Efficient Regular Simple Path Queries under Transitive Restricted Expressions

Qi Liang
Guangzhou University
qiliang@e.gzhu.edu.cn

Dian Ouyang
Guangzhou University
dian.ouyang@gzhu.edu.cn

Fan Zhang
Guangzhou University
zhangf@gzhu.edu.cn

Jianye Yang
Guangzhou University
jyyang@gzhu.edu.cn

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@sjtu.edu.cn

Zhihong Tian
Guangzhou University
tianzhihong@gzhu.edu.cn

ABSTRACT

There are two fundamental problems in regular simple path queries (RSPQs). One is the reachability problem which asks whether there exists a simple path between the source and the target vertex matching the given regular expression, and the other is the enumeration problem which aims to find all the matched simple paths. As an important computing component of graph databases, RSPQs are supported in many graph database query languages such as PGQL and openCypher. However, answering RSPQs is known to be NP-hard, making it challenging to design scalable solutions to support a wide range of expressions. In this paper, we first introduce the class of *transitive restricted expression*, which covers more than 99% of real-world queries. Then, we propose an efficient algorithm framework to support both reachability and enumeration problems under transitive restricted expression constraints. To boost the performance, we develop novel techniques for reachability detection, the search of candidate vertices, and the reduction of redundant path computation. Extensive experiments demonstrate that our exact method can achieve comparable efficiency to the state-of-the-art approximate approach, and outperforms the state-of-the-art exact methods by up to 2 orders of magnitude.

KEYWORDS

Regular expression, Simple path, Reachability, Enumeration

PVLDB Reference Format:
Qi Liang, Dian Ouyang, Fan Zhang, Jianye Yang, Xuemin Lin, and Zhihong Tian. Efficient Regular Simple Path Queries under Transitive Restricted Expressions. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Newth-QiLiang/Regular-Simple-Path-Queries.

1 INTRODUCTION

Graphs are often used to model entities and their connections in complex systems, such as social networks, the World Wide Web, and computer networks. In many real applications, each edge between two vertices is associated with a label to represent a specific relation. In recent years, *regular path queries* have been extensively studied on edge-label graphs [1, 3, 10, 30]. There are two fundamental problems: (1) the *reachability query* asks whether there exists a path between a given pair of source and target vertices satisfying the regular expression [18, 31, 41], and (2) the *enumeration query* aims to find all paths between two given vertices that satisfy a given regular expression [11, 25, 26]. In the literature, regular path queries are usually studied under some possible semantics, including arbitrary path, shortest path, and simple path [26]. To avoid overwhelming and redundant results, in this paper, we focus on regular path queries under simple path semantics, named *Regular Simple Path Queries (RSPQs)*, which requires that there are no repeated vertices along a resulting path.

Applications. RSPQs can be applied in many real scenarios. Some examples are the following.
(1) Knowledge Retrieval in Knowledge Graphs. RSPQs are building blocks in many graph query languages such as PGQL [40] and openCypher[1] [14, 33]. In knowledge graphs or graph databases, many information retrieval tasks can be solved by RSPQs. For example, if we want to know whether a user U lives in New York City in a knowledge graph, we should check whether there exists a simple path from user U to New York City. In addition, the label of the path should be limited since we only consider the relationship of living rather than other relationships. We can use the Reachability Query $(U, NewYork, (\mathit{LivedIn} \circ (PartOf)^*)$ to answer the above question where $\mathit{LivedIn}$ records the place of residence, and $PartOf$ denotes the containment relationship of territorial entities. We use the regular expression $(PartOf)^*$ to expand the place of residence to the city level, because the residence may be recorded as a street rather than a city. If we can find a simple path satisfying the expression such as $U \xrightarrow{LiveIn}$ Grand Street $\xrightarrow{PartOf}$ New York, then we know U lives in New York City.
(2) Cyber-attack Detection in Computer Network Traffic. In the cybersecurity community, advanced persistent threat (APT) detection has been one of the most important tasks. Recent advances in APT detection tend to utilize the provenance graph [2, 16, 28], where vertices are network entities, and edges represent activities. An APT attack usually consists of a sequence of network activities, which can be easily detected by a pattern-matching method. However, in the real world, the attackers may add noise to their activities to

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] The paths supported in openCypher are restricted to simple paths (as described in Section 2.3 of [33]), Similarly, PGQL also supports regular simple path queries.

Table 1: Structure of property paths from Table 4 in [8]. A **means** $(a_1 + a_2 \cdots + a_k)$.

Name	Relative	LCR?	TRE?	Name	Relative	LCR?	TRE?
a^*	50.48%	✓	✓	A^*	0.60%	✓	✓
$a_1 \circ a_2 \cdots \circ a_k$	24.26%	×	✓	$a \circ b^* \circ c$	0.22%	×	✓
$a \circ b^*$	17.07%	×	✓	$a^* \circ b^*$	0.11%	×	✓
A	5.52%	×	✓	$\epsilon \mid A$	0.06%	×	✓
$a \circ b^* \circ c^*$	1.49%	×	✓	$a \circ b \circ c^*$	0.05%	×	✓

escape detection [28]. As described in [32], in the provenance graph, a compromised browser writing to a system file may correspond to a path where a vertex representing a Firefox process forks new processes, only one of which ultimately writes to the system file. By modeling such paths as RSPQs, e.g., $(Browser, File, ((Fork)^* \circ Write))$, we are able to hunt suspicious APTs.

(3) Pathway Analysis in Biological Networks. Pathway queries are essential in the analysis of biological networks, where vertices represent entities such as compounds and edges represent various forms of interactions [12, 18, 23]. For instance, in [18], a network analyzer can quickly determine if two compounds have a given pathway, which refers to specific forms of interactions, using reachability queries. Additionally, simple path enumeration is a common type of pathway query in biological networks [14], which provides detailed information about the paths between entities.

Limitations of Existing Studies. Although RSPQs have been extensively studied [1, 5, 9, 10, 26, 27, 41], there are several major limitations in existing solutions to RSPQs.

(1) Only support a small subset of regular expressions. Most existing works [31, 39, 44] only focus on label-constraint reachability (LCR) queries. Given a set of labels $L = \{a_1, \cdots, a_k\}$, an LCR query requires the label of each edge on the resulting path must belong to L. This type of label constraint could be expressed to the regular expression $(a_1 + \cdots + a_k)^*$. A recent analysis of SPARQL query logs [8] suggests that $\approx 48\%$ of the RSPQs cannot be expressed by LCR .

(2) Unable to balance the efficiency and accuracy. The reachability query of RSPQs is NP-hard under arbitrary expressions [27]. Therefore, exact methods that support all regular expressions, such as BBFS [41], are computationally expensive, and even unacceptable in some extreme cases. Wadhwa et al. [41] propose ARRIVAL, an approximate method to solve this problem, which is much faster than BBFS. However, its accuracy relies heavily on the graph structure, which limits its applicability.

(3) No efficient method for enumeration problem of RSPQs. All the above works only study the reachability problem which is easier than the enumeration problem in nature. To the best of our knowledge, recent research [26] has proposed a polynomial delay algorithm (nearly $O(n^3)$ delay, where n means the number of vertices in the graph) for the enumeration problem of RSPQs, which only supports a subset of regular expressions called downward closed expressions. However, it is still not efficient enough and does not support more complex expressions.

Challenges. Because the reachability query of RSPQs is a well-known NP-hard problem, it is impossible to design an exact method to solve RSPQs efficiently under arbitrary expressions unless P = NP. Additionally, even though index structures can greatly enhance computational efficiency, it becomes impractical to construct a suitable index as the diversity of expression types increases. Moreover, constructing an index incurs huge time and space costs, which are unacceptable for large graphs with a large number of labels. Therefore, on the one hand, approximate and index-free algorithms for reachability queries is a feasible solution, which has been studied in [41]. On the other hand, accurate answers hold better significance in various real scenarios, motivating us to design exact methods rather than approximate ones.

The above limitations and challenges motivate us to ask the following questions. Is there a specific type of regular expression that can cover the most frequently encountered expressions in real-world scenarios, and can we develop efficient index-free algorithms to address the above two problems for such special regular expressions? In this paper, we aim to propose an efficient, exact, and index-free approach to solve the above two issues of RSPQs.

Contributions. We observe that there are two basic categories of regular expressions for which it is possible to devise efficient and exact algorithms. Based on this observation, we define a regular expression framework, called *transitive restricted expression* (TRE), which is able to cover more than 99% of regular expressions encountered in real-world scenarios from two studies [7] and [8]. Specifically, TRE could cover all the expression types (more than 99%) in Table 1 while LCR only supports two types of them (around 51.08%). Similar results can be found in [7].

To efficiently handle the TRE, we propose efficient and effective query techniques. In specific, we first divide a regular expression into three parts Pre, $Type$, and Suf. Then, we propose an approach for the two issues of RSPQs by dealing with the three parts in two phases. In the first phase, we develop an algorithm to find all the simple paths that match Pre and Suf respectively. In the second phase, we process $Type$ in addition to the output of the first phase. By combining the results of the above two phases together, we obtain the final answers. More specifically, we analyze the most time-consuming part (i.e., $Type$) and design an efficient algorithm for reachability queries under $Type$ and a polynomial time-delay algorithm to tackle the enumeration problem for $Type$.

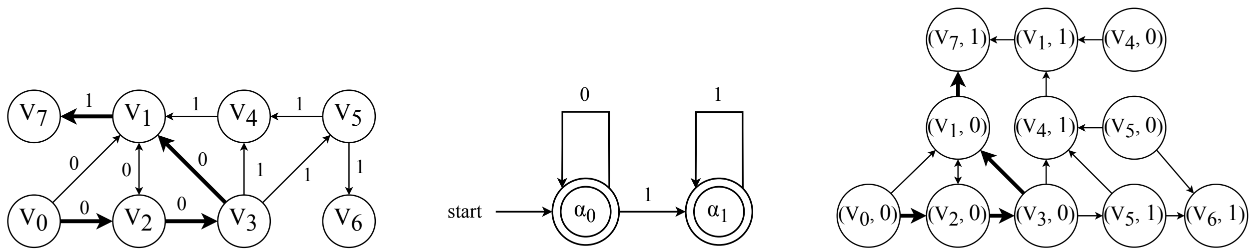
Our principal contributions are the following.

- We propose a new type of regular expressions named TRE consisting of three parts Pre, $Type$, and Suf which can cover 99% of expressions in real-world queries.
- Based on TRE constraints, we develop an index-free and exact framework, which can solve both reachability and enumeration queries efficiently.
- To further improve the performance, we develop novel techniques for reachability detection, the search of candidate vertices, and the reduction of redundant path computation.
- Our empirical study shows that our approach can return exact solutions while achieving comparable efficiency to the state-of-the-art approximate method. It is also demonstrated that our approach outperforms the state-of-the-art exact approaches by up to two orders of magnitude in terms of efficiency.

2 PRELIMINARIES

In this section, we first formally introduce the definition of directed labeled graphs. Then we give the problem statement of regular simple path reachability and enumeration query.

A directed labeled graph is denoted as $G = (V, E, \mathcal{L}, \phi)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges, $\mathcal{L}$ is a



(a) Edge-labeled graph G (b) DFA A (c) Product graph $P_{G,A}$

Figure 1: 1(a) A edge-labeled graph G, 1(b) the automation for expression $0^* \circ 1^*$, 1(c) the product graph $P_{G,A}$ based on G and A.

finite non-empty set of labels, and $\phi : E \rightarrow \mathcal{L}$ is a function that maps every edge to a label. Note that multiple edges between two vertices must have distinct labels.

Given two vertices $s, t \in V$, a path p from s to t in G is a sequence of edges : $\langle (v_0, l_0, v_1), \cdots, (v_{n-1}, l_{n-1}, v_n) \rangle$ where $v_0 = s$, $v_n = t$ and $l_i \in \mathcal{L}, 0 \leq i \leq n-1$. If there is no repeating vertex in path p, we say p is a simple path. The label of a path p is denoted by $\phi(p) = l_0 l_1 \cdots l_{n-1} \in \mathcal{L}^*$.

Definition 1 (Regular Expression). *A regular expression R over the alphabet $\mathcal{L}$ is defined as $R ::= \epsilon \mid l \mid R_1 \circ R_2 \mid R_1 + R_2 \mid R^*$, where (i) ϵ denotes the empty string, (ii) $l \in \mathcal{L}$ denotes a character in the alphabet, (iii) $\circ$ denotes the concatenation operator, (iv) $+$ denotes the alternation operator, (v) R, R_1 and R_2 are regular expressions, and (vi) $*$ represents the Kleene star. A regular language $L(R)$ is the set of strings that can be described by regular expression R. We say that a path p matches a regular expression R if the label of p could be described by R, i.e., $\phi(p) \in L(R)$.*

Example 1. *In Figure 1(a), considering the regular expression $R = 0^* \circ 1^*$, the path $V_0 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{0} V_1 \xrightarrow{1} V_7$ (indicated by the bold edges) corresponds to a match for the expression R because the label of path p, which is 0001, can be described by $0^* \circ 1^*$.*

Next, we introduce a type of restricted regular expression called *transitive restricted expression.* TRE is a combination of simple linked expression (LE) and downward closed expression (DCE). Hence, before introducing TRE, we first give the definition of LE and DCE.

Definition 2 (Length-fixed Expression). *A length-fixed expression S is the expression that does not contain the Kleene star, i.e., $R ::= \epsilon \mid l \mid R_1 \circ R_2 \mid R_1 + R_2$, where R_1 and R_2 are length-fixed expressions.*

Definition 3 (Downward Closed Expression [26]). *An expression R is called downward closed if, for any sequence $L = l_1 l_2 \cdots l_n$ that can be described by the expressions R, any subsequence $l_{i_1} \cdots l_{i_k}$, $0 < i_1 < \cdots < i_k < n+1$ can also be described by R.*

Based on this definition, we know that any sequence that can be derived from L by deleting some or no elements without changing the order of the remaining elements can be also described by the R. Note that even ϵ (i.e., the empty string) can be also described by R. Then, we give the following remark.

Remark 1. *$R = (l_1 + l_2 + \cdots + l_k)^*$, where $l_i \in \mathcal{L}, k \in \mathbb{N}$, represents a fundamental type of downward closed expressions. Moreover, the expression $R = R_0 \circ R_1 \circ \cdots \circ R_k, k \in \mathbb{N}$ is also downward closed if $\forall i, 0 \leq i \leq k, R_i$ is downward closed.*

Proof. Obviously, $(l_1 + l_2 + \cdots + l_k)^*$ is a downward closed expressions. Then, we give a proof sketch to show $R_0 \circ R_1$ is also a downward closed expressions. This result can expand to k based on mathematical induction. Assume that there are two strings (s_1 and s_2) that can be accepted by R_1 and R_2 respectively. For the concatenation of s_1 and s_2 (named s), all subsequences that satisfy the corresponding definition in the definition of downward closed expression could be seen as two parts. Moreover, the first subsequence must be accepted by R_1, and the second subsequence must be accepted by R_2. Then the subsequences must be accepted by $R_0 \circ R_1$. Hence, $R_0 \circ R_1$ is a downward closed expression. □

Definition 4 (Transitive Restricted Expression). *We define the transitive restricted expression as $Pre \circ Type \circ Suf$, where Pre and Suf are length-fixed expressions, and $Type$ is a downward closed expression.*

With the formalization of the above concepts, now we establish the definition of regular simple path reachability query and enumeration query respectively.

Definition 5 (Regular Simple Path Reachability Query). *Given a graph G, a transitive restricted expression R, the source vertex s and the target vertex t, the regular simple path reachability query (Reachability Query) returns true if there exists at least one simple path between s and t that matches the expression R.*

Definition 6 (Regular Simple Path Enumeration Query). *Given a graph G, a transitive restricted expression R, the source vertex s and the target vertex t, the regular simple path enumeration query (Enumeration Query) returns all simple paths $P = \{p \mid p$ is a simple path and $\phi(p) \in L(R)\}$ between s and t.*

Example 2. *In Figure 1(a), assuming the source vertex is V_0, the target vertex is V_6 and the transitive restricted expression R is $0^* \circ 1^*$. The Reachability Query will return true while the Enumeration Query will give two simple paths that matched the expression R as $V_0 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{1} V_5 \xrightarrow{1} V_6$ and $V_0 \xrightarrow{0} V_1 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{1} V_5 \xrightarrow{1} V_6$.*

3 EXISTING SOLUTION

Given Reachability Query and Enumeration Query in a directed labeled graph G, we use different algorithms for these two queries generally. For both of them, we usually use deterministic finite

Table 2: Frequently used notations.

Notations	Definitions
$V_1 \xrightarrow{l} V_2$	an edge (V_1, V_2) with label l, i.e., (V_1, l, V_2)
$A.\alpha_0$	the start state of DFA A
$A.F$	the final state set of DFA A
$N^P_{out}(u, \alpha)$	the out-neighbors of vertex (u, α)
$N^P_{in}(v, \beta)$	the in-neighbors of vertex (v, β)
$P_{G,A}$	the product graph $P_{G,A}$

automaton and product graphs to solve these problems. Therefore, we provide the definition of deterministic finite automaton and product graph. We introduce a basic DFS algorithm using a product graph. This algorithm is a part of our optimal solutions. Then, we introduce Bidirectional BFS and DFS-based algorithms in the product graph for these two kinds of queries respectively.

Definition 7 (Deterministic Finite Automaton). *Given a regular expression R, its corresponding deterministic finite automaton (DFA) is defined as $A = (S, \mathcal{L}, \delta, \alpha_0, F)$, where S is a set of states, $\mathcal{L}$ is the input alphabet, $\delta : S \times \mathcal{L} \to S$ is the state transition function, $\alpha_0 \in S$ is the start state of DFA, and $F \subseteq S$ is the set of final states of DFA. δ^* is the extended transition function defined as $\delta^*(\alpha, w \circ l) = \delta(\delta^*(\alpha, w), l)$, where $\alpha \in S, l \in \mathcal{L}, w \in \mathcal{L}^*, \delta^*(\alpha, \epsilon) = \alpha$, and ϵ is the empty string. We call a string w can be accepted by A if $\delta^*(\alpha_0, w) = \alpha_f, \alpha_f \in F$. Note that if w can be described by R, it must be accepted by the corresponding DFA A.*

For a regular expression R in the query, we construct a corresponding DFA[2]. Any path in DFA from α_0 to F matches R.

Definition 8 (Product Graph). *Given a graph $G = (V, E, \mathcal{L}, \phi)$ and a DFA $A = (S, \mathcal{L}, \delta, \alpha_0, F)$, the corresponding product graph $P_{G,A}$ is defined as $P_{G,A} = (V_P, E_P)$, where $V_P = V \times S$, $E_P \subseteq V_P \times V_P$, $((u, \alpha), (v, \beta)) \in E_P$ iff $(u, v) \in E$ and $\delta(\alpha, \phi(u, v)) = \beta$. $N^P_{out}(u, \alpha) = \{(v, \beta) \mid ((u, \alpha), (v, \beta)) \in E_P\}$ denotes the out-neighbors of vertex (u, α) and $N^P_{in}(v, \beta) = \{(u, \alpha) \mid ((u, \alpha), (v, \beta)) \in E_P\}$ denotes the in-neighbors of vertex (v, β).*

Example 3. *Figure 1(b) shows the DFA A of regex $0^* \circ 1^*$, where the double circle of state means it is one of the final states, and Figure 1(c) shows the product graph $P_{G,A}$, where we use numbers instead of symbols (e.g., 0 means α_0) for simplicity. We assign the start state (i.e., 0) to vertex V_0 and start traversing the graph. When we cross the edge $V_0 \xrightarrow{0} V_2$, the state 0 will transfer to state 0 according to the transition function, so we arrive at vertex c with state 0. In this way, we transform the edge $V_0 \xrightarrow{0} V_2$ into the edge $(V_0, 0) \to (V_2, 0)$. Similarly, the path $V_0 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{0} V_1 \xrightarrow{1} V_7$ is expressed as the path $(V_0, 0) \to (V_2, 0) \to (V_3, 0) \to (V_1, 0) \to (V_7, 1)$.*

3.1 DFS based algorithm

For a path p in G, assume we want to check if p matches R. We have to check whether the state can reach F in A which is the corresponding DFA of R. $P_{G,A}$ combines the information of G and A. Note that every path in G has its unique corresponding path in $P_{G,A}$, and vice versa. Meanwhile, for any path p' in $P_{G,A}$, A has its unique corresponding path. Hence, the intuitive idea to find a path matching R is finding the corresponding path in $P_{G,A}$.

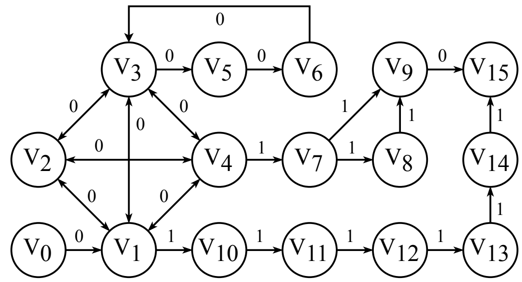


Figure 2: A bad-case example for RSPQs.

We present a fundamental DFS algorithm to handle both Reachability Query and Enumeration Query. For a given graph G, regex R, and start vertex s, we construct the corresponding DFA A and product graph $P_{G,A}$. The DFS algorithm starts from the vertex $v_s = (s, A.\alpha_0) \in V_P$. v_s presents a vertex in $s \in V$ and the state $\alpha_0 \in S$. Then we search from v_s. If there exists an eligible simple path from source vertex s with start state to target vertex t with final states in $P_{G,A}$, we conclude that Reachability Query is true. If we identify all potential paths and none of them is eligible, the answer is false.

Algorithm 1 provides the details of the basic DFS algorithm. We use p to save the current path information during the exploration (Line 1). We search all potential simple paths from the source vertex s with start state $A.\alpha_0$ (Line 4). If the current vertex u is target vertex t and the corresponding state α is in the set of final states (Line 7), Line 8 returns true for Reachability Query and Line 9 outputs the current path for Enumeration Query and then continues the exploration. Otherwise, Lines 10-12 loop over $N^P_{out}(u, \alpha)$ to extend path p. Line 11 checks whether we can extend p by adding v to generate p satisfying the simple path constraint. While DFS supports all expressions, it may encounter the same traps multiple times, leading to expensive time overhead.

Example 4. *In Figure 2, considering the query is $(V_0, V_{15}, (0)^* \circ (1)^*))$. We will find only one path $(V_0, V_1, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15})$ as the result. However, if the DFS explores the path following vertex ID order, the path (V_0, V_1, V_2) is found, and we cannot extend the path to the target because any path must cross V_1 twice or cannot satisfy the regular expression (i.e., V_9 cannot arrive at V_{15}). Similarly, all paths from V_1 to V_i with $2 \le i \le 9$ cannot reach V_{15}. However, the DFS algorithm explores these traps many times.*

3.2 Reachability Query

Bidirectional BFS (BBFS) [41] is an online algorithm for Reachability Query in this paper. BBFS detects all the potential simple paths whose labels are prefixes of expressions simultaneously from both source vertex s and target vertex t using the *breadth-first search (BFS)* strategy. When a forward path and a backward path meet at the same intermediate vertex, BBFS combines them as the whole path and checks whether the path is a simple path and matches the regex. When an eligible path is found, it means that s and t are *reachable*. Otherwise, they are *not reachable*. Because BBFS simply detects all half simple paths from both the source vertex and the target vertex, in the worst case, BBFS may explore unnecessary

[2]We first construct the NFA for the expression R using Thompson's construction algorithm [38], then we will transfer the NFA to the equivalent DFA A.

Algorithm 1: DFS

Input: The product graph $P_{G,A} = (V_P, E_P)$, source vertex s, target vertex t, the regex R and DFA A;
Output: return the answer of Reachability Query or Enumeration Query;
1 $p \leftarrow \emptyset$;
2 **foreach** $v \in V_p$ **do**
3     $visited[v] \leftarrow$ false;
4 DFS $(s, A.\alpha_0, t, A.F)$;
5 **Procedure** DFS (u, α, t, F)
6     $p \leftarrow p \cup \{u\}$; $visited[u] \leftarrow$ true;
7   **if** $\alpha \in F$ *and* $u = t$ **then**
8   If query is Reachability Query, print true and exit;
9   If query is Enumeration Query, print p;
10   **foreach** $(v, \beta) \in N^P_{out}(u, \alpha)$ **do**
11   **if** $visited[v]$ *is false* **then**
12   DFS (v, β, t, F) ;
13     $p \leftarrow p - \{u\}$; $visted[u] \leftarrow$ false;

paths that are not in the path connecting the source and target vertices. Computing these paths uses much time and cannot get the reachability for the query. In order to address the drawback, we propose an efficient approach for solving Reachability Query with a theoretical guarantee.

Example 5. *In Figure 2, assume that the query is* $(V_0, V_{15}, (0)^* \circ (1)^*)$. *We can find a path* $(V_0, V_1, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15})$ *and return the reachable answer. Similar to example 4, all paths contain edge* $(V_1, 0, V_i)$ *with* $2 \leq i \leq 4$ *cannot reach* V_{15} *while satisfying the expression. However, BBFS explores all these useless paths to return a true answer.*

3.3 Enumeration Query

Except for DFS based algorithm, a recent work by Martens and Trautner [26] shows that Enumeration Query can be solved with polynomial delay under DCE and proposes an algorithm with an $O(n^3)$ delay by extending Yen's algorithm [43]. The key idea is that for DCE, the regular shortest path is also a regular simple path. Therefore, by finding the top-k shortest paths and making k infinity, all the simple paths that match DCE can be obtained.

This method is much faster than DFS. However, before finding the next shortest path, it needs to prohibit certain edges to avoid generating repeated results. This process involves checking all output paths, which may be time-consuming, especially when a large number of shortest paths have been discovered.

Our objective is to propose a more efficient algorithm than [26] to address Enumeration Query under DCE. Subsequently, we plan to extend this efficient algorithm to support all TRE.

4 ALGORITHM OVERVIEW

For TRE, solving Reachability Query and Enumeration Query under length-fixed expressions is straightforward as we only need to explore a fixed number of steps to complete the search. In this paper, we use DFS-based algorithm to compute Pre and Suf of query expressions. The most time-consuming part is dealing with $Type$, because there are Kleene stars in this part, the length of the path can be very large up to $|V|$. Therefore, finding a faster way to answer Reachability Query and Enumeration Query under downward closed expressions is crucial to designing an efficient

Algorithm 2: Framework

Input: Graph $G = (V, E, \mathcal{L}, \phi)$, source and target vertices s, t, and the TRE R;
Output: returns reachability or enumeration result
1 $Pre \circ Type \circ Suf \leftarrow R$;
2 $A_{Pre} \leftarrow$ ConstructDFA(Pre) ;
3 $A_{Suf} \leftarrow$ ConstructDFA(Suf) ;
4 Algorithm 1 Line 1-3;
5 $p \leftarrow \emptyset$;
6 $P_f \leftarrow$ForwardDFS $(s, A_{Pre}.\alpha_0, A_{Pre}.F)$;
7 $P_b \leftarrow$BackwardDFS $(t, A_{Suf}.F, A_{Suf}.\alpha_0)$;
8 **foreach** $p_1 = (v_1 = s, v_2, \ldots, v_x) \in P_f$ **do**
9   **foreach** $p_2 = (u_1, u_2, \ldots, u_y = t) \in P_b$ **do**
10         $V' \leftarrow V - (\{p_1 - v_x\} \cup \{p_2 - u_1\})$;
11         $E' \leftarrow \{(u, v) | u, v \in V', (u, v) \in E\}$;
12   Query on $G' = (V', E')$ from v_x to u_1 with $Type$;
13 **Procedure** ForwardDFS (u, α, F)
14   Algorithm 1 Line 6;
15     $P \leftarrow \emptyset$;
16   **if** $\alpha \in F$ **then**
17         $P \leftarrow P \cup \{p\}$;
18   **foreach** $(v, \beta) \in N^P_{out}(u, \alpha)$ **do**
19   Algorithm 1 Line 11 ;
20         $P \leftarrow P \cup$ ForwardDFS (v, β, F) ;
21   Algorithm 1 Line 13;
22   return P;

algorithm under TRE constraints. Next, we propose the framework to compute Reachability Query and Enumeration Query.

Algorithm 2 gives a framework of our algorithm for Reachability Query and Enumeration Query. Given a graph G and a query $q(s, t, R)$, we first divide the regex R into three parts Pre, $Type$, and Suf where Pre and Suf are length-fixed expressions and $Type$ is a downward closed expression. Next, we explore the paths from the source vertex s forwardly based on Pre and record all information of suitable paths (Line 6). Similarly, we explore the paths from target vertex t backwardly based on Suf and record the path information (Line 7). Here we give the details of the forward search (Lines 13-22), while the backward search can be handled similarly.

Then, we combine the forward paths (i.e., P_f) and backward paths (i.e., P_b) in pairs and get the last vertex of the forward path (i.e., v_x) and the first vertex of the backward path (i.e., u_1) while putting other vertices of forward and backward paths into set S. Finally, we delete all vertices in set S to satisfy the simple constraint and run the efficient algorithm under $Type$ between v_x and u_1 to answer Reachability Query and Enumeration Query (Lines 8-12).

Example 6. *In figure 1(a), assume that query is* $(V_0, V_6, 0^* \circ 1^* \circ 1)$, *then we divide the regex into* $Pre = \emptyset$, $Type = 0^* \circ 1^*$ *and* $Suf = 1$ *at first. Then we do the forward and backward DFS explorations starting from* V_0 *and* V_6 *respectively. Due to the fact that* Pre *is empty, so we only do backward DFS exploration from* V_6 *and find one path* $V_5 \xrightarrow{1} V_6$. *Next, we delete vertex* V_6 *in* G *and check if there exists one simple path from* V_0 *to* V_5 *that matches* $Type$ *or find all the simple paths. Finally, we return reachable or find the simple paths* $V_0 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{1} V_5 \xrightarrow{1} V_6$ *and* $V_0 \xrightarrow{0} V_1 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{1} V_5 \xrightarrow{1} V_6$.

5 REGULAR SIMPLE PATH REACHABILITY

In this section, we introduce the algorithm named RTRE (Reachability for TRE) under $Type$. We first show an important theorem, and then present the details.

THEOREM 5.1. *For DCE, if we find an arbitrary path between s and t that matches the expression, there must exist at least one simple path between these two vertices that also matches this constraint.*

PROOF. If the arbitrary path does not contain any cycle, it is also a simple path that matches the regex. We consider the situation that the arbitrary path has at least one cycle. W.l.o.g., suppose that the path $p = s \xrightarrow{l_0} v_0 \cdots \xrightarrow{l_i} v_i \cdots \xrightarrow{l_j} v_j \xrightarrow{l_{j+1}} v_{j+1} = v_i \cdots \xrightarrow{l_t} t$ has one cycle, which matches the given regex R. When we delete the cycle in p, the new path will be a simple path and its corresponding label will be $l_0 \cdots l_i, l_{j+1} \cdots l_t$, which still matches the given expression based on the definition of DCE. The new path becomes the simple path between s and t that matches the regex R. If the path has more than one cycle, we can do this process repeatedly until the new path has no cycle. The final path that contains no cycle is the simple path that still matches the expression. ◻

According to theorem 5.1, for DCE, we can check for the existence of regular simple paths by finding a regular shortest path, which allows us to explore each edge only once.

Based on this observation, we design an algorithm based on bidirectional BFS with a block technique which also provides a theoretical guarantee for answering Reachability Query efficiently. We perform two directional BFS walks starting from the source and target vertices in the product graph $P_{G,A}$ simultaneously. For each vertex in $P_{G,A}$, we create two vectors to keep track of whether they are visited in forward and backward explorations, respectively. If we find a vertex is visited in both forward and backward searches, we return *reachable*, otherwise, we return *not reachable*. The key difference between RTRE and BBFS [41] is that RTRE explores edges in $P_{G,A}$ only once, whereas this exploration is performed multiple times in BBFS, making RTRE generally faster than BBFS. Additionally, BBFS saves all potential path information, while RTRE only records whether a vertex is explored, resulting in lower space requirements compared to BBFS.

Algorithm 3 presents the details of RTRE. Lines 1-7 complete the initialization of RTRE. Lines 9-15 record information about forward walks, and return reachable when the forward walks meet backward walks (Lines 11-12). Lines 17-22 record information about backward walks, and the remaining steps are similar to the progress of forward walks. If we finish exploring forward walks or backward walks (Line 8), we return unreachable.

EXAMPLE 7. *Assume that the query is $(V_0, V_5, 0^* \circ 1^*)$ in Figure 1(a). We need to find a simple path from $(V_0, 0)$ to $(V_5, 0)$ or $(V_5, 1)$ in $P_{G,A}$. Table 3 shows details of forward and backward explorations and records vertices that have been explored in each step. Step 0 means initialization. When we perform the forward search in step 3, we find the vertex $(V_3, 0)$ has been explored in both forward and backward searches. Hence, we can stop the search and return reachable.*

Table 3: The exploration of RTRE

Step	Forward search	Backward search
0	$(V_0, 0)$	$(V_5, 0), (V_5, 1)$
1	$(V_0, 0), (V_1, 0), (V_2, 0)$	$(V_5, 0), (V_5, 1)$
2	$(V_0, 0), (V_1, 0), (V_2, 0), (V_7, 1)$	$(V_5, 0), (V_5, 1), (V_3, 0)$
3	$(V_0, 0), (V_1, 0), (V_2, 0), (V_7, 1), (V_3, 0)$	

Algorithm 3: RTRE

Input: The product graph $P_{G,A}$, source and target vertices s, t, and regex $Type$;
Output: returns true if t is reachable from s matching regex $Type$

1 $Q_F, Q_B, F, B \leftarrow \emptyset$;
2 $A \leftarrow$ ConstructDFA($Type$);
3 $Q_F.push((s, A.\alpha_0))$;
4 $F \leftarrow F \cup \{(s, A.\alpha_0)\}$;
5 **foreach** $\beta \in A.F$ **do**
6     $Q_B.push((t, \beta))$;
7     $B \leftarrow B \cup \{(t, \beta)\}$;
8 **while** $Q_F \neq \emptyset$ *and* $Q_B \neq \emptyset$ **do**
9     $(u, \alpha) \leftarrow Q_F.pop()$;
10   **foreach** $(v, \beta) \in N^P_{out}(u, \alpha)$ **do**
11   **if** $(v, \beta) \in B$ **then**
12   **return** *true*;
13   **if** $(v, \beta) \notin F$ **then**
14             $F \leftarrow F \cup \{(v, \beta)\}$;
15             $Q_F.push((v, \beta))$;
16     $(u, \alpha) \leftarrow Q_B.pop()$;
17   **foreach** $(v, \beta) \in N^P_{in}(u, \alpha)$ **do**
18   **if** $(v, \beta) \in F$ **then**
19   **return** *true*;
20   **if** $(v, \beta) \notin B$ **then**
21             $B \leftarrow B \cup \{(v, \beta)\}$;
22             $Q_B.push((v, \beta))$;
23 **return** *false*

THEOREM 5.2. *The time complexity of RTRE is $O(n \cdot k + m \cdot k^2)$, and the space complexity is $O(n \cdot k + m)$, where n is the number of vertices in G, m is the number of edges in G and k is the number of states in the corresponding DFA A of given regex.*

PROOF. If there are k states in DFA A, then there are at most $n \cdot k$ vertices and $m \cdot k^2$ edges in the corresponding product graph $P_{G,A}$. Since every edge in $P_{G,A}$ is explored only once, the time complexity is $O(n \cdot k + m \cdot k^2)$. Each vertex with every state may be saved. The space complexity is $O(n \cdot k + m)$. ◻

6 REGULAR SIMPLE PATH ENUMERATION

For Reachability Query, by Theorem 5.1, if we want to find the reachability for DCE we just search every vertex once in $P_{G,A}$. However, for Enumeration Query, we must enumerate every distinct simple path. Compared with the length-fixed expression Pre and Suf in a query expression, the challenge of regular simple path enumeration is still the DCE. Hence we propose some pruning techniques to help us accelerate enumeration. We find that not all vertices in $P_{G,A}$ can reach the final state. Firstly, we propose a candidate detection method for pruning unnecessary vertices. Secondly, we try to prune the search branch that does not contain any eligible path by previous searching. Thirdly, we improve the pruning rules by the property of DCE based on Theorem 5.1 and propose a faster and polynomial delay enumeration algorithm.

Algorithm 4: CandidateDetection

Input: The product graph $P_{G,A}$, target vertex t, DFA A;
Output: The pruned product graph;

1 $Q \leftarrow \emptyset$;
2 $B \leftarrow \emptyset$;
3 **foreach** $\alpha \in A.F$ **do**
4   Q.push$((t, \alpha))$;
5     $B \leftarrow B \cup \{(t, \alpha)\}$;
6 **while** $Q \neq \emptyset$ **do**
7     $(u, \alpha) \leftarrow Q.pop()$;
8   **foreach** $(v, \beta) \in N^P_{in}(u, \alpha)$ **do**
9   **if** $(v, \beta) \notin B$ **then**
10             $B \leftarrow B \cup (v, \beta)$;
11             $Q.push((v, \beta))$;
12 **return** $P_{G,A}{}' = (V' = \{(u, \alpha) | B[u][\alpha] = true\}, E' = \{(u, v) | u, v \in V'\})$

6.1 Candidate Detection

Although graphs can be very large, when the source and target vertices s and t are given, it is possible to identify many vertices in the graph that do not need to be explored because they cannot reach t under the expression's constraint alone (i.e., ignoring the simple constraint temporarily). In this subsection, we present a technique called *Candidate Detection*, which detects all the candidate vertices in the product graph $P_{G,A}$.

We determine the candidate vertices by using the backward BFS strategy from the target vertex t with the final states of DFA A. Algorithm 4 provides the details. During the exploration from t with the final states (Lines 6-11), we record every vertex in $P_{G,A}$ that has been visited (Line 10). All these visited vertices are the candidate vertices in $P_{G,A}$. Once the candidate vertices are known, we delete other vertices along with their corresponding edges in $P_{G,A}$, to obtain a new product graph P' (Line 12). Algorithm 4 guarantees that each vertex in P' can reach t.

Example 8. *In Figure 2, considering the query is* $(V_0, V_{15}, (0)^* \circ (1)^*)$. *We notice that vertices* V_7, V_8, V_9 *cannot reach* V_{15} *while satisfying the expression. Therefore, we delete the corresponding vertices and edges.*

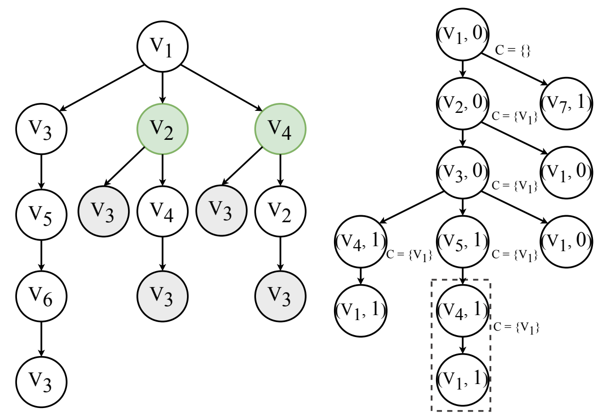
6.2 Pruning by Conflict Sets

After obtaining the new product graph P', we need to find the exact simple paths that satisfy the given regex. DFS is an intuitive solution to enumerate all simple paths. However, DFS may waste time in futile branches of the search tree. We are trying to find the feature (conflict sets) of futile braches. Since we explore the search tree using the DFS strategy, once we visit a new vertex M (via one of its parents), we explore the subtree rooted at M and then return to the parent of M. Now we assume that M cannot reach the target vertices, which means there are some vertices (referred to as conflict vertices) that are explored repeatedly in both paths from the source vertex to M and the subtree rooted at M. We may arrive at M again in the future (from another parent of M), and if the new path between a source vertex and M also includes those conflict vertices, we do not need to explore the subtree rooted at M again. Therefore, our goal is to utilize the conflict information obtained from the exploration of the subtree to prune some useless searches in the future.

Example 9. *Considering the query is still* $(V_0, V_{15}, (0)^* \circ (1)^*)$ *in Figure 2. Figure 3(a) shows a part of the search tree rooted at vertex* V_1. *Assuming that we finish exploring the subtree rooted at* V_3 *from edge* $V_1 \xrightarrow{0} V_3$, *then we find a useless path* $V_3 \xrightarrow{0} V_5 \xrightarrow{0} V_6 \xrightarrow{0} V_3$ *since* V_3 *is visited twice. We notice that if the current path contains* V_3, *we do not need to explore this useless path. The vertices with shadows in Figure 3(a) show the roots of the subtrees that can be pruned.*

Computing Conflict Sets. Next, we describe how to compute the conflict set C_M of node M in a bottom-up fashion. Initially, we define the leaves of the search tree, which can be categorized into the following three types, w.l.o.g., assuming that we have explored the simple path p_c and arrived at node M in $P_{G,A}$.

(1) A leaf is a *conflict node* in the subtree rooted at M if and only if we should stop exploring this node due to the simple constraint.
(2) A leaf can be a target node, indicating we find a result.
(3) Otherwise, we refer to this leaf as a *normal node.*



(a) The search tree in Figure 2   (b) The search tree in Figure 1(c)

Figure 3: Example of pruning by conflict sets.

Now, we compute the conflict set of node M by using its children and their corresponding conflict sets. Assume that node M has k children $M_1, \cdots, M_k$, and we have already computed the conflict set $C_{M_1}, \cdots, C_{M_k}$ of these children, respectively. Based on the following cases, we can compute the conflict set C_M of node M.

If there exists a child node M_i such that M_i is the target node, we set $C_M = \emptyset$. Otherwise, if child nodes are all normal nodes, we set $C_M = \cap_{i=1}^{k} C_{M_i}$. Otherwise, we use set C to record all the *conflict nodes*. If there are at least two distinct nodes in C, we set $C_M = \emptyset$. Otherwise, we set $C_M = (\cap_{M_i \in C} C_{M_i}) \cap C$.

Note that the conflict sets may be computed multiple times for the same node, and the updates are done in an additional manner rather than a coverage manner, which means that new values are inserted into the conflict sets without deleting old values. Additionally, in this part, we only consider the simple constraint based on node information, so in the conflict sets, we only include the node information while ignoring the state information. For example, if a node (u, α) has two conflict nodes (v, β_1) and (v, β_2), even though these two nodes have different states, the conflict set of (u, α) is v but not $\emptyset$ because the node v is the essential reason why exploration is stopped, rather than the state β_1, β_2.

Algorithm 5: Conflict DFS

Input: Product graph $P_{G,A}$, source and target vertices s, t, and regex $Type$;
Output: all simple paths between s and t matching $Type$

1 $A \leftarrow$ ConstructDFA($Type$);
2 $P' \leftarrow$ CandidateDetection($P_{G,A}$,t, A);
3 Initialize Conflict set C to empty;
4 Return `Conflict DFS(`$s, A.\alpha_0, t, A, \{s\}$`)`;
5 **Procedure** `Conflict DFS(`u, α, t, A, p, P'`)`
6   Result $\leftarrow \emptyset$;
7   **if** $u = t$ *and* $\alpha \in A.F$ **then**
8   **return** *{p}*;
9   **foreach** $(v, \beta) \in N_{out}^{P'}(u, \alpha)$ **do**
10   **if** $p \cap C[v][\beta] \neq \emptyset$ **then**
11   break;
12   **if** $v \notin p$ **then**
13   Result $\leftarrow$ Result $\cup$ `Conflict DFS(`$v, \beta, t, A, p \cup \{v\}, P'$`)`;
14   Update the conflict set for (u, α);
15   **return** *Result*;

Algorithm 5 shows how to utilize conflict sets for pruning the search tree. Line 10 illustrates our pruning technique. Specifically, if we find one node in the current path that is present in its conflict set, we can terminate the exploration, as the new path after exploration must conflict with the simple constraint. Note that Conflict DFS can handle any expressions, but in some bad cases, it may still cost exponential steps to return results.

THEOREM 1. *Conflict DFS returns all the simple paths between the start vertex s and the target vertex t.*

PROOF. Based on the computation of conflict sets, the conflict set C of vertex $(u, state)$ has vertex v, which means all the exploration starting at vertex $(u, state)$ must arrive at vertex v. Otherwise, C should not contain v. Therefore, if the current path contains vertex v, there exists no simple path as a final result. When the current path does not contain v, Conflict DFS will explore from vertex $(u, state)$. Overall, Conflict DFS does not affect the correctness. □

EXAMPLE 10. *Figure 3(b) depicts the search tree rooted at $(V_1, 0)$ for query $(V_0, V_7, 0^* \circ 1^*)$ in figure 1(a). When we reach $(V_3, 0)$ and assume to go to $(V_4, 1)$ first, we find $(V_4, 1)$ has only one conflict node $(V_1, 1)$, so the conflict set of $(V_4, 1)$ is V_1. As we arrive at $(V_4, 1)$ again through $(V_5, 1)$, we notice that the current path contains node $(V_1, 0)$, which leads us to stop the search. The dashed box represents the useless searches. Based on the above method, we can compute all conflict sets and obtain the result $(V_0, 0) \rightarrow (V_1, 0) \rightarrow (V_7, 1)$ after finishing the exploration starting from path $(V_0, 0) \rightarrow (V_1, 0)$.*

6.3 Pruning by Block

The conflict set keeps track of the vertices that have been visited. If the current partial path does not include all the vertices in the conflict set, we cannot determine whether the current search branch contains an eligible simple path.

EXAMPLE 11. *As in the example 9, the shadow vertices in Figure 3 are pruned by Conflict DFS. But, we have to detect them and prune them. For instance, we will explore all the paths from V_3 to V_1, V_2, V_4 (such as $V_3 \rightarrow V_1$, $V_3 \rightarrow V_2 \rightarrow V_1$, $V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow V_1$, etc.) leading to the exponential number of paths, and then finish exploring the subtree rooted at V_3. Before backtracking to V_1, we compute the conflict set of V_3. Unfortunately, the conflict set is empty, which leads to another bad case when we arrive at V_3 through another vertex (like V_2), we have to explore the above paths although these paths cannot arrive at the target vertex V_{15}. Worse, similar situations also occur in the exploration of V_2 and V_4.*

To address the limitation of conflict sets, we propose a more efficient method called ETRE (Enumeration for TRE) than Conflict DFS under $Type$. After candidate detection, this method involves using a blocked set (B) to determine whether each vertex in the new product graph P' needs to be explored. Specifically, in the DFS exploration, we will put some vertices into the blocked set after they have been visited. Note that the vertex in the blocked set cannot be explored until it is not in the blocked set.

However, simply putting vertices into the blocked set is not adequate to complete the exploration, as it fails to consider the simplicity constraint. For instance, in Figure 1(c), if we go through point $(V_1, 0)$ to point $(V_4, 1)$ and add all these vertices into B, the next point will be $(V_1, 1)$. However, owing to the simple constraint, reaching $(V_1, 1)$ is not permissible. Therefore, we must maintain a record of all the vertices information in the current path p. When we encounter a vertex (u, α) during the exploration of P', we also need to verify whether its vertex information (i.e., u) is present in p. We will explore the vertex only if (u, α) is not in B and the current path p does not contain u. Otherwise, we terminate the exploration and backtrack. Hence, in the aforementioned example mentioned, when we reach $(V_1, 1)$, we halt the exploration since V_1 is already in p. If a vertex can be encompassed by the blocked set, we refer to it as a blocked vertex (or simply, we block this vertex).

Next, we illustrate the principle of the blocked set using the same query in example 10. We show how the blocked set evolves and how repetitions of useless explorations can be pruned. At this stage, readers do not need to worry about how to update the blocked set when backtracking to find the next regular simple path, as this process will be addressed after this example. In Figure 1(a), assume that we explore the path $V_0 \xrightarrow{0} V_1 \xrightarrow{0} V_2 \xrightarrow{0} V_3 \xrightarrow{1} V_4$ first. The corresponding exploration in the product graph is as follows:

- $(V_0, 0) \rightarrow (V_1, 0)$: Put $(V_1, 0)$ into B and add V_0, V_1 to p.
- $(V_1, 0) \rightarrow (V_2, 0)$: Put $(V_2, 0)$ into B and add V_2 to p.
- $(V_2, 0) \rightarrow (V_3, 0)$: Put $(V_3, 0)$ into B and add V_3 to p.
- $(V_3, 0) \rightarrow (V_4, 1)$: Put $(V_4, 1)$ into B and add V_4 to p.
- Then, we do not explore vertex $(V_1, 1)$ because V_1 is in the current path p. We backtrack to $(V_3, 0)$ to find other paths. So we reach $(V_5, 1)$ and add $(V_5, 1)$ to B. Then, we attempt to explore the vertex $(V_4, 1)$ again. However, due to $(V_4, 1)$ being in B, we abort this search and backtrack.
- Next, we will backtrack to vertex $(V_1, 0)$ and find one result $V_0 \xrightarrow{0} V_1 \xrightarrow{1} V_7$.

While this appears promising, we must acknowledge a major issue that has been overlooked. During backtracking to find the next path, previously blocked vertices may become available again, which can impact the accuracy of the blocked set. We extend the previous example to illustrate this problem. After finding the result $V_0 \xrightarrow{0} V_1 \xrightarrow{1} V_7$, we will explore vertex $(V_2, 0)$ as we have finished all the exploration rooted at $(V_1, 0)$. However, we cannot explore vertex $(V_2, 0)$ since $(V_2, 0)$ is still in B. Nevertheless, there exists

Algorithm 6: Unblock

```
Input: Block set B, unblock list U, vertex u, state α
Procedure Unblock(B, U, u, α)
    B ← B − {(u, α)} ;
    foreach (v, β) ∈ U[u][α] do
        Unblock(B, U, v, β) ;
    U[u][α] ← ∅
```

a path $V_0 \xrightarrow{0} V_2 \xrightarrow{0} V_1 \xrightarrow{1} V_7$ that satisfies our requirements. The problem is that the blocked set is not updated in a timely manner. Therefore, we need to evaluate under which conditions the vertex could be unblocked (i.e., delete it from the blocked set) at the same time when we block it.

There are two cases in which we should update the blocked set. Firstly, when we find a path that satisfies our request, we should unblock all the vertices in this path. Secondly, there are cases where we need to unblock certain vertices while unblocking others. For example, if we unblock vertex $(V_1, 0)$, we should also unblock vertex $(V_3, 0)$, as if $(V_1, 0)$ could be explored again, $(V_3, 0)$ will be also available. This ensures that the blocked set is updated in a comprehensive and accurate manner, taking into account both the path information and the state information of the vertices.

Therefore, we add an unblock list for every vertex, which maintains a list of vertices that should be unblocked when the vertex itself is unblocked. This allows for cascading unblock operations. In our example, when we find the result $V_0 \xrightarrow{0} V_1 \xrightarrow{1} V_7$, the vertex $(V_1, 0)$ will be unblocked. We delete $(V_1, 0)$ from the blocked set. Since $(V_3, 0)$ is in the unblock list of $(V_1, 0)$, it will be deleted as well, leading to the deletion of $(V_2, 0)$, and this process continues until the unblock list is empty. This ensures that all the vertices that need to be unblocked are properly updated, allowing for correct handling of cascade unblock operations.

Overall, ETRE could be considered as an improved method of conflict sets. Conflict sets record local information, which can lead to repeated exploration of unproductive paths when previous paths change. On the other hand, ETRE maintains global information to efficiently prune the search space, ensuring that no futile searches are explored between two results outputs. The pseudocode for the ETRE algorithm is presented in Algorithm 7.

Specifically, we block every vertex during the exploration (Line 9). When we meet the blocked vertex or the current path contains its vertex information, we will terminate the further exploration (Line 14). When we finish the exploration of one vertex and find one result, we will unblock this vertex (Lines 16-17). Note that the unblock progress is recursive until no vertex can be unblocked. Otherwise, we will keep blocking this vertex and record the information about how can we unblock it (Lines 19-20).

Theorem 6.1. *ETRE returns all the simple paths between the start vertex s and the target vertex t that match the given downward closed expression R.*

Proof. First of all, it is important to realize that ETRE is a truncated depth-first search: all paths are explored except for paths that contain blocked vertices. There are two cases where a vertex can be blocked. In the first case, the vertices in the currently explored path are blocked. When we arrive at them again in the exploration, we need to terminate the exploration since it leads to a non-simple

Algorithm 7: ETRE

```
Input: product graph P_{G,A}, source and target vertices s, t, and Regex Type;
Output: all simple paths between s and t matching Type
1  A ← ConstructDFA(Type);
2  P′ ← CandidateDetection(P_{G,A}, t, A);
3  p ← {s};
4  B ← ∅;
5  foreach (u, α) ∈ P′ do
6      U[u][α] ← ∅;
7  return ETRE (s, A.α_0, t, p, P′);
8  Procedure ETRE (u, α, t, p, P′)
9      B ← B ∪ {(u, α)} ;
10     Result ← ∅ ;
11     foreach (v, β) ∈ N^{P′}_{out}(u, α) do
12         if v = t and β ∈ A.F then
13             Result ← Result ∪ {p};
14         if v ∉ p and (v, β) ∉ B then
15             Result ← Result ∪ ETRE (v, β, t, p ∪ {v}, P′)
16     if Result ≠ ∅ then
17         Unblock(B, U, u, α);
18     else
19         foreach (v, β) ∈ N^{P′}_{out}(u, α) do
20             U[v][β] ← {(u, α)};
21     return Result;
```

path. In the second case, during backtracking, even if some vertices are not in the current path, they are still blocked. Due to the blocking and unblocking of all the vertex states, the specific state of a vertex does not impact the correctness of the result. Now assuming that we have explored the path $s \rightarrow \cdots \rightarrow v_i \rightarrow \cdots \rightarrow v_k$, if we have already explored the subtree rooted at v_k and determined v_k should not be unblocked, it implies that v_k is not in a result path. This indicates that v_k must conflict with a vertex in a later part of the path, represented by v_i. Therefore, v_k will be unblocked only if v_i is unblocked. However, v_i can only be unblocked if we have finished exploring the subtree rooted at v_i, which means it does not affect the subsequent exploration. Considering the recursive nature of the process, block and unblock techniques can not affect the correctness of the exploration. ◻

Theorem 6.2. *Let n, m be the number of vertices and edges in G, and k be the number of states in DFA A. For DCE, ETRE satisfies polynomical delay. Between two paths being output, ETRE takes at most $O(n \cdot k + m \cdot k^2)$ steps. The total time complexity of ETRE is $O((c+1)(n \cdot k + m \cdot k^2))$, and the space complexity is also $O(n \cdot k + m \cdot k^2)$ where c is the number of results.*

Proof. The proof of this theorem is based on the design that the only way to unblock one vertex is by a call Unblock, which only happens when a path is output. Whenever a path $s \rightarrow v_1 \cdots \rightarrow v_i \rightarrow t$ is output, Unblock will be executed for v_i (i.e., the prefix path is $s \rightarrow v_1 \cdots \rightarrow v_i$), then for v_{i-1}, until it is called for v_1. However, every call for Unblock will unblock different vertices. Indeed, if now the prefix path is $s \rightarrow v_1 \cdots \rightarrow v_j$, a call to Unblock only unblocks the vertices that at this moment cannot be used in a path to the target vertex t, but once v_j is unblocked, they will become available again. Therefore, every edge can be explored at most twice between two paths output. Hence, either a path will be output after $O(m \cdot k^2)$, or all the vertices will be blocked and we terminate the algorithm. The term $O(n \cdot k)$ is for the initialization of the blocked switches of all vertices at the start of the algorithm. We use $O(n \cdot k)$ space to block vertices and $O(m \cdot k^2)$ space to record the unblock list U. □

7 REACHABILITY VERTEX PAIR QUERY

The problem of returning the pairs of vertices connected by simple paths satisfying a given regular expression is NP-hard. [27] proposes an efficient method that can address this problem in polynomial time under restricted regular expressions (which are equal to downward closed expressions, the details can be seen in [27]) even considering the simple paths. Unfortunately, using the method in [27] to evaluate simple queries under the expression form $Pre \circ Type$ takes exponential time. Note that $Type \circ Suf$ is very similar to $Pre \circ Type$. Hence, we only discuss $Pre \circ Type$.

In this section, we introduce how our method can be extended to solve this problem under the above expressions. Algorithm 8 gives the details of our method. To be specific, the expression is firstly broken into two parts (i.e., Pre and $Type$). Then we use the DFS method to find all the vertices that the start vertex s can reach under Pre constraint through simple paths and record the corresponding simple paths (Line 4). Then for every path, we delete all the vertices in the path except the final vertex (Lines 6-7) and run the efficient algorithm under $Type$ from v_s to answer the query (Lines 8-16). Specifically, we can find all the vertices that one vertex can reach by finding single source shortest paths. We design an algorithm based on BFS with a block technique to find the vertices. We perform a BFS walk from the source vertice in the product graph $P_{G,A}$ and create a vector to keep track of whether the vertices in $P_{G,A}$ are visited in the exploration, which contributes to the exploration from every vertex can be only constructed once (Lines 14-15). During the exploration, we record all the visited vertex with the final states as results (Lines 12-13). We do the above progress for all the vertices in G and then we find all pairs of vertices connected by simple paths satisfying the expression (Line 3).

Theorem 7.1. *The time complexity of Algorithm 8 is* $O(n\cdot(m_d)^{k_p} + c(n \cdot k_t + m \cdot k_t^2))$, *and the space complexity is* $O((m_d)^{k_p} + n \cdot k_t + m)$, *where* m_d *is the maximal degree of* G, c *is the number of all forward paths,* n *is the number of vertices in* G, m *is the number of edges in* G, *and* k_p *and* k_t *is the number of states in the corresponding DFA of* Pre *and* $Type$, *respectively.*

Proof. It takes $O(n \cdot (m_d)^{k_p})$ time to find all the forward paths, and we need $O(n \cdot k_t + m \cdot k_t^2)$ to find the results for each forward path. It takes at most $O((m_d)^{k_p})$ space to save the forward paths, and we need $O(n \cdot k_t)$ space to record the information about whether vertices are visited. The final time cost is $O(n \cdot (m_d)^{k_p} + c(n \cdot k_t + m \cdot k_t^2))$, and the final space cost is $O((m_d)^{k_p} + n \cdot k_t + m)$. □

8 EXPERIMENTS

In this section, we present experimental results to demonstrate the efficiency of our methods for both Reachability Query and Enumeration Query.

All algorithms are implemented in C++ and compiled with g++ with O3 optimization. The experiments are performed on a machine with an Intel Xeon 2.1GHz CPU and 256G memory.

Algorithm 8: Reachability Vertices Pair Query

Input: Graph $G = (V, E, \mathcal{L}, \phi)$ and the expression R;
Output: returns the pairs of nodes connected by simple paths satisfying R

1 $Pre \circ Type \leftarrow R$; $Q, Result \leftarrow \emptyset$;
2 $A_{Pre} \leftarrow$ ConstructDFA(Pre); $A_{Type} \leftarrow$ ConstructDFA($Type$) ;
3 **foreach** $s \in V$ **do**
4 $\quad P_f \leftarrow$ `ForwardDFS` $(s, A_{Pre}.\alpha_0, A_{Pre}.F)$;
5 $\quad$ **foreach** $p_1 = (v_1 = s, v_2, \ldots, v_x) \in P_f$ **do**
6 $\quad\quad V' \leftarrow V - (\{p_1 - v_x\})$; $E' \leftarrow \{(u, v) | u, v \in V', (u, v) \in E\}$;
7 $\quad\quad Q.push((v_x, A_{Type}.\alpha_0))$; $F \leftarrow \emptyset$;
8 $\quad\quad$ **while** $Q \neq \emptyset$ **do**
9 $\quad\quad\quad (u, \alpha) \leftarrow Q.pop()$;
10 $\quad\quad\quad$ **foreach** $(v, \beta) \in N^P_{out}(u, \alpha)$ **do**
11 $\quad\quad\quad\quad$ **if** $\beta \in A_{Type}.F$ *and* $(s, v) \notin Result$ **then**
12 $\quad\quad\quad\quad\quad Result \leftarrow Result \cup (s, v)$;
13 $\quad\quad\quad\quad$ **if** $(v, \beta) \notin F$ **then**
14 $\quad\quad\quad\quad\quad F \leftarrow F \cup \{(v, \beta)\}$;
15 $\quad\quad\quad\quad\quad Q_F.push((v, \beta))$;
16 **return** $Result$

Table 4: Statistics of datasets used in the experiments. ($K = 10^3, M = 10^6, B = 10^9$)

Name	Dataset	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	$\frac{\|E\|}{\|V\|\|\mathcal{L}\|}$	Type	Syn
AD	Advogato	6.5K	51.1K	4	1.96	Trust	
EC	econ-psmigr1	3.1K	543K	8	21.89	Economic	✓
TR	Wiki-trust	139K	740K	8	0.66	Interaction	✓
HS	StringsHS	19K	1.24M	8	8.15	Biological	
BG	BioGrid	64K	1.5M	7	3.34	Biological	
FC	StringsFC	19K	2.04M	8	13.42	Biological	
ND	NotreDame	326K	1.47M	8	0.56	Web	✓
SF	Web-stanford	282K	2.3M	8	1.01	Web	✓
BK	Baidu-baike	416K	3M	8	0.90	Web	✓
GG	Web-google	876K	5M	8	0.71	Web	✓
DA	Rec-dating	169K	17M	10	10.05	Recommendation	
YT	Youtube	14.9K	13.6M	5	182.55	Social	
EP	Soc-Epinsion1	75K	508K	8	0.84	Social	✓
SO	StackOverflow	2.6M	63M	3	8.07	Social	
ZS	zhishihudong	2.4M	18.9M	8	0.98	Miscellaneous	✓
FS	friendster	65M	2.6B	30	1.33	Miscellaneous	✓
WD	Wikidata	296M	958M	5419	0.001	Miscellaneous	

Datasets: Table 4 lists the basic information of 17 real-graphs used in our experiments, most of which are used in the related work [31, 36]. These graphs belong to various categories such as social networks, trust networks, interaction networks, and knowledge graphs. Eight of the graphs have natural edge labels, while for the remaining graphs without edge labels, we synthetically generate labels that are exponentially distributed with $\lambda = \frac{|\mathcal{L}|}{\alpha}$, where $\alpha = 1.7$. The last column indicates whether we synthetically generate labels. HS, BG, FC, and FS are undirected graphs, while FC and HS represent the protein networks of the organism *felis catus* and *homo sapiens*, respectively. For all undirected graphs, an undirected edge $(u, l, v) \in E$ was replaced by two directed edges (u, l, v) and (v, l, u) to create a directed graph. For all undirected graphs, an undirected edge $(u, l, v) \in E$ was replaced by two directed edges (u, l, v) and (v, l, u) to create a directed graph.

Comparisons. We investigate the following methods in Reachability Query and obtain source codes from the respective authors.

- **BBFS [41]:** the baseline algorithm.
- **ARRIVAL [41]:** the state-of-the-art approximate method.
- **RTRE:** Algorithm 2 + Algorithm 3.

The methods studied in Enumeration Query are the following:

Table 5: Queries used in experiments

Name	Type	Name	Type
Q_1	a^*	Q_4	$(a_1 + a_2 \cdots + a_k)^*$
Q_2	$a \circ b^*$	Q_5	$a_1 \circ a_2 \cdots \circ a_k$
Q_3	$a \circ b^* \circ c^*$	Q_6	Random TREs

- **Yen's algorithm [26]:** the version of Yen's algorithm [43] for Enumeration Query under DCE.
- **DFS:** the baseline algorithm.
- **Conflict DFS:** Algorithm 5.
- **ETRE:** Algorithm 2 + Algorithm 7.

The methods studied in Reachability Vertices Pair Query are the following:

- **Wood's algorithm [27]:** the start-of-the-art method.
- **RVPM :** Algorithm 8.

Queries. We choose the top-5 frequent queries from Table 1 [3].

Tabel 5 lists the expressions we use in experiments. The number k in Q_4 and Q_5 is chosen from 2 to 6. Q_6 must consist of $Type$, and Pre and Suf are randomly added to Q_6. The length of labels in $Type$ is chosen from a range of 2 to 6, and the number in Pre and Suf is from a range of 1 to 4. We make two different forms for $Type$, i.e., $R = (l_1 + l_2 + \cdots + l_k)^*$ and $R = (l_0) * \circ (l_1) * \circ \cdots \circ (l_k)*$. The form of Pre and Suf is $R = (l_0) \circ (l_1) \circ \cdots \circ (l_k)$.

The source vertex, target vertex, and label are chosen randomly from the graph. We generate 1000 queries for Reachability Query comprising 500 true queries and 500 false queries. Additionally, we have created 100 queries for Enumeration Query, ensuring the existence of at least one result. Due to limited space, we cannot include all experimental results in this paper. The additional experimental results and discussions are given in the long version of our paper, which is available in our source code repository.

8.1 Performance on Reachability Query

As some queries may be challenging instances that cannot be completed within a reasonable time, we set a time limit of 50 seconds for each query. A query is considered *out of time* if it does not finish within the time limit, and we record its query time as 50 seconds for evaluation purposes.

8.1.1 Comparison with BBFS [41]. We compare the performance of BBFS with RTRE. Tabel 6, 7, 11 demonstrate that RTRE is up to three orders of magnitude faster than BBFS on almost of graph datasets under recursive queries (i.e., Q_1, Q_2, Q_6). Although there may be instances where BBFS perform well, such as in EC, RTRE is still faster than BBFS. However, because RTRE makes no effect on non-recursive queries, RTRE shows the same efficiency as BBFS.

8.1.2 Comparison with ARRIVAL [41]. ARRIVAL approximates the diameter of graphs in order to initialize the parameter *walkLength*, which may be time-consuming, especially in larger graphs like SO, WD, and FS. In our comparison of ARRIVAL and RTRE under recursive queries, we observe that RTRE demonstrates comparable efficiency to ARRIVAL since ARRIVAL limits the search path length to reduce the exploration, and RTRE optimizes the exploration under DCE constraint, which makes sure that every edge is visited only once to improve the efficiency. Both of these methods show much better performance than BBFS in almost of cases.

[3]We do not choose A since the final result path only has one edge for this query.

Table 6: Query time in microseconds for Q_1. (TQ: true query, FQ: false query, Rec: recall, OOM: out of memory, OOT: initialization time > 12h.)

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	217	53	**53**	**21**	112	22	0.88
ND	151K	121K	**99**	**33**	326	62	0.81
ZS	4.6K	53	**276**	**39**	314	92	0.61
BG	870K	311K	891	**214**	**513**	297	0.87
BK	670K	47K	**532**	**33**	634	213	0.76
DA	2.4K	149	**648**	**53**	2.6K	971	0.93
EC	131	52	**33**	**15**	139	62	0.99
EP	186	36	**71**	**20**	288	51	1
FC	1.1M	872K	898	**194**	**364**	210	0.94
GG	375	69K	**88**	**37**	258	122	0.91
HS	581K	855K	851	**105**	**421**	232	0.96
SF	2.2K	60K	266	**39**	**101**	48	0.72
SO	11K	265	1K	46	2.5K	552	0.98
TR	375	46	**120**	**29**	343	32	0.97
YT	292K	7.7M	**508**	11K	934	**632**	0.63
WD	5.2M	33K	**443K**	**36**	OOT	OOT	OOT
FS	OOM	OOM	**7.2K**	**46**	OOT	OOT	OOT

Table 7: Query time in microseconds for Q_2.

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	188	49	**29**	**4**	127	40	0.7
ND	200K	32K	**118**	**8**	351	279	0.82
ZS	205K	61	**318**	**21**	363	341	0.52
BG	97K	127K	**254**	**41**	408	206	0.83
BK	360K	30K	**378**	**20**	629	751	0.64
DA	6.1K	591	**655**	**32**	2.6K	1.7K	0.82
EC	171	165	**28**	**27**	222	155	0.99
EP	259	42	**74**	**7**	454	61	0.99
FC	1.8M	1.6M	1.6K	**194**	**543**	304	0.82
GG	1.8K	149K	**120**	**20**	346	338	0.82
HS	430K	1.8M	874	**130**	**535**	346	0.87
SF	200K	212K	432	**35**	**113**	132	0.37
SO	201K	4.4K	**1.3K**	**75**	5.2K	676	0.96
TR	920	51	**174**	**5**	488	96	0.95
YT	7.7K	6.2M	**234**	7.6K	786	**474**	0.41
WD	4.6M	36K	**325K**	**75**	OOT	OOT	OOT
FS	OOM	OOM	**7.1K**	**22**	OOT	OOT	OOT

ARRIVAL, on the other hand, may return approximate results with low *recall* in both real-world graphs (e.g., BG with 34% on Q_6) and synthetically generated labeled graphs (e.g., YT with 39% on Q_6). The discrepancy in the *recall* can be attributed to the fact that *recall* is based on strongly connected graphs, whereas real-world graphs are typically not strongly connected.

However, ARRIVAL shows faster query time under non-recursive queries since RTRE has no difference with BBFS in this case. The results in Table 10 motivate the design of efficient and exact algorithms for RSPQs under non-recursive expressions.

Table 12 provides information on the percentage of queries that can be completed within 1 millisecond (<1ms) and those that exceed the time limit (>50s). In general, RTRE has the highest percentage

Table 8: Query time in microseconds for Q_3.

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	760	146K	**50**	**11**	126	51	0.6
ND	270K	99K	**138**	**14**	278	315	0.62
ZS	13K	111	490	**31**	**397**	644	0.28
BG	374K	424K	**293**	536	381	**210**	0.45
BK	1.3M	314K	940	**834**	**660**	1.4K	0.58
DA	6K	316	**854**	**40**	2.9K	1.1K	0.74
EC	190	158	**37**	**19**	232	103	0.99
EP	7K	31K	**103**	**38**	493	79	0.78
FC	561K	1.4M	911	**146**	**559**	120	0.63
GG	82K	288K	**166**	**42**	387	634	0.67
HS	354K	1.1M	845	1.6K	**620**	**266**	0.8
SF	174K	495K	507	**120**	**124**	179	0.35
SO	718K	370K	6.1K	40.7K	**4.1K**	**316**	0.83
TR	959	72	**220**	**8**	578	221	0.9
YT	9.1K	8.4M	**353**	18K	841	**526**	0.22
WD	4.7M	44K	**583K**	**77**	OOT	OOT	OOT
FS	OOM	OOM	**15.7K**	**31**	OOT	OOT	OOT

Table 9: Query time in microseconds for Q_4.

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	152	49	**61**	**32**	132	66	0.86
ND	1.9M	1.9M	563	409	**287**	**290**	0.82
ZS	721K	93.7K	**917**	**128**	1.4K	1.4K	0.05
BG	1.7M	3.8M	1.5K	4.4K	**484**	**936**	0.8
BK	1.8M	312K	**1.1K**	**121**	1.8K	1.9K	0.76
DA	1.9K	258	**654**	**150**	3.7K	932	0.98
EC	191	159	**85**	60	246	**56**	0.99
EP	285	203	**132**	**84**	553	157	0.98
FC	670K	499K	692	**245**	**586**	320	0.98
GG	12.3M	868K	2.6K	**137**	**644**	1.4K	0.68
HS	244K	302	**533**	**114**	683	186	0.99
SF	8.6M	3.5M	3.1K	946	**200**	**319**	0.53
SO	12K	1K	**1.1K**	**85**	2.1K	489	0.98
TR	445	128	**159**	99	533	**64**	0.71
YT	57.6K	15.8M	**372**	30.6K	801	**990**	0.6
WD	4M	88K	**402K**	**107**	OOT	OOT	OOT
FS	OOM	OOM	**8.9K**	**144**	OOT	OOT	OOT

of queries completed within 1 ms. For instance, BBFS completes 26.9% of queries within 1ms in HS and ARRIVAL completes 58.1%, while RTRE completes 89.2%. Moreover, we observe that BBFS may have a higher ratio of queries completed within 1ms compared to ARRIVAL, indicating that BBFS may perform better than ARRIVAL in favorable instances, as ARRIVAL explores length-constrained simple paths randomly, maybe leading to missing the true paths. However, ARRIVAL shows better performance than BBFS in unfavorable cases. Note that RTRE and ARRIVAL can complete all queries within 50 seconds, while BBFS has varying timeout ratios in different graphs (e.g., YT with 36.7%). Overall, RTRE demonstrates better efficiency than BBFS and ARRIVAL in both favorable and unfavorable instances.

8.1.3 Performance against Query-Label Size. We conduct an analysis of query time for RTRE and BBFS with respect to the number

Table 10: Query time in microseconds for Q_5.

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	59	62	59	62	**43**	**44**	0.7
ND	173	**55**	175	63	**151**	66	0.94
ZS	**89**	**81**	90	84	242	96	0.36
BG	**84**	917	**84**	918	430	**147**	0.002
BK	242	13.9K	243	13.9K	**228**	**107**	0.95
DA	2.6K	121K	2.6K	121K	**1.3K**	**1K**	0.45
EC	2.6K	154K	2.6K	154K	**273**	**341**	0.9
EP	177	2.5K	**179**	2.5K	373	**109**	0.95
FC	1.5M	84K	1.5M	84K	**554K**	**394**	0.66
GG	102	**50**	103	52	**63**	234	0.21
HS	1.6M	136K	1.6M	136K	**663**	**409**	0.93
SF	439	54	436	53	**92**	**83**	0.64
SO	**704**	149K	706	149K	965	**1.5K**	0.96
TR	**394**	2.5K	395	2.5K	460	**97**	0.71
YT	346	291K	348	291K	**292**	**223**	0.24
WD	**2.4K**	**47**	2.4K	50	OOT	OOT	OOT
FS	156K	144K	156K	144K	OOT	OOT	OOT

Table 11: Query time in microseconds for Q_6.

Name	BBFS		RTRE		ARRIVAL		
	TQ	FQ	TQ	FQ	TQ	FQ	Rec
AD	11.6K	1.5M	**104**	127	212	**64**	0.64
ND	9.2M	2M	**2.4K**	535	3K	**455**	0.06
ZS	1.1M	14.1K	11.4K	**525**	**1.3K**	630	0.05
BG	4.3M	7.5M	5.2K	15.6K	**1.2K**	**618**	0.34
BK	13.1M	1.1M	**9.4K**	14.9K	16.2K	**2.1K**	0.31
DA	133K	1.3K	**2.6K**	**473**	12.1K	3.6K	0.66
EC	803	344	**166**	134	318	**85**	0.99
EP	381K	1.1M	**538**	984	1.1K	**137**	0.71
FC	187K	2.3M	**846**	10K	1.2K	**350**	0.79
GG	40.5M	2.2M	8.8K	1.3K	**6.8K**	**1.0K**	0.01
HS	69.3K	1.2K	**817**	50.7K	1.2K	**289**	0.76
SF	18.2M	3.1M	5.3K	835	**495**	**421**	0.06
SO	364K	3.1M	5.2K	265K	**4.8K**	**1.1K**	0.91
TR	46K	170K	**1.0K**	608	1.4K	**43**	0.71
YT	66.2K	24.9M	**813**	17.3K	1.9K	**1.5K**	0.39
WD	51.9K	167.5	**1.3K**	**165**	OOT	OOT	OOT
FS	OOM	OOM	**213K**	**3.9M**	OOT	OOT	OOT

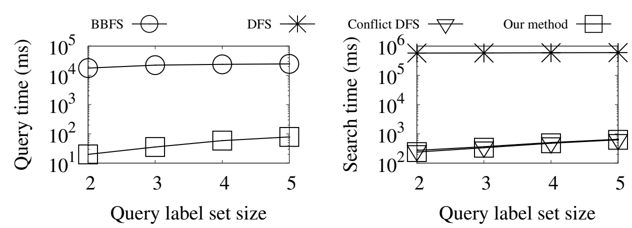


Figure 4: Impact of the number of labels in the query regex on Query time (left) and Search time (right).

of labels in the *Type* on YT, as shown in the left of Figure 4. We observe that the query time increases linearly as the number of labels increases since the number of paths that match the regex also increases with the increase in the number of labels, requiring more exploration of paths. Nevertheless, RTRE is still at least 2 orders of magnitude faster than BBFS on YT.

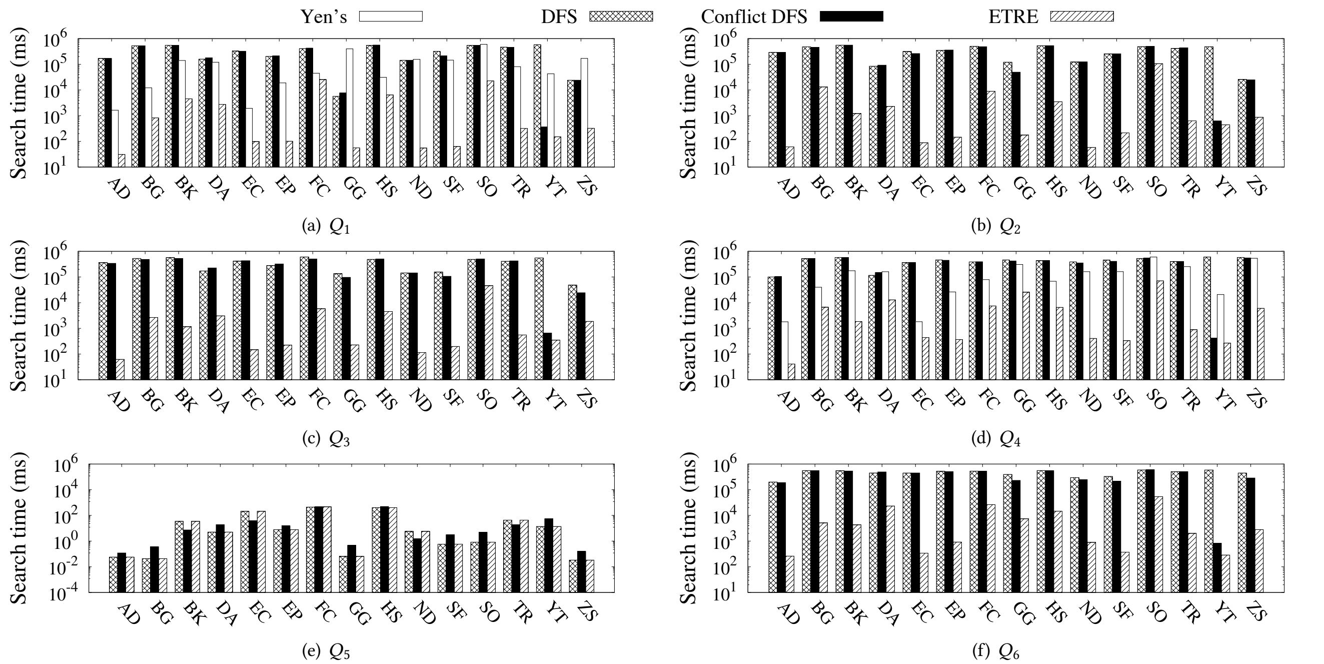


Figure 5: Search time comparison

Table 12: Query time distribution for Reachability Query

Name	BBFS		RTRE		ARRIVAL	
	<1ms	>50s	<1ms	>50s	<1ms	>50s
AD	0.960	0.021	0.987	0.00	**1.00**	0.00
ND	0.917	0.04	**0.956**	0.00	0.92	0.00
ZS	0.955	0.01	**0.971**	0.00	0.897	0.00
BG	0.495	0.108	0.602	0.00	**0.783**	0.00
BK	0.785	0.059	**0.807**	0.00	0.757	0.00
DA	0.445	0.001	**0.71**	0.00	0.535	0.00
EC	0.866	0.00	**1.00**	0.00	0.999	0.00
EP	0.921	0.019	**0.97**	0.00	0.929	0.00
FC	0.242	0.006	**0.847**	0.00	0.597	0.00
GG	0.82	0.108	**0.867**	0.00	0.798	0.00
HS	0.269	0.006	**0.892**	0.00	0.581	0.00
SF	0.789	0.077	0.891	0.00	**0.926**	0.00
SO	0.442	0.04	**0.745**	0.00	0.712	0.00
TR	0.939	0.003	**0.959**	0.00	0.955	0.00
YT	0.334	0.340	**0.646**	0.00	0.471	0.00

8.2 Performance on Enumeration Query

Metrics. For each algorithm, we evaluate their performance based on *search time*, which is calculated as the duration from the start of a query until the first 1000 results are found. To ensure the experiments are feasible within a reasonable time frame, we set 10 minutes as the time limit.

8.2.1 Comparison with Yen's algorithm [26]. Recall that Yen's algorithm only supports DCE, so we compare the performance of ETRE and Yen's algorithm specifically on Q_1 and Q_4. Figure 5(a) and 5(d) illustrates the average search time on Q_1 and Q_4, respectively. ETRE is nearly 2-3 orders of magnitude faster on almost of datasets compared to Yen's algorithm. We also use another metric named result number, which means the number of simple paths that can be found within 1 minute, to test the performance of Yen's algorithm and our method. We randomly generate DCE and compare the performance of ETRE and Yen's algorithm. Figure 6 shows that ETRE can find over 100 times more paths than Yen's algorithm.

Table 13: Query time distribution for Enumeration Query

Name	DFS			Conflict DFS			ETRE	
	Fail	Out	Finish	Fail	Out	Finish	Out	Finish
AD	0.04	0.29	0.44	0.01	0.27	0.49	0.00	**1.00**
BG	0.05	0.92	0.02	0.03	0.92	0.04	0.00	**0.90**
BK	0.07	0.91	0.09	0.07	0.89	0.11	0.00	**0.94**
DA	0.04	0.63	0.02	0.02	0.72	0.00	0.02	**0.80**
EC	0.01	0.74	0.20	0.00	0.74	0.19	0.00	**1.00**
EP	0.07	0.86	0.08	0.06	0.83	0.11	0.00	**0.98**
FC	0.06	0.84	0.04	0.01	0.83	0.01	0.00	**0.21**
GG	0.03	0.17	0.32	0.01	0.29	0.56	0.01	**0.96**
HS	0.06	0.90	0.04	0.00	0.88	0.00	0.00	**0.38**
ND	0.10	0.04	0.49	0.06	0.21	0.58	0.00	**0.99**
SF	0.10	0.20	0.41	0.08	0.29	0.62	0.00	**0.99**
SO	0.00	0.95	0.02	0.00	0.96	0.01	0.04	**0.02**
TR	0.01	0.82	0.03	0.01	0.81	0.04	0.00	**0.97**
YT	0.16	0.96	0.01	0.00	0.00	**1.00**	0.00	**1.00**
ZS	0.12	0.58	0.21	0.09	0.47	0.49	0.00	**0.98**

The faster search time of ETRE and more paths that ETRE outputs can be attributed to a key factor. Although the time taken to find the first result is similar between ETRE and Yen's algorithm, the cost of progress before finding the next result differs significantly. Yen's algorithm algorithm requires deleting some edges in the graph

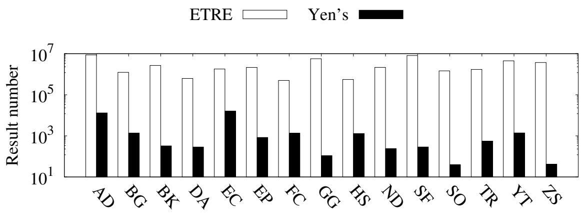


Figure 6: The result number comparison.

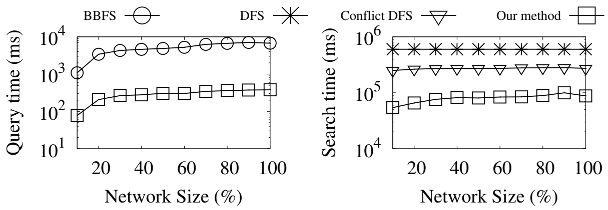


Figure 7: Time comparison on WD having different sizes. The left is for the reachability query and the right is for the enumeration query.

to avoid repeated results, which incurs a rapidly growing cost as the number of results increases. In contrast, ETRE only needs to unblock some vertices in the graph, resulting in nearly constant cost, which contributes to superior performance.

8.2.2 Comparison with DFS and Conflict DFS. Figure 5 shows the average search time of DFS, Conflict DFS and ETRE. In comparison, ETRE is nearly 1000 times faster than DFS and Conflict DFS on most graphs, as they tend to repeatedly explore unnecessary paths while ETRE avoids these useless explorations as much as possible.

Table 13 presents the search time distribution of three algorithms for Enumeration Query under TRE, where *Fail* indicates that the algorithm is unable to find a path within 10 minutes, *Out* means less than 1000 results are found within 10 minutes, *Finish* denotes that 1000 results are found within 100 seconds. Conflict DFS exhibits a lower percentage of *Fail* and *Out* and a higher percentage of *Finish* compared to DFS on most of the graphs, indicating the effectiveness of conflict sets. ETRE demonstrates higher efficiency than both DFS and Conflict DFS, with negligible percentages of *Out* and over 99% of *Finish*.

8.2.3 Performance against Query-Label Size. We illustrate the impact of the number of labels in the $Type$ on the search time of ETRE in the right of Figure 4. The search time increases linearly with the number of labels, indicating that ETRE is less sensitive to the number of labels in the query regex.

8.3 Scalability Evaluation

We report query and search times for different algorithms on induced subgraphs of the dataset WD, varying sizes for both Reachability Query and Enumeration Query. Figure 7 illustrates that query time for Reachability Query and search time for Enumeration Query generally increase with network size for most algorithms except between 90% and 100% size.

The search space grows up varying graph size increasing. However, we just have to find limited eligible paths (1 path for the reachability problem and 1000 paths for the enumeration problem). More edges may result in more eligible paths in the search space. DFS and Conflict DFS have little increase since they cannot finish most of the queries within the time limit, so we record their search time as 10 minutes to evaluate performance. Overview, the query time of our methods increases linearly and our methods (i.e., RTRE and ETRE) outperform the baselines (i.e., BBFS and DFS).

8.4 Memory Evaluation

We construct your suggested experiments and add the result to our revised paper. We test the extra memory requirements for 6 types of queries on 15 datasets. We do not compute the graph memory cost since all methods must read the graph. Due to the space limit of the paper, we only add the result on Q_6 to our revised paper and other results can be seen in the long vision of our paper.

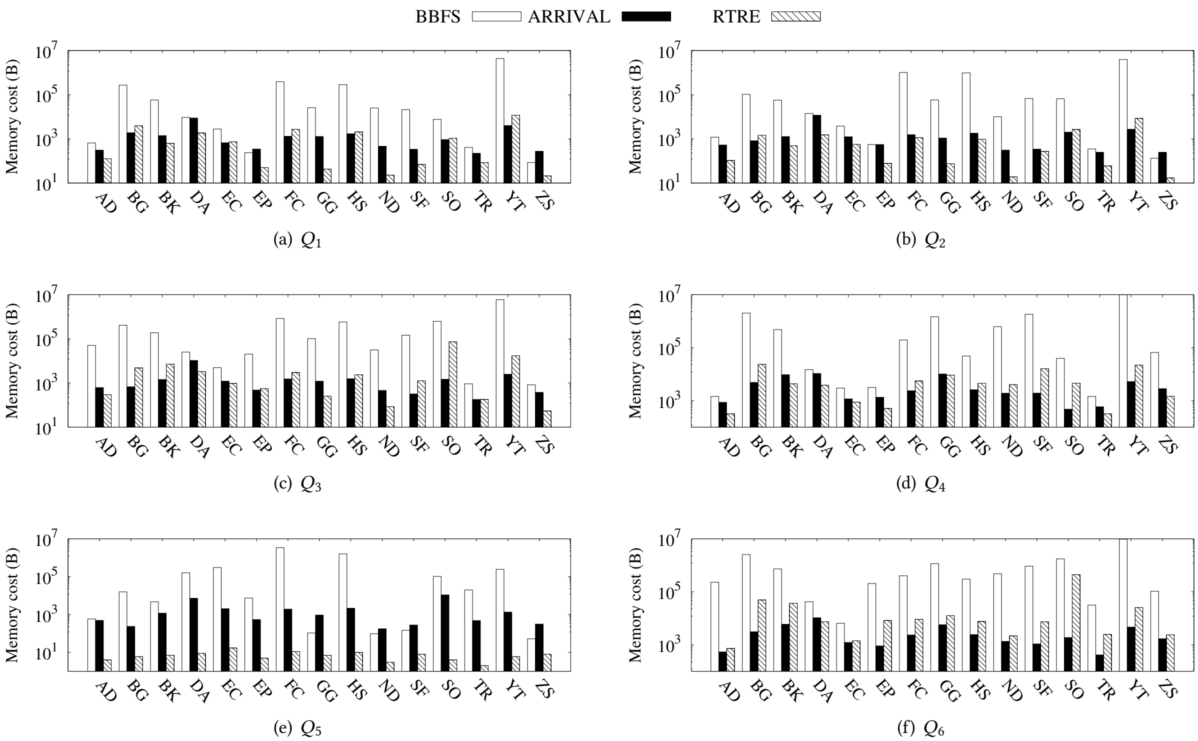
For Reachability Query, Figure 8 demonstrates that BBFS costs the most extra memory since it needs to contain all the potential simple paths that require lots of memory. In general, RTRE requires similar memory to ARRIVAL since RTRE and ARRIVAL use two different methods to avoid saving all the paths as much as possible. RTRE makes sure every edge is explored only once, and it only records whether the vertices are visited in the forward and backward exploration. ARRIVAL limits the search length and the number of paths and returns an approximate result. Hence, RTRE may cost an extra $O(n \times k)$ space to record the information in bad cases while ARRIVAL only needs to record a few paths. Sometimes, ARRIVAL may explore lots of repeated paths whereas RTRE can avoid these useless explorations, resulting in less memory cost.

For Enumeration Query, when the expression does not contain $Type$ ETRE executes DFS for other parts. So DFS and ETRE show the same memory cost in Q_5. For the other five queries, Figure 9 illustrates that DFS needs the least extra space as it only records the current exploring path which is related to the depth of simple paths. DFS and Yen's algorithm need more memory since they need BFS exploration and record extra information (such as the information about candidate vertices). ETRE costs the most extra memory because it does not only record the candidate vertices but also records the information about when the block vertices can be unblocked. Please note the unit of memory cost is Byte, so even if ETRE costs most space, its cost is less than 100 MB, which is a reasonable space cost.

8.5 Reachability Vertices Pair Query Evaluation

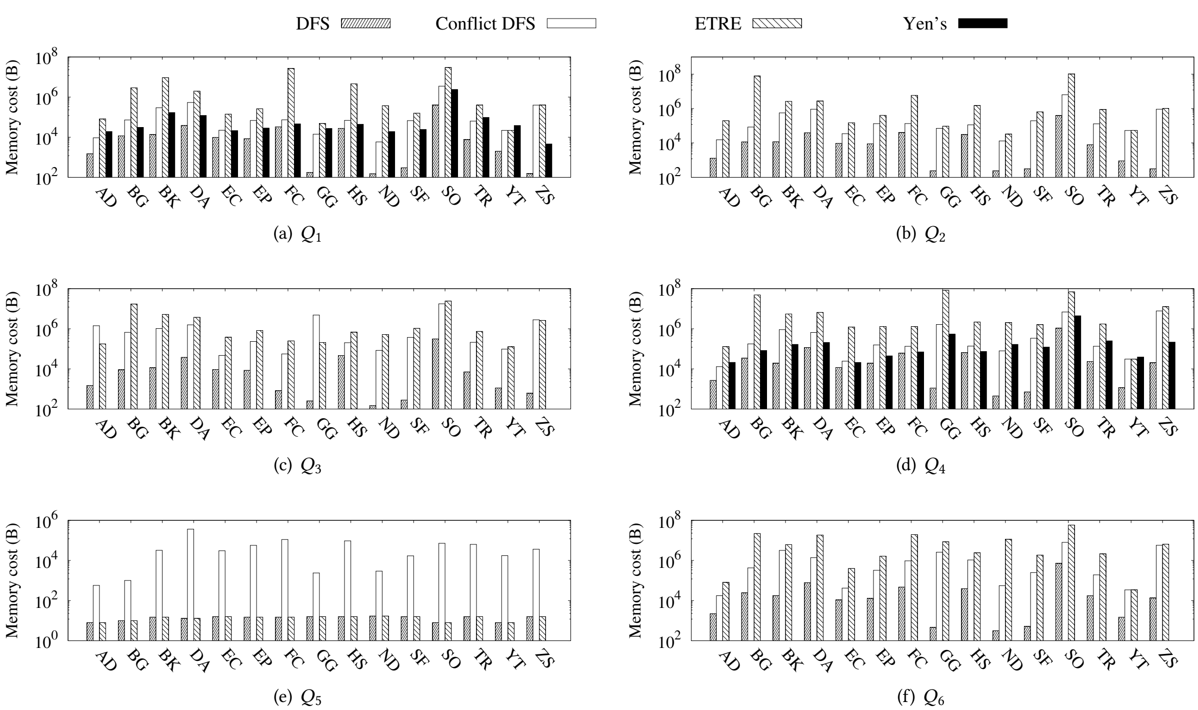
We construct the experiments to test the efficiency of our method comparison with Wood's algorithm. As some queries may be challenging instances that cannot be completed within a reasonable time, we set a time limit of 1 hour for each query. A query is considered out of time if it does not finish within the time limit, and we record its query time as 1 hour for evaluation purposes.

We compare the performance of our method with Wood's algorithm on two datasets (i.e., AD and EC). Figure 10 demonstrates that our method has comparable efficiency to Wood's algorithm on Q_1, Q_4, and Q_5, and both of them can be completed within 1 hour. However, our method is up to three orders of magnitude faster than Wood's algorithm on Q_2 and Q_3. It is because although these two types are very simple, Wood's algorithm takes exponential time to evaluate the queries while our method can finish these queries in polynomial time. Unfortunately, both of these two algorithms are



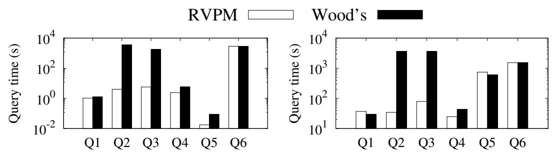
(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4 (e) Q_5 (f) Q_6

Figure 8: Memory requirement comparison on Reachability Query.



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4 (e) Q_5 (f) Q_6

Figure 9: Memory requirement comparison on Enumeration Query.



(a) AD (b) EC

Figure 10: Query time comparison.

out of time in Q_6 since it will take exponential time to evaluate the random queries under TREs.

9 RELATED WORK

Regular Path Query (RPQ): Most of the existing works focus on Reachability Vertices Pair Query. Koschmieder and Leser [20] use the rare label to improve efficiency. Wang et al. [42] answer regular path queries by evaluating partial answers evaluation. Jachiet et al. [17] propose a variation of the relational algebra to solve this problem. Other methods [13, 21, 24] for RPQ evaluation construct indexes to optimize RPQ evaluation. Na et al. [29] propose a light-weight transitive closure to evaluate RPQs. Arroyuelo et al. [4] evaluate RPQs with high space efficiency. Pacaci et al. [30] focus on evaluating RPQs on streaming graphs. We do not compare these works since they do not consider simple path semantics.

Simple Path Enumeration: Birmele et al. [6] investigate the s-t path enumeration problem in undirected graphs, which cannot extend to directed graphs. Tarjan [37] and Johnson [19] propose algorithms for enumerating simple cycles in directed graphs. A recent work [22] proposes an efficient algorithm for enumerating all simple temporal cycles in temporal graphs. These methods cannot extend straightly to solve RSPQs since label information should be considered. Recent researches focus on hop-constrained s-t simple path enumeration. Several theoretical works [15, 35] achieve polynomial delay. Peng et al. [32] propose BC-DFS and JOIN, achieving high efficiency. Sun et al. [36] also propose an index-based method for real-time enumeration of hop-constrained s-t simple paths. HP-index [34] maintains paths between hot vertices and enables real-time detection of hop-constrained cycles in large dynamic graphs. These algorithms are not suitable for solving the RSPQs as they are based on the length constraint whereas almost of RSPQs do not limit the length of paths. The purpose of the above methods is similar to ours. We all want to achieve polynomial delay and avoid unnecessary paths as much as possible. However, our method is based on an efficient pruning technique, which contributes to exploring the useless paths only once, and it considers label constraints as well as avoids length limitations.

10 CONCLUSION

In this paper, in order to address two fundamental problems of RSPQs efficiently, we summarize a type of regular expression that covers more than 99% of real-world queries. Then, we propose an efficient algorithm framework to solve both Reachability Query and Enumeration Query. Our experimental analyses on extensive datasets demonstrate that our methods has comparable efficiency to the approximate method and outperforms significantly the state-of-the-art exact methods.

REFERENCES

[1] Serge Abiteboul and Victor Vianu. 1997. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 122–133.

[2] Abdulellah A. Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z. Berkay Celik, X. Zhang, and Dongyan Xu. 2021. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *USENIX Security Symposium*.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[4] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2021. Time-and Space-Efficient Regular Path Queries on Graphs. *arXiv preprint arXiv:2111.04556* (2021).

[5] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 261–272.

[6] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. 2013. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1884–1896.

[7] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.* 11, 2 (2017), 149–161.

[8] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the maze of Wikidata query logs. In *The World Wide Web Conference*. 127–138.

[9] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 1999. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 194–204.

[10] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. *ACM SIGMOD Record* 16, 3 (1987), 323–330.

[11] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. 2013. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 214–223.

[12] Brendan Elliott, Mustafa Kirac, Ali Cakmak, Gokhan Yavas, Stephen Mayes, En Cheng, Yuan Wang, Chirag Gupta, Gultekin Ozsoyoglu, and Zehra Meral Ozsoyoglu. 2008. PathCase: pathways database system. *Bioinformatics* 24, 21 (2008), 2526–2533.

[13] George HL Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. (2016).

[14] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying.. In *EDBT*. 520–523.

[15] Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Efficient algorithms for listing k disjoint st-paths in graphs. In *Latin American Symposium on Theoretical Informatics*. Springer, 544–557.

[16] Xueyuan Han, Thomas Pasquier, Adam Bates, James W. Mickens, and Margo I. Seltzer. 2020. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *NDSS*.

[17] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 681–697.

[18] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 123–134.

[19] Donald B Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1975), 77–84.

[20] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *International Conference on Scientific and Statistical Database Management*. Springer, 177–194.

[21] Jochem Kuijpers, George Fletcher, Tobias Lindaaker, and Nikolay Yakovets. 2021. Path Indexing in the Cypher Query Pipeline.. In *EDBT*. 582–587.

[22] Rohit Kumar and Toon Calders. 2018. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1441–1453.

[23] Ulf Leser. 2005. A query language for biological networks. *Bioinformatics* 21, suppl_2 (2005), ii33–ii39.

[24] Baozhu Liu, Xin Wang, Pengkai Liu, Sizhuo Li, and Xiaofei Wang. 2021. Pairpq: An efficient path index for regular path queries on knowledge graphs. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 106–120.

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[26] Wim Martens and Tina Trautner. 2018. Evaluation and enumeration problems for regular path queries. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[27] Alberto O Mendelzon and Peter T Wood. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

[28] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat Venkatakrishnan. 2019. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019).

[29] Inju Na, Yang-Sae Moon, Ilyeop Yi, Kyu-Young Whang, and Soon J Hyun. 2022. Regular path query evaluation sharing a reduced transitive closure based on graph reduction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1675–1686.

[30] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.

[31] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825.

[32] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Hop-constrained st Simple Path Enumeration: Towards Bridging Theory and Practice. *Proc. VLDB Endow.* 13, 4 (2019), 463–476.

[33] Stefan Plantikow, Mats Rydberg, and Petra Selmer. [n.d.]. CIP2017-01-18 Configurable Pattern Matching Semantics. https://github.com/boggle/openCypher/blob/isomatch/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc#paths

[34] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[35] Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. 2014. Efficiently listing bounded length st-paths. In *International Workshop on Combinatorial Algorithms*. Springer, 318–329.

[36] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained st Path Enumeration. In *Proceedings of the 2021 International Conference on Management of Data*. 1758–1770.

[37] Robert Tarjan. 1973. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.* 2, 3 (1973), 211–216.

[38] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[39] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 345–358.

[40] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[41] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1463–1480.

[42] Xin Wang, Junhu Wang, and Xiaowang Zhang. 2016. Efficient distributed regular path queries on RDF graphs using partial evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1933–1936.

[43] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *management Science* 17, 11 (1971), 712–716.

[44] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems* 40 (2014), 47–66.