Textile & Leather Review

ISSN 2623-6281| www.tlr-journal.com | 10.31881/TLR

Efficient Algorithm for Path Enumeration in Heterogeneous Information Networks

Jing Zha, Yiqing Lian, Yufeng Hu, Yiyue Yang, Ju Hu

How to cite: Zha J, Lian Y, Hu Y, Yang Y, Hu J. Efficient Algorithm for Path Enumeration in Heterogeneous Information Networks. Textile & Leather Review. 2026; 9:4371-4390. https://doi.org/10.31881/TLR.2026.4371

How to link: https://doi.org/10.31881/TLR.2026.4371

Published: 25 April 2026

Efficient Algorithm for Path Enumeration in Heterogeneous Information Networks

Jing Zha[1*], Yiqing Lian[2], Yufeng Hu[2], Yiyue Yang[1], Ju Hu[2]

[1]China Southern Power Grid Digital Network Research Institute Co. LTD, Guangzhou 510700, Guangdong, China

[2]China Southern Power Grid Company Limited, Guangzhou 510700, Guangdong, China

*zhajing@csg.cn

Article

https://doi.org/10.31881/TLR.2026.4371

Published 25 April 2026

ABSTRACT

Regular simple path enumeration is a fundamental problem in heterogeneous information networks (HIN), with important applications in finance, security, and neuroscience, and textile structure analysis. In textile systems, different yarn types (e.g., warp, weft, and conductive fibers) can be naturally modeled as heterogeneous entities, while their interactions (e.g., over, under, and contact relations) form heterogeneous links, making HINs a suitable representation for complex fabric structures and conductive textile pathways. In this work, we study the regular simple path enumeration problem. A straightforward approach is to use a search-based algorithm to enumerate all paths with regular constraints and verify path simplicity. Large, densely connected graphs present unique challenges. In this paper, we propose an efficient graph relabeling algorithm to reduce computation. We also develop new techniques to improve performance. These techniques enhance reachability detection, optimize candidate vertex search, and minimize redundant path computation. Extensive experiments demonstrate that our method can compute the exact result for different kinds of regular path queries efficiently.

KEYWORDS

heterogeneous information networks, path enumeration, DFS algorithm, structural analysis

INTRODUCTION

Heterogeneous Information Networks (HINs) have emerged as a powerful framework for modeling complex systems where entities and relationships are of multiple types, such as social networks, bibliographic databases, and biological systems. Furthermore, this framework is increasingly recognized as a potent tool for modeling complex structures In such systems, different yarn types (e.g., warp, weft,

conductive fibers) act as heterogeneous entities, and their physical interactions (e.g., 'over', 'under', 'contact') form heterogeneous relationships. A fundamental problem in HINs is regular simple path enumeration, which involves identifying all valid paths between two nodes that adhere to both structural constraints (simple paths with no repeating vertices) and semantic constraints (paths matching a given regular expression). This problem has critical applications in fraud detection, recommendation systems, knowledge graph exploration, and neuroscience [1–4].

The regular path queries (RPQs) provide a flexible formalism for specifying such constraints. By expressing permissible edge sequences as regular expressions, RPQs enable declarative querying of diverse patterns of interest. While existing work has largely focused on evaluating RPQs for reachability—deciding whether two vertices are connected by a path conforming to a given expression—the problem of enumerating all regular simple paths that satisfy a constraint remains underexplored. This gap is significant: in many applications, analysts and downstream algorithms require not only a yes/no answer but also the complete set of valid paths for further reasoning or ranking. This paper proposes a general, efficient algorithmic framework to address this precise class of enumeration problems. Regular path query enumeration (RPQE) problem is to detect all valid paths.

Applications. RSPEs can be applied in many real-world scenarios. Some examples are the following. Knowledge Retrieval in Knowledge Graphs. RSPQs are building blocks in many graph query languages such as PGQL [5] and openCypher. Similarly, PGQL also supports regular simple path queries [6,7]. In bibliographic knowledge graphs, many information retrieval tasks can be naturally expressed as regular simple path queries. For example, suppose we want to determine whether an author A has published a paper in the SIGMOD conference. This task can be formulated as a reachability query (A, SIGMOD, (Wrote◦ PublishedIn)), where Wrote denotes the authorship relation between an author and a paper, and PublishedIn specifies the venue of publication. To account for hierarchical structures among venues (e.g., workshops belonging to conferences), the expression can be extended to (Wrote◦ PublishedIn). If we can find a simple path such as A($^{\text{Wrote->}}$) p($^{\text{PublishedIn->}}$) SIGMOD, then we can conclude that author A has published at SIGMOD. In textile domain, yarns of different types, such as warp, weft, and conductive fibers, can be naturally modeled as heterogeneous entities, while their interactions, including over, under, and contact relations, can be represented as heterogeneous edges. Under this formulation, many structural analysis tasks in textiles can be expressed as regular simple path queries.

For example, conductive pathways in smart textiles, repeated weave patterns, and local structural anomalies can all be modeled and discovered through constrained path enumeration.

APT attack in Computer Network Traffic. In the cybersecurity domain, detecting advanced persistent threats (APTs) remains a critical challenge [8-10].

In this work, we address these challenges by proposing an efficient framework for Regular Simple Path Enumeration in HINs. Our approach integrates three key ideas: (i) translating the regular expression into a deterministic finite automaton (DFA) to guide path exploration; (ii) leveraging the network schema to relabel edges and reduce branching complexity; and (iii) dynamically maintaining reachability information within the product graph of the HIN and DFA to prune infeasible search directions. Together, these innovations enable both full path enumeration and reachability checking within a unified framework.

In this paper, for the RSPE problem in HIN, we conduct extensive experiments on large-scale real-world datasets, including Amazon, DBLP, DoubanBook, AMiner, and Yelp. The results demonstrate that our method significantly outperforms baseline approaches in both runtime efficiency and scalability, while preserving completeness.

In summary, this paper makes the following contributions:

• We introduce the regular simple path problem in heterogeneous information networks.

• We develop a DFA-guided DFS algorithm enhanced by schema-based relabeling and dynamic reachability pruning, which jointly achieve efficient and complete enumeration.

• We empirically validate our framework on multiple large-scale datasets, showing superior performance compared to the competitors.

PRELIMINARIES

In this section, we first formally introduce the definition of a heterogeneous inform- ation network (HIN). Then we give the problem statement of the regular simple path enumeration query.

A heterogeneous information network (HIN) is an information network composed of multiple types of entities and relationships, enabling the unified modeling of behavioral dynamics and attribute characteristics.

Definition 1 (Heterogeneous Information Network)**.** An information network is a directed graph G = (V, E) with a vertex mapping function $\varphi : V \rightarrow A$ and an edge mapping function $\psi : E \rightarrow B$, where each vertex v

$\in$ V belongs to a particular label $\varphi(v) \in A$, and each edge e $\in$ E belongs to a particular label $\psi(e) \in B$. When the types of vertices satisfy $|A| > 1$ or the types of edges satisfy $|B| > 1$, the network is a heterogeneous information network.

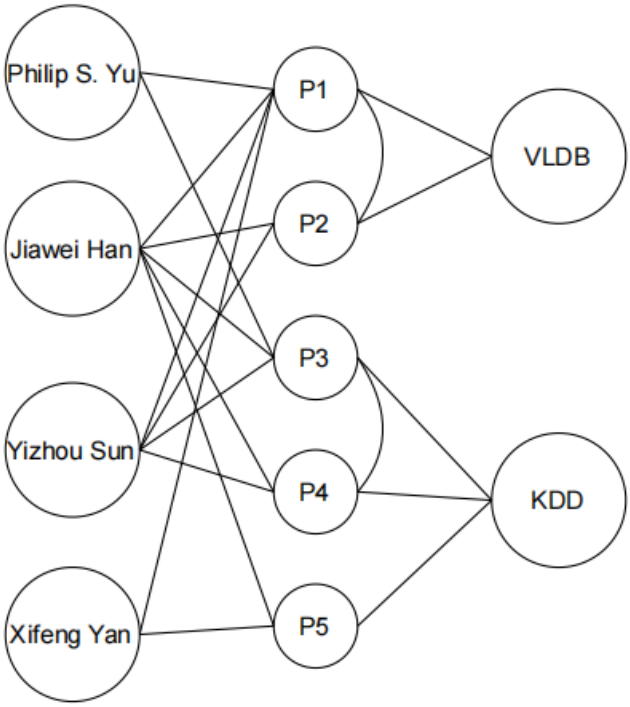

Figure 1. A heterogeneous information network

Given two vertices $s, t \in V$, a path p from s to t in G is a sequence of edges : $\langle(v0, e0, v1), \cdots, (vn-1, en-1, vn)\rangle$ where $v0 = s$, $vn = t$. If there is no repeating ver- tex in the path p, we say that p is a simple path. The label of a path p is denoted by $LP(p) = \varphi(v0)\psi(e0)\varphi(v1)\psi(e1)\cdots\psi(en-1)\varphi(vn)$.

Example 1. Figure 1 is a heterogeneous information network that contains 11 vertices. The left 4 vertices present authors. The medium 5 vertices present papers. The right 2 vertices present conferences. The edges between authors and papers indicate that the paper belongs to the author.

The edges between papers and conferences indicate that the paper is published in the conference. The edges between the papers indicate that the paper cites the other one. When we have a query to find the common papers between two authors, we have the query (Jiawei Han, Yizhou Sun, author, author(Wrote->)paper(<-Wrote)author) to find the path between two given authors following the type constraint of vertices and edges.

Definition 2 (Regular Expression). A regular expression R over the alphabet A is defined as R ::= ϵ | α | R1 ◦ R2 | R1 + R2 | R∗ , where (i) ϵ denotes the empty string, (ii) α ∈ A denotes a character in the alphabet, (iii) ◦ denotes the concatenation operator, (iv) + denotes the alternation operator, (v) R, R1 and R2 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 can be described by R, i.e., LP(p) ∈ L(R).

Problem Statement. Given a heterogeneous graph G = (V, E, φ, ψ) and a regular constraint R, the regular simple path enumeration (RSPE) problem aims to find all valid simple paths P with respect to the regular constraint.

Example 2. Figure 2 is a heterogeneous information network with new vertex and edge labels. v1 and v2 present conference. The conference label is c. v3v4v5v6v7 are paper vertices with label b. v8v9v10v11 are author vertices with label a. For the query (v9,v10, a1b(2c2b) ∗ 1a) is to find all simple paths that from v9 to v10 by the regular expression a1b(2c2b) ∗ 1a constraint. For example, $p = (v_9 \xrightarrow{1} \text{v3} \xrightarrow{2} \text{v1} \xrightarrow{2} \text{v4} \xrightarrow{1} v_{10})$ is a valid result.

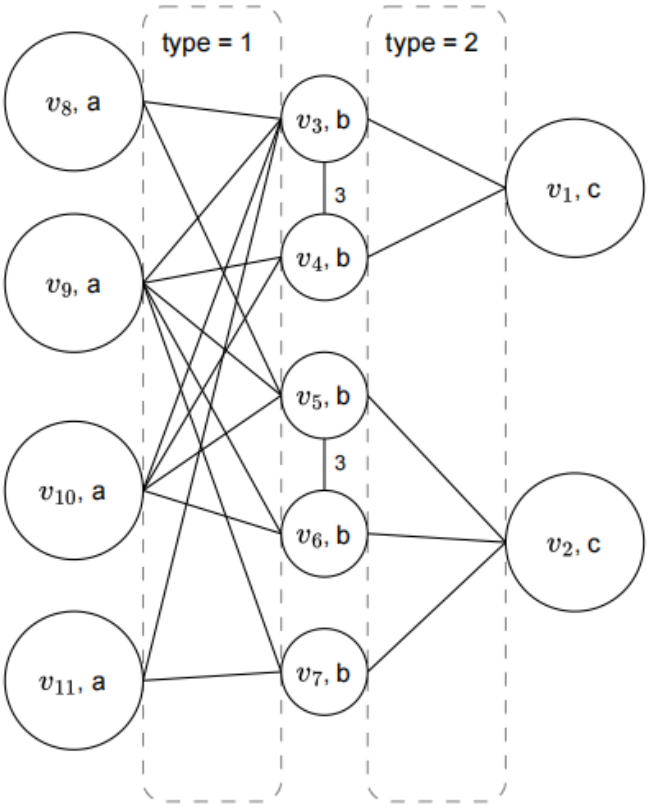

Figure 2. A heterogeneous information network with new labels

BASELINE FOR REGULAR SIMPLE PATH ENUMERATION

Depth First Searching Method in HIN

For a HIN G, and the query (s, t, R), the straightforward way to find regular simple path is searching all potential paths and detecting the results. However, this method is inefficient, because the detected path is not simple or conformed to regular constraint. Depth-First-Search (DFS) is a straightforward method to enumerate all paths. We can add a simple constraint checking in DFS. In order to follow the regular constraint, we can use DFS to search on HIN and check the regular constraint for the current part paths. In order to reduce search branches in DFS, we use two checking rules for simple path constraints and regular path constraints. For simple path constraints, we set a boolean array for every vertex in the vertex set V.

For the regular path constraints, the deterministic finite automaton (DFA) is a good tool. DFA functions as the foundational execution engine for regular expressions by efficiently deciding whether a given input string belongs to the language described by a regex. By systematically reading each symbol, the DFA transitions deterministically through its states, and if it ends in an accepting state after processing the entire input, the string is considered a match. This deterministic behavior—where each state and input symbol yields exactly one next state—enables linear-time matching, offering consistently fast and unambiguous performance for pattern recognition tasks

Definition 3 (Deterministic Finite Automaton)**.** Given a regular expression R, its corresponding deterministic finite automaton (DFA) is defined as $A = (S, A, \delta, \alpha 0, F)$, where S is a set of states, A is the input alphabet, $\delta : S \times A \rightarrow S$ 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. 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.

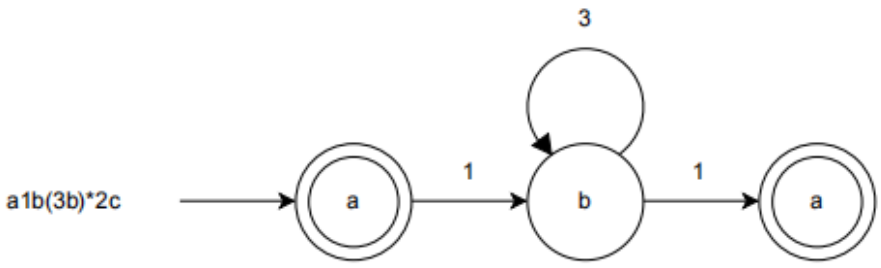

Figure 3. DFA

For regular path constraints, we build a DFA for the regular constraint. First, we set the current regular position at the start state of the DFA. Then, we search for a new vertex and a new edge. If the state is under constraints in DFA, we move to the new state. Finally, the DFS search prints the valid regular simple path if we find the terminating vertex t and the state in DFA is at the terminate state. The deterministic finite automaton can be implemented using the Thompson construction method. This involves first performing lexical analysis on the regular expression to parse out basic regular expression units, such as characters, concatenation operators, and closures. The Thompson construction method is then used to gradually construct a non-deterministic finite automaton (NFA) from the parsed regular expression units. In this method, a corresponding NFA sub-structure is created for each regular expression unit. Finally, the subset construction method is used to convert the NFA into a DFA.

Algorithm 1 presents the algorithm in detail. The presented DFS algorithm operates on a product structure formed by combining a heterogeneous information network (HIN) G = (V, E, φ, ψ), a DFA and a target regular expression R.

Lines 1-3 initialize the path vector p and the visited flag array. A depth-first search (DFS) is performed from each edge (Line 4). The DFS procedure described in Lines 5-14. DFS has 4 parameters. u is current vertex in G, α is current state in A. t is the terminal vertex in HIN. F is the terminal state in DFA. We add the current vertex u to the current path p (Line 6). We mark u as visited (Line 7). If the current state α belongs to F and the current vertex u is equal to t, we find a path from s to t complying with the regular expression constraint (Lines 8-9). We find the neighbor v of u (Line 10). If the label of edge (u, v) and vertex v are complying with the regular expression (Lines 11-12), we can check the simple constraint (Line 13). If all constraints are complied, we can DFS at vertex v(Lines 14). After DFS all neighbors, we pop u out of the current vertex and make the visit stamp of u to false (Lines 15-16).

Example 3. Given the HIN of Figure 2 and the starting point $(v9, a0)$, we start the DFS with the query $(v9, v10,$ a1b(2c2b)* 1a). We search the neighbor $v3$. Because edge $\psi(v3, v9) = 1$, we can reach the new state b in DFA. Next, we can search the neighbor $v10$. $\psi(v9, v10) = 1$ and we can reach state a1, which is the terminal state. The path $p = (v9, v3, v10)$ is found as a valid result. The paths $(v9, v3, v1, v4, v10)$, $(v9, v4, v10)$, $(v9, v4, v1,$ $v3, v10)$, $(v9, v5, v10)$, $(v9, v5, v2, v6, v10)$, $(v9, v6, v10)$, $(v9, v6, v2, v5, v10)$ are valid paths.

IMPROVED SOLUTION

Relabel HIN Searching Algorithm

Because there are labels for both vertices and edges in HIN, we have to check the repeated vertex and the regular constraints when we meet any new vertex. The search space may be very large. Reducing the checking cost may make the algorithm processing efficiently. Next, we discuss how to accelerate constraints checking by constructing product meta graph.

For a HIN G, every vertex and an edge is assigned a label. When we combine the vertex with the same label, we can construct a schema graph to present the relations between vertices with different labels. Here, we introduce the definition of the schema graph.

Algorithm 1: DFS regular simple path in HIN

Input: The HIN G=(V, E, ϕ, ψ), DFA A = (S, A, δ, α0, F) source vertex s, target vertex t, the regex R;

Output: return the answer;

1 p $\leftarrow$ $\varnothing$;

2 foreach v $\in$ V **do**

3 |visited[v] $\leftarrow$ false;

4 DFS(s, A.α0, t, A.F) ;

5 Procedure DFS(u, α, t, F)

6 p $\leftarrow$ p $\cup$ {u};

7 visited[u] $\leftarrow$ true;

8 **if** α $\in$ F $\wedge$ u = t **then**

9 print p as a result;

10 **foreach** v $\in$ Nout(u) **do**

11 $\beta \leftarrow \delta(\alpha, \psi(e))$;

12 $\gamma \leftarrow \delta(\beta, \phi(v))$;

13 **if** $\beta, \gamma \in S \wedge$ visited[v]=false **then**

14 |DFS(v, γ, t, F);

15 p $\leftarrow$ p —{u};

16 visted[u] $\leftarrow$ false;

Definition 4 (Heterogeneous Schema)**.** The network schema is a meta-template for a heterogeneous network G = (V, E, ϕ, ψ) with a vertex mapping ϕ : V → A and edge mapping ψ : E → B. The network schema is a directed graph defined over object types A, with edges as relations from B, denoted as TG = (A, B).

The vertex in HIN can be mapped to the vertex in the schema. Hence, any path in HIN can be mapped to the schema. We call it meta path.

Definition 5 (Meta Path)**.** A meta path P is a path defined on the graph of network schema TG = (A, B), and takes the form $A_0 \xrightarrow{B_1} A_1 \xrightarrow{B_2} A_2 \cdots \xrightarrow{B_k} A_k$, *which defines a composite edge* B1 ◦ B2 · · · ◦ Bk between vertex types A0 and Ak, where denotes the concatenation operator on labels of vertices and edges.

In the schema, the labels exist in both vertices and edges. An edge is the relationship between two vertices. Hence, we can map two vertices and an edge to a new label triplet structure (vertex, edge label, vertex).

Definition 6 (triplet label)**.** Given a heterogeneous information network, which is a directed graph G = (V, E) with a mapping function ϕ : V → A, ψ : E → B, we have the corresponding schema TG = (A, B). For each edge e = (u, v) ∈ B, we relabel the edge as e = l. All label of edge e ∈ B, we have the new label set L.

Next, we give the algorithm how to relabel the edges. In Algorithm 2, we initialize the new label set L to empty set (Line 1). For each edge in the schema, we create a new triplet label and put it to L (Lines 2-3). Then we sort the label set L with lexicographical order (Line 4). Actually, any other order can be used here. In order to simplify the computation, we use lexicographical order. Next, we relabel each edge in G (Lines 5-6) and delete the labels of vertices (Lines 7-8). Finally, we return the G with the new labels.

Example 4. Figure 4 is the heterogeneous schema of HIN in 2. There are three kinds of vertices: authors, papers, and conferences. We use a, b and c to present the label. There are three kinds of edges. Type 1 edges are between a and b. Type 2 edges are between band c. Type 3 edges are between b vertices. We relabel these three kinds of edges and delete the labels of the vertices. The relabeled HIN is computed.

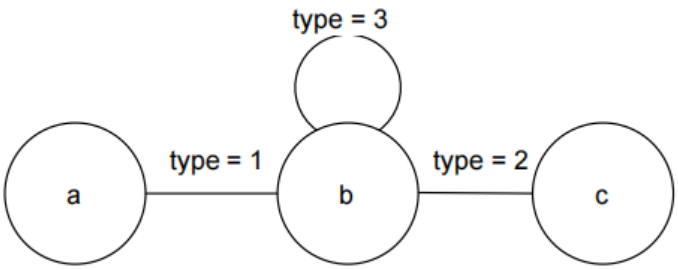

Figure 4. Relabel DFA

Algorithm 2: Relabel

Input: The HIN G = (V, E), TG = (A, R);

Output: The relabel HIN G = (V, E) and schema TG = (A, R);

1 L = ∅;

2 foreach e = (u, v) ∈ A **do**

3 L ← L ∪ {(ϕ(u), ψ(e), ϕ(v))}

4 sort L with lexicographical order;

5 foreach e′ = (x, y) ∈ E **do**

6 ψ(e′) ← the number of (ϕ(x), ψ(e′), ϕ(y)) in L;

7 foreach x ∈ V **do**

8 ϕ(x) ← null;

9 return G, TG;

Algorithm 2 is the relabel algorithm.

Definition 7 (Relabel HIN Product Graph). Given a graph G = (V, E, L, φ) and a DFA A = (S, L, δ, α0, F), the corresponding product graph PG,A is defined as PG,A = (VP, EP), where VP = V × S, EP ⊆ VP × VP, ((u, α), (v, β)) ∈ EP iff (u, v) ∈ E and δ(α, φ(u, v)) = β . Noutp (u, α) = {(v, β) | ((u, α), (v, β)) ∈ EP} denotes the out-neighbors of vertex (u, α) and Ninp (v, β) = {(u, α) | ((u, α), (v, β)) ∈ EP} denotes the in-neighbors of vertex (v, β).

Example 5. For the graph in Figure 2 and the DFA in Figure 3, we construct the HIN product graph in Figure 5. Each vertex in the HIN product graph presents a vertex in HIN and a state in DFA. For example, vertex (v8, a) presents the start vertex is v_8 and the state is a currently. We can find the neighbor (v3, b), which indicates that there is a path from v_8 to v_3 and the state is b in DFA. The path (v8, a) → (v3, b) → (v9, c) presents the path between v_8 and v_9. Meanwhile, the state in DFA is from a to c.

When we have the HIN product graph, we can enumerate the regular simple paths in the graph.

In order to enumerate the simple paths for a given query q = (s, t, R). We have to guarantee the reachability between s and t. The straightforward method to compute the regular simple path is to search all potential paths and detect the results. However, this method is inefficient, because the detected path is not simple or conformed to regular constraint. Depth-First-Search (DFS) is a straightforward method to enumerate all paths. We can add simple constraint checking in DFS. In order to follow the regular constraint, we can use DFS to search on HIN product graph to conform to the

regular constraint. The procedure of DFS in HIN and HIN product graph is similar. For each vertex in HIN, there are multiple vertices in HIN product graph. Hence, in the DFS, we have to check the repeated vertices and comply with the simple constraint.

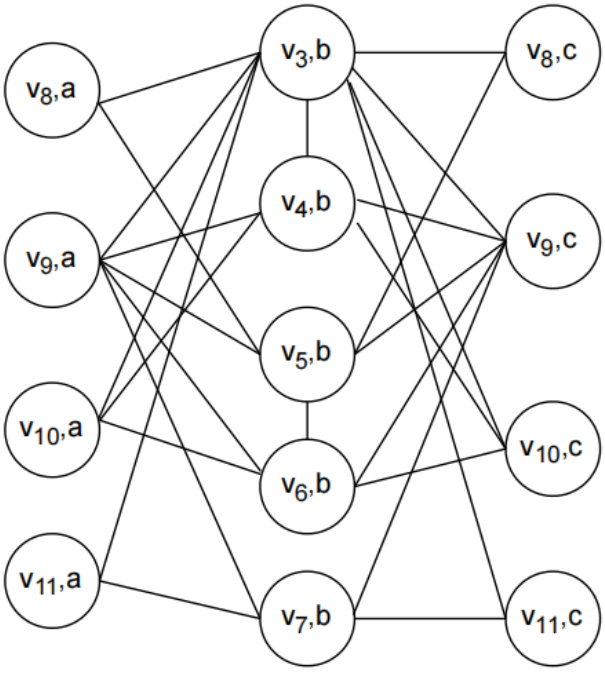

Figure 5. HIN product graph

Example 6. Given a query to find the regular path between $v8$ and $v9$ in the HIN product graph shown in Figure 5, we start the DFS from $v8$ and terminate at $v9$. The state a is the start state and c is the end state. We start our DFS at (v8, a) and search for (v8, a)'s neighbor (v3, b), where we do not meet any vertex twice. Then we detect (v3, b)'s neighbor (v8, c). Although (v8, c) is the end state, but $v8$ has passed before. This is not a simple path and we backtrack to *(v3, b). We detect (v9, c) and this is a simple path. Meanwhile $v9$ is the terminal vertex and c is the end state. We find the valid path $p1$ = ((v8, a), (v3, b), (v9, c)). Next, we backtrack to (v3, b) and detect the neighbors (v10, c) and (v11, c). Because $v10$ and $v11$ are not terminal vertices, we backtrack to (v8, a). We continue DFS and find the next valid path p2 = ((v8, a), (v5, b), (v9, c)).

For both DFS algorithms in HIN or relabeled HIN, the main time consumption is the search space. In order to trim the branch of DFS search tree, we create a mark algorithm to record whether there exists a simple path for each vertex in the HIN product graph.

Reachability Mark

In order to record whether there exists a regular simple path from a vertex to the terminal in HIN product graph, we create a reachability stamp to record it. For each vertex in the relabeled HIN product graph, we assign a boolean variable to indicate whether a potential regular simple path exists from that vertex to the terminal vertex. If no valid path exists, all its neighbors are marked. For instance, the array C(u, α) denotes whether any in-neighbor of vertex (u, α) has a regular simple path to the terminal. When the state of (u, α) is updated, its in-neighbors might gain a new valid path. The initial reachability stamp can be computed by searching for a path between start and terminal vertices in the HIN product graph.

For a given HIN and query, we can create a DFA from the regular expression in the query. Then we construct the HIN product graph by DFA and HIN. When we want to find the regular simple path, we can use the stamp array to determine whether we can find a simple path from the current vertex to the terminal.

Algorithm 3: SRM

Input: product graph P', source and target vertices s, t, Expression R and DFA A;
Output: all simple paths between s and t matching R

```
p ← {s};
B ← ∅;
foreach (u, α) ∈ P′ do
    C[u][α] ← ∅;
return Search(B, s, A.α0, t, p, P′);
Procedure Search(B, u, α, t, p, P′)
    B ← B ∪ {(u, α)} ;
    Result ← ∅ ;
    foreach (v, β) ∈ N_out^P'(u, α) do
        if v = t and β ∈ A.F then
            Result ← Result ∪ {p};
        if v ∉ p and (v, β) ∉ B then
            Result ← Result ∪ Search(B, v, β, t, p ∪ {v}, P′)
    if Result≠∅ then
        check(B, C, u, α);
```

```
16        else
17            foreach (v, β) ∈ N_out^P'(u, α) do
18                C[v][β] ← {(u, α)};
19        return Result
20    Procedure check(B, C, u, α)
21        B ← B − {(u, α)} ;
22        foreach (v, β) ∈ C[u][α] do
23            check(B, C, v, β);
24        C[u][α] ← ∅
```

Algorithm 3 is the search algorithm with stamp checking approach. First, we initialize the current path p, stamp array B and C (Lines 1-4). We execute the search procedure (Line 5). We use 5 parameters in the search procedure. u is the current vertex. We start the search at s. α is the current state. We start at $\alpha 0$, which is the start state in DFA. t is the terminal vertex. p is the current path. P' is the HIN product graph. We record that the vertex (u, α) is detected and put it in B (Line 7). We initialize the current search result to be empty (Line 8). We enumerate each neighbor of (u, α) in HIN product graph (Line 9). If the terminating vertex t is reached, we put the path to the result (Lines 10-11). If the neighbor has not been met before, we continue our search (Lines 12-13). If we find any result at the current vertex (u, α), we have to clean the stamp (Line 15). In the check procedure, we clean the stamp. B records all vertices where there is no valid simple path to the terminal vertex. We delete the state (Lines 21-24). If we do not find a valid result, we record the state (u, α) where there are no valid results (Lines 17-18).

In short, we use the relabeled graph to reduce redundant repeated prefix sub-paths. We use the reachability mark to reduce computation about invalid paths.

EXPERIMENTS

Experimental Setup

Datasets. Table 1 summarizes the basic statistics of the 5 real-world heterogeneous information networks used in our experiments. They are Amazon, Aminer, DoubanBook, DBLP and Yelp. They are social networks.

Comparisons. We evaluate the following methods:

- **DFS** The DFS method in HIN product graph.
- **SRM** Searching algorithm with Reachability Mark.

We obtain the source codes of all compared methods from their respective authors. All algorithms are implemented inC++ and compiled using g++ with-O3 optimization enabled. Experiments are conducted on a Linux machine with a 2.8 GHz Intel Xeon processor and 256GB of RAM. To eliminate the impact of disk latency, all graphs are fully loaded into memory before execution. Our evaluation focuses on in-memory processing, and disk I/O time is not included in the reported results.

Table 1. Datasets used in the experiments. (K = 103, M = 106)

Name	Dataset	|V|	|ε|	vertex label	edge label
AM	Amazon	10m	25m	5	4
AI	Aminer	439k	1002k	5	4
DO	DoubanBook	51k	2225k	7	7
DB	DBLP	37k	174k	5	4
YE	Yelp	31k	488k	5	5

Table 2. Query time in microsecond for Aminer

Query	Query with Start-End Vertices		Start-Only Query		Query without Start-End Vertices	
	DFS	SRM	DFS	SRM	DFS	SRM
Q1	20	283230	36	27256	2006	34168
Q2	21	101134	24	147040	2815	5121567
Q3	32	264297	37	323234	5435	5667305
Q4	28	29997	27	27377	101	29595
Q5	23	22.9	38	38	17944	17902

Table 3. Query time in microsecond for Yelp

Query	Query with Start-End Vertices		Start-Only Query		Query without Start-End Vertices	
	DFS	SRM	DFS	SRM	DFS	SRM
Q1	6460938	35839	122	1905	1461	3733
Q2	6313996	273728	133	15992	1405	206455
Q3	3001560	29082	88	19982	988	319759
Q4	12007655	234347	97	1926	653	2790
Q5	68	69	42	44	1953	1966

Queries. We choose the six frequent queries. Q1=a^* is a single label with Kleene star. Q2=a $\circ$ b^* is Q1 and a previous label. Q3=a $\circ$ b^* $\circ$ c^*. Q4 = $(a1+ a2 \cdots + ak)^*$. Q5=a1 $\circ$ a2 $\cdots \circ$ ak. Q6 is a random regular expression without Kleene star.

We select three types of queries. All types of queries have a regular expression. The first type of query has start and end vertices. The second type of query only has a start query. The third type of query does not have start and end vertices. We generate 1000 queries and compute the average time computation.

Computation Time.

Table 2 shows the computation time comparison on the Aminer dataset. **For Query with Start–End Vertices**, DFS achieves up to 3-4 orders of magnitude better performance than SRM on Q1-Q4, while for Q5 the runtime of the two methods is comparable; **for Start- Only Query**, DFS achieves up to 3-4 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods is also comparable on Q5; **for Query without Start–End Vertices**, DFS achieves up to 1-3 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods remains comparable on Q5.

Table 3 shows the computation time comparison on the Yelp dataset. **For Query with Start–End Vertices**, SRM achieves up to 1-2 orders of magnitude better performance than DFS on Q1-Q4, while for Q5 the runtime of the two methods is comparable; **for Start-Only Query**, DFS achieves up to 1-3 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods is also comparable on Q5; **for Query without Start–End Vertices**, DFS achieves up to 1-3 orders of magnitude better performance than SRM on Q2-Q4, and DFS is significantly faster than SRM on Q1 (even though they are in the same order of magnitude), while the runtime of the two methods remains comparable on Q5.

Table 4. Query time in microsecond for Amazon

Query	Query with Start-End Vertices		Start-Only Query		Query without Start-End Vertices	
	DFS	SRM	DFS	SRM	DFS	SRM
Q1	20	669377	45	677870	946304	1953544
Q2	19	1356876	34	2250875	1019200	60550215
Q3	29	3248888	41	3801717	941589	60909950
Q4	22	1162814	42	622009	1869058	3343411
Q5	27	27	25	26	55	57

Table 5. Query time in microsecond for DoubanBook

Query	Query withStart-End Vertices		Start-Only Query		Query without Start-End Vertices	
	DFS	SRM	DFS	SRM	DFS	SRM
Q1	15002288	79557	245	3642	1860	6853
Q2	3000495	556774	83	169656	1856	631369
Q3	60	384904	52	247291	2562	609966
Q4	6374276	331510	177	3818	111	5018
Q5	407	410	79	80	4676	4699

Table 6. Query time in microsecond for DBLP

Query	Query withStart-End Vertices		Start-Only Query		Query without Start-End Vertices	
	DFS	SRM	DFS	SRM	DFS	SRM
Q1	18	2559	17	2488	1317	4737
Q2	18	12922	18	17523	1306	447183
Q3	24	15158	26	19766	1311	474904
Q4	23	2553	23	2466	686	3538
Q5	24	25	25	28	1370	1411

Table 4 shows the computation time comparison on the Amazon dataset. **For Query with Start–End Vertices**, DFS achieves up to 4-5 orders of magnitude better performance than SRM on Q1-Q4, while for Q5 the runtime of the two methods is comparable; **for Start- Only Query**, DFS achieves up to 4-5 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods is also comparable on Q5; **for Query without Start–End Vertices**, DFS achieves up to 1-2 orders of magnitude better performance than SRM on Q1-Q3. Although they are in the same order of magnitude on Q4, DFS is still significantly faster than SRM, and the runtime of the two methods remains comparable on Q5.

Table 5 shows the computation time comparison on the DoubanBook dataset. **For Query with Start–End Vertices**, SRM achieves up to 1-3 orders of magnitude better performance than DFS on Q1, Q2, and Q4, DFS achieves up to 4 orders of magnitude better performance than SRM on Q3, and for Q5 the runtime of the two methods is comparable; **for Start-Only Query**, DFS achieves up to 1-4 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods is also

comparable on Q5; **for Query without Start–End Vertices**, DFS achieves up to 1-2 orders of magnitude better performance than SRM on Q1-Q3. and the runtime of the two methods remains comparable on Q5. Table 6 shows the computation time comparison on the DBLP dataset. **For Query with Start–End Vertices**, DFS achieves up to 2-3 orders of magnitude better performance than SRM on Q1-Q4, while for Q5 the runtime of the two methods is comparable; **for Start-Only Query**, DFS achieves up to 2-3 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods is also comparable on Q5; **for Query without Start–End Vertices**, DFS achieves up to 1-2 orders of magnitude better performance than SRM on Q1-Q4, and the runtime of the two methods remains comparable on Q5.

RELATED WORK

Liang's method using the block-based algorithm to compute regular expression con- straint simple path under a special constraint [11]. The enumeration of all simple st paths or cycles in static graphs has been extensively studied. Notably, Johnson's algorithm efficiently enumerates simple cycles in directed graphs using DFS with vertex blocking. Several theoretical works have achieved polynomial delay guarantees. Peng et al. introduced BC-BFS with a "never repeat mistakes" strategy, which significantly improved upon earlier approaches. Sun et al. proposed an index-based method for real-time enumeration of hop-constrained st simple paths. The HP-index framework maintains paths between high-degree hop vertices, enabling efficient enumeration of hop-constrained cycles in large dynamic graphs. For dynamic graph scenarios, Zhang et al. developed an efficient partial-path index. Other notable contributions include FPGA-based solutions and distributed algorithms. However, all these methods are specifically designed for hop-constrained settings and cannot be directly applied to temporal cycle enumeration. Blanuša et al. presented scalable parallelizations of state-of-the-art algorithms for enumerating simple, temporal, and hop-constrained cycles. The work compares with parallelized approaches [12-17]. Additionally, several studies address cycle detection in dynamic graphs rather than full enumeration, which lies outside the scope of our problem setting.

CONCLUSIONS

In this work, we study the problem of simple path enumeration ineterogeneous information networks, which is fundamental in applications in real world. As highlighted in our introduction, this

problem is not only prevalent in online social and information networks but also represents a key computational challenge in modeling complex physical systems. We introduce a method transform the HIN to a relabel HIN by meta graph. By this, we prune the label in the vertices. We construct the product graph and design DFS based path enumeration algorithm. We also propose a dynamic reachability mark prune DFS algorithm. We compare these two methods in the experiments in 5 real world datasets.

While our experiments focused on demonstrating scalability and efficiency on large-scale, general-purpose network datasets (like Amazon and Yelp), the core properties of our algorithm are directly applicable to the challenges posed by HIN analysis. The proposed techniques-especially the efficient pruning of infeasible paths.

This work lays a validated algorithmic foundation. A clear and promising direction for future research is the application of this framework to HIN. When the HIN undergoes dynamic changes, how to efficiently update the results based on existing calculations is a worthwhile issue for future research.

Author Contributions

Conceptualization – J.Z., Y.L. and Y.H.; methodology – J.Z., Y.L. and Y.H.; formal analysis – J.Z., Y.L. and Y.Y.; investigation – J.Z. and J.H.; resources – Y.L. and Y.Y.; writing-original draft preparation – J.Z., Y.H. and Y.Y.; writing-review and editing – J.Z. and Y.L.; visualization – J.H.; supervision – J.Z. All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest

The author(s) declared no potential conflicts of interest with respect to the research, author-ship, and/or publication of this article.

Funding

This work was supported in part by Research on Deep Deception Trapping and Attack Reproduction Technologies for Highly Concealed Unknown Threats in New Energy Power Systems (ZBKJXM20240172).

Acknowledgements

Not applicable.

REFERENCES

[1] Cruz IF, Mendelzon AO, Wood PT. A graphical query language supporting recursion. ACM SIGMOD Record. 1987; 16(3):323-330. doi: 10.1145/38714.38749

[2] Abiteboul S, Vianu V. Regular path queries with constraints. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems; 11-15 May 1997; Tucson, AZ, USA. New York, NY, USA: Association for Computing Machinery; 1997. p. 122–133. doi: 10.1145/263661.263676

[3] Angles R, Arenas M, Barceló P, Hogan A, Reutter J, Vrgo˘c D. Foundations of modern query languages for graph databases. ACM Computing Surveys (CSUR). 2017; 50(5):1-40. doi: 10.1145/310403

[4] Pacaci A, Bonifati A, Özsu MT. Regular path query evaluation on streaming graphs. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data; 14-19 June 2020; Portland, OR, USA. New York, NY, USA: Association for Computing Machinery; 2020. p. 1415-1430. doi: 10.1145/3318464.3389733

[5] van Rest O, Hong S, Kim J, Meng X, Chafi H. PGQL: A property graph query language. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES); 2016; San Francisco, CA, USA. New York, NY, USA: Association for Computing Machinery; 2016. p. 1–6. doi: 10.1145/2960414.296042

[6] Plantikow S, Rydberg M, Selmer P. CIP2017-01-18 Configurable Pattern Matching Semantics. San Francisco: openCypher; 2017. Available from: https://github.com/boggle/openCypher/blob/isomatch/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc#cip2017-01-18-configurable-pattern-matching-semantics

[7] Green A, Junghanns M, Kießling M, Lindaaker T, Plantikow S, Selmer P. openCypher: New Directions in Property Graph Querying. In: Proceedings of the EDBT/ICDT 2018 Joint Conference; 26-30 March 2018; Vienna, Austria. New York, NY, USA: Association for Computing Machinery; 2018. p. 520-523. doi: 10.5441/002/edbt.2018.62

[8] Milajerdi SM, Eshete B, Gjomemo R, Venkatakrishnan V. POIROT: Aligning attack behavior with kernel audit records for cyber threat hunting. In: Proceedings of the 2019 ACM SIGSAC Conference on

Computer and Communications Security; November 2019; London, UK. New York, NY, USA: Association for Computing Machinery; 2019. p. 1717-1732. doi: 10.1145/3319535.3363217

[9] Han X, Pasquier T, Bates A, Mickens JW, Seltzer MI. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In: Proceedings of the Network and Distributed System Security Symposium (NDSS); 23-26 February 2020; San Diego, CA, USA. San Diego, CA, USA: Internet Society; 2020. p. 1-15. doi: 10.14722/ndss.2020.24046

[10] Alsaheel AA, Nan Y, Ma S, Yu L, Walkup G, Celik ZB, et al. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In: Proceedings of the 30th USENIX Security Symposium; August 2021; Virtual. Berkeley, CA, USA: USENIX Association; 2021. p. 3005-3022. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/alsaheel

[11] Liang Q, Ouyang D, Zhang F. Yang J, Lin X, Tian Z. Efficient regular simple path queries under transitive restricted expressions. In: Proceedings of the 50th International Conference on Very Large Data Bases; 30 August-1 September 2024; New York, NY, USA. New York, NY, USA: Association for Computing Machinery; 2024. p. 1710-1722. doi: 10.14778/3654621.3654636.

[12] Rao VB, Murti V. Enumeration of all circuits of a graph. In: Proceedings of the IEEE; April 1969; New York, NY, USA: IEEE; 1969. p. 700-701. doi: 10.1109/PROC.1969.7032

[13] Johnson DB. Finding all the elementary circuits of a directed graph. SIAM Journal on Computing. 1975; 4(1):77-84. doi: 10.1137/0204007

[14] Mateti P, Deo N. On algorithms for enumerating all circuits of a graph. SIAM Journal on Computing. 1976; 5(1):90-99. doi: 10.1137/0205007

[15] Ponstein J. Self-avoiding paths and the adjacency matrix of a graph. SIAM Journal on Applied Mathematics. 1966; 14(3):600-609. doi: 10.1137/0114051

[16] Tarjan R. Enumeration of the elementary circuits of a directed graph. SIAM Journal on Computing. 1973; 2(3);211-216. doi: 10.1137/0202017

[17] Tiernan JC. An efficient search algorithm to find the elementary circuits of a graph. Communications of the ACM. 1970; 13(12):722-726. doi: 10.1145/362814.362819