Evaluating Regular Path Queries in GQL and SQL/PGQ: How Far Can The Classical Algorithms Take Us?

Benjamín Farías
PUC Chile and IMFD Chile
bffarias@uc.cl

Carlos Rojas
IMFD Chile
cirojas6@uc.cl

Domagoj Vrgoč
PUC Chile and IMFD Chile
dvrgoc@ing.puc.cl

ABSTRACT

We tackle the problem of answering regular path queries over graph databases, while simultaneously returning the paths which witness our answers. We study this problem under the arbitrary, all-shortest, trail, and simple-path semantics, which are the common path matching semantics considered in the research literature, and are also prescribed by the upcoming ISO Graph Query Language (GQL) and SQL/PGQ standards. In the paper we present how the classical product construction from the theoretical literature on graph querying can be modified in order to allow returning paths. We then discuss how this approach can be implemented, both when the data resides in a classical B+ tree structure, and when it is assumed to be available in main memory via a compressed sparse row representation. Finally, we perform a detailed experimental study of different trade-offs these algorithms have over real world queries, and compare them with existing approaches.

1 INTRODUCTION

Graph databases [3, 33] have gained significant popularity in recent years, and are used in areas such as Knowledge Graphs [22], Biology [23], and The Semantic Web [4]. The subject also received substantial coverage in the research literature [5, 27], and there are multiple vendors offering graph database products [29, 37, 38, 40–42, 49]. With the proliferation of different engines and query languages, there was also an increasing need for a standard dialect for expressing graph queries. This led to several standardization efforts, including SPARQL [21], which is a W3C standard for querying edge-labeled graphs, and a more recent initiative by the ISO committee to define a native query language for property graphs. The latter resulted in the upcoming Graph Query Language (GQL) standard, as well as SQL/PGQ (which extends SQL with capabilities for graph pattern matching on property graphs) [11].

A common class of graph-specific queries that are represented in all of these languages are *regular path queries* (or *RPQs* for short), which have long been studied in the database literature [5, 9, 10, 27]. Intuitively, an RPQ is an expression of the form $(?x, \mathtt{regex}, ?y)$, where regex is a regular expression, and $?x$, $?y$ are variables. When evaluated over an edge-labeled graph G, the query extracts all pairs of nodes (n1, n2) such that there is a path in G linking n1 to n2, and whose edge labels form a word that matches regex. This is the traditional semantics of RPQs used in the research literature [5], and implemented in the SPARQL standard [21]. Notice that these approaches simply return endpoints of paths, and not paths themselves. However, having a path witnessing an RPQ query answer is a useful feature, and as such is present in languages such as GQL, SQL/PGQ, and their academic predecessors such as G-Core [3]. To illustrate different types of paths one might wish to obtain, consider the graph database in Figure 1. In this graph we have node identifiers (such as Joe, or Rome), edge identifiers (such as e1), and edge labels (e.g. knows for e7). Since we study RPQs, which only access nodes, edges, and edge labels, we will work with a simplified model of graph databases to keep the discussion concise. Real-world property graphs also allow node labels, and property-value pairs on both nodes and edges. Our results transfer verbatim to this setting.

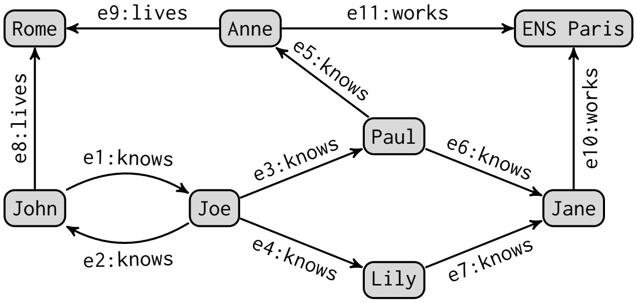


Figure 1: A sample graph database.

A natural query over the database of Figure 1 is to find all friends-of-friends of Joe. As an RPQ, we would express this with an expression $(\mathsf{Joe}, \mathsf{knows}^+, ?x)$, signaling that we wish to start at the node Joe, and traverse any number of knows-labeled edges. The query would then return all the people in the database, including Joe. If we also wish to return paths witnessing these connections, we might run into some issues. Most notably, given the cycle formed by the edges e1 and e2, there is an infinite number of paths that start with Joe, and return to Joe. To cope with these sorts of issues, GQL and SQL/PGQ [11] provide two mechanisms (also referred to as *path modes*) that ensure the number of returned paths to be finite.

The first mechanism are restrictors, which restrict the set of allowed paths. The most common restrictors are SIMPLE, which requires the paths to not repeat any node (apart from the first and last one), and TRAIL, which requires no edge to be repeated. If we adhere to these restrictors, now the number of paths that leave Joe and come back to this same node is precisely one; namely, we can take the edge e2 and come back to Joe via the edge e1, and cannot continue looping, since we would either repeat an edge or a node.

The second mechanism are selectors, which select some paths from the set of all matching paths. For instance, when looking for paths witnessing a connection, a natural candidate is to find a shortest such path. GQL and SQL/PGQ allow doing so by applying the selector ANY SHORTEST to a query, which will return a single shortest path for each pair of connected nodes. There is also an option to find *all* shortest paths via the ALL SHORTEST selector. For instance, consider the RPQ $(\mathsf{Joe}, \mathsf{knows}^+ \cdot \mathtt{works}, ?x)$, which looks for the working place of Joe's friends. In the graph of Figure 1, there are three different shortest paths linking Joe to ENS Paris (traced by the edges e3 → e5 → e11, e4 → e7 → e10, and e3 → e6 → e10, respectively), and we might wish to retrieve all of them.

Overall, GQL and SQL/PGQ provide an interesting challenge in terms of RPQ evaluation, since they combine the requirement to find appropriately labeled paths, with restrictions on such paths. In this paper we study how, given an RPQ, one can attach a GQL or SQL/PGQ path mode to this RPQ and return the required paths.

Our contribution. The main objective of our work is to show how *classical graph search algorithms can be used to efficiently evaluate regular path queries under all path modes prescribed by GQL and SQL/PGQ*. As such, our contributions can be summarized as follows:

- We extend the graph product approach [5, 26] used in the theoretical literature to evaluate RPQs, with the ability to return paths. We do so by adapting the classical BFS/DFS graph search algorithms to take into account any RPQ semantics as prescribed by GQL or SQL/PGQ.
- We develop a new algorithm for finding *all shortest paths* witnessing an RPQ answer, based on an extension of the classical BFS algorithm. We also pinpoint the precise requirements needed for other path modes to work correctly.
- We implement all of the algorithms inside of MillenniumDB, an open-source graph engine [47], and test their performance both in a classical setup of B+tree-based storage, and when data is available in a main memory index.

Limitations. For all of our algorithms we will assume that the starting point for the path is fixed. Namely, we will consider RPQs such as $(\mathsf{Joe}, \mathsf{knows}^+, ?x)$, which assume that the path starts with the node Joe. While RPQs allow both ends of the path to be variables, we fix the starting point for two reasons: (i) this approach allows us to use standard graph traversal algorithms to evaluate RPQs; and (ii) RPQs are usually part of a larger query which selects one or more nodes from which path searching is done [7]. We remark that having multiple starting points is not an issue for our algorithms, but they would act as independent searches. We will also not consider top-k selectors from GQL [11], since even without those we have to handle 11 different path evaluation modes. Finally, the GQL and SQL/PGQ standard allow mixing path modes for different parts of a path. For instance, one can define a query that concatenates two paths: one a simple path, and another a trail, and also require that such concatenated path is a shortest path. In this paper we handle only the base case of having a *single* RPQ and a *single* path mode.

Organization. We define graph databases and RPQs in Section 2. Algorithms for the WALK restrictor are studied in Section 3, while TRAIL and SIMPLE are tackled in Section 4. We discuss implementation details in Section 5, and experimental results in Section 6. Related work is discussed in Section 7. We conclude in Section 8. Additional details, code, and proofs can be found at [14].

2 GRAPH DATABASES AND RPQS

In this section we define graph databases and regular path queries.

Graph databases. Let Nodes be a set of node identifiers and Edges be a set of edge identifiers, with Nodes and Edges being disjoint. Additionally, let Lab be a set of labels. Following the research literature [4, 11, 16, 26], we define graph databases as follows.

Definition 2.1. A *graph database* G is a tuple (V, E, ρ, λ), where:

- $V \subseteq \mathsf{Nodes}$ is a finite set of nodes.
- $E \subseteq \mathsf{Edges}$ is a finite set of edges.
- $\rho : E \rightarrow (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ means that e is a directed edge going from v_1 to v_2.
- $\lambda : E \rightarrow \mathsf{Lab}$ is a total function assigning a label to an edge.

Notice that, similarly as in [26], we use a simplified version of property graphs [11], where we only consider nodes, edges, and edge labels. Most importantly, we omit properties (with their associated values) that can be assigned to nodes and edges, as well as node labels. This is done since the type of queries we consider only use nodes, edges, and edge labels. However, all of our results transfer verbatim to the full version of property graphs. We also remark that our results apply directly to RDF graphs [30] and edge-labeled graphs [5, 27], which do not use explicit edge identifiers.

Paths. A *path* in a graph database $G = (V, E, \rho, \lambda)$ is a sequence

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

where $n \geq 0$, $e_i \in E$, and $\rho(e_i) = (v_{i-1}, v_i)$ for $i = 1, \ldots, n$. If p is a path in G, we write $\mathsf{lab}(p)$ for the sequence of labels $\mathsf{lab}(p) = \lambda(e_1) \cdots \lambda(e_n)$ occurring on the edges of p. We write $\mathsf{src}(p)$ for the starting node v_0 of p, and $\mathsf{tgt}(p)$ for the end node v_n of p. The length of a path p, denoted $\mathsf{len}(p)$, is defined as the number n of edges it uses. We will say that a path p is a:

- WALK, for any p.[1]
- TRAIL, if p does not repeat an edge. That is, if $e_i \neq e_j$ for any pair of edges e_i, e_j in p with $i \neq j$.
- ACYCLIC, if p does not repeat any node. That is, $v_i \neq v_j$ for any pair of nodes v_i, v_j in p with $i \neq j$.
- SIMPLE, if p does not repeat a node, except that possibly $\mathsf{src}(p) = \mathsf{tgt}(p)$. That is, if $v_i \neq v_j$ for any pair of nodes v_i, v_j in p with $(i, j) \in \{0, \ldots, n\}^2 - \{(0, n)\}$.

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

Regular path queries. The class of queries we study in this paper are *regular path queries*, or *RPQs* for short. RPQs have been well studied in the research literature [4, 5, 27], they form the backbone of most practical graph query languages [17, 21, 37, 39, 41], and are also the basis of navigation in the upcoming GQL and SQL/PGQ standards [11]. For us, a regular path query will be an expression of the form

$$\text{selector? restrictor } (v, \texttt{regex}, ?x)$$

where $v \in \mathsf{Nodes}$ is a node, regex is a regular expression, and $?x$ is a variable. Following GQL and SQL/PGQ [11], we use *selectors* and *restrictors* to specify which paths are to be returned by the RPQ. The grammar of selectors and restrictors is as follows:

restrictor : WALK | TRAIL | SIMPLE | ACYCLIC

selector : ANY | ANY SHORTEST | ALL SHORTEST

Traditionally [4], the $(v, \texttt{regex}, ?x)$ part of an RPQ tells us that we wish to find all the nodes v' of our graph G for which there is

[1] The term *path* is used in the database literature to denote what is called a *walk* in graph theory. GQL and SQL/PGQ use WALK as a keyword for denoting any path.

a path p from v to v', such that $\mathsf{lab}(p)$ is a word in the language of the regular expression $\texttt{regex}$.[2] Since the set of all such paths can be infinite [4], restrictors allow us to specify which paths are considered valid, while selectors filter out results from a given set of valid paths. Next, we formally define the semantics of an RPQ.

Let G be a graph database and q an RPQ of the form:

$$\mathsf{restrictor}\ (v, \texttt{regex}, ?x)$$

namely, we omit the optional selector part for now. We use the notation $\mathsf{Paths}(G, \mathsf{restrictor})$ to denote the set of all paths in G that are valid according to restrictor. For example, $\mathsf{Paths}(G, \mathsf{TRAIL})$ is the set of all trails in G. We then define the semantics of q over G, denoted $[\![q]\!]_G$, where $q = \mathsf{restrictor}\ (v, \texttt{regex}, ?x)$, as follows:

$$\begin{aligned}[\![\mathsf{restrictor}\ (v, \texttt{regex}, ?x)]\!]_G = \{\ (p, v')\ \ | &\ p \in \mathsf{Paths}(G, \mathsf{restrictor}),\\ &\mathsf{src}(p) = v, \mathsf{tgt}(p) = v',\\ &\mathsf{lab}(p) \in \mathcal{L}(\texttt{regex})\ \}\ .\end{aligned}$$

Here $\mathcal{L}(\texttt{regex})$ denotes the language of the regular expression $\texttt{regex}$. Intuitively, for an RPQ "$\mathsf{TRAIL}\ (v, \texttt{regex}, ?x)$", we will return all pairs (p, v') such that p is a TRAIL in our graph that connects v to v' (and $\mathsf{lab}(p) \in \mathcal{L}(\texttt{regex})$), and similarly for other restrictors. Finally, the semantics of queries that also use selectors is defined on a case-by-case basis. For this, we will use q to denote the selector-free RPQ $q = \mathsf{restrictor}\ (v, \texttt{regex}, ?x)$. We now have:

- $[\![\mathsf{ANY}\ \mathsf{restrictor}\ (v, \texttt{regex}, ?x)]\!]_G$ returns, for each node v' reachable from v by a path p with $(p, v') \in [\![q]\!]_G$, a *single such pair* $(p, v') \in [\![q]\!]_G$, chosen non-deterministically.
- $[\![\mathsf{ANY\ SHORTEST}\ \mathsf{restrictor}\ (v, \texttt{regex}, ?x)]\!]_G$ returns, for each node v' reachable from v by a path p with $(p, v') \in [\![q]\!]_G$, a *single pair* $(p, v') \in [\![q]\!]_G$, where p is $\mathsf{SHORTEST}$ among all paths p' for which $(p', v') \in [\![q]\!]_G$.
- $[\![\mathsf{ALL\ SHORTEST}\ \mathsf{restrictor}\ (v, \texttt{regex}, ?x)]\!]_G$ will return, for each v' reachable from v by a path p with $(p, v') \in [\![q]\!]_G$, the *set of all pairs* $(p, v') \in [\![q]\!]_G$ with p $\mathsf{SHORTEST}$ among paths p' for which $(p', v') \in [\![q]\!]_G$.

Intuitively, we can think of paths being grouped by v' before the selector is applied. Notice that the semantics of ANY and ANY SHORTEST is non-deterministic when there are multiple (shortest) paths connecting some v' with the starting node v. While the selector is optional in our RPQ syntax, GQL and SQL/PGQ prohibit the WALK restrictor to be present without any selector attached to it, in order to ensure a finite result set. Therefore, in this paper we will assume that queries using the WALK restrictor will always have an associated selector. This gives rise to 11 total combinations of query prefixes to specify the type of path(s) that are to be returned for each node reachable by the $(v, \texttt{regex}, ?x)$ part of the query.

In this paper, we use $|[\![q]\!]_G|$ to denote the *size* of all the paths in $[\![q]\!]_G$. Notice that this is not simply the cardinality of the set $[\![q]\!]_G$, but instead, the sum of the lengths of all the paths present in $[\![q]\!]_G$. Since any algorithm for computing $[\![q]\!]_G$ has to, at the very least, write down each symbol on the paths in $[\![q]\!]_G$, this factor needs to be accounted for when analyzing the runtime of our algorithms.

Regular expressions and automata. We assume basic familiarity with regular expressions and finite state automata [32]. If $\texttt{regex}$ is a regular expression, we will denote by $\mathcal{L}(\texttt{regex})$ the language of $\texttt{regex}$. We use $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ to denote a non-deterministic finite automaton (NFA). Here Q is a set of states, Σ a finite alphabet of edge labels, δ the transition relation over $Q \times \Sigma \times Q$, q_0 the initial state, and F the set of final states, respectively. An NFA is deterministic (or DFA for short), if δ is a function. Each $\texttt{regex}$ can be converted into an equivalent NFA of size linear in $|\texttt{regex}|$. There are several standard ways of converting an expression to an automaton, for instance using Thompson's, or Glushkov's construction [32]. In this paper, we assume that the automaton has a single initial state and that no ε-transitions are present. An NFA is called *unambiguous* if it has at most one accepting run for every word. Every DFA is unambiguous, but the converse is not necessarily true.

3 THE WALK SEMANTICS

In this section we describe algorithms for evaluating RPQs under the WALK semantics. That is, we treat queries of the form:

$$\mathsf{selector\ WALK}\ (v, \texttt{regex}, ?x)$$

Notice that here the selector part is non-optional, since the set of all walks can be infinite. One method for evaluating RPQs, long standing in the theoretical literature [5, 26, 27], is known under the name product construction, or product graph, and since it will form the basis of many algorithms we present, we describe it next.

Product graph. Given a graph database $G = (V, E, \rho, \lambda)$, and an expression of the form $q = (v, \texttt{regex}, ?\texttt{x})$[3], the *product graph* is constructed by first converting the regular expression $\texttt{regex}$ into an equivalent non-deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$. The product graph $G_\times$, is then defined as the graph database $G_\times = (V_\times, E_\times, \rho_\times, \lambda_\times)$, where:

- $V_\times = V \times Q$
- $E_\times = \{(e, (q_1, a, q_2)) \in E \times \delta \mid \lambda(e) = a\}$
- $\rho_\times(e, d) = ((x, q_1), (y, q_2))$ if:
  - $d = (q_1, a, q_2)$
  - $\lambda(e) = a$
  - $\rho(e) = (x, y)$
- $\lambda_\times((e, d)) = \lambda(e)$.

Intuitively, the product graph is the graph database obtained by the cross product of the original graph and the automaton for $\texttt{regex}$. Considering the expression $q = (v, \texttt{regex}, ?\texttt{x})$, we can now find all v' reachable from v in the original graph database G via a path whose label belongs to $\mathcal{L}(\texttt{regex})$, by detecting all nodes of the form $(v', q') \in G_\times$, with $q' \in F$ that are reachable (by any path) from (v, q_0). Notice that in $G_\times$ we need not worry whether the path belongs to $\mathcal{L}(\texttt{regex})$, since this is guaranteed by the product construction, which mimics the standard cross product of two NFAs [32]. Therefore all such v's can be found by using any standard graph search algorithm (BFS/DFS) on $G_\times$ starting in the node (v, q_0). A by-product of this approach is that it is not necessary to construct $G_\times$ in its entirety, but rather, we build (a portion of) $G_\times$ on-the-fly, as we are exploring the product graph. Next, we show how the product construction can be used to return paths.

[2] Notice that we assume that the starting node in an RPQ is fixed. RPQs can generally also have a variable in the place of v, but we consider the more limited case because it extends itself naturally to traditional graph search algorithms.

[3] Notice that the product construction does not depend on selectors or restrictors, since it was traditionally not used to return paths.

3.1 ANY (SHORTEST) WALKS

We begin by treating the WALK restrictor combined with selectors ANY and ANY SHORTEST. Namely, we provide the algorithm for queries of the form:

$$q = \text{ANY (SHORTEST)? WALK } (v, \texttt{regex}, ?x) \qquad (1)$$

The idea here follows directly from the product construction; namely, given a graph database G and a query q as described above, we can perform a classical graph search algorithm such as BFS or DFS starting at the node (v, q_0) of the product graph $G_\times$, built from the automaton $\mathcal{A}$ for `regex` and G. Since both these algorithms also support reconstructing a single (shortest in the case of BFS) path to any reached node, we obtain the desired semantics for RPQs of the form (1). Our solution is presented in Algorithm 1.

Algorithm 1 Evaluation for a graph database $G = (V, E, \rho, \lambda)$ and an RPQ *query* = ANY (SHORTEST)? WALK $(v, \texttt{regex}, ?x)$.

```
1:  function AnyWalk($G$, $query$)
2:      $\mathcal{A} \leftarrow$ Automaton(regex)      ▹ $q_0$ initial state, $F$ final states
3:      Open.init()
4:      Visited.init()
5:      ReachedFinal.init()
6:      startState $\leftarrow (v, q_0, null, \bot)$
7:      Visited.push(startState)
8:      Open.push(startState)
9:      if $v \in V$ and $q_0 \in F$ then
10:         ReachedFinal.add($v$)
11:         Solutions.add($v$)
12:     while Open $\neq \emptyset$ do
13:         current $\leftarrow$ Open.pop()      ▹ current = $(n, q, edge, prev)$
14:         for next = $(n', q', edge') \in$ Neighbors(current, $G$, $\mathcal{A}$) do
15:             if $(n', q', *, *) \notin$ Visited then
16:                 newState $\leftarrow (n', q', edge',$ current)
17:                 Visited.push(newState)
18:                 Open.push(newState)
19:                 if $q' \in F$ and $n' \notin$ ReachedFinal then
20:                     ReachedFinal.add($n'$)
21:                     path $\leftarrow$ getPath(current).extend($n', edge'$)
22:                     Solutions.add(path)
        return Solutions

23: function getPath(state = $(n, q, edge, prev)$)
24:     if $prev == \bot$ then                  ▹ Initial state
25:         return $[v]$
26:     else                                    ▹ Recursive backtracking
27:         return getPath($prev$).extend($n, edge$)
```

The basic object we will be manipulating is a *search state*, which is simply a quadruple of the form $(n, q, e, prev)$, where n is a node of G we are currently exploring, q is the state of $\mathcal{A}$ in which we are currently located, while e is the edge of G we used to reach n, and *prev* is a pointer to the search state we used to reach (n, q) in $G_\times$. Intuitively, the (n, q)-part of the search state allows us to track the node of $G_\times$ we are traversing, while e, together with *prev* allows to reconstruct the path from (v, q_0) that we used to reach (n, q). The algorithm then needs the following three data structures:

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

The algorithm explores the product of G and $\mathcal{A}$ using either BFS or DFS, starting from (v, q_0). In line 6 we initialize a search state based on (v, q_0), and add it to Open and Visited. Lines 9–11 check whether the zero-length path containing v is an answer (in which case v needs to be a node in G). The main loop of line 12 is the classical BFS/DFS algorithm that pops an element from Open (line 13), and starts exploring its neighbors in $G_\times$. When exploring each state $(n, q, e, prev)$ in Open, we scan all the transitions (q, a, q') of $\mathcal{A}$ that originate from q, and look for neighbors of n in G reachable by an a-labeled edge (line 14). Here, writing $(n', q', edge') \in \text{Neighbors}((n, q, edge, prev), G, \mathcal{A})$ simply means that $\rho(edge') = (n, n')$ in G, and that $(q, \lambda(edge'), q')$ belongs to the transition relation of $\mathcal{A}$. If the pair (n', q') has not been visited yet, we add it to Visited and Open, which allows it to be expanded later on in the algorithm (lines 15–18). Furthermore, if q' is also a final state, and n' was not yet added to ReachedFinal (line 19), we found a new solution; meaning a WALK from v to n' whose label is in the language of `regex`. This walk can then be reconstructed using the *prev* part of search states stored in Visited in a standard fashion (function getPath). Finally, we need to explain why the ReachedFinal set is used to store each previously undiscovered solution in line 20. Basically, since our automaton is non-deterministic and can have multiple final states, two things can happen:

(i) The automaton might be ambiguous, meaning that there could be two different runs of the automaton that accept the same word in $\mathcal{L}(\texttt{regex})$. This, in turn, could result in the same path being returned twice, which is incorrect.

(ii) There could be two different paths p and p' in G that link v to some n, and such that both $\text{lab}(p) \in \mathcal{L}(\texttt{regex})$ and $\text{lab}(p') \in \mathcal{L}(\texttt{regex})$, but the accepting runs of $\mathcal{A}$ on these two words end up in different end states of $\mathcal{A}$. Again, this could result with n and a path to it being returned twice.

Both of these problems are solved by using the ReachedFinal set, which stores a node the first time it is returned as a query answer (i.e. as an endpoint of a path reaching this node). Then, each time we try to return the same node again, we do so only if the node was not returned previously (line 19). This basically means that for each node that is a query answer, a single path is returned, without any restrictions on the automaton used for modeling the query.

The described procedure continues until Open is empty, meaning that there are no more states to expand and all reachable nodes have been found with a WALK from the starting node v. It is important to mention that, when choosing to use BFS, this algorithm has the added benefit of returning the SHORTEST WALK to every reachable node, since it always prioritizes the shortest paths when traversing the product graph. For DFS an arbitrary path is returned.

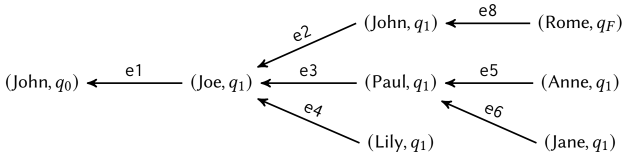
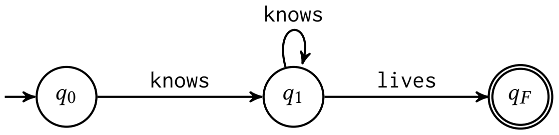


Figure 2: Visited after running Algorithm 1 in Example 3.1.

Example 3.1. Consider again the graph G in Figure 1, and let

$$q = \text{ANY SHORTEST WALK } (\text{John}, \text{knows}^+ \cdot \texttt{lives}, ?x).$$

Namely, we wish to find places where friends of John live. Looking at the graph in Figure 1, we see that Rome is such a place, and the shortest path reaching it starts with John, and loops back to him using the edges e1 and e2, before reaching Rome (via e8), as required. To compute the answer, Algorithm 1 first converts the regular expression knows$^+$ · `lives` into the following automaton:



To find shortest paths, we use the BFS version of Algorithm 1 and explore the product graph starting at (John, q_0). The algorithm then explores the only reachable neighbor (Joe, q_1), and continues by visiting (John, q_1), (Paul, q_1) and (Lily, q_1). When expanding (John, q_1) the first solution, Rome, is found and recorded in ReachedFinal. The algorithm continues by reaching (Anne, q_1) and (Jane, q_1) from (Paul, q_1). When the (Lily, q_1) node is then expanded, it would try to reach (Jane, q_1) again, which is blocked in line 15. Expanding (Anne, q_1) would try to revisit Rome, but since this solution was already returned, we ignore it. The structure of Visited upon executing the algorithm is illustrated in Figure 2. Here we represent the pointer *prev* as an arrow to other search states in Visited, and annotate the arrow with the edge witnessing the connection. Notice that we can revisit a node of G (e.g. John), but not a node of $G_\times$ (e.g. (Jane, q_1)). □

We can summarize the results about Algorithm 1 as follows:

Theorem 3.2. *Let G be a graph database, and q the query:*

$$\text{ANY (SHORTEST)? WALK } (v, \texttt{regex}, ?x).$$

If $\mathcal{A}$ is the automaton for `regex`*, then Algorithm 1 correctly computes $[\![q]\!]_G$ in time $O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|)$.*

3.2 ALL SHORTEST WALKS

To fully cover the WALK semantics of RPQs, we next show how to evaluate queries of the form:

$$q = \text{ALL SHORTEST WALK } (v, \texttt{regex}, ?x) \qquad (2)$$

over a graph database G. For this, we will extend the BFS version of Algorithm 1 in order to support finding *all* shortest paths between a pair (v, v') of nodes, instead of a single one. Intuition here is quite simple: to obtain all shortest paths, upon reaching v' from v by a path conforming to `regex` *for the first time*, the BFS algorithm will do so using a shortest path. The length of this path can then be recorded (together with v'). When a new path reaches the same, already visited node v', if it has the length equal to the recorded length for v', then this path is also a valid answer to our query. The algorithm implementing this logic is presented in Algorithm 2.

Algorithm 2 Evaluation algorithm for a graph database G and an RPQ *query* = ALL SHORTEST WALK $(v, \texttt{regex}, ?x)$.

```
function AllShortestWalk($G$, $query$)
    $\mathcal{A} \leftarrow$ Automaton(regex)    ▷ $q_0$ initial state, $q_F$ final state
    Open.init()
    Visited.init()
    if $v \in V$ then
        startState $\leftarrow (v, q_0, 0, \perp)$
        Visited.push(startState)
        Open.push(startState)
    while Open $\neq \emptyset$ do
        current $\leftarrow$ Open.pop() ▷ current = $(n, q, depth, prevList)$
        if $q == q_F$ then
            currentPaths $\leftarrow$ getAllPaths(current)
            Solutions.add(currentPaths)
        for next = $(n', q', edge') \in$ Neighbors(current, $G$, $\mathcal{A}$) do
            if $(n', q', *, *) \in$ Visited then
                $(n', q', depth', prevList') \leftarrow$ Visited.get($n'$, $q'$)
                if $depth + 1 == depth'$ then
                    $prevList'$.add($\langle$ current, $edge' \rangle$)
            else
                $prevList$.init()
                $prevList$.add($\langle$ current, $edge' \rangle$)
                newState $\leftarrow (n', q', depth + 1, prevList)$
                Visited.push(newState)
                Open.push(newState)
    return Solutions

function getAllPaths(state = $(n, q, depth, prevList)$)
    if $prevList == \perp$ then                ▷ Initial state
        return $[v]$
    for prev = $(prevState, prevEdge) \in prevList$ do
        for prevPath $\in$ getAllPaths($prevState$) do
            paths.add(prevPath.extend($n$, $prevEdge$))
    return paths
```

As before, we use $\mathcal{A}$ to denote the NFA for `regex`. We will additionally assume that $\mathcal{A}$ is unambiguous and that it has a single accepting state q_F. The main difference to Algorithm 1 is in the *search state* structure. A search state is now a quadruple of the form $(n, q, depth, prevList)$, where:

- n is a node of G and q a state of $\mathcal{A}$;
- *depth* is the length of (any) shortest path reaching n from v; and
- *prevlist* is a *list* of pointers to any previous search state that allows us to reach n via a shortest path.

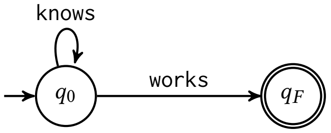
We assume that *prevList* is a *linked list*, initialized as an empty node, and accepting sequential insertions of pairs $\langle searchState, edge \rangle$ through the add() method. Intuitively, *prevList* will allow us to reconstruct *all* the shortest paths reaching a node, since there can be multiple ones. When adding a pair $\langle searchState, edge \rangle$, we assume *searchState* to be a pointer to a previous search state, and *edge* will be used to reconstruct the path passing through the node in this previous search state. Finally, we again assume that Visited is a dictionary of search states, with the pair (n, q) being the search key. Namely, there can be at most one tuple $(n, q, depth, prevList)$ in Visited with the same pair (n, q). We assume that with Visited.get(n, q), we will obtain the unique search state having n and q as the first two elements, or a null pointer if no such search state exists.

Algorithm 2 explores the product graph obtained from G and $\mathcal{A}$ using BFS, since it needs to find shortest paths. Therefore, we assume Open to be a queue. The execution is very similar to Algorithm 1, with a few key differences. First, if a node (n', q') of the product graph $G_\times$ has already been visited (line 15), we do not directly discard the new path, but instead choose to keep it if and only if it is also shortest (line 17). In this case, the *prevList* for n' is extended by adding the new path (line 18). If a new pair (n', q') is discovered for the first time, a fresh *prevList* is created (lines 19–24). The second difference to Algorithm 1 lies in the fact that we now return solutions only after a state has been removed from Open (lines 10–11). Basically, when a state is popped from the queue, the structure of the BFS algorithm assures that we already explored all shortest paths to this state. We enumerate all the shortest paths reaching this node using GETALLPATHS, which applies backtracking to be able to compute all *walks* from the starting node to the current node of interest. The following example illustrates what happens when there are multiple shortest paths between two nodes.

Example 3.3. Consider again the graph G in Figure 1, and let

$$q = \text{ALL SHORTEST WALK } (\text{Joe}, \text{knows}^* \cdot \text{works}, ?x).$$

In the Introduction, we showed there are three paths in $[\![q]\!]_G$, all three linking Joe to ENS Paris. The three paths are traced by the edges e3 → e5 → e11, e4 → e7 → e10, and e3 → e6 → e10, respectively. To compute $[\![q]\!]_G$, Algorithm 2 converts the regular expression knows* · works into the following automaton $\mathcal{A}$:



Algorithm 2 then starts traversing the product graph $G_\times$ by pushing the node (Joe, q_0) to both Open and Visited (together with distance 0, and empty *prevlist*). When this search state is popped from the queue, its neighbors in $G_\times$, namely (Paul, q_0), (Lily, q_0) and (John, q_0) are pushed to both Visited and Open, with $depth$ = 1 and *prevList* pointing to (Joe, q_0). The algorithm proceeds by visiting (Anne, q_0) from (Paul, q_0). Similarly, (Jane, q_0) is visited from (Paul, q_0). The interesting thing happens in the next step when (Lily, q_0) is the node being expanded to its neighbor (Jane, q_0), which is already present in Visited. Here we trigger lines 15–18 of the algorithm for the first time, and update the *prevList* for (Jane, q_0), instead of ignoring this path as Algorithm 1 does. When we try to explore neighbors of (John, q_0), we try to revisit (Joe, q_0), so lines 15–18 are triggered again. This time the depth test in line 17 fails (we visited Joe with a length 0 path already), so this path is abandoned. We then explore the node (ENS, q_F) in $G_\times$ by traversing the neighbors of (Anne, q_0). Finally, (ENS, q_F) will be revisited as a neighbor of (Jane, q_0) on two different shortest paths.

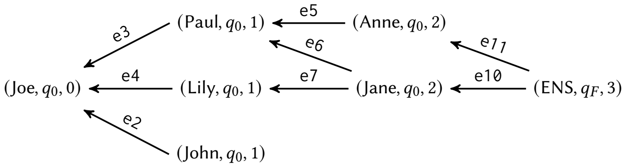


Figure 3: Visited after running Algorithm 2 in Example 3.3.

Figure 3 shows Visited upon executing the algorithm. Here we represent *prevList* as a series of arrows to other states in Visited, and only draw $(n, q, depth)$ in each node. For instance, (Jane, q_0, 2) has two outgoing edges, representing two pointers in its *prevList*. The arrow is also annotated with the edge witnessing the connection (as stored in the search state). Calling GETALLPATHS in the final search state; namely the one with (ENS, q_F, 3) in the first three components, will then correctly enumerate the three shortest paths connecting Joe to ENS Paris. □

For Algorithm 2 to work correctly, we crucially need $\mathcal{A}$ to be both unambiguous and have a single accepting state. It is not difficult to show that multiple accepting states can be supported by having a reachedFinal set similar as in Algorithm 1 (with a few extra complications), however, the ambiguity condition cannot be lifted. However, this is not a huge deal, since the vast majority of RPQs in practice are, in fact, unambiguous [7]. Summing up, we obtain:

THEOREM 3.4. *Let G be a graph database, and q the query:*

$$\text{ALL SHORTEST WALK } (v, \text{regex}, ?x).$$

If the automaton $\mathcal{A}$ for regex *is unambiguous and has a single accepting state, then Algorithm 2 correctly computes $[\![q]\!]_G$ in time $O(|\mathcal{A}| \cdot |G| + |[\![q]\!]_G|)$.*

The $O(|\mathcal{A}| \cdot |G|)$ factor is the cost of running traditional BFS on $G_\times$. This is because the nodes of $G_\times$ we revisit (lines 15–18) do not get added to the queue Open again, so we explore the same portion of $G_\times$ as in Algorithm 1. However, we can potentially add extra edges to Visited, so $[\![q]\!]_G$ can be bigger than for ANY SHORTEST WALK mode. Also notice that GETALLPATHS will traverse each path in $[\![q]\!]_G$ precisely once, making the enumeration of results optimal.

4 TRAIL, SIMPLE AND ACYCLIC

In this section we devise algorithms for finding trails, simple paths, or acyclic paths. It is well established in the research literature that even checking whether there is a single path between two nodes that conforms to a regular expression and is a simple path is NP-complete [5, 10]. It is not difficult to see that an equivalent result holds for trails and acyclic paths. Therefore, we know that, in essence, the "best" known algorithm for finding such paths is a brute-force enumeration of all possible candidate paths in the product graph. Our algorithms will follow this intuition and prune the search space whenever possible. Of course, in the worst case all such algorithms will be exponential; however, as we later show, on real-world graphs, the number of paths will not be so big, and the pruning technique will ensure that all dead-ends are discarded. We begin by showing how to find *all* trails and simple/acyclic paths.

4.1 Returning all paths

We start by dealing with queries of the form:

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

where restrictor is TRAIL, SIMPLE, or ACYCLIC. In Algorithm 3 we present a solution for computing the answer of q over a graph database G. Intuitively, when run over G, Algorithm 3 will construct the automaton $\mathcal{A}$ for regex with the initial state q_0. It will then start enumerating all paths in the product graph of G and $\mathcal{A}$ starting in (v, q_0), and discarding ones that do not satisfy the restrictor of q. To ensure correctness, we will need the automaton to be unambiguous, and will keep the following auxiliary structures:

- *search state*, is a tuple $(n, q, depth, e, prev)$, where n is a node, q an automaton state, $depth$ the length of the shortest path reaching n in G, e an edge used to reach the node n; and $prev$ a pointer to another search state stored in Visited.
- Visited is a set storing already explored search states.
- ReachedFinal is a *dictionary* of pairs $(n, depth)$, where n is a node reached in some query answer, and $depth$ the length of the shortest paths to this node. Here n is the dictionary key, so ReachedFinal.get(n) returns the pair $(n, depth)$.

Algorithm 3 explores the product graph by enumerating all paths starting in (v, q_0). For this, we can use a breadth-first (Open is a queue), or a depth-first (Open is a stack) strategy. For the ALL SHORTEST selector, a queue needs to be used. The execution is very similar to Algorithm 1, but Visited is not used to discard solutions. Instead, every time a state from Open is expanded (lines 13–15), we check if the resulting path satisfies the restrictor of q via the isValid function. The function traverses the path defined by the search states stored in Visited recursively. Here we are checking whether the path in the *original graph G* satisfies the restrictor, and not the path in the product graph. If the explored neighbor allows to extend the current path according to the restrictor, we add the new search state to Visited, and Open (lines 17–18). If we reach a final state of the automaton (line 19), we retrieve the explored path using the getPath function, which is identical to the one from Algorithm 1, but now uses extended search states. Since the *prev* pointer in each search state is unique, this is always a single path. If the ALL SHORTEST selector is *not* present, we simply add the newly found solution (lines 21–22). In the presence of the selector, we need to make sure to add only *shortest* paths to the solution set. The dictionary ReachedFinal is used to track discovered nodes, and stores the length of the shortest path to this node. If the node is seen for the first time, the dictionary is updated, and a new solution added (lines 23–25). Upon discovering the same node again (lines 26–28), a new solution is added only if it is shortest.

Algorithm 3 Evaluation algorithm for a graph database G and an RPQ $query$ = (ALL SHORTEST)? restrictor $(v, \texttt{regex}, ?x)$. Here restrictor is a string and ALL SHORTEST is a Boolean value.

```
 1: function AllRestrictedPaths($G$, $query$)
 2:     $\mathcal{A} \leftarrow$ Automaton($regex$)        ▷ $q_0$ initial state, $F$ final states
 3:     Open.init()
 4:     Visited.init()
 5:     ReachedFinal.init()
 6:     startState $\leftarrow (v, q_0, 0, null, \bot)$
 7:     Visited.push(startState)
 8:     Open.push(startState)
 9:     if $v \in V$ and $q_0 \in F$ then
10:         ReachedFinal.add($\langle v, 0 \rangle$)
11:         Solutions.add($v$)
12:     while Open $\neq \emptyset$ do
13:         current $\leftarrow$ Open.pop()   ▷ current = $(n, q, depth, e, prev)$
14:         for next = $(n', q', edge')$ $\in$ Neighbors(current, $G$, $\mathcal{A}$) do
15:             if isValid(current, next, restrictor) then
16:                 new $\leftarrow (n', q', depth + 1, edge'$, current)
17:                 Visited.push(new)
18:                 Open.push(new)
19:                 if $q' \in F$ then
20:                     path $\leftarrow$ getPath(new)
21:                     if !(ALL SHORTEST) then
22:                         Solutions.add(path)
23:                     else if $n' \notin$ ReachedFinal then
24:                         ReachedFinal.add($\langle n', depth + 1 \rangle$)
25:                         Solutions.add(path)
26:                     else
27:                         optimal $\leftarrow$ ReachedFinal.get($n'$).$depth$
28:                         if $depth + 1$ == optimal then
29:                             Solutions.add(path)
    return Solutions

30: function isValid(state, next, restrictor)
31:     s $\leftarrow$ state
32:     while s $\neq \bot$ do
33:         if restrictor == ACYCLIC then
34:             if s.$node$ == next.$node$ then
35:                 return False
36:         else if restrictor == SIMPLE then
37:             if s.$node$ == next.$node$ and s.$prev \neq \bot$ then
38:                 return False
39:         else if restrictor == TRAIL then
40:             if s.$edge$ == next.$edge$ then
41:                 return False
42:         s $\leftarrow$ s.$prev$
    return True
```

Example 4.1. Consider again the graph G in Figure 1, and

$$q = \text{SIMPLE (John}, \text{knows}^+ \cdot \texttt{lives}, ?x).$$

Namely, we wish to use the same regular pattern as in Example 3.1, but now allowing only simple paths. Same automaton as in Example 3.1 is used in Algorithm 3. The algorithm will start by visiting (John, q_0), followed by (Joe, q_1). After this we will visit (John, q_1), (Paul, q_1) and (Lily, q_1). In the next step we will try to expand (John, q_1), but will detect that this leads to a path which is not simple (we could have detected this in the previous step, but this way the pseudo-code is more compact). We will continue exploring neighbors, building the Visited structure depicted in Figure 4. In Figure 4 we use the same notion for *prev* pointers as in previous examples. For brevity, we do not show *depth*, but this is simply the length of the path needed to reach (John, q_0) in Figure 4. We remark that the node (Jane, q_1) appears twice since it will be present in the search state $(\text{Jane}, q_1, 3, \text{e6}, prev)$ and in $(\text{Jane}, q_1, 3, \text{e7}, prev')$. □

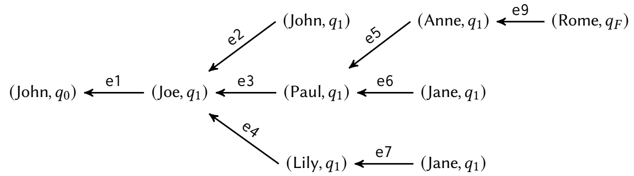


Figure 4: Visited after running Algorithm 3 in Example 4.1.

The correctness of the algorithm crucially depends on the fact that $\mathcal{A}$ is unambiguous, since otherwise we could record the same solution twice. Termination is assured by the fact that eventually all paths that are valid according to the restrictor will be explored, and no new search states will be added to Open. Unfortunately, since we will potentially enumerate all the paths in the product graph, the complexity is exponential. More precisely, we obtain the following:

Theorem 4.2. *Let G be a graph database, and q the query:*

$$(\mathsf{ALL\ SHORTEST})?\ \mathsf{restrictor}\ (v, \mathtt{regex}, ?x),$$

where restrictor *is* TRAIL, SIMPLE*, or* ACYCLIC*. If the automaton $\mathcal{A}$ for* regex *is unambiguous, then Algorithm 3 correctly computes $[\![q]\!]_G$ in time $O\big((|\mathcal{A}| \cdot |G|)^{|G|} + |[\![q]\!]_G|\big)$.*

4.2 ANY and ANY SHORTEST

To treat queries of the form:

$$q = \mathsf{ANY}\ (\mathsf{SHORTEST})?\ \mathsf{restrictor}\ (v, \mathtt{regex}, ?x)$$

where restrictor is TRAIL, SIMPLE, or ACYCLIC, minimal changes to Algorithm 3 are required. Namely, for this case we would assume that ReachedFinal is a *set* instead of a dictionary, and that it only stores nodes n reachable from v by a path conforming to restrictor, and whose label belongs to $\mathcal{L}(\mathtt{regex})$. We would then replace lines 21–29 of Algorithm 3 with the following:

```
if $n' \notin$ ReachedFinal then
    ReachedFinal.add($n'$)
    Solutions.add(path)
```

Basically, when n' is discovered as a solution for the first time, we return a path associated with it, and never return a path reaching n' as a solution again. To make ANY SHORTEST work correctly, we need to use BFS (i.e. Open is a queue), while for ANY we can use either BFS or DFS. Unfortunately, due to the aforementioned results of [10] stating that checking the existence of a simple path between a fixed pair of nodes is NP-complete, we cannot simplify the brute-force search of Algorithm 3. One interesting feature is that, due to the fact that ReachedFinal is a set, and therefore for each solution node a single path is returned, we no longer need the requirement that $\mathcal{A}$ be unambiguous. That is, we obtain:

Theorem 4.3. *Let G be a graph database, and q the query:*

$$\mathsf{ANY}\ (\mathsf{SHORTEST})?\ \mathsf{restrictor}\ (v, \mathtt{regex}, ?x),$$

where restrictor *is* TRAIL, SIMPLE*, or* ACYCLIC*. Then we can compute $[\![q]\!]_G$ in time $O\big((|\mathcal{A}| \cdot |G|)^{|G|} + |[\![q]\!]_G|\big)$.*

5 IMPLEMENTATION DETAILS

We implemented the algorithms of Section 3 and Section 4 inside of MillenniumDB [47], a recent open-source graph database engine developed by the authors. MillenniumDB provides the infrastructure necessary to process generic queries such as RPQs, and takes care of parsing, generation of execution plans, and data storage. The source code of our implementation can be obtained at [14].

Data access. By default, MillenniumDB stores graph data on disk using B+trees, and loads the necessary pages into a main memory buffer during query execution. MillenniumDB indexes several relations, however, our algorithms only use the following two:

Edges(NodeFrom, Label, NodeTo, EdgeId)
Nodes(NodeId)

The first relation allows us to find neighbors of a certain node reachable by an edge with a specific label, as used, for instance, in Algorithm 1 (line 14). The second table keeps track of nodes that are materialized in the database. In essence, the table Nodes is only used when checking whether the start node of an RPQ actually exists in the database (see e.g. Algorithm 1, line 9). All B+trees in MillenniumDB are accessed using the standard linear iterator interface used in relational databases [31]. Additionally, all of the proposed algorithms can be easily extended with the ability to traverse edges backwards, as required, for instance, by C2RPQs [9] and SPARQL property paths [21]. To support this, MillenniumDB also includes the inverse Edges relation, with the following schema:

Edges^{-}(NodeTo, Label, NodeFrom, EdgeId).

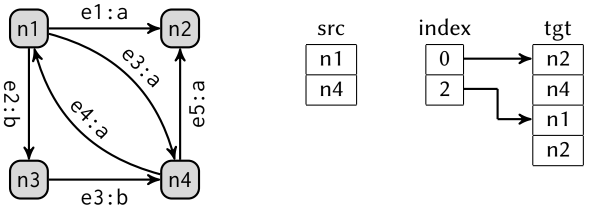
We can then easily extend all of the described algorithms to allow RPQs with backward edges, and use the EDGES^{-} table whenever looking for a neighbor accessed via a backward edge.

Compressed Sparse Row. In addition to B+trees, we also support data access via the *Compressed Sparse Row (CSR)* representation [8], which is a popular in-memory index for fast access to edge data [42]. Basically, a CSR allows us to compress the standard matrix representation of a graph, which is of size $n \times n$ for a graph of n vertices, to roughly size $|E|$, where E is the set of edges, while still allowing fast access to each node's neighbors. For us, a CSR consists of three arrays:

- src, an array of source nodes, ordered by their identifier;
- index, an array of integers, where the number in position i tells us where the neighbors of the ith source node start in the array tgt; and
- tgt an array of target nodes.

Given that in regular path queries we only need to access neighbors reachable by a specific label, we support creating the CSR of a subgraph consisting of edges with this particular label, which can be much smaller than the CSR of the entire graph. As an illustration, Figure 5 shows a graph database (left), and its CSR for the edges labeled with a (right). Notice that the node n3 is never mentioned in the CSR since it has no a-labeled edges associated with it.

Basic intuition in this example is that, for instance, n4 is the second source node for a-labeled edges. Therefore in the second position of the index array, we store the index of the array tgt where we start listing the neighbors of n4 that can be reached by an a-labeled edge. In our example, these are n1 and n2.



Graph database G

CSR for the label a

Figure 5: A graph database and a CSR for the label a.

In our implementation, we will use CSR as an in-memory index for fast query evaluation, and will also allow to construct it on-the-fly as needed by the query. We remark that CSR is always created from the B+tree data residing on disk, and supports both backward and forward looking edges (each with their separate CSR).

Pipelined execution. In MillenniumDB all the algorithms are implemented in a pipelined fashion using linear iterators. In particular, this means that the solution is returned to the user as soon as it is encountered. This requires, for instance, that the main while loop of Algorithm 1 be halted as soon as a new solution is detected in line 19, and similarly for Algorithm 3. In the case of Algorithm 2 the situation is a bit more complex, as solutions can be detected while scanning the neighbors (lines 14–24), instead of upon popping an element from the queue (lines 11-13). All of these issues can be resolved by noting that linear iterators for neighbors of a node can be paused, and their execution resumed. For completeness, we include an expanded pseudo-code of our algorithms written in a pipelined fashion, in our online appendix [14]. The main benefit of the pipelined execution is that paths can be encountered on demand, and the entire solution set need not be constructed in advance.

Query variables. Throughout the paper we assumed that the RPQ part of our query takes the form $(v, \texttt{regex}, ?x)$, where v is a node identifier, and $?x$ is a variable. Namely, we were searching for all nodes reachable from a fixed point v. It is straightforward to extend our algorithms to patterns of the form $(v, \texttt{regex}, v')$, where both endpoints of the path are known. That is, we can run any of the algorithms as described in the paper, and check whether v' is a query answer.

6 EXPERIMENTAL EVALUATION

Here we empirically evaluate our hypothesis that classical search algorithms over graphs provide a good basis for returning paths that belong to an RPQ query answer. We start by describing the experimental setup, and then discuss the obtained results. The source code of our implementation, together with the information needed to replicate the experiments, can be downloaded at [14].

6.1 Experimental setup

Datasets and queries. For our experiments we use MillenniumDB Path Query Challenge [13], a benchmarking suite for path queries recently developed by the authors of this paper. The purpose of this query challenge is to test the performance of algorithms returning paths that witness RPQ query answers under different semantics, as described in Section 2. As such, [13] uses two data/query set combinations called Real-world and Synthetic. The challenge is based on a more extensive SPARQL benchmark called WDBench [1]. Next we describe the two test beds:

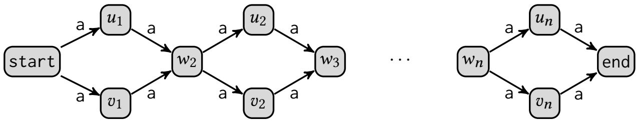


Figure 6: Graph database with exponentially many paths.

Real-world. The idea of this test is to gauge the potential of the described algorithms in a real-world setting. In terms of data, a curated version of the Wikidata [46] dataset is used. More precisely, we use the truthy dump of Wikidata, and keep only direct properties from this RDF dataset. This allows us to have the graph structure in the dataset, which is the only thing explored by the RPQs. Transforming this dataset into a graph database results in a graph with 364 million nodes and 1.257 billion edges. The dataset can be downloaded at [2]. As for queries, these are extracted from the real-world queries posted by Wikidata users, as found in Wikidata's query logs [25]. From these, a total of 659 RPQ patterns were selected, of which 592 have at least one endpoint fixed, and are thus supported by our algorithms. These 592 are used in our tests under the restrictor and selector options described in Section 2.

Synthetic. Here we check scalability by exhibiting a dataset with a large number of paths between two fixed nodes. The database used here, taken from [26], is presented in Figure 6. The single RPQ we use has the pattern $(\text{start}, \mathsf{a}^*, \text{end})$ and is combined with different selectors and restrictors. Notice that in our graph, all paths from start to end match a^*, and are, at the same time, shortest, trails and simple paths. Furthermore, there are 2^n of such paths, while the graph has only $3n + 1$ nodes and $4n$ edges. While returning all these paths is unfeasible for any algorithm, returning some portion of them (100,000 in our experiments) can be done efficiently by our algorithms. This test will also allow us to detect difficult cases for BFS vs DFS implementations we described thus far.

Tested systems. As described in Section 5, our implementation is tested with three different memory management strategies:

- B+tree version, which is the default (data on disk);
- CSR-c, which creates a CSR for each label as needed by the query, but caches the already created CSRs; and
- CSR-f version, which assumes CSRs for all labels to be available in main memory.

Additionally, we will use both a BFS traversal strategy and a DFS traversal strategy (when available) to run our algorithms.

In order to compare with state of the art in the area, we selected four publicly available graph database systems that allow for benchmarking with no legal restrictions. These are:

- Neo4J version 4.4.12 (Neo4j for short);
- Jena TDB version 4.1.0 [38] (Jena);
- Blazegraph version 2.1.6 [44] (BlazeGraph); and
- Virtuoso version 7.2.6 [12] (Virtuoso).

From the aforementioned systems, Neo4j uses the ALL TRAIL semantics by default, and also supports ANY SHORTEST WALK and ALL SHORTEST WALK. We therefore compare with Neo4j for these query modes. On the other hand, while Jena, BlazeGraph, and Virtuoso do not return paths, according to the SPARQL standard [21], they should be detecting the presence of an arbitrary walk, so we incorporate them in our comparison when testing the ANY WALK semantics. Other systems we considered are DuckDB [42], Oracle Graph Database [29] and Tiger Graph [41], which support (parts of) SQL/PGQ. Unfortunately, [29] and [41] are commercial systems with limited free versions, while the SQL/PGQ module for DuckDB [42] is still in development.

How we ran the experiments. All experiments were run on a commodity server with an Intel®Xeon®Silver 4110 CPU, and 128GB of DDR4/2666MHz RAM, running Linux Debian 10 with the kernel version 5.10. The hard disk used to store the data was a SEAGATE model ST14000NM001G with 14TB of storage. Note that this is a classical HDD, and not an SSD. Neo4j was used with default settings and no limit on RAM usage. Jena and BlazeGraph were assigned 64GB of RAM, and Virtuoso was set up with 64GB or more as recommended in the user instructions. MillenniumDB was allowed a 32GB buffer for handling the queries. Since we run large batches of queries (100+), these are executed in succession, in order to simulate a realistic load to a database system. When it comes to MillenniumDB, we run tests both with data residing on disk and being buffered into main memory, and with data being pre-loaded into CSRs. As recommended by WDBench [1] and MillenniumDB Path Query Challenge [13], *all queries were run with a limit of 100,000 results and a timeout of 1 minute.*

6.2 Real-world data

Due to the large number of different versions of the algorithms we presented, coupled with the different storage models used, we group our results by the path modes tested in each experiment.

ANY (SHORTEST)? WALK mode. Here we are testing the performance of Algorithm 1 from Subsection 3.1. When the BFS strategy is used, we obtain a shortest path for each reachable node, while DFS returns an arbitrary path. Here we compare with SPARQL systems, since they are required to find a single path for each reachable node. We discuss the results along the following dimensions:

- Runtime. The results are presented in Figure 7 (left). As we see, the six implementations of Algorithm 1 significantly outperform SPARQL systems. For different implementations of Algorithm 1, we can see that the B+tree versions are the slowest since they have to load all the data from disk to main memory, while having CSRs in main memory gives a significant boost to performance. Building CSRs on demand is in-between the two as expected. Comparing BFS vs. DFS based algorithms, we can observe that BFS-based ones are marginally faster.
- Memory. The total size of all CSRs was 4GB[4]. For B+tree versions, initializing MillenniumDB already uses around 9GB of buffer, while usage for the query runs topped at 11.8GB (BFS) and 11GB (DFS); meaning that only 2.8GB (BFS) and 2GB (DFS) were used for execution, showing that B+tree versions can take advantage of data locality in RPQs.

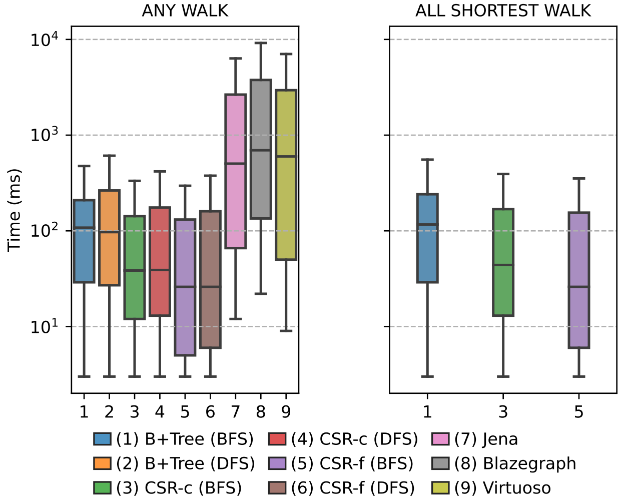


Figure 7: Runtime for the WALK semantics with a limit of 100,000 results and a timeout of 60 seconds.

- Timeouts. Implementations of Algorithm 1 had few timeouts. More precisely, the B+tree versions had one (BFS) and two (DFS) timeouts, while both CSR versions of DFS had one timeout. As far as SPARQL systems are concerned, Jena had 96 timeouts, BlazeGraph 87, and Virtuoso 24. Neo4j, using the ANY SHORTEST selector, timed out on all but two queries, so we leave it out of the comparison.

Overall, we can see that Algorithm 1 shows stable performance when a single path needs to be returned, both when coupled with standard B+tree storage, or in-memory indices such as CSRs.

ALL SHORTEST WALK mode. Next, we discuss the performance of Algorithm 2 from Subsection 3.2 as follows:

- Runtime. Runtimes are illustrated in Figure 7 (right). As we can see, the performance is similar as when returning a single path, as Theorem 3.4 would suggest.
- Memory. CSRs again use 4GB, while the B+tree version topped at 11.5GB, similar to the single path version.
- Timeouts. All versions had a single timeout on the same query. Neo4j also supports this path mode, but it timed out on all but two queries, so we exclude it from Figure 7.

Recall that for Algorithm 2 only BFS can be used. Interestingly, our results suggest that there is no significant difference between returning a single shortest path, or all shortest paths. We stress here that the limit of 100,000 applies to the returned paths (not the end nodes). Namely, if there are 100,000 shortest paths between two nodes, this will count as 100,000 results, while the ANY WALK mode would find 100,000 paths between different pairs of nodes.

ANY TRAIL and ANY SIMPLE modes. Next, we test the performance of returning a single trail or a single simple path. Here we use the variant of Algorithm 3 from Subsection 4.2. Since ACYCLIC evaluation is virtually identical to SIMPLE, we do not include it in our experiments. The results are as follows:

[4] Constructing all the CSRs from B+tree storage takes around 52 seconds. In cached CSR versions this time is included in the query execution.

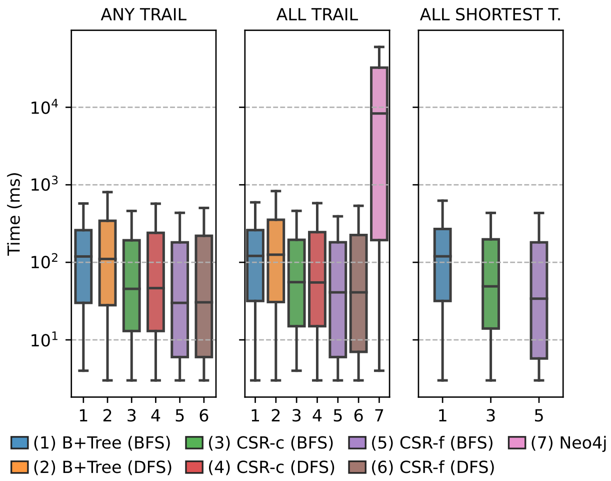


Figure 8: Runtime for the TRAIL semantics with a limit of 100,000 results and a timeout of 60 seconds.

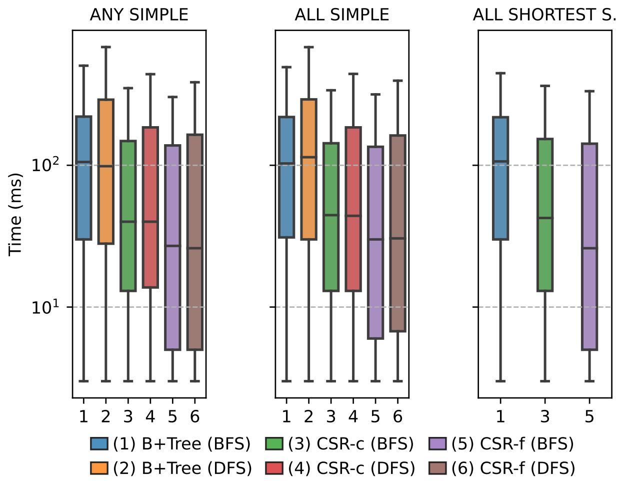


Figure 9: Runtime for the SIMPLE semantics with a limit of 100,000 results and a timeout of 60 seconds.

- Runtime. The runtimes for ANY TRAIL are reported in Figure 8 (left) and for ANY SIMPLE in Figure 9 (left). When BFS is used we get ANY SHORTEST. Unlike Theorem 4.3 would suggest, we see that the runtimes are only marginally slower than for the WALK restrictor. The main reason is that we are looking for the first 100,000 results, so the algorithm will not need to explore a huge set of paths, especially if the out-degree of explored nodes is not large.
- Memory. CSRs again use 4GB, while the B+tree BFS version topped at 13.9GB, and the DFS version at 11.4GB, only slightly more than for the WALK restrictor.
- Timeouts. All BFS-based versions timed out on 14 experiments, while the DFS ones did so on 13 of them.

Interestingly, while the theoretical literature classifies these evaluation modes as intractable [5], and algorithms proposed in Section 4 take a brute-force approach to solving them, over real-world data they do not seem to fare significantly worse than algorithms for the WALK restrictor. As already stated, this is most likely due to the fact that they can either detect 100,000 results rather fast, and abandon the exhaustive search, or because the data itself permits no further graph exploration.

Retrieving all TRAILS and SIMPLE paths. Unlike for the WALK restrictor, for TRAIL and SIMPLE, the number of paths is finite, so these restrictors can appear without an additional selector. Here we are testing the performance of Algorithm 3 from Subsection 4.1. The results are as follows:

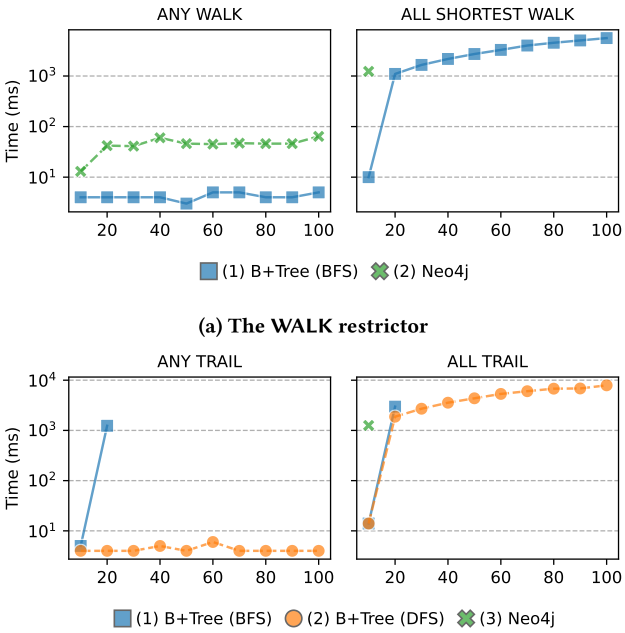
- Runtime. Times for retrieving all trails are reported in Figure 8 (middle), and for all simple paths in Figure 9 (middle). The runtimes are again similar as for ANY WALK, so finding the first 100,000 results seems not to be an issue. Comparing to Neo4j, in the TRAIL case (Figure 8 (middle)), we can see a significant improvement when Algorithm 3 is used.
- Memory. Memory usage for CSRs is as before, while for B+trees it topped at 11.6GB for BFS and 11.5GB for DFS, meaning 2.6GB (BFS) and 2.5GB (DFS) effective usage.
- Timeouts. All BFS versions timed out twice, and all DFS versions 3 times. Neo4j timed out in 99 cases.

The conclusions here are quite similar as for ANY TRAIL and ANY SIMPLE: despite the theoretical limitations of these algorithms, they seem to work sufficiently well for real-world use cases. Finally, the results for ALL SHORTEST TRAIL (Figure 8 (right)) and ALL SHORTEST SIMPLE (Figure 9 (right)), are analogous. We remark that unlike for e.g. ANY TRAIL, the BFS version for ALL TRAIL is not the same as ALL SHORTEST TRAIL, and similarly for SIMPLE, as discussed in Section 4.

6.3 Synthetic data

Here we test the performance of the query looking for paths between the node start and the node end in the graph of Figure 6. We scale the size of the database by setting $n = 10, 20, 30, \dots, 100$. This allows us to test how the algorithms perform when the number of paths is large, i.e. 2^n. For each value of n we will look for the first 100,000 results. Since all returned paths are shortest, trails, simple paths, and acyclic paths, for brevity we report only on performance considering the WALK restrictor and the TRAIL restrictor. This also gives us the opportunity to compare with Neo4j.

SHORTEST WALKS. The runtimes for the ANY SHORTEST WALK and ALL SHORTEST WALK modes is presented in Figure 10a. Due to the small size of the graph, we only test the B+tree version of our implementation, and perform a hot run; meaning that we run the query twice and report the second result. This is due to minuscule runtimes which get heavily affected by initial data loading. As we can observe, for ANY SHORTEST WALK (Figure 10a (left)) both engines perform well (we also pushed this experiment to $n = 1000$ with no issues). Memory usage is negligible in this experiment, and there are no timeouts. In the case of ALL SHORTEST WALK (Figure 10a (right)), Neo4j timed out on all but the smallest graph, while our implementation of Algorithm 2 shows stable performance when returning 100,000 paths. We tried scaling to $n = 1000$ and the results were quite similar, with almost no memory being used.



(a) The WALK restrictor

(b) The TRAIL restrictor

Figure 10: Runtime on the graph database of Figure 6.

TRAILS. This experiment allows us to determine which traversal strategy (BFS or DFS) is better suited for Algorithm 3 in extreme cases such as the graph of Figure 6. We present the results for ANY TRAIL and TRAIL modes in Figure 10b. Considering first the ANY TRAIL case, the BFS-based algorithm will time out already for $n = 30$, which is to be expected, since it will construct all paths of length $1, 2, \ldots, 29$, before considering the first path of length 30. In contrast, DFS will find the required paths rather fast. Memory usage also reflects this, since the BFS-based algorithm use 4.7GB of RAM when timing out, while DFS uses 0.4GB in all cases. When it comes to the TRAIL mode, which retrieves *all* trails, the situation is rather similar (here we also compare with Neo4j since it finds all trails). This shows us that for long paths DFS is the strategy of choice, since the BFS-based traversal needs to consider a huge number of paths before encountering the first query result. Memory usage here mimics the ANY TRAIL case. Overall, this experiment allows us to pinpoint problems that BFS-based approaches might have in Algorithm 3, particularly when there is a large number of paths that merge, and when the paths are of large length. Interestingly, this behavior never showed up in the Real-world dataset.

7 RELATED WORK

Our work is a continuation of [26], where we studied an abstract approach to RPQ query answering and *representing* paths in query answers. In the present paper we describe concrete algorithms for the RPQ evaluation modes considered in [26] (and multiple new ones), discuss how these can be implemented, and test their performance when integrated in different graph database pipelines. In terms of related work, we identify two main lines: the first one on evaluating RPQs, and the second one on reachability indices.

RPQ evaluation. Some of the most representative works in this area are [6, 15, 18, 19], where the focus is on finding nodes reachable by an RPQ-conforming path, and not on returning paths. Most of these works propose methods similar to Algorithm 1. In [20] the authors focus on retrieving paths, but not conforming to RPQ queries. An approach for finding top-k shortest paths in RPQ answers is explored in [34], using BFS-style search. Here the first k paths discovered by BFS are retrieved, so some non-shortest paths might be returned, unlike in the GQL and SQL/PGQ semantics we use in Algorithm 2. Interesting optimizations for the base BFS-style algorithm are presented in [24, 43], where vectorized execution of BFS with multiple starting points is explored. Since our focus is the performance of classical graph algorithms, we leave incorporating these optimizations into our workflow for future work. Regarding the simple path semantics, [48] proposes a sampling-based algorithm which is an efficient approximate answering technique, but will not necessarily find *all* answers. There is also a rich body of theoretical work on RPQ evaluation (see [5] for an overview).

Reachability indices. There is extensive work on speeding-up graph traversal algorithms via reachability indexes [36, 51]. Some of the more famous ones are [35, 45, 50], with [35] being incorporated in [19] to find nodes reachable by paths conforming to an RPQ. In terms of indices developed for regular path queries, apart from some early theoretical work [28], there is also a recent proposal [52], that allows quick detection of RPQ-conforming paths up to a fixed length. All of these works are somewhat orthogonal to our approach, since our main aim is to test algorithms for retrieving paths witnessing RPQ answers. Incorporating one or several of these indices into our algorithms is a promising direction of future work.

Our work. In summary, we believe our work to be the first to describe evaluation algorithms for regular path queries under *all* evaluation modes supported by the upcoming GQL and SQL/PGQ standards [11], and study how classical graph algorithms can be incorporated into a graph database pipeline to evaluate such queries.

8 CONCLUSIONS AND LOOKING AHEAD

In this paper we tested the hypothesis claiming that classical graph search algorithms can be adopted to find paths that are returned by RPQs under GQL and SQL/PGQ evaluation modes. For this, we used the standard BFS and DFS algorithms and adapted them to the setting of RPQ query answering. We also showed how such algorithms can be implemented, and performed an extensive experimental evaluation of their performance. We believe that the presented results confirm our initial hypothesis, and show that classical algorithms, at least in the case of queries that have a fixed starting point, are a valuable tool for evaluating RPQs. In future work we would like to extend the presented algorithms for the case when the starting point in the path is unknown, cover the GQL top-k semantics, and see how optimization techniques such as reachability indices [36, 52], or vectorized and parallel execution of BFS and DFS [24, 43] can speed up the algorithms we proposed.

ACKNOWLEDGMENTS

Work supported by ANID – Millennium Science Initiative Program – Code ICN17_002 and ANID Fondecyt Regular project 1221799.

REFERENCES

[1] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. WDBench: A Wikidata Graph Query Benchmark. In *The Semantic Web - ISWC 2022*. https://doi.org/10.1007/978-3-031-19433-7_41

[2] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. *WDBench: A Wikidata Graph Query Benchmark*. https://github.com/MilleniumDB/WDBench.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutiérrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD Conference 2018*.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017). https://doi.org/10.1145/3104031

[5] Pablo Barceló Baeza. 2013. Querying graph databases. In *PODS 2013*. 175–188. https://doi.org/10.1145/2463664.2465216

[6] Jorge A. Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč. 2017. Evaluating Navigational RDF Queries over the Web. In *Hypertext 2017*. https://doi.org/10.1145/3078714.3078731

[7] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[8] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[9] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2002. Rewriting of Regular Expressions and Regular Path Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 443–465.

[10] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *SIGMOD 1987*. 323–330.

[11] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22*. https://doi.org/10.1145/3514221.3526057

[12] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8. http://sites.computer.org/debull/A12mar/vicol.pdf

[13] Benjamín Farías, Carlos Rojas, and Domagoj Vrgoč. 2023. MillenniumDB Path Query Challenge. In *AMW 2023*.

[14] Benjamín Farías, Carlos Rojas, and Domagoj Vrgoč. 2023. Regular path queries in MillenniumDB. https://github.com/MillenniumDB/RPQPaper

[15] Valeria Fionda, Giuseppe Pirrò, and Claudio Gutiérrez. 2015. NautiLOD: A Formal Language for the Web of Data Graph. *ACM Trans. Web* 9, 1 (2015), 5:1–5:43.

[16] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. A Researcher's Digest of GQL (Invited Talk). In *ICDT 2023*. https://doi.org/10.4230/LIPIcs.ICDT.2023.1

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD 2018*. https://doi.org/10.1145/3183713.3190657

[18] Andrey Gubichev. 2015. *Query Processing and Optimization in Graph Databases*. Ph.D. Dissertation. Technical University Munich. https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20150625-1238730-1-7

[19] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *GRADES 2013*. https://doi.org/10.1145/2484425.2484443

[20] Andrey Gubichev and Thomas Neumann. 2011. Path Query Processing on Very Large RDF Graphs. In *WebDB 2011*. http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf

[21] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/

[22] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2022. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (2022), 71:1–71:37. https://doi.org/10.1145/3447772

[23] John et.al. Jumper. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (Aug. 2021), 583–589.

[24] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. 2017. Parallel Array-Based Single-and Multi-Source Breadth First Searches on Large Dense Graphs.. In *EDBT*. 1–12.

[25] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *ISWC 2018*.

[26] Wim Martens, Matthias Niewerth, Tina Popp, Carlos Rojas, Stijn Vansummeren, and Domagoj Vrgoč. 2023. Representing Paths in Graph Database Pattern Matching. *Proc. VLDB Endow.* 16, 7 (2023), 1790–1803. https://www.vldb.org/pvldb/vol16/p1790-martens.pdf

[27] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *VLDB 1989*. 185–193.

[28] Tova Milo and Dan Suciu. 1999. Index Structures for Path Expressions. In *ICDT '99*. https://doi.org/10.1007/3-540-49257-7_18

[29] Oracle [n.d.]. Oracle Graph Database. https://www.oracle.com/database/graph/.

[30] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[31] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems*. McGraw-Hill.

[32] Jacques Sakarovitch. 2009. *Elements of automata theory*. Cambridge University Press.

[33] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[34] Vadim Savenkov, Qaiser Mehmood, Jürgen Umbrich, and Axel Polleres. 2017. Counting to k or how SPARQL1.1 Property Paths Can Be Extended to Top-k Path Queries. In *SEMANTiCS 2017*. https://doi.org/10.1145/3132218.3132239

[35] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE 2013*. IEEE, 1009–1020.

[36] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2016. Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2016), 683–697.

[37] Amazon Neptune Team. 2021. What Is Amazon Neptune? https://docs.aws.amazon.com/neptune/latest/userguide/intro.html

[38] Jena Team. 2021. Jena TDB. https://jena.apache.org/documentation/tdb/

[39] MillenniumDB Team. 2021. MillenniumDB Source Code. https://github.com/MillenniumDB/MillenniumDB

[40] Stardog Team. 2021. Stardog 7.6.3 Documentation. https://docs.stardog.com/

[41] TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/

[42] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS. In *CIDR 2023*.

[43] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[44] Bryan B. Thompson, Mike Personick, and Martyn Cutcher. 2014. The Bigdata® RDF Graph Database. In *Linked Data Management*, Andreas Harth, Katja Hose, and Ralf Schenkel (Eds.). Chapman and Hall/CRC, 193–237. http://www.crcnetbase.com/doi/abs/10.1201/b16859-12

[45] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 845–856.

[46] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[47] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2021. MillenniumDB: A Persistent, Open-Source, Graph Database. *CoRR* abs/2111.01540 (2021). https://arxiv.org/abs/2111.01540

[48] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1463–1480.

[49] Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH '12*, Gary T. Leavens (Ed.). https://doi.org/10.1145/2384716.2384777

[50] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. 2010. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 276–284.

[51] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph reachability queries: A survey. *Managing and Mining Graph Data* (2010), 181–215.

[52] Chao Zhang, Angela Bonifati, Hugo Kapp, Vlad Ioan Haprian, and Jean-Pierre Lozi. 2022. A Reachability Index for Recursive Label-Concatenated Graph Queries. arXiv:2203.08606 [cs.DB]

A APPENDIX

B MISSING PROOFS

Proof sketch of Theorem 3.2. The $O(|\mathcal{A}| \cdot |G|)$ factor is the cost of running standard BFS/DFS algorithm on $G_\times$. Basically, we will visit each node (n, q) if $G_\times$ at most twice (lines 14–15 of Algorithm 1), and each edge (e, d) of $G_\times$ at most once (line 14). The correctness follows from the observation that each v' reachable from v by a path conforming to `regex` will be added to ReachedFinal once. The $O(|[\![q]\!]_G|)$ factor is the cost of writing down the paths in $[\![q]\!]_G$. Since GetPath procedure traverses each path precisely once, we get the desired result. □

Proof sketch of Theorem 3.4. The $O(|\mathcal{A}| \cdot |G|)$ factor stems from the fact that Algorithm 2 does precisely the same number of steps as Algorithm 1. Namely, Algorithm 2 is similar to running traditional BFS over $G_\times$, since the nodes of $G_\times$ we revisit (lines 15–18) do not get added to the queue Open again. The $O(|[\![q]\!]_G|)$ factor is the (optimal) cost of writing down all the paths in $[\![q]\!]_G$, and we get it since GetAllPaths will traverse each path in $[\![q]\!]_G$ precisely once. □

Proof sketch of Theorem 4.2 and Theorem 4.3. The correctness is explained in Section 4. For complexity, both versions of the algorithm (enumerating all, or a single path) basically need to iterate over all paths over the product graph. The stopping criterion is that the path be SIMPLE, TRAIL or ACYCLIC i the original graph. Therefore, the longest length will be $|G|$, and the total number of all paths in the worst case will be $(|\mathcal{A}| \cdot |G|)^{|G|}$, since we exploring paths in the product graph (so that the `regex` is satisfied by the path labels). In the worst case we need to consider all such paths. □