Polynomials for Multidimensional Provenance in Graph Databases

Tianyi Liu

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Science (Computer Science)
Concordia University
Montréal, Québec, Canada

July 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Tianyi Liu**

Entitled: **Polynomials for Multidimensional Provenance in Graph Databases**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

_____ Chair
Dr. Joey Paquet

_____ Examiner
Dr. Joey Paquet

_____ Examiner
Dr. Gregory Butler

_____ Supervisor
Dr. Gösta Grahne

_____ Supervisor
Dr. Nematollaah Shiri

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2021 _____
Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Polynomials for Multidimensional Provenance in Graph Databases

Tianyi Liu

In this thesis, we study the provenance of querying graph databases. Compared to using the semiring of polynomials as the most general form of provenance for relational databases (Green, Karvounarakis, & Tannen, 2007), we show that the most general provenance for querying graph databases can be represented by regular expressions over paths in the database. In this work we focus on the single-source provenance, which is a more general representation and contains more information than the single-source, single-target problem considered in (Ramusat, Maniu, & Senellart, 2018). We present an algorithm that computes single-source multidimensional provenances for graph databases, where each dimension represents an application provenance semiring, and also propose a potential application, by using parse tree techniques and deriving results for various application provenances.

Acknowledgments

This work would not have been possible without the financial support of FRS and NSERC. I am especially indebted Dr. Grahne and Dr. Shiri, who have guided me through the darkness and been supportive of my research goals and who worked actively to provide me with the exclusive academic time to pursue those goals.

I also want to thank my partner Xianen, my friends Miya, Sandeep, Aria. They encouraged me and gave me a lot of advice that I really appreciate.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For a few decades Graph Databases have been one of the trending areas in technology academically and industrially. The relationship is considered as the first priority in graph databases, where edges can be labeled, directed with more information, while in relational databases, relationships are implied, and they are less suitable for operations in, for example, the World Wide Web, since relational databases might involve a large number of joins and lead to a high cost. The characteristics of graph databases make them suitable in many applications nowadays, such as the Semantic Web (Hayes & Gutierrez, 2004; Arenas & Pérez, 2011), bioinformatics (Fabregat et al., 2018), and road transportation networks (Añez, De La Barra, & Pérez, 1996), etc. Various query languages can be applied to graph database search, including regular path queries (RPQ's) (Barceló Baeza, 2013), which select nodes connected by paths, of which the word matches the regular language over the labeling alphabet.

The provenance of a query result is used to record the lineage information of

the query result, and also the processes and methodology by which it was produced. There is already existing research working on the provenance of querying relational databases. Among them, (Green, Karvounarakis, & Tannen, 2007) provides a general form of provenance with semirings, which are algebraic structures $(K, +, \cdot, 0, 1)$ such that $(K, +, 0)$ and $(K, \cdot, 1)$ are commutative monoids, using polynomial and formal power series annotations to record the information about why and how query results are generated. However, little research has been done with the provenance of querying graph databases (Ramusat et al., 2018; Ramusat, Maniu, & Senellart, 2021). Therefore we are trying to follow the framework of provenance semirings from (Green, Karvounarakis, & Tannen, 2007) to enrich the semantics of the provenance of query results in graph databases.

1.1 Problems and Motivation

The differences between relational databases and graph databases indicate that we cannot apply the provenance for querying relational databases to graph databases directly. Since most operations (except for set difference operation) are commutative and also the sets in relational databases are unordered, the semirings mentioned in (Green, Karvounarakis, & Tannen, 2007) are commutative semirings. The most general form of the provenance of the relational database query result is the semirings of polynomials (Green, Karvounarakis, & Tannen, 2007).

In comparison, one of the most important operations in graph databases, the

concatenation, should satisfy certain orders and is not commutative. Therefore, the type of semiring we can use to represent the provenance of querying graph databases will be different, and that leads us to the first problem we want to solve in this thesis, which is what is the most general form of provenance for RPQ's over graph databases?

Moreover, one of the characteristics of graph databases is that they allow multiple properties over edges, which means a graph database may include multiple weights for different dimensions, where the weight of a dimension is one specific application value. However, each time when we want to compute the value from one dimension, we need to query the graph database and get that value, for example, to get the shortest paths from a source node to a target node. If next time we want to use the same query and compute the value from a different dimension, we need to query the graph database again, since the query results do not include the edge information. Therefore, another question we would like to solve is to find a way to include multiple dimensions information in the provenance to compute different values for different purposes without recomputing the querying result.

To bring more light to it, (Hangal, MacLean, Lam, & Heer, 2010) observed that the influence of one user over another is asymmetric, and most contemporary networks, such as LinkedIn, only return the source-target path with the shortest distance, which can not reflect social tie strength and influence. This problem implies the potential need for multidimensional provenance.

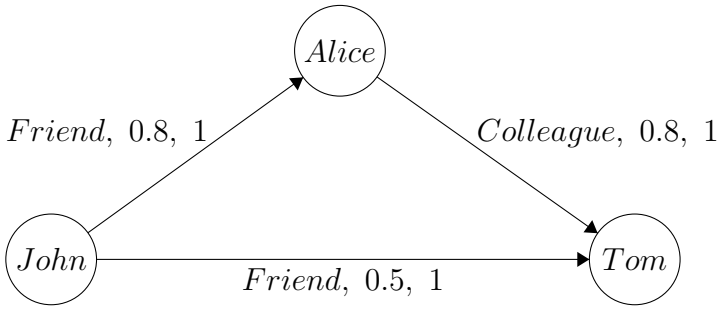


Figure 1: A social network

Figure 1 is part of a social network, which has three nodes representing users, and edges representing the relationship and two different dimensions, one is the influence value, and another one is the distance (in this example, it takes a constant-distance setting which has a distance of 1 between every pair of nodes). If our query is $(Friend + Colleague)^*$ and the source node is John, then there are two paths in the result, one is $(John, Friend, Alice)(Alice, Colleague, Tom)$ and another one is $(John, Friend, Tom)$. Now if we want to pick the path with the shortest distance, we calculate the distance of each path by counting the number of edges along the path, and the second path would be our answer because it has a distance of 1, which is shorter than the first path with a distance of 2. However if we want to find the path with the highest influence, we do the same calculation with the second element of each edge, and get the first path as result this time. The influence of the first path is 0.64, while the second one is 0.5. If we somehow record the edge information with different dimensions as the provenance of the target Tom, then we should be able to get these two results without computing the query result twice.

In this thesis, we are generalizing the provenance polynomials for multidimensional

provenance in graph databases, which to the best of our knowledge, has no similar research to compare with. Therefore, our main goal is to provide the theoretical definitions and proofs, and provide a feasible algorithm for computing the provenance of querying graph databases.

1.2 Thesis Contributions

The contributions of this thesis are as follows:

- We study the semantic of relational databases and graph databases, and identify a more detailed provenance semantic for querying graph databases.

- We study several existing algorithms that compute query results given regular path queries and graph databases, the state elimination algorithm that computes the regular expression for each query result, as well as the heuristics for computing the state elimination order, which affects the size of the resulting regular expressions. We compare two different heuristics and our results show that one of them, namely the state weight heuristics of (Barceló Baeza, 2013) has a better performance considering both time efficiency and the size of regular expressions it generates.

- We propose an algorithm to compute the single-source multidimensional provenance of the RPQ's over graph databases, which is based on the state elimination algorithm. We also provide a postprocessing parse-tree method for different

purposes, which could provide potential benefits in practical scenarios. We evaluate the performance of the proposed algorithm and compare the performance of different types of queries in terms of feasibility and the quality of provenance, which is the size of the regular expression generated.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 introduces the research background and provides a survey of related works. Chapter 3 provides the semantics of semirings, querying graph databases with multidimensional provenance and homomorphisms between semirings. Chapter 4 presents a design and implementation of the proposed algorithms followed by a complexity analysis. Chapter 5 illustrates the experiments and results of our proposed solutions using a library of real-life graphs and also synthetic graphs. Concluding remarks and directions for possible future works are discussed in Chapter 6.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Semirings and Homomorphism

Following (Green, Karvounarakis, & Tannen, 2007), we consider the framework for provenance based on the algebraic structure of semirings. A semiring $(K, \oplus, \otimes, \overline{0}, \overline{1})$ has the properties such that:

- $(K, \oplus, \overline{0})$ is a commutative monoid with neutral element $\overline{0}$.
- $(K, \otimes, \overline{1})$ is a monoid with neutral element $\overline{1}$.
- Multiplication left and right distributes over addition $\forall x, y, z \in K$:
  - $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$.
  - $z \otimes (x \oplus y) = (z \otimes x) \oplus (z \otimes y)$.

- Multiplication by $\overline{0}$ annihilates K: $\forall x \in K, x \otimes \overline{0} = \overline{0} \otimes x = \overline{0}$.

And among the classes of semirings, we will focus on idempotent, k-closed, and star semirings.

Definition 2.1 *Let $(K, \oplus, \otimes, \overline{0}, \overline{1})$ be a semiring. An element $a \in K$ is idempotent if $a \oplus a = a$. K is said to be idempotent when all elements of K are idempotent.*

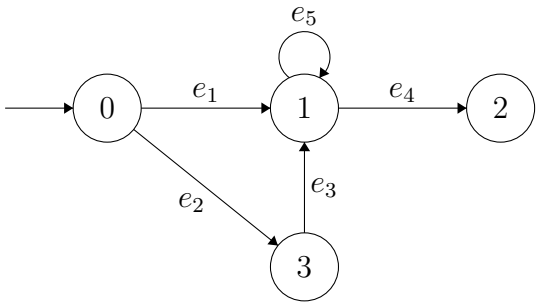
Definition 2.2 *Given $k \geq 0$, a semiring $(K, \oplus, \otimes, \overline{0}, \overline{1})$ is said to be k-closed if $\forall x \in K, \oplus_{n=0}^{k+1} x^n = \oplus_{n=0}^{k} x^n$.*

Definition 2.3 *A semiring $\mathcal{K} = (K, \oplus, \otimes, \overline{0}, \overline{1})$ is complete if $\sum_{i \in I}(x \cdot x_i) = x \cdot (\sum_{i \in I} x_i)$, and $\sum_{i \in I}(x_i \cdot x) = (\sum_{i \in I} x_i) \cdot x$, for all $x, x_i \in K$ and infinite I.*

Definition 2.4 *A* complete star-semiring *$\mathcal{K}$ is a structure $(K, \oplus, \otimes, \circledast, 0, 1)$, where $(K, \oplus, \otimes, 0, 1)$ is a complete semiring with idempotent $\oplus$, such that $x^{\circledast} = \bigoplus_{j \geq 0} x^j$, where $x^0 = 1$, $x^{j+1} = x \otimes x^j = x^j \otimes x$ for all $j \geq 0$.*

Definition 2.5 *A mapping $h : K \to K'$ from the semiring $\mathcal{K} = (K, \oplus, \otimes, \overline{0}, \overline{1})$ into the semiring $\mathcal{K}' = (K', \oplus', \otimes', \overline{0}', \overline{1}')$ is a homomorphism if (1) $h(a \oplus b) = h(a) \oplus' h(b)$; (2) $h(a \otimes b) = h(a) \otimes' h(b)$; (3) $h(\overline{0}) = \overline{0}'$ and $h(\overline{1}) = \overline{1}'$. A mapping of star semirings is a semiring homomorphism that preserves the star operation additionally, which is $h(a^{\circledast}) = h(a)^{\circledast'}$.*

Here we list some of the useful semirings, and we use Figure 2(a) as an example, where the source node is node 0, and the target node is node 2:



(a) A directed graph

E	Distance
e_1	4
e_2	1
e_3	2
e_4	2
e_5	3

(b) Each edge is assigned with a distance value

E	Access control level
e_1	P
e_2	C
e_3	P
e_4	S
e_5	P

(c) Each edge is assigned with an access level value

Figure 2: Semiring examples

Why-provenance semiring. $(Why(X), \cup, \Cup, \varnothing, \{\varnothing\})$. In graph databases, the why-provenance of a target node is a set of all sets of edges that contribute to the target node, which is also called the proof witness basis in (Buneman, Khanna, & Tan, 2001). The domain of the why provenance $Why(X)$ is the set of sets of edges, and X is the set of nodes, both of which come from the graph database. The $\oplus$ operation in the why-provenance is the set union, while the $\otimes$ operation is defined as the pointwise union, i.e. $A \Cup B = \{a \cup b : a \in A, b \in B\}$, where A and B are sets of edges. The why-provenance of node 2 in Figure 2(a) is $\{\{e_1, e_5, e_4\}, \{e_2, e_3, e_5, e_4\}, \{e_1, e_4\}, \{e_2, e_3, e_4\}\}$.

Minimal witness basis semiring (Green, 2009). $(irr(E), \oplus, \otimes, \varnothing, \{\varnothing\})$, which records the minimal witness basis (Buneman et al., 2001) of the query result. E is the set of all edges from the graph database, and $irr(E)$ is the set of irredundant

subsets of E, where a set W is in $irr(E)$ if for any sets $A, B \subseteq W$, neither is a subset of the other. To get $irr(W)$ for a set $W \subseteq E$, we can repeat checking every pair of sets $A, B \subseteq W$, and delete B from W if $A \subseteq B$. The $\oplus$ operation of two sets of edges I, J is defined as $I \oplus J \stackrel{\text{def}}{=} irr(I \cup J)$, while the $\otimes$ operation of two sets of edges I, J is defined as $I \otimes J \stackrel{\text{def}}{=} irr(I \Cup J)$. The minimal witnesses of node 2 in Figure 2(a) is $\{\{e_1, e_4\}, \{e_2, e_3, e_4\}\}$, since $\{e_1, e_4\} \subseteq \{e_1, e_4, e_5\}$ and $\{e_2, e_3, e_4\} \subseteq \{e_2, e_3, e_5, e_4\}$.

Lineage semiring. $(Lin(X), \cup, \cup, \varnothing, \varnothing)$. Lineage is the simplest form of provenance information, and it saves all distinct edges that contribute to the query result. The domain $Lin(X)$ is the set of all edges in the graph database, and X is the set of nodes in the graph database. Both $\oplus$ and $\otimes$ operations are set unions. In Figure 2(a), the lineage of node 2 is $\{e_1, e_2, e_3, e_4, e_5\}$.

Tropical semiring. $(\mathbb{R}^+ \cup \{\infty\}, min, +, *, \infty, 0)$. Tropical semiring is normally dealing with traditional single-source or all-pair shortest-distance problems (Mohri, 2002), where its domain is to represent the distance for each edge. The $\oplus$ operation is to take the shortest path of two paths, and the $\otimes$ operation is to calculate the sum of two consecutive edges. For the star operation, $x^* = 0$ for all $x \in \mathbb{R}^+ \cup \{\infty\}$. Figure 2(b) shows the distance every edge is, and using this edge information we can get the shortest distance from node 0 to node 2 is $min(4+0+2, 1+2+0+2) = 5$, and the shortest path is $e_2 \rightarrow e_3 \rightarrow e_4$.

Influence semiring. $([0, 1], max, \times, *, 0, 1)$. Influence semiring is used for problems that find the influence from one user to another on the social network, which we have shown an example in Figure 1. The domain of the influence semiring is the

decimals ranging from 0 to 1 representing the influence value between users, where the highest influence value is 1. The $\oplus$ operation is to pick the path with the most significant influence value of two paths, where the $\otimes$ is to calculate the multiplication of influence values of two consecutive edges. For all $x \in [0, 1]$, $x^* = 1$.

Access control semiring. $(\mathbb{A}, min, max, *, 0, P)$ where $\mathbb{A} = \{P, C, S, T, 0\}$. Here P means "public", C is "confidential", S is "secret", T is "top secret", and 0 is "secret that nobody can access it" (Foster, Green, & Tannen, 2008). The order of these five access levels is $P < C < S < T < 0$. The access control semiring is to compute the access level of nodes in the graph database. The $\oplus$ operation is to choose the path with a lower access level between two paths, while the $\otimes$ operation is to set the access level of two consecutive edges as the highest value between them, and $x^* = P$, for all $x \in \mathbb{A}$. Figure 2(c) shows the access level for each edge. To compute the access level from node 0 to node 2, first we compute the access level of two paths $e_1 \to e_5^* \to e_4$ and $e_2 \to e_3 \to e_5^* \to e_4$. The access level of the first path is $max(max(P, P), S) = S$, while the the access level of the second path is $max(max(max(C, P), P), S) = S$. Then we can get the access level from node 0 to node 2 by calculating $min(S, S) = S$.

Regular expression semiring. $\mathcal{K}_{\mathcal{E}(G)} = (\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$. Regular expression is the most general provenance form of querying graph databases, where $\mathcal{E}(G)$ is the set of all regular expressions over edges. The $\oplus$ operation is the union, the $\otimes$ operation is the concatenation and the star operation $*$ is the Kleen closure. We will show the homomorphism from regular expression to other semirings in the following chapters.

2.1.2 Graph Theory

An edge-labeled graph database $G = (V, E)$ with a finite alphabet Σ, where V is a finite set of nodes, and $E \subseteq V \times \Sigma \times V$ is a set of edges. Each edge e in E can be represented as $e = (v, \mathtt{a}, v')$, where $v, v' \in V$ and $\mathtt{a} \in \Sigma$. A directed graph is a graph in which every edge has a direction, for example, $e = (v, \mathtt{a}, v')$ shows that there is an edge starting from node v to node v'. The vertices are to be thought of as objects, and the edges as relations between the objects. That is, an edge $(u, \mathtt{a}, v)$ can be viewed as a tuple (u, v) in a binary relation $\mathtt{a}$.

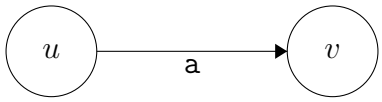


Figure 3: A direct edge from node u to node v

A *path* π in a graph database G is a sequence $\pi = (v_0, \mathtt{a}_1, v_1)(v_1, \mathtt{a}_2, v_2) \ldots (v_{n-1}, \mathtt{a}_n, v_n)$ of adjacent edges in G. We define the source of a path as $\mathsf{first}_\pi = v_0$ and the target of a path as $\mathsf{last}_\pi = v_n$. The empty path is denoted by ϵ. The *word* represented by a path π is the concatenation of labels along the path $\mathsf{word}_\pi = \mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n$. The set of all paths from v_0 to v_n in G is denoted $\Pi_{v_0,v_n}(G)$, and the set of all paths in G is denoted $\Pi(G)$.

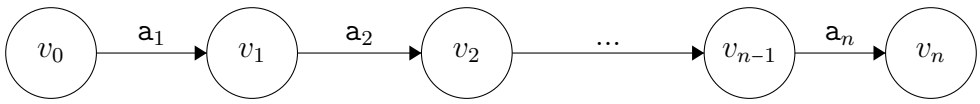


Figure 4: An example path

A weighted graph G is a graph (V, E, w) in which every edge $e \in E$ is associated

with a weight $w[e]$, and w is a weight function, $w : E \to K$ for semiring $(K, \oplus, \otimes, \overline{0}, \overline{1})$. Then the weight of a path $\pi = e_1 e_2 ... e_n$ is $w[\pi] := \otimes_{i=1}^{n} w[e_i]$, and the weight of all the path from node i to node j is $w(i,j) := \oplus_{\pi \in \Pi_{i,j}(G)} w[\pi]$. Depending on the problems we want to solve, the weight can be distance, security level, or probability, etc.

We can extend the weighted graph to the ones that contain multiple dimensions of weights, based on the nature of graph databases that edges can have multiple properties. An edge can be additionally labeled with a finite list of *weights*, and each weight comes from a semiring. Then we will show that in Chapter 3, for each weighted semiring, the *weighted provenance* can be obtained by a homomorphism from the most general form, regular expressions (or we can call it provenance polynomials, which is inherited from (Green, Karvounarakis, & Tannen, 2007)). Before introducing the weights, we use $\underline{(u, \mathtt{a}, v)}$ to represent the regular expression defining the language $\{(u, \mathtt{a}, v)\}$, and remark that the mapping $h(u, \mathtt{a}, v) = \mathtt{a}$ maps the regular expressions over edges homomorphically to regular expressions over Σ.

Let $\mathcal{K}_i$, for $i = 1, \ldots, k$, be a family of semirings. A *weighted graph database* is a graph database $(G, w_1, \ldots, w_k)$, where w_i maps edges of G to elements of the semiring $\mathcal{K}_i$. We represent a semiring labeled edge $(u, \mathtt{a}, v)$ as $(u, \mathtt{a}, \bar{w}, v)$, where $\bar{w} = (w_1(u, \mathtt{a}, v), \ldots, w_k(u, \mathtt{a}, v))$.

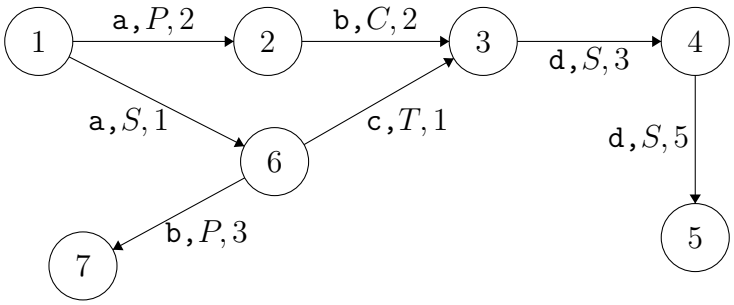


Figure 5: A weighted graph database

Figure 5 shows a weighted graph database (G, w_1, w_2). $\mathcal{K}_1$ is the access control semiring $(\mathbb{A}, min, max, 0, P)$, and $\mathcal{K}_2$ is the tropical semiring $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$. An edge of the graph is a tuple of the form $(s, \mathtt{a}, \mathtt{w}_1, \mathtt{w}_2, t)$, where $s, t \in V$, $\mathtt{a} \in \Sigma$, $\mathtt{w}_1 \in \mathbb{A}$, and $\mathtt{w}_2 \in \mathbb{R}^+ \cup \{\infty\}$.

2.1.3 Querying Graph Databases

A regular path query Q has the form $RPQ(x, y) \coloneqq (x, R, y)$, where R is a regular expression over Σ, which defines a language $L(Q) \subseteq \Sigma^*$, where Σ^* is the set of all words over Σ. When we apply the query Q to a graph database G, Q is satisfied if and only if there exists at least one path π such that word_π is in R. The formal definition of the answer $Q(G)$ is shown as follows,

$$Q(G) = \{(\mathsf{first}_\pi, \mathsf{last}_\pi) : \pi \in \mathsf{paths}(G) \text{ and } \mathsf{word}_\pi \in L(Q)\}.$$

There are three main categories of problems associated with graphs, 1) *one-pair* problem, which focuses on paths between a given pair of nodes; 2) *single-source*

problem, which focuses on paths from a given source node to the rest of the nodes; 3) *all-pairs* problem, which focuses on paths between all pairs of nodes. In this thesis, we focus on and study the single-source problem since it is more general, and it includes the one-pair problem as a special case. In addition, the single-source problem can be extended to the all-pairs problem. In this case, we assume that the graph database has a designated source vertex[1] v_s, so the answer of Q applied to G will be all the paths π started from v_s and $word_\pi \in L(Q)$

$$Q(G) = \{\mathsf{last}_\pi : \pi \in \mathsf{paths}(G),\ \mathsf{first}_\pi = v_s \text{ and } \mathsf{word}_\pi \in L(Q)\}.$$

Following the idea that an edge $(u, \mathtt{a}, v)$ represents a relational tuple $\mathtt{a}(u, v)$, the join becomes the binary composition $\mathtt{a}(u, v) \circ \mathtt{b}(v, w)$, and that is the edge concatenation $(u, \mathtt{a}, v)(v, \mathtt{b}, w)$. Therefore, we define the provenance of a target node $v \in Q(G)$ as all the paths π from the source node v_s to v such that $word_\pi \in L(Q)$:

$$\mathcal{P}_v = \{\pi : \pi \in \mathsf{paths}(G),\ \mathsf{first}_\pi = v_s,\ \mathsf{last}_\pi = v,\ \text{and } \mathsf{word}_\pi \in L(Q)\}.$$

Since there might be several paths from the source node to one of the target nodes, we can represent them as a regular expression over edges. Given $G = (V, E)$, we can form a finite set of labeled edges, giving rise to the set $\mathcal{E}(G)$ of all regular expressions over edges. Since regular expressions also include the Kleene closure $*$, we finally arrive at a structure: $\mathcal{K}_{\mathcal{E}(G)} = (\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$, which is known as a Kleene

[1] In practice, the source vertex would be part of the query, but for notational convenience, we assume that the graph database has a unique source node.

Algebra (Kozen, 1997) or equivalently as a star-semiring. Formally, this means that $(\mathcal{E}(G), +, \cdot, \varnothing, \epsilon)$ is a semiring and that $r^* = \epsilon + rr^* = \epsilon + r^*r$ for all $r \in \mathcal{E}(G)$. We define the provenance polynomial of $v \in Q(G)$ as $prov_v = r$, where $L(r) = P_v$.

Now we go back to the example of Figure 5 for further illustration. Given a query $Q = \mathtt{a(b+c)d}^*$ and using node 1 as the single source. Since we have a single source node, we can simplify the result set to only contain reachable target nodes. Therefore we get the query result $Q(G) = \{3, 4, 5\}$. Here we compute the provenance of node 4, which is

$$\mathsf{prov}_4 = \big((1, \mathtt{a}, P, 2, 2)(2, \mathtt{b}, C, 2, 3) + (1, \mathtt{a}, S, 1, 6)(6, \mathtt{c}, T, 1, 3)\big)(3, \mathtt{d}, S, 3, 4),$$

and we can use it as an example to show how to apply homomorphisms from $\mathcal{K}_{\mathcal{E}(G)}$ to $\mathcal{K}_i$. $\mathcal{K}_1$ is the access control semiring. The homomorphism $h_1 : \mathcal{K}_{\mathcal{E}(G)} \to \mathcal{K}_1$ is defined by $h_1(s, \mathtt{a}, \mathtt{w}_1, \mathtt{w}_2, t) = \mathtt{w}_1$, $h_1(\mathtt{e}_1 + \mathtt{e}_2) = \min(h_1(\mathtt{e}_1), h_1(\mathtt{e}_2))$, $h_1(\mathtt{e}_1\mathtt{e}_2) = \max(h_1(\mathtt{e}_1), h_1(\mathtt{e}_2))$, and $h_1(\mathtt{e}^*) = P$. The provenance for vertex 4 in access control semiring is $h_1(\mathsf{prov}_4) = \max(\min(\max(P, C), \max(S, T)), S) = S$. Similarly, the provenance for vertex 4 in the tropical semiring is $\min(2 + 2, 1 + 1) + 3 = 5$. To get the regular expression provenance for vertex 4, we use the previously mentioned homomorphism h, with $h(\mathsf{prov}_4) = \mathtt{(ab+ac)d}$.

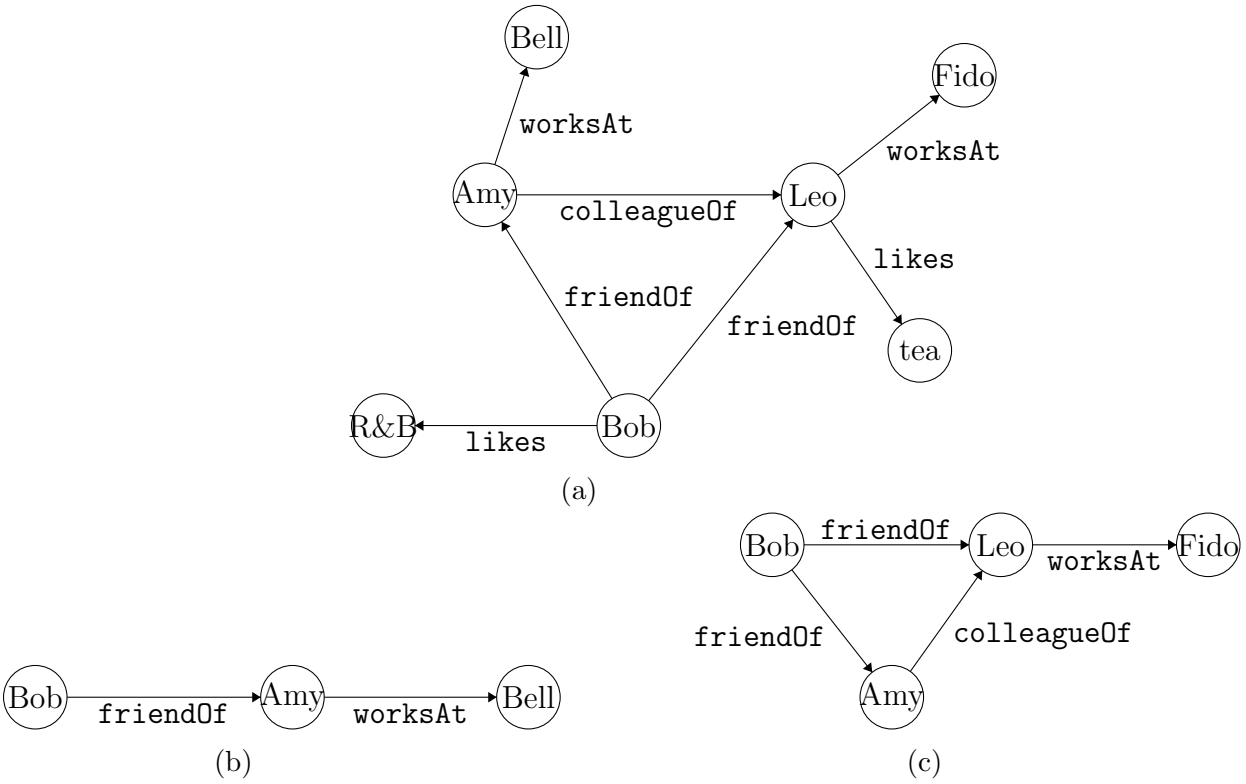


Figure 6: A graph database and query results

Figure 6 is another example where Figure 6(a) gives a graph G that records relationships between several users. Consider the source node as Bob, and a regular path query $(friendOf + colleagueOf)^* worksAt$. The query result $Q(G) = \{Bell, Fido\}$ and the paths between the source node and the end nodes are shown in Figure 6(b) and Figure 6(c).

From these paths in Figure 6(b) and Figure 6(c), we can get the provenance polynomials of the target nodes in the query result separately. The provenance of Bell is $P_{Bell} = (Bob, friendOf, Amy)(Amy, worksAt, Bell)$, and the provenance of Fido is $P_{Fido} = ((Bob, friendOf, Amy)(Amy, colleagueOf, Simon) + (Bob, friendOf, Simon))(Simon, worksAt, Fido)$. More details will be discussed in

Chapter 3.

One of the well-known methods to compute regular path queries over graph databases is using automata (Goldman & Widom, 1997). First, we introduce the associated DFA A_Q of Q. Kleene's theorem (Kleene, 2016) shows that regular expressions and finite automatons have the same expressive power, which allows us to convert a finite automaton to a regular expression and vice versa. A deterministic finite automata is $A_Q = \{S, \Sigma, \delta, s_0, F\}$, where S is the set of states, Σ is the alphabet, δ is the transition relation $\delta \subseteq S \times \Sigma \times S$, such that if $(s, \mathtt{a}, t)$ and $(s, \mathtt{a}, t')$ in δ then $t = t'$. s_0 is the initial state and F is the set of final states. Once we get the FA converted from the given query, we can compute the intersection of the graph database $G = (V, E, w)$ and A_Q as the product graph $P_{G \times A_Q}$. The product graph $P_{G \times A_Q}$ is an NFA that is defined as $P_{G \times A_Q} = (V \times S, E')$. Each new node in $P_{G \times A_Q}$ is a pair of nodes (v, s), one from the graph database $v \in V$, and another from the query FA $s \in S$. Each edge can be represented as $E' = \{((v_i, s_i), \mathtt{a}, (v_j, s_j)) : \mathtt{a} \in \Sigma, (v_i, \mathtt{a}, v_j) \in E$ and $\delta(s_i, \mathtt{a}) = s_j\}$, where for every edge $e' = ((v_i, s_i), \mathtt{a}, (v_j, s_j)) \in E'$, the label of edge e' is the same as the label of edge $e = (v_i, \mathtt{a}, v_j)$ back to G.

Once we have the product graph $P_{G \times A_Q}$, we want the regular expression for each answer node, since that is the most general form of the provenance. Kleene's algorithm (Kleene, 2016) transforms a given nondeterministic finite automaton (NFA) into a regular expression. Given a NFA $M = (Q, \Sigma, \delta, q_0, F)$, with $Q = q_0, ..., q_n$, Kleene's algorithm works by constructing a sequence of matrices M^k in which the entry R^k_{ij} is a regular expression for all paths from vertex i to vertex j with no intermediate vertex

on the path (except possibly for the endpoints) that is numbered higher than k. R_{ij}^0 is the sum of the labels of the edges from node i to j in M, and is $\varnothing$ if there are no edges. Also, if $i = j$, we need to add ϵ to R_{ij}^0 since the path of length 0 is represented by ϵ. To compute the R_{ij}^{k+1}, we use the formula $R_{ij}^{k+1} = R_{ij}^k + R_{ik}^k \cdot (R_{kk}^k)^* \cdot R_{kj}^k$ (Kleene, 2016).

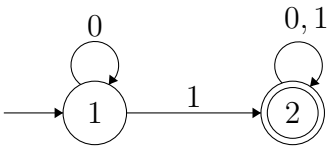


Figure 7: A DFA

Here we use an example of converting an NFA to a regular expression, where the FA M is shown in Figure 7, representing a language that accepts all strings with at least one 1. For the first matric M^0, we get the R_{ij}^0 of all pairs of vertices in M. We can easily find that R_{12}^0 is 1 since there is a direct path with label 1 from vertex 1 to 2. R_{11}^0 is $\epsilon + 0$ because the start state and the end state are the same one, and there is a loop on state 1 with label 0. Similarly we have $R_{22}^0 = \epsilon + 0 + 1$ and $R_{21}^0 = \varnothing$.

To start the induction, we need to compute the regular expressions that include paths that go through state 1 first, and then states 1 and 2. First, with the rule $R_{ij}^1 = R_{ij}^0 + R_{ik}^0 \cdot (R_{kk}^0)^* \cdot R_{kj}^0$, the regular expressions are built and shown in Table 1, where we apply the simplification rules $(\epsilon + R)^* = R^*$, $\varnothing R = R\varnothing = \varnothing$ and $\varnothing + R = R + \varnothing = R$.

	By Direct Computation	Simplified Version
R^1_{11}	$\epsilon + 0 + (\epsilon + 0)(\epsilon + 0)^*(\epsilon + 0)$	0^*
R^1_{12}	$1 + (\epsilon + 0)(\epsilon + 0)^*1$	0^*1
R^1_{21}	$\varnothing + \varnothing(\epsilon + 0)^*(\epsilon + 0)$	$\varnothing$
R^1_{22}	$\epsilon + 0 + 1 + \varnothing(\epsilon + 0)^*1$	$\epsilon + 0 + 1$

Table 1: Regular expressions R^1_{ij}

Then, applying the inductive rule with $k = 2$ and we have $R^2_{ij} = R^1_{ij} + R^1_{ik} \cdot (R^1_{kk})^* \cdot R^1_{kj}$. Table 2 shows the final regular expressions, where $R^2_{12} = 0^*1(0 + 1)^*$ is the regular expression equivalent to M, since state 1 is the initial state and state 2 is the only final state.

	By Direct Computation	Simplified Version
R^2_{11}	$0^* + 0^*1(\epsilon + 0 + 1)^*\varnothing$	0^*
R^2_{12}	$0^*1 + (0^*1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0^*1)(0 + 1)^*$
R^2_{21}	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\varnothing$	$\varnothing$
R^2_{22}	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0 + 1)^*$

Table 2: Regular expressions R^2_{ij}

However, the cost of Kleene's algorithm is expensive. The time complexity is $\mathcal{O}(n^3)$ and the space complexity is $\mathcal{O}(n^2 4^n)$. So instead of using Kleene's algorithm, we can use a similar one called the state elimination algorithm, which is an algorithm that can convert a finite automaton to a regular expression by removing intermediate states one by one. To be more detailed, for each final state in the given FA, it keeps eliminating non-initial non-final states one by one and updates transitions associated, until only the initial and final states are left, when we can get the resulting regular expression. However the size of regular expressions generated from the state elimination algorithm depends heavily on the removal order, which can make a huge difference.

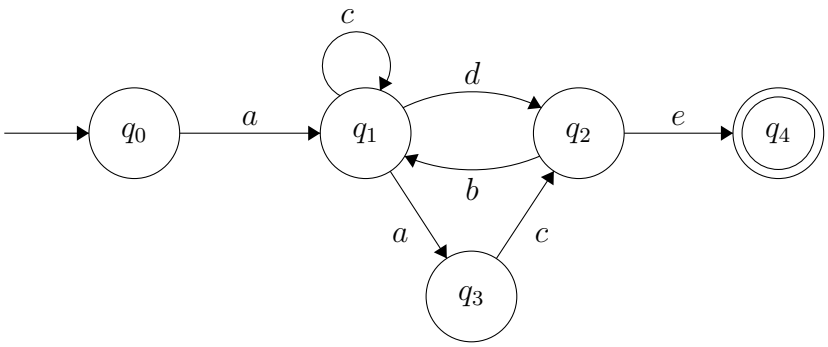


Figure 8: An example finite automata

Figure 8 is an example FA where we want to get its equivalent regular expression. We compute the length of a regular expression as the sum of letters and operators $+$, $\cdot$, and $*$ here and in the following chapters. A removal sequence $q_3 \to q_2 \to q_1$ results in regular expression $a(c+(d+ac)b)^*(d+ac)e$, which has length of 20, whereas another removal sequence $q_1 \to q_2 \to q_3$ leads to the regular expression $ac^*d(bc^*d)^*e + (ac^*a + ac^*d(bc^*d)^*bc^*a)(c(bc^*d)^*bc^*a)^*c(bc^*d)^*e$, which has a length of 75.

2.2 Related Works

Graph databases have been one of the trending areas in technology academically and industrially since the late 1960s. Before graph databases appeared on the horizon, relational databases were mainly used for storing and retrieving data, which depended heavily on schemas and tables. That nature made it difficult to add or delete relationships between objects, or deal with the operations that involve a large number of joints of large tables. The growing popularity of the internet and the explosion of data generated an urgent demand for access to databases from the Web, so graph

databases were getting more and more attention. Comparing to relational databases, graph databases are more focused on relationships between objects, where edges can be labeled, directed with more information, and therefore it is more suitable in many applications nowadays, such as the Semantic Web (Hayes & Gutierrez, 2004; Arenas & Pérez, 2011), bioinformatics (Fabregat et al., 2018), and road transportation networks (Añez et al., 1996), etc.

Provenance has been studied in the past few decades, and it mainly answers why, how, and where data is generated. A survey (Cheney, Chiticariu, & Tan, 2009) covers these three types of provenance. Why-provenance or the lineage (Cui, Widom, & Wiener, 2000) of a query result is the set of contributing tuples to the output. How-provenance (Green, Karvounarakis, & Tannen, 2007) tries to figure out how tuples contribute to the output. In relational databases, where-provenance (Buneman et al., 2001) is also a concerning problem since users want to detect which tables are these involved tuples coming from. When we do the query processing over annotated relations, operations over annotations are involved and propagated. Vicknair et al. (Vicknair et al., 2010) compares the ability of recording and querying data provenance between a relational database (MySQL) and a graph database (Neo4j).

Green et al. (Green, Karvounarakis, & Tannen, 2007) observed the similarity of relational algebra operations over c-tables (Imieliński & Lipski, 1984), event tables (Fuhr & Rölleke, 1997; Zimányi, 1997), bag semantics and the why-provenance. Therefore, they proposed the semiring of polynomials as the most general form of the provenance of querying relational databases. To provide more semantic details, first,

they introduced K-relations, where K is a commutative semiring $(K, +, \cdot, 0, 1)$, and K-relation is a function R that annotates each tuple with elements in K. K-relations can be transformed to another K'-relations by semiring homomorphism, and because of that, it is possible to transform the most general provenance form, polynomials, to other semirings for different purposes. The semirings of polynomials are defined as follows: X be the set of tuple ids of a database instance I. The positive algebra provenance semiring for I is the semiring of polynomials with variables from X and coefficients from $\mathbb{N}$, with the operations defined as usual: $(\mathbb{N}[X], +, \cdot, 0, 1)$. To illustrate the semiring of polynomials and its homomorphism, here we use the example shown in (Green, Karvounarakis, & Tannen, 2007).

A	B	C	
a	b	c	p
d	b	e	r
f	g	e	s

(a) Relation with annotation

A	C	
a	c	$2p^2$
a	e	pr
d	c	pr
d	e	$2r^2+rs$
f	e	$2r^2+rs$

(b) Query Result

A	C	
a	c	{p}
a	e	{p,r}
d	c	{p,r}
d	e	{r,s}
f	e	{r,s}

(c) Why provenance $(Why(X), \cup, \uplus, \varnothing, \{\varnothing\})$

E	
p	2
r	5
s	1

A	C	
a	c	$2 \cdot 2 + 2 \cdot 2 = 8$
a	e	$2 \cdot 5 = 10$
d	c	$2 \cdot 5 = 10$
d	e	$5 \cdot 5 + 5 \cdot 5 + 5 \cdot 1 = 55$
f	e	$1 \cdot 1 + 1 \cdot 1 + 5 \cdot 1 = 7$

(d) Bag semantics $(\mathbb{N}, +, \cdot, 0, 1)$

E	
p	2
r	5
s	1

A	C	
a	c	$min(2+2, 2+2) = 4$
a	e	$2 + 5 = 7$
d	c	$2 + 5 = 7$
d	e	$min(5+5, 5+5, 5+1) = 6$
f	e	$min(1+1, 1+1, 5+1) = 2$

(e) Tropical semiring $(R^+ \cup \{\infty\}, min, +, \infty, 0)$

Figure 9: A relational database and query results

Given a query $q(R) \stackrel{\text{def}}{=} \pi_{AC}(\pi_{AB}R \bowtie \pi_{BC}R \cup \pi_{AC}R \bowtie \pi_{BC}R)$ and a relational database in Figure 9(a), where each tuple is annotated with a unique id from the set $X = \{p, r, s\}$. The result tuples and their provenance polynomials (which are in the most general form) are shown in Table 9(b), and we can get the information of the how-provenance and the why-provenance from these polynomials. For example, the provenance of (d, e) is $2r^2 + rs$, which shows that there are two ways to get (d, e) as result, one is to be computed by tuple r and itself twice, and the other one uses r and s once. Then with the polynomial, we can do homomorphism from $\mathbb{N}[X] \rightarrow K$, where K is other commutative semirings, and get different results. For instance, we can use the homomorphism from polynomials to the tropical semiring, which includes $h(p) = 2$, $h(r) = 5$, $h(s) = 1$, $h(x + y) = min(h(x), h(y))$, and $h(xy) = h(x) + h(y)$. Then the result value of ac will be $h(2p^2) = h(p^2 + p^2) = min(h(pp), h(pp)) = min(h(p) + h(p), h(p) + h(p)) = min(2 + 2, 2 + 2) = 4$. The rest of tables in Figure 9 shows the homomorphism from $\mathbb{N}[X]$ to the why-provenance and bag semantics respectively.

Also, Green et al. (Green, Karvounarakis, Ives, & Tannen, 2007) show the potential utilization of provenance in incremental view maintenance, to be specific, for updating deletion propagation without recomputing the query results. We can use the example in Figure 9 to show how it works. Assuming Figure 9(a) is the original relational database, and Figure 9(b) shows the query result, where the query is $q(R) \stackrel{\text{def}}{=} \pi_{AC}(\pi_{AB}R \bowtie \pi_{BC}R \cup \pi_{AC}R \bowtie \pi_{BC}R)$. Now if at some points we delete the tuple (d, b, e), which has the identifier r from the database, then we can update the

query result directly by setting $r = 0$. As a result, (a, e), (d, c), (d, e) and (f, e) will be removed from the query result, which leaves (a, c) as the only answer.

Research was focused on the provenance over relational databases and datalog queries initially, and later on, it is extended to other types of databases, such as XML databases, graph databases, etc. Based on the different nature and also applications of these databases, the definition and semirings are different. The provenance for XML can be used for security applications and a general strong representation systems for incomplete and probabilistic annotated databases. It focuses more on the lineage information, where and how provenance, since XML databases allow incomplete properties and it is hard to track back to the source tuples that contribute to the results. In (Foster et al., 2008; Cheney, 2009), the semiring annotations go from existing models of provenance for relational databases to XML and the query languages. As for RDF, (Li Ding & McGuinness, 2005) uses RDF molecule, which is the finest and lossless component of an RDF graph, to track the provenance of the RDF graph. (Dividino, Sizov, Staab, & Schueler, 2009) deals with multiple dimensions for RDF, such as source, authorship, and certainty. (Flouris, Fundulaki, Pediaditis, Theoharis, & Christophides, 2009) uses colors to capture the provenance of RDF. Ramusat et al. (Ramusat et al., 2018, 2021) reduce the provenance of querying graph databases to the reachability problem, and compare the performance of several algorithms, such as Mohri's algorithm, the node elimination algorithm and Dijkstra's algorithm. However, the provenance defined in their research is less general, for example, the provenance generated does not contain the lineage information, and they

only consider the single-source, single-target problem.

Given a directed graph database and an RPQ, the provenance of querying graph databases can be computed by constructing a product graph and converting it to a regular expression. The Kleene theorem (Kleene, 2016) shows that every n-state finite automaton over the alphabet Σ has an equivalent regular expression. Kleene's algorithm can be used to solve the all-pairs reachability problem, which computes reachability information between arbitrary pairs of nodes in a directed graph. The state elimination algorithm is one of the methods that convert FAs to regular expressions, which can lead to an exponential size of result since the upper bound for the size of result regular expression is $\mathcal{O}(nk4^n)$ (Hopcroft, Motwani, & Ullman, 2006), where n is the number of states and k is the number of alphabetic symbols. A lot of research has been focused on the heuristics of the state removal sequence for the state elimination algorithm to reduce the size of result regular expressions. (Delgado & Morais, 2005) uses a greedy algorithm (DM algorithm) to calculate the state weight and remove the state with the smallest weight for each round. (Han & Wood, 2007) purposes a divide-and-conquer algorithm (HW algorithm) to split FAs horizontally and vertically to obtain a shorter regular expression, based on bridge states and the structural properties of FAs. The bridge state in an FA A in (Han & Wood, 2007) is defined to satisfy three conditions: 1) the bridge state is a non-initial non-final state; 2) the path of every string $w \in L(A)$ must pass through the bridge state; 3) the bridge state does not participate in any cycle except for a self-loop. However the method depends heavily on the structure of the FA, and as experiments in (Moreira,

Nabais, & Reis, 2010) show that this HW algorithm is less effective when the size and complexity of FAs increase, when bridge states become rarer. (Moreira et al., 2010) counts the number of cycles passing through for each state, and eliminates the state with the lowest number of cycles. Comparison has been made on performance with DM algorithm and HW algorithm, where the experimental results show that DM algorithm has a better performance at most times, while the counting-cycle algorithm outperforms when the size of the alphabet and the number of states are small. (Ramusat et al., 2021) applies a degree-based state removal heuristic, and our experiments show that it has a similar execution time comparing to DM algorithm, however, it leads to a longer regular expression on average.

Chapter 3

Polynomials for Multidimensional Provenance in Graph Databases

We propose regular path expressions as the most general provenance representation of querying graph databases, provide the homomorphism from regular path expressions to several useful semirings, and we present an algorithm to compute the multidimensional provenance polynomials of querying graph databases. Our algorithm is based on the state elimination algorithm and the state weight heuristic for the state removal order. Meanwhile, we use parse trees for post-processing the regular expressions to get the application provenance values. The whole set of algorithms is suitable for graphs with cycles, and it brings us the most general and detailed provenance.

3.1 Semirings and Homomorphism

To demonstrate that the most general form of querying graph databases is regular expressions over edges, we will first show that the regular path expression is also a complete star-semiring, then show the homomorphism from the set of all paths of graphs to regular path expressions, and the homomorphism from regular path expressions to several semirings.

3.1.1 Graph Databases and Kleene's Algorithm

A graph database $G = (V, E, \Sigma)$ is an edge-labelled directed graph, where $V = 1, ..., n$ is a finite set of vertices, Σ is a finite set of (predicate) labels, and $E \subseteq V \times \Sigma \times V$ is a set of (directed) labelled edges. A path from vertex i to vertex j in a graph database G is a sequence $\pi = (i, \mathsf{a}_1, i_1)(i_1, \mathsf{a}_2, i_2)...(i_{n-1}, \mathsf{a}_n, j)$ of adjacent edges of G. We define the source node of a path π as $\mathsf{first}_\pi = i$ and the target node as $\mathsf{last}_\pi = j$. The empty path is denoted ϵ.

A path π such that $\mathsf{first}_\pi = i$ and $\mathsf{last}_\pi = j$ will sometimes be denoted $\pi_{i,j}$. The set of all paths from i to j in G is denoted $\Pi_{i,j}(G)$, and the set of all paths in G is denoted $\Pi(G)$.

We consider the following operations on paths and sets of paths. Define "$\cdot$" by concatenation, i.e. $\{\pi\} \cdot \{\pi'\} = \{\pi''\}$, where π'' is the concatenation of π with π' if $\mathsf{last}_\pi = \mathsf{first}_{\pi'}$ and $\{\pi\} \cdot \{\pi'\} = \varnothing$ otherwise. For sets of paths P and P', their concatenation $P \cdot P'$ is obtained by applying the concatenation point-wise $P \cdot P' =$

$\bigcup_{\pi\in P,\pi'\in P'}\{\pi\}\cdot\{\pi'\}$. Furthermore, let $P^* = \cup_{j\geq 0}P^j$, where $P^0 = \epsilon$, $P^{j+1} = P\cdot P^j$, for $j \geq 0$. It can be easily verified that $\mathcal{K}_{\text{paths(G)}} = (2^{\Pi(G)}, \cup, \cdot, *, \varnothing, \epsilon)$ is a complete star-semiring. Let $\Pi^k_{i,j}(G)$ be the set of paths from i to j with intermediate vertices in $\{1, ..., k\}$, where $k \leq n$. Then clearly

$$\Pi^k_{i,j}(G) = \Pi^{k-1}_{i,j}(G) \cup \Pi^{k-1}_{i,k}(G)\cdot\Pi^{k-1}_{k,k}(G)^*\cdot\Pi^{k-1}_{k,j}(G) \tag{1}$$

and $\Pi_{i,j}(G) = \Pi^n_{i,j}(G)$.

For a labelled graph $G = (V, E, \Sigma)$, we define the set $\mathcal{E}(G)$ of regular expressions over edges in G by stating that $\varnothing$, ϵ, and $\underline{(i, \mathtt{a}, j)}$ for each edge $(i, \mathtt{a}, j) \in E$, are regular expressions, and the set of regular expressions is closed with the operations $\cdot$, $+$, and $*$. Then $\mathcal{K}_{\mathcal{E}(G)} = (\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$ can be verified to be a complete star-semiring.

For $\mathtt{e} \in \mathcal{E}(G)$, the language of $\mathtt{e}$, denoted $L(\mathtt{e})$, is defined in the usual way, with $L(\underline{(i, \mathtt{a}, j)}) = \{(i, \mathtt{a}, j)\}$. We can now compute regular expressions from node i to j as $\mathtt{e}_{i,j} \in \mathcal{E}(G)$ such that $L(\mathtt{e}_{i,j})$ includes all the paths that start with i and end with j, which can also be represented as $\Pi_{i,j}(G)$. The expressions are computed recursively as $\mathtt{e}^0_{i,j}, \mathtt{e}^1_{i,j}, ..., \mathtt{e}^n_{i,j} = \mathtt{e}_{i,j}$ as follows: Let $(i, \mathtt{a}_1, j), (i, \mathtt{a}_2, j), ..., (i, \mathtt{a}_k, j)$ be all the edges from node i to j in V . Then $\mathtt{e}^0_{i,j} = \underline{(i, \mathtt{a}_1, j)} + \underline{(i, \mathtt{a}_2, j)} + ... + \underline{(i, \mathtt{a}_k, j)}$ and $\mathtt{e}^0_{i,i} = \underline{(i, \mathtt{a}_1, i)} + \underline{(i, \mathtt{a}_2, i)} + ... + \underline{(i, \mathtt{a}_k, i)}$. Recursively we then define

$$\mathtt{e}^k_{i,j} = \mathtt{e}^{k-1}_{i,j} + \mathtt{e}^{k-1}_{i,k}\cdot(\mathtt{e}^{k-1}_{k,k})^*\cdot\mathtt{e}^{k-1}_{k,j} \tag{2}$$

Where $\mathtt{e}^k_{i,j}$ is a regular expression for all paths from node i to node j with no

intermediate node on the path that is numbered higher than k. The next proposition is proved by induction, using equations (1) and (2).

Proposition 1 *Let* $\mathsf{e}_{i,j} \in \mathcal{E}(G)$. *Then* $L(\mathsf{e}_{i,j}) = \Pi_{i,j}(G)$.

3.1.2 Homomorphism

To show that the regular expression semiring $\mathcal{K}_{\mathcal{E}(G)}$ is the most general form of querying graph databases, we need to prove that there is a homomorphism from $\mathcal{K}_{\mathcal{E}(G)}$ to other useful semirings.

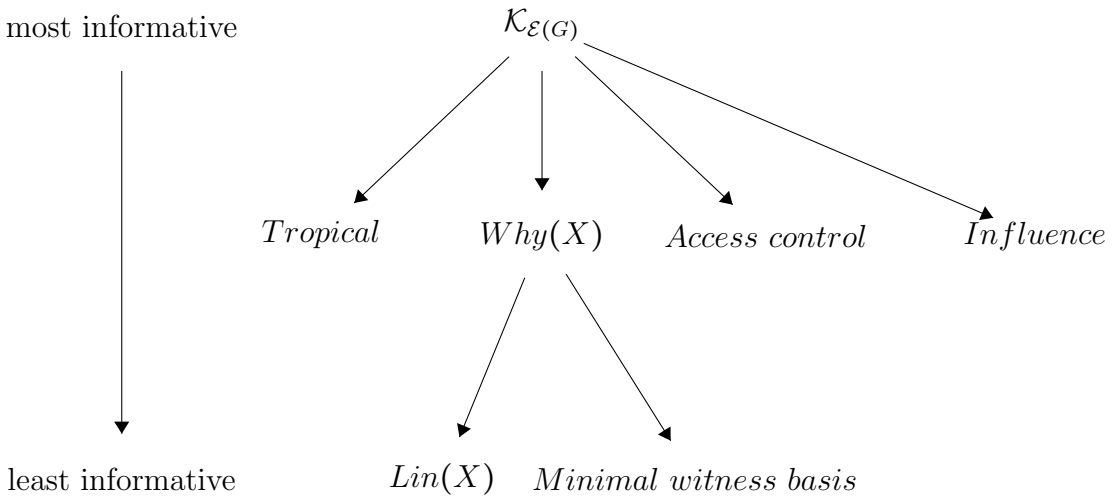


Figure 10: A Provenance hierarchy tree

We identify a provenance hierarchy tree for graph databases to show the informative level of different semirings, inspired from (Green, 2009) and we make a few bit changes in Figure 10. In this hierarchy tree, the upper semiring has more information recorded than the lower semiring, where the semiring of regular expressions is the most informative one at the top. A path downward from semirings $\mathcal{K}_1$ to $\mathcal{K}_2$

indicates that there exists a surjective semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$. There are seven semirings included in this hierarchy tree, which are

- the semiring of regular expressions $(\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$
- the Tropical semiring $(\mathbb{R}^+ \cup \{\infty\}, min, +, *, \infty, 0)$
- the Access Control semiring $(\mathbb{A}, min, max, *, 0, P)$
- the Influence semiring $([0, 1], max, \times, *, 0, 1)$
- the Why-provenance semiring $(Why(X), \cup, \Cup, \varnothing, \{\varnothing\})$
- the Minimal witness basis semiring $(irr(E), \oplus, \otimes, \varnothing, \{\varnothing\})$
- the Lineage semiring $(Lin(X), \cup, \cup, \varnothing, \varnothing)$

The main difference between our provenance hierarchy tree and the one in (Green, 2009) is that ours does not include $\mathbb{B}[X]$ and $Trio(X)$. Before we explain why, we give the definitions of these two semirings and the semiring of polynomials from (Green, 2009).

Definition 3.1 *(Provenance Polynomials). The provenance polynomials semiring for X is the semiring of polynomials with variables from X and coefficients from $\mathbb{N}$, with the operations defined as usual:* $(\mathbb{N}[X], +, \cdot, 0, 1)$.

Definition 3.2 *(Boolean Provenance Polynomials). The Boolean provenance polynomials semiring for X is the semiring of polynomials over variables X with Boolean coefficients:* $(\mathbb{B}[X], +, \cdot, 0, 1)$*, where $+$ represents set union and $\cdot$ represents bag union.*

Definition 3.3 *(Trio Semiring). The Trio semiring* $(Trio(X), +, \cdot, 0, 1)$ *for* X *is the quotient semiring of* $\mathbb{N}[X]$ *by* $\approx_f$ *, denoted* $Trio(X)$*, where* $+$ *represents bag union and* $\cdot$ *represents set union. Denote by* $\approx_f$ *the equivalence relation on* $\mathbb{N}[X]$ *defined by* $a \approx_f b \stackrel{def}{=} f(a) = f(b)$.

The provenance polynomials semiring can be treated as the most general provenance for querying relational databases, while $\mathbb{B}[X]$ and $Trio[X]$ provides information of which bags of source tuples and how many times a given set of source tuples is involved respectively. So $\mathbb{B}[X]$ can be obtained by removing the coefficient from the provenance polynomials, while $Trio[X]$ can be computed by dropping the exponents from the provenance polynomials.

There are two reasons why the provenance hierarchy tree for graph databases does not include $\mathbb{B}[X]$ and $Trio[X]$. In graph databases, $\mathbb{B}[X]$ is to compute the set of bags of edges that contribute to the target nodes. Since the Kleene Star operator $*$ is included in regular expressions, the size of the set is infinite as long as a star operator appears in the regular expression, which makes $\mathbb{B}[X]$ less useful for graph databases.

Next, to compute $Trio(X)$ from a semiring of regular expressions, we need to remove the "exponents" from the regular expressions, and that "exponents" always come from the star operator. Let f be the mapping from $\mathcal{E}(G)$ to $Trio(X)$, e.g., $f(a^*bc) = [\{a, b, c\}]$. Also, since there is no "coefficients" in regular expressions, the bag has no duplicate elements. Therefore, we can find that this mapping is the same one for the homomorphism from the semiring of regular expressions to the why-semiring, and that is why there is no need to put $Trio(X)$ into the provenance

hierarchy tree for graph databases.

With this revised hierarchy tree, we still need one more step before we start doing the homomorphism, which is to show that $\mathcal{E}(G)$ is the quotient set of all equivalent regular expressions. The semiring of regular expressions is defined as $\mathcal{K}_{\mathcal{E}(G)} = (\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$, where $\mathcal{E}(G)$ is the set of all regular expressions over paths in G, and it is already verified as a complete star-semiring. To extend it to a more general semiring, here we introduce the concept of the equivalence relations. For $E, F \in \mathcal{E}(G)$, E and F are equivalent, represented as $E \Leftrightarrow F$, if E and F can be shown to be equivalent by the axioms of Kleene algebra (Kozen, 1997). Then $\Leftrightarrow$ is an equivalence relation. Let $[E]$ denote the corresponding equivalence class of expression E, and extend the operators to equivalence classes in the standard way, for example, $[E] + [F] = [E + F]$. Furthermore, let $[\mathcal{E}(\mathsf{G})]$ denote the quotient of $\mathcal{E}(\mathsf{G})$ wrt $\Leftrightarrow$. It is straightforward to verify that $([\mathcal{E}(\mathsf{G})], +, \cdot, *, \varnothing, \epsilon)$ is a complete star-semiring. Now we can prove the homomorphism in a general way.

Theorem 3.1 *Suppose $G = (V, E, \Sigma)$ is a graph database. Let $\mathcal{K} = (K, \oplus, \otimes, \circledast, 0, 1)$ be a complete star-semiring, h a mapping from the atomic regular expressions $\underline{(i, \mathsf{a}, j)}$ in $\mathcal{E}(G)$ to K, and $\hat{h}$ it's extension to a homomorphism from $\mathcal{K}_{\mathcal{E}(G)}$ to $\mathcal{K}$. Then let h' be a mapping from E to K defined by $h'(i, \mathsf{a}, j) = h(\underline{(i, \mathsf{a}, j)})$, and $\hat{h'}$ it's extension to a homomorphism from $\mathcal{K}_{\mathrm{paths(G)}}$ to $\mathcal{K}$. For every $\mathsf{e}_{i,j} \in \mathcal{E}(G)$, we then have*

$$\hat{h}(\mathsf{e}_{i,j}) = \hat{h'}(\Pi_{i,j}(G)).$$

Proof: We show by an induction of k that $\hat{h}(\mathsf{e}^k_{i,j}) = \hat{h'}(\Pi^k_{i,j})$. For the base case

$$\begin{aligned}
\hat{h}(\mathsf{e}^0_{i,j}) &= \hat{h}\Big(\underline{(i,\mathtt{a}_1,j)} + \underline{(i,\mathtt{a}_2,j)} + \cdots + \underline{(i,\mathtt{a}_k,j)}\Big) \\
&= \hat{h}\Big(\underline{(i,\mathtt{a}_1,j)}\Big) \oplus \hat{h}\Big(\underline{(i,\mathtt{a}_2,j)}\Big) \oplus \cdots \oplus \hat{h}\Big(\underline{(i,\mathtt{a}_k,j)}\Big) \\
&= \hat{h'}(\{(i,\mathtt{a}_1,j)\}) \oplus \hat{h'}(\{(i,\mathtt{a}_2,j)\}) \oplus \cdots \oplus \hat{h'}(\{(i,\mathtt{a}_k,j)\}) \\
&= \hat{h'}\Big(\{(i,\mathtt{a}_1,j)\} \cup \{(i,\mathtt{a}_2,j)\} \cup \ldots \cup \{(i,\mathtt{a}_k,j)\}\Big) \\
&= \hat{h'}(\Pi^0_{i,j}(G)).
\end{aligned}$$

Similarly we have $\hat{h}(\mathsf{e}^0_{i,i}) = \hat{h'}(\Pi^0_{i,i})$. For the inductive step

$$\begin{aligned}
\hat{h}(\mathsf{e}^{k+1}_{i,j}) &= \hat{h}\Big(\mathsf{e}^{k-1}_{i,j} \;+\; \mathsf{e}^{k-1}_{i,k} \cdot (\mathsf{e}^{k-1}_{k,k})^* \cdot \mathsf{e}^{k-1}_{k,j}\Big) \\
&= \hat{h}\Big(\mathsf{e}^{k-1}_{i,j}\Big) \oplus \hat{h}\Big(\mathsf{e}^{k-1}_{i,k}\Big) \otimes \hat{h}\Big(\mathsf{e}^{k-1}_{k,k}\Big)^{\circledast} \otimes \hat{h}\Big(\mathsf{e}^{k-1}_{k,j}\Big) \\
&= \hat{h'}\Big(\Pi^{k-1}_{i,j}(G)\Big) \oplus \hat{h'}\Big(\Pi^{k-1}_{i,k}(G)\Big) \otimes \hat{h'}\Big(\Pi^{k-1}_{k,k}(G)\Big)^{\circledast} \otimes \hat{h'}\Big(\Pi^{k-1}_{k,j}(G)\Big) \\
&= \hat{h'}\Big(\Pi^{k-1}_{i,j}(G) \cup \Pi^{k-1}_{i,k}(G) \cdot (\Pi^{k-1}_{k,k}(G))^* \cdot \Pi^{k-1}_{k,j}(G)\Big) \\
&= \hat{h'}(\Pi^{k+1}_{i,j}(G)).
\end{aligned}$$

Lemma 1 *Let $\mathsf{e} \in \mathcal{E}(G)$ and h, h' as in Theorem 3.1. Then $\hat{h}(\mathsf{e}) = \hat{h'}(L(\mathsf{e}))$.*

Proof: For the base case we have $\hat{h}(\underline{(i,\mathtt{a},j)}) = \hat{h'}(\{(i,\mathtt{a},j)\}) = \hat{h'}(L(\underline{(i,\mathtt{a},j)}))$. For the inductive step $\hat{h}(\mathsf{e}_1 + \mathsf{e}_2) = \hat{h}(\mathsf{e}_1) \oplus \hat{h}(\mathsf{e}_2) = \hat{h'}(L(\mathsf{e}_1)) \oplus \hat{h'}(L(\mathsf{e}_2)) = \hat{h'}(L(\mathsf{e}_1) \cup L(\mathsf{e}_2)) = \hat{h'}(L(\mathsf{e}_1 + \mathsf{e}_2))$. Similar reasoning for $\cdot$ and *.

Corollary 1 *Let $\mathsf{e} \in \mathcal{E}(G)$ such that $L(\mathsf{e}) = L(\mathsf{e}_{i,j})$. Then $\hat{h}(\mathsf{e}) = \hat{h'}(\Pi_{i,j}(G))$.*

Proof: $\hat{h}(\mathsf{e}) = \hat{h'}(L(\mathsf{e})) = \hat{h'}(L(\mathsf{e}_{i,j})) = \hat{h'}(\Pi_{i,j}(G))$. The first equality is by Lemma 1 and the third by Proposition 1.

This theorem shows that given a graph G, the regular expression from node i to j we get from Kleene's algorithm or its equivalent ones, are equivalent to the regular expression of all paths from node i to j in G, and it allows us to use Kleene's algorithm or the state elimination algorithm to compute the provenance of querying graph databases. In the following, we show the homomorphism from the semiring of regular expressions to the why-provenance semiring, the minimal witness semiring and the lineage semiring.

Why-provenance semiring. Why-provenance semiring records the set of sets of edges that contribute to the query results. Then one can verify that $(2^{2^{V\times\Sigma\times V}}, \cup, \Cup, \varnothing, \{\varnothing\})$ is a commutative idempotent semiring.

To do the homomorphism from the semiring of regular expressions to Why semiring, we define a mapping $h : \mathcal{E}(\mathsf{G}) \to 2^{2^{V\times\Sigma\times V}}$ by $h(\boldsymbol{\varnothing}) = \varnothing$, $h(\boldsymbol{\epsilon}) = \{\varnothing\}$, $h(\mathbf{a}) = \{\{\mathsf{a}\}\}$, $h(E + F) = h(E) \cup h(F)$, $h(E \cdot F) = h(E) \Cup h(F)$, and $h(E^\star) = h(E) \cup \{\varnothing\}$. Then h is a semiring homomorphism from $([\mathcal{E}(\mathsf{G})], +, \cdot, \star, \boldsymbol{\varnothing}, \boldsymbol{\epsilon})$ to $(2^{2^{V\times\Sigma\times V}}, \cup, \Cup, \varnothing, \{\varnothing\})$.

Minimal witness basis semiring. Minimal witness basis semiring $(irr(E), \oplus, \otimes, \varnothing, \{\varnothing\})$ records the sets of minimal witnesses that contribute to the query results.

To do the homomorphism from the semiring of regular expressions to Minimal witness semiring, we define a mapping $h : \mathcal{E}(\mathsf{G}) \to irr(E)$ by $h(\boldsymbol{\varnothing}) = \varnothing$, $h(\boldsymbol{\epsilon}) = \{\varnothing\}$, $h(\mathbf{a}) = \{\{\mathsf{a}\}\}$, $h(I + J) = irr(h(I) \cup h(J))$, $h(I \cdot J) = irr(h(I) \Cup h(J))$, and

$h(I^*) = \{\varnothing\}$. Then h is a semiring homomorphism from $([\mathcal{E}(\mathsf{G})], +, \cdot, \star, \boldsymbol{\varnothing}, \boldsymbol{\epsilon})$ to $(irr(E), \oplus, \otimes, \varnothing, \{\varnothing\})$.

Lineage semiring. Lineage semiring $(2^{V\times\Sigma\times V}, \cup, \cup, \varnothing, \varnothing)$ records the set of edges contributing to the query results, To apply the homomorphism from the semiring of regular expressions to Lineage semiring, we define a mapping $h : \mathcal{E}(\mathsf{G}) \to 2^{V\times\Sigma\times V}$ by $h(\boldsymbol{\varnothing}) = \varnothing$, $h(\boldsymbol{\epsilon}) = \{\varnothing\}$, $h(\mathbf{a}) = \{\mathsf{a}\}$, $h(E + F) = h(E \cdot F) = h(E) \cup h(F)$, and $h(E^*) = h(E)$. Then h is a semiring homomorphism from $([\mathcal{E}(\mathsf{G})], +, \cdot, \star, \boldsymbol{\varnothing}, \boldsymbol{\epsilon})$ to $(2^{V\times\Sigma\times V}, \cup, \cup, \varnothing, \varnothing)$.

3.2 Proposed solution

Computing the product graph followed by the state elimination can generate the equivalent regular expression of the $Q(G)$. The basic idea is to keep the final states in the product graph separate with no outgoing edges, and that can be ensured by prepossessing. Then due to the fact that the removal sequence affects result regular expressions significantly, we choose a heuristic that has been proven to have the best performance on average. Finally, to deal with multidimensional provenance, we use the parse tree technique to do the homomorphism and compute the result value. Therefore, the proposed algorithm can be split into three phases, preprocessing the query FA, computing $Q(G)$, and post-processing the result regular expressions.

3.2.1 Preprocessing the Query FA

The first phase is to generate a FA from the given RPQ, and here we use the concept of Glushkov automata (Caron & Ziadi, 2000), and Glushkov Construction Algorithm (Glushkov, 1961; McNaughton & Yamada, 1960). Given a regular expression with size n, Glushkov's algorithm constructs an NFA with $n+1$ states and it has no $\epsilon-$transitions by the following steps:

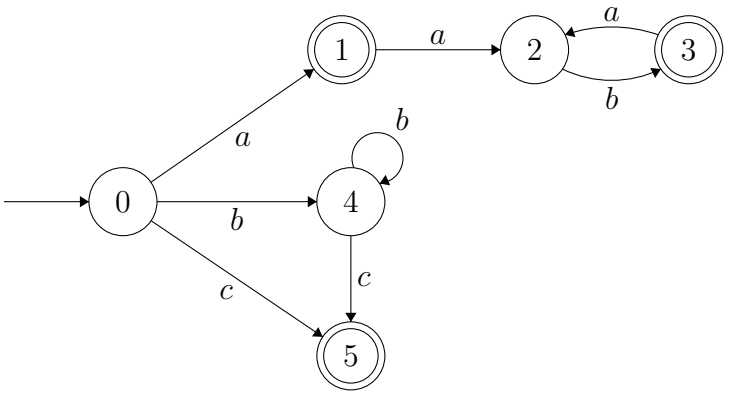
1. Linearize the regular expression by adding index to the symbols in the regular expression. For example, $E = a(ab)^* + b^*c$ becomes $E' = a_1(a_2b_3)^* + b_4^*c_5$. We use a set $Pos(E)$ to record the positions for this expression, and a mapping function h to map the position to the symbol itself. In this example, $Pos(E) = \{1, 2, 3, 4, 5\}$

2. Compute the sets $First, Last, Follow$, where $First$ is the set of letters that occurs as first letter of a word in $L(E')$, $Last$ contains the set of letters that end a word of $L(E')$, and $Follow$ is the set of pair of symbols that occur in words of $L(E')$. From the previous example, we can get $First = \{1, 4, 5\}$, $Last = \{1, 3, 5\}$, and $Follow = \{12, 23, 32, 44, 45\}$.

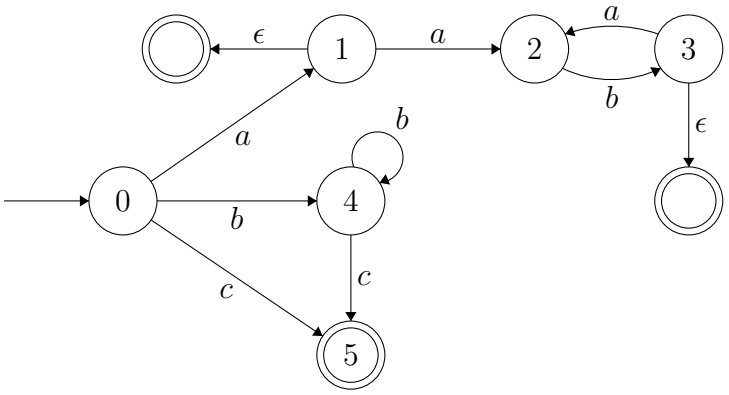
3. Compute the FA that represents the regular expression. The result FA is $M = (Q, \Sigma, \{0\}, F, \delta)$, where $Q = Pos(E) \cup \{0\}$ such that 0 is not in $Pos(E)$, and the set of final states is $Last$ if $\epsilon \notin L(E)$ and $Last \cup \{0\}$ otherwise. The transition function $\delta = \{(x, h(y), y) : x \in Pos(E) \text{ and } xy \in F\} \cup \{(0, h(x), x) : x \in First\}$.

Then for each final state q_f that has outgoing edges, we introduce new final state

q'_f and new edges $e = (q_f, \epsilon, q'_f)$, and mark q_f back to non-final state. The identifier of q'_f is related to q_f for further processing. Figure 11(a) shows the result FA of example regular expression $E = a(ab)^* + b^*c$. Figure 11(a) contains 3 final states, 1, 3 and 5, where 1 and 3 have outgoing edges. To process the FA, we introduce two new final states and add edges from 1 and 3 to them with a label ϵ respectively. Figure 11(b) shows the one after adding final states and edges to the previous FA.



(a) Result Glushkov FA



(b) After preprocessing

Figure 11: A Glushkov finite automata

3.2.2 Computing the Provenance

The second phase is to compute the product graph of a given graph database and the query FA we get from Phase 1. Then we do the state elimination, using the state weight heuristic.

A *Finite State Automata (FA)* is a 5-tuple $A = (P, \Sigma, \delta, p_s, F)$, where P is a finite set of states, Σ is a finite set of alphabet symbols (edge labels), $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $p_s \in Q$ is the start state, and $F \subseteq P$ is a set of final states. Let $\mathtt{w} = \mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n$ where $\mathtt{a}_i \in \Sigma$, be a word in Σ^*. An *accepting computation path* of $\mathtt{w}$ in A is a sequence $(p_s, \mathtt{a}_1, p_1)(p_1, \mathtt{a}_2, p_2) \ldots (p_{n-1}, \mathtt{a}_n, p_f)$ of tuples of δ, where $p_f \in F$. The *language* accepted by A, denoted $L(A)$, is the set of all words in Σ^* for which there exists an accepting computation path in A.

We now consider the problem of computing $Q(G)$ and the provenance for each object $v \in Q(G)$. We start with an FA $A_Q = (P, \Sigma, \delta, p_s, \{p_f\})$, where $L(A_Q) = L(Q)$. Without loss of generality we assume that A_Q has no transitions into its start state, that there is a single final state, and that the final state does not have any outgoing transitions. Given a graph database $G = (V, E, v_s, w_1, \ldots, w_k)$, where v_s is the source node, and w_1, w_2 ... w_k are weighted functions, we first consider G as an FA $A_G = (V, \Sigma, E, v_s, V)$. We then construct the product automaton $A_Q \times A_G = (P \times V, \rho, (p_s, v_s), \{p_f\} \times V)$, where

$$\rho = \{(p, u), \mathtt{a}, \bar{w}, (q, v)) : (p, \mathtt{a}, q) \in \delta, (u, \mathtt{a}, \bar{w}, v) \in E\}.$$

It is a well known fact that $L(A_Q \times A_G) = L(A_Q) \cap L(A_G)$. Then we can define the query result as

$$\mathsf{eval}_Q(G) = \{v : \pi \text{ is an accepting computation path in } A_Q \times A_G \text{ and } \mathsf{last}_\pi = (p_f, v)\}.$$

Proposition 2 $\mathsf{eval}_Q(G) = Q(G)$,

Proof: Let $v \in \mathsf{eval}_Q(G)$ with accepting computation path

$$\pi_1 = ((p_s, v_s), \mathtt{a}_1, \bar{w}_1, (p_1, v_1)) \ldots ((p_{n-1}, v_{n-1}), \mathtt{a}_n, \bar{w}_n, (p_f, v))$$

in $A_Q \times A_G$. From the construction of $A_Q \times A_G$ it follows that the path $\pi_2 = (v_s, \mathtt{a}_1, \bar{w}_1, v_1) \ldots (v_{n-1}, \mathtt{a}_n, \bar{w}_n, v)$ is an accepting computation path in A_G and thus a path in G. Since $\mathsf{last}_{\pi_2} = v$, $\mathsf{first}_{\pi_2} = v_s$, and $\mathsf{word}_{\pi_2} = \mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n \in L(Q)$, it follows that $v \in Q(G)$. The inclusion $Q(G) \subseteq \mathsf{eval}_Q(G)$ is shown similarly.

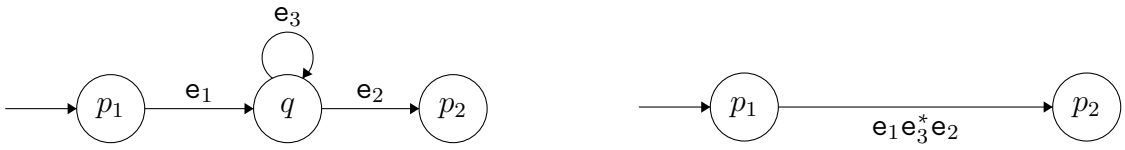
Next, we consider the problem of computing the provenance polynomial $\mathtt{e}_v$ for $v \in \mathsf{eval}_Q(G)$. For this we use an extension of the well known state-elimination procedure for obtaining a regular expression from an FA $A = (P, \Sigma, \delta, p_s, F)$ (Brzozowski & McCluskey, 1963). The basic method is based on viewing the label $\mathtt{a}$ in a transition $(p, \mathtt{a}, q)$ as a regular expression denoting the language $\{\mathtt{a}\}$. For each final state $p_f \in F$ we construct a regular expression $\mathtt{e}_{p_f}$ by removing all intermediate state $q \in P \setminus \{p_s, p_f\}$ one by one until only the initial state p_s and the final state p_f are left. For each pair $(p_1, \mathtt{e}_1, q), (q, \mathtt{e}_2, p_2)$ of transitions, we replace them with the transition $(p_1, \mathtt{e}_1\mathtt{e}_2, p_2)$,

as shown in Figure 12.

(a) Before eliminating q (b) After eliminating q

Figure 12: A state elimination example

If there is a self-loop (q, e_3, q), the replaced transition is $(p_1, \mathsf{e}_1\mathsf{e}_3^*\mathsf{e}_2, p_2)$, which is shown below in Figure 13.



(a) Before eliminating q (b) After eliminating q

Figure 13: A state elimination example where the intermediate state invovles self-loop

If there is another transition (p_1, e_4, p_2) this is merged with the new transition into $(p_1, \mathsf{e}_1\mathsf{e}_3^*\mathsf{e}_2 + \mathsf{e}_4, p_2)$, as Figure 14 displays.

(a) Before eliminating q (b) After eliminating q

Figure 14: A state elimination example where multiple paths are invovled

After all pairs $(p_1, \mathsf{e}_1, q), (q, \mathsf{e}_2, p_2)$ have been processed, the state q with its associated edges are removed. When all states in $P \setminus \{p_s, p_f\}$ are removed and transitions are replaced, we are left with (p_s, e_1, p_s), (p_s, e_2, p_f), (p_f, e_3, p_f), (p_f, e_4, p_s), for some regular expressions $\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3$, and e_4. This gives us regular expression

$\mathsf{e}_{p_f} = (\mathsf{e}_1 + \mathsf{e}_2\mathsf{e}_3^*\mathsf{e}_4)^*\mathsf{e}_2\mathsf{e}_3^*$. Figure 15 illustrates the final step. It is well known that $L(\mathsf{e}_{p_f}) = L(A_{p_f})$, where $A_{p_f} = (P, \Sigma, \delta, p_s, \{p_f\})$. The final regular expression for A is then $+_{p_f \in F}\ \mathsf{e}_{p_f}$.

(a) Before computing (b) After computing

Figure 15: A state elimination example where only the initial state and final state are left

To compute the provenance polynomial e_v for $v \in Q(G)$, we start with the FA $A_Q \times A_G$. First we replace each transition $((p,s), \mathsf{a}, \bar{w}, (q,t)) \in \rho$ with transition $((p,s),(s,\mathsf{a},\bar{w},t),(q,t))$. Here $(s,\mathsf{a},\bar{w},t)$ is to be seen as an atomic regular expression in $\mathcal{E}(G)$ over edges. Let's call this FA B. Then we use the state-elimination technique to each FA $B_{(p_f,v)}$, where $(p_f, v) \in \{p_f\} \times V$, obtaining regular expression $\mathsf{e}_v = +_{(p_f,v)\in\{p_f\}\times V}\ \mathsf{e}_{(p_f,v)}$

Proposition 3 *For $v \in Q(G)$, $L(\mathsf{e}_v) = \mathcal{P}_v$.*

Proof: Let $\pi \in \mathcal{P}_v$. Then π is a path in G, $\mathsf{first}_\pi = v_s$, $\mathsf{last}_\pi = v$, and $\mathsf{word}_\pi \in L(Q)$. By construction, there is then an accepting computation path

$$\pi_1 = ((p_s, v_s), \mathsf{a}_1, \bar{w}_1, (p_1, v_1)) \ldots ((p_{n-1}, v_{n-1}), \mathsf{a}_n, \bar{w}_n, (p_f, v))$$

in $A_Q \times A_G$, and consequently

$$\pi_2 = ((p_s, v_s), (v_s, \mathsf{a}_1, \bar{w}_1, v_1), (p_1, v_1)) \ldots ((p_{n-1}, v_{n-1}), (v_{n-1}, \mathsf{a}_n, \bar{w}_n, v), (p_f, v))$$

is an accepting computation path in B. Then $\mathsf{word}_{\pi_2} \in L(B)$. Since $L(B) = L(\mathsf{e}_v)$ and $\mathsf{word}_{\pi_2} = \pi$ it follows that $\pi \in L(\mathsf{e}_v)$. Similarly, it is shown that $L(\mathsf{e}_v) \subseteq \mathcal{P}_v$.

3.2.3 Determining State Removal Order

The size of the regular expression resulting from the state removal algorithm heavily depends on the order in which states are removed. Several heuristics for determining the removal order have been studied [7, 16, 20]. Experiments show that the heuristics proposed by Delgado and Morais [7], the DM algorithm, produces the most compact regular expressions. The DM algorithm repeatedly computes the weight of each state and eliminates the state with the smallest weight. For states q_1 and q_2, let $\mathrm{len}(q_1, q_2)$ denote the length of the label of the transition from q_1 to q_2. For a state q, let $p_1, \ldots, p_m$ be the states from which there is a transition into q, and $r_1, \ldots, r_n$ states into which there is an outgoing transition from q. At each intermediate step of the state elimination algorithm, the weight of state q is computed as

$$\mathrm{weight}(q) = (n-1) \times \sum_{i=1}^{m} \mathrm{len}(p_1, q) + (m-1) \times \sum_{i=1}^{n} \mathrm{len}(q, r_i) + (m \times n - 1) \times \mathrm{len}(q, q).$$

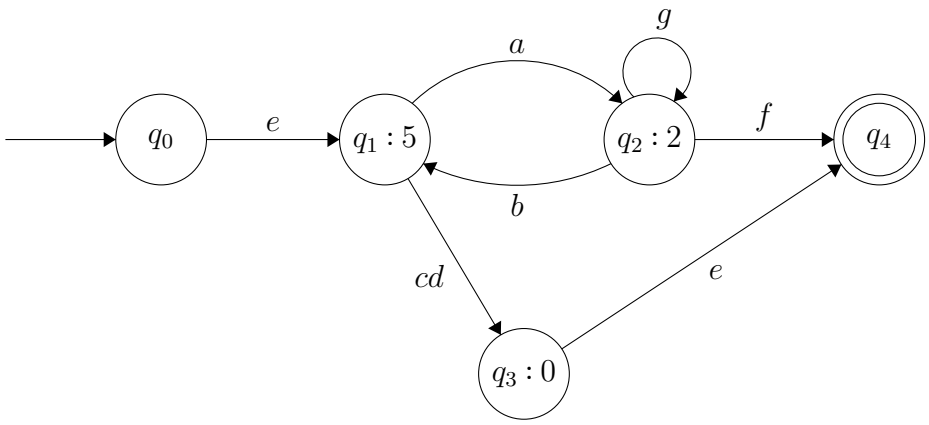


Figure 16: A generalized transition graph

Figure 16 is a FA with regular expressions as edge labels, which is in an intermediate state for state removal, and we use the state weight heuristic DM to compute the removal sequence. Using state q_2 as an example, first we get the number of incoming edges and outgoing edges from q_2, that is, $m = 1$ and $n = 2$. Also, q_2 has a loop of length 1. The weight of q_2 is determined as $\text{weight}(q_2) = (1-1)\times 1+(1-1)\times 1+(2-1)\times 1+(1\times 2-1)\times 1 = 2$. Similarly, we determine the weights of q_1 and q_3 as 5 and 0, respectively. This suggests the state order removal $q_3 \to q_2 \to q_1$, which yields the regular expression $e(ag^*b)^*(ag^*f + cde)$, with length 26. Note that in this example, using the removal order $q_1 \to q_2 \to q_3$ would yield the regular expression $eb(g + cb)^*f + (ecd + eb(g + cb)^*ccd)e$, whose length is 43.

Yann et al. (Ramusat et al., 2021) proposed a new state removal heuristic based on the degree of each node. The degree of each node is defined as the sum of incoming and outgoing edges from the node, and recomputed after a node with the minimum degree is eliminated. Using the same example in Figure 16, the degree of q_1, q_2 and q_3 initially are 4, 5 and 2, respectively. After eliminating the node q_3, which has the minimum degree, the degree of q_1 and q_2 stays the same, which leads to removing q_1

next and q_2 in the end. The result regular expression is $eb(g + cb)^*(f + ccde) + ecde$, which is of length 32.

3.2.4 Post-processing the Provenance Polynomials

The last phase includes the conversion from the regular expression of each destination node to parse tree, from which we can compute the actual result over different semirings.

In (Green, Karvounarakis, & Tannen, 2007), the most general form of provenance over relational databases is polynomials, which can be projected to different semirings by homomorphism. The difference between polynomials and regular expressions is that the latter does not support commutativity since it requires a certain order for edges in each path. Given the fixed source node n_1, the provenance of a target node n_2 is $r = (n_1, a, 1, 0.6, n_3)(n_3, c, 2, 0.2, n_3)^*(n_3, b, 3, 0.5, n_2) + (n_1, a, 2, 0.2, n_2)$, wherein each tuple, the second element is an edge label, and the third element belongs to tropical semiring while the fourth element belongs to influential semiring. Now if we want to compute the shortest distance from node n_1 to n_2, we will take the homomorphism from r to tropical semiring, by replacing the concatenation operator with plus, the union operator + with min , and the star operator a^* with 0. Then we extract the third element from each tuple in r, and compute the result value by $min(1 + 0 + 3, 2) = 2$. Similarly, if we want to get the influence from node n_1 to n_2, we replace the operators accordingly, then the result influence will be $max(0.6 \times 1 \times 0.5, 0.2) = 0.3$.

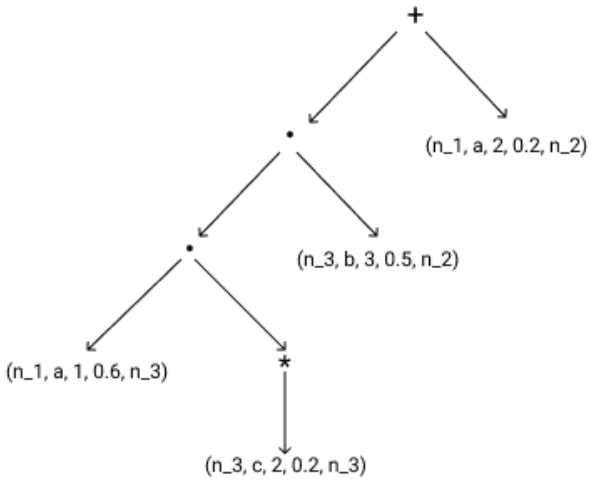


Figure 17: A parse tree

This process can be done by parse tree. After we get a regular expression for each target node, we create parse trees for them separately. For example, the parse tree of the previous regular expression r is shown in Figure 17. Since it allows multidimensional provenance, each leaf node of the parse tree may include multiple elements for different semirings. Then during the homomorphism phase, we traverse the parse tree and calculate the value by replacing operators based on the definition of the semiring we choose and return the final value and its provenance as result. The complete answer of tropical semiring for this example should be $n_2 : 2, (n_1, a, 2, 0.2, n_2)$, while the influence result is $n_2 : 0.3, (n_1, a, 1, 0.6, n_3)(n_3, b, 3, 0.5, n_2)$.

Chapter 4

Implementation

In this chapter, we provide descriptions of procedures and functions used in the development of the programs and used in our experiments to show the practical merits of the ideas and techniques proposed in our work.

The program architecture is shown in Figure 18. Three main blocks are surrounded by dashed frames, which correspond to (1) Phase 1: converting RPQ to FA, (2) Phase 2: eliminating intermediate states, and (3) Phase 3: generating parse trees, from left to right respectively.

Algorithm 1 is our main algorithm *ProvAl*. Given a graph database G, an RPQ Q and a fixed source node s, the *ProvAl* algorithm returns a HashMap, where the key is the target node t that can be reached from the source node, and the value is P_t, which is the provenance from the source node s to t.

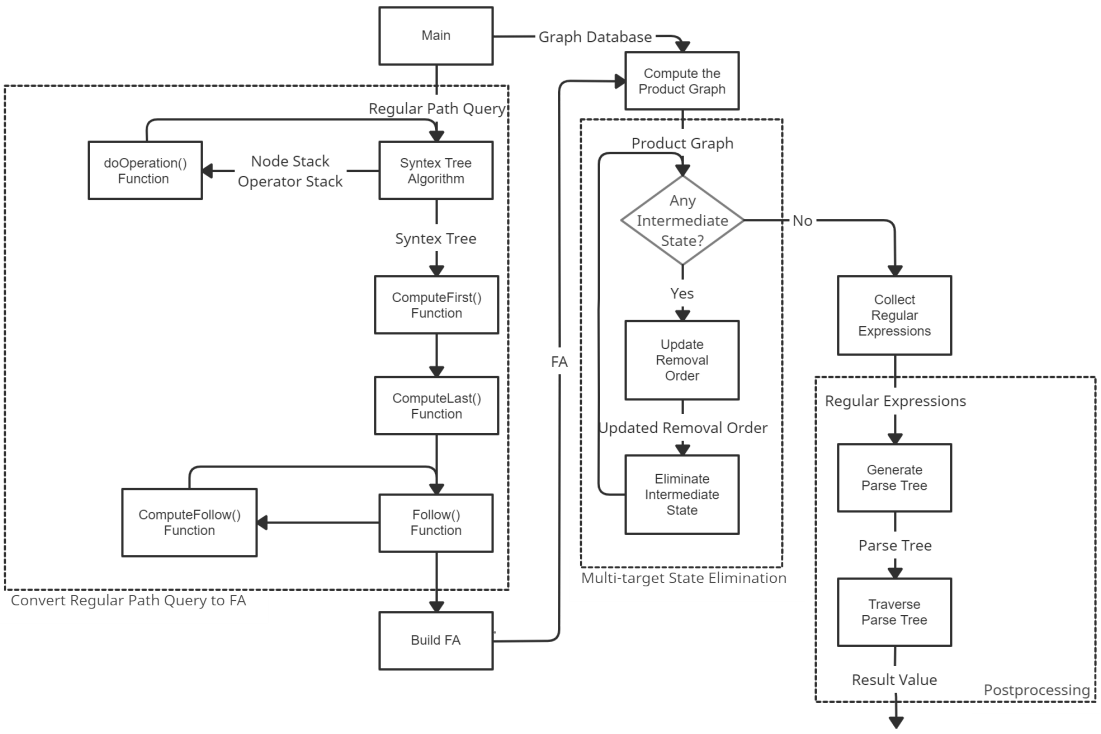


Figure 18: Program architecture

The *ProvAl* algorithm is based on Glushkov's algorithm and the state elimination algorithm, where Glushkov's algorithm is used to construct the automaton from the given query. We modified and used the state elimination algorithm to compute the regular expression from the source node to the end nodes of the product graph. In addition, we applied the state weight heuristic method during the state elimination process, which has been shown to have a better performance than other heuristics on average.

Using the parse-tree technique, we also provide a postprocessing algorithm to deal with multidimensional provenance.

Algorithm 1 The *ProvAl* algorithm

Input: A regular path query Q, and a graph database G with the source node s

Output: A HashMap of provenance regular expressions with the target node as the key and regular expression as the value

1: construct a FA M representing Q, where q_0 is the initial state;
2: preprocess M;
3: build the product graph $A_{M\times G}$ of M and G, where the start node is (s, q_0);
4: apply state elimination until only the source node and final nodes are left, which returns a HashMap *result*;
5: return *result*;

4.1 Phase 1: Converting RPQ to FA

To build a Glushkov FA from the given RPQ Q, we compute three sets $First$, $Last$ and $Follow$, which include the information of the first symbols, last symbols, and two consecutive symbols that appear in $L(Q)$. For computing these sets, we generate a syntax tree based on the given query Q.

4.1.1 Syntax Tree

To create a syntax tree for generating the Glushkov FA, we use two stacks, one for nodes and one for operators, and a set *nodes* which records all the nodes in the syntax tree. Algorithm 2 shows details, where it first adds an index to each symbol in the given query Q (line 2), e.g. $a \cdot b$ to $a_1 \cdot b_2$, and then starts traversing the regular expression Q (lines 3-23).

For each symbol in Q, if the current symbol is a character (from 'a' to 'z'), it

creates a new node with the symbol and its index next to it (line 5-6), pushes it onto the node stack and also adds it to *nodes* (lines 7-8). If the current symbol is a left parenthesis, it simply pushes it to the operator stack (lines 10-11). When the current symbol is a right parenthesis, it will start processing operations according to the operator symbol at the top of the stack (line 14), until it sees a left parenthesis, which is then popped out of the operator stack (line 16).

For the rest of the cases, when the current symbol is one of the operators, it checks if the operator stack is empty or not, and also if the priority of the current operator is lower than the operator at the top of the stack. If so, it pops the operator symbol out of the stack and performs the operation accordingly. This process repeats until the condition is not satisfied anymore, and then it pushes the current operator to the operator stack (lines 18-21). After reading all the symbols in Q, if there are still operators left in the operator stack, it continues to process the operations until all the operators are popped (lines 24-26). In the end, the algorithm returns the set of nodes in the syntax tree and the node at the top of the node stack (line 27), which is the root of the syntax tree.

Algorithm 2 Syntax Tree algorithm

Input: A regular expression Q

Output: A syntax tree representing Q, and a set of nodes *nodes*

```
1: initialize a stack stackNode, operator, and a set nodes;
2: add index to each symbol in Q
3: for symbol s in Q do
4:     if s is a character then
5:         get the index n next to s;
6:         create a new node newNode with value s_n;
7:         add newNode to the node set;
8:         push newNode to the node stack;
9:         i = i + 1;
10:    else if s is the left parenthesis '(' then
11:        push s to the operator stack;
12:    else if s is the right parenthesis ')' then
13:        while the top operator of the operator stack is not left parenthesis do
14:            call function doOperation() and perform the operation;
15:        end while
16:        pop the left parenthesis out of the operator stack;
17:    else
18:        while the operator stack is not empty and if the priority of s is lower than
            the top operator in operator do
19:            call function doOperation() and perform the operation;
20:        end while
21:        push s to the operator stack;
22:    end if
23: end for
24: while there are operators left in the operator stack do
25:    call function doOperation() and perform the operation;
26: end while
27: return the set of nodes nodes and the top node of the node stack;
```

The function *doOperation* that operates for the top operator of the stack is shown

in Algorithm 3. First, it pops the top operator out of the stack and creates a new node with the operator as the value (lines 2-3). If the top operator is '&' or '|', it pops two nodes from the node stack as its left child and right child (lines 5-8). If the top operator is the star '$*$', it only needs to pop one node from the stack and use it as the left child of the new node '$*$' (lines 10-11). In the end, it pushes the new node to the node stack (line 13).

Algorithm 3 doOperation function

Input: The node stack, and the operator stack

```
1:  if the operator stack is not empty then
2:      pop the top operator out of the operator stack as the current operator;
3:      create a new node with value as the current operator;
4:      if the current operator is '|' or '&' then
5:          pop the top node out of the node stack as node_2;
6:          pop the top node out of the node stack as node_1;
7:          set node_1 as the left child of the new node;
8:          set node_2 as the right child of the new node;
9:      else
10:         pop the top node out of the node stack as node;
11:         set node as the left child of the new node;
12:     end if
13:     push the new node to the node stack;
14: end if
```

4.1.2 Computing *First*, *Last* and *Follow*

Here we adopt and implement the rules for computing the sets $First$, $Last$ and $Follow$ proposed in (Caron & Ziadi, 2000), and then apply them to the syntax tree. First, we consider computing the set $First$, which includes all possible symbols that appear as

the first letter in the language $L(Q)$. For this, we use the rules from (Caron & Ziadi, 2000) shown below. Algorithm 4 shows more details when dealing with the syntax tree.

1. $First(\varnothing) = First(\epsilon) = \varnothing$

2. $First(x) = x$

3. $First(F + G) = First(F) \cup First(G)$

4. $$First(F \cdot G) = \begin{cases} First(F), & \text{if } \epsilon \notin L(F) \\ First(F) \cup First(G), & \text{otherwise} \end{cases}$$

5. $First(E^*) = First(E^+) = First(E)$

In Algorithm 4, if the current node has no children and its value is neither null nor empty string, we apply the second rule, $First(x) = x$ (lines 2-5). Otherwise, there are the following three cases, depending on the operator of the current node: (1) if the current node is the star operator '$*$', we need to compute the first letter set of its left child (lines 8-9); (2) if the current node is the union operator '|', then we determine the first letter set of both its left child and right child and add it to the result set (lines 11-13); (3) if the current node is the concatenation operator '&', here if the left child does not generate an empty string, it only computes the first letter set of the left child (lines 15-17). Otherwise, if the left child generates an empty string, it needs to compute the right child and add the result to the result set (lines 18-20).

The rules for computing $Last$ are very similar to those used for $First$, and the only difference is the rule for concatenation.

$$First(F \cdot G) = \begin{cases} First(F), & \text{if } \epsilon \notin L(F) \\ First(F) \cup First(G), & \text{otherwise} \end{cases}$$

Here if the right child can generate an empty string, then both the last letter sets of left and right child should be added to the result set.

Algorithm 4 computeFirst function

Input: the current node
Output: A set of nodes $First$

```
1: if the current node has no children then
2:     if the value of the current node is neither ∅ nor ϵ then
3:         add the current node to First
4:     end if
5:     return First
6: else
7:     if the value of the current node is '*' then
8:         compute the set First_left of the left child of the current node;
9:         set First_left as First of current node;
10:    else if the value of the current node is '|' then
11:        compute the set First_left of the left child of the current node;
12:        compute the set First_right of the right child of the current node;
13:        get the union of First_left and First_right as First
14:    else
15:        compute the set First_left of the left child of the current node;
16:        compute the set First_right of the right child of the current node;
17:        set First_left as First of the current node;
18:        if the left child of the current node can generate an empty string then
19:            add nodes from the First_right to First
20:        end if
21:    end if
22:    return First
23: end if
```

When creating new nodes in the syntax tree and computing $First$ and $Last$, we

also update the $First$ and $Last$ sets for each node, which saves the information of the first and last letters each node can generate. Using these $First$ and $Last$ information, Function 5 traverses the syntax tree and updates the set $Follow$ for each node. First, it calls Function 6 (line 1), and continues to traverse its children on left and right (lines 2-3), or traverses only its left child if the current node is the star operation (line 6).

Algorithm 5 Follow function

```
1: function FOLLOW(curr)
2:     computeFollow(curr);
3:     if the current node curr has left and right child then
4:         Follow(curr.leftChild);
5:         Follow(curr.rightChild);
6:     else if the value of the current node curr is '*' then
7:         Follow(curr.leftChild);
8:     end if
9: end function
```

Function 6 computes the follow set given the current node. If the current node is the concatenation operator '&', it adds the nodes from $First$ of the right child to $Follow$ of each node in $Last$ of the left child (lines 2-8). If the current node is star operator '$*$', then it adds the nodes from $First$ to $Last$, where both $First$ and $Last$ are of the current node (lines 10-14).

Algorithm 6 computeFollow function

```
1: function computeFollow(curr)
2:     if the value of the current node curr is '&' then
3:         get the Last set of the left child of the current node as Last_left;
4:         get the First set of the right child of the current node as First_right;
5:         for each node n_1 in Last_left do
6:             for each node n_2 in First_right do
7:                 add n_2 to the Follow set of n_1;
8:             end for
9:         end for
10:    else if the value of the current node curr is '*' then
11:        for each node n_1 in Last of the current node do
12:            for each node n_2 in First of the current node do
13:                add n_2 to the Follow set of n_1;
14:            end for
15:        end for
16:    end if
17: end function
```

After computing the sets $First$, $Last$ and $Follow$, we can build the FA by initializing $|Q|+1$ nodes with index as the node identifier. The identifier of the initial state is $0 \notin \Sigma$, final states are nodes from $Last$, and transition functions are from $Follow$. For each node $first$ in $First$, there is an edge from 0 to $first$ with $h(first)$ (the symbol of $first$) as the edge label.

Then we can preprocess the FA by making the final states that have no outgoing edges, using Algorithm 7, where if any final state $q_f \in Q_F$ has any outgoing edges, it creates a new final state (line 3), adds an ϵ−transition from q_f to the new final state (line 4), and changes q_f back to the non-final state (line 5).

Algorithm 7 Preprocessing FA algorithm

Input: The RPQ Q which is represented as FA M

Output: A pre-processed FA

```
1: for final state q_f of Q_F, which is a set of final states of M do
2:     if q_f has outgoing edges then
3:         create a new final state q_newFinal;
4:         create a new edge δ(q_f, λ) = q_newFinal;
5:         change q_f to non-final state;
6:     end if
7: end for
8: return M;
```

4.2 Phase 2: Computing the Provenance of Query Results

We now briefly review the State Elimination algorithm, which can be applied to solve the single-source provenance problem. Given a FA, the algorithm first preprocesses the FA to ensure the initial state q_0 does not have incoming edges, which can be done by introducing λ-productions. Then for each final state q_f, we repeat removing the state $q_i \in (Q \setminus U) \setminus \{q_0, q_f\}$, where U is the set of all states that are already removed, while keeping the regular expressions of each path that passes through q_i. For every node q_j where there is an edge $(q_j, q_i) \in E$, q_k where there is an edge $(q_i, q_k) \in E$ and $q_j, q_k \notin U$, we introduce a new edge (q_j, q_k) with $R_{ji}(R_{ii})^* R_{ik}$ as the regular expression of the edge. In the end, when all the states are removed except for the initial state and final state, we can get the regular expression for the paths from q_0 to q_f.

The size of the result regular expression obtained by the state elimination algorithm depends on the order in which the states are removed. Among several heuristics for state removal order (Delgado & Morais, 2005; Han & Wood, 2007; Moreira et al., 2010), the heuristic proposed by Delgado and Morais (Delgado & Morais, 2005) is the most efficient heuristic for the most general case, however, it has some limitations for graph and query size. This algorithm to which we refer as DM determines the weight of each state and eliminates the one with the lowest weight. The experimental results which showed the statistical significance of the DM algorithm can be found in (Gruber, Holzer, & Tautschnig, 2009; Moreira et al., 2010). To determine the weight $\text{weight}(q)$ of a state q at every iteration of the state elimination process, they use the following formula:

$$\text{weight}(q) = (n-1) \times \sum_{i=1}^{m} \text{len}(p_1, q) + (m-1) \times \sum_{i=1}^{n} \text{len}(q, r_i) + (m \times n - 1) \times \text{len}(q, q)$$

which has been explained in detail in Chapter 3.

Function 8 shows details of the DM algorithm. The input includes the current state *currState*, a list of states *visited* whose weight is already computed, and an array *weight* which records the weight of each state. First, it checks if *currState* has already been visited or not (line 1). If not, it adds *currState* to *visited*, and starts computing its weight (lines 2-18). It initializes the number of outgoing and incoming edges, the total weight of incoming and outgoing edges, and the weight of the loop at *currState* (lines 3-8). Then it computes the weight of incoming edges

(lines 9-11), outgoing edges (lines 13-15), and the loop (line 17). After summing these three weights up, it updates the weight of *currState* in *weight* (line 18), and keeps computing if *currState* has any outgoing edges to other states (lines 19-23).

Algorithm 8 State weight function

```
1: function stateWeight(currState,visited,weight)
2:     if currState is not in visited then
3:         add currState to visited;
4:         in = 0, out = 0, weight_in = 0, weight_out = 0, weight_loop = 0, loop_len = 0;
5:         get all states that have an edge to currState as incoming;
6:         get all states that currState has an edge to as outgoing;
7:         in = incoming.size;
8:         out = outgoing.size;
9:         get the length of the regular expression of the loop on currState as loop_len;
10:        for state s in incoming do
11:            get the length of edge label between (s, currState) as len;
12:            weight_in += len × (out − 1);
13:        end for
14:        for state t in outgoing do
15:            get the length of the edge label between (currState, t) as len;
16:            weight_out += len × (in − 1);
17:        end for
18:        weight_loop = loop_len × (in × out − 1);
19:        weight[currState] = weight_in + weight_out + weight_loop;
20:        if outgoing.size > 0 then
21:            for state next in outgoing do
22:                call updateOrder(next);
23:            end for
24:        end if
25:    end if
26: end function
```

In our work, we combined using the state elimination algorithm, with the DM heuristic. Our goal is to compute the provenance of each target node t, where s is the single source given from the graph database. Rather than repeating the state elimination process for each target node, in this thesis, we keep the initial state and multiple final states that do not have outgoing edges when performing the state elimination. The label of the edge from the initial state s to any one of the final states t represents the provenance of t, and no outgoing edges from final states ensures that we do not need to make a removal sequence for final states.

Algorithm 9 State elimination algorithm

Input: The RPQ Q which is represented as FA M, the graph database G

Output: A list of provenance regular expressions of the form $t : r$

1: initialize a HashMap $resultRegexList$;
2: compute the product graph $A_{M\times G}$;
3: **for each** final state $q_f \in A_{M\times G}$ **do**
4: $\quad prov_{q_f} = StateElimination(A_{M\times G})$;
5: $\quad$ add q_f and $prov_{q_f}$ to $resultRegexList$;
6: **end for**
7: return $resultRegexList$;

Algorithm 9 shows details of the original state elimination algorithm, while Algorithm 10 shows the new one. In Algorithm 9, we need to perform the state elimination for each final state (lines 3-6). However in Algorithm 10, since we already preprocess the FA that represents the query, there will not be any final states with outgoing edges in the product graph, therefore we can directly perform the state elimination algorithm only once (line 2).

Algorithm 10 New state elimination algorithm

Input: The RPQ Q which is represented as FA M, the graph database G

Output: A list of provenance regular expressions of the form $t : r$

1: compute the product graph $A_{M \times G}$;
2: $resultRegexList = SE(A_{M \times G})$;
3: return $resultRegexList$;

Algorithm 11 presents the state elimination part (lines 4-12) and generates the final regular expressions for each target node (lines 13-25). First, it initializes a list of visited nodes and a Hashmap for collecting result regular expressions (lines 1-2). While there is any non-initial non-final state left in the product graph, it updates the removal sequence first (line 4), and chooses the first unvisited node from the removal order, and eliminates it from the product graph (lines 6-9). The steps of removing an intermediate state and introducing new edges are described in Algorithm 12. Once all nodes except the initial and final nodes are removed, it traverses the nodes that are in the set of final states, extracting their regular expressions from the start node (line 14). Since every node in the product graph is consisting of a graph node and a FA state, we need to combine the regular expressions for the same graph node together (lines 16-24). In the end, Algorithm 11 returns the Hashmap $resultRegex$ that records all provenance information of all target nodes (line 26).

Algorithm 11 Multi-target state elimination algorithm

Input: the product graph M

Output: A list of provenance regular expressions

```
1: initialize a new list visited;
2: initialize a new HashMap resultRegex;
3: while there is any intermediate state left do
4:     update the removal order eliminateOrder;
5:     for currState in eliminateOrder do
6:         if currState is neither initial nor final and currState is not in visited then
7:             add currState to visited;
8:             eliminate currState;
9:             break;
10:        end if
11:    end for
12: end while
13: for state finalState in finalStates do
14:    get the regular expression R of paths from M.initialState to finalState;
15:    get the id of finalState as stateId;
16:    if resultRegex already records the provenance of finalState then
17:        prevRegex = resultRegex.get(stateId);
18:        prevRegex.add(R);
19:        resultRegex.put(stateId, prevRegex);
20:    else
21:        initialize a new list newRegexList;
22:        newRegexList.add(R);
23:        resultRegex.put(stateId, newRegexList);
24:    end if
25: end for
26: return resultRegex
```

Function 12 describes details of eliminating the intermediate state. Given the current intermediate state *currState*, it gets the set of incoming states as *incoming*, the set of outgoing states as *outgoing*, and the regular expression of the loop at *currState* as $prov_{currState,currState}$ (lines 1-3). For each state $from$ in *incoming* and each state

to in *outgoing*, it generates the regular expression using $prov_{from,to}$, $prov_{from,currState}$, $prov_{currState,currState}$, and $prov_{currState,to}$ (lines 6-7). If states $from$ and *to* are the same, it updates the regular expression of the loop at state $from$ (line 9). Otherwise, it updates the regular expression of the edge starting from state $from$ to state *to* (line 11). Then it deletes the edge between state $from$ and *currState* (lines 13), and between *currState* and *to* (lines 15) respectively. The complexity of the revised state elimination algorithm is $O(n^3)$.

Algorithm 12 Eliminate intermediate state function

```
1: function ELIMINATESTATE(currState)
2:     get all states that have an edge to currState as incoming;
3:     get all states that currState has an edge to as outgoing;
4:     get the loop of currState as prov_{currState,currState};
5:     for state from in incoming do
6:         for state to in outgoing do
7:             get the regular expressions prov_{from,to}, prov_{from,currState}, prov_{currState,to};
8:             newRegex = prov_{from,to}+prov_{from,currState}prov*_{currState,currState}prov_{currState,to};
9:             if from and to are the same state then
10:                set newRegex as the regular expression of loop of from;
11:            else
12:                set newRegex as the regular expression of the edge from from to
    to;
13:            end if
14:            delete currState from the incoming states of to;
15:        end for
16:        delete currState from the outgoing states of from;
17:    end for
18: end function
```

4.3 Phase 3: Postprocessing the Multidimensional Provenance

We used Antlr[1] in our research to implement the construction of parse trees and the traversal visitors. Antlr is a versatile parser generator for processing text given an input grammar and strings or text file. It generates a tree based on the grammar and the input string, and then with a customized visitor, it traverses the tree with defined operators in the visitor. It then returns the result value, and also the associated regular path expression that leads to the result. One advantage of this visitor interface is that we can define different visitors to deal with different semirings, which can be defined explicitly in the grammar. Here we present the grammar we use for the parse tree. The rules for *expr* include the concatenation, union and star operations, as well as the parenthesis, and the single edge regular expression (*tuple*). We also define several terminals, such as *tuple* that can include more than one semiring value, $PCSTO$ that is used for access control semiring, and INT and $FLOAT$ for integers and float numbers, respectively.

```
grammar Semi;
prog
    : expr                      #printExpr
    ;
expr
    : expr '*'                   #aster
        | expr '.' expr    #concate
        | expr '+' expr  #or
        | tuple              #tup
```

[1] https://www.antlr.org/

```
        | LP expr RP       #parens
    ;
tuple
    : LP nodef=INT COMMA label=LABEL COMMA w1=FLOAT COMMA w2=INT
       COMMA nodet=INT RP
    ;
LP  : '(' ;
RP  : ')' ;
PCST0  : [PCSTN];
INT: [0-9]+;
LABEL : ([a-z]|'\u03BB');
FLOAT : INT '.' INT;
COMMA: ',';
WS  : [ \t\r\n]+ -> skip ;
```

The customized visitor realizes the homomorphism from the semiring of regular expressions $\mathcal{K}_{\mathcal{E}(G)} = (\mathcal{E}(G), +, \cdot, *, \varnothing, \epsilon)$ to a certain application semiring. To be more specific, it implements how the operators $\cdot$, $+$, and $*$ work when traversing the parse tree. The visitor follows the structure of grammar that is shown above, so it needs to override the functions of visiting *expr*, the star operation, the concatenation operation, the union operation, the edge regular expression *tuple*, and the parentheses. Since we need to record the set of paths that lead to the result value (e.g. the shortest distance), we also include the regular path expression when traversing the parse tree.

Chapter 5

Experiments

In this chapter, we provide details of our experiments and report the effectiveness and scalability results. In the experiments, we use both real-world and synthetic graph databases. We also study the effect of queries with different sizes and structures, and experimentally evaluate and compare the performance of two state removal heuristics.

5.1 Equipment and Datasets

The computer we use for the experiments is a typical PC which has 16GB of RAM and runs Intel Core i5-8400 CPU.

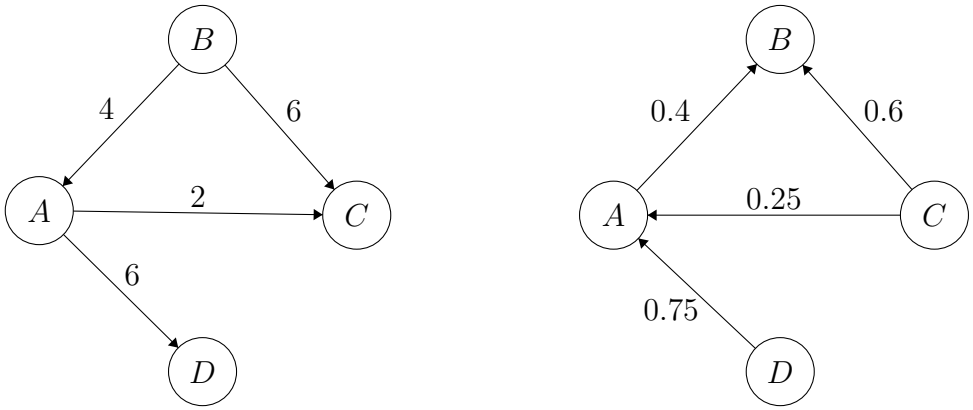
For the datasets, we use two real-world sparse graph databases, the Retweet network (De Domenico, Lima, Mougel, & Musolesi, 2013) from SNAP[1], and the Yeast protein-to-protein interaction network (Maniu, Senellart, & Jog, 2019). We start with sparse graph databases to see how our algorithm performs. Since many real-world

[1]http://snap.stanford.edu/data/index.html

networks are sparse, such as social, computer, and biological networks, as well as transportation and citation networks, etc., focusing on sparse graph databases can lead to some practical potentials. We add different annotations to these datasets based on different uses. In our experiments, we considered the tropical semiring, the influence semiring and the access control semiring. We also created and used several synthetic graph datasets with fixed alphabet size as 10. The yeast graph database has 2,361 nodes and 7,182 edges.

Here we briefly describe the preprocessing steps for the retweet network data, we use for the experiments. The original retweet network dataset is a directed graph that contains 256,491 nodes and 328,132 edges, where each edge (A, n, B) indicates that A retweets n times the posts from B. To build the influence graph based on the original retweet dataset, we use the formula from (Hangal et al., 2010). To compute the influence of user A on user B, it divides the numbers of times of B retweeting A by the total number of retweets from B. The new directed influence graph consists of edges (A, B) that shows the influence value of user A on user B.

$$Influence(A, B) = \frac{Retweets(B, A)}{\sum_X Retweets(B, X)}$$



(a) Retweet Network (b) Influence network

Figure 19: Retweet data (a) and the corresponding influence data (b)

Figure 19(a) shows an example of a given retweet network and Figure 19(b) shows the influence graph obtained. The origin retweet network data has four users A, B, C, and D, and each edge from user x to y represents how many times x retweets y. It can be seen that B retweets A 4 times and retweets C 6 times. So to compute the influence of A and C to B, first we determine the number of retweets from B, which is 10. Then using the formula above, we get $Influence(A, B)$ = 4/10 = 0.4, and $Influence(C, B)$ = 6/10 = 0.6. Similarly, the influence of D to A obtained is 0.75, while the influence of C to A is 0.25. Figure 19(b) shows the complete influence graph data.

5.2 Results and Analysis

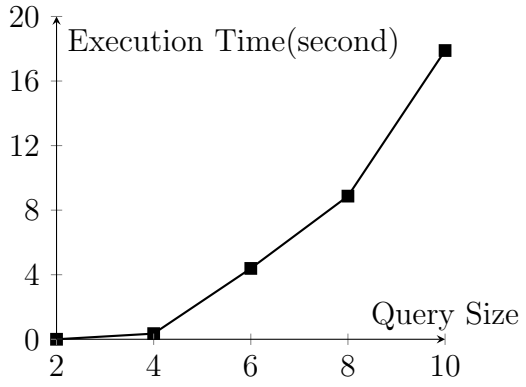
Our experimental study focuses on studying the feasibility and scalability of our proposed state elimination algorithm. We measure the execution time and the size of

the resulting regular expressions. We also compare the heuristics of state elimination order between the DM algorithm and degree-based algorithm, introduced earlier in Section 3. Since we consider the single-source problem, the results may be less convincing if we only pick one or two nodes as the source node, especially when the input graph database is large. Therefore, we record the results of using every node as the source node, and report the average as the result for the smaller size of graph databases. For larger graph databases, we take half or one-third of the nodes as the source node and report the average results. We used three different queries for each test.

5.2.1 Main Results

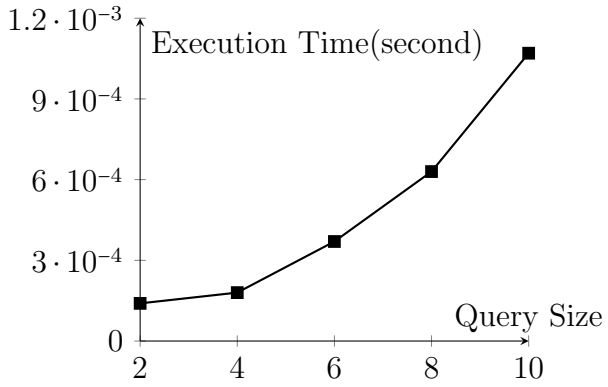
As can be seen from the experiment results shown in Figures 20 and 22, for the Retweet graph and the synthetic datasets, the time to build the parse tree is negligible, compared to the time for computing the query results, which involves building the product graph and performing state elimination. This implies that after we get all pairs of nodes with a fixed source node, computing certain information, such as the shortest path or access level, between the source node and one of the target nodes can benefit by using parse trees. However the result of the Yeast network in Figure 21 is an exception from the conclusion above, where the time for computing product graph and performing state elimination is close to the time for parse trees, and the execution times are relatively small comparing to the execution times of other datasets. This result is due to the graph structure of the Yeast network, which leads to a smaller

size of product graph and resulting regular expressions.



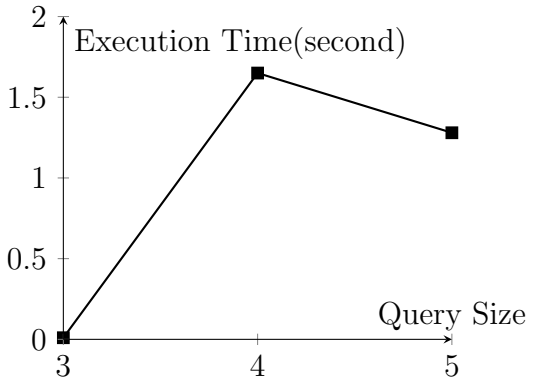
$\|Q\|$	Time for SE(s)	Time for Parse(s)	Length of RE
2	0.00010	0.00001	20
4	0.35	0.00003	64
6	4.39	0.00004	79
8	8.87	0.00004	109
10	17.89	0.00003	117

Figure 20: Experiment results of Retweet Network

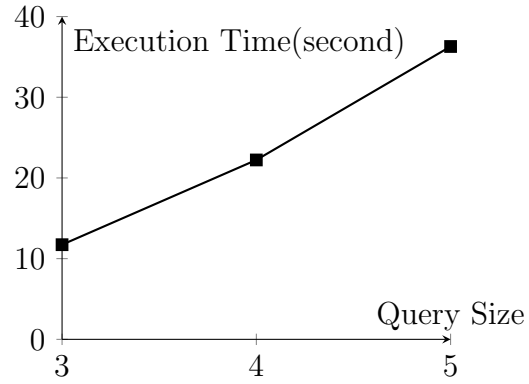


$\|Q\|$	Time for SE(s)	Time for Parse(s)	Length of RE
2	0.00014	0.00011	16
4	0.00018	0.00009	26
6	0.00037	0.00007	48
8	0.00063	0.00005	59
10	0.00107	0.00004	56

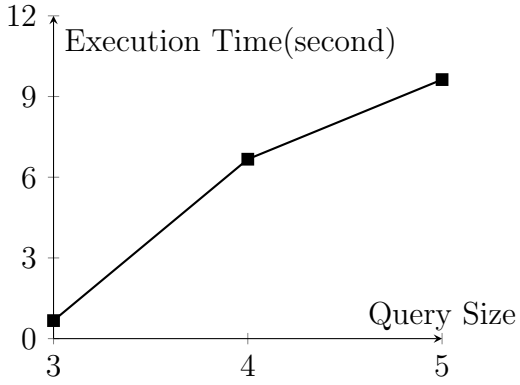
Figure 21: Experiment results of Yeast Network



(a) $|V| = 1,000$, $|E| = 10,000$



(b) $|V| = 5,000$, $|E| = 50,000$



(c) $|V| = 10,000$, $|E| = 100,000$

Graph Size	$\|Q\|$	Time for SE(s)	Time for Parse(s)	Length of RE
$\|V\| = 1,000$, $\|E\| = 10,000$	3	0.01	0.00004	273
	4	1.65	0.00011	939
	5	1.28	0.00009	930
$\|V\| = 5,000$, $\|E\| = 50,000$	3	11.73	0.00055	6688
	4	22.24	0.00083	12572
	5	36.29	0.00082	12145
$\|V\| = 10,000$, $\|E\| = 100,000$	3	0.67	0.00006	106
	4	6.67	0.00028	184
	5	9.63	0.00037	195

(d)

Figure 22: Experiment results of synthetic graph datasets

Figures 20, 21, and 22(a) - (c) show the execution time of querying five different

datasets, two of which are real-world graph databases: Figure 20 is the retweet graph, and Figure 22 is the yeast graph. Figures 22(a) - (c) are synthetic graph databases of different sizes: 22(a) with size $|V| = 1,000, |E| = 10,000$, 22(b) with size $|V| = 5,000, |E| = 50,000$, and 22(c) with size $|V| = 10,000, |E| = 100,000$. For the real-world graph databases, the execution time increases as the query size grows, and the growth is not linear, as expected since the complexity of the state elimination algorithm is exponential. However, from Figures 22(a) and 22(c) we can see that for some data, the execution time is approaching linear, which indicates the potential and suitability of the proposed new state elimination algorithm.

5.2.2 Comparison of Different Types of Queries

We also compare the execution time of different types of queries, using the synthetic graph with 1,000 nodes and 10,000 edges. The result is shown in Figure 23.

Here we mainly compare five different types of queries within the same query size, and give a generalized form for each type as follows. Additional information can be found in Table 3:

1. the loop in the query has multiple edge labels, e.g., $(a + b)^* c$;
2. the loop in the query only has one edge label, e.g., $ab^* c$;
3. the loop in the query only has two consecutive edge labels, e.g., $(ab)^* c$;
4. the query has two or more branches, where the size of the longest component is of size 2, e.g., $a^* b + c$;

5. the query has two or more branches, where the size of the longest component is of size 3, e.g., $ab^*c + d$.

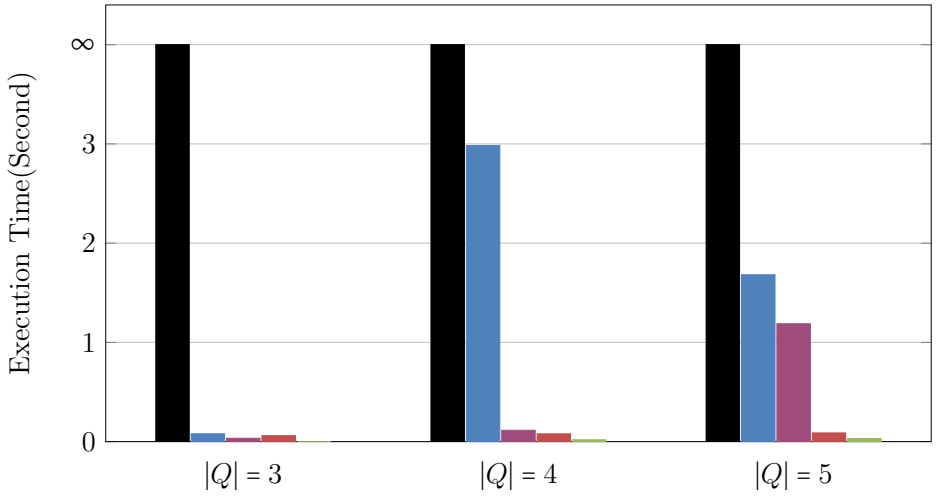


Figure 23: Execution times of different types of queries

Type of queries	$\|Q\| = 3$	$\|Q\| = 4$	$\|Q\| = 5$
Loop that has multiple edge labels	$(a+b)^*c$	$a(b+c)^*d$	$(a+b)^*c(d+e)^*$
Loop that has one edge label	ab^*c	ab^*cd^*	ab^*cd^*e
Loop that has two consecutive edge labels	$(ab)^*c$	$a(bc)^*d$	$(ab)^*c(de)^*$
Query that has multiple branches, and the longest branch size is 2	$a^*b + c$	$ab^* + cd^*$	$ab^* + cd^* + e$
Query that has multiple branches, and the longest branch size is 3	none	$ab^*c + d$	$ab^*c + de^*$

Table 3: Examples of different types of queries

As shown in Figure 23, the first type of query always leads to indefinite execution, which was expected due to the exponential results of generating product graphs. With the presence of loops in the queries, those that contain loops with only a single edge

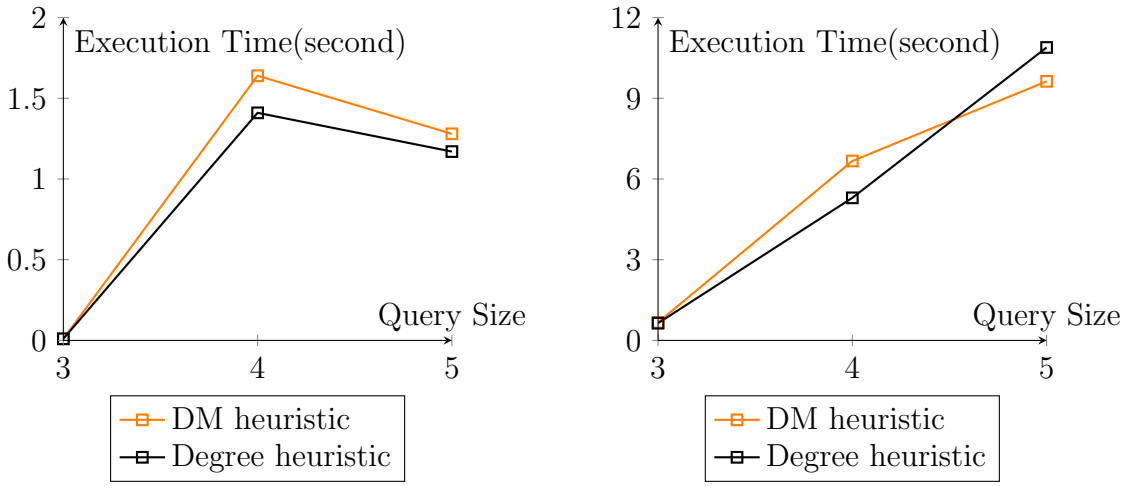
label needed a longer execution time than the loops that have more edge labels. For query size $|Q| = 4$, the former execution time measured was up to 30 times more than the later one. Besides, the second type of query that only contains a loop with a single edge label had the longest execution times compared to other types of queries, in general. The queries with more branches have a better execution time compared to linear queries. A significant difference can be seen in the graph for $|Q| = 4$ and $|Q| = 5$.

5.2.3 Comparing Different State Elimination Heuristics

Since we use the same graph database, Yeast, as in (Ramusat et al., 2021), it is natural to compare their experiment results with ours. However, the problem studied in (Ramusat et al., 2021) was the reachability problem and they directly used the graph database and performed state elimination to get the provenance of paths between two random nodes. This might involve unnecessary state removal because not every node is intermediate and relevant to the input pair of nodes. Therefore, here we compare the two heuristics of state removal sequence, which are the state-weight algorithm (DM algorithm) and the degree-based algorithm mentioned in (Ramusat et al., 2021), using the Retweet graph and a synthetic graph data with size $|V| = 1,000$ and $|E| = 10,000$.

From Figure 24, we can see that the execution times of the DM and the degree-based algorithms are very similar, and the degree-based algorithm is slightly faster than the DM algorithm when $|Q| = 5$ for the synthetic graph database(1.17s and 1.28s,

respectively), and when $|Q| = 4$ for both graph databases, where the execution time for the degree-based algorithm is faster by 0.23s and 1.37s than the DM algorithm. However, the DM algorithm generates shorter size regular expressions, which in turn results in reduced memory cost for saving provenance information.



(a) $|V| = 1,000$, $|E| = 10,000$

(b) $|V| = 256,491$, $|E| = 328,132$

Graph Size	Query Size	Execution time(s)		Size of regular expression	
		DM	Degree	DM	Degree
$\|V\| = 1,000$, $\|E\| = 10,000$	3	0.01	0.01	273	304
	4	1.64	1.41	939	1,339
	5	1.28	1.17	930	1,274
$\|V\| = 256,491$, $\|E\| = 328,132$	3	0.67	0.64	106	106
	4	6.67	5.30	184	193
	5	9.63	10.89	195	235

Figure 24: Experiment results of state removal heuristics

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The provenance of querying relational databases has been studied for the past few decades. An interesting scenario would be to annotate the graph databases with provenance, these annotations will be propagated to the results of queries over graph databases, which can bring us meaningful information about how, why, where the result has been computed, or security clearance, etc.

In this thesis, we provide a detailed set of definitions of provenance for querying graph databases, including a refined version of the single-source provenance and homomorphisms to various semirings. We then present a state-elimination-based algorithm to compute the single-source multidimensional provenance of the RPQ over graph databases and test it with both real-world and synthetic graph databases. This new algorithm works well for certain types of queries, such as the ones that

have multiple branches, or the ones that have consecutive edge labels in one loop. We also compare the performance of two heuristics for state removal sequence, where the results show that they have a similar execution time with the small size of graph databases while the DM algorithm generates a shorter regular expression, which saves more memory in practical scenarios.

6.2 Future Work

Our future work includes exploring the provenance of querying graph databases with more complex queries, such as C2RPQ. We already explored with a limited set of queries in our experiments, however, there is unknown that needs to be further researched, for example, additional applicable sets of queries.

On the other hand, the structure of input graph databases might also affect the performance of querying results. In (Ramusat et al., 2021), Ramusat et al. observed that lower treewidth of the graph leads to a better performance of querying graph databases in practice. Therefore it is beneficial to identify how other graph structures impact the efficiency of our algorithm, which will shed some light on finding the types of real-world graph databases to which our algorithm is suitable.

Also, we would like to explore further how provenance can be used for incremental maintenance. It has already been proven that maintenance of delete operations in databases can benefit from provenance polynomials (Green, Karvounarakis, Ives, & Tannen, 2007), however how provenance can simplify the insertion operations has

still not been evaluated, so it will be an interesting topic.

References

Añez, J., De La Barra, T., & Pérez, B. (1996). Dual graph representation of transport networks. *Transportation Research Part B: Methodological*, *30*(3), 209-216. Retrieved from https://www.sciencedirect.com/science/article/pii/0191261595000240 doi: https://doi.org/10.1016/0191-2615(95)00024-0

Arenas, M., & Pérez, J. (2011). Querying semantic web data with SPARQL. In *Proceedings of the thirtieth acm sigmod-sigact-sigart symposium on principles of database systems* (p. 305–316). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1989284.1989312 doi: 10.1145/1989284.1989312

Barceló Baeza, P. (2013). Querying graph databases. In *Proceedings of the 32nd acm sigmod-sigact-sigai symposium on principles of database systems* (p. 175–188). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2463664.2465216 doi: 10.1145/2463664.2465216

Brzozowski, J. A., & McCluskey, E. J. (1963). Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. Electron. Comput.*, *12*(2), 67–76. Retrieved from https://doi.org/10.1109/PGEC.1963.263416 doi: 10.1109/PGEC.1963.263416

Buneman, P., Khanna, S., & Tan, W. C. (2001). Why and where: A characterization of data provenance. In *Proceedings of the 8th international conference on database theory* (p. 316–330). Berlin, Heidelberg: Springer-Verlag.

Caron, P., & Ziadi, D. (2000). Characterization of Glushkov automata. *Theoretical Computer Science*, *233*(1), 75-90. Retrieved from https://www.sciencedirect.com/science/article/pii/S030439759700296X doi: https://doi.org/10.1016/S0304-3975(97)00296-X

Cheney, J. (2009). Provenance, XML, and the scientific web. *Programming Language*

Techniques for XML (Plan-X).

Cheney, J., Chiticariu, L., & Tan, W.-c. (2009, 01). Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, *1*, 379-474. doi: 10 .1561/1900000006

Cui, Y., Widom, J., & Wiener, J. L. (2000, June). Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, *25*(2), 179–227. Retrieved from `https://doi.org/10.1145/357775.357777` doi: 10.1145/357775.357777

De Domenico, M., Lima, A., Mougel, P., & Musolesi, M. (2013, Oct). The anatomy of a scientific rumor. *Scientific Reports*, *3*(1). Retrieved from `http://dx.doi .org/10.1038/srep02980` doi: 10.1038/srep02980

Delgado, M., & Morais, J. (2005). Approximation to the smallest regular expression for a given regular language. In M. Domaratzki, A. Okhotin, K. Salomaa, & S. Yu (Eds.), *Implementation and application of automata* (pp. 312–314). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dividino, R., Sizov, S., Staab, S., & Schueler, B. (2009). Querying for provenance, trust, uncertainty and other meta knowledge in RDF. *Journal of Web Semantics*, *7*(3), 204-219. Retrieved from `https://www.sciencedirect.com/ science/article/pii/S1570826809000237` (The Web of Data) doi: https:// doi.org/10.1016/j.websem.2009.07.004

Fabregat, A., Korninger, F., Viteri, G., Sidiropoulos, K., Marin-Garcia, P., Ping, P., ... Hermjakob, H. (2018, 01). Reactome graph database: Efficient access to complex pathway data. *PLOS Computational Biology*, *14*, e1005968. doi: 10.1371/journal.pcbi.1005968

Flouris, G., Fundulaki, I., Pediaditis, P., Theoharis, Y., & Christophides, V. (2009). Coloring RDF triples to capture provenance. In *Lecture notes in computer science* (Vol. 5823, p. 196). doi: 10.1007/978-3-642-04930-9_13

Foster, J. N., Green, T. J., & Tannen, V. (2008). Annotated XML: Queries and provenance. In *Proceedings of the twenty-seventh acm sigmod-sigact-sigart symposium on principles of database systems* (p. 271–280). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/ 1376916.1376954` doi: 10.1145/1376916.1376954

Fuhr, N., & Rölleke, T. (1997, January). A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, *15*(1), 32–66. Retrieved from https://doi.org/10.1145/239041.239045 doi: 10.1145/239041.239045

Glushkov, V. M. (1961, oct). The abstract theory of automata. *Russian Mathematical Surveys*, *16*(5), 1–53. Retrieved from https://doi.org/10.1070/rm1961v016n05abeh004112 doi: 10.1070/rm1961v016n05abeh004112

Goldman, R., & Widom, J. (1997). Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd international conference on very large data bases* (p. 436–445). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Green, T. J. (2009). Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th international conference on database theory* (p. 296–309). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1514894.1514930 doi: 10.1145/1514894.1514930

Green, T. J., Karvounarakis, G., Ives, Z. G., & Tannen, V. (2007). Update exchange with mappings and provenance. In *Proceedings of the 33rd international conference on very large data bases* (p. 675–686). VLDB Endowment.

Green, T. J., Karvounarakis, G., & Tannen, V. (2007). Provenance semirings. In *Proceedings of the twenty-sixth acm sigmod-sigact-sigart symposium on principles of database systems* (p. 31–40). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1265530.1265535 doi: 10.1145/1265530.1265535

Gruber, H., Holzer, M., & Tautschnig, M. (2009). Short regular expressions from finite automata: Empirical results. In *Proceedings of the 14th international conference on implementation and application of automata* (p. 188–197). Berlin, Heidelberg: Springer-Verlag. Retrieved from https://doi.org/10.1007/978-3-642-02979-0_22 doi: 10.1007/978-3-642-02979-0_22

Han, Y.-S., & Wood, D. (2007). Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science*, *370*(1), 110-120. Retrieved from https://www.sciencedirect.com/science/article/pii/S0304397506007523 doi: https://doi.org/10.1016/j.tcs.2006.09.025

Hangal, S., MacLean, D., Lam, M. S., & Heer, J. (2010). All friends are not equal: Using weights in social graphs to improve search. In *Workshop on social network mining & analysis, acm kdd.* Retrieved from http://vis.stanford.edu/papers/weighted-social-graphs

Hayes, J., & Gutierrez, C. (2004). Bipartite graphs as intermediate model for RDF. In *Proceedings of the 3rd international conference on semantic web conference* (p. 47–61). Berlin, Heidelberg: Springer-Verlag. Retrieved from https://doi.org/10.1007/978-3-540-30475-3_5 doi: 10.1007/978-3-540-30475-3_5

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation (3rd edition).* USA: Addison-Wesley Longman Publishing Co., Inc.

Imieliński, T., & Lipski, W. (1984, September). Incomplete information in relational databases. *J. ACM*, *31*(4), 761–791. Retrieved from https://doi.org/10.1145/1634.1886 doi: 10.1145/1634.1886

Kleene, S. C. (2016). Representation of events in nerve nets and finite automata. In C. E. Shannon & J. McCarthy (Eds.), *Automata studies. (am-34), volume 34* (pp. 3–42). Princeton University Press. Retrieved from https://doi.org/10.1515/9781400882618-002 doi: doi:10.1515/9781400882618-002

Kozen, D. C. (1997). *Automata and computability* (1st ed.). Berlin, Heidelberg: Springer-Verlag.

Li Ding, P. P. d. S., Yun Peng, & McGuinness, D. L. (2005, April). *Tracking RDF graph provenance using RDF molecules* (Tech. Rep.). UMBC.

Maniu, S., Senellart, P., & Jog, S. (2019, March). An experimental study of the treewidth of real-world graph data. In *ICDT 2019 – 22nd International Conference on Database Theory* (p. 18). Lisbon, Portugal. Retrieved from https://hal.inria.fr/hal-02087763 doi: 10.4230/LIPIcs.ICDT.2019.12

McNaughton, R., & Yamada, H. (1960). Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, *EC-9*(1), 39-47. doi: 10.1109/TEC.1960.5221603

Mohri, M. (2002, January). Semiring frameworks and algorithms for shortest-distance problems. *J. Autom. Lang. Comb.*, *7*(3), 321–350.

Moreira, N., Nabais, D., & Reis, R. (2010). State elimination ordering strategies: Some experimental results. In I. McQuillan & G. Pighizzini (Eds.), *Proceedings*

twelfth annual workshop on descriptional complexity of formal systems, DCFS 2010, saskatoon, canada, 8-10th august 2010 (Vol. 31, pp. 139–148). Retrieved from https://doi.org/10.4204/EPTCS.31.16 doi: 10.4204/EPTCS.31.16

Ramusat, Y., Maniu, S., & Senellart, P. (2018, July). Semiring provenance over graph databases. In *10th USENIX workshop on the theory and practice of provenance (tapp 2018).* London: USENIX Association. Retrieved from https://www.usenix.org/conference/tapp2018/presentation/ramusat

Ramusat, Y., Maniu, S., & Senellart, P. (2021, March). Provenance-based algorithms for rich queries over graph databases. In *EDBT 2021 - 24th International Conference on Extending Database Technology.* Nicosia / Virtual, Cyprus. Retrieved from https://hal.inria.fr/hal-03140067

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th annual southeast regional conference.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1900008.1900067 doi: 10.1145/1900008.1900067

Zimányi, E. (1997). Query evaluation in probabilistic relational databases. *Theoretical Computer Science*, *171*(1), 179-219. Retrieved from https://www.sciencedirect.com/science/article/pii/S0304397596001296 doi: https://doi.org/10.1016/S0304-3975(96)00129-6