Study of New Semantics for Regular Path Queries

TRE Report*

Sara KHICHANE
Paris Cité University

Abstract

Graph database management systems have increased in popularity over the last decades. In database theory, we abstract such databases as labelled graphs. Most real query languages are based on the well-known formalism of regular path queries (RPQs). Such a query is defined by a regular expression R. Any walk in the graph labelled with a word conforming to R is called a match, and in general there are infinitely many matches. The main challenge is to efficiently compute a finite and meaningful subset of matches to present to a user. This selection process is referred to as the **RPQ semantics** under which the query is evaluated. Several approaches are used in practice and theory. In this article, we explore two new RPQ semantics, their properties and connections.

Keywords and phrases Graph database, regular path queries, query, RPQ semantics, walk, match.

1 Introduction

Graph databases have emerged as indispensable tools for modeling complex relationships and interconnected data structures [17]. They are found in various domains such as social networks, recommendation systems [12], and biology [20]. In order to extract information from graph databases, users often rely on querying mechanisms. Most query languages for graph databases include regular path queries (RPQ) [5]. An RPQ is defined by a regular expression R over an alphabet Σ. The query retrieves all "walks" (sequences of nodes and edges, also known as paths in the database community) in the graph that carry a label which conforms to R. Those walks are called the matches of the query. In the database theory community, an RPQ is typically evaluated under homomorphism-based semantics [1], where all pairs of the matches' endpoints are returned. This semantics is currently used in the SPARQL[1] query language. However, in many scenarios, one may need to retrieve the entire walks instead of the endpoints only. Moreover, the bag (or multiset) of matches is infinite (i.e. whenever the graph database contains a cycle).

In order to address these challenges, database management systems use different approaches to compute a finite and meaningful output from the matches. They consist in putting additional restrictions on the final bag of matches. In practice, two of the most common semantics are trail semantics and shortest-walk semantics. **Trail semantics** eliminates all walks with repeated edges. It is used by the Cypher[2] property graph query language [10], which was originally developed for the Neo4j graph database management system. **Shortest-walk semantics**, on the other hand, filters out results based on length. It is used by the query languages G-CORE [2] and GSQL [8] (developed for TigerGraph[3]). The International Organization for Standardization (ISO) has recently introduced a standard property graph query language called GQL [13], which permits mixing both semantics [11]. Both semantics have their own limitations as already noted in [7]. Trail semantics may discard valuable matches, leading to loss of important information. For instance in the

* Supervised by Amélie GHEERBRANT (Paris Cité University), Victor MARSAULT (Université Gustave Eiffel), and Antoine MEYER (Université Gustave Eiffel)
[1] https://www.w3.org/TR/sparql11-query/
[2] https://opencypher.org/
[3] https://www.tigergraph.com/gsql/

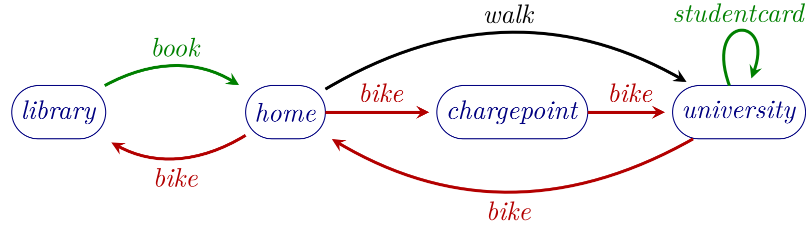


Figure 1 Graph Database.

graph database of Figure 1, the query $(bike^* \cdot studentcard \cdot bike^* \cdot book \cdot bike^*)$ from *home* to *university* would not return any result under trail semantics, as the only way to get the book while having a student card is to go through one and the same edge labelled by *bike* twice. Furthermore, deciding if there exists a walk from a source to a target under trail semantics is NP-complete. Mendelzon and Wood [15] studied the problem for simple walks, which are walks with no repeated vertices. They found that the problem is typically hard for the queries (a^*ba^*) and (a^*a^*). On the other hand, under shortest-walk semantics, the existence problem described above is NL-complete [15], but valuable matches may also be discarded. For instance, the query (Σ^*) from *home* to *university* returns the walk $(home \xrightarrow{walk} university)$ only. Not only the result doesn't give a general enough overview of the graph database, but a shortest walk is not always the best in practice. For instance going from *home* to *university* by *bike* might take less time than going by *walk*.

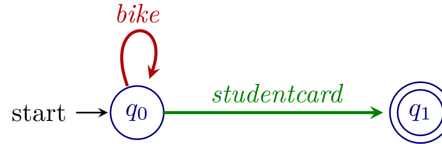


Figure 2 Automaton for Regular Expression: $(bike^* \cdot studentcard)$

These limitations show that there is room for new candidate semantics attempting to compute a concise yet expressive bag of matches. Two new semantics, that take into account the structure of the query, were recently introduced in [7]. Since we can represent an RPQ by an automaton or a regular expression, the authors define **simple-run semantics** where the query is given as a finite automaton and its counterpart **binding-trail semantics** operating on a regular expression instead. Both semantics eliminate walks that contain cycles in the database which coincide with the cycles of the automaton or the regular expression. These cycles appear in the product graph (defined in Section 2.1). For instance, the query in Figure 2 would return the walk $(home \xrightarrow{bike} chargepoint \xrightarrow{bike} university \xrightarrow{studentcard} university)$ under simple-run semantics, as the cycle $(university \xrightarrow{studentcard} university)$ doesn't coincide with a cycle in the automaton. These semantics give a representation of the graph database in some sense: they feature a sort of coverage (Lemma 13 in [7]). However, the problem of deciding if a given walk is returned by the semantics is NP-hard. Moreover, they depend on the structure of the automaton or the expression, which we believe might be an issue, as queries that represent the same regular language will sometimes yield different matches if represented by different automata or expressions.

In this report, we introduce and study two new semantics for RPQs: minimal-walk semantics and shortest-coverage semantics. **Minimal-walk semantics** is a new approach that filters

out all walks containing unnecessary cycles, where by unnecessary we mean that such cycles can be removed from the walk while still obtaining a match. **Shortest-coverage semantics**, on the other hand, expands shortest-walk semantics by considering the "coverage" of the returned set of results. There is no uniform definition of coverage in the literature, but for this semantics, we consider the vertex coverage of the database, such that for each vertex, all the shortest walks that matches the query and contain this vertex are returned. We will study the properties of these semantics and their connections to the existing ones.

Outline

This report first provides an overview of the concepts used throughout the paper, our chosen model of graph databases and regular path queries semantics in Section 2. Section 3 goes on to define minimal-walk semantics and shortest-coverage semantics, discuss their properties and compare them to existing semantics. Section 4 presents some of the computational problems most relevant to the study of RPQ semantics and the results we found for our semantics. Future work is discussed in Section 5 and Section 6 summarizes our results.

2 Preliminaries

In this section, we provide an overview of the concepts and notations used throughout the paper. We begin by presenting the definitions of automata and regular expressions. Afterwards, we define our model of a graph database and regular path queries, followed by an explanation of the semantics associated with regular path queries. Lastly, we define several enumeration complexity classes that will be referenced in the paper.

2.1 Graph Databases, Regular Path Queries and Matches

We start by giving the basic definition of a graph and define a breadth-first search that will be used in the rest of the paper.

▶ **Definition 1.** *A* ***graph*** *is a tuple* $G = (V, E)$ *where:*

- V *is a finite set of vertices.*
- E *is a finite set of edges between the vertices, such that:* $E \subseteq V \times V$

Given a graph $G = (V, E)$, one may use a breadth-first search algorithm (BFS) to find the shortest path between two vertices s and t in G [4]. In what follows, we use a slightly modified BFS algorithm that starts from several sources and ends at several targets, given in Algorithm 1.
The approach consists in enqueueing all of the sources instead of just one. The algorithm then visits all the sources then their neighbors, and so on. When a target is reached, the algorithm stops and returns the path from the target back to the source.

▶ Remark 2. In what follows, we use several variants of Algorithm 1.

▶ **Theorem 3.** *Algorithm 1 runs in time* $O(|V| + |E|)$*.*

Proof. It is known that the BFS algorithm runs in $O(|V| + |E|)$ time [4]. The initialization of the *visited* and *parent* arrays takes $O(|V|)$ time. The main loop iterates over all vertices and edges in the graph, similar to the classical BFS. Therefore, the total time complexity is $O(|V| + |E|)$. ◀

Algorithm 1 Multisource Multitarget BFS

```
Data: G = (V, E), sources, targets
Result: Shortest paths between sources and targets
1  visited[v ∈ V] ← 0; // array of boolean values indexed by the vertices
2  queue ← Empty Queue;
3  parent[v ∈ V] ← v; // array of predecessors indexed by the vertices
4  for s ∈ sources do
5      enqueue (queue, s);
6      visited[s] ← 1;
7  while queue ≠ ∅ do
8      v ← dequeue (queue);
9      if v ∈ targets then
           // Retrieving the walk from the current vertex back to its source
10         return GetWalk (v, parent);
11     for v′ ∈ neighbors(v) do
12         if v′ ∉ visited then
13             enqueue (queue, v′);
14             visited[v′] ← 1;
15             parent[v′] ← v;
```

In this report, we choose to model graph databases as directed, multi-labeled, multi-edge graphs, similar to [7].

▶ **Definition 4.** *A **database** D is a tuple $(\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ where:*

- *Σ is a finite set of symbols, or **labels**;*
- *V is a finite set of **vertices**;*
- *E is a finite set of **edges**;*
- *$\textsc{Src} : E \to V$ is the source function;*
- *$\textsc{Tgt} : E \to V$ is the target function;*
- *$\textsc{Lbl} : E \to 2^\Sigma$ is the labelling function.*

*We denote by $|D|$ the **size of** D and define it as $|V| + |E|$.*
*We say that a database is **single-labeled** if $\forall e \in E, |\textsc{Lbl}(e)| = 1$.*

We can adapt the MultiBFS algorithm (Algorithm 1) to find the shortest path between a set of sources and targets in a graph database $D = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$, by considering the neighbors of a vertex v as follows:

$$\text{neighbors}(v) = \{v' \mid \exists e \in E, \textsc{Src}(e) = v, \textsc{Tgt}(e) = v'\}$$

▶ **Definition 5.** *A **walk** w is a non-empty finite sequence of alternating vertices and edges starting and ending with a vertex, i.e. a sequence of the form $w = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ where $k \geq 0$, n_i is a vertex and e_i is an edge $\forall i$ where $0 \leq i < k$.*

▶ **Definition 6.** *A **walk** w **in** D is a walk, where $n_0, \ldots, n_k \in V$, $e_0, \ldots, e_{k-1} \in E$ and $\forall i, 0 \leq i < k, \textsc{Src}(e_i) = n_i$ and $\textsc{Tgt}(e_i) = n_{i+1}$.*
For ease of notation, we sometimes write $w = n_0 \xrightarrow{e_0} n_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{k-1}} n_k$. If D is single-edge, we sometimes use $\xrightarrow{\textsc{Lbl}(e_i)}$ instead of $\xrightarrow{e_i}$.

*We call k the **length of** w and denote it by $|w|$. We extend the functions* $\textsc{Src}$, $\textsc{Tgt}$ *and* $\textsc{Lbl}$ *to walks in D as follows. For each walk $w = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ in D:*

- $\textsc{Src}(w) = n_0$
- $\textsc{Tgt}(w) = n_k$
- $\textsc{EndPoints}(w) = (n_0, n_k) = (\textsc{Src}(w), \textsc{Tgt}(w))$
- $\textsc{Lbl}(w) = \{u_0u_1 \ldots u_{k-1} \mid u_0 \in \textsc{Lbl}(e_0), \ldots, u_{k-1} \in \textsc{Lbl}(e_{k-1})\}$

*Two walks $w = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ and $w' = (n'_0, e'_0, n'_1, \ldots, e'_{k-1}, n'_k)$ **concatenate** if $\textsc{Tgt}(w) = \textsc{Src}(w')$, in which case we define their concatenation as $(n_0, e_0, n_1, \ldots, e_{k-1}, n_k, e'_0, n'_1, \ldots, e'_{k-1}, n'_k)$ and denote it by $w \cdot w'$, or simply ww' for short.*

▶ Remark 7. We represent a walk $w = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ as two lists of vertices $V_w = (n_0, n_1, \ldots, n_k)$ and edges $E_w = (e_0, e_1, \ldots, e_{k-1})$. We can recover the original walk from these lists by pairing the vertices and edges in the order they appear. In the rest of the paper, to avoid confusion, we say "ith edge of w" to refer to the edge $E_w[i]$ and "ith vertex of w" to refer to the vertex $V_w[i]$.

We choose to model a regular path query as a non-deterministic finite automaton (NFA). We use the classical terminology for formal language theory.

▶ **Definition 8.** *An **automaton** is a 5-tuple $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$ such that:*

- *Σ is a finite set of symbols;*
- *Q is a finite set of states;*
- *$I \subseteq Q$ is the set of initial states;*
- *$F \subseteq Q$ is the set of final states;*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the set of transitions. As usual, we extend δ by adding the reflexive closure of δ : For every $q \in Q$, $(q, \varepsilon, q) \in \delta$. We use δ^* to denote the reflexive and transitive closure of δ : For every $q \in Q$, $(q, \varepsilon, q) \in \delta^*$. For every $q, q', q'' \in Q$, $a \in \Sigma, v \in \Sigma^*$, if $(q, a, q') \in \delta$ and $(q', v, q'') \in \delta^*$, then $(q, uv, q'') \in \delta^*$.*

*We denote by $|\mathcal{A}|$ the **size of** $\mathcal{A}$ and define it as $|Q| + |\delta|$.*
*We define the **language of** $\mathcal{A}$ as follows: $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q \in I, \exists q' \in F, (q, w, q') \in \delta^*\}$*

▶ **Definition 9.** *A **regular** expression R over an alphabet Σ is a formula obtained inductively from the letters in Σ, one unary function $*$ (Kleene closure), and two binary functions $+$ (union) and $\cdot$ (concatenation), according to the following grammar:*

$$R ::= \emptyset \mid \varepsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \text{ where } a \in \Sigma.$$

*The **language of** R is defined as usual.*

▶ **Definition 10.** *A **computation** in $\mathcal{A}$ is an alternating sequence of states and transitions that is defined similarly to walks in databases. We extend* $\textsc{Src}, \textsc{Lbl}, \textsc{Tgt}$ *and* $\textsc{EndPoints}$ *over computations. A computation is called **successful** if it starts in an initial state and ends in a final state.*

In the remainder of the paper, we fix a database $D = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ and an automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$.

▶ **Definition 11.** *The **product graph** $D \times \mathcal{A} = (\Sigma, V_{DA}, E_{DA}, \textsc{Src}_{DA}, \textsc{Tgt}_{DA}, \textsc{Lbl}_{DA})$ is the product of the database D and the automaton $\mathcal{A}$. It is defined as follows:*

- $V_{DA} = V \times Q$

- $E_{DA} = \{(e, (q, a, q')) \mid e \in E, (q, a, q') \in \delta, a \in \textsc{Lbl}(e)\}$
- *For each* $e_{DA} = (e, (q, a, q')) \in E_{DA}$, $\textsc{Src}_{DA}(e_{DA}) = (\textsc{Src}(e), q)$, $\textsc{Tgt}_{DA}(e_{DA}) = (\textsc{Tgt}(e), q')$ *and* $\textsc{Lbl}_{DA}(e_{DA}) = \{a\}$

▶ Remark 12. We note that the product graph is a single-labeled graph database.

Figure 5 illustrates the product graph $D_1 \times A_1$ where D_1 is the database in Figure 3 and $\mathcal{A}_1$ the query represented by the automaton in Figure 4.

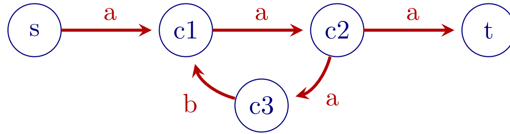


Figure 3 Graph Database D_1.

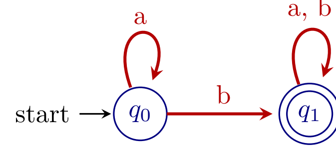


Figure 4 Automaton $\mathcal{A}_1$ for Regular Expression: $a^*b(a+b)^*$.

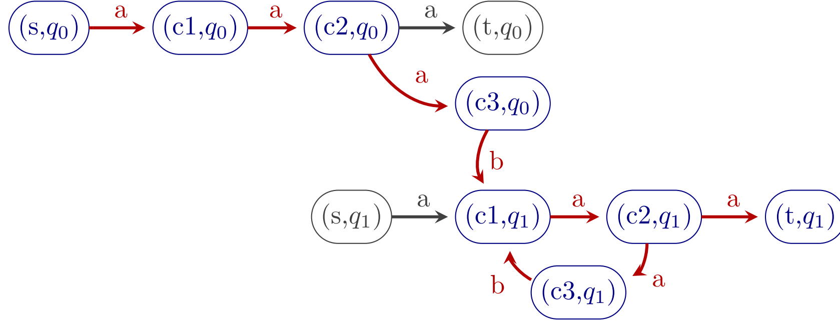


Figure 5 Product graph of the graph database in Figure 3 and the automaton in Figure 4.

▶ **Definition 13.** *A* ***run*** r *is a walk in* $D \times \mathcal{A}$ *such that* $\textsc{Src}(w) \in V \times I$ *and* $\textsc{Tgt}(w) \in V \times F$. *We denote the set of all runs in* $D \times \mathcal{A}$ *by* $\textsc{Run}_{\mathcal{A}}(D)$.

One may recover the original database from a product graph using the following projection operation.

▶ **Definition 14.** *We denote by* π_D *the projection function that maps vertices and edges from the product graph* $D \times \mathcal{A}$ *back to those of the original database* D. *It is defined as follows:*

- $\pi_D(v, q) = v$, *for each* $(v, q) \in V_G$
- $\pi_D(e, (q, a, q')) = e$, *for each* $(e, (q, a, q')) \in E_G$

The function is extended to runs: $\pi_D(r) = (\pi_D(n_0), \pi_D(e_0), \ldots, \pi_D(e_{k-1}), \pi_D(n_k))$ *for each run* $r = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ *in* $D \times \mathcal{A}$.

▶ **Definition 15.** $\textsc{Match}_{\mathcal{A}}$ *is the function that maps a database* D *to the bag of walks matching the query specified by automaton* $\mathcal{A}$. *It is defined as follows:*

$$\textsc{Match}_{\mathcal{A}}(D) = \{\!\{\pi_D(R) \mid R \in \textsc{Run}_{\mathcal{A}}(D)\}\!\}$$

▶ Remark 16. We use $\{\!\{\}\!\}$ to denote a bag (or multiset, or mset) to avoid confusion with the usual set notation $\{\}$. $\textsc{Match}_{\mathcal{A}}(D)$ is indeed a bag and not a set, because there might exist several distinct runs that correspond to the same walk in the original database.

We use the same notation to denote the set of matches between two specific vertices of a database:

▶ **Definition 17.** $\textsc{Match}_{\mathcal{A}}$ *is the function that maps a database D and two endpoints $s, t \in V$ to the bag of walks from s to t that match the query specified by automaton $\mathcal{A}$ on D. It is defined as follows:*

$$\textsc{Match}_{\mathcal{A}}(D, s, t) = \{\!\{ w \mid w \in \textsc{Match}_{\mathcal{A}}(D), \textsc{EndPoints}(w) = (s, t) \}\!\}$$

In the rest of the paper, we use Algorithm 2 to determine if a walk w of length n is in $\textsc{Match}_{\mathcal{A}}(D)$. The approach is based on the polynomial NFA acceptance algorithm [18].

Algorithm 2 Match Algorithm

```
Data: w, D, A
Result: w ∈ Match_A(D)
1 for i ← 0 to |w| do
2     E_i ← ∅;
3 E_0 ← I;
4 for i ← 1 to |w| do
5     e_i ← ith edge of w;
6     E_i ← {q' | ∃q ∈ E_{i−1}, a ∈ Lbl(e_i) such that(e_i, (q, a, q')) ∈ E_DA};
7 return E_|w| ∩ F ≠ ∅;
```

▶ **Theorem 18.** *Algorithm 2 correctly determines if a walk w is in $\textsc{Match}_{\mathcal{A}}(D)$.*

Proof. Let us show by induction on i that E_i is the set of states of $\mathcal{A}$ that can be reached from some initial state in I by reading a sequence of letters labeling the first i edges of w.

- Base case: $i = 0$. $E_0 = I$.
- Inductive step: suppose the property holds for E_{i-1}. For every state $q' \in E_i$, there exists $q \in E_{i-1}$ and $a \in \textsc{Lbl}(e_i)$ such that $(e_i, (q, a, q')) \in E_{DA}$. Conversely, E_i contains only such states. Therefore, E_i is the set of states that can be reached from the initial states by reading the first i edges of the walk.

In particular, $E_{|w|}$ is the set of states that can be reached from the initial states by reading all the edges of the walk. By Definition 15, a walk w of length n is in $\textsc{Match}_{\mathcal{A}}(D)$ if and only if $\exists r \in \textsc{Run}_{\mathcal{A}}(D)$ such that $\textsc{Src}(r) \in \textsc{Src}(w) \times I$ and $\textsc{Tgt}(r) \in \textsc{Tgt}(w) \times F$. This is equivalent to $E_{|w|} \cap F \neq \emptyset$. Thus, the algorithm correctly determines if w is in $\textsc{Match}_{\mathcal{A}}(D)$. ◀

▶ **Theorem 19.** *Algorithm 2 runs in time $O(|w| \cdot |\mathcal{A}|)$.*

Proof. Each set E_i can be represented by its characteristic function $f_i : \{0, 1, \ldots, |Q| - 1\} \to \{0, 1\}$, where each $q \in Q$ is represented by an integer e in $\mathbb{N}$, such that $f_i(e) = 1$ if $q \in E_i$, else $f_i(e) = 0$. Thus, we can implement this set in the algorithm as an array of booleans of size $|Q_A|$. The index of the list represents the state q and the value at that index represents $f_i(q)$. The initialization of one set E_i takes $O(|\mathcal{A}|)$ time and adding a state to the set takes constant time, as it is simply changing a value at an index in the array. Therefore, we have:

- The initialization of the $|w|$ sets E_i $(1 \leq i \leq n)$ takes time $O(|w| \cdot |\mathcal{A}|)$.
- The computation of one E_i sets takes $O(|\mathcal{A}|)$ time, as it iterates over all states in E_{i-1} and their outgoing transitions (for each label a in e_i) in $\mathcal{A}$.

Thus, the algorithm runs in $O(|w| \cdot |\mathcal{A}|)$. ◀

2.2 Regular Path Queries Semantics

Since the $\textsc{Match}_{\mathcal{A}}$ function may return an infinite bag of walks, we introduce semantics as a way to map the results to a finite bag, based on a specific filtering or ordering approach. In this section, we use the notation introduced in [7] to denote a semantics.
In what follows, we define several existing semantics for regular path queries.

▶ **Definition 20.** *The* ***shortest-walk semantics*** *of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{SW}$, is a mapping that associates, for each database D, a bag of walks. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{SW}(D) = \bigcup_{(s,t)\in V^2} \{\!\{w \in \textsc{Match}_{\mathcal{A}}(D,s,t) \mid \forall w' \in \textsc{Match}_{\mathcal{A}}(D,s,t), |w| \leq |w'|\}\!\}$$

▶ **Definition 21.** *The* ***trail semantics*** *of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{T}$, is a mapping that associates, for each database D, a bag of walks. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{T}(D) = \{\!\{w \in \textsc{Match}_{\mathcal{A}}(D) \mid \forall i,j, (i \neq j \implies E_w[i] \neq E_w[i])\}\!\}$$

▶ **Definition 22.** *The* ***acyclic-walk semantics*** *(also known as simple-path or simple-walk) of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{AW}$, is a mapping that associates, for each database D, a bag of walks. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{AW}(D) = \{\!\{w \in \textsc{Match}_{\mathcal{A}}(D) \mid \forall i,j, (i \neq j \implies V_w[i] \neq V_w[i])\}\!\}$$

▶ **Definition 23.** *The* ***simple-run semantics*** *of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{SR}$, is a mapping that associates, for each database D, a bag of walks. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{SR}(D) = \{\!\{\pi_D(r) \mid r \in \textsc{Run}_{\mathcal{A}}(D), \forall i,j, (i \neq j \implies V_r[i] \neq V_r[i])\}\!\}$$

2.3 Enumeration Complexity

For enumeration problems, classical complexity is not informative enough, since the size of the output, i.e. the enumerated set of results, is almost always exponentially bigger than the size of the input. Thus, in order to make complexity of enumeration algorithms relevant, more parameters are needed and so new classes are defined. In what follows, we recall some of the enumeration classes defined in [3] and [19].
Let $A \subseteq \Sigma^* \times \Sigma^*$ be a binary predicate. We write $A(x)$ for the set of y such that $A(x, y)$ holds. The **enumeration problem** P_A is the function which associates $A(x)$ to x.
We denote by $T(x, i)$ the number of elementary steps taken by an algorithm M on the input x up to when the ith output is produced.

▶ **Definition 24.** *A problem P_A is in* ***EnumP*** *if deciding whether $y \in A(x)$ is in P and all $y \in A(x)$ are of polynomial size in $|x|$.*

All enumeration classes defined below are in the class EnumP. The problems in EnumP can be seen as the task of listing the solutions (or witnesses) of NP problems.

▶ **Definition 25.** *A problem $P_A \in EnumP$ is in* ***OutputP*** *if there is a polynomial p and an algorithm which solves it in total time $O(p(|x|, |A(x)|))$.*

An algorithm enumerates $A(x)$ in preprocessing time $h(|x|)$ and incremental time $f(k)g(|x|)$ if on every input x:

- $T(x, 1) \leq h(|x|)$.

- $T(x,k) - T(x,1) \leq f(k) \cdot g(|x|)$ for every $k \leq |A(x)|$.

▶ **Definition 26.** *A problem $P_A \in EnumP$ is in **IncP$_a$** if there is a polynomial p and an algorithm which solves it on input x in preprocessing time $O(p(|x|))$ and incremental time $O(k^a \cdot |x|^b)$, for a, b constants.*
$IncP$ is the union of all $IncP_a$ for $a \geq 1$. We write $IncP = \bigcup_{a \geq 1} IncP_a$.

▶ **Definition 27.** *The **delay** of an algorithm on input x is the maximum delay between two solutions it produces, that is: $\max_{1 \leq k \leq |A(x)|} |T(x, k+1) - T(x,k)|$.*
*A problem $P_A \in EnumP$ is in **DelayP** if there is a polynomial p and an algorithm which solves it on input x with delay $O(p(|x|))$.*

A memoryless enumeration algorithm is an algorithm that can only use the last solution and the input to compute the next solution. If an algorithm is memoryless and has a polynomial delay, then it is in NextP, which is defined as follows:

▶ **Definition 28.** *A problem $P_A \in EnumP$ is in **NextP** if for every instance x there is a total order $<_x$ on $A(x)$ such that the following problems can be solved in polynomial time:*

- *Given x, output the first element of $A(x)$ for $<_x$.*
- *Given x and $y \in A(x)$, output the next element of $A(x)$ for $<_x$ or a special value if there is none.*

3 Minimal-walk and Shortest-coverage Semantics

3.1 Minimal-walk Semantics

Minimal-walk semantics offers a nuanced approach to filtering results in RPQ queries, ensuring finite results while providing some sort of coverage. Unlike acyclic-walk semantics, minimal-walk semantics does not eliminate all walks that contain cycles. Nor does it eliminate all walks that are the projection of runs containing cycles, like simple-run semantics. Instead, it eliminates a walk if removing one or more cycles from it results in a walk that is still a match. Another way to see it is that in an ordered family of walks that are all matches, where each walk is a subsequence of the following walk in the ordered family, minimal-walk semantics will keep the shortest one.
It is important to note that it is independent from the chosen structure of the query: given two queries $\mathcal{A}_1, \mathcal{A}_2$, if $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$, then the queries return the same bag of results under this semantics.

▶ **Definition 29.** *Let w and w' be two walks. We write $w' \lll w$ if $\exists w_0$, w_1 and w_2 walks such that : $w = w_0 \cdot w_1 \cdot w_2$, $\textsc{Tgt}(w_0) = \textsc{Src}(w_2)$ and $w' = w_0 \cdot w_2$.*

▶ **Definition 30.** *A walk w' is a **subwalk** of a walk w if $w' \lll^* w$, where $\lll^*$ is a reflexive transitive closure of $\lll$.*

We notice that notion of subwalk can be defined equivalently as follows.

▶ **Proposition 31.** *Let w be a walk in D, w' is a subwalk of w if it satisfies the following conditions with regard to w:*

1. *w' is a subsequence of w, i.e. w' is obtained by deleting some or no elements from w without changing the order of the remaining elements.*
2. *w' is a walk in D.*

3. $\textsc{Src}(w) = \textsc{Src}(w')$
4. $\textsc{Tgt}(w) = \textsc{Tgt}(w')$

Proof. Let w be a walk in D and $k = |w|$.

- $\Longrightarrow$: w' is subwalk of $w \Longrightarrow w'$ satisfies the conditions of Proposition 31.
  We prove this by induction on n such that $w' \ll^n w$.
  - Base case: $n = 0$. In this case, $w' = w$. Therefore, w' satisfies the conditions of Proposition 31 with regard to w.
  - Inductive step: $n > 0$. $w' \ll^n w$ means that $\exists w''$ a walk such that $w'' \ll^{n-1} w$ and $w' \ll w''$. By definition, $\exists w_0, w_1, w_2$ walks such that $w'' = w_0 \cdot w_1 \cdot w_2$ and $w' = w_0 \cdot w_2$. This implies that w' is obtained by removing zero or more elements from w'', establishing w' as a subsequence of w''. By induction hypothesis, w'' satisfies the conditions 1, 2, 3 and 4 with regard to w. We have $\textsc{Src}(w') = \textsc{Src}(w_0) = \textsc{Src}(w'') = \textsc{Src}(w)$ and $\textsc{Tgt}(w') = \textsc{Tgt}(w_2) = \textsc{Tgt}(w'') = \textsc{Tgt}(w)$ which satisfies the conditions 3 and 4. Moreover, w' is obtained by removing some or no elements from w'' without altering the order of the remaining elements and w'' is a subsequence of w, which makes w' a subsequence of w (condition 1). Finally, as w' results from the concatenation of two walks, it is itself a walk in D (condition 2).
- $\Longleftarrow$: w' satisfies the conditions of proposition 31 $\Longrightarrow w'$ is subwalk of w.
  w' is derived by removing some or no elements from w without altering the order of the remaining elements, ensuring that w' remains a walk in D with the same source and target nodes as w. If no elements are removed, then $w' = w$, making w' a subwalk of w. If at least one element is removed, we consider the removal of a maximal continuous sequence of elements resulting in the walk w''. If the removed sequence contains $\textsc{Src}(w)$, then the last element removed must be an edge e such that $\textsc{Tgt}(e) = \textsc{Src}(w)$. This comes from condition 3 that states that $\textsc{Src}(w) = \textsc{Src}(w')$. Thus, w' is a subwalk of w. We can easily deduce the same for when the last element is $\textsc{Tgt}(w)$, using condition 4. Otherwise, let a be the first element of the removed sequence, and let b be the element following the last removed element in the sequence. We distinguish 4 cases:

  1. a and b are two nodes: We denote by e the edge that precedes the removed sequence. Since w is a walk in D, we have $\textsc{Tgt}(e) = a$. Moreover, since w'' is a walk in D then we also have $\textsc{Tgt}(e) = b$. Hence, $a = b$. Therefore, $\exists w_0, w_1, w_2$ walks such that $w = w_0 \cdot w_1 \cdot w_2$, $w'' = w_0 \cdot w_2$ and $\textsc{Tgt}(w_0) = \textsc{Src}(w_2) = a$. Thus, $w'' \ll w$.
  2. a is a node and b an edge: Since w is an alternating sequence of nodes and edges, w'' will contain two consecutive nodes. Therefore, w'' is not a walk.
  3. a is an edge and b a node: Using the same reasoning as before, w'' is not a walk as it will contains two consecutive edges.
  4. a and b are two edges: In this case, we consider c the element preceding the removed sequence and d the last element of it. We use a reasoning symmetric to case 1 to deduce that $c = d$ and $w'' \ll w$.

  Furthermore, w' is obtained by the removal of several continuous sequences. Thus, $w' \ll^* w$ and w' is subwalk of w.

◀

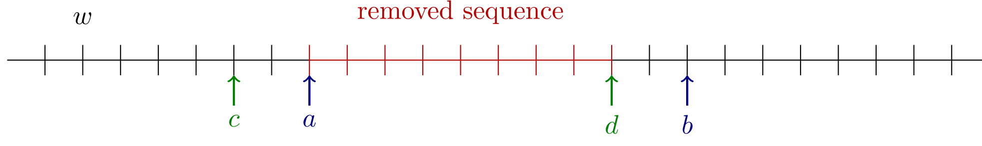


Figure 6 Illustration to help proof of Proposition 31.

▶ **Definition 32.** *We denote by* $\sqsubseteq$ *the order relation defined as follows:*

$$w \sqsubseteq w' \iff w \textit{ is a subwalk of } w'$$

We denote by $\sqsubset$ *the associated strict order.*

▶ **Proposition 33.** *The relation* $\sqsubseteq$ *induced by the subwalk relation is a partial order.*

Proof. Let S be a bag of walks. The relation $\sqsubseteq$ satisfies both the reflexivity and transitivity properties in S as it is defined as a reflexive and transitive closure of $\ll$.
For the antisymmetry, consider w and w' walks such that $w \sqsubseteq w'$ and $w' \sqsubseteq w$. We can distinguish two cases. Either we remove zero elements from w to obtain w' or zero elements from w' to obtain w. In this case, $w = w'$. Alternatively, we remove at least one element from w to obtain w', and at least one element from w' to obtain w. In this case, we have $|w| < |w'|$ and $|w'| < |w|$, which is a contradiction. Therefore, $w = w'$ and thus it satisfies the antisymmetry property (i.e. $w \sqsubseteq w'$ and $w' \sqsubseteq w$ if and only if $\forall w, w' \in S, w = w'$).
Therefore, the relation $\sqsubseteq$ is a partial order. ◀

▶ Remark 34. In most cases, the partial order induced by the subwalk relation is not total i.e. $\exists w, w' \in S$ such that $w \not\sqsubseteq w'$ and $w' \not\sqsubseteq w$. Take for instance, $w = (s \xrightarrow{a} t)$ and $w' = (s \xrightarrow{b} t)$.

▶ **Lemma 35.** *The relation* $\sqsubset$ *is a well-founded order on any bag of walks.*

Proof. Assume the relation $\sqsubset$ is not a well-founded order on a bag of walks S. Therefore, $\exists w \in S$ with an infinite descending chain of strict subwalks, which results in removing an infinite number of elements from w. This contradicts the fact that a walk is a finite sequence of nodes and edges. ◀

▶ **Definition 36.** *We say a walk* w *is* ***minimal*** *in a bag* S *of walks if it is minimal with respect to the* $\sqsubseteq$ *partial order i.e.* $\nexists w' \in S$ *such that* $w' \sqsubset w$*.*

Let $S = \textsc{Match}_{\mathcal{A}_1}(D_1)$ where D_1 is the graph database in Figure 3 and $\mathcal{A}_1$ the query represented by the automaton in Figure 4.

- The walk $w_1 = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c3 \xrightarrow{b} c1 \xrightarrow{a} c2 \xrightarrow{a} t)$ is a minimal walk in S since there is no subwalk of w that is also a match for the query in $\mathcal{A}$.
- The walk $w_2 = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c3 \xrightarrow{b} c1 \xrightarrow{a} c2 \xrightarrow{a} c3 \xrightarrow{b} c1 \xrightarrow{a} c2 \xrightarrow{a} t)$ is not minimal in S since $w_1 \in S$ and $w_1 \sqsubset w_2$.

▶ **Lemma 37.** *Every non-minimal walk in a bag of walks contains a cycle.*

Proof. Let w be a non-minimal walk in a bag S. Then, $\exists w' \in S$ such that $w' \sqsubset w$. By Definition 30, $w' \ll^* w$ and $w' \neq w$. We consider one $\ll$ operation that removes a non-empty sequence of elements from w (there exists at least one such operation as $w' \neq w$). Therefore, $\exists w''$ such that $w'' \ll w$ and $w'' \neq w$. By Definition 29, $\exists w_0, w_1, w_2$ such that $w = w_0 \cdot w_1 \cdot w_2$ and $w'' = w_0 \cdot w_2$. We have $\textsc{Tgt}(w_0) = \textsc{Src}(w_2)$. Moreover since $w'' \neq w$, w_1 is not empty (i.e. $|w_1| \neq 0$). Thus, w contains a cycle. ◀

▶ **Definition 38.** *The **minimal-walk semantics** of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{MW}$, is a mapping that associates, for each database D, the bag of minimal walks in D. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{MW}(D) = \{\!\{w \in \text{MATCH}_{\mathcal{A}}(D) \mid w \text{ is minimal in } \text{MATCH}_{\mathcal{A}}(D)\}\!\}$$

▶ **Lemma 39.** *Let r, r' be two runs in $D \times \mathcal{A}$.*

$$r \sqsubset r' \implies \pi_D(r) \sqsubset \pi_D(r')$$

Proof. First, let us show $r' \ll r \implies \pi_D(r) \ll \pi_D(r')$. We denote by r_0, r_1, r_2 the runs such that $r = r_0 \cdot r_1 \cdot r_2$ and $r'' = r_0 \cdot r_2$. We have $\pi_D(r) = \pi_D(r_0) \cdot \pi_D(r_1) \cdot \pi_D(r_2)$ and $\pi_D(r'') = \pi_D(r_0) \cdot \pi_D(r_2)$. Therefore, $\pi_D(r'') \ll \pi_D(r)$. Second, iterating this reasoning yields that $r' \ll^* r \implies \pi_D(r) \ll^* \pi_D(r')$. Finally, let us now assume $r \sqsubset r'$, which implies that $|r'| < |r|$ and $\pi_D(r) \ll^* \pi_D(r')$. The projection function preserves the length of the runs, so $|\pi_D(r')| < |\pi_D(r)|$. Therefore, $\pi_D(r') \sqsubset \pi_D(r)$. ◀

▶ **Lemma 40.** *For any graph database D and any query $\mathcal{A}$, $[\![\mathcal{A}]\!]_{MW}(D)$ is a finite bag.*

Proof. For every run r in the product graph $D \times \mathcal{A}$, if $|r| > (|D| \times |\mathcal{A}|)$, then r contains a cycle. We denote by r' the run obtained by removing the cycle and note that $r' \sqsubset r$. So, by Lemma 39, $\pi_D(r') \sqsubset \pi_D(r)$, which means that $\pi_D(r)$ is not a minimal walk. We deduce that:

$$[\![\mathcal{A}]\!]_{MW}(D) \subseteq \{\!\{\pi_D(R) \mid R \in \text{RUN}_{\mathcal{A}}(D), |r| \leq |\mathcal{A}| \times |D|\}\!\}.$$

Thus, $[\![\mathcal{A}]\!]_{MW}(D)$ is a finite bag. ◀

3.2 Shortest-coverage Semantics

Shortest-walk semantics filters matches by focusing solely on their length. It fails to consider the coverage of the vertices in the graph database. In contrast, shortest-coverage semantics ensures that the results are the shortest possible while also covering all the vertices in the graph database. Essentially, it applies the concept of shortest-walk semantics for each vertex. This ensures that final results not only minimize length but also effectively traverse and cover all vertices.

▶ **Definition 41.** *Let S be a bag of walks, v a node and w a walk in S. w is a **shortest walk covering** v **in** S if v occurs in w and any walk w' where v occurs satisfies $|w| \leq |w'|$.*

We use the graph database D_1 in Figure 3 and the query $\mathcal{A}_2$ represented by the regular expression $(a+b)^*$ to illustrate the definition of shortest coverage with respect to each vertex in the graph database. Let $S = \text{MATCH}_{\mathcal{A}_2}(D_1, s, t)$.

- The bag of the shortest walks covering $c1$ in S is $\{\!\{(s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} t)\}\!\}$.
- The bag of the shortest walks covering $c2$ in S is $\{\!\{(s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} t)\}\!\}$.
- The bag of the shortest walks covering $c3$ in S is $\{\!\{(s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c3 \xrightarrow{b} c1 \xrightarrow{a} c2 \xrightarrow{a} t)\}\!\}$

▶ **Definition 42.** *The **shortest-coverage semantics** of a query automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{SC}(D)$, is a mapping that associates, for each graph database D, a bag of walks. It is defined as follows:*

$$[\![\mathcal{A}]\!]_{SC}(D) = \bigcup_{(s,t,v)\in V^3} \{\!\{w \in \text{MATCH}_{\mathcal{A}}(D, s, t) \mid w \text{ is a shortest walk covering } v \text{ in } \text{MATCH}_{\mathcal{A}}(D, s, t)\}\!\}$$

3.3 Comparison of Semantics with respect to Inclusion

▶ **Definition 43.** *We say that a semantics X is* ***included in*** *a semantics Y and write $X \subseteq Y$ if :*

$$\forall D, \forall \mathcal{A}, \quad [\![\mathcal{A}]\!]_X(D) \subseteq [\![\mathcal{A}]\!]_Y(D)$$

We say that X is strictly included in Y and write $X \subsetneq Y$ if $X \subseteq Y$ and $Y \nsubseteq X$.
We say that X and Y are incomparable if $X \nsubseteq Y$ and $Y \nsubseteq X$.

In order to situate our semantics in the broader context of existing semantics, we compare them with the shortest-walk semantics, acyclic-walk semantics, simple-run semantics, and trail semantics, with respect to inclusion. We summarize in Figure 7 the results demonstrated later in this section.

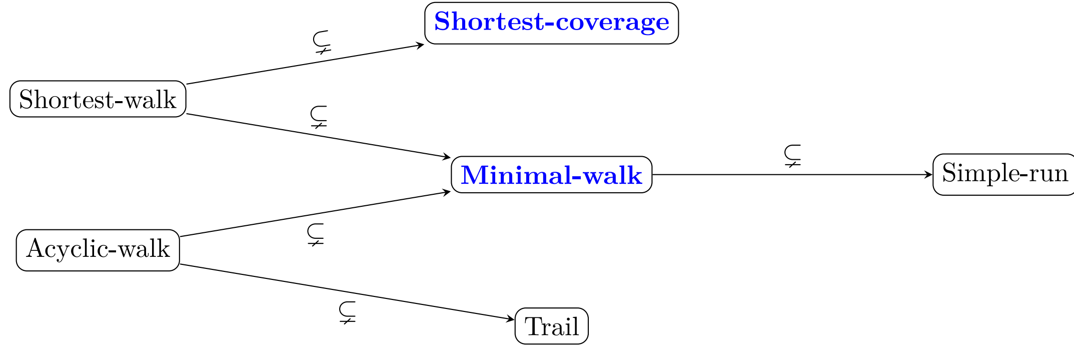


Figure 7 Recap of the inclusions demonstrated below.

Ex.	D	$\mathcal{A}$	w	MW	SC	SW	T	AW	SR
1	Fig.3	$(a+b)^*$	$w_1 = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c3 \xrightarrow{b} c1 \xrightarrow{a} c2 \xrightarrow{a} t)$	✗	✓	✗	✗	✗	✗
2	Fig.3	Fig.4	w_1	✓	-	-	✗	✗	-
3	Fig.8	$(a+b+c)^*$	$w_2 = (s \xrightarrow{a} c1 \xrightarrow{c} c2 \xrightarrow{c} t)$	✓	✗	✗	✓	✓	✓
4	Fig.9	a^*	$w_3 = (s \xrightarrow{a} c1 \xrightarrow{a} c1 \xrightarrow{a} t)$	✗	✗	-	✓	-	-
5	Fig.10	Fig.11	$w_4 = (s \xrightarrow{a} c1 \xrightarrow{b} c1 \xrightarrow{b} t)$	✗	-	-	-	-	✓

Table 1 Examples to illustrate the non-inclusions between semantics.

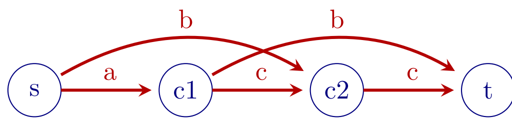


Figure 8 Graph Database D_2.

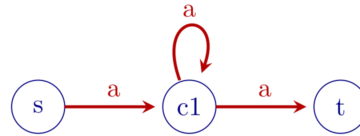


Figure 9 Graph Database D_3.

▶ **Proposition 44.** *Shortest-walk semantics is strictly included in minimal-walk semantics.*

Proof. $SW \subseteq MW$: Let $w \in [\![\mathcal{A}]\!]_{SW}(D)$, we fix a source s and a target t such that $\textsc{Src}(w) = s$ and $\textsc{Tgt}(w) = t$. Assume $w \notin [\![\mathcal{A}]\!]_{MW}(D)$, then $\exists w' \in [\![\mathcal{A}]\!]_{MW}(D)$ such that $w' \sqsubset w$ (Lemma 35), which implies that $|w'| \leq |w|$. This contradicts the fact that w is a shortest walk from s to t in D that matches the query specified by $\mathcal{A}$. Hence, $w \in [\![\mathcal{A}]\!]_{MW}(D)$.
$MW \nsubseteq SW$: See Example 3 in Table 1. $w_2 \notin [\![\mathcal{A}]\!]_{SW}(D)$ as we have another match $w' = (s \xrightarrow{a} c1 \xrightarrow{b} t)$ such that $|w'| < |w_2|$. ◀

▶ **Proposition 45.** *Shortest-walk semantics is strictly included in shortest-coverage semantics.*

Proof. $SW \subseteq SC$: Let $w \in [\![\mathcal{A}]\!]_{SW}(D)$, then w is a shortest walk in D that matches the query specified by $\mathcal{A}$. By definition, w is a shortest walk that covers all its vertices. Therefore, $w \in [\![\mathcal{A}]\!]_{SC}(D)$.
$SC \not\subseteq SW$: See Example 2 in Table 1. $w_1 \notin [\![\mathcal{A}]\!]_{SW}(D)$ as we have another match $w' = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} t)$ such that $|w'| < |w_1|$. ◀

▶ **Proposition 46.** *Minimal-walk semantics and trail semantics are incomparable.*

Proof. $MW \not\subseteq T$: See Example 2 in Table 1. $w_1 \notin [\![\mathcal{A}]\!]_T(D)$ since the edge $c1 \xrightarrow{a} c2$ is repeated twice in w_1.
$T \not\subseteq MW$: See Example 4 in Table 1. $w_3 \notin [\![\mathcal{A}]\!]_{MW}(D)$ as we have another match $w' = (s \xrightarrow{a} c1 \xrightarrow{a} t)$ such that $w' \sqsubset w_3$. ◀

▶ **Proposition 47.** *Shortest-coverage semantics and trail semantics are incomparable.*

Proof. $SC \not\subseteq T$: See Example 1 in Table 1. $w_1 \notin [\![\mathcal{A}]\!]_T(D)$ since the edge $c1 \xrightarrow{a} c2$ is repeated twice in w_1.
$T \not\subseteq SC$: See Example 4 in Table 1. $w_3 \notin [\![\mathcal{A}]\!]_{SC}(D)$ as it isn't a shortest walk covering any of its vertices. ◀

▶ **Proposition 48.** *Acyclic-walk semantics is strictly included in minimal-walk semantics.*

Proof. $AW \subseteq MW$: Let $w \in [\![\mathcal{A}]\!]_{AW}(D)$, then w is an acyclic walk in D that matches the query specified by $\mathcal{A}$. Assume $w \notin [\![\mathcal{A}]\!]_{MW}(D)$, then $\exists w' \in [\![\mathcal{A}]\!]_{MW}(D)$ such that $w' \sqsubset w$ (Lemma 35). Therefore, w contains a cycle (Lemma 37). This contradicts the fact that w is an acyclic walk. Hence, $w \in [\![\mathcal{A}]\!]_{MW}(D)$.
$MW \not\subseteq AW$: See Example 4 in Table 1. $w_3 \notin [\![\mathcal{A}]\!]_{AW}(D)$ since the vertices $c1$ and $c2$ are repeated twice. ◀

▶ **Proposition 49.** *Shortest-coverage semantics and acyclic-walk semantics are incomparable.*

Proof. $SC \not\subseteq AW$: See Example 1 in Table 1. $w_1 \notin [\![\mathcal{A}]\!]_{AW}(D)$ since the vertices $c1$ and $c2$ are repeated twice.
$AW \not\subseteq SC$: See Example 3 in Table 1. $w_2 \notin [\![\mathcal{A}]\!]_{SC}(D)$ as it isn't a shortest walk covering any of its vertices. ◀

▶ **Proposition 50.** *Minimal-walk semantics is strictly included in simple-run semantics.*

Proof. $MW \subseteq SR$: Let $w \in [\![\mathcal{A}]\!]_{MW}(D)$, then w is a minimal walk in D that matches the query specified by $\mathcal{A}$. Assume $w \notin [\![\mathcal{A}]\!]_{SR}(D)$, then $\exists r$ in $D \times \mathcal{A}$ such that $w = \pi_D(r)$ and r is not a simple-run i.e. r contains a cycle. We denote by r' the run obtained by removing the cycle i.e. $r' \sqsubset r'$.So, $\pi_D(r') \sqsubset \pi_D(r)$ (Lemma 39), which means that $\pi_D(r)$ is not a minimal walk (Lemma 37). This contradicts the fact that w is a minimal walk.
$SR \not\subseteq MW$: See Example 5 in Table 1. The run $r = ((s, q_0) \xrightarrow{a} (c1, q_1) \xrightarrow{b} (c1, q_2) \xrightarrow{b} (t, q_2))$ is a simple run that matches the query. Therefore, $w_4 = \pi_D(r) \in [\![\mathcal{A}]\!]_{SR}(D)$. However, $w_4 \notin [\![\mathcal{A}]\!]_{MW}(D)$ as we have another match $w' = (s \xrightarrow{a} c1 \xrightarrow{b} t)$ such that $w' \sqsubset w_4$. ◀

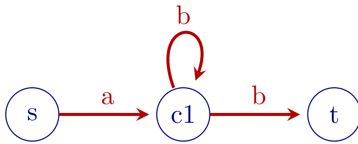


Figure 10 Graph Database D_4.

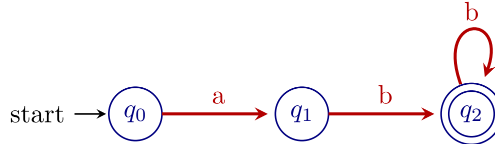


Figure 11 Automaton for Regular Expression: (ab^+).

▶ **Proposition 51.** *Shortest-coverage semantics and simple-run semantics are incomparable.*

Proof. $SC \not\subseteq SR$: See Example 1 in Table 1. We use the automaton in Figure 13 to represent the query. $w_1 \in [\![\mathcal{A}]\!]_{SC}(D)$ as it is a shortest walk that covers c_3. However, $w_1 = \pi_D(r)$ such that $r = ((s, q_0) \xrightarrow{a} (c1, q_0) \xrightarrow{a} (c2, q_0) \xrightarrow{a} (c3, q_0) \xrightarrow{b} (c1, q_0) \xrightarrow{a} (c2, q_0) \xrightarrow{a} (t, q_0))$, which is not a simple run. Therefore, $w_1 \notin [\![\mathcal{A}]\!]_{SR}(D)$.
$SR \not\subseteq SC$: See Example 3 in Table 1. We use the automaton in Figure 12 to represent the query. The run $r = ((s, q_0) \xrightarrow{a} (c1, q_0) \xrightarrow{c} (c2, q_0) \xrightarrow{c} (t, q_0))$ in $D \times \mathcal{A}$ is a simple run. Therefore, $w_2 = \pi_D(r) \in [\![\mathcal{A}]\!]_{SR}(D)$. However, $w_2 \notin [\![\mathcal{A}]\!]_{SC}(D)$ as it isn't a shortest walk covering any of its vertices. ◀

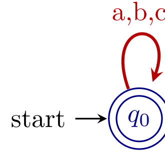


Figure 12 Automaton for Regular Expression: $(a + b + c)^*$.

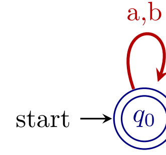


Figure 13 Automaton for Regular Expression: $(a + b)^*$.

▶ **Proposition 52.** *Minimal-walk semantics and shortest-coverage semantics are incomparable.*

Proof. $MW \not\subseteq SC$: See Example 3 in Table 1.
$SC \not\subseteq MW$: See Example 1 in Table 1. ◀

4 Computational Problems

In this section, we define some of the most common computational problems in querying graph databases. We restate the known results for these problems under the semantics that already exist in the literature. We then present the results for the new semantics that we have introduced in this paper.

4.1 Computational Problems Definitions and Known Results

4.1.1 Existence of a Matching Walk

Determining if there exists a walk in a graph database that matches a query automaton is a fundamental problem in graph databases. It is motivated by the different applications of graph databases. The problem is tractable under some semantics, where finding a witness is straightforward. However, it becomes intractable under other semantics, where the problem is NP-complete. We define the problem as follows:

PROBLEM	TUPLE MEMBERSHIP UNDER X SEMANTICS
INPUT	A graph database $D = (V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$, a query automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$, a pair of nodes $(s, t) \in V^2$.
OUTPUT	Does there exist a walk $w \in [\![\mathcal{A}]\!]_X(D)$ such that $\textsc{EndPoints}(w) = (s, t)$?

▶ **Theorem 53** ([16]). *TUPLE MEMBERSHIP under shortest-walk semantics is NL-complete.*

▶ **Theorem 54** ([15]). *Tuple Membership under trail or acyclic-walk semantics is NP-complete.*

Proof sketch. The proof of this theorem comes from reducing the Edge-Two-Disjoint-Paths problem, which is known to be NP-complete [9], to Tuple Membership under trail semantics.
The Edge-Two-Disjoint-Paths problem is defined as follows: given a graph $G = (V, E)$ and two pairs of nodes (s_1, t_1) and (s_2, t_2), does there exist two disjoint paths from s_1 to t_1 and from s_2 to t_2 in G (i.e. with no common edge)?
We construct a graph G' from G by labeling each edge with a and adding a new edge (t_1, s_2) labeled with b. Finding if there exists a walk from s_1 to t_2 in G' that matches the query a^*ba^* under trail semantics is equivalent to finding two disjoint paths in G. Therefore, the NP-completeness of the Edge-Two-Disjoint-Paths problem implies the NP-completeness of Tuple Membership under trail semantics.
The proof for acyclic-walk semantics follows the same lines as trail semantics. The only difference is that we reduce the Vertex-Two-Disjoint-Paths problem where the two disjoint paths are disjoint with respect to the vertices instead of the edges. ◁

A witness for the existence of walk under simple-run semantics is the shortest matching walk, hence the following result.

▶ **Theorem 55** ([7]). *Tuple Membership under simple-run semantics is NL-complete.*

4.1.2 Walk Membership

Our goal is to determine if a walk w is a match for a query automaton $\mathcal{A}$ on a graph database D under the two semantics that we have defined. We can state the problem as follows:

Problem	Walk Membership Under X Semantics
Input	A graph database $D = (V, E, \text{Src}, \text{Tgt}, \text{Lbl})$, a query automaton $\mathcal{A} = \langle Q, \Sigma, \delta, I, F\rangle$, a walk w.
Output	$w \in [\![\mathcal{A}]\!]_X(D)$?

▶ **Theorem 56.** *Walk Membership under shortest-walk, trail and acyclic walk semantics is NL-complete.*

The problem in Theorem 56 is equivalent to deciding if a word is accepted by a non-deterministic finite automaton (NFA), with the addition of checking that the walk is a trail or an acyclic walk in the case of trail and acyclic-walk semantics. The proof of this theorem is based on a reduction from the reachability problem in the product graph $D \times \mathcal{A}$.

▶ **Theorem 57** ([7]). *Walk Membership under simple-run semantics is NP-complete.*

4.1.3 Enumeration of Matching Walks

In order to evaluate a query on a graph database, we enumerate all the walks that match the query. The problem is tractable under simple-run semantics but intractable under trail or acyclic-walk semantics, which is a direct consequence of the NP-completeness of Tuple Membership under these semantics. Since the bag of matches is usually exponential in the size of the input, we don't use classical complexity to evaluate an enumeration algorithm.

Instead, enumeration complexity (defined in 2.3), where other parameters are taken into account, is used. We define the problem as follows:

Problem	Walks Enumeration Under X Semantics
Input	A graph database $D = (V, E, \text{Src}, \text{Tgt}, \text{Lbl})$, a query automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$, a source s, a target t.
Output	$\{\!\{ w \in [\![\mathcal{A}]\!]_X(D) \mid \text{EndPoints}(w) = (s,t) \}\!\}$.

▶ **Theorem 58.** *Walks Enumeration under trail or acyclic-walk semantics cannot be solved in polynomial preprocessing unless P = NP.*

▶ **Theorem 59.** *Walks Enumeration under simple-run is in DelayP.*

The enumeration of simple-runs was done using Yen's algorithm [21].

4.1.4 Distinct Enumeration of Matching Walks

A variation of the enumeration problem is the **distinct** enumeration problem, where we output each distinct walk only once. The results for the distinct enumeration problem are the same as the enumeration problem for simple-run, trail, and acyclic-walk semantics. In this problem, we add the EndPoints in the input. Since no duplicates are allowed, obtaining all the results can be done by a simple union. We define the problem as follows:

Problem	Distinct Walks Enumeration Under X Semantics
Input	A graph database $D = (V, E, \text{Src}, \text{Tgt}, \text{Lbl})$, a query automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$, a source s, a target t.
Output	$\{ w \in [\![\mathcal{A}]\!]_X(D) \mid \text{EndPoints}(w) = (s,t) \}$.

▶ **Theorem 60** ([6])**.** *Distinct Walks Enumeration under shortest-walk semantics is in NextP.*

Distinct shortest walks can be enumerated with a memoryless algorithm [6] using preprocessing in $O(|D| \times |\mathcal{A}|)$ and a delay in $O(\gamma \times |\mathcal{A}|)$, where γ is the length of a shortest walk.

Prolongability of a Walk

We introduce a new problem that we call Prolongability of a walk for the sake of solving Distinct Walks Enumeration. The problem is to determine if a walk can be prolonged to a longer walk that matches the query. It is defined as follows:

Problem	Prolongability Under X Semantics
Input	A graph database $D = (V, E, \text{Src}, \text{Tgt}, \text{Lbl})$, a query automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$, a walk w, a target t.
Output	Does there exist a walk w', such that $w \cdot w' \in [\![\mathcal{A}]\!]_X(D)$ and $\text{Tgt}(w') = t$?

The problem can be used as an oracle to solve Distinct Walks Enumeration in polynomial preprocessing and polynomial delay. Algorithm 3 shows how. The idea is perform a depth-first search on the product graph $D \times \mathcal{A}$. At each step, we call the Prolongability oracle to determine if the current walk can be prolonged to a longer walk that matches the query. If the walk can be prolonged, we continue the depth-first search. If the walk cannot be prolonged, we stop exploring the current branch of the search tree.

Algorithm 3 Distinct Enumeration of Matching Walks Using Prolongability

```
Data: D, A, s, t
Result: All w ∈ [[A]]_X(D)
1 RecDistEnum ((s), D, A, t); // (s) is a walk of length 0
2 Function RecDistEnum(w, D, A, t)
3     v ← Tgt(w);
4     if v = t ∧ membership (w, D, A) then
5         output(w);
6     for e ∈ out(v) do
          // out(v) is the set of outgoing edges from v in D
7         w′ ← w · (v -e-> Tgt(e));
8         if Prolongability (D, A, w′, t) then
9             RecDistEnum (w′, D, A, t);
```

▶ **Theorem 61.** *Distinct Walks Enumeration under X semantics can be solved with no preprocessing and a delay in $O(len_{max} \cdot deg_{max} \cdot f(|D|, |\mathcal{A}|))$, where f is the maximum complexity between Prolongability and Walk Membership under X semantics, len_{max} is the maximum length of a walk in $[\![A]\!]_X(D)$, and deg_{max} is the maximum out-degree of a vertex in D.*

Proof. We use Algorithm 3, where a branch is only explored if we know that it is successful. At each step of the depth-first search, we call the Prolongability oracle to determine if the current walk can be prolonged to a longer walk that matches the query. We consider deg the out-degree of a vertex. Consider the delay between outputting two consecutive walks w' and w. After outputting, w' one backtracks from w' until we find a prefix $v \cdot a$ of w' such that $v \cdot b$ is prolongable and $b > a$. Then w is the smallest suffix of $v \cdot b$ to output. To get w one needs to continue the depth-first search by up to a certain vertex then exploring w. Therefore, the Prolongability oracle is called at most $\sum_{v \in vertices(w)} deg(v)$ times during backtracking and at most $\sum_{v \in vertices(w')} deg(v)$ times going forward. Hence, outputting the next solution w of a solution w' can be done in $O((\sum_{v \in vertices(w)} deg(v) + \sum_{v \in vertices(w')}) \cdot f(|D|, |\mathcal{A}|))$. In the worst case scenario, the solution w is the longest walk in the graph, and each of its vertices has the maximum out-degree, which makes the delay bounded by $O(len_{max} \cdot deg_{max} \cdot f(|D|, |\mathcal{A}|))$. ◀

4.2 Minimal-walk Semantics: Computational Problems

4.2.1 Existence of a Walk under Minimal-walk Semantics

Just as it is the case for simple-run semantics, a shortest walk is a witness for the existence of a minimal walk that matches the query.

▶ **Theorem 62.** *Tuple Membership under minimal-walk semantics is NL-complete.*

Proof. Tuple Membership under minimal-walk semantics is equivalent to Tuple Membership under shortest-walk semantics, which we know is NL-complete (Lemma 53).
The existence problem is equivalent to the emptiness. Therefore, we show the following:

$$\forall D, \forall \mathcal{A}, \quad [\![\mathcal{A}]\!]_{MW}(D) = \emptyset \iff [\![\mathcal{A}]\!]_{SW}(D) = \emptyset$$

$\forall D, \forall \mathcal{A}$,
$\implies$: if $[\![\mathcal{A}]\!]_{MW}(D) = \emptyset$, then $[\![\mathcal{A}]\!]_{SW}(D) = \emptyset$ since $[\![\mathcal{A}]\!]_{SW}(D) \subseteq [\![\mathcal{A}]\!]_{MW}(D)$.
$\impliedby$: if $[\![\mathcal{A}]\!]_{SW}(D) = \emptyset$, then there is no walk in D that matches the query specified by $\mathcal{A}$, i.e. $\textsc{Match}_{\mathcal{A}}(D) = \emptyset$. Therefore, $[\![\mathcal{A}]\!]_{MW}(D) = \emptyset$. ◀

4.2.2 Walk Membership under Minimal-walk Semantics

In order to determine if a walk w is a match for a query automaton under minimal-walk semantics, we use the Algorithm 4. The idea is to create a structure that we formally define in Definition 63 called the subwalk automaton. The subwalk automaton is a finite automaton where a copy of each vertex in w is a state (if a vertex is repeated, the copies are also repeated and in order) and the transitions are labeled with the edge labels of w. We add ε-transitions between a state and the next state that represents the same vertex. The language of the subwalk automaton returns the labels of the subwalks of w. We then perform a BFS on the product of the subwalk automaton and the query to find the shortest walk from the start state to the final state. If the length of the shortest walk is less than the length of w, then w is not a minimal walk, as there exists a subwalk of w that is a match for the query. Conversely, if the shortest walk in the subwalk automaton has the same length as w, then w is a minimal walk.

▶ **Definition 63.** *The **subwalk automaton** $D_{\sqsubseteq w}$ of a walk $w = n_0 \xrightarrow{e_1} n_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} n_k$ in a graph database D is the automaton*[4] $D_{\sqsubseteq w} = \langle Q_{\sqsubseteq w}, \Sigma, \delta_{\sqsubseteq w}, I_{\sqsubseteq w}, F_{\sqsubseteq w} \rangle$ *where:*

- $Q_{\sqsubseteq w} = \{i \mid 0 \leq i \leq k\}$
- $I_{\sqsubseteq w} = \{0\}$
- $F_{\sqsubseteq w} = \{k\}$
- $\delta_{\sqsubseteq w} = \{(i, a, i+1) \mid 0 \leq i < k \text{ and } a \in \textsc{Lbl}(e_{i+1})\} \cup$
  $\{(i, \varepsilon, j) \mid n_i = n_j, 0 \leq i < j \leq k \text{ and } \nexists \ell \in]i, j[\text{ such that } n_\ell = n_i\}$

Algorithm 4 Walk Membership Under Minimal-walk Semantics

Data: w, D, $\mathcal{A}$
Result: $w \in [\![\mathcal{A}]\!]_{MW}(D)$?
1 **if** $w \notin \textsc{Match}_{\mathcal{A}}(D)$ **then**
2   **return** false;
3 $D_{\sqsubseteq w} \leftarrow$ subwalkautomaton(w, D);
4 $D_{\sqsubseteq w} \leftarrow$ removeε-transitions(w, D);
5 $\mathit{anyShortestWalk} \leftarrow$ MultiBFS($D_{\sqsubseteq w} \times \mathcal{A}$, $\textsc{Src}(w) \times I$, $\textsc{Tgt}(w) \times F$);
6 **return** $|\mathit{shortestWalk}| = |w|$;

---

[4] We use a slightly modified version of an automaton where we allow ε-transitions.

Algorithm 5 Subwalk Automaton Construction

Data: $w = n_0 \xrightarrow{e_1} \dots \xrightarrow{e_k} n_k$, D
Result: $D_{\sqsubseteq w}$
1 $Q_{\sqsubseteq w} \leftarrow \{i \mid 0 \leq i \leq k\}$;
2 $I_{\sqsubseteq w} \leftarrow \{0\}$;
3 $F_{\sqsubseteq w} \leftarrow \{k\}$;
4 $\delta_{\sqsubseteq w} \leftarrow \emptyset$;
5 $Copy[v \in V] \leftarrow \emptyset$;
`// dictionnary indexed by the vertices`
6 **for** $i \in 0 \dots k$ **do**
7 $\quad n_i \leftarrow i$th node of w **if** $Copy[n_i] \neq \emptyset$ **then**
8 $\quad\quad \delta_{\sqsubseteq w} \leftarrow \delta_{\sqsubseteq w} \cup \{(Copy[n_i], \varepsilon, i)\}$;
9 $\quad Copy[n_i] \leftarrow i$;
10 **for** $i \in 0 \dots |w| - 1$ **do**
11 $\quad \delta_{\sqsubseteq w} \leftarrow \delta_{\sqsubseteq w} \cup \{(i, \textsc{Lbl}(e_{i+1}), i+1)\}$
12 **return** $D_{\sqsubseteq w} = (Q_{\sqsubseteq w}, \Sigma, \delta_{\sqsubseteq w}, I_{\sqsubseteq w}, F_{\sqsubseteq w})$

Proof of Correctness

We consider in what follows $w = n_0 \xrightarrow{e_1} n_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} n_k$ a walk in $\textsc{Match}_{\mathcal{A}}(D)$.

▶ **Lemma 64.** *Let n_i, n_j be the ith and jth nodes of w respectively.*

$$n_i = n_j \wedge i < j \iff (i, \varepsilon, j) \in \delta^*_{\sqsubseteq w}$$

Proof.
- $\implies$: We have $n_i = n_j \wedge i < j$. We proceed by induction on the number of n_k such that $i < k < j$ and $n_k = n_i$
  - Base case: $n = 0$. In this case, n_j is the first copy of n_i after i. Therefore, $(i, \varepsilon, j) \in \delta_{\sqsubseteq w}$ by definition of $\delta_{\sqsubseteq w}$.
  - Inductive step: We have $n + 1$ occurrences of n_i between i and j. Let ℓ be the index of the last occurrence. By induction hypothesis, $(i, \varepsilon, \ell) \in \delta^*_{\sqsubseteq w}$ and by definition $(\ell, \varepsilon, j) \in \delta_{\sqsubseteq w}$. Thus, by transitivity of $\delta^*_{\sqsubseteq w}$, $(i, \varepsilon, j) \in \delta^*_{\sqsubseteq w}$.
- $\impliedby$: Follows directly from the fact that $(i, \varepsilon, j) \in \delta_{\sqsubseteq w} \implies n_i = n_j$. ◀

▶ **Lemma 65.** *Let w be a walk in D.*

$$\mathcal{L}(D_{\sqsubseteq w}) = \{\textsc{Lbl}(w') \mid w' \sqsubseteq w\}$$

Proof. We denote by w_i the prefix of w that ends at the ith node.

- $\implies$: $\forall i \in \{0, \dots, k\}, \forall m$ such that $(0, m, i) \in \delta^*_{\sqsubseteq w}$, we show that $\exists w' \sqsubseteq w_i$ such that $m \in \textsc{Lbl}(w')$. We proceed by induction on i.
  - Base case: $i = 0$. Let m be such that $(0, m, 0) \in \delta^*_{\sqsubseteq w}$. By definition of $\delta^*_{\sqsubseteq w}$, $m = \varepsilon$. Note that $w_0 = (n_0)$ hence $\textsc{Lbl}(w_0) = \{\varepsilon\}$ hence $w' = w_0$ is a subwalk of w_0 that satisfies $\varepsilon \in \textsc{Lbl}(w')$.
  - Inductive step: $i+1$. Let m be such that $(0, m, i+1) \in \delta^*_{\sqsubseteq w}$. There is m' and $a \in \Sigma \cup \{\varepsilon\}$ such that $(0, m', j) \in \delta^*_{\sqsubseteq w}$, $(j, a, i+1) \in \delta_{\sqsubseteq w}$ and $j < i$. By induction hypothesis, $\exists w' \sqsubseteq w_i$ such that $m' \in \textsc{Lbl}(w')$. If $a = \varepsilon$, then $m = m'$. Using Lemma 64, we get

that $n_j = n_{i+1}$. We have $w_j \sqsubseteq w_{i+1}$ since w_{i+1} is obtained by removing the cycle between n_i and n_j. Thus, $w' \sqsubseteq w_{i+1}$ such that $m \in \textsc{Lbl}(w')$. Otherwise, $a \neq \varepsilon$, then $i = j$ by definition of $\delta_{\sqsubseteq w}$. Therefore, $a \in \textsc{Lbl}(e_{i+1})$ and $(w' \cdot (n_i \xrightarrow{e_{i+1}} n_{i+1})) \sqsubseteq w_{i+1}$. Thus, $\exists w'' \sqsubseteq w$ such that $m \in \textsc{Lbl}(w'')$, where $w'' = w' \cdot (n_i \xrightarrow{e_{i+1}} n_{i+1})$ and $m = m'a$.

- $\Longleftarrow$: $\forall i \in \{0, \dots, k\}, \forall m$ such that $\exists w' \sqsubseteq w_i$ and $m \in \textsc{Lbl}(w')$, we show that $(0, m, i) \in \delta^*_{\sqsubseteq w}$.
  - Base case: $i = 0$. Let $m \in \textsc{Lbl}(w')$ such that $\exists w' \sqsubseteq w_0$. We have $w' = n_0$. Thus $m = \varepsilon$. We have $(0, \varepsilon, 0) \in \delta^*_{\sqsubseteq w}$ by definition of δ^*.
  - Inductive step: $i+1$. Let $m \in \textsc{Lbl}(w')$ such that $\exists w' \sqsubseteq w_{i+1}$. If e_{i+1} isn't the last edge of w', then $\exists j < i$ such that $w' \sqsubseteq w_j$. Therefore, we $n_j = n_{i+1}$, thus from Lemma 64, we have $(j, \varepsilon, i+1) \in \delta^*_{\sqsubseteq w}$. Moreover, by induction hypothesis, $(0, m, j) \in \delta^*_{\sqsubseteq w}$. Thus, by transitivity of $\delta^*_{\sqsubseteq w}$, we have $(0, m, i+1) \in \delta^*_{\sqsubseteq w}$. Otherwise, $w' = w'' \cdot (n_i \xrightarrow{e_{i+1}} n_{i+1})$. There is m' and a such that $m = m'a$ and $m' \in \textsc{Lbl}(w'')$. We have $w'' \sqsubseteq w_i$, thus by induction hypothesis, $\exists (0, m, i) \in \delta^*_{\sqsubseteq w}$. Moreover, $(i, a, i+1) \in \delta_{\sqsubseteq w}$ by definition of $\delta_{\sqsubseteq w}$. Therefore, $(0, m, i+1) \in \delta^*_{\sqsubseteq w}$.

◀

▶ **Lemma 66.** *Let w be a walk in a graph database D and $\mathcal{A}$ be a query automaton.*

$$\mathcal{L}(D_{\sqsubseteq w} \times \mathcal{A}) = \{\textsc{Lbl}(w') \mid w' \sqsubseteq w \text{ and } w' \in \textsc{Match}_{\mathcal{A}}(D)\}$$

Proof. $m \in \mathcal{L}(D_{\sqsubseteq w} \times \mathcal{A})$
$\iff m \in \mathcal{L}(D_{\sqsubseteq w})$ and $m \in \mathcal{L}(\mathcal{A})$
$\iff m \in \{\textsc{Lbl}(w') \mid w' \sqsubseteq w\}$ and $m \in \mathcal{L}(\mathcal{A})$
$\iff \exists w' \sqsubseteq w$ such that $\textsc{Lbl}(w') = m$ and $m \in \mathcal{L}(\mathcal{A})$
$\iff \exists w' \sqsubseteq w$ such that $\textsc{Lbl}(w') = m$ and $w' \in \textsc{Match}_{\mathcal{A}}(D)$ ◀

▶ **Lemma 67.** *Let w be a walk in D and m the shortest word in $\mathcal{L}(D_{\sqsubseteq w} \times \mathcal{A})$. If $|m| < |\textsc{Lbl}(w)|$, then w is not a minimal walk.*

Proof. Let m be the shortest word in $\mathcal{L}(D_{\sqsubseteq w})$ such that $|m| < |\textsc{Lbl}(w)|$.
$\iff \exists w' \sqsubset w$ such that $\textsc{Lbl}(w') = m$ and $w' \in \textsc{Match}_{\mathcal{A}}(D)$.
$\iff w$ is not a minimal walk. ◀

▶ **Theorem 68.** *Algorithm 4 solves Walk Membership under minimal-walk semantics.*

Proof. Using Lemma 67. ◀

Time Complexity

▶ **Lemma 69.** *The size of $D_{\sqsubseteq w}$ is in $O(|w|)$.*

Proof. We have $|D_{\sqsubseteq w}| = |Q_{\sqsubseteq w}| + |\delta_{\sqsubseteq w}|$. By Definition 63, $|Q_{\sqsubseteq w}| = |w|$. In the worst case scenario, $\delta_{\sqsubseteq w}$ contains $2 \cdot |w|$ transitions: $|w|$ transitions of the form $(i, a, i+1)$ and $|w|$ transitions of the form (i, ε, j), where $i, j \in Q_{\sqsubseteq w}$ and $a \in \Sigma$. Therefore, the size of $D_{\sqsubseteq w}$ is in $O(|w|)$. ◀

▶ **Lemma 70.** *Algorithm 5 has a time complexity of $O(|w| \log(|w|))$.*

Proof. We construct $I_{\sqsubseteq w}$, and $F_{\sqsubseteq w}$ in constant time. The algorithm constructs $Q_{\sqsubseteq w}$ and the non ε-transitions in $\delta_{\sqsubseteq w}$ by iterating over w. The ε-transitions are constructed using a dictionary *Copy* that maps each vertex to the index of the last occurrence of the vertex in w. We choose to use an AVL tree to implement the dictionary, which has a time complexity of $O(\log(|w|))$ for each insertion. Therefore, the time complexity of the algorithm is $O(|w|\log(|w|))$. ◀

▶ **Theorem 71.** *Algorithm 4 has a time complexity of* $O(|w|\log(|w|) + |w|^2 \cdot |\mathcal{A}|)$.

Proof. The time complexity of the algorithm is as follows. The construction of the subwalk automaton $D_{\sqsubseteq w}$ has a time complexity of $O(|w|\log(|w|))$ as stated in Lemma 70 The size of $D_{\sqsubseteq w}$ is in $O(|w|)$ (Lemma 69), hence the removal of the ε-transitions results in an automaton of size in $O(|w|^2)$ Finally, BFS algorithm also has a time complexity of $O(|w|^2 \cdot |\mathcal{A}|)$. Thus, the time complexity of the algorithm is $O(|w|\log(n) + |w|^2 \cdot |\mathcal{A}|)$. ◀

4.3 Shortest-coverage Semantics: Computational Problems

4.3.1 Existence of a Walk under Shortest-coverage Semantics

Just as it is the case for minimal-walk semantics, a shortest walk is a witness for the existence of a walk that matches the query and is a shortest walk covering its vertices.

▶ **Theorem 72.** *Tuple Membership under shortest-coverage semantics is NL-complete.*

Proof. Tuple Membership under shortest-coverage semantics is equivalent to Tuple Membership under shortest-walk semantics, which we know is NL-complete (Lemma 53). The existence problem is equivalent to the emptiness. Therefore, we show the following:

$$\forall D, \forall \mathcal{A}, \quad [\![\mathcal{A}]\!]_{SC}(D) = \emptyset \iff [\![\mathcal{A}]\!]_{SW}(D) = \emptyset.$$

$\forall D, \forall \mathcal{A}$,
$\implies$: if $[\![\mathcal{A}]\!]_{SC}(D) = \emptyset$, then $[\![\mathcal{A}]\!]_{SW}(D) = \emptyset$ since $[\![\mathcal{A}]\!]_{SW}(D) \subseteq [\![\mathcal{A}]\!]_{SC}(D)$ (Proposition 45).
$\impliedby$: if $[\![\mathcal{A}]\!]_{SW}(D) = \emptyset$, then there is no walk in D that matches the query specified by $\mathcal{A}$, i.e. $\textsc{Match}_{\mathcal{A}}(D) = \emptyset$. Thus $\forall (s,t) \in V^2$, $\textsc{Match}_{\mathcal{A}}(D,s,t) = \emptyset$. Therefore, $[\![\mathcal{A}]\!]_{SC}(D) = \emptyset$. ◀

4.3.2 Walk Membership under Shortest-coverage Semantics

In order to determine if a given walk w is a match for a query under shortest-coverage semantics, we use the Algorithm 6. The idea is to perform two MultiBFS (Algorithm 1): one on $D \times \mathcal{A}$ from the source states $(\textsc{Src}(w), q_0)$ to the target states $(\textsc{Tgt}(w), q_f)$ and the other on the transpose of $D \times \mathcal{A}$ from the target states $(\textsc{Tgt}(w), q_f)$ back to the source states $(\textsc{Src}(w), q_0)$, where $q_0 \in I$ and $q_f \in F$. During each BFS, we store the distance from the source during the first BFS and the distance from the target during the second BFS. For each node v of w, we test with the `Witness` function in Algorithm 7 if there is a vertex (v, q) in $D \times \mathcal{A}$ with a sum of distances inferior to the length of w. If such a vertex exists, then w is not a shortest walk covering v. Conversely, if there is no such vertex, then w is a shortest walk covering v, and the `Witness` function returns true. We say that w is a witness for v.

Algorithm 6 Walk Membership Under Shortest-coverage Semantics

```
Data: w, 𝒜, D
Result: w ∉ ⟦𝒜⟧_SC(D)?
1  if w ∈ Match_𝒜(D) then
2      return false;
3  VertexW[v ∈ V] ← 0;
   // Array indexed by the vertices of V
4  for i ∈ 0...|w| do
5      v_i ← ith vertex of w;
6      VertexW[v_i] ← 1;
7  D × 𝒜 ← product(D, 𝒜);
8  distances1 ← MultiBFS(D × 𝒜, Src(w) × I, Tgt(w) × F);
9  distances2 ← MultiBFS((D × 𝒜)^T, Tgt(w) × F, Src(w) × I);
10 for v ∈ V do
11     if VertexW[v] = 1 then
           // Check if w is a shortest walk covering v
12         if Witness (v, w, distances1, distances2, 𝒜) then
13             return true
14 return false
```

Algorithm 7 Witness: Shortest Walk Covering a Vertex

```
Data: v, w, distances1, distances2, 𝒜
Result: Is w one of the shortest walks covering v?
1  Function Witness(v, w, distances1, distances2, 𝒜)
2      for q ∈ Q do
3          if distances1[(v, q)] + distances2[(v, q)] < |w| then
4              return false;
5      return true;
```

Proof of Correctness

▶ **Theorem 73.** *Algorithm 6 solves* Walk Membership *under shortest-coverage semantics.*

Proof. Let w be a walk such that $w \in \textsc{Match}_{\mathcal{A}}(D, \textsc{Src}(w), \textsc{Tgt}(w))$. Let *distances*1 and *distances*2 be the distances obtained from the BFS algorithms on the product $D \times \mathcal{A}$.
The correctness of the algorithm is based on the `Witness` function in Algorithm 7. We prove in what follows the equivalence between w being a shortest walk covering one of its vertices v and the `Witness` function returning **true** for this walk and this vertex:
Consider w a shortest walk covering v.
$\iff \nexists w' \in \textsc{Match}_{\mathcal{A}}(D, \textsc{Src}(w), \textsc{Tgt}(w))$ such that w' covers v and $|w'| < |w|$.
$\iff \nexists w_1, w_2$ such that $\textsc{Tgt}(w_1) = \textsc{Src}(w_2) = v$, $|w_1| + |w_2| < |w|$ and $w_1 \cdot w_2 \in \textsc{Match}_{\mathcal{A}}(D, \textsc{Src}(w), \textsc{Tgt}(w))$.
$\iff \nexists w_1, w_2$ such that $\textsc{Tgt}(w_1) = \textsc{Src}(w_2) = v$, $|w_1| + |w_2| < |w|$ and $\exists \rho_1, \rho_2$ where $\rho_1 \cdot \rho_2 \in \textsc{Run}_{\mathcal{A}}(D)$, $\pi_D(\rho_1) = w_1$ and $\pi_D(\rho_2) = w_2$.
$\iff \nexists \rho_1, \rho_2, q$ such that $\textsc{Tgt}(\rho_1) = \textsc{Src}(\rho_2) = (v, q)$, $|\pi_D(\rho_1)| + |\pi_D(\rho_2)| < |w|$ and

$\pi_D(\rho_1 \cdot \rho_2) \in \textsc{Match}_{\mathcal{A}}(D, \textsc{Src}(w), \textsc{Tgt}(w))$.
$\iff \nexists q$ such that a run in $D \times \mathcal{A}$ from $(\textsc{Src}(w), q_0)$ to $(\textsc{Tgt}(w), q_f)$ covers (v, q) and has a length inferior to $|w|$.
$\iff$ the `Witness` function returns **true** for v and w.
A walk is in $[\![\mathcal{A}]\!]_{SC}(D)$ if and only if it is a shortest walk covering at least one of its vertices (Definition 42), which we have shown is equivalent to the `Witness` function returning **true** for this walk and this vertex. ◀

Time Complexity

▶ **Lemma 74.** *Let w be a walk in D. If $w \in [\![\mathcal{A}]\!]_{SC}(D)$, then $|w| \leq (2 \times |V| \times |Q|)$.*

Proof. Let w be a walk such that $w \in [\![\mathcal{A}]\!]_{SC}(D)$ thus $\exists v \in V$ such that w is a shortest walk covering v. Therefore, $\exists w_1, w_2$ such that $w = w_1 \cdot w_2$ and $\textsc{Src}(w_1) = v = \textsc{Tgt}(w_2)$. We have $|w| = |w_1| + |w_2|$. We show that $|w_1| \leq (|V| \times |Q|)$ and $|w_2| \leq (|V| \times |Q|)$.

- $|w_1| \leq (|V| \times |Q|)$:
  For the sake of contradiction, assume that $|w_1| > |V| \times |Q|$. This means that $\exists r$ such that $\pi_D(r) = w_1$. Since the projection function preserves the length of the runs, we have $|r| > (|V| \times |Q|)$. In the run r, we have $|r| + 1$ vertices and the number of distinct vertices in $D \times \mathcal{A}$ is $|V| \times |Q|$. By the pigeonhole principle, since $|r| + 1 > (|V| \times |Q|)$, we are distributing $|r| + 1$ elements in $|V| \times |Q|$ boxes. Therefore, there exists a box that contains at least two elements. This means that there exists a vertex (v, q) that is visited at least twice in the run r, i.e. there exists a cycle in the run r. Since, the cycle is in the run r, it is also in the walk w_1 and removing the cycle from w_1 gives a walk w_1' such that $w_1' \cdot w_2 \in [\![\mathcal{A}]\!]_{SC}(D)$. Therefore, we have $w_1' \cdot w_2 \in [\![\mathcal{A}]\!]_{SC}(D)$ and $|w_1' \cdot w_2| < |w|$, which contradicts the fact that w is a shortest walk covering v. Therefore, $|w_1| \leq (|V| \times |Q|)$.
- $|w_2| \leq (|V| \times |Q|)$:
  The same reasoning can be applied to w_2.

Therefore, $|w| = |w_1| + |w_2| \leq 2 \times |V| \times |Q|$. ◀

▶ **Theorem 75.** *Algorithm 6 has a time complexity of $O(|\mathcal{A}| \times |D|)$.*

Proof. The time complexity of the algorithm is as follows. The creation of $VertexW$ has a time complexity of $O(|w|)$. By Lemma 74, $|w| \leq (2 \times |D| \times |\mathcal{A}|)$, which means that this step is in $O(|D| \times |\mathcal{A}|)$. The construction of the product of the graph database D and the query automaton $\mathcal{A}$ is done in $O(|\mathcal{A}| \times |D|)$. The BFS algorithms are done on the product graph $D \times \mathcal{A}$ and thus have a time complexity of $O(|\mathcal{A}| \times |D|)$. The `Witness` function has two loops of size $|Q|$ and $|V|$. Therefore, the total time complexity of the algorithm is $O(|\mathcal{A}| \times |D| + |Q| \times |V|)$, which is bounded by $O(|\mathcal{A}| \times |D|)$. ◀

We note that the algorithm can be optimized by stopping the BFS after $|w|$ iterations. Therefore, instead of exploring the entire product graph, we can explore only a subgraph of the neighborhoods of the sources $(\textsc{Src}(w) \times I)$ and the targets $(\textsc{Tgt}(w) \times F)$, of length $|w|$. This leads to only exploring the vertices in the subgraph. This optimization improves complexity in best-case scenarios while maintaining the same complexity in worst-case scenarios. We show in Lemma 76 that the algorithm is still correct with this optimization.

▶ **Lemma 76.** *A BFS in algorithm 6 can be made to stop after $|w|$ iterations.*

Proof. The correctness of the Algorithm 8 is not affected by stopping after $|w|$ iterations. The algorithm relies mainly on the `Witness` function, where we check if the sum of the distances from the source and the target is less than the length of w. If one BFS does more than $|w|$ iterations, then we already know that the walk currently being explored is longer than w. Therefore, we can already answer the `Witness` function with **false**. Thus, we can avoid unnecessary iterations and stop the BFS after $|w|$ loops. ◀

4.3.3 Enumeration of Walks under Shortest-coverage Semantics

We can enumerate the bag $[\![\mathcal{A}]\!]_{SC}(D)$ using Algorithm 8. The idea is to get the matrix of distances between all pairs of vertices in the product graph $D \times \mathcal{A}$ using the Floyd-Warshall algorithm [4]. Then, for each pair of vertices (s, t), we find the shortest walks covering each vertex v. We do this by finding a state q such that the sum of the distances from (s, i) to (v, q) and from (v, q) to (t, f) is minimal, where $i \in I$ and $f \in F$. We then use the algorithm `EnumShortest` introduced in [6] to efficiently enumerate the shortest walks from (s, i) to (v, q) and from (v, q) to (t, f), i.e. the walks with the minimal sum of distances. Finally, we output the product of these two results.

Algorithm 8 Enumeration of Matching Walks Under Shortest-coverage Semantics

Data: D, $\mathcal{A}$

Result: Shortest walks for each pair of nodes (s, t)

1. $D \times \mathcal{A} \leftarrow$ product(D, $\mathcal{A}$);
2. $Distances \leftarrow$ Floyd-Warshall($D \times \mathcal{A}$);
3. **for** $(s, t) \in V^2$ **do**
4.   **for** $v \in V$ **do**
5. $\rho_1 \leftarrow []$;
6. $\rho_2 \leftarrow []$;
7. $d \leftarrow \min_{(i,q,f) \in I \times Q \times F} Distances[(s, i), (v, q)] + Distances[(v, q), (t, f)]$;
       `// List of witness states for` v
8. $witnessstates \leftarrow \{q \in Q \mid Distances[(s, i), (v, q)] + Distances[(v, q), (t, f)] = d\}$;
9.   **for** $q \in witnessstates$ **do**
         `// Find the shortest walks from` (s, i) `to` (v, q) `where` $i \in I$
10. $\rho_1 \leftarrow$ `EnumShortest` $(D \times \mathcal{A}, (s, I), (v, q))$;
          `// Find the shortest walks from` (v, q) `to` (t, f) `where` $f \in F$
11. $\rho_2 \leftarrow$ `EnumShortest` $(D \times \mathcal{A}, (v, q), (t, F))$;
12.   **output** $(\pi_D(\rho_1) \times \pi_D(\rho_2))$;

Proof of Correctness

▶ **Theorem 77.** *Algorithm 8 solves* Walks Enumeration *under shortest-coverage semantics.*

The proof of this theorem is similar to the proof of the correctness of Algorithm 6. For each vertex v, a list of witness states is found. For each witness state q, we output the product of the projections of the shortest runs from (s, I) to (v, q) and from (v, q) to (t, F), which are all the shortest walks covering v.

Time Complexity

▶ **Theorem 78.** *Walks Enumeration can be solved with Algorithm 8 in DelayP.*

Proof. The time complexity of the algorithm is as follows. The construction of the product of the graph database D and the query automaton $\mathcal{A}$ has a time complexity of $O(|\mathcal{A}| \times |D|)$. The Floyd-Warshall algorithm runs in $O((|\mathcal{A}| \times |D|)^3)$ [4]. Therefore, the preprocessing time is polynomial in the size of the input. For each pair of vertices (s, t) and for each vertex v, we find the witness states which a time complexity of $O(|Q|)$. The `EnumShortest` algorithm has a preprocessing in $O(|D| \times |\mathcal{A}|)$ and a delay in $O(\gamma \times |\mathcal{A}|)$, where γ is the length of a shortest walk, which we know is in $O(|D| \times |\mathcal{A}|)$. Therefore, the delay of the algorithm is in $O(|\mathcal{A}| \times |D| \times |Q|)$ and thus is polynomial in the size of the input. We conclude that the enumeration Algorithm 8 is in DelayP. ◀

▶ Remark 79. Note that Algorithm 8 can output the same walk several times. The multiplicity of each walk corresponds to the product of number of runs of which it is the projection and the number of vertices of which it is the shortest covering walk. In the following section, we study the distinct enumeration of the walks.

4.3.4 Distinct Enumeration of Walks under Shortest-coverage Semantics

In order to solve Distinct Walks Enumeration under shortest-coverage semantics, we use Algorithm 3 which relies on the Prolongability oracle. In what follows, we show that Prolongability under shortest-coverage semantics can be solved in P (in Algorithm 9). The idea is to precompute the matrix of distances using the Floyd-Warshall algorithm then use the distances to check if a walk can be prolonged to cover a vertex v. Let s' be the target vertex of the walk w we want to prolong. The vertex may appear in w or in the prolongation of w. We verify both cases as the order of the vertices s, s' and v in the walk w is important distance-wise.

Proof of Correctness

▶ **Theorem 80.** *Algorithm 9 solves Prolongability under shortest-coverage semantics.*

Proof. If a walk w is prolongable, then there exists a walk w' such that $w \cdot w' \in [\![\mathcal{A}]\!]_{SC}(D)$. This means that $\exists v \in V$ such that $w \cdot w'$ is a shortest walk covering a vertex $v \in V$. We write s, s' and t to denote $\textsc{Src}(w \cdot w'), \textsc{Tgt}(w)$ and $\textsc{Tgt}(w \cdot w')$ respectively. $w \cdot w'$ contains the vertices s, s', v and t. The vertices either occur in the order: s, s', v, t or in the order s, v, s', t. If v occurs in w and w', then both cases are possible.

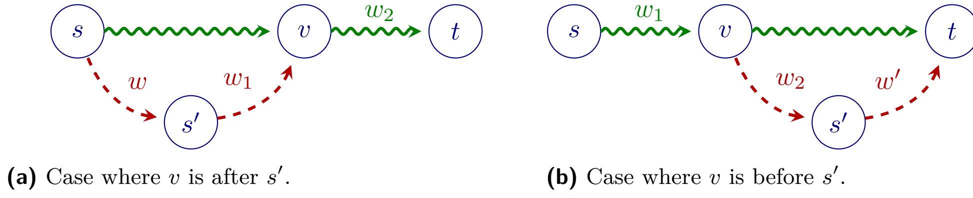
- If $w \cdot w'$ contains s, s', v and t in this order (see Figure 14a):
  We write $w \cdot w' = w \cdot w_1 \cdot w_2$ where w_1 is the walk from s' to v and w_2 is the walk from v to t. Then, $\exists \rho, \rho_1, \rho_2 \in \textsc{Run}_{\mathcal{A}}(D), \exists i \in I, \exists q' \in \delta^*(i, \textsc{Lbl}(w)), \exists q \in Q$ such that $\pi_D(\rho) = w$, $\pi_D(\rho_1) = w_1$, $\pi_D(\rho_2) = w_2$, $\textsc{Src}(\rho) = (s, i)$, $\textsc{Tgt}(\rho) = (s', q')$ and $\textsc{Tgt}(\rho_1) = (v, q)$. Since $w \cdot w'$ is a shortest walk covering v we have : $|\rho \cdot \rho_1| = \mathit{Distances}[(s, i), (v, q)]$. The fact that the walk can be prolonged to cover v means $\exists f \in F$ such that $|\rho \cdot \rho_1 \cdot \rho_2| = \mathit{Distances}[(s, i), (v, q)] + \mathit{Distances}[(v, q), (t, f)] = |\rho| + \mathit{Distances}[(s', q'), (v, q)] + \mathit{Distances}[(v, q), (t, f)]$, which is what the algorithm checks. Thus, the algorithm returns **true** if w is prolongable.
- If $w \cdot w'$ contains s, v, s' and t in this order (see Figure 14b):
  We write $w \cdot w' = w_1 \cdot w_2 \cdot w'$ where w_1 is the walk from s to v, w_2 is the walk

Algorithm 9 Prolongability under Shortest-coverage Semantics

```
Data: w, D, A, t
Result: Is w prolongable?
1  D × A ← product(D, A);
2  Distances ← Floyd-Warshall(D × A);
   // Checks that |w| < 2 × |V| × |Q| (Lemma 74)
3  if |w| > 2 × |V| × |Q| then
4      return false;
5  VertexW[v ∈ V] ← 0;
   // Array indexed by the vertices of V
6  for i ∈ 0 ... |w| do
7      v_i ← ith vertex of w;
8      VertexW[v_i] ← 1;
9  s ← Src(w); s' ← Tgt(w);
   // We consider the covered vertex in the prolongation of w
   // Set of states that can be reached by reading the label of w
10 GoodStates ← δ*(I, Lbl(w));
11 for v ∈ V do
12     d ← min_{(i,q,f)∈I×Q×F} Distances[(s, i), (v, q)] + Distances[(v, q), (t, f)];
13     for q' ∈ GoodStates do
14         d' ← min_{(i,q,f)∈I×Q×F} |w| + Distances[(s', q'), (v, q)] + Distances[(v, q), (t, f)];
15         if d' = d then
16             return true;
   // We consider the covered vertex in w
17 d'' ← min_{(i,q',f)∈I×GoodStates×F} |w| + Distances[(s', q'), (t, f)];
18 for k ∈ 0 ... |w| do
19     v_k ← kth vertex of w;
20     d ← min_{(i,q,f)∈I×Q×F} Distances[(s, i), (v_k, q)] + Distances[(v_k, q), (t, f)];
21     if d'' = d then
22         return true;
23 return false;
```

from v to s' and w' is the prolongation from s' to t. Suppose that v is in the index k of w. Then, $\exists \rho_1, \rho_2, \rho' \in \textsc{Run}_{\mathcal{A}}(D), \exists i \in I, \exists q \in Q, \exists q' \in \delta^*(I, \textsc{Lbl}(w))$ such that $\pi_D(\rho_1) = w_1$, $\pi_D(\rho_2) = w_2$, $\pi_D(\rho') = w'$, $\textsc{Src}(\rho_1) = (s, i)$, $\textsc{Tgt}(\rho_1) = (v, q)$, $\textsc{Src}(\rho_2) = (v, q)$ and $\textsc{Tgt}(\rho_2) = (s', q')$. Since $w \cdot w'$ is a shortest walk covering v we have : $|\rho_1| = \mathit{Distances}[(s, i), (v, q)]$. The fact that the walk can be prolonged to cover v means $\exists f \in F$ that satisfies $|\rho_1 \cdot \rho_2 \cdot \rho'| = \mathit{Distances}[(s, i), (v, q)] + \mathit{Distances}[(v, q), (t, f)]$, which is what the algorithm checks. Thus, the algorithm returns **true** if w is prolongable.

We use similar reasoning to show that if the algorithm returns **true**, then w is prolongable. Therefore, the algorithm is correct. ◀



(a) Case where v is after s'.

(b) Case where v is before s'.

Figure 14 Illustration for the proof of Theorem 80.

Time Complexity

▶ **Theorem 81.** *Prolongability under shortest-coverage semantics can be solved in* $O((|\mathcal{A}| \times |D|)^3)$.

Proof. The time complexity of the algorithm is as follows. Finding the source and target vertices of the walk has a time complexity of $O(|D| \times |\mathcal{A}|)$. Since we showed in Lemma 74 that the length of the a matching walk under shortest-coverage is bounded by $2 \times |V| \times |Q|$, checking the length of the walk can be done in $O(|V| \times |Q|)$. The Floyd-Warshall algorithm has a time complexity of $O((|\mathcal{A}| \times |D|)^3)$ [4]. Finding the smallest distances between the vertices by iterating over $i \in I, q \in Q$ and $f \in F$ has a time complexity of $O(|I| \times |Q| \times |F|)$. We first consider v in the prolongation of w, we calculate the distance for each state that can be reached by reading the label of w (which is in $O(|D| \times |\mathcal{A}|)$). This has a time complexity of $O(|Q| \times |I| \times |F| \times |Q|)$. This is done for each vertex $v \in V$ and hence has a time complexity of $O((|I| \times |Q| \times |F|) \times (|Q| \times |V|))$. We then consider v in w, we calculate the distance for each state that can be reached by reading the label of the edges of w. We do this while iterating over w. This has a time complexity of $O((|I| \times |Q| \times |F|) \times (|Q| \times |w|))$. The complexity in both cases is bounded by the complexity of the Floyd-Warshall algorithm. Therefore, the algorithm has a time complexity of $O((|\mathcal{A}| \times |D|)^3)$. ◀

▶ **Theorem 82.** *Distinct Walks Enumeration under shortest-coverage semantics can be solved in DelayP.*

Proof. We can solve both Prolongability and Walk Membership under shortest-coverage semantics in P (Theorem 81, Theorem 75 respectively). Moreover, len_{max} is in $O(|D| \times |\mathcal{A}|)$ as shown in Lemma 74 and deg_{max} is bounded by $|Q| \times |V|$. Therefore, we can solve Distinct Walks Enumeration under shortest-coverage semantics in DelayP (Theorem 61). ◀

5 Perspectives

In this section, we discuss some open problems that can be addressed in future works.

Enumeration of Walks under Minimal-walk Semantics

Walks Enumeration under minimal-walk semantics is still an open problem. Several approaches have been tested, but none of them guarantee a polynomial-delay. For instance, since minimal-walk semantics is included in simple-run semantics, a natural approach to enumerate the minimal walks would be to enumerate simple-runs and simply filter the results using the Minimal Walk Membership oracle. However, this approach leads to an algorithm that is not in DelayP.

An exponential delay can be observed in the enumeration of minimal walks by the method explained above. For example, consider the graph database D_n in Figure 15 and the query $\mathcal{A}$ in Figure 16.

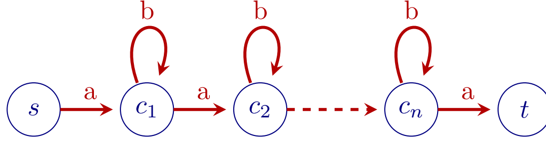


Figure 15 Graph Database D_n with $n + 2$ vertices.

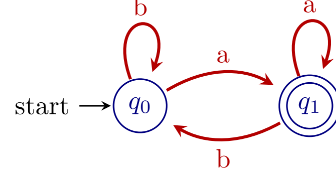


Figure 16 Automaton.

There are 2^n walks in D_n, if each of the n loops is taken at most once. Since none of the loops create a cycle in the product graph $D_n \times \mathcal{A}$, each of the 2^n walks has a simple run. Thus, $[\![\mathcal{A}]\!]_{SR}(D_n)$ contains 2^n walks. For instance, we have the walk $w = (s \xrightarrow{a} c_1 \xrightarrow{a} c_2 \xrightarrow{b} c_2 \xrightarrow{a} \ldots \xrightarrow{a} c_n \xrightarrow{a} t)$ takes the loop on c_2 but none of the other loops.
However, there is only one walk is in $[\![\mathcal{A}]\!]_{MW}(D)$, which is the walk $w = (s \xrightarrow{a} c_1 \xrightarrow{a} c_2 \ldots \xrightarrow{a} c_n \xrightarrow{a} t)$.

▶ **Theorem 83.** *Prolongability under minimal-walk semantics is NP-complete.*

Proof. The problem is in NP, as we can use the prolongation w' of the walk w as a certificate. The certificate is polynomial as the length of a walk under minimal-walk semantics cannot be greater than $|D| \times |\mathcal{A}|$ (as previously mentioned in the proof of Lemma 40, if it is greater than $|D| \times |\mathcal{A}|$, then the run of which it is the projection will contain a cycle, which is a contradiction). Verifying that $w \cdot w' \in [\![\mathcal{A}]\!]_{MW}(D)$ can be done in polynomial time using the Walk Membership under minimal-walk semantics oracle.
To prove the hardness, we use a reduction from the Vertex-Two-Disjoint-Paths problem. Let G, s_1, t_1, s_2 and t_2 be its inputs. We construct D from G by labeling all edges of G with a, adding a self-loop labeled with c to s_1 and adding an edge from t_1 to s_2 labeled with b. Let $\mathcal{A}$ be an automaton that represents the regular expression $ca^*ba^* + a^*$ and w the walk $(s_1 \xrightarrow{c} s_1)$. To simplify notations, we use $\xrightarrow{a^*}$ to denote a walk of any length where all edges are labeled with a. We prove the following implications.

- There exists two disjoint paths from s_1 to t_1 and from s_2 to t_2 $\implies$ w can be prolonged:
  If there are two disjoint paths from s_1 to t_1 and from s_2 to t_2, then we can extend the walk w and get $w \cdot w' = (s_1 \xrightarrow{c} s_1 \xrightarrow{a^*} t_1 \xrightarrow{b} s_2 \xrightarrow{a^*} t_2)$. The walk $w \cdot w'$ is in $[\![\mathcal{A}]\!]_{MW}(D)$ as the only cycle $(s_1 \xrightarrow{c} s_1)$ in the walk cannot be removed.
- There is no such pair of paths $\implies$ w cannot be prolonged:
  If there is no such pair of paths, then any path from s_1 to t_1 and from s_2 to t_2 will have to share at least a vertex. Therefore, the walk w that starts with the label c can only be prolonged by a walk w' that goes through the label b. Thus, w' will contain the two paths from s_1 to t_1 and from s_2 to t_2 separated by the edge labeled with b, and hence necessarily contains a cycle. This means that $w \cdot w'$ will contain at least two cycles: the cycle $(s_1 \xrightarrow{c} s_1)$ and the cycle $(v \xrightarrow{a^*} t_2 \xrightarrow{b} s_2 \xrightarrow{a^*} t_2)$, where v is the shared vertex between the two paths.
  Removing both cycles from $w \cdot w'$ will result in a walk $w'' = (s_1 \xrightarrow{a^*} t_2)$. w'' is a match for the query and $w'' \sqsubset (w \cdot w')$, which means that $(w \cdot w') \notin [\![\mathcal{A}]\!]_{MW}(D)$.
  So, the walk w cannot be prolonged.

We have shown that finding two vertex disjoint paths from s_1 to t_1 and from s_2 to t_2 is equivalent to extending the walk $w = (s_1 \xrightarrow{c} s_1)$ to a walk $w \cdot w'$ such that $w \cdot w' \in$

$[\![\mathcal{A}]\!]_{MW}(D)$. Therefore, Prolongability under minimal-walk semantics is NP-hard and thus NP-complete. ◀

We note that Prolongability under minimal-walk semantics being NP-complete doesn't imply that Walks Enumeration or Distinct Walks Enumeration under minimal-walk semantics are not in DelayP. For instance, there is an algorithm in DelayP for to enumerate maximal cliques in a graph [14], while the prolongability problem (a more generalized version of Prolongability that consist of extending a solution and not necessarily a walk) is NP-complete. The result simply states that a depth-first search approach to solve Distinct Walks Enumeration under minimal-walk semantics is not in DelayP.

Static Analysis

Other problems can be addressed in future works, such as the static analysis of the different semantics. For instance, we have Containment under X semantics. The Containment problem is the problem of checking if the set of answers of a query automaton $\mathcal{A}_1$ is included in the set of answers of another query automaton $\mathcal{A}_2$ for every database. Recently, Nadime Francis has shown that Containment under simple-run semantics is undecidable using a reduction from Post's Correspondence Problem. The proof is yet to be published. Although the proof cannot be adapted directly, it may lead to believing that Containment under minimal-walk semantics can also be undecidable.

6 Conclusion

In this paper, we introduced and studied two new RPQ semantics for graph databases: minimal-walk and shortest-coverage semantics. Shortest-coverage semantics is as good as shortest-walk semantics in terms of computational problems complexity, while offering more results by considering the vertex coverage of the graph database. Minimal-walk semantics has good computational properties in terms of existence and membership problems, but the enumeration problems are still open.

Problem	**Trail**	**Shortest-walk**	**Simple-run**	**Minimal-walk**	**Shortest-coverage**
Tuple Membership	NP-compl.	NL-compl.	NL-compl.	NL-compl.	NL-compl.
Walk Membership	NL-compl.	NL-compl.	NP-compl.	P	P
Walks Enumeration	$\notin$ DelayP	DelayP	DelayP	Open	DelayP
Distinct Walks Enum.	$\notin$ DelayP	NextP	Open	Open	DelayP

Table 2 Summary of the computational problems' complexity for different semantics.

Table 2 summarizes the complexity of the computational problems for the different semantics. Surprisingly enough, good computational properties don't imply that the semantics is useful in practice. For instance, trail semantics is the most used in practice, as previously mentioned, but it is the most complex in terms of computational problems. This accentuates the need for a framework that situates any given semantics with criteria other than the complexity of computational problems.

References

[1] Renzo Angles et al. "Foundations of Modern Graph Query Languages". In: (2016). URL: http://arxiv.org/abs/1610.06264.

[2] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages". In: *SIGMOD'18*. ACM, 2018. ISBN: 9781450347037. DOI: 10.1145/3183713.3190654.

[3] Florent Capelli and Yann Strozecki. "Incremental delay enumeration: Space and time". In: *Discrete Applied Mathematics* 268 (2019), pp. 179–190. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2018.06.038.

[4] T.H. Cormen et al. *Introduction to Algorithms, third edition*. Computer science. MIT Press, 2009. ISBN: 9780262033848. URL: https://books.google.fr/books?id=i-bUBQAAQBAJ.

[5] Isabel F. Cruz, Alberto O. Mendelzon and Peter T. Wood. "A graphical query language supporting recursion". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: Association for Computing Machinery, 1987, pp. 323–330. ISBN: 0897912365. DOI: 10.1145/38713.38749. URL: https://doi.org/10.1145/38713.38749.

[6] Claire David, Nadime Francis and Victor Marsault. "Distinct Shortest Walk Enumeration for RPQs". In: *Proceedings of the ACM on Management of Data* 2.2 (2024), pp. 1–22.

[7] Claire David, Nadime Francis and Victor Marsault. "Run-Based Semantics for RPQs". In: *Principles of Knowledge Representation and Reasoning (KR'23)*. 2023. DOI: 10.24963/kr.2023/18.

[8] Alin Deutsch et al. "TigerGraph: A Native MPP Graph Database". In: abs/1901.08248 (2019). arXiv: 1901.08248. URL: http://arxiv.org/abs/1901.08248.

[9] Steven Fortune, John Hopcroft and James Wyllie. "The directed subgraph homeomorphism problem". In: *Theoretical Computer Science* 10.2 (1980), pp. 111–121. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(80)90009-2. URL: https://www.sciencedirect.com/science/article/pii/0304397580900092.

[10] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657.

[11] Nadime Francis et al. "GPC: A Pattern Calculus for Property Graphs". In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 241–250. ISBN: 9798400701276. DOI: 10.1145/3584372.3588662. URL: https://doi.org/10.1145/3584372.3588662.

[12] Zan Huang et al. "A graph-based recommender system for digital library". In: *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*. JCDL '02. Portland, Oregon, USA: Association for Computing Machinery, 2002, pp. 65–73. ISBN: 1581135130. DOI: 10.1145/544220.544231. URL: https://doi.org/10.1145/544220.544231.

[13] International Organization for Standardization. *GQL*. Standard. Apr. 2024. URL: https://www.iso.org/standard/76120.html.

[14] E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan. “Generating All Maximal Independent Sets: NP-Hardness and Polynomial-Time Algorithms”. In: *SIAM Journal on Computing* 9.3 (1980), pp. 558–565. DOI: 10.1137/0209042. eprint: https://doi.org/10.1137/0209042. URL: https://doi.org/10.1137/0209042.

[15] Alberto O. Mendelzon and Peter T. Wood. “Finding Regular Simple Paths in Graph Databases”. In: *SIAM Journal on Computing* 24.6 (1995), pp. 1235–1258. DOI: 10.1137/S009753979122370X. eprint: https://doi.org/10.1137/S009753979122370X. URL: https://doi.org/10.1137/S009753979122370X.

[16] Alberto O. Mendelzon and Peter T. Wood. “Finding Regular Simple Paths in Graph Databases”. In: *SIAM Journal on Computing* 24.6 (1995), pp. 1235–1258. DOI: 10.1137/S009753979122370X. eprint: https://doi.org/10.1137/S009753979122370X. URL: https://doi.org/10.1137/S009753979122370X.

[17] Sherif Sakr et al. “The future is big graphs: a community view on graph processing systems”. In: *Commun. ACM* 64.9 (Aug. 2021), pp. 62–71. ISSN: 0001-0782. DOI: 10.1145/3434642. URL: https://doi.org/10.1145/3434642.

[18] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory, Volume I: Languages and Parsing*. EATCS Monographs on Theoretical Computer Science. Springer, 1988.

[19] Yann Strozecki. *Enumeration Complexity: Incremental Time, Delay and Space*. 2023. arXiv: 2309.17042 [cs.CC].

[20] Santiago Timón-Reina, Mariano Rincón and Rafael Martínez-Tomás. “An overview of graph databases and their applications in the biomedical domain”. In: *Database* 2021 (May 2021), baab026. ISSN: 1758-0463. DOI: 10.1093/database/baab026. eprint: https://academic.oup.com/database/article-pdf/doi/10.1093/database/baab026/37952095/baab026.pdf. URL: https://doi.org/10.1093/database/baab026.

[21] Jin Y. Yen. “Finding the K Shortest Loopless Paths in a Network”. In: *Management Science* 17.11 (1971), pp. 712–716. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2629312 (visited on 02/06/2024).

A Appendix A: Almost a Semantics

We can note that using the **direct strict subwalks** obtained by one application of the $\ll$ operator is not sufficient to determine if a walk is minimal with respect to the $\sqsubseteq$ order. For example, consider the graph database in Figure 17 and the query in Figure 18. The walk $w = (s \xrightarrow{a} c1 \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c2 \xrightarrow{a} t)$ is a match. After one application of the $\ll$ operator, we get the walk $w' = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} c2 \xrightarrow{a} t)$ that is not a match. However, w is not a minimal walk since we have $w'' = (s \xrightarrow{a} c1 \xrightarrow{a} c2 \xrightarrow{a} t)$ that is also a match for the query and is obtained after two applications of the $\ll$ operator.

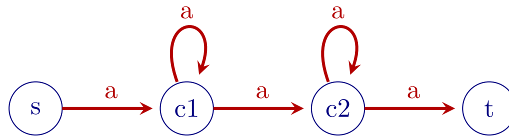


Figure 17 Graph Database D_5.

One may find it interesting to define a variant of the minimal-walk semantics, which relies on a different definition of minimality restricted to only direct strict subwalks i.e. a walk

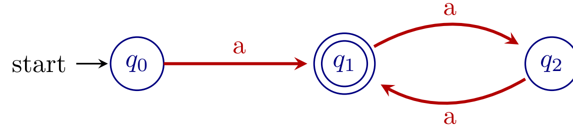


Figure 18 Automaton for Regular Expression Query: $(a(aa)^*)$.

is minimal if it doesn't have a direct strict subwalk that is a match. We will refer to this variant as **direct minimal-walk semantics**.

We can define a polynomial algorithm to resolve the walk membership problem of this semantics. The algorithm 10 takes a walk w, a query $\mathcal{A}$ and a graph database D as inputs and returns a boolean value indicating if $w \in [\![\mathcal{A}]\!]_{DirectMW}(D)$, i.e. if the walk is returned under this variant of minimal-walk semantics.

We start by creating $|w|$ sets of accessibility states such that each E_i contains the states that can be reached from the initial state q_0 after i steps in w. We then check if the set of states E_n contains a state in the set of final states F (as previously done in Algorithm 2).

If it does, we continue by creating $|w|$ sets of co-accessible states such that each coE_i that contains the states from which the final states can be reached after i steps in w.

For each pair of indices (i, j) in the list of possible removed continuous sequences L, we make use of the sets we have created and we check if $E_{i-1} \cap coE_j \neq \emptyset$. If $\exists (i, j) \in L$ such that $E_{i-1} \cap coE_j \neq \emptyset$, then we are able to find a computation of the automaton that matches w after removing the sequence i.e a subwalk of w that is a match. Therefore $w \notin [\![\mathcal{A}]\!]_{DirectMW}(D)$. Conversely, if $\forall (i, j) \in L$, $E_{i-1} \cap coE_j = \emptyset$ and $E_n \cap F \neq \emptyset$, then $w \in [\![\mathcal{A}]\!]_{DirectMW}(D)$.

The construction of the sets E_i and coE_i has a time complexity of $O(|w| \times |\mathcal{A}|)$ (explained in the proof of 19). The maximum number of cycles in a walk w is $|w|^2$. Therefore, the time complexity of the algorithm is $O(|w| \times |\mathcal{A}| + |w|^2)$.

Note that, this variant is not a semantics as it does not result in a finite bag of results. Interestingly enough, the language it produces is not even be regular. Let's consider the graph database in Figure 19 and the query $(a + b + c)^*$, where we can observe the issue. The

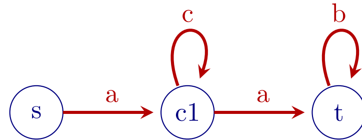


Figure 19 Graph Database D_6.

language $\mathcal{L} = \{ac^n ab^n \mid n > 1\} \cup \{aaa\}$ produced by this semantics is not regular nor finite.

Algorithm 10 Walk Membership Under **Direct** Minimal-walk Semantics

Data: w, $\mathcal{A}$, D

Result: $w \in [\![\mathcal{A}]\!]_{DirectMW}(D)$?

```
1  if w ∉ Match_A(D) then
2  │  return false
3  L ← FindCycles(w);
4  if L = ∅ then
5  │  return true
6  E_0 ← {q_0};
7  for i ← 1 to n do
8  │  for q ∈ E_{i−1} do
9  │  │  E_i ← E_i ∪ δ(q, w_i)
10 if E_n ∩ F = ∅ then
11 │  return false
12 coE_n ← F;
13 for i ← n to 1 do
14 │  for q ∈ coE_{i+1} do
15 │  │  coE_i ← coE_i ∪ δ^{−1}(q, w_i)
16 for (i, j) ∈ L do
17 │  if E_i − 1 ∩ coE_j ≠ ∅ then
18 │  │  return false
19 return true
```