Extending Graph Pattern Matching with Regular Expressions

Xin Wang[1], Yang Wang[1], Yang Xu[2], Ji Zhang[3] and Xueyan Zhong[1]

[1] Southwest Petroleum University, China
xinwang.ed@gmail.com, wangyang@swpu.edu.cn, zhongxueyan@sohu.com
[2] Southwest Jiaotong University, China xuyang@my.swjtu.edu.cn
[3] University of Southern Queensland, Australia Ji.Zhang@usq.edu.cn

Abstract. Graph pattern matching, which is to compute the set $M(Q, G)$ of matches of Q in G, for the given pattern graph Q and data graph G, has been increasingly used in emerging applications *e.g.,* social network analysis. As the matching semantic is typically defined in terms of subgraph isomorphism, several problems are hence raised: (1) the semantic is often too rigid to identify meaningful matches, and (2) the problem is intractable, which calls for efficient matching methods. Motivated by these, this paper extends pattern graphs with regular expressions, and investigates the top-k graph pattern matching problem. (1) We introduce *regular patterns*, which revises traditional pattern graphs by incorporating regular expressions; extend traditional notion of subgraph isomorphism by allowing edge to regular path mapping; and define matching based on the revised notion. With the extension, more meaningful matches could be captured. (2) We propose a relevance function, that is defined in terms of tightness of connectivity, for ranking matches. Based on the ranking function, we introduce *top-k graph pattern matching* problem, denoted by TopK. (3) We show that TopK is intractable. Despite hardness, we develop an algorithm with *early termination property*, *i.e.,* it finds top-k matches without identifying entire match set. (4) Using real-life and synthetic data, we experimentally verify that our top-k matching algorithms are effective, and outperform traditional counterparts.

1 Introduction

Graph pattern matching has been widely used in social data analysis [3, 23], among other things. A number of algorithms have been developed for graph pattern matching that, given a pattern graph Q and a data graph G, compute $M(Q, G)$, the set of matches of Q in G (*e.g.,* [10, 16]). As social graphs are typically very large, with millions of nodes and billions of edges, several challenges to social data analysis with graph pattern matching are brought out. (1) Traditionally, graph pattern matching is defined in terms of subgraph isomorphism [15]. This semantic only allows edge-to-edge mapping, which is often too strict to identify important and meaningful matches. (2) Existing matching algorithms often return an excessive number of results, which makes inspection a daunting task. While, in practice, users are often interested in top-ranked matches [20]. (3) The sheer size of social graphs makes matching computation costly: it is NP-complete to decide whether a match exists (cf. [21]), not to mention identifying the complete match set.

These highlight the need for extending matching semantic and exploring the *top-k graph pattern matching* problem. That is given a pattern graph Q enriched with more meaningful constrains, a data graph G and an integer k, it is to find k best matches of

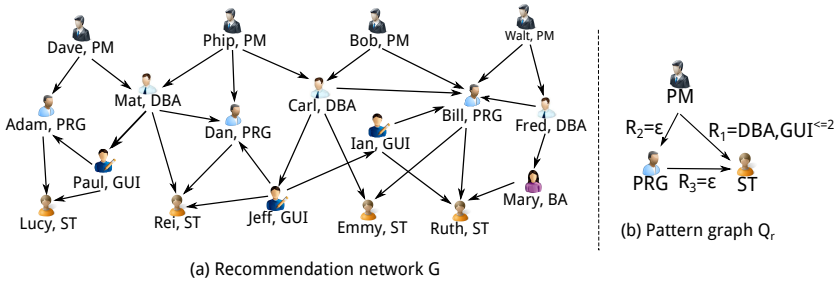


Fig. 1. Querying big graphs

Q from $M(Q, G)$, based on certain functions measuring match quality. The benefits of identifying top-k matches are twofold: (1) users only need to identify k matches of Q rather than a large match set $M(Q, G)$; (2) if we have an algorithm for computing top-k matches with *the early termination property*, *i.e.,* it finds top-k matches of Q *without* computing the entire match set $M(Q, G)$, then we do not have to pay the price of full-fledged graph pattern matching.

Example 1. A fraction of a recommendation network is given as a graph G in Fig. 1(a). Each node in G denotes a person, with attributes such as *job title*, *e.g.,* project manager (PM), database administrator (DBA), programmer (PRG), business analyst (BA), user interface developer (GUI) and software tester (ST). Each directed edge indicates a recommendation, *e.g.,* (Dave, Adam) indicates that Dave recommended Adam earlier.

To build up a team for software development, one issues a pattern graph Q_r depicted in Fig. 1(b) to find qualified candidates. The search intention constrains team members to satisfy following requirements: (1) with *job title*: PM, PRG and ST; (2) satisfying following recommendation relations: (i) PM recommended PRG before; (ii) ST has been recommended by PRG, and also been recommended by PM via a chain, that includes a DBA and at most two GUIs. Here, recommendation relations given above can be represented as regular expressions, *i.e.,* $R_1 = \mathsf{DBA}, \mathsf{GUI}^{\leq 2}$, $R_2 = \epsilon$ and $R_3 = \epsilon$, where ϵ indicates an empty string.

Traditionally, graph pattern matching is defined in terms of subgraph isomorphism, which only supports edge-to-edge mapping and is too rigid to find matches. To identify matches of Q_r, matching semantic needs to be extended. Indeed, when edge to regular path mapping is supported, match set $M(Q_r, G)$ includes following matches: $t_{r1} = (\mathsf{Dave}, \mathsf{Adam}, \mathsf{Lucy})$, $t_{r2} = (\mathsf{Phip}, \mathsf{Dan}, \mathsf{Rei})$ and $t_{r3} = (\mathsf{Bob}, \mathsf{Bill}, \mathsf{Ruth})$.

In practice, $M(Q_r, G)$ is possibly a large set, while users may only be interested in top-k matches of Q_r. It is hence unnecessary and too costly to compute the entire large set $M(Q_r, G)$. In light of this, an algorithm with the *early termination property* is desired, as it identifies top-k matches without inspecting entire $M(Q_r, G)$.

To measure relevance of top-k matches, one may consider tightness of connectivity [4]. Observe that matches t_{r1} and t_{r2} are connected more tightly than t_{r3} due to shorter inner distances, and are considered more relevant to the query. Hence, $\{t_{r1}, t_{r2}\}$ makes a good candidate for top-2 matches in terms of relevance. □

This example shows that *top-k graph pattern matching* with enriched matching semantic and top-k search may rectify the limitations of existing matching algorithms. To make practical use of it, several questions have to be answered. (1) What ranking function should be used to measure the quality of matches? (2) What is the complexity of computing top-k matches based on the function? (3) How can we develop *early termination* algorithms for computing top-k matches?

Contributions. This paper investigates the following issues. (1) We revise the traditional notion of graph pattern matching. With the revision, mapping between an edge and a regular path is supported, thus, sensible matches can be captured. (2) We give a full treatment for *top-k graph pattern matching*, with effective algorithms that preserve *early termination property*.

(1) We incorporate regular expressions in pattern graphs and introduce *regular patterns* (or *r-pattern* for short) Q_r. We revise the traditional notion of subgraph isomorphism by supporting edge to regular path mapping, and define graph pattern matching with the revised notion, that's given Q_r and G, it is to compute the set $M(Q_r, G)$ of matches of Q_r in G, where each edge e_p of Q_r is mapped to a path in G satisfying regular expression specified by e_p. We also introduce a function to rank matches of Q_r, namely, *relevance function* $w(\cdot)$ that is defined in terms of tightness of connectivity. Based on the function, we introduce the *top-k graph pattern matching problem* (Section 2).

(2) We investigate the *top-k graph pattern matching problem* (Section 3). Given a graph G, a *r-pattern* Q_r, and a positive integer k, it is to find k top-ranked matches of Q_r, based on the relevance function $w(\cdot)$. We show that the decision version of the problem is NP-hard. Despite hardness, we provide an algorithm with *early termination property* [8], that is, it stops as soon as top-k matches are found, without computing entire match set.

(3) Using both real-life and synthetic data, we experimentally verify the performance of our algorithms (Section 4). We find the following. Our top-k matching algorithms are not only effective, but also efficient. For example, on *Youtube*, our algorithm takes only 10 seconds, and inspects 38% of matches of Q_r, when it terminates. Moreover, they are not sensitive to the increase of k, and size $|G|$ of G.

These results yield a promising approach to querying big graphs. All the proofs, and complexity analyses can be found in [1].

Related work. We next categorize the related work as follows.

Pattern matching with regular expressions. Given a pattern graph Q and a data graph G, the problem of graph pattern matching is to find all the matches in G for Q. Typically, matching is defined in terms of subgraph isomorphism [6] and graph simulation [16]. Extended from traditonal notions, a host of work [2, 9, 11, 13, 22, 24] incorporate matching semantic with regular expressions and explored regular pattern matching problem. Among these, a few works focus on regular simple path queries (RSPQs), which is a special case of regular pattern matching. For example, [11] introduced a parallel algorithm to answer regular reachability queries based on partial evaluation. [24] still developed parallel algorithms via a set of operations with/without indices, for extremely large graphs; while the technique is more appropriate for regular reachability queries. [2] characterizes the frontier between tractability and intractability for (RSPQs, and summarizes the data complexity. Path indexes were used by [13] to generat efficient execution plans for RSPQs. For generalized regular pattern matching, [22] formalized regular queries as nonrecursive Datalog programs, and shows elementary tight bounds for the containment problem for regular queries. [9] incorporates regular expressions on pattern graphs, while the matching semantic is graph simulation, which is different to ours.

Top-k search. Top-k search is to retrieve k top-ranked tuples from query result.

Topk-k search with early termination. Given a monotone scoring function and sorted lists, one for each attribute, Fagin's algorithm [7] reads attributes from the lists in par-

allel with sorted access, and constructs tuples with the attributes. It stops sorted access when k tuples are constructed. For each object that has been seen but misses attributes, it does random access to find those missing attributes. Extending Fagin's algorithm, the threshold algorithm [8] reads all attributes of a tuple from ordered lists via sorted access in parallel, in the meanwhile, it performs random access to other lists to complement missing attributes. The process repeats until a threshold τ is reached. Other top-k search methods essentially extend the algorithms of [7, 8] (see [18] for a survey).

Top-k graph pattern matching. There has been a host of work on this topic. For example, [27] proposes to rank matches, *e.g.,* by the total node similarity scores, and identify k matches with highest ranking scores. [15] investigates top-k query evaluation for twig queries, which essentially computes isomorphism matching between rooted graphs. To provide more flexibility for top-k pattern matching, [4] extends matching semantics by allowing edge-to-path mapping, and proposes to rank matches based on their compactness. To diversify search results and balance result quality and efficiency, an algorithm with approximation ratio is proposed to identify diversified top-k matches [25]. Instead of matching with subgraph isomorphism, graph simulation [16] is applied as matching semantic, and pattern graph is designated with an output node, referred to as "query focus" in [12], then match result includes a set of nodes that are matches of the "query focus".

Our work differs from the prior work in the following. (1) The problem we studied has different matching semantics from earlier works. With the new semantic, more sensible matches could be captured. (2) The high computational cost of previous methods for top-k matching hinders their applications in the real world. In contrast, our algorithm possesses *early termination property* and yields a practical method for querying real-life social graphs.

2 Preliminaries

In this section, we first review data graphs and pattern graphs. We then introduce graph pattern matching, followed by the *top-k graph pattern matching problem*.

2.1 Data Graphs and Pattern Graphs

We start with notions of data graphs and pattern graphs.

Data graphs. A *data graph* (or simply a graph) is a directed graph $G = (V, E, L_v)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$, in which (v, v') denotes an edge from v to v'; and (3) L_v is a function defined on V such that for each node v in V, $L_v(v)$ is a tuple $(A_1 = a_1, \cdots, A_n = a_n)$, where $A_i = a_i (i \in [1, n])$, indicating that the node v has a value a_i for the attribute A_i, and denoted as $v.A_i = a_i$.

We shall use the following notations for data graphs G. (1) A *path* ρ from v_0 to v_n, denoted as $\rho(v_0, v_n)$, in G is a sequence $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots v_{n-1} \xrightarrow{e_n} v_n$, where (a) $v_i \in V$ for each $i \in [0, n]$, and (b) $e_j = (v_{j-1}, v_j)$ in E for each $j \in [1, n]$. (2) We say a path ρ is nonempty if it contains more than one edge.

Pattern graphs. A *pattern graph* is a directed graph $Q_r = (V_p, E_p, f_v, f_e)$, where (1) V_p is the set of *pattern nodes*; (2) E_p is the set of *pattern edges*; (3) f_v is a function defined on V_p such that for each node $u \in V_p$; $f_v(u)$ is a predicate defined as a conjunction of atomic formulas of the form of 'A op a', such that A denotes an attribute of the node u, a is a value, and op is a comparison operator in the set $\{<, \leq, =, \neq, >, \geq\}$; and (4) f_e is a function on E_p such that $f_e(e_p)$ associates a regular expressions R on pattern edge e_p. We define R as below:

$$R ::= \epsilon \mid l \mid l^{\leq k} \mid l^{+} \mid RR\ .$$

Here (1) ϵ is the empty string; (2) l is a label in Σ; (3) k is a positive integer, and $l^{\leq k}$ is a string $l \cdots l$, that consists of j ($j \in [1, k]$) occurrence of label l; (4) l^{+} indicates one or more occurrences of label l; and (5) RR indicates concatenation.

Intuitively, the predicate $f_v(u)$ of a node u specifies a search condition on labels and data contents. We say that a node v in a data graph G satisfies the search condition of a pattern node u in Q_r, denoted as $v \sim u$, if for each atomic formula 'A op a' in $f_v(u)$, there exists an attribute A in $L_v(v)$ such that $v.A$ op a. The function f_e assigns a regular expression on each edge (u, u') of Q_r. As will be seen shortly, regular expressions enable matching from edge-to-edge mapping to edge to regular path mapping. We refer to a pattern as a *normal pattern* and denote it as $Q = (V_p, E_p, f_v)$, when each edge of Q takes a unit weight, *i.e.,* $R = \epsilon$.

We denote $|V_p| + |E_p|$ as $|Q|$ (the size of Q), and $|V| + |E|$ as $|G|$ (the size of G).

Example 2. As shown in Fig. 1 (b), pattern graph Q_r is a *r-pattern*, in which each node takes a predicate, *e.g., job title*="PM", to specify search conditions on nodes; and each edge carries a regular expression as edge weight, *e.g.,* $f_e(\mathsf{PM}, \mathsf{ST}) = (\mathsf{DB}, \mathsf{GUI}^{\leq 2})$. As will be seen shortly, regular expressions associated on pattern edges e_p impose constrains on paths that are mapped from e_p.

2.2 Graph Pattern Matching Revised

We now introduce the graph pattern matching problem based on the revised pattern graph. Consider a data graph $G = (V, E, L_v)$ and a *r-pattern* $Q_r = (V_p, E_p, f_v, f_e)$.

Graph Pattern Matching. A *match* of Q_r in G is an n-ary node-tuple $t = \langle v_1, \cdots, v_n \rangle$, where $v_i \in V$ ($i \in [1, n]$, $n = |V_p|$), and there exists a *bijective function* h from V_p to the nodes in t such that (1) for each node $u \in V_p$, $h(u) \sim u$; and (2) (u, u') is an edge in Q_r if and only if there exists a path ρ from $h(u)$ to $h(u')$ in G that satisfies the regular expression $f_e(u, u') = R$, *i.e.,* the path ρ constitutes a string that is in the regular language defined by R.

Indeed, (1) for a *r-pattern* Q_r, the regular expression $f_e(e_p)$ of a pattern edge $e_p = (u_1, u_2)$ imposes restriction of mapping from an edge e_p to a path ρ in G: that is, a pattern edge e_p can be mapped to a nonempty path $\rho = v_1 \xrightarrow{e_1} v_1' \xrightarrow{e_2} \cdots v_{n-1}' \xrightarrow{e_n} v_2$ in a data graph G, such that the string of node labels $L_v(v_1.A)L_v(v_2.A) \cdots L_v(v_n.A)$ satisfies $f_e(e_p)$; and (2) a *normal pattern* enforces edge-to-edge mappings, as found in subgraph isomorphism. Traditional graph pattern matching is defined with *normal patterns* [6].

We shall use following notations for graph pattern matching.

(1) The *answer* to Q_r (resp. Q) in G, denoted by $M(Q_r, G)$ (resp. $M(Q, G)$), is the set of node-tuples t in G that matches Q_r (resp. Q). (2) Abusing notation of *match*, in a graph G, we say a pair of nodes (v, v') is a *match* of a pattern edge $e_p = (u, u')$, if (a) v, v' are the matches of u, u', respectively, and (b) either $\rho(v, v')$ satisfies R when $f_e(e_p) = R$, or (v, v') is an edge when e_p belongs to a normal pattern Q. (3) Given a Q_r, we use $Q_r(e_p)$ to denote a *r-pattern* that contains a single edge $e_p = (u_s, u_t)$ with regular expression R associated, where e_p is in Q_r. Then, for a pair of nodes (v, v'), to determine whether it is a match of $Q_r(e_p)$ is equivalent to a regular reachability query $q_R = (v, v', u_s, u_t, R)$.

Example 3. Consider *r-pattern* Q_r in Fig. 1 (b). Each edge e_p of Q_r is associated with a regular expression that specifies labels on the path that e_p maps to. Taking a match $\{\mathsf{Dave}, \mathsf{Adam}, \mathsf{Lucy}\}$ as example, it is mapped via h from V_p, and for each pattern edge (u, u'), there exists a path from $h(u)$ to $h(u')$ satisfying regular expression $f_e(u, u')$, *e.g.,* pattern edge $(\mathsf{PM}, \mathsf{ST})$ is mapped to a path $\mathsf{Dave} \rightarrow \mathsf{Mat} \rightarrow \mathsf{Paul} \rightarrow \mathsf{Lucy}$, satisfying R_1 of Q_r.

Remark. General regular expressions, which consider operations *e.g.,* alternation and Kleene star [20], possess more powerful expressive abilities via higher computational cost. To strike a balance between expressive power and computational cost, we choose a subclass R of general regular expressions, as R already has sufficient expressive power to specify edge relationships commonly found in practice.

2.3 Top-k Graph Pattern Matching Problem

In practice, match set could be excessively large on big graphs, while users are interested in the best k matches of pattern graphs [18]. This motivates us to study the *top-k graph pattern matching problem* (TopK). To this end, we first define a function to rank matches, and then introduce TopK.

Relevance function. On a match t of Q_r in G, we define the relevance function $w(\cdot)$ as following:

$$w(t) = \sqrt{\frac{|E_p|}{f_d(t)}},$$

where $f_d(t) = \Sigma_{(u,u')\in E_p, h(u), h(u')\in t}(\mathsf{dist}(h(u), h(u')))^2$. That is, the relevance function favors those matches that are connected tightly. The more tightly the nodes are connected in a match t, the more relevant t is to Q_r, as observed in study [4]. Thus, matches with high $w(\cdot)$ values are preferred.

Top-k matching problem. We now state the *top-k graph pattern matching problem*. Given a graph G, a *r-pattern* Q_r, and a positive integer k, it is to find a set $\mathcal{S}$ of matches of Q_r, such that $|\mathcal{S}| = k$ and

$$w(\mathcal{S}) = \underset{\mathcal{S}'\subseteq M(Q_r,G), |\mathcal{S}'|=k}{\arg\max} \sum_{t\in\mathcal{S}'} w(t).$$

Abusing notation $w(\cdot)$, we use $w(\mathcal{S})$ to denote $\sum_{t\in\mathcal{S}} w(t)$, referred to as the *relevance* of $\mathcal{S}$ to Q_r. Thus, TopK is to identify a set of k matches of Q_r, that maximizes total relevance. In other words, for all $\mathcal{S}' \subseteq M(Q_r, G)$, if $|\mathcal{S}'| = k$, then $w(\mathcal{S}) \geq w(\mathcal{S}')$.

Example 4. Recall graph G and *r-pattern* Q_r in Fig. 1. The relevance values of matches of Q_r are $w(t_{r1}) = \sqrt{\frac{3}{11}}$, $w(t_{r2}) = \sqrt{\frac{3}{11}}$, $w(t_{r3}) = \sqrt{\frac{3}{18}}$, thus, t_{r1}, t_{r2} are more relevant to Q_r than t_{r3}.

We summarize notations used in the paper in Table 1.

3 Efficient Top-k Graph Pattern Matching

In this section, we study the *top-k graph pattern matching problem*. The main results of the section are as follows.

Theorem 1. *The* TopK *(1) is* NP*-hard (decision problem); (2) can be solved by an early termination algorithm in $O(|G|^{|Q_r|} \cdot \mathsf{log}(|G|^{|Q_r|}) + |G|!|G|)$ time.*

Symbols	Notations
$Q = (V_p, E_p, f_v)$	a *normal pattern*
$Q_r = (V_p, E_p, f_v, f_e)$	a *regular pattern*
Q_T	a spanning tree of pattern graph Q_r (resp. Q)
$G = (V, E, L_v)$	a data graph
$G_q = (V_q, E_q, L'_v, L_e)$	a twisted graph of G
$M(Q_r, G)$ (resp. $M(Q, G)$)	a set of matches of Q_r (resp. Q) in G
$\|V_p\| + \|E_p\|$	$\|Q_r\|$ (resp. $\|Q\|$), the size of Q_r (resp. Q)
$\|V\| + \|E\|$	$\|G\|$, the size of G
$w(\cdot)$	the relevance function
$q_R = (v, v', u_s, u_t, R)$	a regular reachability query
$Q_r(e_p)$	a *r-pattern* that contains a single edge e_p
$Q_r(R)$	a query automaton of regular expression R

Table 1. A summary of notations

Proof. We first show Theorem 1(1). We defer the proof of Theorem 1(2) to Section. 3.2, where an *early termination* algorithm is given as a constructive proof.

The decision problem of TopK is to decide, given a pattern Q_r, a graph G, a positive integer k and a bound B, whether there exists a subset $\mathcal{S}$ of $M(Q_r, G)$ with size k, such that $w(\mathcal{S}) \geq B$. We show that TopK is NP-hard, by reduction from the NP-complete *subgraph isomorphism problem* (ISO).

Given a normal pattern Q, and a data graph G, ISO is to determine whether there exists a subgraph G_s of G that is isomorphic to Q. Given such an instance of ISO, we construct an instance of TopK as follows: (a) we construct a pattern Q_o that has the same node and edge sets as Q; (b) we construct a graph G_o with the same set of nodes and edges as G; and (c) we set $k = 1$ and $B = 1$. The construction is obviously in PTIME. We next verify that there exists a subgraph G_s of G that is isomorphic to Q_r, if and only if, there exists a subset $\mathcal{S}$ of $M(Q_r, G)$ with $|\mathcal{S}| = 1$ and $w(\mathcal{S}) = 1$.
(1) Assume that there exists a subgraph G_s of G that is isomorphic to Q. One can verify that there must exist a subgraph G'_s of G_o that is isomorphic to Q_o, and moreover, $\mathcal{S}$, that includes a single element G'_s and has $w(\mathcal{S}) = 1$, is the top-1 match set.
(2) Conversely, if there exists a subset $\mathcal{S}$ of $M(Q, G)$ with $|\mathcal{S}| = 1$ and $w(\mathcal{S}) = 1$, it is easy to see that there must exist a subgraph G_s of G that is isomorphic to Q.

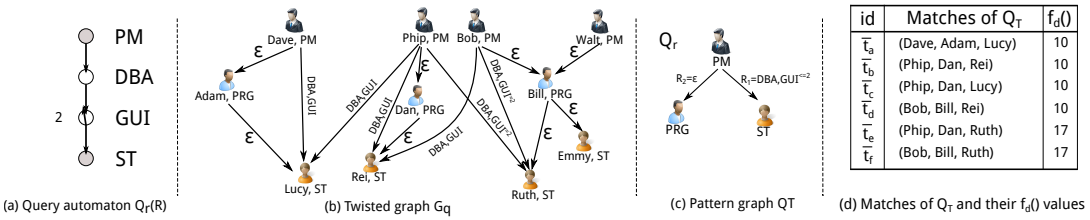
As ISO is NP-complete, so is NP-hardness of TopK. □

Due to intractability, there does not exist polynomial time algorithm for TopK. To tackle the issue, one may simply develop an algorithm, that is denoted as $\mathsf{Naive_r}$, and works as following: (1) find all the matches of Q_r in G, and (2) pick k most relevant matches from match set $M(Q_r, G)$. Though straightforward, the algorithm is quite cost-prohibitive, as it requires to (1) process mapping from an edge to a regular path instantly, (2) find out entire match set from big G, and (3) pick k top-ranked matches from (possibly very big) match set $M(Q_r, G)$.

To rectify these, two main challenges have to be settled: (1) how to efficiently deal with edge to regular path mapping when identifying matches of Q_r? and (2) how to terminate earlier as soon as top-k matches are identified? In the following, we provide techniques to address the issues, for TopK.

3.1 Query Automaton

A regular pattern Q_r takes a regular expression R on each edge. To find matches of Q_r in a graph G, one needs to identify matches (v, v') of $Q_r(e_p)$ in G, for each pattern



(a) Query automaton $Q_r(R)$ (b) Twisted graph G_q (c) Pattern graph Q_T (d) Matches of Q_T and their $f_d()$ values

Fig. 2. Automaton $Q_r(R)$, Twisted graph G_q, spanning tree Q_T of Q_r and matches of Q_T

edge $e_p = (u_s, u_t)$. This calls for techniques to evaluate regular reachability queries $q_R = (v, v', u_s, u_t, R)$. In light of this, we introduce *query automaton*, which is a variation of nondeterministic finite state automaton (NFA), to facilitate evaluation of regular reachability queries.

Query automaton. A *query automaton* $Q_r(R)$ of a regular expression R is defined as a six tuple $(V_q, E_q, L_s, L_i, u_s, u_t)$, where (1) V_q is a finite set of states, (2) $E_q \subseteq V_q \times V_q$ is a set of transitions between the states, (3) L_s is a function that assigns each state a label in R, (4) L_i is a function that assigns a positive integer or a symbol "$*$" to a state, and (5) u_s and u_t in V_q are the initial and final states, respectively.

Abusing notations, we say that a node v in G is a *match* of a state u_v in $Q_r(R)$, denoted as $v \sim u_v$, if and only if (a) there exists an attribute A in $L_v(v)$ such that $v.A = L_s(u_v)$, and (b) there exist a path ρ from v to t and a path ρ' from u_v to u_t, such that ρ and ρ' have the same label. A state u is said a *child* of u', if there is an edge (u', u) in E_q, *i.e.,* u' can transit to u. In contrast to the transition of traditional NFA, the transitions of our query automaton follows the restriction imposed by regular expression R, *i.e.,* at state u_v, for each edge (v, v') on a path in graph G, a transition $u_v \to u_v$ can be made via $(u_v, u_v) \in E_q$ if $v \sim u_v$ and $v' \sim u'_v$.

Given a regular expression R, its corresponding query automaton $Q_r(R)$ can be constructed in $O(|R|\log|R|)$ time, following the conversion of [17]; moreover the function L_i is defined as follows: for each entry in R, if it is of the form $l^{\leq k}$ (resp. l^*), then the corresponding state s is associated with an integer k (resp. label $*$) via $L_i(\cdot)$ to indicate maximum number (resp. unlimited number) of occurrence of s; otherwise, $L_i(s)$ is set as 1 by default.

Example 5. Recall regular expression $R_1 = \mathsf{DBA}, \mathsf{GUI}^{\leq 2}$ of Q_r in Fig. 1(b). Its query automaton $Q_r(R)$ is depicted in Fig. 2(a). The set V_q has four states PM, DBA, GUI, ST, where the initial and final states are PM and ST, respectively. The set E_q of transitions is $\{(\mathsf{PM}, \mathsf{DBA}), (\mathsf{DBA}, \mathsf{GUI}), (\mathsf{GUI}, \mathsf{GUI}), (\mathsf{GUI}, \mathsf{ST})\}$. Observe that the edge $(\mathsf{GUI}, \mathsf{GUI})$ is associated with weight 2, indicating that transition from GUI to GUI is restricted to no more than twice. In contrast to NFA, the query automaton $Q_r(R)$ is to accept paths in *e.g., G* of Fig. 1(a), and its transitions are made by matching the labels of its states with the value of attribute *job title* of nodes on the paths. □

Lemma 1. *In a graph G, a pair of nodes (v_s, v_t) is a match of a regular reachability query $q_R = (v_s, v_t, u_s, u_t, R)$ if and only if v_s is a match of u_s in Q_r.*

Proof. We show Lemma 1 by contradiction. Assume that (a) (v_s, v_t) is a match of q_R, while (b) v_s is not a match of u_s in $Q_r(R)$. Condition (a) ensures that there exists a path from v_s to v_t such that node labels on the path equals to R, then by definition, v_s must be a match of u_s, which contradicts condition (b). Hence assumption is invalid, and Lemma 1 follows. □

Input: A graph G, a reachability query $q_R = (v_s, v_t, u_s, u_t, R)$.
Output: The Boolean answer ans to q_R in G.

1. initialize a stack $\mathsf{T} := (u_s, v_s)$; ans:=false;
2. construct query automaton $Q_r(R) = (V_q, E_q, L_s, L_i, u_s, u_t)$;
3. **while** $\mathsf{T} \neq \emptyset$ **do**
4. $(u, v) := \mathsf{T}.\mathsf{pop}()$;
5. **for each** child u' of u **and each** child v' of v **do**
6. **if** $u' = u_t$ **and** $v' \sim u'$ **then** ans:=true; **break** ;
7. **else if** $\gamma(u') \leq L_i(u')$ **and** $v' \sim u'$ **then** push (u', v') in T;
8. **return** ans;

Fig. 3. Procedure RegReach

3.2 An Early Termination Algorithm

As is shown, the inefficiency of the algorithm $\mathsf{Naive}_\mathsf{r}$ is mostly caused by the search strategy it applied. To avoid exhaustive search, we develop an algorithm for TopK with *early termination property*, *i.e.,* it stops as soon as top-k matches are found, without identifying entire (possibly very big) match set.

Below, we start from a notion *twisted graph* (TG for short) and its construction strategy, followed by the illustration of the *early termination* algorithm.

Twisted Graph. Given a graph G and a *r-pattern* Q_r, the twisted graph G_q of Q_r in G is a quadruple (V_q, E_q, L'_v, L_e), where V_q and E_q are the sets of nodes and edges, respectively; L'_v is defined the same as L_v in G; and L_e specifies a set of regular expressions, *i.e.,* node labels on $\rho(v, v')$ in graph G, for each edge (v, v') of G_q.

Note that, each edge (v, v') in G_q may take multiple regular expressions, as there may exist multiple paths from v to v' in G, that satisfy different regular expressions imposed by pattern edges. All the regular expressions associated on a single edge of G_q are concatenated with disjunction.

Example 6. Recall graph G and *r-pattern* Q_r in Fig. 1. The corresponding twisted graph G_q is shown Fig. 2 (b). Observe that the regular expression associated with each edge in G_q imposes either edge-to-edge mapping (via ϵ) or edge-to-path mapping. □

Construction of TG. The construction algorithm for TG, denoted as TwistGen (not shown), takes graph G and pattern Q_r as input, and works as follows. It first identifies node pairs (v, v') in G with $v \sim u$ and $v' \sim u'$, for each pattern edge $e_p = (u, u')$ of Q_r. It then verifies whether there exists a path from v to v' in G, that satisfies regular expression R imposed by e_p. It next includes nodes v, v' and an edge (v, v') with regular expression R in G_q, if there exists a path from v to v' satisfying R. Indeed, the verification of (v, v') as a match of (u, u') can be processed via evaluation of regular reachability queries. Below, we present a procedure RegReach, which evaluates regular reachability queries by simulating query automaton.

Regular reachability queries. The procedure, denoted as RegReach is shown in Fig. 3. It takes a graph G and a regular reachability query $q_R = (v, v', u_s, u_t, R)$ as input, and returns a boolean value to indicate whether there exists a path $\rho(v, v')$ in G satisfying regular expression R. More specifically, it first initializes a stack T with node pair (u_s, v_s), and a boolean variable ans as false (line 1). It then constructs a query automaton $Q_r(R)$ following the conversion of [17] (line 2). In the following, it performs depth first search to identify whether v_s can reach v_t via a path satisfying R (lines 3-7). RegReach first pops out the upper most element (u, v) from stack T (line 4), it next verifies whether the

final state of automaton is encountered, *i.e.,* the child v' of v satisfies search condition imposed by the child u' of u, which is u_t. If so, RegReach sets ans as true, and breaks **while** loop (line 6); otherwise, it checks whether (a) $\gamma(u')$ is larger than $L_i(u')$, *i.e.,* occurrence upper bound of state u, where $\gamma(u')$ indicates the visiting time of u' during the traversal, and (b) $v' \sim u'$, and pushes (u', v') in T, if both conditions (a) and (b) are satisfied (line 7). After **while** loop terminates, RegReach returns ans as final result (line 8).

Example 7. Given graph G in Fig. 1(a), and a regular reachability query $q_R = (\mathsf{Dave}, \mathsf{Lucy}, \mathsf{PM}, \mathsf{ST}, R_1)$, RegReach evaluates q_R as following. It first initializes stack with $(\mathsf{PM}, \mathsf{Dave})$, and then constructs a query automaton $Q_r(R)$, as shown in Fig. 2(a). It next conducts guided search in graph G. The search starts from PM in $Q_r(R)$ and Dave in G, respectively, and visits DB, GUI, ST in $Q_r(R)$ and Mat, Paul, Lucy in G. As the terminal state in $Q_r(R)$ is reached, the path Dave $\rightarrow$ Mat $\rightarrow$ Paul $\rightarrow$ Lucy is accepted by the query automaton, and a truth answer is returned. □

Proposition 1. *Given a data graph G and a r-pattern Q_r, matches of Q_r in G can be found from the twisted graph G_q of G.*

Proof. We show Proposition 1 by (1) outlining an algorithm, that computes $M(Q_r, G_q)$ on G_q, and (2) showing $M(Q_r, G_q) = M(Q_r, G)$.

(1) Algorithm. The algorithm takes Q_r, G_q as input, and simulates a traditional isomorphism checking algorithm, *i.e.,* VF2 [6] with two exceptions. (1) For each pattern edge $e_p = (u_1, u_2)$ with regular expression R, and a node v_1 as a candidate match of u_1, it verifies whether there exists an edge (v_1, v_2) in G_q such that R appears in $L_e(v_1, v_2)$, for consistency checking. (2) It exhaustively finds all the matches, rather than checking existence of a match of Q_r. When the algorithm terminates, it returns a set $M(Q_r, G_q)$ of matches.

(2) Result equivalence. We show $M(Q_r, G_q) = M(Q_r, G)$ by contradiction. Suppose $M(Q_r, G_q) \neq M(Q_r, G)$, then either (a) there exists a match t_a that is in $M(Q_r, G_q)$ but not in $M(Q_r, G)$, or (b) there exists a match t_b that is in $M(Q_r, G)$ but not in $M(Q_r, G_q)$. For case (a), one may verify that $t_a \notin M(Q_r, G)$, indicates that at least one edge e_p in Q_r can not be mapped to any node pair of t_a. While, $t_a \in M(Q_r, G_q)$ denotes that e_p can be mapped to edge (v, v') in G_q. This contradicts the assumption (a), because if (v, v') in G_q is a mapping from e_p, then node pair (v, v') in G can be mapped from e_p as well. For case (b), one can easily find contradiction along the same line as case (a). Thus, $M(Q_r, G_q)$ must be equivalent to $M(Q_r, G)$.

Putting these together, Proposition 1 follows. □

Indeed, Proposition 1 suggests one way to cope with the first challenge, *i.e.,* we can construct a twisted graph G_q first, and compute top-k matches on G_q, instead of big G. To address the second challenge, a sufficient condition, which is used as termination condition is missing. Fortunately, Proposition 2 fills this critical void.

Proposition 2. *Given a list L_c of matches $\bar{t}_i$ ($i \in [1, n]$) of Q_T in G_q, that is sorted in ascending order of $f_d(\bar{t}_i)$. Do sorted access to entries of L_c. An entry $\bar{t}_j$ of L_c can not contribute to a top-k match of Q_r, if (1) at least k valid matches have been identified from the first l matches of Q_T ($l < j$) and (2) $f_d(t^m) < f_d(\bar{t}_j)$.*

Here, Q_T is a spanning tree of Q_r, t_i refers to a match of Q_r, that is generated from $\bar{t}_i$, t^m indicates a match of Q_r, that's generated from $\bar{t}_l$ ($l < j$) and has the maximum $f_d(\cdot)$ value among all the matches that have been generated.

Input: Data graph G, pattern graph Q_r and integer k.
Output: A k-element set of matches of Q_r.

1. set $\mathsf{M} := \emptyset$;
2. a *twisted graph* G_q:=$\mathsf{TwistGen}(G, Q_r)$;
3. generate a spanning tree Q_T of Q_r;
4. $\mathsf{M} := \mathsf{TMat}(G_q, Q_r, Q_T, k)$;
5. **return** M;

Procedure TMat
Input: G_q, Q_r, Q_T, k.
Output: A set of matches of Q_r.

1. a min-heap $\mathsf{S} := \emptyset$; set $\mathsf{L} := \emptyset$;
2. pick a node u from Q_T; initialize $\mathsf{can}(u)$;
3. **for each** v in $\mathsf{can}(u)$ **do**
4. identify match set H of Q_T s.t. $\bar{t}$ in H contains v as a match of u;
5. $\mathsf{L} := \mathsf{L} \cup \mathsf{H}$;
6. sort matches $\bar{t}$ in L in ascending order of $f_d(\bar{t})$;
7. **while** $\mathsf{L} \neq \emptyset$ **do**
8. pick the top-most match $\bar{t}$ in L; generate match t with $\bar{t}$;
9. $\mathsf{S} := \mathsf{S} \cup \{t\}$; $\mathsf{L} := \mathsf{L} \setminus \{\bar{t}\}$;
10. **if** termination condition is satisfied **then**
11. **break** ;
12. **if** $|\mathsf{S}| > k$ **then** retain k matches with least $w(t)$ in S;
13. **return** S;

Fig. 4. Algorithm TopKR

Proof. We show Proposition 2 by contradiction. Assume that, following sorted access, entry $\bar{t}_j$ can contribute to a top-k match of Q_r, when (1) at least k valid matches of Q_r have been identified with $\bar{t}_l$ ($l < j$), and (2) $f_d(t^m) < f_d(\bar{t}_j)$. Observe that (a) $f_d(\bar{t}_j) \geq f_d(\bar{t}_{j-1})$, as entries in L_c are sorted in ascending order of their $f_d(\cdot)$ values; and (b) $f_d(\bar{t}_j) \leq f_d(t_j)$, for t_j that's generated from $\bar{t}_j$, since $\bar{t}_j$ is a subgraph of t_j. By (a) and (b), $f_d(\bar{t}_{j-1}) \leq f_d(\bar{t}_j) \leq f_d(t_j)$. Then by assumption that $f_d(t^m) < f_d(\bar{t}_j)$, one can easily infer that $f_d(t^m) < f_d(\bar{t}_j) \leq f_d(\bar{t}_{j+1}), \cdots, \leq f_d(\bar{t}_n)$, for any t_j (resp. $t_{j+1}, \cdots, t_n$) that's generated from $\bar{t}_j$ (resp. $\bar{t}_{j+1}, \cdots, t_n$). This indicates that t_j can not contribute to a match with less $f_d(\cdot)$ value than that of t^m. By assumption that there already exist at least k matches identified with $\bar{t}_l$ ($l < j$), then top-k matches must be chosen from existing matches, hence $\bar{t}_j$ can not contribute to a top-k match. This leads to the contradiction of the assumption, Proposition 2 hence follows. □

We are now ready to outline the algorithm for TopK.

Algorithm. The algorithm, denoted as TopKR, is shown in Fig. 4. It first initializes a set M to maintain top-k matches of Q_r (line 1), constructs a *twisted graph* G_q with procedure TwistGen that is introduced earlier (line 2), and generates a spanning tree Q_T of Q_r, with an algorithm introduced in [26] (line 3). TopKR then invokes TMat to identify top-k matches (line 4), and returns output M of TMat as final result (line 5). In particular, TMat works in two steps: (a) it generates a set of matches of Q_T (a spanning tree of Q_r), as candidate matches of Q_r; and (b) it repeatedly verifies whether these candidate matches are embedded in true matches, until termination condition is met. Below, we elaborate TMat with details.

Procedure TMat takes a twisted graph G_q, a pattern Q_r, its spanning tree Q_T and an integer k as input, and works as following. (1) It initializes an empty min-heap S to

maintain matches of Q_r, and an empty set L to keep track of candidate matches (line 1). (2) It picks a node u from pattern Q_T, and initializes a set $\mathsf{can}(u)$ that includes all the candidate matches v of u, *i.e.,* $v \sim u$, in G_q (line 2). (3) TMat generates matches $\bar{t}$ of Q_T as candidate matches of Q_r (lines 3-4). Specifically, TMat traverses G_q starting from node v, following the structure of Q_T, and identifies matches $\bar{t}$ of Q_T. After traversal finished, TMat extends L with H, which contains a set of match candidates. The above process repeats $|\mathsf{can}(u)|$ times. When all the candidate matches are generated, TMat ranks $\bar{t}$ in L in ascending order of their $f_d(\cdot)$ values (line 6). (4) TMat then iteratively verifies whether each of candidate matches can turn to a valid match of Q_r until termination condition is encountered (lines 7-11). In each round iteration, only the top most candidate $\bar{t}$ will be used to generate true match (line 8). Note that the generation simply follows a structural expansion on G_q with $\bar{t}$. After match t is generated, TopKR extends S with it, and removes $\bar{t}$ from L (line 9). At this moment, TopKR verifies termination condition specified in Prop. 2, and breaks **while** loop if S already contains at least k matches, and no other match with less relevance exists (lines 10-11). After loop, if S contains more than k matches, TopKR simply picks top-k ones from it (line 12). TopKR returns set S as top-k match set (line 13).

Example 8. Given graph G and *r-pattern* Q_r in Fig. 1, TopKR finds top-2 matches as follows. It first constructs a *twisted graph* G_q with TwistGen, and generates a spanning tree Q_T. It then identifies a set of matches of Q_T, and sorts them in ascending order of their $f_d()$ values. The *twisted graph* G_q, spanning tree Q_T, and sorted match candidates are shown in Fig. 2(b-d), respectively. TopKR identifies top-2 matches with match candidates $\bar{t}_a, \cdots, \bar{t}_l$ successively. In particular, after the first four candidates are checked, only two valid matches t_a, t_b are obtained from $\bar{t}_a$, $\bar{t}_b$, respectively, whereas $\bar{t}_c$ and $\bar{t}_d$ can not contribute any valid match. After the fourth iteration, TopKR found that termination condition is encountered, and terminates computation immediately, with $\{t_a, t_b\}$ as top-2 matches. □

Correctness. To see the correctness of TopKR, observe the following. (1) TwistGen, that is employed by TopKR, correctly computes a *twisted graph* G_q, in which all matches of Q_r are embedded. Note that during the construction of a *twisted graph*, RegReach (a) correctly constructs a query automaton with technique given in [17]; (b) simulates depth first search to identify whether there exists a path $\rho(v, v')$ satisfying R, and breaks the loop once final state of $Q_r(R)$ is reached or keeps searching if forthcoming node can be visited and included in the path. (2) A spanning tree Q_T of Q_r can be correctly generated with an algorithm in [26]. (3) TMat generates the complete set of matches of Q_T, as candidate matches of Q_r, and repeatedly verifies whether these candidate matches can turn to matches of Q_r via a **while** loop. Observe that the **while** loop will execute $|\mathsf{L}|$ times, and when it terminates either (a) termination condition is satisfied, or (b) all the matches are found, from which top-ranked ones can be identified. If condition (a) happens, S must contain k best matches, which is warranted by Prop. 2. In case condition (b) occurs, it can be easily verified that all the candidate matches are processed and the entire set of matches of Q_r are generated. Thus, S must be the top-k match set after selection.

Complexity. The computational cost of TopKR consists of two parts, *i.e.,* initialization and match identification. (1) The cost of initialization also includes two parts, *i.e.,* twisted graph construction and spanning tree generation. For the first part, observe that it takes RegReach (a) $O(|R|\mathsf{log}|R|)$ time for query automaton construction, and (b) $O(|V|(|V| + |E|))$ time for evaluating a regular reachability query. Thus, *twisted graph* construction is in $O(|V|^2(|V| + |E|) + |E_p||R|\mathsf{log}|R|)$ time, since at most $|E_p|$

regular expressions need to be transformed to query automatons, and the depth first traversal will be repeated at most $|V|$ times. Taking cost of spanning tree construction into account, the initialization takes TopK $O(|Q_r| + |V|^2(|V| + |E|) + |E_p||R|\log|R|)$ time. (2) Match identification is conducted by TMat, which is in $O(|V_q|!|V_q| + |V_q|^{|V_p|}\log(|V_q|^{|V_p|}))$ time (V_q refers to the node set of the *twisted graph* G_q). Specifically, the candidates generation is in $O(|V_q|!|V_q|)$ time in the worst case (lines 3-5), and the sorting for candidate matches (line 7) is in $O(|V_q|^{|V_p|}\log(|V_q|^{|V_p|}))$ time, since there may exist at most $|V_q|^{|V_p|}$ matches in L (line 6). The match verification is in $O(|V_q|^{|V_p|} + |V_q|^{|V_p|}\log(|V_q|^{|V_p|}))$ time (lines 7-11). To see this, observe that (a) match identification takes $O(|V_q|^{|V_p|})$ time, as the **while** loop (line 7) runs at most $|V_q|^{|V_p|}$ times, and each loop takes TMat constant time to generate a match t (line 8) and verify termination condition (line 10); and (b) top-k match selection (line 12) takes $O(|V_q|^{|V_p|}\log(|V_q|^{|V_p|}))$ time. (3) Putting initialization and match identification together, TopKR is in $O(|Q_r| + |V|^2(|V| + |E|) + |E_p||R|\log|R| + |G|^{|Q_r|} \cdot \log(|G|^{|Q_r|}) + |G|!|G|)$ time, which is bounded by $O(|G|^{|Q_r|} \cdot \log(|G|^{|Q_r|}) + |G|!|G|)$, since parameters $|Q_r|$, $|V|^2(|V| + |E|)$ and $|E_p||R|\log|R|$ can be omitted from the perspective of worst-case time complexity.

Early termination. Algorithm TopKR has the *early termination* property, since it combines the match identification and ranking as a whole, and terminates as soon as top-k matches are found, without computing and sorting the entire match set. As will be seen in experimental study, while TopKR has the same worst-case complexity as $\mathsf{Naive_r}$, it substantially outperforms $\mathsf{Naive_r}$.

The analysis above completes the proof of Theorem 1(2). □

4 Experimental Evaluation

Using real-life and synthetic data, we conducted two sets of experiments to evaluate performance of our algorithms.

Experimental setting. We used the following datasets.

(1) Real-life graphs. We used two real-life graphs: (a) *Amazon* [19], a product co-purchasing network with $548K$ nodes and $1.78M$ edges. Each node has attributes such as title, group and sales-rank, and an edge from product x to y indicates that people who buy x also buy y. (b) *Youtube* [5], a recommendation network with $1.6M$ nodes and $4.5M$ edges. Each node is a video with attributes such as category, age and rate, and each edge from x to y indicates that y is in the related list of x.

(2) Synthetic data. We designed a generator to produce synthetic graphs $G = (V, E, L_v)$, controlled by the number of nodes $|V|$ and edges $|E|$, where L_v are assigned from a set of 15 labels. We generated synthetic graphs following the linkage generation models [14]: an edge was attached to the high degree nodes with higher probability. We use $(|V|, |E|)$ to denote the size of G.

(3) Pattern generator. We designed a generator to produce meaningful patterns. The generator uses following parameters: $|V_p|$ for the number of pattern nodes, $|E_p|$ for the number of pattern edges, label f_v from an alphabet Σ of labels taken from corresponding data graphs, and bounds b and c for $f_e(e_p)$ of *r-patterns*, such that one edge e_p is constrained by a regular expression $l_1^{\leq b}, \cdots, l_k^{\leq b}$ $(1 \leq k \leq c)$ and has weight $\Sigma_{i\in[1,k]}b$. For synthetic graphs G, we manually constructed a set of 10 patterns with node labels drawn from the same Σ of G. We refer to $d_w(e)$ as the maximum edge weight of a pattern Q_r and denote $(|V_p|, |E_p|, d_w(e))$ as the size $|Q_r|$ of Q_r. We use $(|V_p|, |E_p|, d_w(e))$ and $(|V_p|, |E_p|)$ to indicate the size of Q_r interchangeably, when it is clear from the context.

(4) Implementation. We implemented the following algorithms, all in Java. (a) TopKR, for identifying top-k matches of Q_r. (b) Naive$_r$, a naive algorithm for TopK, that performs regular reachability queries with techniques of [11] and generates a TG, identifies matches from the TG with revised VF2 [6], and selects top-k matches. (c) TopK$_u$, for computing top-k matches for a normal pattern Q. Observe that all the matches of a normal pattern Q have the same relevance value. Hence, TopK$_u$ applies different strategies to generate twisted graphs and identify top-k matches. More specifically, in the initialization stage, TopK$_u$ generates a TG *w.r.t.* a normal pattern Q by including those edges in G that have endpoints as matches of endpoints of a pattern edge e in Q. During matching computation, TopK$_u$ simply generates a list L of candidate matches of Q, and repeatedly verifies whether each of them can turn to a valid match, until k matches are found. Here, no ranking for entries in L and complex verification for termination (Prop 2) are needed. (d) Naive, a naive top-k algorithm for normal patterns, that constructs twisted graphs first along the same line as TopK$_u$, finds the complete set of matches using VF2 [6], ranks and selects top-k matches from the match set.

All the experiments were run on a machine with an Intel Core(TM)2 Duo 3.7GHz CPU, and 16GB of memory, using Ubuntu. Each experiment was repeated 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness. We studied the effectiveness of the algorithms TopK vs. Naive$_r$ (resp. TopK$_u$ vs. Naive) on real-life graphs. To measure effectiveness of our *early termination* algorithms, we define a ratio IR as $\frac{|M^i(Q_r,G)|}{|M(Q_r,G)|}$, where $|M^i(Q_r,G)|$ indicates the amount of matches identified by the algorithm when it terminates. We do not report the effectiveness of Naive$_r$ and Naive, as their IR are always 100%.

Varying $|Q_r|$. Fixing k=10 and $d_w(e) = 3$ (resp. $d_w(e) = 4$) for all e in Q_r, we varied $|Q_r|$ from $(3, 3)$ to $(7, 14)$, and evaluated IR over two real-life graphs. The results that are reported in Table 2, tell us the following: (1) TopKR is able to reduce excessive matches. For example, when $|Q_r| = (3, 3)$ and $d_w(e) = 3$, TopKR only inspected 38% (resp. 46%) matches when top-k matches are identified, on *Youtube* (resp. *Amazon*). On average, IR for TopKR is only 66.8% (resp. 64.6%) on *Youtube* (resp. *Amazon*), which indicates that TopKR benefits from its *early termination property*. (2) the effectiveness of *early terminate property* degrades when $|Q_r|$ gets larger. For example, when $|Q_r| = (7, 14)$, IR $= 100\%$ on both graphs, indicating that TopKR can not terminate earlier. The main reason is that: for larger patterns, the $f_d(\cdot)$ value grows bigger for seen matches, hence termination condition can not be easily satisfied.

Under the same setting, we evaluated IR for normal patterns and found that TopK$_u$ effectively reduces excessive matches. For example, when $|Q| = (3, 3)$, TopK$_u$ only inspected 9% (resp. 11%) matches when top-k matches are identified, on *Youtube* (resp. *Amazon*). On average, IR for TopK$_u$ is only 36.2% (resp. 32.2%) on *Youtube* (resp. *Amazon*), which indicates that TopK$_u$ benefits from its early termination property.

		Youtube		*Amazon*	
		$d_w(e) = 3$	$d_w(e) = 4$	$d_w(e) = 3$	$d_w(e) = 4$
$\|Q_r\|$	*(3,3)*	38%	36%	46%	44%
	(4,6)	46%	37%	50%	48%
	(5,10)	68%	65%	60%	50%
	(6,12)	90%	88%	76%	72%
	(7,14)	100%	100%	100%	100%

Table 2. Varying $|Q_r|$ and $d_w(e)$ (k=10)

Varying k. Fixing pattern size $|Q_r| = (4, 6)$ and $d_w(e) = 3$ (resp. $d_w(e) = 4$) , we varied k from 5 to 30 in 5 increments, and reported IR for TopKR on real-life graphs. As shown in Table 3, for *regular patterns* with $d_w(e) = 3$ (resp. $d_w(e) = 4$), the ratio IR of TopKR increased from 46% to 93% (resp. 42% to 89%) and 37% to 100% (resp. 34% to 98%), on *Youtube* and *Amazon*, respectively, when k increased from 5 to 30. In general, with the increase of k, IR gets larger either, as expected. While unlike the trend observed when varying $|Q_r|$, when k changes from 5 to 15, IR keeps unchanged for all cases. This is because, a fixed number of matches of the spanning tree of Q_r (as match candidates) needs to be examined before TopKR can terminate.

In the same setting as test for Q_r, we evaluated IR for normal patterns. We find that the ratio IR of $\mathsf{TopK_u}$ increased from 9% to 56% and 9% to 53%, on *Youtube* and *Amazon*, respectively, when k increased from 5 to 30. The increment of IR is mainly caused by that, more matches have to be examined for choosing as top-k matches, for a larger k, which is as expected.

		Youtube		*Amazon*	
		$f_e(e) = 3$	$f_e(e) = 4$	$f_e(e) = 3$	$f_e(e) = 4$
k	*5*	46%	42%	37%	34%
	10	46%	42%	37%	34%
	15	46%	42%	37%	34%
	20	46%	42%	62%	64%
	25	68%	66%	62%	64%
	30	93%	89%	100%	98%

Table 3. Varying k and $d_w(e)$ ($|Q_r|$=(4,6))

Varying $d_w(e)$. From Tables 2 and 3, we can also see the influence of the changes of $d_w(e)$. That is when $d_w(e)$ gets larger, IR drops with slight decrease as well. This shows that the *early termination property* favors patterns with large $d_w(e)$.

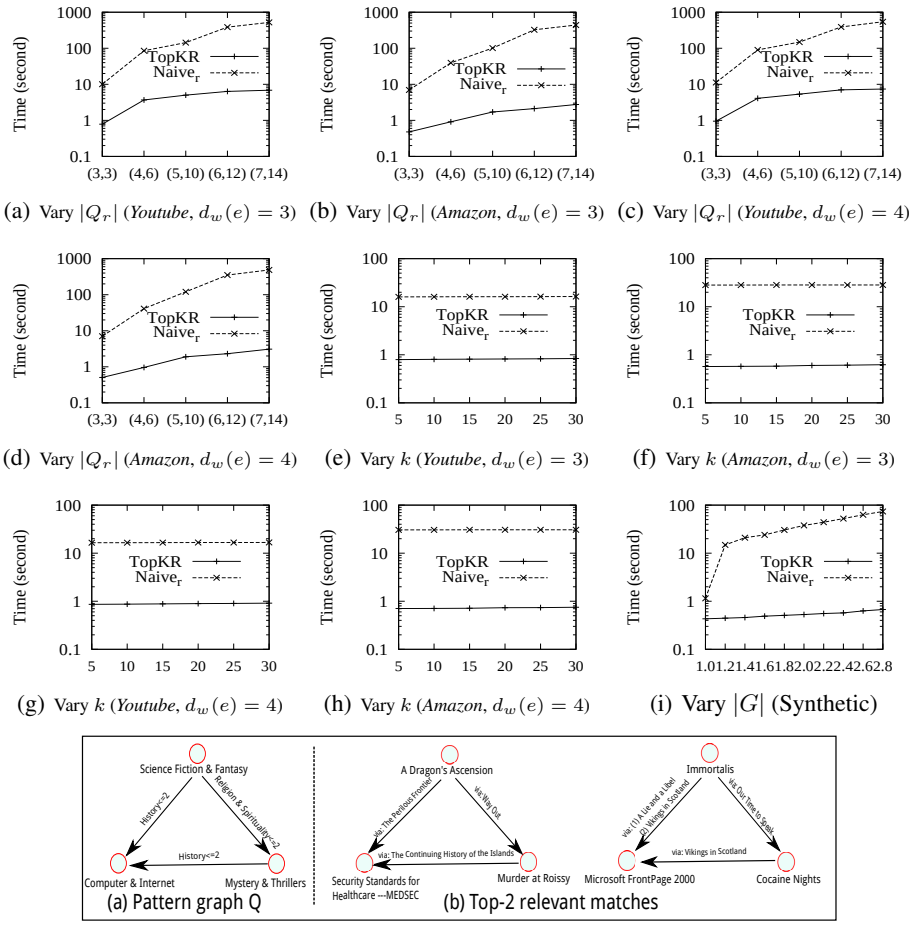
Exp-2 Efficiency. We next evaluated the efficiency of the algorithm TopKR vs. $\mathsf{Naive_r}$ (resp. $\mathsf{TopK_u}$ vs. Naive) in the same setting as in Exp-1.

Varying $|Q_r|$. As shown in Figures 5(a)- 5(d), (1) TopKR substantially outperforms $\mathsf{Naive_r}$: it takes TopKR, on average, 3.71% (resp. 3.94%) and 2.42% (resp. 2.48%) of the time of $\mathsf{Naive_r}$, on *Youtube* and *Amazon*, respectively, for patterns with $d_w(e) = 3$ (resp. $d_w(e) = 4$). The performance advantage is mainly caused by (a) a much more efficient strategy, applied by TopKR, to identify matches of spanning trees of Q_r, and (b) *early termination property* TopKR possesses; and (2) $\mathsf{Naive_r}$ is more sensitive to $|Q_r|$ than TopKR, since the cost of match set computation, which accounts for more than 95% of entire running time of $\mathsf{Naive_r}$, heavily depends on Q.

Under the same setting, we evaluated efficiency for normal patterns. The results tell us that (1) $\mathsf{TopK_u}$ substantially outperforms Naive. For example, $\mathsf{TopK_u}$ takes, on average, 0.68% and 1.04% of the time of Naive, on *Youtube* and *Amazon*, respectively.

Varying k. Figures 5(e)- 5(h) report the efficiency results for patterns with size $|Q_r| = (4, 6)$ and $d_w(e) = 3$ (resp. $d_w(e) = 4$). We find the following: (1) TopKR and $\mathsf{Naive_r}$ are both insensitive to k, since most of the computational time is spent for twisted graph construction and matching evaluation. (2) TopKR always outperforms $\mathsf{Naive_r}$, which is consistent with the observations in Figures 5(a)- 5(d).

In the same setting, we conducted efficiency test for normal patterns and found that $\mathsf{TopK_u}$ always outperforms Naive. For example, $\mathsf{TopK_u}$ only takes, on average, 0.42% (resp. 1.45%) time of Naive, on *Youtube* and *Amazon*, respectively.



(a) Vary $|Q_r|$ (*Youtube*, $d_w(e) = 3$) (b) Vary $|Q_r|$ (*Amazon*, $d_w(e) = 3$) (c) Vary $|Q_r|$ (*Youtube*, $d_w(e) = 4$)

(d) Vary $|Q_r|$ (*Amazon*, $d_w(e) = 4$) (e) Vary k (*Youtube*, $d_w(e) = 3$) (f) Vary k (*Amazon*, $d_w(e) = 3$)

(g) Vary k (*Youtube*, $d_w(e) = 4$) (h) Vary k (*Amazon*, $d_w(e) = 4$) (i) Vary $|G|$ (Synthetic)

(j) Case study on *Amazon*

Fig. 5. Performance evaluation

Varying $d_w(e)$. Figures 5(a)- 5(h) also show the influence on the efficiency with respect to the changes of $d_w(e)$. We find that it takes longer for TopKR on patterns with larger $d_w(e)$, since patterns with larger edge weight may result in a bigger match set, thus more time is required to verify matches, and select top-k ones from them. For example, TopKR spends 109% and 122% of the time when $d_w(e)$ of Q_r increased from 3 to 4. on *Youtube* and *Amazon*, respectively.

Exp-3 Scalability. Fixing $|Q_r| = (4, 6)$ and $d_w(e) = 2$, we varied $|G|$ from $(1M, 2M)$ to $(2.8M, 5.6M)$ in $(0.2M, 0.4M)$ increments, and evaluated the scalability of the algorithms with synthetic data. The results, shown in Fig. 5(i), tell us that TopKR is less sensitive to $|G|$ than Naive$_r$, hence scale better with $|G|$ than Naive$_r$, and moreover, both algorithms are not sensitive to $d_w(e)$.

Exp-4 Case study. On *Amazon*, we manually inspected top-2 matches found by TopKR for Q_r of Fig. 5(j)(a). As shown in Fig. 5(j)(b), two matches found by TopKR are quite relevant to Q_r; while they have different relevance values, since the match on the right hand side has a longer path, which starts from Immortalis to Microsoft FrontPage 2000 via two history nodes, *i.e.,* "A Lie and a Libel", and "Vikings in Scotland".

Summary. We summarize our findings as below. Our top-k matching algorithms work well. (a) They are effective: top-10 matches of patterns with size $(3, 3, 3)$ can be found by TopKR when only 38% (resp. 46%) matches on *Youtube* (resp. *Amazon*) are identified. (b) They are efficient: it only takes TopKR 0.782s (resp. 0.477s) to find top-10 matches of Q_r with $|Q_r| = (3, 3, 3)$ over *Youtube* (resp. *Amazon*). (c) TopKR scales well with $|G|$.

5 Conclusion

We have introduced the *top-k graph pattern matching problem*. We have revised the semantic of graph pattern matching by allowing edge to regular path mapping, and defined functions to measure match relevance. We have established the complexity for TopK, and provided an algorithm with *early termination property* for TopK. As verified analytically and experimentally, our method remedies the limitations of prior algorithms, via semantic extension, and substantially improves efficiency on big social graphs.

References

1. Full version. *https://github.com/xgnaw/sun/raw/master/regularGPM.pdf*.
2. G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA*, pages 261–272, 2013.
3. J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.
4. J. Cheng, X. Zeng, and J. X. Yu. Top-k graph pattern matching over large graphs. In *ICDE*, pages 1033–1044, 2013.
5. X. Cheng, C. Dale, and J. Liu. Youtube. *http://netsg.cs.sfu.ca/youtubedata/*, 2008.
6. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.
7. R. Fagin. Combining fuzzy information from multiple systems. *JCSS*, 58(1):83–99, 1999.
8. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.
9. W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.
10. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1), 2010.
11. W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *PVLDB*, 5(11):1304–1315, 2012.
12. W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13):1510–1521, 2013.
13. G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT, Bordeaux, France, Bordeaux, France, March 15-16*, pages 636–639, 2016.
14. S. Garg, T. Gupta, N. Carlsson, and A. Mahanti. Evolution of an online social aggregation network: an empirical study. In *IMC*, 2009.
15. G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.
16. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
17. J. Hromkovic, S. Seibert, and T. Wilke. Translating regular expressions into small -free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.
18. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

19. J. Leskovec and A. Krevl. Amazon dataset. *http://snap. stanford.edu/ data/ index.html*, June 2014.
20. A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, pages 185–193, 1989.
21. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
22. J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. *Theory Comput. Syst.*, 61(1):31–83, 2017.
23. L. G. Terveen and D. W. McDonald. Social matching: A framework and research agenda. In *ACM Trans. Comput.-Hum. Interact.*, 2005.
24. H. Wang, J. Han, B. Shao, and J. Li. Regular expression matching on billion-nodes graphs. *CoRR*, abs/1904.11653, 2019.
25. X. Wang and H. Zhan. Approximating diversified top-k graph pattern matching. In *DEXA*, 2018.
26. Wikipedia. Prim's algorithm. *https://en.wikipedia.org/wiki/Prim's_algorithm*, 2019.
27. L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *Ph.D. workshop in CIKM*, 2007.