Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Run-Based Semantics for RPQs

Claire David , Nadime Francis , Victor Marsault
Université Gustave Eiffel, CNRS, LIGM, France
{claire.david, nadime.francis, victor.marsault}@univ-eiffel.fr

Abstract

RPQs (regular path queries) are an important building block of most query languages for graph databases. They are generally evaluated under homomorphism semantics; in particular only the endpoints of the matched walks are returned.

However, practical applications often need the full matched walks to compute aggregate values. In those cases, homomorphism semantics are not suitable since the number of matched walks can be infinite. Hence, graph-database engines adapt the semantics of RPQs, often neglecting theoretical red flags. For instance, the popular query language Cypher uses trail semantics, which ensures the result to be finite at the cost of making computational problems intractable.

We propose a new kind of semantics for RPQs, including in particular simple-run and binding-trail semantics, as a candidate to reconcile theoretical considerations with practical aspirations. Both ensure the output to be finite in a way that is compatible with homomorphism semantics: projection on endpoints coincides with homomorphism semantics. Hence, testing the emptiness of result is tractable, and known methods readily apply. Moreover, simple-run and binding-trail semantics support bag semantics, and enumeration of the bag of results is tractable.

1 Introduction

When querying data graphs, users are not only interested in retrieving data, but also in *how* these pieces of data relate to each other. This is why most languages for querying data graphs, both in theory and in practice, are *navigational* languages. Informally, the querying process starts at some vertex and then *walks* through the graph: it follows edges from vertex to vertex, retrieving and testing data along the way, until the walk ends in some final vertex.

In database theory, this process is usually abstracted as Regular Path Queries (RPQs, Cruz, Mendelzon, and Wood 1987). An RPQ is defined by a regular expression R and is traditionally evaluated under *walk semantics* (also known as *homomorphism semantics*, Angles et al. 2017). In that case, it returns all pairs of vertices in the graph that are linked by a walk whose label conforms to R. This formalism enjoys many nice properties and has become an important building block of most query languages over graph databases.

However, RPQs do not entirely meet the needs of real-life graph database systems. Indeed, limiting the output of

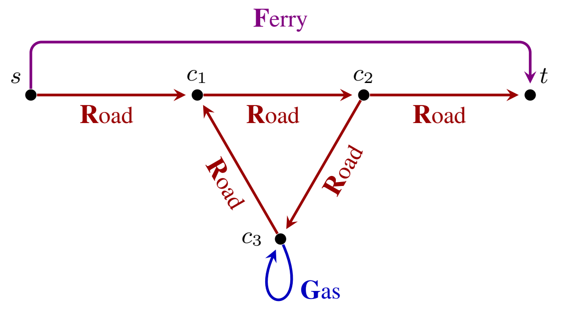


Figure 1: Our running graph database D

the query to the endpoints of the walk is not enough for many real-life applications, which might also require the number of matching walks (e.g. to rank answers or evaluate connectivity), or even the walks themselves (e.g. for route planning) (Robinson, Webber, and Eifrem 2015). Under walk semantics, the *space of matches* is infinite: there are infinitely many matching walks when the graph contains cycles, which renders these questions meaningless. Most graph database management systems have their own way of addressing this issue, with none of them being entirely satisfactory. We briefly describe the most common approaches below, as well as their shortcomings; we use the database given in Figure 1 and the following queries to illustrate them.

$$Q_1 = (\mathbf{R}\text{oad} + \mathbf{F}\text{erry})^*$$
$$Q_2 = (\mathbf{R}\text{oad} + \mathbf{F}\text{erry})^*\ \mathbf{G}\text{as}\ (\mathbf{R}\text{oad} + \mathbf{F}\text{erry})^*$$

Topological restriction This solution roots out unboundedness by forbidding cycles. Walks are only returned if no vertex (*simple-walk semantics*) or no edge (*trail semantics*) is visited twice. For instance, the language Cypher uses trail semantics (Francis et al. 2018). Moreover, the new query language GQL[1](Deutsch et al. 2022; Francis et al. 2023), currently designed by ISO[2] to be the first standard language for property graphs, implements several topological restrictions.

The output walks are in some sense representative of the possibilities in the space of matches. For instance, Q_1

[1]https://www.iso.org/standard/76120.html
[2]ISO stands for International Standards Organisation.

returns $s \to t$ and $s \to c_1 \to c_2 \to t$ under trail semantics, two unrelated possibilities in the space of matches. This is a crucial feature in real systems, in which pattern matching is usually just a first step before further processing. For instance, one could evaluate the connectivity between s and t by counting the number of walks from s to t matching Q_1.

The main weakness of this approach is that computational problems are intractable even for very simple queries. For instance, deciding whether two vertices are linked by a walk conforming to Q_2 is NP-complete in the size of the database under both trail (Martens, Niewerth, and Trautner 2020) and simple-walk semantics (Bagan, Bonifati, and Groz 2020). These semantics are also error-prone in that desirable results might be discarded unintentionally. For instance, Q_2 returns no walk from s to t under trail semantics; and in a bigger, more realistic, graph database the walk $s \to c_1 \to c_2 \to c_3 \to c_1 \to c_2 \to t$ would not be considered for further processing. These kinds of unwanted behaviours happen beyond theoretical settings: e.g., in (Robinson, Webber, and Eifrem 2015, p.132), the authors propose a query to solve a real-life scenario; the query does not work as intended due to trail semantics.

Witness selection Another approach consists in choosing a metric (length, cost, etc.), and then selecting only a few best-ranking walks in the space of matches. For instance, the semantics of GSQL (TigerGraph Team 2021; Deutsch et al. 2019) and G-Core (Angles et al. 2018) only return the shortest walks matching the query; GQL allows returning the k shortest walks (Deutsch et al. 2022). However, the length of the path is an arbitrary metric that may not fit every application: here, Q_1 would return the ferry route $v = s \to t$ over the road route $w = s \to c_1 \to c_2 \to t$ but it does not necessarily mean that v represents a faster or shorter route than w in reality. To circumvent this issue, GQL mentions other metrics, such as k-cheapest, as possible extensions to investigate. On the other hand, witness selection generally makes counting and aggregating meaningless: it counts or aggregates over something that is not representative of the space of matches.

Reducing expressivity Some systems disallow queries or operations that may lead to infinite outputs or ill-defined behaviours. Kleene stars in GQL queries are only allowed if they appear under some form of topological restriction or witness selection. In SPARQL[3], counting the number of walks matched by a property path is only allowed when the underlying regular language is finite. Otherwise, the returned number collapses to 0 (no walk matches the query) or 1 (at least one walk matches the query). Similarly, SPARQL equivalents of queries Q_1 or Q_2 only return the endpoints of matched walks. Note also that switching silently from bag to set semantics depending on the query is error-prone.

In this article, we propose another approach called *run-based*. We present two run-based semantics: *simple-run semantics*, whose input query is given as a finite automaton and provide sound theoretical foundations; and *binding-trail semantics* which operate directly on a regular expression in order to be closer to practical use. Akin to topological restriction, we aim at producing a finite output that faithfully represents the space of matches, and we do so by discarding cyclic results. Intuitively, run-based semantics discard a result only if a cycle in the walk coincides with a cycle in the computation of the query. For instance, binding-trail semantics filter out walks in which one edge is matched twice to the same atom of the regular expression. Indeed, the walk $w = s \to c_1 \to c_2 \to c_3 \to c_1 \to c_2 \to t$ is **not** in the output of Q_1: the edge $c_1 \to c_2$ is matched twice to the same Road atom. On the other hand, w is kept in the output of Q_2, because the two occurrences of the edge $c_1 \to c_2$ are matched to two different Road atoms. In general, the output under run-based semantics depends on the *syntax* of the query. This seeming drawback also provides a finer control of the output; see Remarks 14 and 30.

The paper is organised as follows. Section 2 covers necessary preliminaries and Section 3 gives the definition of simple-run semantics. In Section 4, we revisit classical computational problems and show that simple-run semantics enjoy efficient PTIME or polynomial-delay algorithms for emptiness, tuple membership and walk enumeration. Counting answers remains #P-Complete. Section 5 defines binding-trail semantics as an adaptation of simple-run semantics to queries given as regular expressions. As a side result, we show that any regular expression (in fact, its Glushkov automaton) may encode the same behaviour and topology as any arbitrary automaton, which means that all complexity lower and upper bounds translate from one setting to the other. Finally, we conclude this document in Section 6 by discussing possible extensions of our semantics.

2 Preliminaries

2.1 Graph Databases

In this document, we model graph databases as directed, multi-labeled, multi-edge graphs, and simply refer to them as *databases* for short. We will use the database shown in Figure 1, page 1, as our running example. Databases are formally defined as follows.

Definition 1. *A (graph)* 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; and $\textsc{Lbl} : E \to 2^{\Sigma}$ is the* labelling *function.*

Definition 2. *A (directed)* walk *w in D is a non-empty finite sequence of alternating vertices and edges of the form $w = (n_0, e_0, n_1, \ldots, e_{k-1}, n_k)$ where $k \geq 0$, $n_0, \ldots, n_k \in V$, $e_0, \ldots, e_{k-1} \in E$, such that:*

$$\forall i, 0 \leq i < k, \quad \textsc{Src}(e_i) = n_i \quad \text{and} \quad \textsc{Tgt}(e_i) = n_{i+1}$$

For ease of notation, we use $\to$ to avoid naming the edge that connects two nodes when it is unique, as in $w = n_0 \to n_1 \to \cdots \to n_k$.

[3]https://www.w3.org/TR/sparql11-query/#propertypaths

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

$$\textsc{Endpoints}(w) = \big(\textsc{Src}(w), \textsc{Tgt}(w)\big)$$

$$\textsc{Lbl}(w) = \big\{\, u_0 u_1 \cdots u_{k-1} \,\big|\, \forall i, 0 \le i < k,\ u_i \in \textsc{Lbl}(e_i) \,\big\} .$$

Finally, $s \xrightarrow{w} t$ *means that* $\textsc{Endpoints}(w) = (s, t)$ *and, for a word $u \in \Sigma^*$, we write* $s \xrightarrow{u} t$ *if there exists a walk w in D such that* $s \xrightarrow{w} t$ *and* $u \in \textsc{Lbl}(w)$.

We say that two walks w, w' concatenate *if* $\textsc{Tgt}(w) = \textsc{Src}(w')$, *in which case we define their* concatenation *as usual, and denote it by $w \cdot w'$, or simply ww' for short.*

Definition 3. *A* trail *is a walk with no repeated edge. A* simple walk *is a walk with no repeated vertex. We let* $\textsc{Trail}$ *(resp.* $\textsc{Simple}$*) denote the bag-to-bag function that takes as input a bag of walks B and returns the bag of the trails (resp. simple walks) in B.*

2.2 Regular Path Queries, Automata, Expressions

An RPQ is defined by a regular language (given as either an automaton or a regular expression). RPQs may be evaluated under various semantics. Several classical semantics, along with the novel *run-based semantics*, are defined in Section 3.

A (nondeterministic) automaton is a 5-tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, I, F \rangle$ where Σ is a finite set of symbols, Q is a finite set of *states*, $I \subseteq Q$ is called the set of *initial* states, $\Delta \subseteq Q \times \Sigma \times Q$ is the set of *transitions* and $F \subseteq Q$ is the set of *final* states. As usual, we extend Δ into a relation over $Q \times \Sigma^* \times Q$ as follows: for every $q \in Q$, $(q, \varepsilon, q) \in \Delta$; and for every $q, q', q'' \in Q$ and every $u, v \in \Sigma^*$, if $(q, u, q') \in \Delta$ and $(q', u, q'') \in \Delta$ then $(q, uv, q'') \in \Delta$. We denote by $\mathrm{L}(\mathcal{A})$ the *language* of $\mathcal{A}$, defined as follows.

$$\mathrm{L}(\mathcal{A}) = \big\{\, u \in \Sigma^* \,\big|\, \exists i \in I,\ \exists f \in F,\ (i, u, f) \in \Delta \,\big\} \quad (1)$$

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 *successful* if it starts in an initial state and ends in a final state.

A regular expression R over an alphabet Σ is a formula obtained inductively from the letters in Σ, one unary function *, and two binary functions $+$ and $\cdot$, according to the following grammar.

$$R ::= \varepsilon \mid a \mid R^* \mid R \cdot R \mid R + R \qquad \text{where } a \in \Sigma \quad (2)$$

We usually omit the $\cdot$ operator and we let $L(R)$ denote the subset of Σ^* described by R.

3 Run-Based Query Evaluation

3.1 Run Database

In Section 3.1, we fix an automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, I, F \rangle$ and a graph database $D = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$.

Definition 4. *The* run database $D \times \mathcal{A}$ *is the database* $D \times \mathcal{A} = (\Sigma, V', E', \textsc{Src}', \textsc{Tgt}', \textsc{Lbl}')$ *where*

$$V' = V \times Q$$

$$E' = \big\{\, (e, (q, a, q')) \in E \times \Delta \,\big|\, a \in \textsc{Lbl}(e) \,\big\}$$

and, for each $e' = (e, (q, a, q')) \in E'$,

$$\textsc{Src}'(e') = \big(\textsc{Src}(e), q\big) \quad \textsc{Tgt}'(e') = \big(\textsc{Tgt}(e), q'\big)$$

$$\textsc{Lbl}'(e') = \{\, a \,\} \quad .$$

We denote the projection from $D \times \mathcal{A}$ to D by π_D: for each $(n, q) \in V'$, $\pi_D((n, q)) = n$; for each $(e, t) \in E'$, $\pi_D((e, t)) = e$; and for each walk $w = (n_0, e_0, \dots, n_k)$, $\pi_D(w) = (\pi_D(n_0), \pi_D(e_0), \dots, \pi_D(n_k))$.

The run database is essentially a product of the automaton with the database. See Figure 2 for an example. In the figure, elements that do not contribute to any run are dashed.

Definition 5. *A walk w in $D \times \mathcal{A}$ is called* a run *if* $\textsc{Src}(w) \in V \times I$ *and* $\textsc{Tgt}(w) \in V \times F$. *We let* $\textsc{Match}_{\mathcal{A}}(D)$ *denote the bag*[4] *of all runs in $D \times \mathcal{A}$.*

A simple verification yields the following property.

Property 6. *For every walk w in D, there exists a run r in $D \times \mathcal{A}$ such that $\pi_D(r) = w$ if and only if* $\textsc{Lbl}(w) \cap \mathrm{L}(\mathcal{A}) \neq \emptyset$.

Remark 7. *Note that the run database $D \times \mathcal{A}$ depends on the structure of the automaton $\mathcal{A}$, and not only on $\mathrm{L}(\mathcal{A})$. Hence, when considering run databases, we cannot assume that $\mathcal{A}$ is deterministic, minimal, or of any particular shape.*

The run database allows rephrasing the most common semantics, as in Definitions 8, 9 and 10.

Definition 8. *Under* walk semantics, *RPQs return all walks of the input database whose label conforms to the query. It is defined as* $[\![\mathcal{A}]\!]_W(D) = \pi_D \circ \textsc{Match}_{\mathcal{A}}(D)$.

We will sometimes refer to the bag $[\![\mathcal{A}]\!]_W(D)$ as the *space of matches*, as it contains all walks that intuitively match the query. Note that it can be infinite, and thus cannot be returned as is. The following two semantics circumvent this issues by restricting $[\![\mathcal{A}]\!]_W(D)$ to a finite bag.

Definition 9. Trail semantics *return only the trails matching the query :* $[\![\mathcal{A}]\!]_T(D) = \textsc{Trail} \circ \pi_D \circ \textsc{Match}_{\mathcal{A}}(D)$.

Definition 10. Simple-walk semantics *return only the matching walks that are simple:* $[\![\mathcal{A}]\!]_{SW}(D) = \textsc{Simple} \circ \pi_D \circ \textsc{Match}_{\mathcal{A}}(D)$.

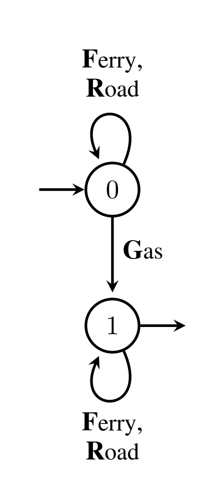
3.2 Simple-Run Semantics

In line with simple-walk and trail semantics, *simple-run semantics* keeps the output finite by filtering out *redundant* results. The difference amounts to the definition of *redundant*. Classical semantics filter based on redundancy in the computed walk (repeated edge, repeated vertex), hence filtering is done *after* projecting the runs to D. In the semantics we propose here, filtering is based on redundancy in the run, hence filtering is done *before* projecting to D.

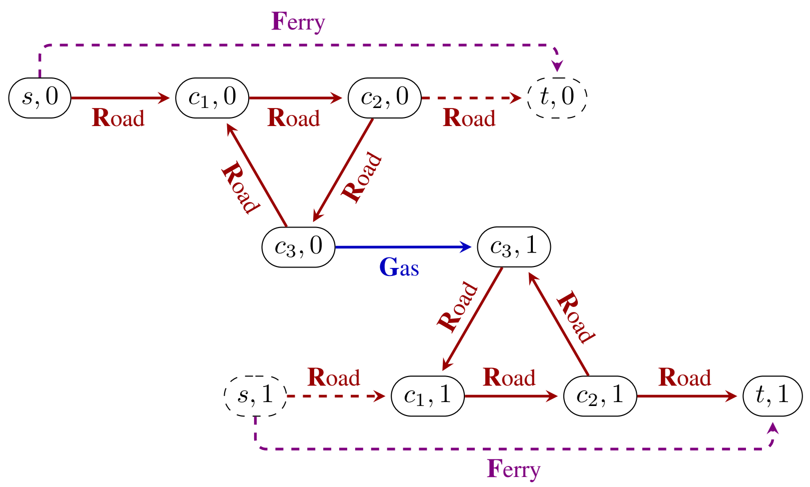
Definition 11. *The* simple-run semantics *of an automaton $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]_{SR}$, is the mapping that associates, to each database D, the following bag of answers.*

$$[\![\mathcal{A}]\!]_{SR}(D) = \pi_D \circ \textsc{Simple} \circ \textsc{Match}_{\mathcal{A}}(D) \quad (3)$$

[4]Although the multiplicity of each element in $\textsc{Match}_{\mathcal{A}}(D)$ is one, we would rather not use the term *set* to avoid confusion when we apply bag-to-bag functions later on.



(a) An automaton $\mathcal{A}$



(b) The run database $D \times \mathcal{A}$

Figure 2: A run database constructed from D (Figure 1) and $\mathcal{A}$ (Figure 2a).

Example 12. *A run in the database from Figure 2b is a walk that goes from the top part to the bottom part. For instance, the walk* $r_1 = (s,0) \to (c_1,0) \to (c_2,0) \to (c_3,0) \to (c_3,1) \to (c_1,1) \to (c_2,1) \to (t,1)$ *is a run, which moreover is simple. Hence its projection* $w_1 = \pi_D(r_1) = s \to c_1 \to c_2 \to c_3 \to c_3 \to c_1 \to c_2 \to t$ *belongs to* $[\![\mathcal{A}]\!]_{SR}(D)$*. On the other hand,* w_1 *is neither a trail nor a simple walk, hence* $w_1 \notin [\![\mathcal{A}]\!]_T(D)$ *and* $w_1 \notin [\![\mathcal{A}]\!]_{SW}(D)$*. In fact,* $[\![\mathcal{A}]\!]_T(D)$ *and* $[\![\mathcal{A}]\!]_{SW}(D)$ *contain no walk going from* s *to* t*.*

One of the main features of simple-run semantics is that it covers the space of matches, in a precise way (Lemma 13). Essentially, if a walk w matching the query is **not** returned, at least one *subwalk* w' of w is returned; moreover, w' is obtained from w by removing superfluous cycles. Note that semantics based on topological restriction do not enjoy the same property, as shown in Example 12.

Lemma 13. *Let* D *be a database,* $\mathcal{A}$ *be an automaton, and* w *be a walk in* $\text{MATCH}_{\mathcal{A}}(D)$*. Then, there exists a decomposition of* w *as* $w = u_1 v_1 u_2 \cdots v_n u_{n+1}$ *such that every* u_i *satisfies* $\text{SRC}(u_i) = \text{TGT}(u_i)$*, and* $v_1 \cdots v_n \in [\![\mathcal{A}]\!]_{SR}(D)$*.*

Proof. By induction on the length of w. The statement obviously holds if w is a single vertex since a walk of length 0 is always simple.

Let $w \in \text{MATCH}_{\mathcal{A}}(D)$. Let r be a run in $D \times \mathcal{A}$ such that $\pi_D(r) = w$. If $w \in [\![\mathcal{A}]\!]_{SR}(D)$, there is nothing to prove: fix $n = 1$, u_1, u_2 as single vertices and $v_1 = w$. Otherwise, it means that r is not simple, that is there is a decomposition of r as $r = r_1 r_2 r_3$ such that $\text{TGT}(r_1) = \text{SRC}(r_2) = \text{TGT}(r_2) = \text{SRC}(r_3)$ and $\text{LEN}(r_2) \neq 0$. Hence, $r_1 r_3$ is a run in $D \times \mathcal{A}$ and the walk $w' = \pi_D(r_1 r_3)$ belongs to $\text{MATCH}_{\mathcal{A}}(D)$. Then, we conclude by induction on w' and reconstruct the decomposition of w. □

Remark 14. *Recall that the run database depends on the automaton itself (Remark 7). This dependence carries over to simple-run semantics:* $[\![\mathcal{A}]\!]_{SR}(D)$ *and* $[\![\mathcal{B}]\!]_{SR}(D)$ *might be different even if* $\text{L}(\mathcal{A}) = \text{L}(\mathcal{B})$*. Choosing* $\mathcal{A}$ *or* $\mathcal{B}$ *governs which representatives of the space of matches are returned, in the sense of Lemma 13.*

Remark 15. *Akin to simple-run semantics, one could define* trail-run semantics *that would return the* trails *of the run database. While trail-run semantics would generally enjoy the same properties as simple-run semantics, the meaning of a trail in the run database is much harder to grasp. Indeed, transitions of the automaton usually have no intrinsic meaning, whereas states encode the content of the memory.*

4 Computational Problems

In Section 4, we restate common computational problems related to query answering. We recall known results for the usual semantics, and give both lower and upper complexity bounds for simple-run semantics.

4.1 Existence of a Matching Walk

The problem TUPLE MEMBERSHIP consists in deciding whether there is a walk matching the query between two given endpoints. Under walk semantics, this problem corresponds to what is called *homomorphism semantics* in most theoretical contexts, hence it is unsurprisingly tractable in that case (Theorem 16). On the other hand, TUPLE MEMBERSHIP is intractable under trail or simple-walk semantics (Theorem 17). We show in Theorem 18 that it is tractable under simple-run semantics.

TUPLE MEMBERSHIP UNDER X SEMANTICS

- Data: A database D, and a pair (s,t) of vertices in D.
- Query: An automaton $\mathcal{A}$.
- Question: Does there exist a walk $w \in [\![\mathcal{A}]\!]_X(D)$ such that $\text{ENDPOINTS}(w) = (s,t)$?

Theorem 16 (Mendelzon and Wood 1995)**.** TUPLE MEMBERSHIP *is NL-complete under walk semantics.*

Theorem 17 (Martens, Niewerth, and Trautner 2020; Bagan, Bonifati, and Groz 2020)**.** TUPLE MEMBERSHIP *is NP-complete under trail or simple-walk semantics. It is already NP-hard for a fixed query in both cases.*

The typical query for which TUPLE MEMBERSHIP is hard under trail semantics is a^*ba^*. Indeed, one has to record which edges are matched by the left a^*, in order not to be matched by the right a^*. Under simple-run semantics it is not necessary to keep that record, which makes TUPLE MEMBERSHIP tractable as stated by Theorem 18.

Theorem 18. TUPLE MEMBERSHIP *is NL-complete under simple-run semantics.*

Theorem 18 is a corollary of Proposition 19 below, which is itself a direct consequence of Lemma 13.

Proposition 19. *Let D be a database, $\mathcal{A}$ be an automaton, and s, t be two vertices in D. We let $P_{s,t}$ denote the set $P_{s,t} = \{ w \in [\![\mathcal{A}]\!]_W(D) \mid \text{ENDPOINTS}(w) = (s,t) \}$. Each walk with minimal length in $P_{s,t}$ belongs to $[\![\mathcal{A}]\!]_{SR}(D)$.*

Proposition 19 implies that simple-run semantics and walk semantics are equivalent for TUPLE MEMBERSHIP. Hence known techniques for computing TUPLE MEMBERSHIP efficiently under walk semantics readily apply to simple-run semantics; and shortest-walk algorithms can be used to produce witnesses for TUPLE MEMBERSHIP.

4.2 Enumeration of Matching Walks

The problem QUERY EVALUATION consists in enumerating the walks returned by the query. It is perhaps the most important computational problem regarding query answering since it is close to what database engines do in practice. QUERY EVALUATION is ill-defined under walk semantics since $[\![\mathcal{A}]\!]_W(D)$ might be infinite[5]. Under trail or simple-walk semantics, QUERY EVALUATION is well-defined but it is intractable (Theorem 20). By using Yen's algorithm, we show that it is tractable under simple-run semantics.

QUERY EVALUATION UNDER X SEMANTICS
• Data: A database D.
• Query: An automaton $\mathcal{A}$.
• Output: All walks in $[\![\mathcal{A}]\!]_X(D)$.

Theorem 20. *Unless P = NP,* QUERY EVALUATION *under trail or simple-walk semantics cannot be enumerated with polynomial-time preprocessing.*

Theorem 20 follows easily from Theorem 17.

Theorem 21. QUERY EVALUATION *under simple-run semantics can be enumerated with polynomial delay and preprocessing.*

Sketch of proof. Computing $[\![\mathcal{A}]\!]_{SR}(D)$ amounts to computing all simple walks from (s,i) to (t,f) in the run database $D \times \mathcal{A}$, for each vertices s and t of D and each initial and final states i and f. This can be done for each (s,i) and (t,f) by using classical algorithms for simple-walk enumeration, such as Yen's algorithm (see Yen 1971; or Martens, Niewerth, and Trautner 2020 for a modern statement). □

[5]It might be of interest to define (non-terminating) enumeration procedure for this infinite bag. It is beyond the scope of this paper.

QUERY EVALUATION enumerates the walks in $[\![\mathcal{A}]\!]_X(D)$, which is a bag. Hence a walk in $[\![\mathcal{A}]\!]_X(D)$ with multiplicity m will be output m times. We call DEDUPLICATED QUERY EVALUATION the problem that enumerates the **distinct** matching walks.

DEDUPLICATED QUERY EVAL. UNDER X SEM.
• Data: A database D.
• Query: An automaton $\mathcal{A}$.
• Output: All walks in $[\![\mathcal{A}]\!]_X(D)$, without duplicates.

Note that Theorem 20 also holds for DEDUPLICATED QUERY EVALUATION for similar reasons. We leave its complexity under simple-run semantics as an open problem.

4.3 Counting Matching Walks

Counting the number of matching walks between two vertices, or TUPLE MULTIPLICITY, is also used in practice, for instance to evaluate the connectivity between two vertices. TUPLE MULTIPLICITY behaves differently under walk semantics, as some tuples might have infinite multiplicity. Under the variants based on witness selection (e.g. shortest walk semantics, as explained in the introduction), the problem takes a different meaning and no longer reflects the level of connectivity between vertices. Under trail or simple-walk semantics, this problem is known to be intractable; the same technique shows that it is also intractable under simple-run semantics (Theorem 22).

TUPLE MULTIPLICITY UNDER X SEMANTICS
• Data: A database D, and a pair (s,t) of vertices in D.
• Query: An automaton $\mathcal{A}$.
• Output: The total multiplicity of all walks $w \in [\![\mathcal{A}]\!]_X(D)$ such that $\text{ENDPOINTS}(w) = (s,t)$.

Theorem 22. TUPLE MULTIPLICITY *is #P-complete under trail, simple-walk and simple-run semantics. It is already #P-hard in data complexity: there exists a fixed automaton $\mathcal{A}$ for which the problem is #P-hard.*

In all cases, the upper bound comes from counting the successful computations of a nondeterministic polynomial-time machine that guesses a trail (resp. simple walk, resp. simple run) going from s to t and checks that it is accepted by $\mathcal{A}$. The hardness proof consists in a reduction from counting trails (or simple walks) in unlabelled graphs, two problems known to be #P-complete (Valiant 1979). One simply has to fix $\mathcal{A}$ as the one-state automaton accepting a^*. For simple-run semantics, one also has to note that for that particular $\mathcal{A}$ it holds $[\![\mathcal{A}]\!]_{SR}(D) = [\![\mathcal{A}]\!]_{SW}(D)$.

4.4 Walk Membership

The last problem we consider here is WALK MEMBERSHIP, which consists in deciding whether a given walk is returned.

This problem is usually considered whenever Tuple Membership and Query Evaluation are intractable; and as a matter of fact, it is known to be tractable for all usual semantics (Theorem 23). Surprisingly, it is intractable under simple-run semantics (Theorem 24).

Walk Membership under X semantics
• Data: A database D and a walk w.
• Query: An automaton $\mathcal{A}$.
• Question: $w \in [\![\mathcal{A}]\!]_X(D)$?

Theorem 23. Walk Membership *is NL-complete under walk, trail or simple-walk semantics.*

Sketch of proof. For walk semantics, the problem amounts to checking acceptance of a word in a nondeterministic finite automaton. For trail (resp. simple-walk) semantics, one has to additionally check that the input walk is a trail (resp. a simple walk). Hardness comes from an easy reduction from ST-connectivity. ☐

Theorem 24. Walk Membership *is NP-complete under simple-run semantics. It is already NP-hard in data complexity: there exists a fixed automaton $\mathcal{A}$ for which the problem is NP-hard.*

Sketch of proof. The hardness proof is done by a direct reduction from 3-SAT. The fixed automaton $\mathcal{A}$ uses the alphabet $\{\mathsf{Var}, \mathsf{Keep}, \mathsf{Invert}, \mathsf{Eval}, \mathsf{Check}\}$, has three states $\{0, 1, \top\}$, $\top$ is the unique initial and final state, and its transition table is given below.

Keep:	0	$\to$	0
	1	$\to$	1
	$\top$	$\to$	$\top$
Reset:	$\{0,1,\top\}$	$\to$	$\top$
Check:	$\{0,\top\}$	$\to$	$\top$

Var:	$\{0,1,\top\}$	$\to$	$\{0,1\}$
Invert:	0	$\to$	1
	1	$\to$	0
Eval:	1	$\to$	$\{0,1\}$
	$\{0,\top\}$	$\to$	$\top$

Figure 3 gives an example of how the database D is built from a specific 3-SAT instance, and for this example, the input walk $w = w_V w_H$ goes through all edges of D, starting with vertical edges (w_V) and then horizontal edges (w_H):

$$w_V = \text{Start} \to x_1 \to \cdots \bar{x}_1 \to \cdots x_2 \to \cdots \bar{x}_4 \to \text{Mid}$$

$$w_H = \text{Mid} \to C_0 \to \bar{x}_1^1 \to \cdots C_1 \to x_1^2 \to \cdots$$
$$C_2 \to x_1^3 \to \cdots C_3 \to \text{End}$$

Each valuation of the variables corresponds in a one-to-one manner to one run in $D \times \mathcal{A}$ for the vertical part. For instance, valuation $x_1 \mapsto 1$, $x_2 \mapsto 1$, $x_3 \mapsto 0$, $x_4 \mapsto 0$ corresponds to the run r_V:

In D :	Start	x_1	x_1^2	x_1^3	$\bar{x}_1$	$\bar{x}_1^1$	x_2	$\cdots$	x_3	$\cdots$	x_4	$\cdots$	Mid
In $\mathcal{A}$:	$\top$	1	1	1	0	0	1	$\cdots$	0	$\cdots$	0	$\cdots$	$\top$

The horizontal walk w_H has three parts ($C_0 \to C_1$, $C_1 \to C_2$ and $C_2 \to C_3$); each $C_{i-1} \to C_i$ checks whether the valuation makes the clause C_i true. Let us take C_1 for instance. There is exactly one run r_1 for the part $C_0 \to C_1$ in order for $r_V r_1$ to be simple, given below.

In D :	C_0	$\bar{x}_1^1$	x_3^1	$\bar{x}_4^1$	C_1
In $\mathcal{A}$:	$\top$	1	1	0	$\top$

Indeed, the state reached at $\bar{x}_1^1$ is necessarily 1, otherwise the full run would not be simple: the vertex $(\bar{x}_1^1, 0)$ of the run database was already visited in the vertical part. One may see that the state reached at vertex C_1 is $\top$, which means that the valuation satisfies C_1. If the valuation did not make C_1 true, there would be no run r_1 such that $r_V r_1$ is simple. ☐

5 Query Given as a Regular Expression

This section aims at applying simple-run semantics to practical settings. In real life, users generally input RPQs as a regular expression and not as an automaton. The natural idea would consist in translating the expression and then applying simple-run semantics to the resulting automaton. In Section 5.1, we explain why this approach leads to unwanted behaviour. Section 5.2 introduces binding-trail semantics as an adhoc adaptation of simple-run semantics to the case where the query is given as a regular expressions. Then, we show that binding-trail semantics enjoy the same computational properties as simple-run semantics.

5.1 Expression to Automaton

Given a regular language, there are many known approaches for producing automata which accept the same language (see for instance Sakarovitch 2021). Then, all algorithms from Section 4 immediately apply. However, the crux of the matter lies in Remark 7: semantics do not only depend on the language accepted by the automaton, but on the automaton itself. Thus, *how* we choose to translate the expression into an equivalent automaton matters, and it seems that each translation algorithm features undesirable quirks. We give two compelling examples, and leave a complete account of the many translation algorithms for future work.

First, one could translate the expression into a minimal DFA, but this choice makes the semantics non-compositional. Consider an expression $R = R_1 + R_2$, one would expect R to return more results than R_1 or R_2; but it is not always the case. For instance, if $R_1 = b^*(ab^*ab^*)^*$ and $R_2 = (a+b)(a+b)^*$, then R is equivalent to $(a+b)^*$. The minimal DFA $\mathcal{A}_1$ associated with R_1 has two states, while the minimal DFA $\mathcal{A}$ associated with R has only one state. Hence one can easily find a database D such that $[\![\mathcal{A}]\!]_{SR}(D) \not\supseteq [\![\mathcal{A}_1]\!]_{SR}(D)$. Similar quirks can be found for concatenation and star. This example illustrates that the translated automaton must not only represent the *regular language* but also the *regular expression as written*.

Second, one could use Glushkov construction (recalled in Definition 25, below), which famously produces an automaton that stays close to the regular expression. However, this choice introduces a left-to-right bias. Typically, $\mathcal{A} = Gl(a^*b^*)$ is not the mirrored automaton of $Gl(b^*a^*)$, and if one considers the following database D, the walk `S` $\to$ `S` $\to$ `T` belongs to $[\![\mathcal{A}]\!]_{SR}(D)$ but not the walk `S` $\to$ `T` $\to$ `T`.

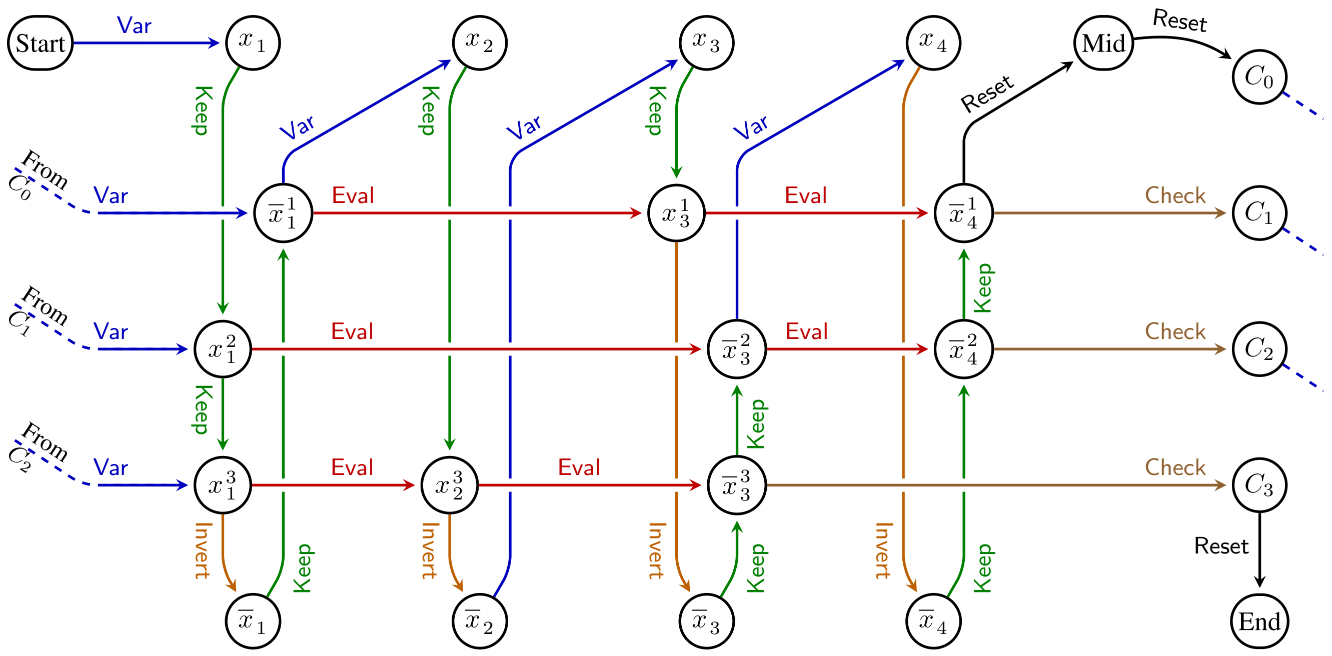
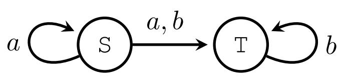


Figure 3: Graph encoding 3-SAT instance $C_1 \wedge C_2 \wedge C_3$ with $C_1 = \neg x_1 \vee x_3 \vee \neg x_4$, $C_2 = x_1 \vee \neg x_3 \vee \neg x_4$ and $C_3 = x_1 \vee x_2 \vee \neg x_3$



Definition 25. *Let R be a regular expression over Σ. A* linearisation *of R is a copy R' of R in which each atom in Σ is replaced by a different symbol in a new alphabet Γ, called the* positions *of R. Given $\alpha \in \Gamma$, we denote as $\overline{\alpha}$ the label of its antecedent in R.*

The Glushkov's automaton *of R, $Gl(R)$, is the automaton $\langle \Sigma, \{i\} \uplus \Gamma, \Delta, \{i\}, F\rangle$ defined as follows:*

$$\Delta = \{ (\alpha, \overline{\beta}, \beta) \mid \exists u, v \in \Gamma^*, u\alpha\beta v \in L(R') \} \cup \{ (i, \overline{\alpha}, \alpha) \mid \exists u \in \Gamma^*, \alpha u \in L(R') \}$$
$$F = \{ \alpha \mid \exists u \in \Gamma^*, u\alpha \in L(R') \} \cup \{ i \} \text{ if } \varepsilon \in L(R')$$

5.2 Binding-Trail Semantics

This section defines *binding-trail semantics*, a counterpart to simple-run semantics that operates directly on a given regular expression R, without translating R into an automaton.

Definition 26. *Let $D = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ be a database, and R a regular expression. Let R' be a linearisation of R and Γ be the corresponding positions of R. A* binding trail *of D matching R is a sequence $(e_1, \alpha_1) \ldots (e_n, \alpha_n)$ of pairs in $E \times \Gamma$ such that:*

- *$e_1 \ldots e_n$ describes a walk of D and $\overline{\alpha_1 \cdots \alpha_n} \in \textsc{Lbl}(e_1 \ldots e_n)$;*
- *$\alpha_1 \cdots \alpha_n$ belongs to $L(R')$;*
- *All (e_i, α_i) are pairwise distinct.*

Binding-trail semantics are then defined by:

$$[\![R]\!]_{BT}(D) = \pi_D \left(\{ t \mid t \text{ is a binding trail of } D \text{ matching } R \} \right)$$

In other words, given a walk w in D, w conforms to binding-trail semantics if w matches R in such a way that the same edge of w cannot be used twice at the same position in R. The following lemma shows how binding-trail semantics relate to the run database.

Lemma 27. *For every regular expression R and database D,*

$$[\![R]\!]_{BT}(D) = \pi_D \circ \textsc{BindingTrail} \circ \textsc{Match}_{\mathcal{A}}(D) ,$$

where $\mathcal{A} = Gl(R)$ and BindingTrail *is the run-bag filter that keeps only the runs*

$$(n_0, q_0)(e_0, \delta_0)(n_1, q_1) \cdots (e_{k-1}, \delta_{k-1})(n_k, q_k)$$

such that all (e_i, q_{i+1})'s, $0 \leq i < k$, are pairwise distinct.

As is the case for simple-run semantics, binding-trail semantics also coincide with walk semantics for Tuple Membership, and produce the same shortest witnesses. Indeed, the proof of Proposition 19 can easily be adapted to prove the following:

Proposition 28. *Let D be a database and R be an expression. Let $\mathcal{A}$ be any automaton such that $L(R) = L(\mathcal{A})$. Let s, t be two vertices in D, and we denote by P the set of walks in $[\![\mathcal{A}]\!]_W(D)$ that go from s to t. Let w be a walk in P with minimal length. Then, w belongs to $[\![R]\!]_{BT}(D)$.*

Lemma 27 hints at the fact that the upper bounds of Section 4 for simple-run semantics immediately apply to binding-trail semantics, due to standard graph reduction techniques translating vertex-disjoint walks to edge-disjoint walks and back. The same does not necessarily hold true for lower bounds. Glushkov automata have additional properties: only one initial state, all incoming edges to a given state have the same label, and so on (Caron and Ziadi 2000). We show that these properties cannot be used to design more efficient algorithms.

Proposition 29. TUPLE MEMBERSHIP, TUPLE MULTIPLICITY, QUERY EVALUATION *and* WALK MEMBERSHIP *are computationally equivalent under binding-trail semantics and under simple-run semantics.*

Proposition 29 is actually a consequence of a much deeper result stating essentially that, given any automaton $\mathcal{A}$, there exists a regular expression R that encodes the topology of $\mathcal{A}$ in the sense that there is a strong connection between the computations of $\mathcal{A}$ and those of $Gl(R)$. Hence, any problem that takes automata as input will likely have hard instances that are of the form $Gl(R)$ for some expression R. Due to space constraints, we only sketch the main idea of the encoding.

Sketch of proof. Let $\mathcal{A} = \langle \Sigma, Q, \Delta, I, F \rangle$. Let $m = \text{CARD}(\Delta)$ and G denote any bijection $G : \Delta \to \{1, \dots, m\}$. Let H be the only bijection $H : \Delta \to \{1, \dots, m\}$ that meets: $\forall e \in \Delta,\ G(e) + H(e) = m + 1$. Finally, let σ be a fresh symbol that is not in Σ. We define the expression R over the alphabet $\Sigma \uplus \{\sigma\}$ as follows:

$$\left(\sum_{q \in Q} \left[\Big(\overbrace{\varepsilon\, +}^{\text{if } q \in I} \sum_{\substack{s \in Q,\ a \in \Sigma \\ e=(s,a,q) \in \Delta}} a^{G(e)} \Big) \sigma \Big(\overbrace{\varepsilon\, +}^{\text{if } q \in F} \sum_{\substack{a \in \Sigma,\ t \in Q \\ e=(q,a,t) \in \Delta}} a^{H(e)} \Big) \right] \right)^*$$

Note that there are exactly $\text{CARD}(Q)$ occurrences of the letter σ in R. We associate each state $s \in Q$ with the occurrence of σ appearing in the term of the external sum when $q = s$. Similarly, each transition $e = (s, a, t)$ with label a in $\mathcal{A}$ is encoded by the word $\sigma a^{m+1} \sigma$ that will be matched by the concatenation of the σ corresponding to s, the subexpression $a^{H(e)}$ on its right, the $a^{G(e)}$ on the left of σ corresponding to t followed by this σ.

We conclude by showing that a successful computation in $\mathcal{A}$ over a word $a_0 \cdots a_n \in \Sigma^*$ is encoded by matching the word $\sigma {a_0}^{m+1} \sigma \cdots {a_n}^{m+1} \sigma$ in R. Moreover, this encoding preserves relevant topological properties. For instance, a computation of $\mathcal{A}$ reuses a state if and only if its encoding reuses a position labelled by σ. □

Remark 30. *Similarly to simple-run semantics (see Remark 14), binding-trail semantics depend on the given regular expression and not only on the corresponding language. This allows for a finer control on which repetitions are permitted by the query. For instance, the three following expressions have different meanings under binding-trail semantics:*

- $[\![a^*]\!]_{BT}$ *returns all trails labelled by a.*
- $[\![a^* \cdot a^*]\!]_{BT}$ *returns the concatenations of two trails.*
- $[\![(a + a)^*]\!]_{BT}$ *returns all walks where edges are repeated at most twice.*

6 Perspectives

Run-based semantics are an attempt at addressing real-life concerns while maintaining good theoretical foundations. As such, our medium-term goal is to make sure that our work is indeed applicable to query languages used in practice. GQL offers a very plausible opportunity for integration, as it is in active development and already supports several semantics. This section aims at closing some of the gaps between theory and practice by discussing how run-based semantics adapt to commonly seen extensions or limitations.

6.1 Syntax Restrictions Used in Practice

Real query languages, such as GQL or Cypher, impose syntax restrictions on the regular expression given as input. We discuss here whether the lower bound complexity results change by imposing those restrictions. Note that no reasonable syntax restriction can change the complexity of TUPLE MULTIPLICITY since the lower bound already holds for the fixed expression a^*.

We consider three syntax restrictions and their respective impact on the complexity of WALK MEMBERSHIP. A regular expression has *star-height 1* if it has no nested Kleene stars. A regular expression is said to have *no union under star* (resp. *no concatenation under star*) if no union (resp. no concatenation) operator occurs in any subexpression nested under a Kleene star.

These restrictions match syntax rules commonly seen in practice: GQL only allows expressions with star-height 1 and Cypher queries cannot express concatenations under star. Moreover, users rarely use the full expressive power at their disposal. In (Bonifati, Martens, and Timm 2020), the authors make an analytical study of over 240,000 SPARQL queries: in the collected data set, every single RPQ has star-height 1, all queries but one have no concatenation under star, and only about 40% of queries use a union under a star.

The expression shown in Section 5.1 to simulate the computations of any arbitrary automaton has no nested stars. Hence all complexity lower bounds hold even when expressions are restricted to star-height 1. That expression has unions under star, but one can do without. Indeed, let $\mathcal{A} = \langle \Sigma, Q, \Delta, I, F \rangle$ be an automaton; for simplicity we assume that $\Sigma = \{0, 1, \dots, k-1\}$ and that $Q = \{0, 1, \dots, n-1\}$. Consider the following expression over the alphabet $\{a, b, c, \sigma\}$.

$$\sum_{i \in I} c^{i+1} \left(\prod_{i \in Q} \left(c^{n-i} \sigma a^{i+1}\right)^* \cdot \prod_{(i,x,j) \in \Delta} \left(a^{n-i} b^x c^{j+1}\right)^* \right)^* \sum_{i \in F} a^{n-i} \qquad (4)$$

As in Section 5.2, one can show that this new expression[6] encodes the behaviour of $\mathcal{A}$. The key arguments are as follows: each letter $x \in \Sigma$ is encoded by $\lambda(x) = a^{n+1} b^x c^{n+1} \sigma$; each word $x_0 \cdots x_n$ is encoded by $c^{n+1} \lambda(x_1) \cdots \lambda(x_n) a^{n+1}$; the states of $\mathcal{A}$ are simulated by the positions with label σ.

[6]Strictly speaking, Equation (4) denotes a *family* of expressions, as concatenation (Π) is noncommutative. However, the reduction works for any of those expressions.

	Trail	Run-based	Homomorphism	Shortest-walk
TUPLE MEMBERSHIP	Intractable	Tractable	Tractable	Tractable
TUPLE MULTIPLICITY	Intractable	Intractable	$*$	$*$
QUERY EVALUATION	Intractable	Tractable	$*$	Tractable
DEDUPLICATED QUERY EVAL.	Intractable	Open	$*$	Tractable
WALK MEMBERSHIP	Tractable	Intractable	Tractable	Tractable

Table 1: Summary of computational complexity

Remark 31. *The internal Kleene stars in Equation (4) are used at most once, so these stars might be replaced by an* optional *operator, sometimes denoted by a "?". In that case, the expression would also be of star-height 1.*

When expressions have no concatenation under star, WALK MEMBERSHIP becomes tractable in combined complexity, as stated below.

Theorem 32. WALK MEMBERSHIP *is in PTIME under binding-trail semantics when restricted to expressions with no concatenation under star. The same holds under simple-run semantics when queries are restricted to the Glushkov automata of such expressions.*

Under binding-trail semantics, matching a starred subexpression with no concatenation in a walk amounts to counting that the number of repetitions of each edge in the walk is less than the number of compatible atoms of the expression. Under simple-run semantics, the proof relies on a reduction to matchings in bipartite graphs (Cormen et al. 2009, Section 26.3).

6.2 Extensions of Regular Expressions

Let us discuss how simple-run and binding-trail semantics behave with respect to some common extensions of RPQs.

One-or-more repetitions Many formalisms allow writing R^+ as a shorthand for $R \cdot R^*$. While expanding the notation is not suitable in our setting (the two R subexpressions in $R \cdot R^*$ would then be matched independently, resulting in a different behaviour), treating R^+ as a new operator poses no particular problem: the definitions of both binding-trail semantics and Glushkov automaton extend naturally over $+$.

Arbitrary repetitions Some formalisms allow an operator '$^{\{n,m\}}$' with $n \in \mathbb{N}$ and $m \in \mathbb{N} \cup \{\infty\}$: $R^{\{n,m\}}$ means that R may be repeated between n and m times. Once again, expanding the notation would change the result. On the other hand, allowing this new operator would make Lemma 13 false, e.g., querying the database from Fig. 1 with $R^{\{6,\infty\}}$ would yield tuple (s, t) under walk semantics but not under binding-trail semantics. The impact of this operator on complexity results is left for future work.

Backward atoms Allowing backward atoms $\overleftarrow{a}$ in expressions, as for instance in 2RPQs (Angles et al. 2017), poses no particular problem: transitions of the automaton that are labelled with a backward atom are simply paired with reversed edges of the database in the run database.

Any-directed atoms Cypher and GQL allow any-directed atoms $\overleftrightarrow{a}$, that match edges labelled by a forward or backward. Allowing them would make Lemma 13 (and Prop. 28) false. For instance, let us query the database from Figure 1 with $R = \Big((\overrightarrow{\mathbf{R}} + \overrightarrow{\mathbf{G}}) \cdot \overleftrightarrow{\mathbf{R}} \cdot (\overrightarrow{\mathbf{R}} + \overrightarrow{\mathbf{G}})\Big)^*$. The walk $w = c_1 \to c_2 \to c_3 \to c_3 \leftarrow c_2 \to t$ contradicts Lemma 13. This issue can be circumvented by expanding $\overleftrightarrow{a}$ into $\overrightarrow{a} + \overleftarrow{a}$ rather than treating it as a new operator, once again with some effect on the semantics.

7 Conclusion

Table 1, below, presents a summary of the computational complexity of popular semantics and compares them with run-based semantics. We also emphasize the following comparison points that do not appear in the table.

- Table 1 paints a negative picture of trail semantics, which seems at odds with its popularity in practice. We believe that one strength of trail semantics is that the output provides some kind of coverage of the space of matches, which enables rich aggregation and post-processing. Run-based semantics improves on this property by giving some guarantees on the coverage (Lemma 13).
- WALK MEMBERSHIP is a theoretical counterpart to QUERY EVALUATION: only the latter is implemented in real systems. Thus, we believe that run-based semantics offer a reasonable compromise by having tractable TUPLE MEMBERSHIP, tractable QUERY EVALUATION and intractable WALK MEMBERSHIP. It is in our view better than the other way around, as in trail semantics.
- Under homomorphism or shortest-walk semantics, some problems are marked with a $*$. Although tractable, complexity comparison with other semantics makes little sense since they behave differently. For instance, counting *all* matching walks leads to unboundedness under homomorphism semantics, and returns an information of dubious value under shortest-walk: 0 or 1 most of the time, and the number of uncomparable minimal runs otherwise.

In conclusion, simple-run and binding-trail semantics provide good computational properties overall, supports bag semantics and rich aggregation. However, while the RPQ formalism is a good model of the navigational part of most query languages over graph databases, it does not capture their ability to collect and compare data values along tested walks. Extending our proposed framework to handle data values lying in the edges and vertices of the database constitutes our main challenge going forward.

References

Angles, R.; Arenas, M.; Barceló, P.; Hogan, A.; Reutter, J. L.; and Vrgoč, D. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50(5).

Angles, R.; Arenas, M.; Barceló, P.; Boncz, P. A.; Fletcher, G. H. L.; Gutierrez, C.; Lindaaker, T.; Paradies, M.; Plantikow, S.; Sequeda, J. F.; van Rest, O.; and Voigt, H. 2018. G-CORE: A core for future graph query languages. In *SIGMOD*, 1421–1432. ACM.

Bagan, G.; Bonifati, A.; and Groz, B. 2020. A trichotomy for regular simple path queries on graphs. *J. Comput. Syst. Sci.* 108:29–48.

Bonifati, A.; Martens, W.; and Timm, T. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29(2-3):655–679.

Caron, P., and Ziadi, D. 2000. Characterization of glushkov automata. *Theor. Comput. Sci.* 233(1-2):75–90.

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2009. *Introduction to Algorithms, third edition.* MIT Press.

Cruz, I. F.; Mendelzon, A. O.; and Wood, P. T. 1987. A graphical query language supporting recursion. In Dayal, U., and Traiger, I. L., eds., *SIGMOD'87*, 323–330. ACM.

Deutsch, A.; Xu, Y.; Wu, M.; and Lee, V. E. 2019. Tigergraph: A native MPP graph database. Preprint arXiv:1901.08248.

Deutsch, A.; Francis, N.; Green, A.; Hare, K.; Li, B.; Libkin, L.; Lindaaker, T.; Marsault, V.; Martens, W.; Michels, J.; Murlak, F.; Plantikow, S.; Selmer, P.; Voigt, H.; van Rest, O.; Vrgoč, D.; Wu, M.; and Zemke, F. 2022. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD'22*. ACM.

Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; and Taylor, A. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD'18*. ACM.

Francis, N.; Gheerbrant, A.; Guagliardo, P.; Libkin, L.; Marsault, V.; Martens, W.; Murlak, F.; Peterfreund, L.; Rogova, A.; and Vrgoč, D. 2023. A Researcher's Digest of GQL. In Geerts, F., and Vandevoort, B., eds., *ICDT'23*, volume 255 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

International Organization for Standardization. GQL. https://www.iso.org/standard/76120.html. Standard under development ISO/IEC CD 39075. Expected to be published in 2023.

Martens, W.; Niewerth, M.; and Trautner, T. 2020. A trichotomy for regular trail queries. In Paul, C., and Bläser, M., eds., *STACS'20*, volume 154 of *LIPIcs*, 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Mendelzon, A. O., and Wood, P. T. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24(6):1235–1258.

Pin, J.-É., ed. 2021. *Handbook of Automata Theory*, volume 1. EMS.

Robinson, I.; Webber, J.; and Eifrem, E. 2015. *Graph databases, second edition.* O'Reilly.

Sakarovitch, J. 2021. Automata and rational expressions. Chapter 2 of (Pin 2021).

TigerGraph Team. 2021. TigerGraph Documentation – version 3.1.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8(3).

World Wide Web Consortium. 2013. SPARQL 1.1 query language, section 9: Property paths. https://www.w3.org/TR/sparql11-query/#propertypaths.

Yen, J. Y. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17(11):712–716.

Acknowledgments

The authors would like to warmly thank Amélie Gheerbrant and Alexandra Rogova for pointing out the example in *Graph databases* (Robinson, Webber, and Eifrem 2015) that fails due to trail semantics. The authors are also very grateful to Antoine Meyer for the many fruitful conversations about the content of this paper. The authors acknowledge the support of *Agence Nationale de la Recherche* under grant *QUID*, code ANR-18-CE40-0031.