



PONTIFICIA UNIVERSIDAD CATÓLICA DE  
CHILE

MASTER'S THESIS

---

# Expressiveness and Complexity of Query Languages for Graph Databases

---

*Autor:*  
Esteban Vásquez

*Advisor:*  
Domagoj Vrigoč

A thesis submitted in fulfillment of the requirements for the degree of Master  
of Mathematics in the Faculty of Mathematics of Pontificia Universidad  
Católica de Chile.

December 12, 2025

# Abstract

Graph databases have become a standard for modeling complex relationships, with Regular Path Queries (RPQs) serving as the fundamental mechanism for navigation and pattern matching. In this work, we investigate an extension of RPQs capable of capturing specific edges along a path using list variables, modeled formally through *automata with list variables*. We provide a comprehensive study of the theoretical properties of this model, with a particular focus on determinism and computational complexity.

Regarding determinism, we introduce and compare three distinct notions: standard determinism, I/O unambiguity, and determinism\*. We demonstrate that, unlike classical automata, automata with list variables cannot always be determinized or converted into an I/O unambiguous form, although an equivalent deterministic\* automaton can always be constructed via a Powerset-like construction.

Furthermore, we analyze the decision problems associated with the output of these automata. We establish that while the *Emptiness* and *Path Membership* problems remain in the complexity class **NL-complete**, problems involving the verification of specific output mappings exhibit higher complexity. Specifically, we prove that the *Path Mapping Membership* and *Mapping Membership* problems are **NL-complete** for single-variable mappings, and where the path is a trail, but become **NP-complete** in the general case with multiple variables. Finally, we present algorithmic solutions for retrieving shortest paths, trails, simple paths, and acyclic paths with their mappings based on the product graph construction, for practical application.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Graph databases and Paths. . . . .	3
2.2	RPQ . . . . .	4
2.3	RPQs with list variables . . . . .	8
<b>3</b>	<b>Automaton with list variables</b>	<b>10</b>
<b>4</b>	<b>Determinism</b>	<b>16</b>
4.1	Classic Determinism . . . . .	16
4.2	I/O Unambiguous . . . . .	18
4.3	Determinism* . . . . .	20
<b>5</b>	<b>Decidability Problems</b>	<b>23</b>
5.1	The Emptiness Problem . . . . .	23
5.2	The Path Membership Problem . . . . .	26
5.3	The Path Mapping Membership Problem . . . . .	29
5.4	The Mapping Membership Problem . . . . .	37
<b>6</b>	<b>Algorithms</b>	<b>43</b>
6.1	Any Shortest . . . . .	45
6.2	All Shortest Paths . . . . .	48
6.3	TRAIL, SIMPLE and ACYCLIC . . . . .	54
<b>7</b>	<b>Conclusions</b>	<b>56</b>

# 1 Introduction

Graph databases have emerged for modelling complex, highly interconnected data in domains ranging from social networks to biological systems. The most common way to navigate these structures are Regular Path Queries (RPQs) [2, 1], which allow users to discover pairs of nodes connected by paths labelled with sequences satisfying a regular expression. Indeed, RPQs form the backbone of modern graph query standards such as GQL and SQL/PGQ [3]. However, classical RPQs suffer from a significant limitation: they are typically existential, returning only the source and target nodes while discarding the path itself and the specific data associated with the edges traversed.

In many practical scenarios, obtaining the specific sequence of edges, or capturing specific data edges along the path, is as important as reachability itself. To address this, we investigate an extension of RPQs that incorporates *list variables*. This mechanism allows the query to bind a sequence of edges to variables, during traversal, enabling data extraction and pattern matching capabilities beyond simple reachability. The semantics of RPQs with list variables was first formally defined in [3, 5], but was not studied in detail in the research literature.

In this thesis, we formally model this extension using *automata with list variables* (Section 3). Our primary contribution is a rigorous theoretical analysis of this model, with a specific emphasis on its structural properties and computational limits.

First, we address the fundamental question of **determinism** (Section 4). We explore different notions of determinism—including I/O Unambiguity and a simplified projection-based determinism—proving that full determinization is not always possible.

Second, we analyze the **decidability and complexity** of evaluating these automata (Section 5). We define and solve the *Emptiness*, *Path Membership*, and *Mapping Membership* problems. A key finding of our work is the complexity boundary governed by the number of variables: while problems involving single-variable mappings remain tractable within **NL-complete**, the general case with multiple variables becomes **NP-complete**.

Finally, to bridge the gap between theory and practice, we provide algorithms (Section 6) based on product graph constructions to retrieve shortest paths and their mappings. This work provides the necessary theoretical foundations for implementing efficient path extraction in graph database systems.

## 2 Preliminaries

In this section, we formally define graph databases, describe the structure of paths, and introduce their main classifications and related definitions, following the GQL and SQL/PGQ standards. Furthermore, we will define the concept of Regular Path Queries (RPQ), which forms the basis for graph pattern matching. This definition will be extended to a type of RPQ capable of capturing edges through list variables.

### 2.1 Graph databases and Paths.

Let **Nodes** be a set of node identifiers, and **Edges** be a set of edge identifiers, with **Nodes** and **Edges** being disjoint. Additionally, let **Lab** be a set of labels. We define graph databases, paths and their fundamental components using conventional definitions established in the field [2].

**Definition 1.** A *graph database*  $G$  is a tuple  $(N, E, \rho, \lambda)$ , where

- $N \subseteq \mathbf{Nodes}$  is a finite set of nodes,
- $E \subseteq \mathbf{Edges}$  is a finite set of edges,
- $\rho : E \rightarrow N \times N$  is a total function, and
- $\lambda : E \rightarrow \mathbf{Lab}$  is a total function assigning a label to each edge.

Intuitively,  $\rho(e) = (v_1, v_2)$  means that  $e$  is a directed edge going from  $v_1$  to  $v_2$ , and  $e : a$  denotes  $\lambda(e) = a$ .

**Definition 2.** In a graph database  $G = (N, E, \rho, \lambda)$ , the sequence

$$p = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$$

is called a **path** if  $n \geq 0$ ,  $e_i \in E$ , and  $\rho(e_i) = (v_{i-1}, v_i)$  for  $1 \leq i \leq n$ .

If  $p$  is a path in  $G$ , we write  $\mathbf{lab}(p) = \lambda(e_1) \cdots \lambda(e_n)$  for the sequence of labels that occur on the edges of  $p$ . We write  $\mathbf{src}(p)$  for the starting node  $v_0$  of  $p$ , and  $\mathbf{tgt}(p)$  for the end node  $v_n$  of  $p$ . The length of a path  $p$ , denoted  $\mathbf{len}(p)$ , is defined as the number  $n$  of edges that it uses.

Next, we distinguish between the following types of paths.

**Definition 3.** We say that a path  $p$  is a *WALK*, for every  $p$ ; is a *TRAIL*, if  $p$  does not repeat an edge, that is,  $(e_i \neq e_j \text{ for every } i \neq j)$ ; is *SIMPLE*, if  $p$  does not repeat a node, except that possibly  $\text{src}(p) = \text{tgt}(p)$  and is *ACYCLIC*, if  $p$  does not repeat a any node  $(v_i \neq v_j \text{ for every } i \neq j)$ .

Additionally, given a set of paths  $P$  over a graph database  $G$ , we say that  $p \in P$  is a *SHORTEST* path in  $P$  if  $\text{len}(p) \leq \text{len}(p')$  for each  $p' \in P$ . We use  $\text{Paths}(G)$  to denote the (possibly infinite) set of all paths in a graph database  $G$ .

Finally, we define the operation between paths called concatenation.

**Definition 4.** Let  $p_1 = v_0 e_1 v_1 \cdots e_k v_k$  and  $p_2 = u_0 f_1 u_1 \cdots f_l u_l$  be two paths, not necessarily distinct, in some graph  $G$ ; such that  $\text{tgt}(p_1) = v_k = u_0 = \text{src}(p_2)$ . We define their *concatenation* as

$$p_1 \cdot p_2 = v_0 e_1 v_1 \cdots e_k v_k f_1 u_1 \cdots f_l u_l$$

From now on, whenever we define a concatenation between  $p_1$  and  $p_2$ , it is assumed that  $\text{tgt}(p_1) = \text{src}(p_2)$ .

## 2.2 RPQ

To ensure consistency with emerging graph query language standards, such as GQL [3], we define the basic building blocks for graph path queries, the regular path queries, RPQs [1], based on regular expressions.

**Definition 5.** A *path query* is expressed using the following syntactic structure:

$$(x, \text{regex}, y)$$

where  $x$  and  $y$  are node variables. And *regex* is the regular expression that specifies the allowed sequence of edge labels on the path.

The core expressive power of the RPQ is based on the component *regex*. We now provide a formal, inductive definition for these expressions over the graph's labelling alphabet.

**Definition 6.** Let  $\text{Lab}$  be the finite set of edge labels in the graph. *Regular expressions, regex*, are inductively defined over this set:

1. Every label  $a \in \text{Lab}$  is a regular expression.

2. The empty string  $\varepsilon$  is a regular expression.
3. If  $r_1$  and  $r_2$  are regular expressions, then the following operations are also regular expressions
  - Union:  $r_1 + r_2$ .
  - Concatenation:  $r_1 \cdot r_2$ .
  - Kleene Closure:  $r_1^*$ .

By convention, the dot notation for concatenation ( $\cdot$ ) is often omitted, and parentheses are simplified where possible. Additionally, the following common abbreviations are used to simplify expression writing:

- $r^*$  is the abbreviation for  $(r + \varepsilon)$ , which denotes zero or more occurrence of  $r$ .
- $r^?$  is the abbreviation for  $(r + \varepsilon)$ , which denotes zero or one occurrence of  $r$ .
- $r^+$  is the abbreviation for  $(r \cdot r^*)$ , which denotes one or more occurrences of  $r$ .

Finally, for each regular expression, we are interested in capturing all paths whose sequence of edge labels is allowed by the regular expression, independently of the source and target nodes. We will call this set the output of the regular expression  $\text{regex}$  over the graph  $G$ , and we define it inductively as follows.

**Definition 7.** The **output**  $\llbracket r \rrbracket_G$  generated by a regular expression  $r, r_1, r_2$  is inductively defined. For each label  $a \in \text{Lab}$  and  $\varepsilon$  the empty string

- $\llbracket \varepsilon \rrbracket_G = \{p \mid p = v, v \in N\}$
- $\llbracket a \rrbracket_G = \{p \mid p = v_1 e v_2 \text{ and } v_1, v_2 \in N, e \in E, \lambda(e) = a\}$
- The output of the union is  $\llbracket r_1 + r_2 \rrbracket_G = \llbracket r_1 \rrbracket_G \cup \llbracket r_2 \rrbracket_G$
- The output of the concatenation is

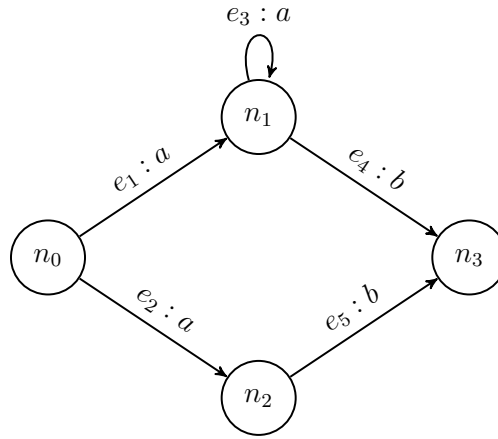
$$\llbracket r_1 \cdot r_2 \rrbracket_G = \{p \mid p = p_1 \cdot p_2, p_1 \in \llbracket r_1 \rrbracket_G, p_2 \in \llbracket r_2 \rrbracket_G\}$$

- The output of the Kleene Clousure is

$$\llbracket r^* \rrbracket_G = \{p \mid p = p_1 \cdots p_k, p_i \in \llbracket r \rrbracket_G\}$$

Lets analyze the following example

**Example 1.** Let  $G$  be the following graph database



For the regular expression  $r = a^*b$ , its output  $\llbracket r \rrbracket_G$  is infinite since, using edge  $e_3$ , paths of arbitrary length can be constructed whose labels are permitted by the expression  $r$ .

Note that if the query is restricted to, for example, a shortest path or a trail, the output is immediately finite.

To avoid situations like the example presented above, restrictors and selectors are defined for regular expressions.

**Definition 8.** The grammar for *restrictors* in the RPQ is as follows:

$$\text{restrictor} = \text{TRAIL} \mid \text{SIMPLE} \mid \text{ACYCLIC}$$

And the output of queries with any restrictor and regex any regular expression, is

$$\llbracket \text{restrictor}(x, \text{regex}, y) \rrbracket_G = \llbracket \text{regex} \rrbracket_G \cap \text{Paths}(G, \text{restrictor})$$

Here, the notation  $\text{Paths}(G, \text{restrictor})$  denotes the set of all paths in  $G$  that are valid according to restrictor.

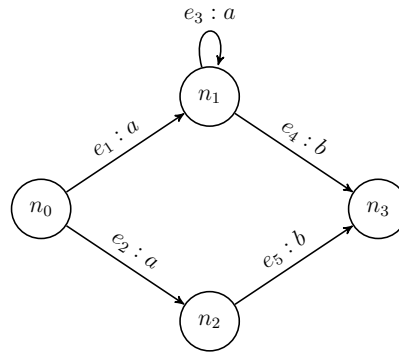
**Definition 9.** The grammar for *selectors* in the RPQ is as follows:

$$\text{selector} = \text{ANY SHORTEST} \mid \text{ALL SHORTEST}$$

And the output of queries with any selector are defined case by case. For *regex* any regular expression:

- $\llbracket \text{ANY SHORTEST } (x, \text{regex}, y) \rrbracket_G$  returns, for each pair of nodes  $(n, n')$  in  $G$ , a single path  $p \in \llbracket \text{regex} \rrbracket_G$  such that  $\text{src}(p) = n, \text{tgt}(p) = n'$  and  $p$  is *SHORTEST* among all paths  $p' \in \llbracket \text{regex} \rrbracket_G$ , chosen non-deterministically.
- $\llbracket \text{ANY SHORTEST restrictor}(x, \text{regex}, y) \rrbracket_G$  returns, for each pair of nodes  $(n, n')$  in  $G$ , a single path  $p \in \llbracket \text{restrictor}(x, \text{regex}, y) \rrbracket_G$  such that  $\text{src}(p) = n, \text{tgt}(p) = n'$  and  $p$  is *SHORTEST* among all paths  $p' \in \llbracket \text{restrictor}(x, \text{regex}, y) \rrbracket_G$ , chosen non-deterministically.
- $\llbracket \text{ALL SHORTEST } (x, \text{regex}, y) \rrbracket_G$  returns, for each pair of nodes  $(n, n')$  in  $G$ , the set of all paths  $p \in \llbracket \text{regex} \rrbracket_G$  such that  $\text{src}(p) = n, \text{tgt}(p) = n'$  and  $p$  is *SHORTEST* among all paths  $p' \in \llbracket \text{regex} \rrbracket_G$ .
- $\llbracket \text{ALL SHORTEST restrictor}(x, \text{regex}, y) \rrbracket_G$  returns, for each pair of nodes  $(n, n')$  in  $G$ , the set of all paths  $p \in \llbracket \text{restrictor}(x, \text{regex}, y) \rrbracket_G$  such that  $\text{src}(p) = n, \text{tgt}(p) = n'$  and  $p$  is *SHORTEST* among all paths  $p' \in \llbracket \text{restrictor}(x, \text{regex}, y) \rrbracket_G$ .

**Example 2.** Let  $G$  be the same graph database in the last example



For the same regular expression  $\text{regex} = a^*b$ , the following outputs are obtained given the next examples using restrictors and selectors

- $\llbracket ALL\ SHORTEST(x, regex, y) \rrbracket_G$  returns for the pair of nodes  $(n_0, n_3)$  the two shortest paths  $p_1 = n_0e_1n_1e_4n_3$  and  $p_2 = n_0e_2n_3e_5n_3$
- $\llbracket ANY\ SHORTEST(x, regex, y) \rrbracket_G$  returns for the pair of nodes  $(n_0, n_3)$  the path  $p_1$  or the path  $p_2$  chosen non-deterministically
- $\llbracket TRAIL(x, regex, y) \rrbracket_G = \{p_1, p_2, p_3\}$  where  $p_3 = n_0e_1n_1e_3n_1e_4n_3$

### 2.3 RPQs with list variables

RPQs can be extended to the case where we are interested in capturing some graph edge through the query; this is achieved by adding list variables to the regular expressions. In this section, we will define that extension.

The main and only difference from an RPQ is based on the component regex. We now provide a formal, inductive definition for these expressions in the new labelling alphabet with variables.

**Definition 10.** *The **regular expressions with list variables** are defined exactly like the previous ones, with only one exception.*

- For every label  $a \in Lab$  and every variable  $z \in V$  the set of variables.  $a^z$  is a regular expression.
- Every label  $a \in Lab$  and the empty string  $\varepsilon$  are regular expressions.
- If  $r_1$  and  $r_2$  are regular expressions, then the union, concatenation, and Kleene Closure are also regular expressions.

The regular expression  $a^z$  is the one that captures an edge with a label  $a$  on the variable  $z$ .

Then, as before, we define the output of these regular expressions over a graph  $G$ , which is the set of pairs  $(p, \mu)$  such that  $p$  is a path whose sequence of edge labels is allowed by the regular expression, and  $\mu$  is the mapping that captures the path edges as indicated by the regular expression.

**Definition 11.** *The **output**  $\llbracket r \rrbracket_G$  generated by a **regular expression with list variables**  $r, r_1, r_2$  is inductively defined. For each label  $a \in Lab$ , each variable  $z \in V$  and  $\varepsilon$  the empty string*

- $\llbracket \varepsilon \rrbracket_G = \{(p, \mu_0) \mid p = v, v \in N\}$

- $\llbracket a \rrbracket_G = \{(p, \mu_0) \mid p = v_1 e v_2 \text{ and } v_1, v_2 \in N, e \in E, \lambda(e) = a\}$
- $\llbracket a^z \rrbracket_G = \{(p, z \mapsto [a]) \mid p = v_1 e v_2 \text{ and } v_1, v_2 \in N, e \in E, \lambda(e) = a\}$
- *The output of the union is  $\llbracket r_1 + r_2 \rrbracket_G = \llbracket r_1 \rrbracket_G \cup \llbracket r_2 \rrbracket_G$*
- *The output of the concatenation is*

$$\llbracket r_1 \cdot r_2 \rrbracket_G = \{(p_1 \cdot p_2, \mu_1 \cdot \mu_2) \mid (p_1, \mu_1) \in \llbracket r_1 \rrbracket_G, (p_2, \mu_2) \in \llbracket r_2 \rrbracket_G\}$$

- *The output of the Kleene Closure is*

$$\llbracket r^* \rrbracket_G = \{(p, \mu) \mid p = p_1 \cdots p_k, \mu = \mu_1 \cdots \mu_k, (p_i, \mu_i) \in \llbracket r \rrbracket_G\}$$

Here  $\mu_0$  is the empty mapping; that is, the mapping such that  $Dom(\mu) = \emptyset$ . The concatenation of two mappings is defined as follows.

**Definition 12.** *Let  $\mu_1, \mu_2$  be two arbitrary mappings. We define their concatenation  $\mu := \mu_1 \cdot \mu_2$  so that*

$$\mu(z) = \begin{cases} \mu_1(z) & \text{if } z \in Dom(\mu_1) \setminus Dom(\mu_2) \\ \mu_2(z) & \text{if } z \in Dom(\mu_2) \setminus Dom(\mu_1) \\ \mu_1(z) \cdot \mu_2(z) & \text{if } z \in Dom(\mu_1) \cap Dom(\mu_2) \end{cases}$$

where  $\mu_1(z) \cdot \mu_2(z)$  denotes the standard concatenation of lists.

For these queries, it is also possible to use restrictors and selectors, as in the previous section. The output definition remains the same, but now considers pairs  $(p, \mu)$  instead of only a path  $p$ . However, selectors and restrictors are applied only to the path.

### 3 Automaton with list variables

Given that a standard construction can be performed for every regular expression *regex* to obtain an equivalent automaton [6], the same is possible for regular expressions containing labels with variables, simply by treating labels  $a^z$  and  $a$  as distinct labels. This construction yields an automaton with list variables, which is the principal object we will study because it offers a simpler, yet equivalent, abstraction for analyzing the semantics of RPQs with list variables.

Next, we define an automaton with list variables, and we also define the output of this type of automaton.

**Definition 13.** *An automaton with list variables is an NFA over an alphabet  $\Sigma^V$ , it is a tuple  $(Q, \Sigma^V, \delta, q_0, F)$ , where:*

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of edge labels .
- $V$  is a finite set of variables.
- $\Sigma^V = \Sigma \cup \{a^z : a \in \Sigma, z \in V\}$ .
- $\delta \subseteq Q \times \Sigma^V \times Q$  the transition relation.
- $q_0 \in Q$  is the initial state.
- $F \subseteq Q$  is the set of final states.

For each  $\sigma \in \Sigma^V$ , the projection of  $\sigma$  in  $\Sigma$  is defined by

$$\pi_{\Sigma}(\sigma) := \begin{cases} \sigma & \text{if } \sigma \in \Sigma, \\ a & \text{if } \sigma = a^z, \text{ for some } a \in \Sigma, z \in V. \end{cases}$$

If  $A = (Q, \Sigma^V, \delta, q_0, F)$  is an automaton with list variables, then a **configuration of  $A$**  is a pair  $(q, \mu)$  such that  $q \in Q$  and  $\mu : V \rightarrow \text{Edges}^*$ .

For a path  $p = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$  on a graph  $G$ , we define a **run  $r$  of  $A$  over  $p$** , which is a sequence of configurations

$$r := (q_0, \mu_0) \xrightarrow{\sigma_1} (q_1, \mu_1) \xrightarrow{\sigma_2} \cdots (q_{n-1}, \mu_{n-1}) \xrightarrow{\sigma_n} (q_n, \mu_n)$$

such that for every index  $1 \leq i \leq n$ :

- $(q_{i-1}, \sigma_i, q_i) \in \delta$ .
- $\lambda(e_i) = \pi_\Sigma(\sigma_i)$ .
- $\mu_0$  is the initial mapping such that  $Dom(\mu_0) = \emptyset$
- if  $\sigma_i = a$  then  $\mu_i = \mu_{i-1}$ .
- if  $\sigma_i = a^z$  then  $\mu_i = \mu_{i-1}$  and  $\mu_i(z) := [\mu_{i-1}(z) \cdot e_i]$ .

A run  $r$  is accepted if and only if  $q_n \in F$ . In that case  $\mu_n$  is denoted by  $\mu_r$ .

**Definition 14.** An *Output of A over p* is the set of all mappings  $\mu$  such that there is an accepting run; this is

$$\llbracket A \rrbracket_p := \{\mu_r \mid r \text{ is an accepting run of } A \text{ over } p\}.$$

Using this, we define the *Output of A over a graph database G* as follows:

$$\llbracket A \rrbracket_G := \{(p, \mu) \mid \mu \in \llbracket A \rrbracket_p, p \text{ is a path of } G\}.$$

**Proposition 1.** For any RPQ with list variables with regular expression *regex*, there exists an equivalent automaton with list variables  $A$ , such that their outputs are equal; that is,  $\llbracket regex \rrbracket_G = \llbracket A \rrbracket_G$ .

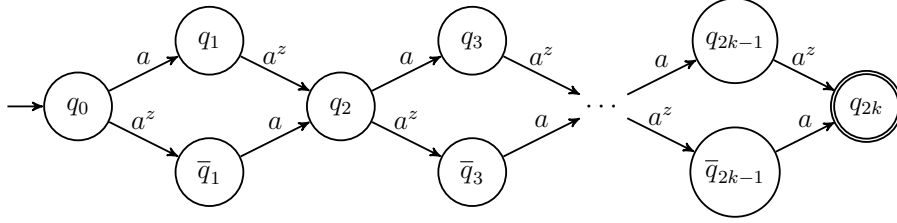
*Proof.* Given the regular expression *regex* with list variables, the standard construction from a regular expression to an automaton is performed [6]. In this specific case, we simply treat  $a$  and  $a^z$  as distinct labels for any label  $a$  and variable  $z$ . with this, we obtain an automaton with list variables that accepts only the paths allowed by the regular expression.  $\square$

Now we analyze an illustrative example that is relevant to the preceding discussion. Let us consider the following automaton for all the examples we will study,  $A_k = (Q, \Sigma^V, \delta, q_0, F)$  where:

- $Q = \{q_{2i} : 0 \leq i \leq k\} \cup \{q_{2i+1}, \bar{q}_{2i+1} : 0 \leq i \leq k-1\}$ .
- $\Sigma^V = \{a, a^z\}$ .
- For  $0 \leq i \leq k-1$ ,

$$\delta = \{(q_{2i}, a, q_{2i+1}), (q_{2i}, a^z, \bar{q}_{2i+1}), (q_{2i+1}, a^z, q_{2i+2}), (\bar{q}_{2i+1}, a, q_{2i+2})\}$$

This automaton has the following structure:

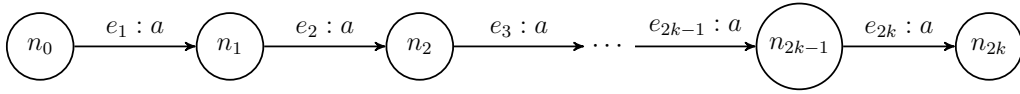


Note that for this particular automaton  $A_k$ , there are  $2^k$  possible runs.

**Example 3.** For the first example, let us define the graph database  $G_1 = (N, E, \rho, \lambda)$  such that:

- $N = \{n_0, \dots, n_{2k}\}$
- $E = \{e_1, \dots, e_{2k}\}$
- $\rho(e_i) = (n_{i-1}, n_i)$
- $\lambda(e_i) = a$

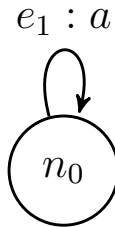
This graph database is represented as follows:



Now for the path  $p = n_0 e_1 n_1 e_2 \dots n_{2k-1} e_{2k} n_{2k}$ . How many pairs  $(r, \mu)$  exist such that  $(p, \mu) \in \llbracket A \rrbracket_G$  and  $\mu = \mu_r \in \llbracket A \rrbracket_p$ ?

In the case of the graph  $G = G_1$  the answer is clear, each run determines a unique mapping. Therefore, there are  $2^k$  distinct pairs  $(r, \mu)$  for the path  $p$ . And there is a one-to-one correspondence between mappings and runs.

**Example 4.** For the second example consider the case with the graph database  $G_2$



defined as follows  $G_2 = (N, E, \rho, \lambda)$  where:

- $N = \{n_0\}$
- $E = \{e_1\}$
- $\rho(e_1) = (n_0, n_0)$
- $\lambda(e_1) = a$

For the path  $p = n_0 e_1 \cdots n_0 e_1 n_0$  where  $\text{len}(p) = 2k$  there is only one mapping  $\mu_0 = [z \mapsto e_1, \dots, e_1]$  for all possible runs. Therefore, there are  $2^k$  pairs  $(r, \mu_0)$ , and the mapping is the same for all runs.

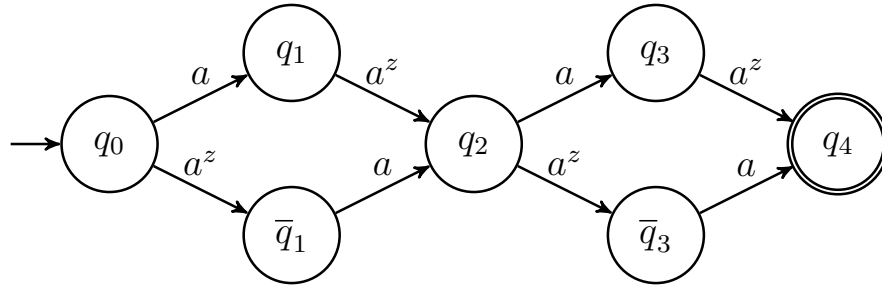
**Example 5.** For the final example, let us consider the following automaton, a simplified version of the one analyzed previously:  $A = (Q, \Sigma^V, \delta, q_0, F)$  where:

- $Q = \{q_0, q_1, \bar{q}_1, q_2, q_3, \bar{q}_3, q_4\}$ .
- $\Sigma^V = \{a, a^z\}$ .
- For  $0 \leq i \leq 1$ ,

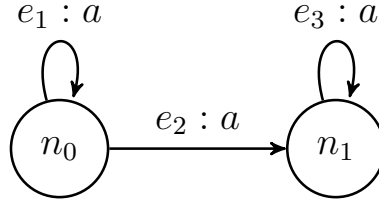
$$\delta = \{(q_{2i}, a, q_{2i+1}), (q_{2i}, a^z, \bar{q}_{2i+1}), (q_{2i+1}, a^z, q_{2i+2}), (\bar{q}_{2i+1}, a, q_{2i+2})\}$$

- $q_0 = q_0$ .
- $F = \{q_4\}$ .

The automaton looks like this:



And let  $G$  be the graph database



where  $G_3 = (N, E, \rho, \lambda)$  and:

- $N = \{n_0, n_1\}$
- $E = \{e_1, e_2, e_3\}$
- $\rho(e_1) = (n_0, n_0), \rho(e_2) = (n_0, n_1), \rho(e_3) = (n_1, n_1)$
- $\lambda(e_1) = a, \lambda(e_2) = a, \lambda(e_3) = a$

Let us list all possible paths in this graph from  $n_0$  to  $n_1$  that have length 4:

1.  $p_1 = e_2 e_3 e_3 e_3$
2.  $p_2 = e_1 e_2 e_3 e_3$
3.  $p_3 = e_1 e_1 e_2 e_3$
4.  $p_4 = e_1 e_1 e_1 e_2$

Let us also list all possible accepting runs of the automaton  $A$

1.  $r_1 = q_0 \xrightarrow{a^z} q_1 \xrightarrow{a} q_2 \xrightarrow{a^z} q_3 \xrightarrow{a} q_4$
2.  $r_2 = q_0 \xrightarrow{a^z} q_1 \xrightarrow{a} q_2 \xrightarrow{a} \bar{q}_3 \xrightarrow{a^z} q_4$
3.  $r_3 = q_0 \xrightarrow{a} \bar{q}_1 \xrightarrow{a^z} q_2 \xrightarrow{a^z} q_3 \xrightarrow{a} q_4$
4.  $r_4 = q_0 \xrightarrow{a} \bar{q}_1 \xrightarrow{a^z} q_2 \xrightarrow{a} \bar{q}_3 \xrightarrow{a^z} q_4$

The possible mappings for each path are as follows, derived from the unique accepting runs.

1. For path  $p_1$  there are  $\mu_1 = [z \mapsto e_2, e_3]$  for the runs  $r_1, r_2$  and  $\mu_2 = [z \mapsto e_3, e_3]$  for the runs  $r_3, r_4$ .

2. For path  $p_2$  there are  $\mu_1 = [z \mapsto e_1, e_3]$  for the runs  $r_1, r_2$  and  $\mu_2 = [z \mapsto e_2, e_3]$  for the runs  $r_3, r_4$ .
3. For path  $p_3$  there are  $\mu_1 = [z \mapsto e_1, e_2]$  for the runs  $r_1, r_3$  and  $\mu_2 = [z \mapsto e_1, e_3]$  for the runs  $r_2, r_4$ .
4. For path  $p_4$  there are  $\mu_1 = [z \mapsto e_1, e_1]$  for the runs  $r_1, r_3$  and  $\mu_2 = [z \mapsto e_1, e_2]$  for the runs  $r_2, r_4$ .

For each mapping, there exist exactly two runs that produce it. Let us observe the following particular features in this example.

For the mapping  $\mu = [z \mapsto e_1, e_3]$ :

- $(p_2, \mu) \in \llbracket A \rrbracket_G$  and  $\mu = \mu_r \in \llbracket A \rrbracket_{p_2}$  for the runs  $r = r_1$  and  $r = r_2$ .
- $(p_3, \mu) \in \llbracket A \rrbracket_G$  and  $\mu = \mu_r \in \llbracket A \rrbracket_{p_3}$  for the runs  $r = r_2$  and  $r = r_4$ .

A similar situation occurs for the mappings  $[z \mapsto e_2, e_3]$  and  $[z \mapsto e_1, e_2]$ . However, for this particular case  $\mu$ , something special occurs, for the pair  $(\mu, r_2)$  there are 2 paths  $p = p_2$  or  $p = p_3$  such that  $(p, \mu) \in \llbracket A \rrbracket_G$  and  $\mu = \mu_r \in \llbracket A \rrbracket_p$ .

We conclude from the examples that, given an automaton  $A$ , a graph  $G$ , and a mapping  $\mu$ , there may exist more than one distinct path such that  $(p, \mu) \in \llbracket A \rrbracket_G$ . Moreover, there may exist more than one possible run for each path.

Another interesting conclusion is that, given a mapping  $\mu$ , and a run  $r$ , there may exist more than one path such that  $(p, \mu) \in \llbracket A \rrbracket_G$  and  $\mu_r = \mu$ .

## 4 Determinism

In this section, we will study different notions of determinism that may exist for automata with list variables.

### 4.1 Classic Determinism

For this purpose, we first define the projection of an automaton, as a simplification of the automaton with list variables to a normal automaton.

**Definition 15.** *A projection of an automaton with list variables  $A$  is the tuple*

$$\pi(A) := (Q, \Sigma, \delta^*, q_0, F)$$

where  $A = (Q, \Sigma^V, \delta, q_0, F)$ , and if  $(q, a, q') \in \delta$  or  $(q, a^z, q') \in \delta$ , then  $(q, a, q') \in \delta^*$ .

With this definition, we define the strongest notion of determinism for automata with list variables.

**Definition 16.** *We call an automaton with list variables  $A$  deterministic if and only if its projection  $\pi(A)$  is deterministic.*

A deterministic automaton has the following property.

**Proposition 2.** *Let  $A$  be a deterministic automaton with list variables and  $G$  be a graph database. Then for every path  $p \in G$  there is at most one accepting run of  $A$  over  $p$ .*

*Proof.* Let  $A$  be a deterministic automaton with a list of variables,  $G$  any graph, and  $p = v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$  a path in  $G$ . Then, there is only one run of  $A$  over  $p$  given by:

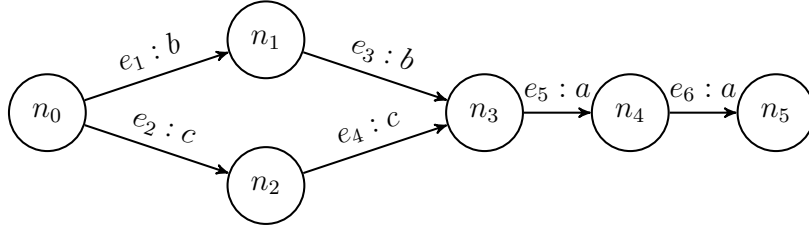
$$r := (q_0, \mu_0) \xrightarrow{o_1} (q_1, \mu_1) \xrightarrow{o_2} \cdots (q_{n-1}, \mu_{n-1}) \xrightarrow{o_n} (q_n, \mu_n)$$

where  $\mu_0$  is the initial mapping with  $Dom(\mu_0) = \emptyset$  and  $\pi_\Sigma(o_i) = \lambda(e_i)$ . Then  $o_i$  and  $\mu_i$  are uniquely determined by  $A$ , since  $\pi(A)$  is deterministic. If this unique run is an accepting run of  $A$  over  $p$ , then  $\mu_n = \mu_r$  and  $(p, \mu_r) \in \llbracket A \rrbracket_G$ .  $\square$

Let us now consider the case where we know  $\mu$  instead of the path  $p$ . Is there a unique path  $p$  associated with  $\mu$ ? The answer is no, as shown by the following example.

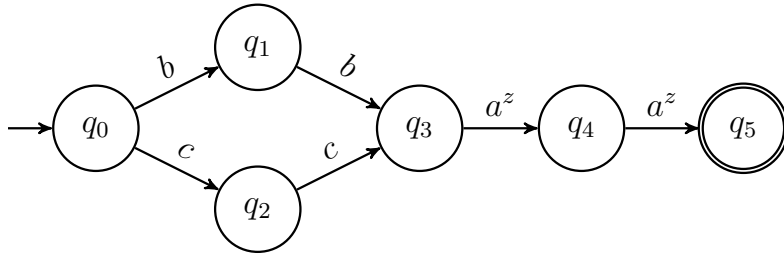
**Example 6.** Let  $G$  be the graph  $(N, E, \rho, \lambda)$  where:

- $N = \{n_0, n_1, n_2, n_3, n_4, n_5\}$
- $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
- $\rho(e_1) = (n_0, n_1); \rho(e_2) = (n_0, n_2); \rho(e_3) = (n_1, n_3); \rho(e_4) = (n_2, n_3); \rho(e_5) = (n_3, n_4); \rho(e_6) = (n_4, n_5)$
- $\lambda(e_1) = b; \lambda(e_2) = c; \lambda(e_3) = b; \lambda(e_4) = c; \lambda(e_5) = a; \lambda(e_6) = a$



Let  $A = (Q, \Sigma^V, \delta, q_0, F)$  with:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma^V = \{a, b, c, a^z\}$
- $\delta = \{(q_0, b, q_1), (q_0, c, q_2), (q_1, b, q_3), (q_2, c, q_3), (q_3, a^z, q_4), (q_4, a^z, q_5)\}$
- $F = \{q_5\}$



This automaton is deterministic; however, given the mapping  $\mu : [z \mapsto e_5, e_6]$ , there exist two paths in  $G$ ,  $p_1 = n_0e_1n_1e_3n_3e_5n_4e_6n_5$  and  $p_2 = n_0e_2n_2e_4n_3e_5n_4e_6n_5$  such that  $(p_1, \mu) \in \llbracket A \rrbracket_G$  and  $(p_2, \mu) \in \llbracket A \rrbracket_G$ .

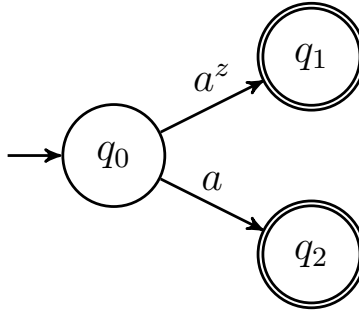
## 4.2 I/O Unambiguous

**Definition 17.** We say that an Automaton with list variables  $A$  is **I/O Unambiguous** if and only if for every path  $p$  and every output  $\mu$  such that  $\mu \in \llbracket A \rrbracket_p$  there exists exactly one accepting run  $r$  with  $\mu_r = \mu$

**Proposition 3.** If  $A$  is a deterministic automaton, then  $A$  is also I/O Unambiguous.

*Proof.* This proposition is a direct consequence of the result in proposition 1. Because determinism implies that for each path  $p$ , there is at most one accepting run.  $\square$

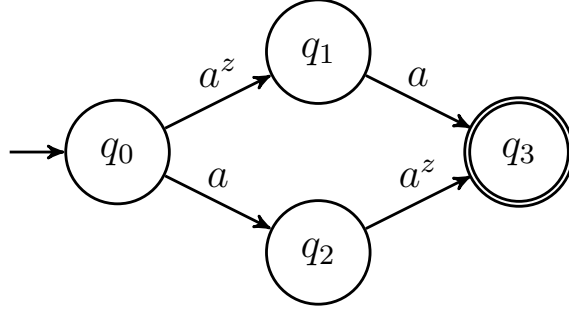
The converse result of the proposition is false; the following automaton is I/O Unambiguous, but not deterministic.



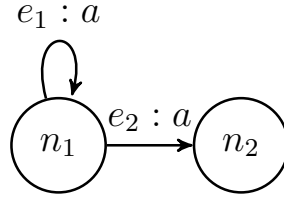
The next natural question that arises is, given an arbitrary automaton, is it always possible to find an equivalent I/O Unambiguous? If the answer were affirmative, this would yield an important result. However, as stated in the following proposition, this is not the case.

**Proposition 4.** There exists an automaton with list variables for which there is no equivalent I/O Unambiguous automaton with list variables.

*Proof.* Let  $A = (\{q_0, q_1, q_2, q_3\}, \{a, a^z\}, \delta, q_0, \{q_3\})$  be the following automaton with list variables



where  $\delta = \{(q_0, a^z, q_1), (q_0, a, q_2), (q_1, a, q_3), (q_2, a^z, q_3)\}$ . And let  $G = (\{n_1, n_2\}, \{e_1, e_2\}, \rho, \lambda)$  be the graph database:



where  $\rho(e_1) = (n_1, n_1)$ ,  $\rho(e_2) = (n_1, n_2)$  and  $\lambda(e_1) = \lambda(e_2) = a$ . Now note that for the path  $p = n_1 e_1 n_1 e_2 n_2$ , it holds that  $\llbracket A \rrbracket_p = \{\mu_1, \mu_2\}$  for  $\mu_1 = [z \rightarrow e_1]$ ,  $\mu_2 = [z \rightarrow e_2]$ .

Assume, for contradiction, that for  $A$  there exists an equivalent I/O Unambiguous automaton with list variables  $B = \{Q, \Sigma^V, \delta', q'_0, F\}$ .

For  $p \in G$ , it holds that  $(p, \mu_1) \in \llbracket B \rrbracket_G$  because  $B$  is equivalent to  $A$ , and this implies that there exists a run  $r_1$  of  $B$  over  $p$  such that

$$r_1 = (q'_0, \emptyset) \xrightarrow{a^z} (q'_1, \mu_1) \xrightarrow{a} (q_f, \mu_1)$$

where  $q'_1, q_f \in Q$  and  $q_f \in F$ .

And the same applies for  $(p, \mu_2) \in \llbracket B \rrbracket_G$ , there exists a second run  $r_2$  of  $B$  over  $p$  such that

$$r_2 = (q'_0, \emptyset) \xrightarrow{a} (q'_2, \emptyset) \xrightarrow{a^z} (q'_f, \mu_2)$$

where  $q'_2, q'_f \in Q$  and  $q'_f \in F$ .

Finally consider the path  $p' \in G$  as  $p = n_1 e_1 n_1 e_1 n_1$ , then the following two runs of  $B$  over  $p'$  exist

$$r'_1 = (q'_0, \emptyset) \xrightarrow{a^z} (q'_1, \mu) \xrightarrow{a} (q_f, \mu)$$

$$r'_2 = (q'_0, \emptyset) \xrightarrow{a} (q'_2, \emptyset) \xrightarrow{a^z} (q'_f, \mu)$$

where  $\mu = [z \mapsto e_1]$ .

The runs  $r'_1$  and  $r'_2$  are distinct sequences of configurations and transitions, yet both yield the same final mapping  $\mu = [z \mapsto e_1]$ . Since  $\mu \in \llbracket B \rrbracket_{p'}$  and there are two distinct accepting runs that result in this mapping,  $B$  is not I/O Unambiguous. This is a contradiction.

We conclude that the automaton  $A$  has no equivalent I/O Unambiguous automaton.  $\square$

**Proposition 5.** *There exists an automaton with list variables for which there is no equivalent deterministic automaton with list variables.*

*Proof.* Suppose that there is an equivalent deterministic automaton for any given automaton. Then there exists an equivalent I/O Unambiguous automaton, which contradicts the result of proposition 4.  $\square$

### 4.3 Determinism\*

We also have the following definition of determinism

**Definition 18.** *Let  $A = (Q, \Sigma^V, \delta, q_0, F)$  be an automaton with list variables, we call  $A$  **deterministic\*** if and only if for all  $q_1, q_2, q_3 \in Q$  and  $\sigma, \sigma' \in \Sigma^V$*

$$(q_1, \sigma, q_2), (q_1, \sigma', q_3) \in \delta \implies \sigma \neq \sigma'$$

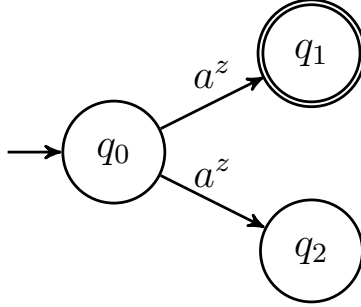
**Proposition 6.** *If  $A$  is an automaton with list variables such that  $A$  is deterministic then  $A$  is also deterministic\**

We would like to compare which notion of determinism is the stronger of I/O Unambiguous and Determinism\*, however, the following proposition tells us that neither is stronger than the other.

**Proposition 7.** *The following statements are true*

- *There exists an automaton with list variables  $A$  such that  $A$  is I/O Unambiguous but  $A$  is not deterministic\*.*
- *There exists an automaton with list variables  $B$  such that  $B$  is deterministic\* but  $B$  is not I/O Unambiguous.*

*Proof.* For the first statement, let  $A$  be the following automaton

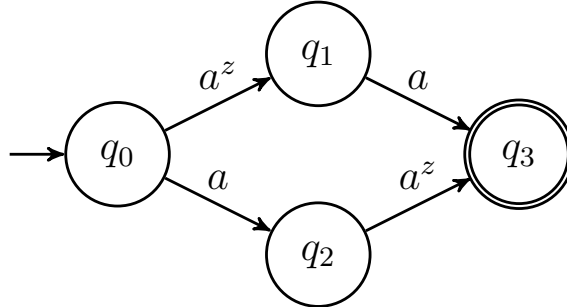


$A$  is clearly not deterministic\*, but  $A$  is I/O Unambiguous because the only one accepting run is the following

$$(q_0, \emptyset) \xrightarrow{a^z} (q_1, \mu)$$

then, for every accepting path there is only one mapping.

For the second statement let  $B$  be the automaton



This automaton is deterministic\* but is not I/O Unambiguous, consider the path  $p = v_0 e_0 v_0 e_0 v_0$  with  $\lambda(e_0) = a$  then for the mapping  $\mu = [z \rightarrow e_0]$  there are two different accepting runs.  $\square$

**Proposition 8.** *Let  $A$  be any finite automaton with list variables; then there exists  $B$  an equivalent automaton such that  $B$  is deterministic\*.*

The proof of this proposition is based on the Powerset Construction [6] in the automaton  $A$ , forming an equivalent automaton that is deterministic\*.

**Proposition 9.** *Let  $A$  be any automaton with list variables such that  $A$  is deterministic\* then  $A$  is also I/O Unambiguous for every pair of input and output,  $(p, \mu)$  where  $p$  is a trail and  $\mu \in \llbracket A \rrbracket_p$ .*

*Proof.* Let  $A$  be an deterministic\* automaton with list variables. Assume  $p$  is a trail and  $\mu \in \llbracket A \rrbracket_p$ . We show that the run  $r$  such that  $\mu_r = \mu$  is unique.

Suppose that there are two accepting runs,  $r$  and  $r'$ , over  $p$  such that  $\mu_{r'} = \mu_r = \mu$ . The output mapping  $\mu$  uniquely determines the sequence of  $\Sigma^V$  transition labels  $\sigma_1 \cdots \sigma_n$  used in  $p$ . This is because an edge  $e_i$  is associated with a specific variable  $z$  (i.e.,  $\sigma_i = a^z$ ) if and only if  $e_i$  is recorded in  $\mu(z)$ . Since the final mappings are identical, the transition label sequences must be equal:  $\sigma_1 \cdots \sigma_n = \sigma'_1 \cdots \sigma'_n$ .

Then let the runs  $r$  and  $r'$  be given by the following:

$$r := (q_0, \mu_0) \xrightarrow{\sigma_1} (q_1, \mu_1) \xrightarrow{\sigma_2} \cdots (q_{n-1}, \mu_{n-1}) \xrightarrow{\sigma_n} (q_n, \mu_n)$$

$$r := (q_0, \mu_0) \xrightarrow{\sigma'_1} (q'_1, \mu'_1) \xrightarrow{\sigma'_2} \cdots (q'_{n-1}, \mu'_{n-1}) \xrightarrow{\sigma'_n} (q'_n, \mu'_n)$$

We already know that for all  $i \in [1, n]$ ,  $\sigma_i = \sigma'_i$  and  $\mu_i = \mu'_i$

Since both runs start in state  $q_0$  and  $\sigma_1 = \sigma'_1$ , therefore, the determinism\* of the automaton guaranties that  $q_1 = q'_1$ , the same argument shows that  $q_2 = q'_2$  and for induction until  $n$  it is demonstrated that  $q_i = q'_i$  for all  $i \in [1, n]$ . So, the entire sequence of configurations  $(q_i, \mu_i)$  that constitutes the run  $r$  is entirely fixed. Therefore,  $r$  and  $r'$  must be identical.

This shows that for a fixed input  $p$  and output  $\mu$ , there is exactly one run  $r$ , which means that  $A$  is I/O Unambiguous.

Thus, for every fixed input path  $p$  and output  $\mu$ , there is exactly one accepting run.  $\square$

## 5 Decidability Problems

In this section, we will study several decidability problems related to the output of an automaton with list variables.

Given a graph database  $G$  and an automaton  $A$  with list variables, we recall that the output of  $A$  over  $G$  is defined as

$$\llbracket A \rrbracket_G := \{(p, \mu) \mid \mu \in \llbracket A \rrbracket_p, \quad p \text{ is a path of } G\}.$$

where,  $\llbracket A \rrbracket_p := \{\mu_r \mid r \text{ is an accepting run of } A \text{ over } p\}$ .

We now introduce the first decidability problem we consider in this section.

### 5.1 The Emptiness Problem

**Definition 19** (Emptiness Problem). *Given a graph database  $G$  and an automaton  $A$  with list variables, the Emptiness Problem asks whether the output of  $A$  over  $G$  is empty; in other words, whether  $\llbracket A \rrbracket_G = \emptyset$ .*

<p><b>Emptiness Problem</b></p> <p><b>Input:</b> <math>G, A</math>.</p> <p><b>Question:</b> Exists a pair <math>(p, \mu) \in \llbracket A \rrbracket_G</math> ?</p>
---

To solve this problem, we will use the following result.

**Proposition 10.** *Let  $G$  be a graph database, and let  $A$  be an automaton with list variables. If  $\llbracket A \rrbracket_G \neq \emptyset$ , then there exists a pair  $(p, \mu) \in \llbracket A \rrbracket_G$  such that  $\text{len}(p) \leq |A| \times |G|$ .*

*Proof.* Suppose that  $\llbracket A \rrbracket_G \neq \emptyset$ , and let  $(p, \mu)$  be some element in this set,  $(p, \mu) \in \llbracket A \rrbracket_G$ .

By definition of the output of  $A$  over  $G$ , we have that  $\mu \in \llbracket A \rrbracket_p$  and  $p = v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$  is a path in  $G$ . By the definition of the output of  $A$  over  $p$ , there exists an accepting run of  $A$  over  $p$  such that  $\mu = \mu_r$ . Let the accepting run  $r$  be the following sequence

$$r := (q_0, \mu_0) \xrightarrow{o_1} (q_1, \mu_1) \xrightarrow{o_2} \cdots (q_{n-1}, \mu_{n-1}) \xrightarrow{o_n} (q_n, \mu_n)$$

and we have for every index  $1 \leq i \leq n$ :

- $(q_i, o_{i+1}, q_{i+1}) \in \delta$
- $\lambda(e_i) = \pi_\Sigma(o_i)$
- $\mu_0 = \emptyset$
- if  $o_i = a$  then  $\mu_{i+1} = \mu_i$
- if  $o_i = a^z$  then  $\mu_{i+1} = \mu_i[z \mapsto \mu_i(z)[e_i]]$

And  $q_n \in F$  because this run  $r$  is accepted.

Now, let us construct the product graph between  $G$  and the automaton  $A$ , and let  $M = |A| \times |G|$  denote its number of vertices. Since  $p$  is a path in  $G$  accepted by  $A$ , we can construct the corresponding path  $p_1 \in G \times A$ .

$$p_1 = (v_0, q_0)(e_1, (q_0, o_1, q_1))(v_1, q_1) \cdots (v_{n-1}, q_{n-1})(e_n, (q_{n-1}, o_n, q_n))(v_n, q_n)$$

If  $\text{len}(p_1) > M$ , the path  $p_1$  must contain a cycle in  $G \times A$  that begins and ends at some node  $(v_i, q_i) = (v_j, q_j)$ . By removing the subpath corresponding to this cycle, we obtain a shorter path  $p_2$  in  $G \times A$ . Since variables are not tracked on the product graph and the transitions are compatible, the resulting path  $p_2$  in  $G$  still has an accepting run in  $A$ . Observe that from the path  $p_1$ ,  $(n - j) - i$  vertices are removed, from which it follows that the length of the path  $p_2$  is  $n > \text{len}(p_2) = n - ((n - j) - i) = j + i$ . We repeat this process until the path  $p_3$  has length  $\leq M$ .

Let us note that  $p_3$  is a path in the product graph  $p_3 \in G \times A$  so if  $p_3 = (v_0, q_0)(e_1, (q_0, o_1, q_1))(v_1, q_1) \cdots (v_m, q_m)$  its corresponds to a path  $p_4 = v_0 e_1 v_1 \cdots v_m$  in  $G$  that has the same length  $m$  as  $p_3$  and takes the automaton  $A$  from the initial state  $q_0$  to the final state  $q_m \in F$ .

We conclude that  $p_4 \in G$  is a path with an accepting run  $r$  in  $A$  and there is a mapping  $\mu_r$  such that  $(p_3, \mu_r) \in \llbracket A \rrbracket_G$  and  $\text{len}(p_3) \leq |A| \times |G|$ .  $\square$

Now we can answer the emptiness problem using this proposition.

**Proposition 11.** *Let  $G$  be a graph database and  $A$  an automaton with list variables. Then the **Emptiness Problem is decidable**, and is solved by Algorithm 1, therefore, it belongs to the complexity class **NL – complete**.*

*Proof.* From the previous proposition, we see that it is sufficient to check for the existence of a path of at most length  $|A| \times |G|$ . If no such path exists,

then by the proposition we conclude that  $\llbracket A \rrbracket_G = \emptyset$ . On the other hand, if such a path exists, we will have an explicit element in  $\llbracket A \rrbracket_G$ .

Given a graph  $G = (V, E, \rho, \lambda)$  and an automaton  $A = (Q, \Sigma^V, \delta, q_0, F)$ , we will use the following algorithm to find a possible element in the output of  $A$  over  $G$ .

This algorithm give us the answer of the Emptiness Problem for an arbitrary graph  $G$  and an automata  $A$ , if accepts then the set  $\llbracket A \rrbracket_G$  is not empty, and if rejects then is empty.

Furthermore, the algorithm operates within nondeterministic logarithmic space, as it only requires keeping track of the current state, the current node in  $G$ , and a counter bounded by  $|A| \times |G|$ , all of which can be stored using logarithmic space. Therefore, the Emptiness Problem belongs to the complexity class **NL**.

To prove **NL-hardness**, we reduce the Graph Reachability Problem, which is **NL-complete** [6], to the Emptiness Problem.

Let  $(G, s, t)$  be an instance of the Reachability Problem. Then  $(G', A)$  is an instance for the Emptiness Problem where:

- The graph database  $G'$  is a single node  $v_0$  and a single self-loop edge  $e_0$  labelled with a symbol  $a$ .
- The automaton  $A$  uses the nodes of  $G$  as its states. The initial state is  $s$ , and the set of final states is the singleton  $\{t\}$ . For every directed edge in  $G$ , we define an  $a$ -labelled transition.

The automaton  $A$  accepts if and only if there is a path from  $s$  to  $t$  in the graph  $G$ , where the automaton is constructed on the fly [6] and the reduction is performed in logarithmic space, so we conclude that the Emptiness Problem is **NL-hard**, and thus **NL-complete**.

□

---

**Algorithm 1** Emptiness Problem

---

```
1: function EMPTINESSPROBLEM( $G, A$ )
2:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
3:    $m \leftarrow |\mathcal{A}| \times |G|$ 
4:   for  $v \in V$  do
5:     startState  $\leftarrow (v, q_0)$ 
6:     current  $\leftarrow$  startState  $\triangleright$  current =  $(v, q)$ 
7:     counter  $\leftarrow 0$ 
8:     while counter  $\leq m$  do
9:       if  $q \in F$  then
10:        Accept
11:        Non-deterministically choose  $v' \in V$  and  $q' \in Q$ 
12:        if there exists  $e \in E$  and  $d \in \delta$  such that  $\rho(e) = (v, v')$  and
13:         $d = (q, o, q')$  then
14:          if  $\lambda(e) = \pi_\Sigma(o)$  then
15:            current  $\leftarrow (v', q')$ 
16:            counter  $\leftarrow$  counter + 1
17:        Reject
```

---

## 5.2 The Path Membership Problem

The following decidability problem is an analogue of the problem just proven, but for the set  $\llbracket A \rrbracket_p$

**Definition 20** (Path Membership Problem). *Given a graph database  $G$ , an automaton with list variables  $A$ , and a path  $p$  in  $G$ , the Path Membership Problem asks whether there exists a mapping  $\mu$  such that the pair  $(p, \mu)$  is in the output of  $A$  over  $G$ , i.e.,  $(p, \mu) \in \llbracket A \rrbracket_G$ .*

**Path Membership Problem**

**Input:**  $G, A, p$ .

**Question:** Exists a mapping  $\mu$  such that  $(p, \mu) \in \llbracket A \rrbracket_G$  ?

**Proposition 12.** *Let  $G$  be a graph database,  $A$  an automaton with list variables, and  $p$  a path in  $G$ . Then the **Path Membership Problem** is **decidable**, and is solved by Algorithm 2, therefore, it belongs to the complexity class **NL – complete**.*

*Proof.* Let  $G$  be a graph database,  $A$  an automaton with list variables, and let  $p = v_0e_1v_1 \cdots v_{n-1}e_nv_n$  be a path in  $G$  with initial node  $v_0$  and final node  $v_n$ . We aim to determine whether there exists a mapping  $\mu$  such that  $(p, \mu) \in \llbracket A \rrbracket_G$ .

By definition, this holds if and only if there exists an accepting run of  $A$  over  $p$ . That is, a sequence of configurations:

$$(q_0, \mu_0) \xrightarrow{o_1} (q_1, \mu_1) \xrightarrow{o_2} \cdots \xrightarrow{o_n} (q_n, \mu_n)$$

such that:

- $(q_{i-1}, o_i, q_i) \in \delta$  for every  $i = 1, \dots, n$ ,
- $\lambda(e_i) = \pi_\Sigma(o_i)$  for every  $i$ ,
- and  $q_n \in F$ .

The algorithm simulates this run non-deterministically. It traverses the path  $p = v_0e_1v_1 \cdots v_{n-1}e_nv_n$ , and at each step  $i$ , it non-deterministically selects a transition  $(q_{i-1}, o_i, q_i) \in \delta$  that is compatible with the label  $\lambda(e_i)$  of the graph, i.e., such that  $\pi_\Sigma(o_i) = \lambda(e_i)$ . Then it updates the current state to  $q_i$ .

At the end of the path, the algorithm accepts if the resulting state  $q_n$  is final (i.e.,  $q_n \in F$ ), and rejects otherwise.

Thus, the algorithm accepts exactly when there exists an accept run of the automaton  $A$  on the path  $p$ , which is equivalent to the existence of a pair  $(p, \mu) \in \llbracket A \rrbracket_G$  for some mapping  $\mu$ .

This shows that the algorithm correctly decides the Path Membership Problem.

The algorithm maintains only the current position in the path, that is, the current state of the automaton, and the current vertex in the graph, which can all be represented using logarithmic space. Hence, the Path Membership Problem is solvable in non-deterministic logarithmic space, and belongs to the class **NL**.

To prove **NL-hardness**, we again reduce the Reachability Problem, to the Path Membership Problem.

Let  $(G, s, t)$  be an instance of the Reachability Problem with  $G = (N, E)$ . Then  $(G', A, p)$  is an instance for the Path Membership Problem where:

- The graph database  $G'$  is a single node  $v_0$  and a single self-loop edge  $e_0$  labeled with a symbol  $a$ .

- The automaton  $A$  uses the nodes  $N$  as its states. The initial state is  $s$ , and the set of final states is a singleton  $\{t\}$ . For every directed edge in  $G$ , we define an  $a$ -labelled transition in  $A$ . In addition, we add an  $a$ -labelled self-transition in the final state.
- The path  $p$  is  $p = v_0 e_0 v_0 \cdots e_0 v_0$  and its length is  $\text{len}(p) = |N|$

The automaton  $A$  accepts the path  $p$  if and only if there is a path from  $s$  to  $t$  in the original graph  $G$ , and as in the previous proof, it is constructed on the fly [6]. The self-loop on  $t$  allows one to extend any such path to exactly length  $|N|$ .

In the path we only count the vertex, and with this, the reduction can be performed in logarithmic space, we conclude that the Path Membership Problem is **NL-complete**.  $\square$

---

**Algorithm 2** Path Membership Problem

---

```

1: function PATHMEMBERSHIPPROBLEM( $A, p$ )
2:   Let  $p = v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$  be the given path in  $G$ 
3:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
4:   startState  $\leftarrow (v_0, q_0)$ 
5:   current  $\leftarrow$  startState  $\triangleright$  current =  $(v, q)$ 
6:   for  $i \in [1, n]$  do
7:     Let  $a_i := \lambda(e_i)$   $\triangleright$  label of edge  $e_i$ 
8:     Non-deterministically choose  $d = (q, o, q') \in \delta$ 
9:     if  $\pi_\Sigma(o) = a_i$  then
10:      current  $\leftarrow (v_i, q')$ 
11:     else
12:       Reject
13:   if  $q \in F$  then
14:     Accept
15:   else
16:     Reject

```

---

### 5.3 The Path Mapping Membership Problem

The next problem is when the mapping and the path are given.

**Definition 21** (Path Mapping Membership Problem). *Given a graph database  $G$ , an automaton with list variables  $A$ , a path  $p$  in  $G$ , and a mapping  $\mu$ , the Path Mapping Membership Problem asks whether the pair  $(p, \mu)$  is in the output of  $A$  over  $G$ , that is,  $(p, \mu) \in \llbracket A \rrbracket_G$ .*

<p><b>Path Mapping Membership Problem</b></p> <p><b>Input:</b> <math>G, A, p, \mu</math>.</p> <p><b>Question:</b> Is <math>(p, \mu) \in \llbracket A \rrbracket_G</math> ?</p>
--

**Proposition 13.** *Let  $G$  be a graph database,  $A$  an automaton with list variables,  $p$  a path in  $G$ , and  $\mu$  a mapping with only one variable  $z$ . Then the **Path Mapping Membership Problem is decidable**, and is solved by Algorithm 3, therefore, it belongs to the complexity class **NL – complete**.*

*Proof.* We prove that the algorithm correctly decides whether  $(p, \mu) \in \llbracket A \rrbracket_G$ .

Given the path  $p = v_0 e_1 v_1 \cdots e_n v_n$  and a mapping  $\mu$  with a single variable  $z$ , the algorithm iterates through the edges  $e_1, \cdots, e_n$  of  $p$  while simulating a run of the automaton  $A$  over this path. At each step, it non-deterministically selects a transition  $(q, o, q')$  such that the label of the edge matches the observable part of  $o$ . If the label involves storing into variable  $z$  (i.e.,  $o = a^z$ ), it checks whether the current edge  $e_i$  is consistent with the  $k$ -th entry in  $\mu(z)$ . The index  $k$  is incremented accordingly.

At the end of the path, the algorithm accepts if the final state is accepting ( $q \in F$ ) and the full mapping  $\mu(z)$  has been consumed, i.e.,  $k = m + 1$ , where  $m = \text{len}(\mu(z))$ . This guarantees that the mapping  $\mu$  is exactly reconstructed by the run, and the run is accepting.

Hence, the algorithm accepts if and only if there exists an accepting run  $r$  of  $A$  over  $p$  such that  $\mu_r = \mu$ , i.e.  $(p, \mu) \in \llbracket A \rrbracket_G$ .

Therefore, the Path Mapping Membership Problem is decidable.

The algorithm only needs to store the current edge index, the automaton state, and the current position in the mapping  $\mu(z)$ , all of which require logarithmic space. Therefore, the Path Mapping Membership Problem (with a single variable) lies in the complexity class **NL**.

To prove **NL-hardness**, let  $(G, A, p)$  be an instance of the Path Membership Problem. Then  $(G, A', p, \mu_0)$  is an instance for the Path Mapping Membership Problem where  $\mu_0$  is the empty mapping.

Since solving this instance is equivalent to deciding the original Path Membership instance, which is **NL-complete** (Proposition 12), the Path Mapping Membership Problem is also **NL-complete**.  $\square$

---

**Algorithm 3** Path Mapping Membership Problem

---

```

1: function PATHMAPPINGMEMBERSHIPPROBLEM( $A, p, \mu$ )
2:   Let  $p = v_0e_1v_1 \cdots v_{n-1}e_nv_n$ 
3:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
4:    $m \leftarrow \text{len}(\mu(z))$ 
5:    $\text{startState} \leftarrow (v_0, q_0)$ 
6:    $\text{current} \leftarrow \text{startState}$   $\triangleright \text{current} = (v, q)$ 
7:    $k \leftarrow 1$ 
8:   for  $i \in [1, n]$  do
9:     Let  $a_i := \lambda(e_i)$ 
10:    Non-deterministically choose  $(q, o, q') \in \delta$ 
11:    if  $o = a_i^z$  then
12:      if  $\mu(z)_k = e_i$  then
13:         $\text{current} \leftarrow (v_i, q')$ 
14:         $k \leftarrow k + 1$ 
15:      else
16:        Reject
17:      else if  $o = a_i$  then
18:         $\text{current} \leftarrow (v_i, q')$ 
19:      else
20:        Reject
21:    if  $q \in F$  and  $k = m + 1$  then
22:      Accept
23:    else
24:      Reject

```

---

Now, we consider the more general case of the Path Mapping Membership Problem where the mapping  $\mu$  may contain multiple variables.

Recall that the problem has already been defined for an arbitrary mapping  $\mu$ , and the previous results addressed the case where  $\mu$  contains only one variable. However, when  $\mu$  contains multiple variables, the complexity of verifying the correctness of the mapping over a given path increases, but with one extra condition the complexity is the same.

For this next proposition we need the following definition,

**Definition 22.** *Let  $\mu$  be a mapping. We define the set*

$$E_\mu = \{e \mid e \in \mu(z) \text{ for some } z \in \text{Dom}(\mu)\}.$$

**Proposition 14.** *Let  $G$  be a graph database,  $A$  an automaton with list variables,  $p$  a path in  $G$ , and  $\mu$  a mapping, such that each edge  $e \in E_\mu$  occurs exactly once in the path  $p$ . Then the **Path Mapping Membership Problem is decidable**, and is solved by Algorithm 4, therefore, it belongs to the complexity class **NL – complete**.*

Note that the imposed condition is satisfied if the path  $p$  is a trail; however, it is even more general since the path could not be a trail and the condition remains true.

*Proof.* We prove that the algorithm correctly decides whether  $(p, \mu) \in \llbracket A \rrbracket_G$ . We first define an auxiliary list  $L$  that arranges the edges of  $E_\mu$  in the order in which they appear along the path  $p$ . This construction is well-defined, since each edge occurs exactly once in this path.

The algorithm then iterates through the edges  $e_1, \dots, e_n$  of  $p$  while simulating a run of the automaton  $A$  over this path. At each step, it non-deterministically selects a transition  $(q, o, q')$  such that the label of the edge matches the observable part of  $o$ . If the label involves storing into some variable  $z \in \text{Dom}(\mu)$  (i.e.,  $o = a^z$ ), it checks whether the current edge  $e_i$  is consistent with the  $k$ -th entry in  $L$ . The index  $k$  is incremented accordingly.

At the end of the path, the algorithm accepts if the final state is accepting ( $q \in F$ ) and the full mapping  $\mu(z)$  has been consumed, i.e.,  $k = m + 1$ , where  $m = |E_\mu|$ . This guarantees that the mapping  $\mu$  is exactly reconstructed by the run, and the run is accepting.

Hence, the algorithm accepts if and only if there exists an accepting run  $r$  of  $A$  over  $p$  such that  $\mu_r = \mu$ , i.e.  $(p, \mu) \in \llbracket A \rrbracket_G$ .

Therefore, the Path Mapping Membership Problem is decidable.

The algorithm only needs to store the current edge index, the automaton state, and the current position in the list  $L$ , all of which require logarithmic space. Therefore, the Path Mapping Membership Problem in this case also lies in the complexity class **NL**.

**NL – hardness** follows as this generalizes the **NL – complete** Path Membership Problem in proposition 13  $\square$

---

**Algorithm 4** Path Mapping Membership Problem (Multiple Variables)

---

```
1: function PMMP2( $A, p, \mu$ )
2:   Let  $p = v_0e_1v_1 \cdots v_{n-1}e_nv_n$ 
3:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
4:    $m \leftarrow |E_\mu|$ 
5:   if  $m > n$  then
6:     Reject
7:   startState  $\leftarrow (v_0, q_0)$ 
8:   current  $\leftarrow$  startState  $\triangleright$  current =  $(v, q)$ 
9:   L = NewList()
10:  for  $j \in [1, n]$  do
11:    if  $e_j \in E_\mu$  then
12:      L.append( $e_j$ )
13:   $k \leftarrow 1$ 
14:  for  $i \in [1, n]$  do
15:    Let  $a_i := \lambda(e_i)$ 
16:    Non-deterministically choose  $(q, o, q') \in \delta$ 
17:    if  $o = a_i^z$ , for some  $z \in \text{Dom}(\mu)$  then
18:      if  $L_k = e_i$  and  $e_i \in \mu(z)$  then
19:        current  $\leftarrow (v_i, q')$ 
20:         $k \leftarrow k + 1$ 
21:      else
22:        Reject
23:      else if  $o = a_i$  then
24:        current  $\leftarrow (v_i, q')$ 
25:      else
26:        Reject
27:  if  $q \in F$  and  $k = m + 1$  then
28:    Accept
29:  else
30:    Reject
```

---

In the most general case of this problem, where the conditions established in the previous propositions are not satisfied, the problem belongs to NP, as stated in the following proposition, which will be demonstrated below.

**Proposition 15.** *The general case of the Path Mapping Membership Problem is NP-complete.*

*Proof.* We reduce the directed Hamiltonian path problem, which is known to be NP-complete [6]. Let  $G^* = (N^*, E^*)$  be a directed graph and  $s, t \in N^*$ . We ask whether there exists a Hamiltonian path from  $s$  to  $t$ , that is, a path that visits each node exactly once, starting at  $s$  and ending at  $t$ .

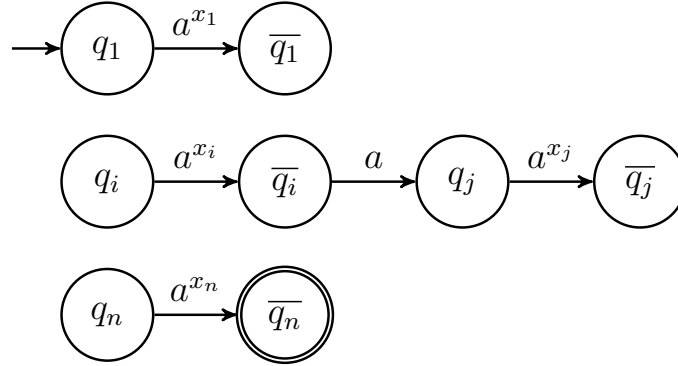
We construct an instance of the Path Mapping Membership problem, that is, a graph  $G$ , an automaton with list variables  $A$ , a path  $p$ , and a mapping  $\mu$ , such that:

$$G^* \text{ has a Hamiltonian path from } s \text{ to } t \iff (p, \mu) \in \llbracket A \rrbracket_G.$$

We begin the construction by defining  $n = |N^*|$ , and let the nodes of  $G^*$  be enumerated as  $\{v_1, v_2, \dots, v_n\}$  such that  $v_1 = s$  and  $v_n = t$ . Then we define the following components:

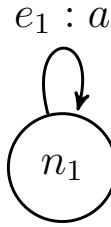
- **Automaton with list variables**  $A = (Q, \Sigma^V, \delta, q_0, F)$ :
  - The set of states is  $Q = \{q_1, \dots, q_n\} \cup \{\bar{q}_1, \dots, \bar{q}_n\}$ .
  - The set of variables is  $V = \{x_1, x_2, \dots, x_n\}$  and  $\Sigma = \{a\}$ .
  - The transition relation  $\delta$  is defined by listing the triplets that compose it:
    - \* For each  $1 \leq i \leq n$ ,  $(q_i, a^{x_i}, \bar{q}_i)$ .
    - \* For each directed edge  $(v_i, v_j) \in E^*$ , we create a transition in  $A$ :  $(\bar{q}_i, a, q_j)$ .
  - The initial state is  $q_0 = q_1$ , and the set of final states is  $F = \{\bar{q}_n\}$ .

For each directed edge  $(v_i, v_j) \in E^*$  the automaton has the following structure:



• **Graph**  $G = (N, E, \rho, \lambda)$ :

- The set of nodes is  $N = \{n_0\}$ .
- The set of edges is  $E = \{e_1\}$ , where  $\rho(e_1) = (n_0, n_0)$  and  $\lambda(e_1) = a$ .



- **Mapping**  $\mu$ : The mapping is fixed to bind each variable  $x_i$  to its corresponding edge  $e_1$ . That is,  $\mu = ([x_1 \rightarrow e_1], [x_2 \rightarrow e_1], \dots, [x_n \rightarrow e_1])$ .
- **Path**  $p$ : The path is  $p = n_0, e_1, n_0, \dots, e_1, n_0$  with  $\text{len}(p) = 2n$ .

The construction of this instance is clearly polynomial in the size of the original graph  $G^*$ .

The next part of the proof is the equivalence proof, where we show both directions of the equivalence.

$$G^* \text{ has a Hamiltonian path from } s \text{ to } t \iff (p, \mu) \in \llbracket A \rrbracket_G.$$

( $\Rightarrow$ ) Suppose that  $G^*$  has a Hamiltonian path  $v_1, v_{i_2}, \dots, v_{i_{n-1}}, v_n$  from  $s$  to  $t$ . We can construct a corresponding accepting run of the automaton  $A$

over the path  $p$  in  $G$  as follows: the run begins at  $q_1$ , consumes the subpath  $\mu(x_1) = e_1$ , and transitions to  $\bar{q}_1$ ; then, since  $(v_1, v_{i_2}) \in E^*$ , there exists a transition  $(\bar{q}_1, a, q_{i_2}) \in \delta$ . The automaton proceeds analogously for each edge  $(v_{i_k}, v_{i_{k+1}})$  in the Hamiltonian path, following the corresponding transitions  $(\bar{q}_{i_k}, a, q_{i_{k+1}})$ . Finally, the automaton reaches  $\bar{q}_n$ , which is a final state. Therefore, the run is accepting and  $(p, \mu) \in \llbracket A \rrbracket_G$ .

( $\Leftarrow$ ) Conversely, suppose that  $(p, \mu) \in \llbracket A \rrbracket_G$ . Then there exists an accepting run of  $A$  over  $p$ . By construction, any valid run of  $A$  must alternate between states  $q_i$  and  $\bar{q}_i$ , where each transition  $(\bar{q}_i, a, q_j)$  corresponds to an edge  $(v_i, v_j) \in E^*$  in the original graph  $G^*$ . Moreover, the automaton must traverse exactly one pair  $(q_i, \bar{q}_i)$  for each variable  $x_i$ , since the mapping  $\mu$  assigns all variables to the same edge  $e_1$  and the automaton structure forces visiting each  $x_i$  exactly once to reach the final state  $\bar{q}_n$ . Hence, the sequence of indices visited by the automaton determines a path  $v_1, v_{i_2}, \dots, v_{i_{n-1}}, v_n$  in  $G^*$  that visits each vertex exactly once, i.e., a Hamiltonian path from  $s$  to  $t$ .

Since the reduction is polynomial and the membership of  $(p, \mu)$  in  $\llbracket A \rrbracket_G$  can be verified in polynomial time given the mapping, the problem belongs to NP. Thus, the general case of the Path Mapping Membership Problem is NP-complete.  $\square$

## 5.4 The Mapping Membership Problem

Finally, the last decidability problem is analogous to the Path Membership Problem, but in this case, the path is unknown and the mapping is given.

**Definition 23** (Mapping Membership Problem). *Given a graph database  $G$ , an automaton with list variables  $A$ , and a mapping  $\mu$ , the Mapping Membership Problem asks if exists a path  $p \in G$  such that the pair  $(p, \mu)$  is in the output of  $A$  over  $G$ , i.e.,  $(p, \mu) \in \llbracket A \rrbracket_G$ .*

<p><b>Mapping Membership Problem</b></p> <p><b>Input:</b> <math>G, A, \mu</math>.</p> <p><b>Question:</b> Exists <math>p</math> a path in <math>G</math> such that <math>(p, \mu) \in \llbracket A \rrbracket_G</math> ?</p>
--

To solve this problem, we will use an algorithm similar to Algorithm 3. For this, we must consider a bound on the length of a possible path associated with the mapping, which depends on the size of the mapping as well as on the automaton and the graph.

**Proposition 16.** *Let  $A$  be an automaton with list variables and  $G$  a graph. If  $\mu$  is a mapping with one variable  $z$  such that there exists a path  $p \in G$  with  $\mu \in \llbracket A \rrbracket_p$  then there exists a path  $p' \in G$  such that  $\mu \in \llbracket A \rrbracket_{p'}$  and  $\text{len}(p') \leq (|A| \times |G|) \times (\text{len}(\mu(z)) + 1)$ .*

*Proof.* Assume there exists a path  $p = v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$  in  $G$  such that  $\mu \in \llbracket A \rrbracket_p$ , where  $\mu$  is a mapping with  $\text{Dom}(\mu) = \{z\}$ .

By the semantics of  $\llbracket A \rrbracket_p$ , this means there exists an accepting run

$$(q_0, \mu_0) \xrightarrow{o_1} (q_1, \mu_1) \xrightarrow{o_2} \cdots \xrightarrow{o_n} (q_n, \mu_n)$$

of the automaton  $A$  over the path  $p$ , where  $\mu_n = \mu$ . Moreover,  $\pi_\Sigma(o_i) = \lambda(e_i)$ .

Let  $m := \text{len}(\mu(z))$  be the number of times the variable  $z$  is updated in the run (i.e., the number of transitions labeled with  $a^z$ ).

We now consider the product graph  $G \times A$ , whose nodes are pairs  $(v, q)$  with  $v \in V$  and  $q \in Q$ . The number of such pairs is  $|G| \times |A|$ , which we denote by  $M$ .

The accepting run over  $p$  corresponds to a path in the product graph:

$$(v_0, q_0)(e_1, (q_0, o_1, q_1))(v_1, q_1)(e_2, (q_1, o_2, q_2)) \cdots (e_n, (q_{n-1}, o_n, q_n))(v_n, q_n)$$

We now observe the following key fact:

- Between two successive updates to the variable  $z$ , the run may contain cycles in  $G \times A$  that do not affect the accumulated value of  $\mu(z)$ . In particular, these cycles can be removed without changing the overall mapping  $\mu$ , as long as we preserve the sequence of edges stored in  $\mu(z)$ .

Thus, we can partition the run into at most  $m + 1$  segments: - One segment before the first  $a^z$  transition, - One segment between each pair of  $a^z$  transitions, - One segment after the last  $a^z$  transition.

Each such segment is a run in the product graph from some state  $(v, q)$  to another, without appending to  $z$ . Therefore, each segment can be shortened (by removing cycles) to have length at most  $M = |A| \times |G|$ .

It follows that the entire run can be compressed to a path  $p'$  in  $G$  with corresponding run in  $A$  over  $p'$  such that  $\mu \in \llbracket A \rrbracket_{p'}$  and  $\text{len}(p') \leq M \times (m+1) = (|A| \times |G|) \times (\text{len}(\mu(z)) + 1)$ .

Hence, the desired path  $p'$  exists.  $\square$

**Proposition 17.** *Let  $G$  be a graph database,  $A$  an automaton with list variables, and  $\mu$  a mapping with only one variable  $z$ . Then the **Mapping Membership Problem is decidable**, and is solved by Algorithm 5, therefore, it belongs to the complexity class **NL – complete**.*

*Proof.* Let  $A$  be an automaton with list variables,  $G$  a graph database, and let  $\mu$  be a mapping with  $\text{Dom}(\mu) = \{z\}$ .

We aim to decide whether there exists a path  $p$  in  $G$  such that  $(p, \mu) \in \llbracket A \rrbracket_G$ . By definition of the output of  $A$  over  $G$ , this means that there exists a path  $p$  such that  $\mu \in \llbracket A \rrbracket_p$ , that is, there exists an accepting run of  $A$  over  $p$  such that the resulting mapping  $\mu_r$  coincides with  $\mu$ .

Let  $n := \text{len}(\mu(z))$  be the number of edges in the list assigned to  $z$  by the mapping. Then, since every  $a^z$  operation appends one edge to  $\mu(z)$  during an accepting run, any such run must contain exactly  $n$  transitions labeled  $a^z$  for some  $a \in \Sigma$ . Moreover, the total number of transitions in the run (and thus the total length of the path  $p$ ) is at least  $n$  and could be longer if there are transitions that do not contribute to the mapping (e.g., transitions labeled  $a$  instead of  $a^z$ ).

Let us denote  $M = |A| \times |G|$ , the number of states in the product graph  $G \times A$ . Since each configuration in a run is determined by a node in  $G$  and a state in  $A$ , the run can be compressed (by removing cycles) to a length of at most  $M$  for each edge in  $\mu(z)$ . Therefore, we can bound the total length of

the path by  $M \times (n + 1)$ .

The algorithm performs a non-deterministic search through all possible runs of  $A$  over paths of  $G$ , trying to match the given mapping  $\mu(z)$  along the way. At each step, the algorithm selects a transition  $(q, o, q') \in \delta$  and an edge  $e \in E$  such that the edge label matches the transition symbol, and the edge's source and target match the current and next graph nodes. The algorithm keeps track of a position index  $k$  to ensure that the sequence of edges assigned to  $z$  matches exactly the one given by  $\mu(z)$ .

Whenever the automaton reaches a final state and the complete mapping has been matched (i.e.,  $k = n + 1$ ), the algorithm accepts. Otherwise, it continues the search, stopping once the maximum bound  $|A| \times |G| \times (n + 1)$  is exceeded.

Because the search is exhaustive up to this bound, and because all valid accepting runs with the given mapping are guaranteed to lie within it, the algorithm correctly decides whether there exists a path  $p$  such that  $(p, \mu) \in \llbracket A \rrbracket_G$ . Therefore, the Mapping Membership Problem is decidable for mappings with a single variable.

The algorithm keeps track of the current node, automaton state, and an index counter over  $\mu(z)$ , which are all logarithmic in size. Thus, the Mapping Membership Problem (with a single variable) is solvable in nondeterministic logarithmic space and is in the class **NL**.

**NL – hardness** follows, as this generalizes the **NL – complete** Emptiness Problem in proposition 11 with the mapping  $\mu$  being the empty mapping.  $\square$

---

**Algorithm 5** Mapping Membership Problem

---

```
1: function MAPPINGMEMBERSHIPPROBLEM( $A, G, \mu$ )
2:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
3:    $n \leftarrow \text{len}(\mu(z))$ 
4:    $m \leftarrow (|A| \times |G|) \times (n + 1)$ 
5:   for  $v \in V$  do
6:     startState  $\leftarrow (v, q_0)$ 
7:     current  $\leftarrow$  startState  $\triangleright$  current =  $(v, q)$ 
8:     counter  $\leftarrow 0$ 
9:      $k \leftarrow 1$ 
10:    while counter  $\leq m$  do
11:      if  $q \in F$  and  $k = n + 1$  then
12:        Accept
13:        Non-deterministically choose  $v' \in V$  and  $q' \in Q$ 
14:        if there exists  $e \in E$  and  $d \in \delta$  such that  $\rho(e) = (v, v')$  and
15:         $d = (q, o, q')$  then
16:          if  $o = a^z$  and  $\lambda(e) = a$  then
17:            if  $\mu(z)_k = e$  then
18:              current  $\leftarrow (v', q')$ 
19:              counter  $\leftarrow$  counter + 1
20:               $k \leftarrow k + 1$ 
21:            else if  $o = a$  and  $\lambda(e) = a$  then
22:              current  $\leftarrow (v', q')$ 
23:              counter  $\leftarrow$  counter + 1
24:        Reject
```

---

Now we consider a more general case of the Mapping Membership Problem, where the mapping  $\mu$  may contain multiple variables.

Recall that the problem is already defined for an arbitrary mapping  $\mu$ , and the previous results addressed the case where  $\mu$  contains only one variable. However, when  $\mu$  contains multiple variables, the situation becomes more complicated.

To address the general case, we begin with a result that establishes a bound on the length of paths that need to be considered. This bound depends on the automaton, the graph, and the total size of the mapping.

**Proposition 18.** *Let  $A$  be an automaton with list variables and  $G$  a graph. If  $\mu$  is a mapping such that there exists a path  $p \in G$  with  $\mu \in \llbracket A \rrbracket_p$  then there exists a path  $p' \in G$  such that  $\mu \in \llbracket A \rrbracket_{p'}$  and  $\text{len}(p') \leq (|A| \times |G|) \times (|\mu| + 1)$  where  $|\mu| := \sum_{z \in \text{Dom}(\mu)} \text{len}(\mu(z))$ .*

*Proof.* The proof is analogous to the single-variable case Proposition 16. The only difference is that now the total number of variable updates is  $|\mu| := \sum_{z \in \text{Dom}(\mu)} \text{len}(\mu(z))$ .

As before, we can divide the accepting run of  $A$  over  $p$  into  $|\mu| + 1$  segments, each bounded in length by  $|A| \times |G|$ , by removing cycles between consecutive variable updates. Therefore, the total length of the path can be bounded by  $(|A| \times |G|) \times (|\mu| + 1)$ .  $\square$

Despite the fact that a similar decision procedure can still be applied, the space requirements increase significantly. The algorithm must simultaneously track the positions within each list  $\mu(z)$  for every variable  $z \in \text{Dom}(\mu)$ , and ensure that the edges used in the run correspond exactly to the prescribed lists in the correct order. This additional bookkeeping results in space usage that is no longer logarithmic in the input size, and therefore, the problem is not guaranteed to lie in the class **NL** under this general setting.

We then show that the problem is *NP complete*.

**Proposition 19.** *The General case of the Mapping Membership problem is NP-Complete.*

*Proof.* Membership in NP is immediate: a nondeterministic verifier can guess a path  $p$  (of polynomial length by Proposition 18) and a run of  $A$  over  $p$ , then verify in polynomial time that the run is accepting and that it reconstructs the mapping  $\mu$ .

For NP-hardness, observe that the reduction from the Directed Hamiltonian Path problem given in the proof of the proposition 14 produces in polynomial time an instance  $(G, A, p, \mu)$  such that

$$G^* \text{ has a Hamiltonian path} \iff (p, \mu) \in \llbracket A \rrbracket_G.$$

Dropping the explicit path  $p$  yields an instance  $(G, A, \mu)$  of the Mapping Membership problem with the property that there exists some  $p'$  with  $(p', \mu) \in \llbracket A \rrbracket_G$  if and only if  $G^*$  has a Hamiltonian path, showing that the problem is NP-hard.

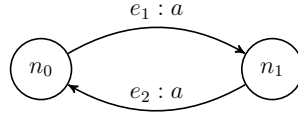
Combining both facts, we conclude that the general Mapping Membership problem is NP-complete.  $\square$

## 6 Algorithms

Finally, we are interested in how to practically find elements in the output of an arbitrary automaton with list variables. In this section, we adapt the algorithms already developed for the general case [4], converting them into algorithms for automata with list variables. These will allow us to find paths with their respective mappings in the output of  $A$  to a list-variable automaton with certain restrictions.

First, let us observe that for the general case, the problem is not so simple since the output could be infinite.

**Example 7.** Consider the following graph database  $G$



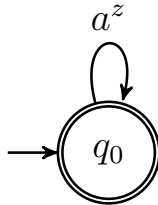
defined as follows  $G = (N, E, \rho, \lambda)$  where:

- $N = \{n_0, n_1\}$
- $E = \{e_1, e_2\}$
- $\rho(e_1) = (n_0, n_1), \rho(e_2) = (n_1, n_0)$
- $\lambda(e_1) = a = \lambda(e_2)$

And let us consider the following automaton  $A = (Q, \Sigma^V, \delta, q_0, F)$  where:

- $Q = \{q_0\}$ .
- $\Sigma^V = \{a^z\}$ .
- $\delta = \{(q_0, a^z, q_0)\}$ .
- $F = \{q_0\}$ .

The automaton looks like this:



For this automaton, there are infinitely many output pairs, which makes listing and finding all of them impossible.

The claim that they are infinite stems from the following. For every  $k \in \mathbb{N}$ , there exists the pair  $(p_k, \mu_k)$  where  $p_k$  is a path of length  $k$  that alternates between  $e_1$  and  $e_2$ , and its corresponding mapping  $\mu_k$  is the one that assigns to  $z$  the list of all edges in  $p_k$ .

As we see in the example, the output of an automaton could be infinite; however, if we restrict ourselves to the shortest paths in the previous example, we would only have two possible output pairs.

Therefore, we will first focus on developing algorithms that find the shortest paths. For this, we will need the following definition.

**Definition 24.** Given a graph database  $G = (N, E, \rho, \lambda)$  and an automaton with list variables  $A = (Q, \Sigma^V, \delta, q_0, F)$

**The product graph**  $G \times A$  is then defined as the graph database  $G \times A = (N_\times, E_\times, \rho_\times, \lambda_\times)$ , where

- $N_\times = N \times Q$ ;
- $E_\times = \{(e, d) \in E \times \delta \mid \lambda(e) = \pi_\Sigma(d)\}$ ;
- $\rho_\times(e, d) = ((n_1, q_1), (n_2, q_2))$  if:
  - $\rho(e) = (n_1, n_2)$
  - $\lambda(e) = a$
  - $d = (q_1, a, q_2)$  or  $d = (q_1, a^z, q_2)$
- $\lambda_\times((e, d)) = \lambda(e)$ .

Intuitively, the product graph is the graph database obtained by the cross product of the graph  $G$  and the automaton  $A$ . And each node of the form  $(n, q)$  in  $G_\times$  corresponds to the node  $n$  in  $G$ , with a state  $q$  in the automaton  $A$  and, furthermore, each path  $P$  of the form  $(v, q_0), (v_1, q_1), \dots, (v_n, q_n)$  in  $G_\times$  corresponds to a path  $p = v, v_1, \dots, v_n$  in  $G$  that has the same length as  $P$  and brings the automaton  $A$  from state  $q_0$  to  $q_n$ .

As such, when  $q_n \in F$  then this path in  $G$  matches  $A$ . In other words, all nodes  $v'$  that can be reached from  $v$  by a path that matches  $A$  can be found by graph search algorithms on  $G \times A$  starting in node  $(v, q_0)$ .

## 6.1 Any Shortest

Given an automaton  $A$  and a graph  $G$ , using the product graph, we can determine any shortest path starting at a given vertex  $v$  of  $G$  with the following algorithm. The idea is that we can perform a graph search algorithm starting at the node  $(v, q_0)$  of the product graph  $G \times A$ .

---

**Algorithm 6** Any Shortest Path in  $G$  that matches  $A$  starting in node  $v$ .

---

```

1: function ANYWALK( $G, A$ )
2:    $\mathcal{A} \leftarrow A$   $\triangleright q_0$  initial,  $F$  final states
3:   Open.init(); Visited.init(); ReachedFinal.init()
4:   startState  $\leftarrow (v, q_0, null, \perp)$ 
5:   Visited.push(startState); Open.push(startState)
6:   while Open  $\neq \emptyset$  do
7:     current  $\leftarrow$  Open.pop()  $\triangleright$  current =  $(n, q, e, prev)$ 
8:     if  $q \in F$  and  $n \notin$  ReachedFinal then
9:       Solutions.add(current)
10:      ReachedFinal.add( $n$ )
11:     for each  $(q, \sigma, q') \in \delta$  do
12:       for each  $e' \in E$  such that  $\rho_1(e') = n$  and  $\lambda(e') = \pi(\sigma)$  do
13:          $n' \leftarrow \rho_2(e')$ 
14:         if  $(n', q', *, *) \notin$  Visited then
15:           if  $\pi_\Sigma(\sigma) = \sigma$  then
16:             newState  $\leftarrow (n', q', e', current)$ 
17:           else
18:             newState  $\leftarrow (n', q', z : e', current)$ 
19:           Visited.push(newState);
20:           Open.push(newState)

```

---

The basic object we manipulate in this algorithm is a *search state*, i.e., a quadruple of the form  $(n, q, e, prev)$ , where  $n$  is the node of  $G$  we are currently exploring,  $q$  is the current state of  $\mathcal{A}$ , while  $e$  is the edge of  $G$  we used to reach  $n$ , and  $prev$  is a pointer to the search state we used to reach  $(n, q)$  in  $G \times A$ . Intuitively, the  $(n, q)$ -part of the search state allows us to track the node of  $G \times A$  we are traversing, while  $e$ , together with  $prev$  allows to reconstruct the path from  $(v, q_0)$  that we used to reach  $(n, q)$ . The algorithm uses four data structures:

- **Open**, which is a queue (in case of BFS), or stack (in case of DFS) of search states, with usual `push()` and `pop()` methods.
- **Visited**, which is a dictionary of search states we have already visited in our traversal, maintained so that we do not end up in an infinite loop. We assume that  $(n, q)$  can be used as a search key to check if some  $(n, q, e, prev) \in \text{Visited}$ . We remark that *prev* always points to a state stored in **Visited**.
- **Solutions**, which is a set containing (pointers to) search states in **Visited** that encode a solution path to be returned; and
- **ReachedFinal**, a set containing nodes we already returned as query answers, in case we re-discover them via a different end state (recall that an NFA can have several end states).

The algorithm explores the product  $G \times A$  using either BFS (**Open** is a queue) or DFS (**Open** is a stack), starting from  $(v, q_0)$ . It starts by initializing the data structures and setting up the start node in  $G \times A$  (lines 2–5). The main loop of line 6 is the classical BFS/DFS algorithm that pops an element  $(n, q, e, prev)$  from **Open** (line 7) and starts exploring its neighbors in  $G \times A$  (lines 11–13). When exploring  $(n, q, e, prev)$ , we scan all the transitions  $(q, \sigma, q')$  of  $\mathcal{A}$  that originate from  $q$  (line 11), and look for neighbors of  $n$  in  $G$  reachable by an  $\pi_\Sigma(\sigma)$ -labeled edge (line 12).

Here  $(n', q', e')$  is a neighbour of  $(n, q)$  in  $G \times \mathcal{A}$ , that is,  $\rho(e') = (n, n')$  in  $G$ , and  $(q, \lambda(e'), q')$  or  $(q, \lambda(e')^z, q')$  is a transition of  $\mathcal{A}$ , when  $\lambda(e') = \pi_\Sigma(\sigma)$ .

If the pair  $(n', q')$  has not been visited yet, we add it to **Visited** and **Open** (lines 14–20), which allows it to be expanded later on in the algorithm.

Depending on the form of the transition  $(q, \sigma, q')$  we define the new state with simply the edge  $e'$ ,  $(n', q', e', current)$ , or  $z : e$  to mark the existence of a transition  $(q, \lambda(e)^z, q')$  (lines 15–18).

When popping from **Open** in line 7, we also check if  $q$  is a final state and that  $n$  has not been reached by a solution path yet (line 8). In this case we found a new solution; i.e., a path from  $v$  to  $n$  which matches  $A$ , so we add it to **Solutions** (line 9) and record it as reached (line 10). The **ReachedFinal** set is used to ensure that each solution is returned only once.

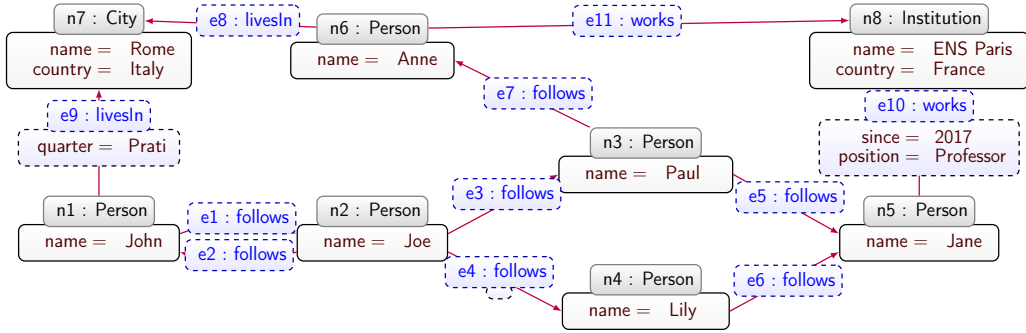
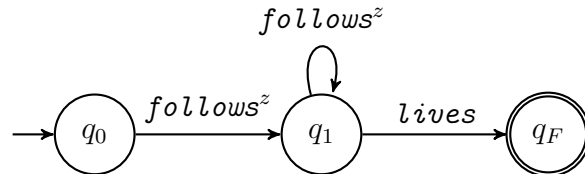


Figure 1: A sample property graph representing a (part of a) social network.

**Example 8.** Consider the graph  $G$  in Figure 1, and the following query

$$\text{ANY SHORTEST WALK } (\text{John}, (\text{follows}^z)^+ \cdot \text{lives}, x).$$

Namely, we wish to find places where people that John follows live. Looking at the graph in Figure 1, we see that Rome is such a place, and the shortest path reaching it starts with John, and loops back to him using the edges  $e_1$  and  $e_2$ , before reaching Rome (via  $e_9$ ), as required. To compute the answer, Algorithm 6 first needs to convert the regular expression  $(\text{follows}^z)^+ \cdot \text{lives}$  into the following automaton  $A$ :



To find shortest paths, we use Algorithm 6 and explore the product graph starting at  $(n_1, q_0)$ . The algorithm then explores the only neighbor that can be reached  $(n_2, q_1)$ , and continues by visiting  $(n_1, q_1)$ ,  $(n_3, q_1)$  and  $(n_4, q_1)$ . When expanding  $(n_1, q_1)$  the first solution,  $n_7$ , is found and recorded in **Solutions**. The algorithm continues by reaching  $(n_6, q_1)$  and  $(n_5, q_1)$  from  $(n_3, q_1)$ . When the  $(n_4, q_1)$  node is then expanded, it would try to reach  $(n_5, q_1)$  again, which is blocked in line 14. Expanding  $(n_6, q_1)$  would try to revisit  $n_7$ , but since this solution was already returned, we ignore it.

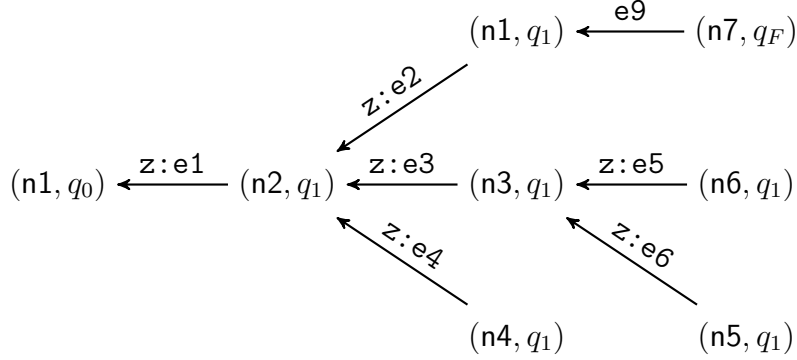


Figure 2: Visited after running Algorithm 6 in Example 8.

The structure of *Visited* after performing the algorithm is illustrated in Figure 2. Here we represent the pointer *prev* as an arrow to other search states in *Visited*, and annotate the arrow with the edge witnessing the connection. Notice that we can revisit a node of  $G$  (e.g.  $n_1$ ), but not a node of  $G \times A$  (e.g.  $(n_5, q_1)$ ). Since *Solutions* contains only  $(n_7, q_F)$ , we enumerate a single path traced by the edges  $e_1 \rightarrow e_2 \rightarrow e_9$ , and the mapping  $\mu = z \mapsto [e_1, e_2]$ .

This example illustrates how the algorithm effectively finds a shortest path if one exists.

## 6.2 All Shortest Paths

We know that we can find any shortest path; however, the path delivered by the algorithm is arbitrary and might even lack an associated mapping, which would not be convenient if we required the path to have an associated mapping with specific characteristics. Therefore, the next algorithm finds all possible shortest paths.

---

**Algorithm 7** All Shortest Paths in  $G$  that matches  $A$  starting in node  $v$ .

---

```

1: function ALLSHORTESTWALK( $G, A$ )
2:    $\mathcal{A} \leftarrow \text{Unambiguousautomaton}(A)$   $\triangleright q_0$  initial,  $F$  final states
3:   Open.init(); Visited.init(); ReachedFinal.init()
4:   startState  $\leftarrow (v, q_0, 0, \perp)$ 
5:   Visited.push(startState); Open.push(startState)
6:   while Open  $\neq \emptyset$  do
7:     current  $\leftarrow$  Open.pop()  $\triangleright$  current =  $(n, q, \text{depth}, \text{prevList})$ 
8:     if  $q \in F$  then
9:       if  $n \notin$  ReachedFinal then
10:        ReachedFinal.add( $(n, \text{depth})$ )
11:        Solutions.add(current)
12:       else if ReachedFinal.get( $n$ ).depth = depth then
13:        Solutions.add(current)
14:       for each  $(q, a, q') \in \delta$  do
15:         for each  $e' \in E$  such that  $\rho_1(e') = n$  and  $\lambda(e') = \pi(a)$  do
16:            $n' = \rho_2(e')$ 
17:           if  $(n', q', *, *) \in$  Visited then
18:              $(n', q', \text{depth}', \text{prevList}') \leftarrow$  Visited.get( $n', q'$ )
19:             if depth + 1 = depth' then  $\triangleright$  New shortest path to  $(n', q')$ 
20:               if  $\pi(a) = a$  then
21:                 prevList'.add( $(\text{current}, e')$ )
22:               else
23:                 prevList'.add( $(\text{current}, z : e')$ )
24:             else
25:               prevList.init()
26:               if  $\pi(a) = a$  then
27:                 prevList.add( $(\text{current}, e')$ )
28:               else
29:                 prevList.add( $(\text{current}, z : e')$ )
30:               newState  $\leftarrow (n', q', \text{depth} + 1, \text{prevList})$ 
31:               Visited.push(newState);
32:               Open.push(newState)

```

---

This algorithm finds all the shortest walks and, like the first explores the product  $G \times A$  using either BFS (**Open** is a queue) or DFS (**Open** is a stack), starting from  $(v, q_0)$ . Again, it starts by initializing the data structures and setting up the start node in  $G \times A$  (lines 2–5). The main loop of line 6 is the same, pops an element  $(n, q, depth, prevList)$  from **Open** (line 7) and starts exploring its neighbors in  $G \times A$  (lines 14–16). When exploring  $(n, q, depth, prevList)$ , we scan all the transitions  $(q, a, q')$  or  $(q, a^z, q')$  of  $\mathcal{A}$  that originate from  $q$  (line 14), and look for neighbors of  $n$  in  $G$  reachable by an  $a$ -labeled edge (line 15).

If the pair  $(n', q')$  has not been visited yet, first we create the list  $prevList$ , then depending on the form of the transition  $(q, a, q')$  we add to  $prevList$  the pair  $(current, e')$ , or the pair  $(current, z : e')$  to mark the existence of a transition  $(q, \lambda(e)^z, q')$  (lines 24–29).

If the pair  $(n', q')$  has been visited, we call the state that had already been stored in Visited as  $(n', q', depth', prevList')$ , then if  $depth + 1 = depth'$  we find another new shortest path to  $(n', q')$  (lines 17–19). And the pair  $(current, e')$  or  $(current, z : e')$ , depending on the transition, is added to  $prevList'$  (lines 20–23)

Then a new state  $(n', q', depth + 1, prevList)$  is defined only when the pair  $(n', q')$  has not been visited yet, then we add it to **Visited** and **Open** (lines 30–32)

When popping from **Open** in line 7, we also check if  $q$  is a final state, and if  $n$  has not been reached by a solution path yet (lines 8–9). In this case we found a new solution; i.e., a WALK from  $v$  to  $n$  which matches  $A$ , so we add it to **Solutions** (line 11) and record it as reached with the length of the path  $(n, depth)$  (line 10).

If  $q \in F$  and  $n \in ReachedFinal$ , then we check if  $depth = ReachedFinal.get(n).depth$  i.e. if the new path has the same minimal length of the previous path that reached  $n$ . If the equality holds, then  $current$  is added to **Solutions**. (lines 12–13).

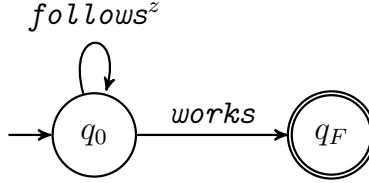
Let us now look at some examples of how this algorithm works.

**Example 9.** *Let us return to the social network graph  $G$  in Figure 1 and consider the following query*

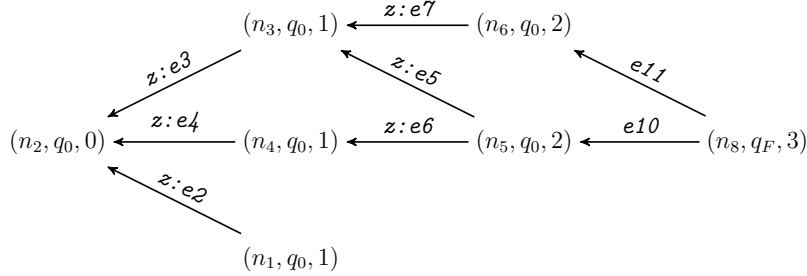
*ALL SHORTEST WALK (Joe, (follows<sup>z</sup>)<sup>\*</sup> · works, x).*

*As we can see, there are three shortest paths connecting Joe to ENS Paris*

matching the query. To compute these, Algorithm 7 first converts the expression  $\text{follows}^* \cdot \text{works}$  into the following automaton  $A$ :



Algorithm 7 then starts traversing the product graph  $G \times A$  from the node  $(n_2, q_0)$ . After executing the algorithm, the structure of **Visited** is as follows



Here we represent  $\text{prevList}$  as a series of arrows to other states in **Visited**, and only draw  $(n, q, \text{depth})$  in each node. For instance,  $(n_5, q_0, 2)$  has two outgoing edges, representing two pointers in its  $\text{prevList}$ . The arrow is also annotated with the edge witnessing the connection (as stored in  $\text{prevList}$ ).

To build this structure, Algorithm 7 explores the neighbors of  $(n_2, q_0)$ ; namely,  $(n_3, q_0)$ ,  $(n_4, q_0)$  and  $(n_1, q_0)$  and puts them to **Visited** and **Open**, with  $\text{depth} = 1$ . The algorithm proceeds by visiting  $(n_6, q_0)$  and  $(n_5, q_0)$  from  $(n_3, q_0)$ . The interesting thing happens in the next step when  $(n_4, q_0)$  is the node being expanded to its neighbour  $(n_5, q_0)$ , which is already present in **Visited**. Here we trigger lines 17–23 of the algorithm for the first time, and update the  $\text{prevList}$  for  $(n_5, q_0)$ , instead of ignoring this path. When we try to explore neighbors of  $(n_1, q_0)$ , we try to revisit  $(n_2, q_0)$ , so lines 17–23 are triggered again. This time the depth test in line 19 fails (we visited  $n_2$  with a length 0 path already), so this path is abandoned. We then explore the node  $(n_8, q_F)$  in  $G \times A$  by traversing the neighbors of  $(n_6, q_0)$ . Finally,  $(n_8, q_F)$  will be revisited as a neighbor of  $(n_5, q_0)$  on a previously unexplored shortest path.

Finally, we can then enumerate all the 3 pairs  $(p, \mu) \in \llbracket A \rrbracket_G$ , such that the path has a starting node  $n_2$ , by only following the arrows from  $(n_8, q_F)$ .

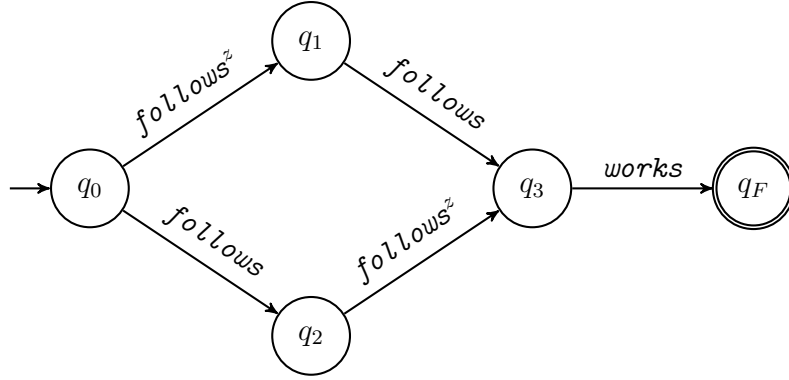
1.  $(p_1 = n_2e_3n_3e_7n_6e_{11}n_8, z \mapsto [e_3, e_7]) \in \llbracket A \rrbracket_G$
2.  $(p_2 = n_2e_3n_3e_5n_5e_{10}n_8, z \mapsto [e_3, e_5]) \in \llbracket A \rrbracket_G$
3.  $(p_3 = n_2e_4n_4e_6n_5e_{10}n_8, z \mapsto [e_4, e_6]) \in \llbracket A \rrbracket_G$

We can observe that there are only three different paths from node  $n_2$  to node  $n_8$  where each one is associated with its respective mapping.

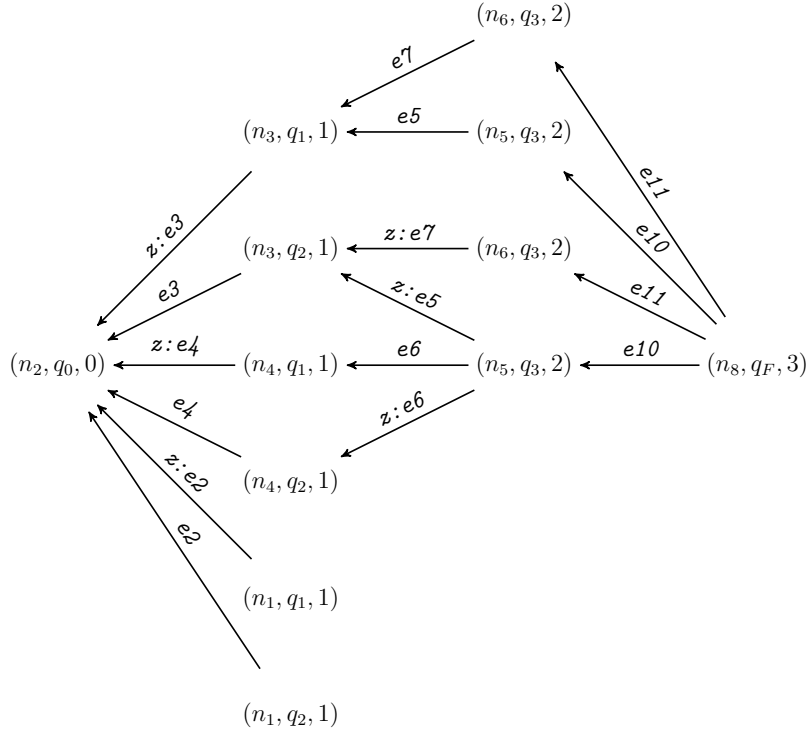
**Example 10.** Another example with the social network graph  $G$  in Figure 1 and the query

*ALL SHORTEST WALK* ( $\text{Joe}, (\text{follows}^z \cdot \text{follows} + \text{follows} \cdot \text{follows}^z) \cdot \text{works}, x$ ).

First converts the expression  $(\text{follows}^z \cdot \text{follows} + \text{follows} \cdot \text{follows}^z) \cdot \text{works}$  into the following automaton  $A$ :



Algorithm 7 then starts traversing the product graph  $G \times A$  from the node  $(n_2, q_0)$ . And after executing the algorithm, the structure of **Visited** is the following



This structure allows us to reconstruct and enumerate all the 6 pairs  $(p, \mu) \in \llbracket A \rrbracket_G$ , such that the path has a starting node  $n_2$ , by only following the arrows from  $(n_8, q_F)$ .

1.  $(p_1 = n_2 e_3 n_3 e_7 n_6 e_{11} n_8, z \mapsto [e_3]) \in \llbracket A \rrbracket_G$
2.  $(p_2 = n_2 e_3 n_3 e_5 n_5 e_{10} n_8, z \mapsto [e_3]) \in \llbracket A \rrbracket_G$
3.  $(p_1, z \mapsto [e_7]) \in \llbracket A \rrbracket_G$
4.  $(p_2, z \mapsto [e_5]) \in \llbracket A \rrbracket_G$
5.  $(p_3 = n_2 e_4 n_4 e_6 n_5 e_{10} n_8, z \mapsto [e_4]) \in \llbracket A \rrbracket_G$
6.  $(p_3, z \mapsto [e_6]) \in \llbracket A \rrbracket_G$

Note that the paths found are the same three unique paths found in the previous example; however, for each path there are two possible mappings, which results in a total of six possible pairs in the output of  $A$  over  $G$ .

### 6.3 TRAIL, SIMPLE and ACYCLIC

We now adapt algorithm for finding trails, simple paths, or acyclic paths from [4] to automaton with list variables.

We will be dealing with queries of the form:

$$q = \text{restrictor}(v, \text{regex}, ?x)$$

where `restrictor` is TRAIL, SIMPLE, or ACYCLIC. The Algorithm 8 shows how to evaluate such queries.

---

**Algorithm 8** Evaluation for  $query = \text{restrictor}(v, \text{regex}, x)$ .

---

```

1: function RESTRICTEDPATHS( $G, query$ )
2:    $\mathcal{A} \leftarrow \text{UnambiguousAutomaton}(regex)$   $\triangleright q_0$  initial,  $F$  final states
3:   Open.init(); Visited.init(); ReachedFinal.init()
4:    $startState \leftarrow (v, q_0, null, \perp)$ 
5:   Visited.push(startState); Open.push(startState)
6:   while Open  $\neq \emptyset$  do
7:      $current \leftarrow \text{Open.pop}()$   $\triangleright current = (n, q, e, prev)$ 
8:     if  $q \in F$  and  $n \notin \text{ReachedFinal}$  then
9:       ReachedFinal.add(n)
10:      Solutions.add(current)
11:      for each  $(q, \sigma, q') \in \delta$  do
12:        for each  $e' \in E$  such that  $\rho_1(e') = n$  and  $\lambda(e') = \pi(\sigma)$  do
13:           $n' \leftarrow \rho_2(e')$ 
14:          for next  $\leftarrow (n', q', e')$  do
15:            if ISVALID(current, next, restrictor) then
16:              if  $(n', q', *, *) \notin \text{Visited}$  then
17:                if  $\pi_\Sigma(\sigma) = \sigma$  then
18:                   $newState \leftarrow (n', q', e', current)$ 
19:                else
20:                   $newState \leftarrow (n', q', z : e', current)$ 
21:                Visited.push(newState)
22:                Open.push(newState)

```

---

Here, our *search state* is a tuple  $(n, q, e, prev)$ , where  $n$  is a node,  $q$  an automaton state,  $e$  an edge used to reach the node  $n$ , and  $prev$  a pointer to

another search state stored in `Visited`, which is a set storing already visited search states.

The function `ISVALID` (line 15) is the core component that enforces the restrictions. Since every search state contains a `prev` pointer to its predecessor, the algorithm can reconstruct the full path from the start node  $v$  to the current node  $n$ . Specifically, given a current state  $current = (n, q, e, prev)$  and a candidate next step  $next = (n', q', e')$ , the function performs the following checks by traversing the `prev` pointers backwards:

- If `restrictor` is `SIMPLE` (or `ACYCLIC`), `ISVALID` returns *true* if and only if the node  $n'$  does not appear in the sequence of nodes implicitly stored in the history of *current*. This prevents the path from visiting the same node twice.
- If `restrictor` is `TRAIL`, `ISVALID` returns *true* if and only if the edge  $e'$  does not appear in the sequence of edges stored in the history of *current*. This ensures that no edge is repeated, although nodes may be revisited.

It is important to note that the usage of the `Visited` set (line 16) in this context serves to prune the search space of the product graph  $G \times A$  to avoid redundant computations, while `ISVALID` handles the topological constraints of the path in  $G$ .

## 7 Conclusions

In this work, we have established a formal framework for automata with list variables, providing a robust model for analyzing Regular Path Queries with data extraction capabilities.

A major theoretical implication of our study concerns the nature of **determinism**. The demonstration that there exist automata for which no equivalent deterministic or I/O Unambiguous automaton exists implies that standard optimization techniques based on determinization cannot be directly applied to this model. However, our introduction of *Determinism\** provides a valuable canonical form, offering a partial solution that preserves the path structure while managing variable assignments via a powerset-like construction and this for trails is the same as I/O Unambiguity.

Regarding computational complexity, our results delineate a precise boundary for efficient query evaluation. We confirmed that the foundational problems of *Emptiness* and *Path Membership* are efficiently solvable in non-deterministic logarithmic space (**NL**). Crucially, however, we identified that the complexity of the *Mapping Membership* problem is sensitive to the number of variables used. The transition from **NL-complete** (for single variables) to **NP-complete** (for multiple variables) serves as a vital guardrail for query language designers: while the language allows for capturing multiple data lists simultaneously, doing so incurs a significant algorithmic cost that must be managed, potentially through the restriction to simple paths or trails as discussed.

Finally, the algorithmic solutions presented to find the shortest paths demonstrate that despite these theoretical hardness results, practical execution is feasible for optimization-focused queries. Future work may explore the integration of these automata into existing query engines and the definition of additional syntactic restrictions to ensure polynomial-time evaluation for the multi-variable case.

## References

- [1] Pablo Barceló Baeza. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*, pages 175–188, 2013. doi:10.1145/2463664.2465216.
- [2] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *International Conference on Management of Data (SIGMOD)*, pages 323–330, 1987. doi:10.1145/38713.38749.
- [3] Alin Deutsch, Nadime Francis, Alastair Green, Keith W. Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022. doi:10.1145/3514221.3526057.
- [4] Benjamín Farias, Wim Martens, Carlos Rojas, and Domagoj Vrgoc. Pathfinder: Returning paths in graph queries. In Gianluca Demartini, Katja Hose, Maribel Acosta, Matteo Palmonari, Gong Cheng, Hala Skaf-Molli, Nicolas Ferranti, Daniel Hernández, and Aidan Hogan, editors, *The Semantic Web - ISWC 2024 - 23rd International Semantic Web Conference, Baltimore, MD, USA, November 11-15, 2024, Proceedings, Part II*, volume 15232 of *Lecture Notes in Computer Science*, pages 135–154. Springer, 2024. doi:10.1007/978-3-031-77850-6\_8.
- [5] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. A researcher’s digest of GQL (invited talk). In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPICs*, pages 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPICs.ICDT.2023.1>, doi:10.4230/LIPICs.ICDT.2023.1.
- [6] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.