimal incremental quantified matching.** Given \(Q\) and \(\Pi(Q^{+e})\), the affected area is defined as

\[
\text{AFF} = \bigcup C(u_i) \cup \{ N(v) \mid v \in C(u_i) \},
\]

where (1) \(u_i\) is in edge \(e_i = (u_i, u'_i)\) or \(u'_i, u_i\) for each \(e_i \in \Delta E\); (2) \(C(u_i)\) includes (a) the match sets cached after \(\text{DMatch}\) processed \(\Pi(Q)\); and (b) the candidate sets initialized by \(\text{QMatch}\) (line 3 of Fig 5) for new nodes \(u_i\) introduced by \(\Pi(Q^{+e})\), which have to be checked; and (c) \(N(v)\) is the set of nodes in cached \(C(.)\) that are reachable from (or reached by) \(v\), via paths that contains only the nodes in \(C(.)\).

An incremental quantified matching algorithm is *optimal* if it incurs \(O(|\text{AFF}|)\) number of verifications. Intuitively, AFF is the set of nodes that are necessarily verified in response to \(\Delta E\), for any such algorithms to find exact matches.

**Proposition 6:** There exists an incremental algorithm that computes each \(\Pi(Q^{+e})(x_o, G)\) by conducting at most \(|\text{AFF}|\) rounds of verification.

As a proof, we present an optimal algorithm **IncQMatch**.

**Procedure IncQMatch.** Algorithm **IncQMatch** (used in line 6, Fig. 5) incrementally computes \(\Pi(Q^{+e})(x_o, G)\) by reusing the cached match sets and the counters computed in the process of \(\text{DMatch}\) for \(\Pi(Q)\). It works as follows.

1. **IncQMatch** initializes \(\Pi(Q^{+e})(u, G)\) for each \(u\) with the cached matches \(\Pi(Q)(u, G)\). It then computes the edge set \(\Delta E\) in \(\Pi(Q^{+e})(x_o, G)\) to be “inserted” into \(\Pi(Q)\).

2. **IncQMatch** then iteratively processes the edges \(e = (u, u')\) in \(\Delta E\). It first identifies those cached matches that are affected by the insertion. It considers two possible cases below.
   - Both \(u\) and \(u'\) are in \(\Pi(Q)\). For each match \(v \in \Pi(Q)(u, G)\), **IncQMatch** adds \(v\) to \(\text{AFF}\) and verifies whether \(v\) matches \(u\) via isomorphism following \(\text{DMatch}\). If \(X(u, v) = \text{true}\), it counts \((v, e)\) as the number of \(v\)'s children \(u'\) that are matches of \(u'\) and checks the quantifier of \(e\). If \((v, e)\) satisfies the quantifier, no change happens. Otherwise \((v, e)\) fails the quantifier. **IncQMatch** removes \(v\) from \(\Pi(Q^{+e})(u, G)\).
   - One or both of nodes \(u\) and \(u'\) are not in \(\Pi(Q)\). For the new node \(u\) (or \(u'\)), **IncQMatch** treats \(e\) as a single edge pattern and verifies each candidate \(v_1\) in \(C(v)\). Since \(\Pi(Q)\) is a “sub-pattern” of \(\Pi(Q^{+e})\) and \(\Pi(Q^{+e})\) is connected, we need only to inspect those matches \(v_1\) reachable from some nodes in cached \(C(.)\), i.e., \(v_1 \in N(v_2)\) for some cached \(v_2\); hence \(v_1 \in \text{AFF}\).

For each \(v\) removed in the steps above, **QMatch** then propagates the impact recursively by (a) reducing all the counters of \(v\)'s parents by 1, and (b) removing invalid matches due to the updated counters and adds them to \(\text{AFF}\), following the same backtracking verification as in **DMatch**, until a fixpoint is reached, i.e., no more matches can be removed.

**Example 7:** Continuing with Example 6, **QMatch** invokes **IncQMatch** to process \(\Pi(Q^{+e})(x_o, z_2), \Delta E = \{(x_o, z_2), (z_2, \text{Redmi 2A})\}\) (see Fig. 3) as follows.

1. **IncQMatch** first initializes the candidate sets as the cached matches in **DMatch** (shown below). For node \(x_o\) not in \(\Pi(Q)\), **IncQMatch** finds \(C(z_2)\) as initialized in **QMatch**.

<table>
<thead>
<tr>
<th>pattern</th>
<th>node</th>
<th>C(.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_o)</td>
<td>C((x_o)={((x_o), (z_2))})</td>
<td></td>
</tr>
<tr>
<td>(z_1)</td>
<td>C((z_1)={(v_1, v_2, v_3)})</td>
<td></td>
</tr>
<tr>
<td>(z_2)</td>
<td>C((z_2)={(v_4)})</td>
<td></td>
</tr>
<tr>
<td>Redmi 2A</td>
<td>C((\text{Redmi 2A}))={(\text{Redmi 2A})}</td>
<td></td>
</tr>
</tbody>
</table>

2. It starts with edge \((z_2, \text{Redmi 2A}), \text{Redmi 2A}\), and initializes AFF as \(C(z_2) \cup C(\text{Redmi 2A})=\{v_4, \text{Redmi 2A}\}\). It next checks whether \(v_4\) and \(\text{Redmi 2A}\) remain matches with counter satisfying the quantifiers. In this process, it only visits the two cached matches \(x_3\) and \(x_4\) following the pattern edges. As both nodes are matches, no change needs to be made.

3. **IncQMatch** next processes edge \((x_o, x_2)\). It adds the set \(C(x_3)=\{(x_3, x_2)\}\) to AFF, and checks whether \(x_2\) and \(x_3\) remain matches. As \(x_2\) has no edge to \(v_4\), \(X(x_2, x_2)\) is updated to false, and \(x_2\) is removed from \(C(x_2)\). It next finds that \(x_3\) is a valid match, by visiting \(v_3, v_3, \text{Redmi 2A}, v_4\). As no more matches can be removed, **IncQMatch** stops the verification. It returns \(\Pi(Q^{+e})(x_o, G)\) as \(\{x_3\}\). After the process, AFF contains \(\{v_3, x_2, x_3, x_2, v_3, \text{Redmi 2A}\}\). It incurs in total 3 rounds of verification for \(v_3, v_2\) and \(x_3\). □

Contrast **IncQMatch** with **DMatch**. (1) **IncQMatch** only visits the cached matches and their edges, rather than the entire \(G\). (2) **IncQMatch** incurs at most \(|\text{AFF}|\) rounds of verifications; hence it is optimal w.r.t. incremental complexity.

**Analysis of QMatch.** Algorithm **QMatch** correctly computes \(Q(x_o, G)\) following the definition of quantified matching (Section 2.2). For its complexity, observe the following.

1. If \(Q\) is positive, i.e., \(E_T^+ = \emptyset\), **IncQMatch** is not needed. Then **QMatch** and a conventional Match-based algorithm \(T\) for subgraph isomorphism have the same complexity. Quantifier checking is incorporated into the search process.

2. If \(E_T^-\) is nonempty, **IncQMatch** invokes at most \(|E_T^-|\) rounds of incremental computation by optimal **IncQMatch**, while \(|E_T^-|\leq |Q|\) and \(Q\) is typically small in practice. For each round, the overall time taken is bounded by \(|\text{AFF}| \ast K\), where \(K\) is the cost of a single verification.

Put together, **QMatch** takes \(O((t(T) + |E_T^-| \ast |\text{AFF}|) \ast K)\) time in total, where \(t(T)\) is the time complexity of a Match-based algorithm \(T\) for conventional subgraph isomorphism. We find in our experiments that **QMatch** and \(T\) have comparable performance, due to small \(|E_T^-|\) and \(|\text{AFF}|\). Moreover, existing optimization for \(T\) can be readily applied to **QMatch**.

Algorithm **QMatch** also makes use of graph simulation [21] to filter candidates and reduce verification cost. We defer this optimization strategy to Appendix-B.

5. **PARALLEL QUANTIFIED MATCHING**

Quantified matching – even within conventional subgraph isomorphism – may be cost-prohibitive over big graphs \(G\).
This suggests that we develop a parallel algorithm for quantified matching that guarantees to scale with big \( G \). We develop such an algorithm, which makes quantified matching feasible in real-life graphs, despite its DP complexity.

5.1 Parallel Scalability

To characterize the effectiveness of parallelism, we advocate a notion of parallel scalability following [16, 26]. Consider a problem \( I \) posed on a graph \( G \). We denote by \( t((|I|, |G|)) \) the running time of the best sequential algorithm for solving \( I \) on \( G \), i.e., one with the least worst-case complexity among all algorithms for \( I \). For a parallel algorithm, we denote by \( T((|I|, |G|, n)) \) the time it takes to solve \( I \) on \( G \) by using \( n \) processors, taking \( n \) as a parameter. Here we assume \( n \ll |G| \), i.e., the number of processors does not exceed the size of \( G \); this typically holds in practice as \( G \) often has trillions of nodes and edges, much larger than \( n \) [20].

**Parallel scalability.** An algorithm is parallel scalable if

\[
T((|I|, |G|, n)) = O\left(\frac{t((|I|, |G|))}{n} + (n|I|)^{O(1)}\right).
\]

That is, the parallel algorithm achieves a linear reduction in sequential running time, plus a “bookkeeping” cost \( O((n|I|)^{O(1)}) \) that is independent of \( |G| \), for a constant \( l \).

A parallel scalable algorithm guarantees that the more processors are used, the less time it takes to solve \( I \) on \( G \). Hence given a big graph \( G \), it is feasible to efficiently process \( I \) over \( G \) by adding processors when needed.

5.2 Parallel Scalable Algorithm

Parallel scalability is within reach for quantified matching under certain condition. We first present some notations. For a node \( v \) in graph \( G \) and an integer \( d \), the \( d \)-hop neighbor \( N_d(v) \) of \( v \) is defined as the subgraph of \( G \) induced by the nodes within \( d \) hops of \( v \). The radius of a QGP \( Q(x_o) \) is the longest shortest distance between \( x_o \) and any node in \( Q \).

The main result of the section is as follows.

**Theorem 7:** There exists an algorithm \( \text{PQMatch} \) that given QGP \( Q(x_o) \) and graph \( G \), computes \( Q(x_o, G) \). It is parallel scalable for graphs \( G \) with \( \sum_{v \in G} |N_d(v)| \leq C_d \frac{|G|}{n} \), taking \( O\left(t(Q,G) + n\right) \) time, where \( d \) is the radius of \( Q(x_o) \), \( C_d \) is a predefined constant, and \( t(Q,G) \) is the worst-case running time of sequential quantified matching algorithms.

The condition is practical: 99% of real-life patterns have radius at most 2 [18], and the average node degree is 14.3 in social graphs [12]; thus \( |N_d(v)| \) is often a small constant.

In addition, we will show that \( \text{PQMatch} \) can be adapted to evaluate QGPs \( Q \) with radius larger than \( d \).

As a proof, below we present \( \text{PQMatch} \). The algorithm works with a coordinator \( S_c \) and \( n \) workers (processors) \( S_i \). It utilizes two levels of parallelism. (a) At the inter-fragment parallelism level, it creates a partition scheme of \( G \) over multiple processors once for all, so that quantified matching is performed on all these fragments in parallel. The same partition is used for all QGPs \( Q(x_o) \) within radius \( d \). (b) At the intra-fragment level, local matching within each fragment is further conducted by multiple threads in parallel.

**Hop preserving partition.** We start with graph partition. To maximize parallelism, a partition scheme should guarantee that for any graph \( G \), (1) each of \( n \) processors manages a small fragment of approximately equal size, and (2) a query can be evaluated locally at each fragment without incurring inter-fragment communication. We propose such a scheme.

Given a graph \( G = (V,E,L) \), an integer \( d \) and a node set \( V' \subseteq V \), \( d \)-hop preserving partition \( \mathcal{P}_d(V') \) of \( V' \) distributes \( G \) to a set of \( n \) processors such that it is

1. balanced: each processor \( S_i \) manages a fragment \( F_i \), which contains the subgraph \( G_i \) of \( G \) induced by a set \( V_i \) of nodes, such that \( \bigcup_i V_i = V' (i \in [1, n]) \) and the size of \( F_i \) is bounded by \( c \frac{|G_i|}{n} \), for a small constant \( c \ll C_d \); and
2. covering: each node \( v \in V' \) is covered by \( \mathcal{P}_d(V') \), i.e., there exists a fragment \( F_i \) such that \( N_d(v) \in F_i \).

We say that \( \mathcal{P}_d(V') \) is complete if \( |V'| = |V| \).

One naturally wants to find an optimal partition such that the number \( |V'| \) of covered nodes is maximized. Although desirable, creating a balanced \( d \)-hop preserving partition is NP-hard. Indeed, conventional balanced graph partition is a special case when \( d=1 \), which is already NP-hard [6].

**Parallel \( d \)-hop preserving partition.** We provide an approximation algorithm for \( d \)-hop preserving partition with an approximation ratio. Better still, it is parallel scalable.

**Lemma 8:** If \( \sum_{v \in G} |N_d(v)| \leq C_d \frac{|G|}{n} \), for any constant \( \epsilon > 0 \), there is a parallel scalable algorithm with approximation ratio \( 1 + \epsilon \) to compute a \( d \)-hop preserving partition.

Below we present such an algorithm, denoted by \( \text{DPar} \). Given a graph \( G \) stored at the coordinator \( S_c \), it starts with a base partition of \( G \), where each fragment \( F_i \) has a balanced size bounded by \( c \frac{|G|}{n} \). This can be done by using an existing balanced graph partition strategy (e.g., [23]). \( \text{DPar} \) then extends each fragment \( F_i \) to a \( d \)-hop preserving counterpart. (1) It first finds the “border nodes” \( F_i \cap O \), which have \( d \)-hop neighbors not residing in \( F_i \), by traversing \( F_i \) in parallel.

(2) Each worker \( S_i \), then computes and loads \( N_d(v) \) for each \( v \in F_i \), by “traversing” \( G \) via disk-based parallel breadth-first search (BFS) search [24]. Moreover, \( \text{DPar} \) uses a balanced loading strategy (see below) to load approximately equal amount of data to each worker in the search. The process repeats until no fragments can be expanded.

**Balancing strategy.** \( \text{DPar} \) enforces a balanced fragment size \( c \frac{|G|}{n} \). It conducts a \( d \)-hop preserving partition \( \mathcal{P}_d(V') \) with approximation ratio \( 1+\epsilon \) subject to the bound, for any given \( \epsilon \). That is, if the size of nodes covered by the optimal \( d \)-hop partition in \( G \) is \( |V'| \), then \( \mathcal{P}_d(V') \) has \( |V'| \geq (1-\epsilon)V' \).

More specifically, at the BFS phase, for each \( v \in \bigcup F_i \), \( O \), \( \text{DPar} \) assigns \( N_d(v) \)’s to workers by reduction to Multiple Knapsack problem (MKP) [13]. Given a set of weighted items (with a value) and a set of knapsack with capacities, MKP is to assign each item to a knapsack subject to its capacity, such that the total value is maximized. \( \text{DPar} \) treats each \( N_d(v) \) as an item with value \( 1 \) and total \( |N_d(v)| \), and each fragment as a knapsack with capacity \( c \frac{|G|}{n} - |F_i| \), with the number of covered nodes as the total value. It solves the MKP instance by invoking the algorithm of [13], which computes an assignment with approximation ratio \( 1 + \epsilon \) for any given \( \epsilon \), in \( O(|V'|^2) \) time. Each worker \( S_i \) then loads its assigned \( N_d(v) \). This gives us a \( d \)-hop preserving partition \( \mathcal{P}_d \) with ratio \( 1 + \epsilon \) (see Appendix A for the reduction).

Partition \( \mathcal{P}_d \) may not be complete, i.e., not every node in \( V \) is covered. To maximize inter-fragment parallelism, \( \text{DPar} \) “completes” \( \mathcal{P}_d \) while preserving the balanced partition size.
Algorithm PQMatch

Input: QGP $Q(x_o)$, graph $G$, coordinator $S_c$, $n$ workers $S_1,\ldots,S_n$
Output: the answer set $Q(x_o, G)$.
1. $DPar(G)$; /*Preprocessing*/
2. $Q(x_o, G) := \emptyset$; post Q to each worker;
3. if every worker $S_i$ returns answer $Q(x_o, F_i)$ then
   4. $Q(x_o, G) := \bigcup Q(x_o, F_i)$;
5. $return Q(x_o, G)$; /*executed at each worker in parallel*/
6. $Q(x_o, F_i) := mqMatch(b, Q, F_i)$; /* b: the # of threads*/
7. $return Q(x_o, F_i)$;

Figure 6: Algorithm PQMatch

For each uncovered node $v$, it assigns $N_d(v)$ to a worker $S_i$ that minimizes estimated size difference $|F_{max}| - |F_{min}|$, where $F_{max}$ (resp. $F_{min}$) is the largest (resp. smallest) fragment if $N_d(v)$ is merged to $F_i$. Since $\sum |N_d(v)| \leq C_d \cdot \frac{|G|}{n}$, this suffices to make $P_d$ both complete and $d$-preserving.

Example 8: Consider graph $G_2$ of Fig. 2 and a set $V' = \{v_5, v_6, v_9\}$. Assume a base partition distributes $\{v_5\}$ to worker $S_1$, $\{v_5, v_9\}$ to $S_2$, $\{v_6, v_8\}$ to $S_3$, respectively. $DPar$ creates a 1-hop preserving partition $P_1$ for $V'$ as follows. (1) Each $S_i$ identifies its border nodes $F_iO$ by a local traversal, e.g., $v_5 \in F_iO$. (2) Each $S_i$ traverses $G_2$ at $S_i$ and finds $N_1(v)$ for $v \in F_iO$, in parallel. At the end, $S_i$ keeps track of the nodes (edges) “requested” by workers as follows.

\[
\begin{array}{c|c}
\text{node} & \text{requested by} \\
\hline
x_4 & S_1, S_3 \\
x_5, x_6, \text{prot.} & S_2, S_3 \\
\text{PhD.} & S_1, S_2, S_3 \\
\end{array}
\]

$DPar$ next determines which site to send the border nodes by solving an MKP instance, shown as follows.

| site | $N_1(v)$ | (estimated) $|F_i|$ |
|------|----------|-------------------|
| $S_1$ | $N_1(v_5)$, $N_1(v_9)$ | 14 ($|N_1(v_5)| = 6$, $|N_1(v_9)| = 8$) |
| $S_2$ | $N_1(v_6)$ | 8 |
| $S_3$ | $N_1(v_8)$ | 15 |

Here $N_1(v)$ includes three nodes $x_4, v_5, \text{PhD}$, and three edges $(x_4, v_5), (v_5, \text{PhD})$ and $(x_4, \text{PhD})$; similarly for the others. This induces a 1-hop preserving partition $P_1$ that covers $\{v_5, v_7, v_8, v_9\}$. To complete $P_1$, $DPar$ selects $S_2$ to load $N_1(v_6)$, where $|N_1(v_6)| = 15$. This minimizes the estimated size $|F_{max}| - |F_{min}| = 19 - 14 = 5$. Here $|F_{max}|$ is estimated as the sum of $|F_2| = 8$ and 11 additional nodes and edges in $N_1(v_6)$ that are not “requested” by $S_2$ (e.g., $(x_4, v_6), (v_6, \text{PhD})$). The completed $P_1$ covers $V'$ with fragment size 14, 19 and 15 for $S_1, S_2$ and $S_3$, respectively.

Parallel algorithm. Using $DPar$, we next develop algorithm PQMatch. As shown in Fig. 6, PQMatch takes as input a QGP $Q(x_o)$ of radius at most $d$, and a graph $G$ distributed across $n$ workers by $DPar$, where fragment $F_i$ of $G$ resides at worker $S_i$. It works as follows. (1) The coordinator $S_c$ posts $Q(x_o)$ to each worker $S_i$ (line 2). (2) Each worker $S_i$ then invokes a procedure $mqMatch$ to compute local matches $Q(x_o, F_i)$ (line 7), where $mqMatch$ implements QMatch using multi-threading (see below). Once verified, $Q(x_o, F_i)$ is sent to $S_c$ (line 6). (3) Once all the workers have sent their partial matches to $S_c$, the coordinator computes $Q(x_o, G)$ as the union of all $Q(x_o, F_i)$ (lines 3-4).

Procedure $mqMatch$. Procedure $mqMatch$ is a multi-threading implementation of PQMatch (Section 4), supporting inter-fragment level parallelism. For pattern edge $e = (u, u')$ with quantifier $\sigma(e) \supset \sigma$ and a candidate $v$ in $C(u)$, it spawns $p$ threads to simultaneously verify the top $p$ selected candidates, one for each. Each thread $i$ maintains local partial matches (in its local memory). When all the $p$ threads backtrack to $v$, the local partial matches are merged, and the local counter of $u$ is updated by aggregating the local storage of each thread $i$ (see Appendix B for more details).

From Lemma 8 and Lemma 9 below, Theorem 7 follows (see Appendix A for a proof). We remark that $G$ is partitioned once by using a $d$-hop preserving partition process. Then for all QGPs with radius within $d$, no re-partitioning is needed. That is, condition $\sum |N_d(v)| \leq C_d \cdot \frac{|G|}{n}$ is needed only for $d$-hop preserving partition to be parallel scalable.

Lemma 9: Given $G$ distributed over $n$ processors by a $d$-hop preserving partition $P_d$, (1) $Q(x_o, G) = \bigcup_{i \in [1, n]} Q(x_o, F_i)$, and (2) $mqMatch$ is parallel scalable for all QGPs $Q(x_o)$ with radius bounded by $d$.

Remark. Algorithm PQMatch can be easily adapted to dynamic query load and graphs. (1) For a query with radius $d' > d$, each worker $S_i$ incrementally computes $N_{d'}(v)$ for each node $v \in F_iO$, via the balanced parallel BFS traversal. (2) When $G$ is updated, coordinator $S_c$ assigns the changes (e.g., node/edge insertions and deletions) to each fragment. Each worker then applies incremental distance querying [15] to maintain $N_d(v)$ of all affected $v \in F_iO$ for $i \in [1, n]$.

6. QUANTIFIED ASSOCIATION RULES

As an application of QGPs, we introduce a set of graph association rules (QGARs) with counting quantifiers, to identify regularity between entities in graphs in general, and potential customers in social graphs in particular.

QGARs. A quantified graph association rule $R(x_o)$ is defined as $Q_1(x_o) \Rightarrow Q_2(x_o)$, where $Q_1$ and $Q_2$ are QGPs, referred to as the antecedent and consequent of $R$, respectively.

The rule states that for all nodes $v_x$ in a graph $G$, if $v_x \in Q_1(x_o, G)$, then the chances are that $v_x \in Q_2(x_o, G)$.

Using QGPs, QGAR $R$ can express positive and negative correlations [40] and social influence patterns with statistical significance [19], which are useful in targeted advertising. (1) If $Q_2$ is a positive QGP, $R(x_o)$ states that if $x_o$ satisfies the conditions in $Q_1$, then “event” $Q_2$ is likely to happen to $x_o$. For instance, $Q_2(x_o)$ may be a single edge $\text{buy}(x_o, y)$ indicating that $x_o$ may buy product $y$. In a social graph $G$, $R(x_o, G)$ identifies potential customers $x_o$ of $y$. (2) When $Q_2$ is, e.g., a single negated edge $\neg \text{buy}(x_o, y)$, $R(x_o)$ suggests that no $v_x \in Q_1(x_o, G)$ will likely buy product $y$.

Example 9: A positive QGAR $R_1(x_o)$: $Q_1(x_o) \Rightarrow \text{buy}(x_o, y)$ is shown in Fig. 7, where $Q_1$ is the QGP given in Example 1, and $Q_2$ is a single edge $\text{buy}(x_o, y)$ (depicted as a dashed edge). It states that if $x_o$ is in a music club and if 80% of people whom $x_o$ follows like an album $y$, then $x_o$ will likely buy $y$. 
A negative QGAR $R_2$ is also shown in Fig. 7, where $Q_2$ is a single negative edge $follow(x_o, y)$. The QGAR states that if $x_o$ and $y$ actively $(\geq k)$ tweet on competitive products (e.g., “Mac” vs “PC”), then $x_o$ is unlikely to follow $y$. Intuitively, $R_2$ demonstrates “negative” social influence [19].

As another example, $R_3$ of Fig. 7 is a rule in which $Q_2$ consists of multiple nodes. Here $Q_1$ in $R_3$ specifies users $x_o$ who actively promote mobile phone Redmi 2 and influence other users; and $Q_2$ predicts the impact of $x_o$ on other users for a new release Redmi 2A. Putting these together, $R_3$ states that if $x_o$ is influential over an earlier version, then $x_o$ is likely to promote the selling of a new release [4]. Intuitively, $Q_1$ identifies $x_o$ as “leaders” [19], who are often targeted by companies for promotion of a product series [4].

To the best of our knowledge, these QGARs are not expressible as association rules studied so far (e.g., [16, 17]).

QGARs also naturally express conventional association rules defined on itemsets. For instance, milk, diaper $\Rightarrow$ beer is depicted as QGAR $R_4(x_o)$ in Fig. 7. It finds customers $x_o$ who, if buy milk and diaper, are likely to purchase beer.

For real-world applications (e.g., social recommendation), we consider practical and nontrivial QGARs by requiring: (a) $Q_1$ and $Q_2$ are connected and nonempty (i.e., each of them has at least one edge); and (b) $Q_1$ and $Q_2$ do not overlap, i.e., they do not share a common edge. We treat $R$ as a QGP composed of both $Q_1$ and $Q_2$ such that in a graph $G$, $R(x_o, G) = Q_1(x_o, G) \cap Q_2(x_o, G)$.

**Interestingness measure.** To identify interesting QGARs, we define the support and confidence of QGARs.

**Support.** Given a QGAR $R(x_o)$ and a graph $G$, the support of $R$ in $G$, denoted as $\text{supp}(R, G)$, is the size $|R(x_0, G)|$, i.e., the number of matches in $Q_1(x_o, G) \cap Q_2(x_o, G)$. We justify the support with the result below, which shows its anti-monotonicity for both pattern topology and quantifiers.

**Lemma 10:** For any extension $R'$ of $R$ by (1) adding new edges to $Q_1$ or $Q_2$, or (2) increasing $p$ in positive quantifiers, $|\text{supp}(R', G)| \leq |\text{supp}(R, G)|$.

**Confidence.** We follow the local close world assumption (LCWA) [14], assuming that graph $G$ is locally complete, i.e., either $G$ includes the complete neighbors of a node for any known edge type, or it has no information about these neighbors. We define the confidence of $R(x_o)$ in $G$ as

$$\text{conf}(R, G) = \frac{|R(x_o, G)|}{|Q_1(x_o, G) \cap X_o|},$$

where $X_o$ is the set of candidates of $x_o$ that are associated with an edge of the same type for every edge $e=(x_o, u)$ in $Q_2$. Intuitively, $X_o$ retains those “true” negative examples under LCWA, i.e., those that have every required relationship of $x_o$ in $Q_2$ but are not a match (see Appendix C for justification).

**Quantified entity identification.** We want to use QGARs to identify entities of interests that match certain behavior patterns specified by QGPs. To this end, we define the set of entities identified by a QGAR $R(x_o)$ in a (social or knowledge) graph $G$ with confidence $\eta$ as follows:

$$R(x_o, \eta, G) = \{v_0 \mid v_0 \in R(x_o, G), \text{conf}(R, G, \eta) \geq \eta\},$$

i.e., entities identified by $R$ if its confidence is above $\eta$.

We study the **quantified entity identification (QEI)** problem: Given a QGAR $R(x_o)$, graph $G$, and a confidence threshold $\eta > 0$, it is to find all the entities in $R(x_o, \eta, G)$.

The QEI problem is DP-hard, as it embeds the quantified matching problem, which is DP-hard (Theorem 1). However, the (parallel) quantified matching algorithms for QGPs can be extended to QEI, without incurring substantial extra cost. Denote as $t(|Q|, |G|)$ the cost for quantified matching of QGP $Q$ in $G$. Then we have the following (Appendix A).

**Corollary 11:** There exist (1) an algorithm to compute $R(x_o, \eta, G)$ in $O(t(|R[G]|) \times |G|)$ time; and (2) a parallel scalable algorithm to compute $R(x_o, \eta, G)$ in $O(t(|R(G)|) + n)$ time with $n$ processors, under the condition of Theorem 7. □

**7. EXPERIMENTAL STUDY**

We conducted three sets of experiments to evaluate (1) the scalability and (2) parallel scalability of our quantified matching algorithms, and (3) the effectiveness of QGAR for identifying correlated entities in large real-world graphs.

**Experimental setting.** We used two real-life graphs: (a) Pokec [2], a social network with 1.63 million nodes of 269 different types, and 30.6 million edges of 11 types, such as follow, like; and (b) YAGO2, an extended knowledge base of YAGO [39] that consists of 1.99 million nodes of 13 different types, and 5.65 million links of 36 types.

We also developed a generator to produce synthetic social graphs $G=(V, E, L)$, controlled by the numbers of nodes $|V|$ (up to 50 million) and edges $|E|$ (up to 100 million), with $L$ drawn from an alphabet $L$ of 30 labels. The generator is based on GTgraph [7] following the small-world model.

**Pattern generator.** For real-life graphs we generated QGPs $Q$ controlled by $|V_Q|$ (size of pattern nodes), $|E_Q|$ (pattern edges), $p\%$ (in quantifiers) and $|E_Q|$ (size of negated edges). (1) We first mined frequent features, including edges and paths of length up to 3 on each of Pokec and YAGO2. We selected top 5 most frequent features as “seeds”, and combined them to form the stratified pattern $Q_s$ of $|V_Q|$ nodes and $|E_Q|$ edges. (2) For frequent pattern edges $e=(u, u')$, we assigned a positive quantifier $\sigma_\epsilon \geq p\%$, where $p\%$ is initialized as 30% unless otherwise specified. This completes the generation of $Q_s$. (3) We added $|E_Q|$ negated edges to $Q_s$, between randomly selected node pairs $(u, u')$, to complete the construction of $Q$. For synthetic graphs, we generated 50 QGPs with labels drawn from $L$.

We denote by $|Q|=|\{V_Q, |E_Q|, p\%\}|$ the size of QGP $Q$, where $p\%$ is the average of $p$ in all its quantifiers.

**Algorithms.** We implemented the following, all in Java.

1. Algorithm $QMatch$, versus (a) $QMatch_a$, a revision of $QMatch$ that processes negated edges using DMatch, not the incremental IncQMatch, and (b) $Enum$, which adopts a state-of-the-art subgraph isomorphism algorithm [35] to enumerate all matches first, and then verify quantifiers. The algorithm in [35] is verified to outperform conventional counterparts, e.g., VF2, by 3 orders of magnitude.

2. Algorithm $PQMatch$, versus (a) $PQMatch_a$, its single-thread counterpart, (b) $PQMatch_{ab}$, the parallel version of $QMatch_a$, and (c) $Penum$, a parallel version of $Enum$, which first invokes a parallel subgraph listing algorithm [37] to enumerate all matches, and then verifies quantifiers. We also implemented (d) DPar for $d$-hop preserving partition.

We deployed the parallel algorithms over $n$ processors for $n \in [4, 20]$. Each processor has 2.6GHz 4vCPU with 16G.
Experimental results. We next report our findings.

Exp-1: Performance of QMatch. We first evaluated the performance of QMatch versus QMatch$_h$ and Enum. Fixing $|Q|=(5, 7, 30\%, 1)$, i.e., patterns with 5 nodes and 7 edges, $p_n=30\%$ and one negative edge, Figure 8(a) reports the performance of QMatch over two real-world graphs Pokec (denote the result as Pokec$_5$) and YAGO2, and a larger synthetic graph $G_s$, of 50 million nodes and 100 million edges. We find the following. (1) QMatch outperforms the other algorithms. It is on average 1.2 and 2.0 times faster than QMatch$_h$ and Enum over YAGO2, 1.3 and 2.0 times faster over Pokec, and 1.3 and 2.6 times faster over $G_s$, respectively. This verifies that our optimization strategies effectively reduce the verification cost. (2) QMatch works reasonably well over real-world social and knowledge graphs. It takes up to 150 (resp. 116) seconds over Pokec (resp. YAGO2), comparable to conventional subgraph isomorphism without quantifiers. Moreover, it takes longer time for larger $|Q|$ (e.g., with size (6, 8, 30\%, 1), the result is denoted as Pokec$_6$), as expected.

Exp-2: Scalability of PQMatch. This set of experiments evaluated the scalability of parallel algorithm PQMatch, compared to PQMatch$_h$, PQMatch$_a$, and Enum. In these experiments, we fixed $|Q|=(6, 8, 30\%, 1)$, $d=2$ for $d$-hop preserving partition and $b=4$ for the number of threads in intra-fragment parallelism, unless stated otherwise.

Varying $n$ (PQMatch). We varied the number $n$ of processors from 4 to 20. As shown in Fig. 8(b) (resp. Fig. 8(c)) over Pokec (resp. YAGO2), (1) PQMatch and PQMatch$_a$ scale well with the increase of processors: for PQMatch, the improvement is 2.8 (resp. 3.2) times when $n$ increases from 4 to 20; this verifies Theorem 7; (2) PQMatch is 3.8 (resp. 5.8) times faster than Enum; and (3) with optimization strategies (incremental evaluation and multi-threads), PQMatch outperforms PQMatch$_a$ and PQMatch$_h$ by 1.5 (resp. 1.1) times and 2.8 (resp. 2.3) times, respectively. (4) PQMatch works reasonably well on large graphs. With 20 processors, it takes 40.3 (resp. 10.2) seconds on Pokec (resp. YAGO2).

Varying $n$ (DPar). We also evaluated the scalability of DPar for $d$-hop preserving partition, with $d=2$ and $d=3$. Here DPar incrementally computed the partition when $d$ is changed from 2 to 3 (see Section 5.1). As shown in Figures 8(d) and 8(e), (1) DPar scales well with $n$: when $d=2$, the improvement is 3.5 (resp. 2.5) times when $n$ increases from 4 to 20 over Pokec (resp. YAGO2). (2) The fragments are well balanced: the “skew” (the ratio of the size of the smallest fragment to the largest one) is at least 80\% when $n=8$, for both Pokec and YAGO2. (3) DPar effectively updates the partition when $d$ is increased.

These justify the parallel scalability of DPar and PQMatch.

Varying $|Q|$. Fixing $p_n=30\%$, $|E_Q^-|=1$ and $n=8$, we varied $(|V_Q|, |E_Q|)$ from (4, 6) to (8, 10) (resp. (3, 5) to (7, 9)) on Pokec (resp. YAGO2). As shown in Figures 8(f) and 8(g), (1) the larger $|Q|$ is, the longer time is taken by all the algorithms, as expected. (2) PQMatch works well on real-life queries. For queries of size (5, 7) (close to real-world queries), it takes up to 35 (resp. 16.3) seconds over Pokec (resp. YAGO2). It works better on more sparse YAGO2. (3) PQMatch outperforms the other algorithms, which is consistent with the results shown in Figures 8(b) and 8(c).
Varying $|E_Q|$ - We also studied the impact of the number of negated edges. The purpose of this test is to evaluate the effectiveness of incremental matching strategy IncQMatch.

Fixing $n = 8$, $|V_Q|$, $|E_Q| = (6, 8)$ and $p_a = 30\%$, we varied $|E_Q|$ from 0 to 4 by selecting $|E_Q|$ edges $e$ and “negating” them by setting $\sigma(e) = 0$. As shown in Figures 8(h) and 8(i), (1) PQMatch and PQMatch$_n$ are rather indifferent to the change of $|E_Q|$, which incurs small extra cost due to their incremental strategy (IncQMatch). (2) In contrast, PQMatch$_n$ and PEnum are more sensitive to the increment of $|E_Q|$. Both algorithms, without IncQMatch, always recompute the matches of pattern $Q^{+e}$ for each negated edge $e \in E_Q$, and hence take more time over larger $|E_Q|$. The improvement of PQMatch over PQMatch$_n$ and PEnum becomes more significant (from 1.1 to 2 times, and 3.1 to 5 times) with larger $|E_Q|$ (from 1 to 4) over Pokec. These results verify the effectiveness of IncQMatch.

Varying $p_a$. Fixing $n = 8$, $|V_Q| = 1$ and $(|V_Q|, |E_Q|) = (6, 8)$ (resp. $(5, 4)$) for Pokec (resp. YAGO2), we evaluated the impact of aggregates by varying $p_a$ from 10% to 90%. As shown in Figures 8(j) and 8(k), (1) with larger $p_a$, PQMatch, PQMatch$_n$ and PEnum take less time, since more candidates are pruned in the verification process. (2) In contrast, PEnum is indifferent to the change of $p_a$, since it always enumerates all the matches regardless of $p_a$. This verifies the effectiveness of the pruning strategies of PQMatch.

Observe that PQMatch is less sensitive than PQMatch$_n$ to $p_a$. When $p_a$ is small, the overhead of PQMatch$_n$ incurred by the recomputation of $Q^{+e}$ for negated edges $e$ is larger, since a large number of candidates need to be verified. With larger $p_a$ (more strict quantifiers), the overhead reduces due to the effective pruning of candidates by PQMatch$_n$. This explains the comparable performance of PQMatch and PQMatch$_n$ when $p_a$ is large (e.g., $p_a=0.9$).

Varying $|G|$. Fixing $n = 4$, we varied $|G|$ from (10M, 20M) to (50M, 100M) using synthetic social graphs. As shown in Fig. 8(l), (1) PQMatch scales well with $|G|$ and is feasible on large graphs. It takes 125 seconds when $|G| = (50M, 100M)$. (2) PQMatch is 1.5, 2.3 and 4.7 times faster than PQMatch$_n$, PQMatch$_n$ and PEnum on average, respectively.

Exp-3: Effectiveness of QGAR. We also evaluated the effectiveness of QGARs. We developed a simple QGAR mining algorithm by extending the algorithm of [16] for mining graph pattern association rule (GPARs). GPARs are a special case of QGARs $Q_2(x_o) \Rightarrow Q_2(x_o)$ that have no quantifiers and restrict $Q_2$ to a single edge. (1) We mined a set of top GPARs using [16] over Pokec and YAGO2, for confidence threshold $\eta = 0.5$. For each GPAR $R$, we initialized a QGAR $R^*$. (2) We extended $Q_2$ in each $R^*$ by adding frequent edges whenever possible, and by gradually enlarging $p_a$ for frequent edges by increment 10% (1 for numeric aggregates). We stopped when the confidence of $R^*$ got below $\eta$. We show three QGARs in Fig. 9, illustrated as follows.

(1) $R_5$ (Pokec) says that if a user has “long-distance” friends, i.e., at least two of her friends do not live in the same city “Presov” where she lives, then the chances are that she share the hobby of traveling. We found 50 matches in Pokec.

(2) $R_6$ (Pokec; confidence 0.8) demonstrates a negative pattern: for a user $x_o$, if more than half of his friends share the same hobby “PC Games”, and none of them like sports, then it is likely $x_o$ does not like sports. $R_6$ has support 4000.

(3) $R_7$ (YAGO2; confidence 0.75) states that if a US professor (a) won at least two academic prizes, and (b) graduated at least 4 students, then the chances are that at least one of her/his students is not a US citizen. It discovers scientists such as Marvin Minsky (Turing Award 1969) and Murray Gell-Mann (Nobel Prize Physics 1969) from YAGO2. Here $Q_2$ in $R_7$ has three (dashed) edges, as opposed to GPARs [16].

These QGARs demonstrate quantified correlation between the entities in social and knowledge graphs, which cannot be captured by conventional association rules and GPARs [16].

Summary. We find the following. Over real-life graphs, (1) quantified matching is feasible: PQMatch (with 20 processors) and QMatch took 40.3s and 342s on Pokec, and 10.2s and 116s on YAGO2, respectively. (2) Better still, PQMatch and DPar are parallel scalable: their performance for smaller graphs is improved by 3 times on average with workers increased from 4 to 20. (3) Our optimization techniques improve the performance of QMatch and PQMatch by 1.27 and 1.3 times on average, and 2.2 and 4.5 times over Enum and PEnum, respectively. (4) QGARs capture behavior patterns that cannot be expressed with conventional graph patterns.

8. CONCLUSION

We have proposed quantified matching, by extending traditional graph patterns with counting quantifiers. We have also studied important issues in connection with quantified matching, from complexity to algorithms to applications. The novelty of this work consists in quantified patterns (QGPs), quantified graph association rules (QGARs), and algorithms with provable guarantees (e.g., optimal incremental matching and parallel scalable matching). Our experimental study has verified the effectiveness of QGPs and the feasibility of quantified matching in real-life graphs.

We will study practical extensions of QGPs to more general graph patterns (e.g., bounded simulation with regular path constraints) and other built-in predicates ($\leq$, $\neq$). Another topic concerns QGAR discovery. It calls for a nontrivial extension of prior pattern mining algorithms (e.g., [16, 17]) to accurately identify quantifiers.


9. REFERENCES


Appendix A: Proofs

Proof of Theorem 1

It is easy to see that Theorem 1 follows from Lemmas 2, 3 and 4. Below we verify these lemmas.

Lemma 2: Lower bounds. We first prove the lower bounds of quantified matching, for positive and negative QGPs.

(1) NP-hardness. The quantified matching for positive QGPs with numeric aggregates is already NP-hard. This can be verified by constructing a polynomial time reduction from subgraph isomorphism (subISO), where a QGP Q is constructed by adding quantifiers $\sigma(e) \geq 1$ for all the edges $e$ in a graph pattern $Q'$ in an instance of subISO. As subISO is NP-hard (cf. [33]), so is quantified matching.

(2) DP-hardness. We next prove that quantified matching for (possibly) negative QGPs is DP-hard by reduction from EXACT-CLIQUE, which is DP-complete (cf. [33]). Given a graph $G$ and a natural number $k$, EXACT-CLIQUE is to determine whether the largest clique of $G$ has size exactly $k$. Given an instance of EXACT-CLIQUE, We construct a QGP $Q(x_o)$ that consists of a $k$-clique $Q_1$, a $k + 1$-clique $Q_2$ such that $x_o$ has a unique match $v_o$ in $G'$, there is an edge $e$ from $x_o$ to each node in $Q_1$ with $\sigma(e) \geq 1$, and to a single node $v'$ in $Q_2$ via a negated edge. The transformation is obviously in PTIME. Moreover, the largest clique of $G$ has size $k$ if and only if $v_o \in Q(x_o, G')$. Therefore, the transformation above is a reduction. As EXACT-CLIQUE is DP-complete, quantified matching with (possibly) negative QGPs is DP-hard.

Lemma 3: Upper bounds. We next prove the upper bounds for quantified matching. We first consider positive QGPs $Q$ with numeric aggregates $\sigma(e) \cap p$ only (Lemma 3). We then extend the result to ratio aggregates $p\%$ (Lemma 4).

(1) Given a QGP $Q(x_o)$, we construct a traditional graph pattern $Q(x_o)$ without quantifiers, by (a) stripping off all quantifiers from $Q$, and (b) for each edge $e(u, u')$ associated with $\sigma(e) \cap p$, if $p > 1$, we make $p$ copies of $u'$ in $Q_o$ as children of $u$, along with copies of edges from $u'$ and so on. Then one can easily verify the following: (a) for any
graph $G$, $Q(x_o, G) = Q_i(x_o, G)$, and (b) the time for constructing $Q_d$ and hence $|Q_i|$ are both a polynomial in $|Q|$; this is because on each simple path in $Q$, there are at most $k$ non-existent quantifiers, for a predefined constant $k$. These make a PTIME reduction from quantified matching with positive numeric aggregates to conventional subgraph isomorphism. Since the latter is in NP, so is the former.

(2) We next prove that the quantified matching for (possibly) negative QGPs is in DP. Following [33], it suffices to construct two languages $L_1$ and $L_2$, such that a node $v_x$ is in $Q(x_o, G)$ if and only if $v_x$ is in $L_1 \cap L_2$.

We consider two languages below:

- $L_1$, the set $\Pi(Q)(x_o, G)$, and
- $L_2$, the set of “yes” instances for a node $v_x$ that is not a match of $x_o$ for $\bigcup_{e \in E^-_Q}(\Pi(Q^+)(x_o, G))$. 

One can verify that (1) $L_1 \in NP$, (2) $L_2 \in coNP$, and (3) a node $v_x$ is in $Q(x_o, G)$ if and only if $v_x$ is in $L_1 \cap L_2$, by the definition of QGPs. Thus quantified matching is in DP.

**Lemma 4: Ratio aggregates.** Given a QGP $Q$ that contains ratio aggregates $\sigma(e) \circ p\%$, we construct a QGP $Q_d$ and graph $G_d$ in PTIME such that $Q_d$ consists of numeric aggregates only, and $Q_d(x_o, G_d) = Q(x_o, G)$. To simplify the discussion, we consider $w.l.o.g.$ positive $Q$. For negated edges $e$, by the definition of $Q(x_o, G)$, $e$ is posited in $Q^+$. Hence, it suffices to consider positive edges.

(a) We transform $G$ to a graph $G_d$ as follows. For each node $v$ with $g$ child in $G$, we add $(1 - p\%)(d - g)$ dummy children with a label that does not match any pattern node in $Q$, and $p\%(d - g)$ dummy children that complete a dummy subgraph $G_d$ at $v$ that is isomorphic to $Q_e$ of $Q$.

(b) We transform $Q$ to $Q_d$ such that for each edge $e$ with quantifier $\sigma(e) \circ p\%$, we replace $p\%$ with a constant $p\% \ast d$.

One may verify that a node $v_x$ is in $Q(x_o, G)$ if and only if its $G_d$-counterpart $v_d \in Q_d(x_d, G_d)$. Moreover, the transformation is obviously in PTIME. Since quantified matching for all numeric quantified $Q_d$ is in NP by Lemma 3, so its counterpart for QGPs $Q$ with ratio quantifiers.

This completes the proof of Theorem 1.

**Proof of Lemma 5**

To show the correctness of DMatch, first observe that DMatch always terminates. Indeed, DMatch follows the verification process of conventional subgraph isomorphism algorithm. The process, in the worst case, enumerates all possible isomorphism mappings from the stratified pattern $Q_s$ to $G$, which are finitely many. Hence DMatch terminates.

We next show that DMatch correctly verifies whether a candidate $v_x$ is a match of $x_o$ in $\Pi(Q)$ via an isomorphism $h_0 \in \Pi(Q)(G)$. It suffices to show that (1) $h_0$ is a match in $Q_s(G)$, and (2) for each $u$ in $\Pi(Q)$ and each edge $e = (u, u')$, $|M_e(h_0(x_o), h_0(u), Q)| \circ p$ for $f(e) = \sigma(e) \circ p$.

(1) When DMatch terminates, for each $u \in \Pi(Q)$ and every candidate $v$ in $C(u)$ with $X(u, v) = true$, $v = h(u)$ for some $h \in Q_s(G)$, guaranteed by the correctness of Match.

(2) For each edge $(u, u')$ in $\Pi(Q)$ and a node $v$ with $X(u, v) = true$, DMatch correctly verifies the quantifiers by checking the updated local counter of $v$ that keeps track of the current $|M_e(h_0(x_o), h_0(u), Q)|$. Add in DMatch waits until either $v$ is determined not a valid match due to that the upper bound fails the quantifier (by the local pruning rule, Appendix B), or the lower bound satisfies the quantifier (in the verification). Hence, $v_x$ is a match if and only if $v_x \in \Pi(Q)(x_o, G)$ when DMatch terminates.

For the space complexity, it takes $O(|V|)$ space to store the auxiliary structures for the nodes in $G$. During the search, DMatch keeps, at each level of the search, at most $p_{m}$ best matches to be verified, where $p_{m}$ is the largest constant in quantifiers. Since there are in total $|\Pi(Q)| \leq |Q|$ levels of search, it takes in total $O(p_{m}|Q| + |V|)$ space.

**Proof of Proposition 6**

We prove Proposition 6 by giving the correctness and complexity analysis of IncQMatch below.

**Correctness.** Given a QGP $\Pi(Q^+)$, algorithm IncQMatch correctly computes $\Pi(Q^+)(x_o, G)$ by processing $\Delta E$ one edge at a time. At any time, the newly formed pattern $Q'$ contains $\Pi(Q)$ as a subgraph. It is easy to verify that $\Pi(Q)(u, G) \subseteq \Pi(Q^+)(u, G), \ for \ all \ u \in \Pi(Q)$. Hence, IncQMatch only needs to determine the nodes to be removed from the cached matches/candidates from QMatch.

We next show that IncQMatch removes a node $v$ from $C(u)$ if and only if it is not a match in $\Pi(Q^+)(u, G)$, for any node $u$ in $\Pi(Q^+)$. (1) If $v$ is not a match, then either $v$ is not in an isomorphism mapping, or $v$ fails the quantifier of at least one edge $(u, u')$. IncQMatch captures both cases by the isomorphism checking and quantifier verification. Hence it guarantees to remove all the $v$ that are not match. (2) Assume by contradiction that IncQMatch removes a node $v$ that is a match. Then either $v$ is not a match via isomorphism, or $v$ fails the counter for an edge $(u, u')$. Both contradict the assumption that $v$ is a match. Hence, IncQMatch only removes the nodes that are not matches in $\Pi(Q^+)(u, G)$.

**Complexity.** During the process, IncQMatch visits and verifies the following sets of nodes: (1) $C(u)$, including the cached matches and candidates of $u$ if $u$ is not in $\Pi(Q)$, where $u$ is in edge $e = (u, u')$ or $e = (u', u)$, for each $e \in \Delta E$; and (2) those nodes reachable from (or can be reached by) those nodes in (1) via a sequence of cached matches/candidates including those nodes in (1). The number of verification hence is bounded by the size of the set combining (1) and (2), which is in total at most $|A|F$.

**Proof of Theorem 7**

We prove Theorem 7 by providing the correctness and complexity analysis below for algorithm PQMatch.

**Correctness.** Given graph $G$ distributed over $n$ processors by a $d$-hop preserving partition $P_d$, PQMatch computes $Q(x_o, G) = \bigcup_{i=1}^{n} Q(x_o, F_i)$ for any QGP $Q(x_o, G)$ with radius bounded by $d$. It suffices to show Lemma 9(1).

**Lemma 9(1).** Observe the following. (1) For any match $v_x \in Q(x_o, F_i)$, QMatch only needs to visit $N_d(v_x)$ to verify whether $v_x$ is a match. (2) For every candidate $v_x \in C(x_o)$, there exists a fragment $F_i$, such that $N_d(v_x) \subseteq F_i$ (including $v_x$) by $d$-hop preservation. Hence, any match of $x_o$ must be from at least one match set $Q(x_o, F_i)$ evaluated at fragment $F_i$. (3) For every match $v_x \in Q(x_o, F_i)$ locally computed at $F_i$, $v_x$ is a match of $x_o$ guaranteed by the correctness of QMatch. Hence PQMatch correctly computes $Q(x_o, G)$ as $\bigcup_{i=1}^{n} Q(x_o, F_i)$ over a $d$-hop preserving partition.
Complexity. Algorithm PMatch consists of three steps: (1) the distribution of Q and construction of d-hop preserving partition, (2) the parallel evaluation, and (3) assembling of partial matches. The time for step (2) and (3) are in \(O(\frac{|G|}{n\epsilon})\) and \(O(n)\), respectively, i.e., are parallel scalable.

Hence it suffices to focus on the parallel scalability of step (1), by proving Lemma 8 and Lemma 9(2).

**Lemma 8** (Parallel scalability of DPar). We first show that procedure DPar is parallel scalable. We will show the approximation ratio of DPar separately in the next proof.

Observe the following. (1) For each worker \(w\), managing a fragment \(F_i\), in the base partition, the border nodes \(E_i, O\) can be computed via a linear scan of \(F_i\). Hence the overall time is in \(O(|C_d| + \frac{|G|}{\epsilon})\) by the condition of Lemma 8. (2) Given \(V' = \bigcup F_i, O\), DPar applies the \((1+\epsilon)\) approximation algorithm of [15], which computes an assignment in time \(O(|V'|^2)\). For \(\epsilon\) small enough for a good approximation, e.g., \(\epsilon = 1\), the time cost is \(O(|V'|)\). Since \(|V'| = \sum F_i, O \leq \sum N_d(v)\) \((v \in V')\), and \(\sum N_d(v)\) is bounded by \(O(C_d + \frac{|G|}{\epsilon})\), the process takes time in \(O(C_d + \frac{|G|}{\epsilon} + \frac{|G|}{\epsilon})\). (3) For each border node \(v \in F_i, O\), each \(w\) fetches \(N_d(v)\) from \(G\) in parallel. In the worst case, each worker takes in total \(O(\sum N_d(v))\) time for all the border nodes \(v \in F_i, O\). As the fetch process is bounded by \(O(C_d + \frac{|G|}{\epsilon})\) at each fragment, the overall parallel partition time is bounded by \(O(C_d + \frac{|G|}{\epsilon})=O(C_d + \frac{|G|}{\epsilon})\). Hence, DPar is parallel scalable.

**Lemma 9(2)** (Parallel scalability of mQMatch). Procedure mQMatch is conducted locally in parallel at each worker. From the correctness of Lemma 9(1), each worker only performs local matching without the need to communicate with others once DPar terminates. Hence, the overall time complexity is \(O(\frac{|G|}{\epsilon})\). The time cost for merging the answers is in \(O(n)\) time. Putting these together, PMatch is in \(O(\frac{|G|}{\epsilon} + O(n))\) time. Lemma 9(2) thus follows, and so does the parallel scalability of PMatch.

**Lemma 8** (Approximation). The d-hop preserving partition problem at the coordinator \(S_i\) is to find an assignment for each \(N_d(v)\) to a worker \(w_i\), such that (a) \(\sum |N_d(v)| \leq \epsilon \frac{|G|}{n}\) for all \(N_d(v)\) assigned to \(S_i\), and (b) \(|V_c|\) is maximized, where \(V_c\) refers the nodes covered in \(V'\). We show that the problem is \(1+\epsilon\)-approximable, by constructing an approximation preserving reduction (APR) [13] to the multiple knapsack problem (MKP) as follows.

- **function f**: for each \(v \in V'\), construct an item \(u_i\) with value 1 and weight \(|N_d(v)|\); and (b) for each worker \(S_i\), construct a bin \(B_i\) with capacity \(\epsilon \frac{|G|}{n} - |F_i|\).

- **function g**: for each item \(u_i\) packed to a bin \(B_i\) in \(s(I_i), g\) maps \(u_i\) to \(v_i\), and \(B_i\) to \(S_i\).

We next show that the transformation above is an APR. Indeed, (1) \(f\) is in \(\text{PTIME}\), and (2) for a feasible solution \(s(I)\), \(g(s(I))\) is also a feasible solution, since the packing does not exceed the capacity constraints in \(s(I)\) if and only if the assignment \(g(s(I))\) does not exceed the capacity of each fragment. (3) Assume \(s(I) \geq s^*(I)\) (\(\alpha = 1\)). One can verify that \(|s^*(I)| \leq |V_c|\) (the size of covered nodes) \(= |g(s^*(I))|\) (the size of packed items), \(|s(I)| = |g(s(I))|\). Hence \(g(s(I)) = s(I) \geq \frac{s^*(I)}{1+\epsilon} = s^*(I)\). Thus, the transformation is an APR. As a result, from [13] it follows that \(d\)-hop preserving partition is \(1+\epsilon\) approximable.

**Remarks.** We use a balanced bound for all fragments. This guarantees the correctness of the reduction to MKP. The choice of MKP is to get a balanced fragment bound and at the same time, to minimize synchronization cost. Our experimental study shows that this leads to quite balanced fragments (Exp-2), with minimum communication cost.

**Proof of Lemma 10**

Assume that \(R'(x_o): Q_1'(x_o) \Rightarrow Q_2'(x_o)\) is extended from \(R(x_o): Q_1(x_o) \Rightarrow Q_2(x_o)\). It suffices to prove that \(R'(x_o, G) \subseteq R(x_o, G)\) for any \(G\). Consider two cases below.

(1) If \(Q_1'(x_o)\) is extended by adding new edge that either positive or negative to \(Q_1(x_o, G)\) (see Section 4.2). Hence, if \(R'(x_o, G) = Q_1'(x_o, G)\cap\neg Q_2(x_o, G) \subseteq Q_1(x_o, G)\cap\neg Q_2(x_o, G) = R(x_o, G)\).

(2) Assume that \(Q_1'(x_o)\) is obtained by increasing \(p\) in a positive quantifier of an edge \(e = (u, v')\) in \(Q_1\). Assume that there exists a node \(v \in R'(x_o, G)\) that is not a match in \(R(x_o, G)\).

Then there must exist a positive edge \((u, v)\) with quantifier \(\sigma(e) \land p\) for which \(v\) does not have enough matches in its children. Nevertheless, for any positive quantifier \(\sigma(e) \land p\) in \(R\) and its counterpart \(\sigma(e) \land p'\) in \(R'\), \(p' \geq p\). As \(v\) is a match of \(R'(x_o, G)\), this contradicts the assumption that \(v\) is in \(R(x_o, G)\). Hence \(R'(x_o, G) \subseteq R(x_o, G)\).

The same argument applies to the case when \(Q_2'(x_o)\) is revised from \(Q_2\). Hence Lemma 10 follows.

**Proof of Corollary 11**

As a constructive proof, we outline two algorithms for computing \(R(x_o, \eta, G)\) with the desired complexity as follows.

**Sequential quantified entity matching.** Given a QAR \(R\), confidence threshold \(\eta\) and \(G\), the first algorithm, denoted as garMatch, (1) invokes PMatch to compute \(Q_1(x_o, G)\) and \(Q_2(x_o, G)\), respectively; (2) computes \(R(x_o, G) = Q_1(x_o, G) \cap Q_2(x_o, G)\); and (3) verifies whether \(\text{conf}(R) = \frac{|\{R(x_o, G)\}|}{|\{(x_o, G)\}|} \geq \eta\). If so, it returns \(R(x_o, G)\).

The correctness and complexity of garMatch follow from their QMatch counterparts (Lemmas 5 and 6).

**Parallel quantified entity matching.** We introduce a parallel algorithm, denoted as dgarMatch, for parallel quantified entity matching. It follows the generic steps of PMatch. The only difference is as follows: (a) each worker evaluates two patterns \(Q_1\) and \(Q_2\) in parallel, and (b) the coordinator \(S_c\) assembles the results to evaluate the confidence of \(R\).

Algorithm dgarMatch starts with a set of base partitions. (1) It constructs a d-hop preserving partition, where \(d\) is a predefined upper bound of the largest radius \(Q_1\) and \(Q_2\) in \(R\). (2) Each worker then computes local match \(Q_1(x_o, F_1)\) and \(Q_2(x_o, F_1)\) in parallel. It also computes the local set \(X_{oi}\). (3) Each worker returns the local matches to the coordinator \(S_i\). Then dgarMatch computes \(R(x_o, G)\) as \(\bigcup\{Q_1(x_o, F)\} \cap\bigcup\{Q_2(x_o, F)\}\), and computes the confidence \(\text{conf}(R, G)\) as \(\frac{|\{R(x_o, G)\}|}{|\{(x_o, G)\}|}\). It next verifies whether \(\text{conf}(R, G) \geq \eta\) and if so, returns \(R(x_o, G)\). Otherwise, it returns \(\emptyset\).

The correctness and complexity of dgarMatch follow from their PMatch counterparts.
Appendix B: Optimization Techniques

Selection and Pruning Rules (Section 4)

We show that the number of verification in algorithm QMatch can be further reduced, by applying the following selection and pruning rules.

1) [pruning rules]. The pruning rules are applied each time when DMATCH backtracks from a candidate \( v' \in S_P(u') \) to a match \( v \in S_P(u) \) for an edge \( e=(u, u') \).

(a) local pruning rule: If the dynamically maintained upper bound \( U(v, e) \) of each \( v \in C(x_0) \) fails the quantifier of \( e \) (because of a verified non-match \( v' \) as its child), then removes \( v \) from \( C(u) \) and \( S_P(u) \).

(b) global pruning rule: monitors the size of current candidates \( C(u') \), excluding those nodes verified to be non-matches (i.e., \( X(u', u) = \text{false} \)).

The lemma below justifies these pruning rules.

**Lemma 12:** Node \( x_0 \) in \( \Pi(Q) \) has a match only if for every node \( u' \) in \( \Pi(Q) \), \( |C(u')| \geq p_m \), where \( p_m = \max p' \) for \( p' \) ranging over \( p \) in \( \sigma(u, u') \) for all \( u'/u \)’s parent \( u \).

If \( C(u') < p_m \) for some pattern node \( u' \), QMatch terminates and returns the current \( \Pi(Q)(x_0, G) \).

2) [selection rule]. The selection rule is applied each time when DMATCH extends the search to a next level, at a match \( v \in S_P(u) \) for edge \( e=(u, u') \). Denote as \( P(u') \) the parent set of \( u' \) in \( G \), the potential of a match \( v' \in C(u') \) is

\[
1 + \frac{|P(v') \cap C(u)|}{|C(u)|} \cdot \sum_{x \in (u', u')} \frac{U(x, e)}{p_e}
\]

The rule picks candidates with highest potential scores.

**Example 10:** Consider \( Q_2 \) with \( p=3 \) (Fig. 2) and a revised \( G_2 \) (Fig. 2) by changing “UK” to “US”. QMatch terminates early without verifying all the nodes, in contrast to Match. As the potential of the candidates \( x_4, x_5 \) and \( x_6 \) for \( x_2 \) are all 1, QMatch starts with \( e=(u, u') \), (1) following edge \( e=(x_2, u') \), the potentials of candidates \( v_8, v_9, v_{10} \) and \( v_{11} = 2 * (1 + \frac{3}{4}) = 4, 4, 4, \frac{7}{4} \), and \( \frac{7}{4} \), respectively. Hence QMatch selects \( v_6, v_7 \) and \( v_5 \) as top 3 candidates, starting with \( v_6 \). (2) When backtracking to \( v_6 \), QMatch finds \( v_8 \) not a match; so it reduces the upper bound of \( x_4, x_6 \) to a match \( v' \in C(u') \). QMatch next backtracks to \( x_5 \). By the local pruning rule, it finds \( U(x_5, e) \) smaller than 3; thus \( v_8 \) is not a match. (3) Applying the global pruning rule, the size \( C(v) \{v_7, v_6\} \) is already smaller than 3. Hence QMatch returns \( \emptyset \) without further checking.

**Proof of Lemma 12**

**Proof:** We show Lemma 12 by contradiction. Assume that a node \( v \in C(x_0) \) is a match of \( x_0 \), and there exists a node \( u' \) in \( \Pi(Q) \) that \( |C(u')| < p_m \), where \( p_m \) is the largest constant in the positive quantifier from edges \( (u, u') \). (1) Since \( v \in C(x_0) \) is a match of \( x_0 \), there exists a node \( u' \in C(u') \) that is a match for \( u \). As \( u' \) is a valid match for \( u \), it must have a parent \( v \) that matches \( u \) in an isomorphic mapping, and satisfies the quantifier of edge \( e=\{(u, u')\} \) for every parent \( u \) of \( u' \). (2) This indicates that \( v \) must have at least \( p_t \) children that match \( u' \) with size at least \( p_t \), for \( v \), with quantifier \( \sigma(e) \cup p_t \), i.e., \( |C(u')| \geq p_t \). (3) Since \( (2) \) holds for every edge \( (u, u'), |C(u')| \geq p_m \). This contradicts the assumption that \( |C(u')| < p_m \). Hence Lemma 12 follows.

**Optimization For QMatch (Section 4)**

We leverage graph simulation [21] to further reduce verification. A node \( v \) in \( G \) simulates a query node \( u \) in \( Q \), i.e., (\( u, v \) is in a simulation relation \( R \) if \( v \) and \( u \) have the same label, and for each \( u \)’s child \( u' \) connected via edge \( e \), \( v \) has a child \( v' \) connected via edge \( e' \) having the same label of \( e \) such that \( (u', v') \in R \). It is known that \( R \) can be computed in quadratic time [21]. One can verify the following.

**Lemma 13:** For any edge \( e=(u, u') \) in \( \Pi(Q)(x_0) \) and its quantifier \( \sigma(e) \cup p \), \( v \) is a match of \( u \) in \( Q \) when \( x_0 \) is mapped to \( v_2 \) only if (1) \( v_2 \) simulates \( x_0, (2) v \) simulates \( u \) in \( Q \), \( (3) |R(v_2, v, G)| (\text{resp.} |R(v, v_2, F)|) \geq p \) (resp. \( p' \)).

Here \( R(v_2, v, G) \) is the set of children of \( v \) simulating \( u' \).

Indeed, one may verify that \( R(v_2, v, G) \) is an upper bound of \( |M_0(h_0(x_0), h_0(u), Q)| \) for positive edges \( e \). The result above suggests that we can use simulation as preprocessing, and remove invalid candidates early by following Lemma 13.

**Proof of Lemma 13**

One may verify that \( v \) is a match for \( u \) via isomorphism only if \( v \) can simulate \( u \). Indeed, If a node \( v \) cannot simulate \( u \), then either \( v \notin C(u) \), or no descendant \( u' \) of \( u \) in \( C(u') \). Both indicates that \( v \) cannot match \( u \) via an isomorphism. Assume that \( v \) is a match of \( u \) when \( v_2 \) is matched with \( x_0 \).

(1) If \( v_2 \) (resp. \( v \)) does not simulate \( x_0 \) (resp. \( u \)), then \( v_2 \) (resp. \( v \)) cannot match \( x_0 \) (resp. \( u \)) via isomorphism. (2) If \( |R(v_2, v, F_i)| \) fails the quantifier \( e \), then the number of possible matches in \( v \)'s children fails the quantifier. Both contradict the assumption. Lemma 13 thus follows.

**Multi-thread Quantified Matching (Section 5)**

Given \( Q(x_0) \) and fragment \( F_i \), mQMatch follows QMatch to verify the candidates in \( C(x_0) \). It dynamically spawns \( p \) threads to simultaneously verify the top \( p \) selected candidates \( v \) for each pattern edge \( e=(u, u') \) for each candidate. For a candidate \( v \in C(u) \), the thread \( i \) for \( v \) spawns a thread for each of the top \( p \) candidates \( v' \in C(u') \) to be verified. As multi-threading can be expensive when \( d \) is large, mQMatch utilizes thread blocking to avoid the overhead: (1) it only spawns at most \( b \) threads, and “sequentializes” the rest \( p-b \) verification with a single thread, and (2) it only spawns threads for the first two search levels.

**Appendix C: Confidence of QGARs (Section 6)**

One might be tempted to define the confidence of \( R(x_0) \) as \( \frac{|R(x_0, G)|}{|Q_1(x_0, G)|} \). Following traditional association rules [38]. However, this does not work well in incomplete graphs.

**Example 11:** For QGAR \( R_t \), consider two matches \( v_1 \) and \( v_2 \) in \( Q_1(x_0, G) \), where user \( v_1 \) has no edge labeled buy. Since \( G \) is usually incomplete, it is an overkill to assume that \( v_1 \) is a negative example as a potential customer of books, since some of its buy edges may possibly be missing from \( G \).

To accommodate incomplete graphs, we follow the local close world assumption (LCWA) [14], which assumes that \( G \) is locally complete, i.e., either \( G \) includes the complete neighbors of a node for any existing edge type, or it knows nothing about the neighbors. We define \( \text{conf}(R, G) \), the confidence of \( R(x_0) \) in \( G \) under LCWA, as \( \frac{|R(x_0, G)|}{|Q_1(x_0, G)|} \).

Continuing with Example 11, user \( v_2 \) is retained in \( X_0 \) but \( v_1 \) is excluded due to missing buy edges. Hence, \( v_1 \) is no longer considered to be a negative match under LCWA.