PathFinder: Returning Paths in Graph Queries

No Author Given

No Institute Given

Abstract. Path queries are a central feature of all modern graph query languages and standards, such as SPARQL, Cypher, SQL/PGQ, and GQL. While SPARQL returns *endpoints* of path queries, it is possible in Cypher, SQL/PGQ, and GQL to return *entire paths*. In this paper, we present the first framework for returning paths that match regular path queries under all fifteen modes in the SQL/PGQ and GQL standards. At the core of our approach is the product graph construction combined with a way to compactly represent a potentially exponential number of results that can match a path query. Throughout the paper we describe how this approach operates on a conceptual level and provide runtime guarantees for evaluating path queries. We also develop a reference implementation on top of an existing open-source graph processing engine, and perform a detailed analysis of path querying over Wikidata to gauge the usefulness of our methods in a real world scenario. Compared to several modern graph query engines, we obtain order-of-magnitude speedups and remarkably stable performance, even for theoretically intractable classes of path queries.

1 Introduction

Graph databases [4, 44] have gained significant popularity and are used in areas such as Knowledge Graph management [27], Semantic Web applications [5] and Biology [31]. The query languages for handling graph data include SPARQL [26], a W3C standard for querying RDF, and Cypher, GQL and SQL/PGQ [16, 23], which are part of the ISO initiative to standardize querying over property graphs.

All these languages have *path queries* as a core feature. In SPARQL, these are supported through *property paths*, which are a variant of *regular path queries* (*RPQs*) [13, 39, 7, 15]. Intuitively, an RPQ is an expression $(?x, \texttt{regex}, ?y)$, where `regex` is a regular expression and $?x, ?y$ are variables. When evaluated over an edge-labeled graph G (e.g., RDF or property graph), it extracts all node pairs (`n1`, `n2`) such that there is a path in G from `n1` to `n2`, whose edge labels form a word that matches `regex`. However, SPARQL property paths only return *endpoints* of paths. In important applications such as money laundering detection or in investigative journalism, where one is trying to understand the actual connections between entities, retrieving the path is of crucial importance.

To accommodate this use case, Cypher, GQL, and SQL/PGQ all extend RPQs with a mechanism for returning paths. To avoid having to return an infinite number of paths (which can happen if the graph is cyclic), these languages

let the user specify restrictions on the type of paths to be returned. These are combinations of e.g., *simple* (no duplicate nodes), *trail* (no duplicate edges), *shortest* (shortest paths that match the expression), or *any* (just return a single, arbitrarily chosen path).

While Cypher, GQL, and SQL/PGQ all recognize the need to return paths that match regular path queries, the support for these features in modern graph databases is surprisingly lacking. As already stated, in SPARQL engines, RPQ matching is fully supported, but paths cannot be returned. In property graph engines that implement (fragments of) GQL or SQL/PGQ, only a handful of path modes is supported, and no property graph engine to date supports all regular path queries like SPARQL engines do. For reference, we summarize the support for path features in several popular SPARQL and property graph engines in the table below. Here, "RPQ" means the support for all regular path queries, while the other columns signal which types of paths can be returned.

	RPQ	WALK	TRAIL	SIMPLE	ACYCLIC	SHORTEST
BlazeGraph [49]	✓	–	–	–	–	–
Jena [46]	✓	–	–	–	–	–
Virtuoso [17]	✓	–	–	–	–	–
Neo4j [54]	partial	✓	✓	–	–	✓
Nebula [50]	partial	partial	✓	–	✓	✓
Memgraph [47]	partial	✓	✓	–	–	✓
Kuzu [30]	partial	✓	–	–	–	✓
DuckPGQ [55]	partial	✓	–	–	–	✓
PathFinder	✓	✓	✓	✓	✓	✓

We see that SPARQL engines support all RPQs, but cannot return paths. On the other hand, while several property graph engines can return some (but not all) types of paths, they only offer a limited support for regular expressions. In addition, both SPARQL engines and property graph engines have significant issues handling path queries [42, 53]. We believe that this is the case because of scalability, that is, *the number of paths that match an RPQ can be exponential in the size of the graph* [34].

Our contribution. To handle the aforementioned issues, in this paper we introduce PathFinder, a unified framework for returning paths in graph query answers. PathFinder supports returning paths, using *all 15 path modes* prescribed in the GQL and SQL/PGQ standards, while at the same time permitting *all RPQs*. To achieve this, we show how classical graph search algorithms can be adapted to run on the product graph [39, 34] and at the same time produce a compact representation of all paths that need to be returned. This construction is done lazily and the paths can be returned as soon as they are detected.

We test our approach over Wikidata [52] using real world queries [33]. Our experiments show two interesting things; namely, that over the real-world data:

- returning up to 100.000 paths witnessing RPQ answers can be done in the same time that other engines need to determine the existence of a path; and
- automata-based approaches for RPQ evaluation can scale well and even significantly outperform regular expression based algorithms for RPQ evaluation while also returning paths.

Additional material. We present the fully pipelined version of our algorithms in Appendix A and showcase additional experiments in Appendix B.

2 Graph Databases and 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 [5, 16, 22, 34], we define graph databases as follows.

Definition 1. *A* graph database G *is a tuple* (V, E, ρ, λ)*, where*

- $V \subseteq$ *Nodes is a finite set of nodes,*
- $E \subseteq$ *Edges is a finite set of edges,*
- $\rho : E \to (V \times V)$ *is a total function, and*
- $\lambda : E \to$ Lab *is a total function assigning a label to each edge.*

Intuitively, $\rho(e) = (v_1, v_2)$ means that e is a directed edge going from v_1 to v_2. Here we use a simplified version of property graphs which omits node and edge properties (with their associated values). 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.

Paths. A sequence $p = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$ is called a *path* in a graph database $G = (V, E, \rho, \lambda)$, if $n \geq 0$, $e_i \in E$, 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 say that a path p is a

- WALK, for every p (as per GQL and SQL/PGQ terminology);
- TRAIL, if p does not repeat an edge ($e_i \neq e_j$ for every $i \neq j$);
- ACYCLIC, if p does not repeat a node ($v_i \neq v_j$ for every $i \neq j$); and
- SIMPLE, if p does not repeat a node, except that possibly $\mathsf{src}(p) = \mathsf{tgt}(p)$.

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

Path queries in GQL and SQL/PGQ. Following the GQL and SQL/PGQ [16] standards, we define *path queries* as expressions of the form

$$\text{selector}?\ \text{restrictor}\ (\mathtt{x}, \mathtt{regex}, \mathtt{y})$$

where selector? means that selector is optional and `regex` is a regular expression. Furthermore, `x` and `y` are variables (denoted as $?x, ?y, \ldots$) or nodes in the graph (denoted $v, v', n, n', \ldots$). *Selectors* and *restrictors* are used to specify which paths are to be returned. Their grammar is:

$$\text{selector} : \text{ANY} \mid \text{ANY SHORTEST} \mid \text{ALL SHORTEST}$$
$$\text{restrictor} : \text{WALK} \mid \text{TRAIL} \mid \text{SIMPLE} \mid \text{ACYCLIC}$$

A selector-restrictor combination is also called a *path mode*. We have 16 path modes: 3 selectors times 4 restrictors, plus the 4 restrictors without selector. However, WALK needs to be preceded by a selector [16] to avoid the need to return infinitely many paths in the presence of cycles, which gives 15 modes.

Next, we formally define the semantics of path queries. Let G be a graph database and q be a path query of the form

$$\mathsf{restrictor}\ (?x, \mathtt{regex}, ?y)\ .$$

For now, we omit the optional selector part and assume that `x` and `y` are distinct variables $?x$ and $?y$ respectively. (We denote variables with question marks to distinguish them from nodes in G.) We use $\mathsf{Paths}(G, \mathsf{restrictor})$ 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 $q = \mathsf{restrictor}\ (?x, \mathtt{regex}, ?y)$, as follows:

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

Here $\mathcal{L}(\mathtt{regex})$ denotes the language of `regex`. Intuitively, for a query "TRAIL $(?x$, `regex`, $?y)$", the semantics returns all tuples (v, v', p) such that p is a TRAIL in our graph that connects v to v' and $\mathsf{lab}(p) \in \mathcal{L}(\mathtt{regex})$. The semantics of selectors is defined on a case-by-case basis. Using q to denote the selector-free query $q = \mathsf{restrictor}\ (?x, \mathtt{regex}, ?y)$, we now have:

- $[\![\mathsf{ANY}\ q]\!]_G$ returns, for each pair of nodes (v, v') such that $(v, v', p) \in [\![q]\!]_G$, for some p, a *single tuple* $(v, v', p_{out}) \in [\![q]\!]_G$.
- $[\![\mathsf{ANY\ SHORTEST}\ q]\!]_G$ returns, for each pair (v, v') such that $(v, v', p) \in [\![q]\!]_G$, for some p, a *single tuple* $(v, v', p_{out}) \in [\![q]\!]_G$ such that p_{out} is SHORTEST among all paths p' with $(v, v', p') \in [\![q]\!]_G$.
- $[\![\mathsf{ALL\ SHORTEST}\ q]\!]_G$ returns, for each pair (v, v') such that $(v, v', p) \in [\![q]\!]_G$ for some p, *all tuples* $(v, v', p_{out}) \in [\![q]\!]_G$ such that p_{out} is SHORTEST among all paths p' with $(v, v', p') \in [\![q]\!]_G$.

Notice that the SHORTEST modes only select shortest paths *grouped by endpoints*. That is, the selected paths do not need to be the shortest in the entire output, they only need to be the shortest matching paths from v to v'. Furthermore, the semantics of ANY and ANY SHORTEST is non-deterministic when there are multiple (shortest) paths connecting a pair of nodes. We also remind that 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.

Finally, if `x` is a node n in G, then the semantics is defined exactly as above, but only keep the answers (v, v', p) where $v = n$. (Analogously if `y` is a node in G.) As such, one can also use queries of the form ALL TRAILS $(n, \mathtt{a}^+, ?y)$.

3 Backbone of PathFinder and the WALK Semantics

In this section we describe the main idea behind PathFinder and show how it can be used to evaluate RPQs under the WALK semantics. That is, we show how to treat queries of the form

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

Recall that the selector is obligatory in GQL, since the set of all walks can be infinite. For simplicity of exposition, we assume for now that the starting point of the RPQ is fixed (i.e., we start from a node v) and discuss the other cases for endpoints later. Our approach relies heavily on the product construction of automata [28, 39]. We present its full construction here, but in practice it is important to construct the product *lazily*, i.e., only construct the part of it that is needed to evaluate the query.

Product graph. Given a graph database $G = (V, E, \rho, \lambda)$ and an expression of the form $q = (v, \texttt{regex}, \texttt{?x})$, the *product graph* is constructed by first converting the regular expression `regex` into an equivalent non-deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$. It consists of a set of states Q, a finite alphabet of edge labels Σ, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial state q_0, and the set of final states F. The construction of the automaton (without ε-transitions) can be done in time linear in the size of `regex` [43]. 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:
  - $d = (q_1, a, q_2)$
  - $\lambda(e) = a$
  - $\rho(e) = (x, y)$; and
- $\lambda_\times((e, d)) = \lambda(e)$.

Each node of the form (u, q) in $G_\times$ corresponds to the node u in G and, furthermore, each path P of the form $(v, q_0), (v_1, q_1), \ldots, (v_n, q_n)$ in $G_\times$ corresponds to a path $p = v, v_1, \ldots, v_n$ in G that (a) has the same length as P and (b) brings the automaton from state q_0 to q_n. As such, when $q_n \in F$, then this path in G matches `regex`. In other words, all nodes v' that can be reached from v by a path that matches `regex` can be found by using standard graph search algorithms (e.g., BFS/DFS) on $G_\times$ starting in the node (v, q_0).

While this approach allows finding pairs (v, v') in answers to an RPQ, we show next how it can be used to also find paths. Our approach is rooted in this fairly standard construction in automata theory, but on the other hand, it has nice advantages: (a) the product graph can indeed be explored lazily *for all 15 path modes*, (b) paths can be always returned *on-the-fly*, allowing pipelined execution, and (c) the approach is highly efficient (Section 5). We point out in which cases subtleties such as *unambiguity of the automaton* need to be taken into account to achieve a correct algorithm.

3.1 ANY (SHORTEST) WALK

We first treat the WALK restrictor combined with selectors ANY and ANY SHORTEST, that is, queries of the form:

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

The idea is that, given a graph database G and a query q of the form (1), 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 BFS and DFS 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). Query evaluation is presented in Algorithm 1. The algorithm is a straightforward adaptation of BFS and DFS on a lazily constructed $G_\times$ with modifications that eliminate multiple paths reaching the same node and multiple accepting runs in an automaton. We present the algorithm in detail since we extend the approach to support different semantics later.

The basic object we manipulate is a *search state*, i.e., a quadruple of the form $(n, q, e, prev)$, where n is the node of G we are currently exploring, q is the current state of $\mathcal{A}$, while e is the edge of G we used to reach n, and $prev$ is a pointer to the search state we used to reach (n, q) in $G_\times$. 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 uses four data structures: Open, which is a queue if we want to run BFS or a stack if we want to run DFS, a dictionary Visited, used to track already visited states to avoid infinite loops (searchable by the (n, q) component of search state), Solutions, which is a set containing (pointers to) search states in Visited that encode a solution path, and ReachedFinal, which ensures that no solution is returned twice.

The algorithm explores the product $G_\times$ of G and $\mathcal{A}$ using either BFS (Open is a queue) or DFS (Open is a stack), starting from (v, q_0). It starts by initializing the data structures and setting up the start node in $G_\times$ (lines 2–5). The main loop of line 6 is the classical BFS/DFS algorithm that pops an element $(n, q, e, prev)$ from Open (line 7) and starts exploring its neighbors in $G_\times$ (lines 11–14). When exploring $(n, q, e, prev)$, 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 11). 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 $(q, \lambda(edge'), q')$ is a transition of $\mathcal{A}$. If the pair (n', q') has not been visited yet, we add it to Visited and Open (lines 12–14), which allows it to be expanded later on in the algorithm. When popping from Open in line 7, we also check if q is a final state and that n has not been reached by a solution path yet (line 8). In this case we found a new solution; i.e., a WALK from v to n whose label is in the language of `regex`, so we add it to Solutions (line 10) and record it as reached (line 9). The ReachedFinal set is used to ensure that each solution is returned only once. (Since $\mathcal{A}$ is nondeterministic, it could happen that two different runs of $\mathcal{A}$ accept the same path, or that two different paths reach the same end node via a different end state.)

Algorithm 1 Evaluation of $query$ = ANY (SHORTEST)? WALK $(v, \texttt{regex}, ?x)$

```
1: function AnyWalk(G, query)
2:     A ← Automaton(regex)                          ▷ q0 initial, F final states
3:     Open.init(); Visited.init(); ReachedFinal.init()
4:     startState ← (v, q0, null, ⊥)
5:     Visited.push(startState); Open.push(startState)
6:     while Open ≠ ∅ do
7:         current ← Open.pop()                      ▷ current = (n, q, edge, prev)
8:         if q ∈ F and n ∉ ReachedFinal then
9:             ReachedFinal.add(n)
10:            Solutions.add(current)
11:        for each (n', q', edge') ∈ Neighbors(current, G, A) do
12:            if (n', q', *, *) ∉ Visited then
13:                newState ← (n', q', edge', current)
14:                Visited.push(newState); Open.push(newState)
```

We note that Solutions stores the pointers to states in Visited, which define a solution path. So, for each tuple $(n, q, e, prev)$ in Solutions after running the algorithm, a path from v to n can be reconstructed using the $prev$ part of search states stored in Visited. Furthermore, solutions can be enumerated once the algorithm terminates, or returned as soon as they are detected (line 10). The latter approach allows for a pipelined execution of the algorithm (see [6] for more details). Using BFS guarantees that the returned path is indeed shortest.

Algorithm 1 is fairly simple, but shows in detail both how to manipulate the product graph in an efficient manner and how to handle arbitrary RPQs while finding a single (shortest) path between each pair of nodes in the result set. In addition, it eliminates duplicate results in the sense that, for every node n of G, it only computes a single tuple of the form $(n, q, e, prev)$ in Solutions if q is accepting. In the following sections we show how this approach can be extended for more complex path modes, starting with finding *all* shortest paths.

3.2 ALL SHORTEST WALKS

We next show how to evaluate queries of the form

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

For this, we extend the BFS version of Algorithm 1 in order to find *all shortest paths* between pairs (v, v') of nodes, instead of a single path — see Algorithm 2. The intuition simple: to obtain all shortest paths, upon reaching v' from v by a path conforming to `regex` *for the first time*, BFS always does 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 length equal to the recorded length for v', then this path is also an answer to our query. Interestingly, *the algorithm explores precisely the same portion of $G_\times$ as Algorithm 1.*

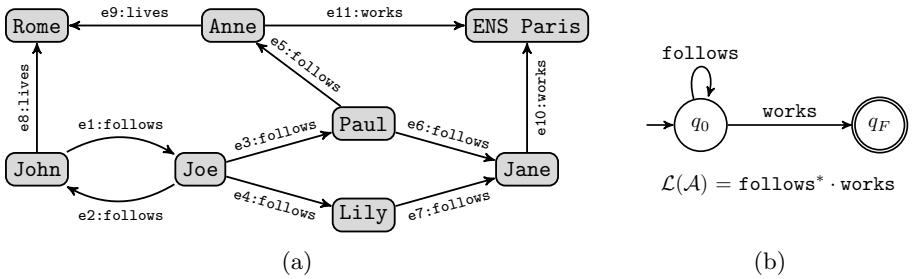
Algorithm 2 Evaluation of $query$ = ALL SHORTEST WALK $(v, \texttt{regex}, ?x)$.

```
1: function AllShortestWalk(G, query)
2:     A ← UnambiguousAutomaton(regex)              ▷ q0 initial, F final states
3:     Open.init(); Visited.init(); ReachedFinal.init()
4:     startState ← (v, q0, 0, ⊥)
5:     Visited.push(startState); Open.push(startState)
6:     while Open ≠ ∅ do
7:         current ← Open.pop()                     ▷ current = (n, q, depth, prevList)
8:         if q ∈ F then
9:             if n ∉ ReachedFinal then
10:                ReachedFinal.add(⟨n, depth⟩)
11:                Solutions.add(current)
12:            else if ReachedFinal.get(n).depth = depth then
13:                Solutions.add(current)
14:        for next ← (n', q', edge') ∈ Neighbors(current, G, A) do
15:            if (n', q', *, *) ∈ Visited then
16:                (n', q', depth', prevList') ← Visited.get(n', q')
17:                if depth + 1 = depth' then        ▷ New shortest path to (n', q')
18:                    prevList'.add(⟨current, edge'⟩)
19:            else
20:                prevList.init()
21:                prevList.add(⟨current, edge'⟩)
22:                newState ← (n', q', depth + 1, prevList)
23:                Visited.push(newState); Open.push(newState)
```

As before, we use $\mathcal{A}$ to denote the NFA for `regex`. We will additionally require that $\mathcal{A}$ is *unambiguous* (it has at most one accepting run for every word), which we need to ensure that we do not return the same path twice. This condition is easy to enforce for real-world RPQs [10, 11]. 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 a shortest path to (n, q) from (v, q_0), while $prevlist$ is a *list* of pointers to any previous search state that allows us to reach n via a shortest path from v. We assume that $prevList$ is a *linked list*, initialized as empty, and accepting sequential insertions of pairs $\langle searchState, edge\rangle$ using add(). Intuitively, $prevList$ allows us to reconstruct *all* shortest paths reaching a node.

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.

Algorithm 2 explores the product graph of G and $\mathcal{A}$ using BFS, so Open is a queue. The main difference to Algorithm 1 is as follows: 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', q') 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 20–23). As in Algorithm 1, we check for solutions after a state has been removed from Open (lines 8–13). 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. *Notice that solutions can actually be returned before (i.e., after line 16 we can test if $q' \in F$).* Returning answers when popping from the queue in lines 8–13 has an added benefit that the paths are grouped for a pair (v, n) of connected nodes. As in Algorithm 1, we need to make sure that reaching the same node n via different accepting states of $\mathcal{A}$ is permitted only when this results in a shortest path. For this ReachedFinal is now a dictionary storing pairs $\langle n, depth \rangle$, where n is a solution node and *depth* the length of the shortest path reaching n from v. We assume n to be the dictionary key. In case that the solution is reached the first time (lines 9–11) we record this information. When reaching the same node with another accepting path (lines 12–13), we record the solution only if the length of the path is the same as the optimal length already recorded. Finally, we can enumerate the solutions by traversing the DAG stored in Visited in a depth-first manner starting from the nodes in Solutions. Next we illustrate how Algorithm 2 works in detail.



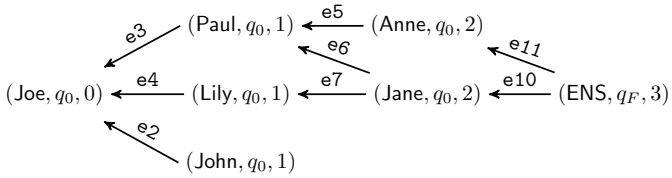
(a) (b)

Fig. 1: A sample graph database (a) and a sample automaton $\mathcal{A}$ (b).

Example 1. Consider the social network graph G in Figure 1 (a) and let

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

As we can see, there are three shortest paths connecting Joe to ENS Paris matching the query. To compute these, Algorithm 2 first converts the expression $\texttt{follows}^* \cdot \texttt{works}$ into the automaton in Figure 1 (b). Algorithm 2 then starts traversing the product graph $G_\times$ from the node (Joe, q_0). Upon executing the algorithm, the structure of Visited is as follows:



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, $(\mathsf{Jane}, q_0, 2)$ has two outgoing edges, representing two pointers in its *prevList*. The arrow is also annotated with the edge witnessing the connection (as stored in *prevList*).

To build this structure, Algorithm 2 explores the neighbors of (Joe, q_0); namely, $(\mathsf{Paul}, q_0), (\mathsf{Lily}, q_0)$ and (John, q_0) and puts them to Visited and Open, with $depth = 1$. The algorithm proceeds by visiting (Anne, q_0) from (Paul, q_0) and (Jane, q_0) 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. 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 a previously unexplored shortest path. We can then enumerate all the paths from (ENS, q_F), by following the edges. □

For Algorithm 2 to work correctly, we crucially need $\mathcal{A}$ to be unambiguous. (this can be achieved by determinizing $\mathcal{A}$). Concerning run-time, Algorithm 2 indeed explores precisely the same portion of $G_\times$ as Algorithm 1, since the nodes of $G_\times$ we revisit (lines 15–18) do not get added to Open again. However, we can potentially add extra edges to Visited, so $[\![q]\!]_G$ can be much bigger (see [34, 18] for examples when the set of shortest paths is exponential). Additionally, we note that when enumerating the output paths, each path is returned in time proportional to its length, which is also known as *output-linear delay* [20], and is optimal in the sense that, to return a path we need to at least write it down. In short, we can conclude that Algorithm 2 runs with $O(|\mathcal{A}| \cdot |G|)$ pre-processing time, after which the results can be enumerated in time proportional to their total length.

4 TRAIL, SIMPLE, and ACYCLIC

We now 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 or a trail is NP-complete [7, 15, 35, 8], so we do not know any efficient algorithms. On the other hand, regular expressions for which this problem is

Algorithm 3 Evaluation for $query =$ (ALL SHORTEST)? restrictor $(v, \texttt{regex}, ?x)$.

```
 1: function AllRestrictedPaths(G, query)
 2:     A ← UnambiguousAutomaton(regex)              ▷ q0 initial, F final states
 3:     Open.init(); Visited.init(); ReachedFinal.init()
 4:     startState ← (v, q0, 0, null, ⊥)
 5:     Visited.push(startState); Open.push(startState)
 6:     while Open ≠ ∅ do
 7:         current ← Open.pop()                     ▷ current = (n, q, depth, e, prev)
 8:         if q ∈ F then
 9:             if !(ALL SHORTEST) then
10:                 Solutions.add(current)
11:             else if n ∉ ReachedFinal then
12:                 ReachedFinal.add(⟨n, depth⟩)
13:                 Solutions.add(current)
14:             else
15:                 optimal ← ReachedFinal.get(n).depth
16:                 if depth = optimal then
17:                     Solutions.add(current)
18:         for next ← (n', q', edge') ∈ Neighbors(current, G, A) do
19:             if isValid(current, next, restrictor) then
20:                 new ← (n', q', depth + 1, edge', current)
21:                 Visited.push(new); Open.push(new)
```

indeed NP-hard are rare in practice [10, 36, 37] and, moreover, regular expressions for which these problems are tractable can be evaluated by cleverly enumerating paths in the product graph [36, 38]. Our algorithms will follow this intuition and prune the search space whenever possible. In the worst case all such algorithms will be exponential but we will show that, on real-world graphs, the number of paths will be manageable, and the pruning technique will ensure that all dead-ends are discarded. We begin by showing how to evaluate queries of the form

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

where restrictor is TRAIL, SIMPLE, or ACYCLIC. Algorithm 3 shows how to evaluate such queries. Intuitively, the algorithm explores the product graph by enumerating all paths starting in (v, q_0) but pruning quickly. Here, our *search state* is a tuple $(n, q, depth, e, prev)$, where n is a node, q an automaton state, $depth$ the length of a shortest path to (n, q) in $G_\times$, e an edge used to reach the node n, and $prev$ a pointer to another search state stored in Visited, which is a set storing already visited search states. Same as in Algorithm 2, we use a dictionary ReachedFinal storing pairs $(n, depth)$, with key n. Here, n is a node reached in some query answer, and $depth$ the length of a shortest path to this node.

The execution is very similar to Algorithm 1, but Visited is not used to discard solutions. Instead, when checking whether the current node can be extended to a neighbor (lines 18–21), we call isValid which checks whether the path

to the current node (i.e. n) in Visited can be extended to the next node (i.e. n') according to restrictor. Notice that we need to check 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 restrictor, we add the new search state to Visited and Open (line 21). When popping from Open, we also check if a potential solution is reached (line 8). If the ALL SHORTEST selector is *not* present, we simply add the newly found solution. 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 already discovered nodes. If the node is seen for the first time, the dictionary is updated, and a new solution added (lines 11–13). Upon discovering the same node again (lines 14–17), a new solution is added only if it is shortest. Once all paths have been explored (Open $= \emptyset$), we can enumerate the solutions as in other algorithms.

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 $O\big((|\mathcal{A}| \cdot |G|)^{|G|}\big)$. This is also the best known runtime since the problem is NP-hard [15].

Adding ANY and ANY SHORTEST. Treating queries of the form

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

can be done with minimal changes to Algorithm 3. Namely, we would use ReachedFinal as a *set* that stores the node first time a solution is found in order not to repeat any results. In addition, we would replace lines 8–17 with:

```
if q ∈ F and n ∉ ReachedFinal then
    ReachedFinal.add(n)
    Solutions.add(current)
```

Same as Algorithm 3, worst-case runtime is $O\big((|\mathcal{A}| \cdot |G|)^{|G|}\big)$, which is also the best known runtime due to NP-completeness of the problem [15].

5 Experimental Evaluation

We empirically evaluate PathFinder and show that the approach scales on a broad range of real-world queries. We start by explaining how PathFinder was implemented, followed by the description of our experimental setup, and the discussion of the obtained results. For code, data, and queries we refer to [6].

Implementation. PathFinder is implemented as the path processing engine of AnonymousDB, a recent graph database system developed by the authors. AnonymousDB provides the infrastructure for processing GQL queries, generating execution plans and storing the data, while PathFinder executes

path queries as described in the paper. The default data storage of AnonymousDB is B+trees, allowing to load the disk data into a main memory buffer as required by our algorithms. The graph is stored as a table of the form Edges(NodeFrom, Label, NodeTo, EdgeId). A permutation that reverses the order of the source and target node of the edge is also stored in order to support efficient evaluation of 2RPQs [13] and SPARQL property paths [26], which permit traversing edges in the reverse direction. By default we assume all the data to be on disk and the main memory buffer to be empty. *All the algorithms are implemented in a fully pipelined fashion, meaning that they can be paused as soon as a result is detected.* While in the paper we assumed the source of the RPQ part of the query to be fixed, the implementation supports ends being fixed or free. The former is evaluated by running our algorithms and checking whether the target node is a solution, while the latter can be achieved by running the algorithm from any start node.

Data and queries. To test the efficiency and scalability of PathFinder we use Wikidata [52] and queries from its public SPARQL query log [33]. We use WDBench [2], a recently proposed Wikidata SPARQL benchmark. WDBench provides a curated version of the data set based on the truthy dump of Wikidata [21], which is an edge-labeled graph with 364 million nodes and 1.257 billion edges, using more than 8,000 different edge labels. The data set is publicly available [3]. WDBench provides multiple sets of queries extracted from Wikidata's public endpoint query log. We use the Paths query set, which contains 659 2RPQ patterns. From these, 592 have a fixed starting point or ending point (or both), while 67 have both endpoints free. We note that some queries require regular expressions which are not supported in all tested systems. The 659 patterns are then used in our tests under different GQL path modes (details below).

Tested systems. We use both BFS and DFS versions of PathFinder, denoted as PF-BFS and PF-DFS respectively. To compare with state of the art in the area we tested the following engines:

- Neo4J version 4.4.12 [54] (Neo4j for short);
- Jena TDB version 4.1.0 [46] (Jena);
- Blazegraph version 2.1.6 [49] (BlazeGraph); and
- Virtuoso version 7.2.6 [17] (Virtuoso).

We also tried loading the data into NebulaGraph [50], Kuzu [30] and Memgraph [47], but neither system could handle data of this size on used hardware.

Tested path modes. SPARQL engines do not support returning paths, but only detecting whether one exists, and Neo4j only supports walk and trail semantics. We therefore tested each of the 659 property path patterns with the WALK and TRAIL restrictors, and all possible selectors.

How we ran the experiments. We used a machine with an Intel®Xeon® Silver 4110 CPU, and 128GB of DDR4/2666MHz RAM, running Linux Debian 10 with kernel v5.10. The hard disk for storing the data was a 14TB HDD SEAGATE model ST14000NM001G. Custom indexes for speeding up the queries

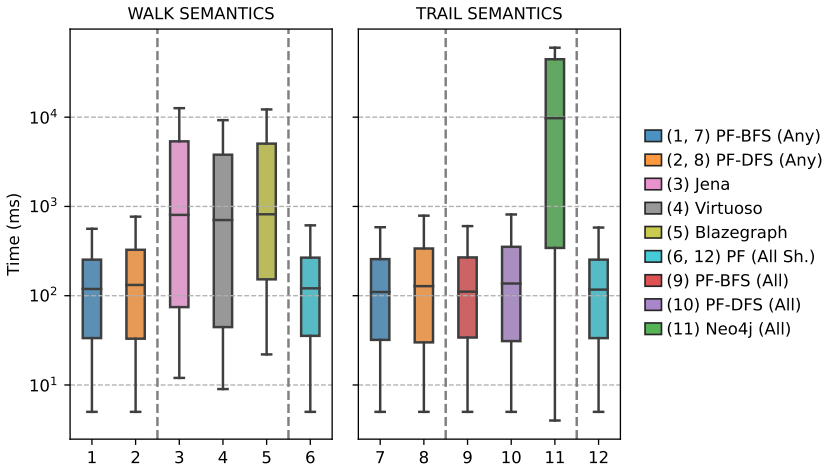


Fig. 2: Runtimes for the Wikidata experiment.

were created for Neo4j and the system was run with the default settings and no limit on RAM usage. Jena, BlazeGraph, Virtuoso and PathFinder were assigned 64GB of RAM for buffering. Since we run large batches of queries, these are executed in succession. *All queries were run with a limit of 100,000 results and a timeout of 1 minute* (as suggested by WDBench [2]). Results are summarized in Figure 2. Next, we analyze the results.

WALKS. In Figure 2 (left) we show the results for the WALK restrictor. We present box plots for the 659 queries. The first two columns represent the BFS and DFS version of Algorithm 1 in PathFinder, which corresponds to ANY (SHORTEST) WALK mode. Here, PathFinder clearly outperforms all other systems, *even if it returns paths whereas the SPARQL engines do not.* Indeed, Jena timed out 95 times, and BlazeGraph and Virtuoso 86 and 24 times, respectively, whereas PathFinder-BFS only had 8 and PathFinder-DFS 9 timeouts. Neo4j even timed out in 657 out of 659 queries in ANY SHORTEST WALK, so we do not include it in Figure 2 (left). The final column shows the performance of PathFinder for the ALL SHORTEST PATH mode of Algorithm 2. Interestingly, *despite requiring a more involved algorithm, finding 100,000 paths under this path mode shows almost identical performance to finding a single shortest path for each reached node.* The number of timeouts here was only 7. Again, while Neo4j does support this path mode, it could only complete 2 out of 659 queries. Overall, we can conclude that PathFinder presents a stable strategy for finding WALKS and significantly outperforms the state-of-the-art.

TRAILS. Results for the TRAIL semantics are shown in Figure 2 (right). The first two columns correspond to ANY SHORTEST TRAIL and ANY TRAIL in PathFinder. This performance is almost identical to the ANY WALK case, however with 25 and 24 timeouts for the BFS and DFS version, respectively. The following three columns correspond to the ALL TRAIL mode, supported by PathFinder-BFS, PathFinder-DFS, and Neo4j. We see an order of

magnitude improvement in PathFinder compared to Neo4j and, additionally, only 9 and 10 timeouts for PathFinder-BFS and PathFinder-DFS, versus 134 for Neo4j. The final column corresponds to ALL SHORTEST TRAIL in PathFinder, which again shows similar performance to other TRAIL-based modes, with 21 timeouts. Overall, all PathFinder variations show remarkably stable performance. Interestingly, while the TRAIL mode is evaluated using a brute-force approach, and is intractable theoretically [7], over real-world data it works remarkably well. This is most likely due to the fact that PathFinder can either detect 100,000 results rather fast, or because the data itself permits no further graph exploration. We also ran the experiments for SIMPLE and ACYCLIC restrictor in PathFinder, with identical results as in the TRAIL case, showing that Algorithm 3 is indeed a good option for real-world use cases.

Take-Home Message. Since all RPQ evaluation algorithms in the existing systems are based on traversing the parse tree of the involved regular expression [9, 40, 29, 51], our results show that automata-based approaches are competitive and can even significantly outperform the regular expression based algorithms. Furthermore, the WALK experiments teach us that, in several interesting cases, it is possible to return up to 100.000 paths that witness RPQ answers even faster than it takes the SPARQL engines to only detect the existence of an answer.

6 Related Work

Most work in the area focuses on finding nodes connected by an RPQ-conforming path and not on returning the paths [56, 24, 19, 8, 35, 36, 14]. Notable extensions include [25] where paths are returned, but not according to an RPQ pattern, and [45], which uses a BFS-style exploration to find the first k paths, meaning some non-shortest paths will be returned. Similar approach for top-k results is presented in [1], but not preferring shortest paths. Closest to our work is [34], where a compact representation of RPQ-conforming paths (called a PMR) is presented for *some* GQL path modes. Compared to [34], we support *all* GQL path modes and present implementable algorithms, whereas [34] mostly focuses on theoretical guarantees. Interestingly, one can show that the Visited structure of our algorithms in fact encodes the PMR of [34] for any GQL path mode.

7 Conclusions

We present PathFinder, a unifying framework for returning paths in RPQ query answers. We believe PathFinder to be the first system that allows returning paths under *every* mode prescribed by the GQL and SQL/PGQ query standards [16], and that our experimental evaluation shows the approach to be highly competitive on realistic workloads. While our work was developed in the context of property graphs, it is straightforward to implement it on top of an existing SPARQL engine (see [6] for an example of this). We showed that returning paths that match RPQs can be practically viable, which opens the question to which extent we want to explore similar functionality for SPARQL [26].

References

1. Aebeloe, C., Montoya, G., Setty, V., Hose, K.: Discovering diversified paths in knowledge bases. Proc. VLDB Endow. **11**(12), 2002–2005 (2018). https://doi.org/10.14778/3229863.3236245, http://www.vldb.org/pvldb/vol11/p2002-aebeloe.pdf
2. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: Wdbench: A wikidata graph query benchmark. In: The Semantic Web - ISWC 2022 (2022). https://doi.org/10.1007/978-3-031-19433-7“41, https://doi.org/10.1007/978-3-031-19433-7_41
3. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench Dataset Download (2022). https://doi.org/10.6084/m9.figshare.19599589
4. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutiérrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: SIGMOD Conference 2018 (2018)
5. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoč, D.: Foundations of Modern Query Languages for Graph Databases. ACM Comput. Surv. **50**(5) (2017). https://doi.org/10.1145/3104031, https://doi.org/10.1145/3104031
6. Authors, A.: PathFinder: A unified approach for handling paths in graph query languages (2023), https://github.com/AnonCSR/PathFinder
7. Baeza, P.B.: Querying graph databases. In: PODS 2013. pp. 175–188 (2013). https://doi.org/10.1145/2463664.2465216, https://doi.org/10.1145/2463664.2465216
8. Bagan, G., Bonifati, A., Groz, B.: A trichotomy for regular simple path queries on graphs. In: Symposium on Principles of Database Systems (PODS). pp. 261–272 (2013)
9. Blazegrahp source code (2024), https://github.com/blazegraph
10. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of wikidata query logs. In: Liu, L., White, R.W., Mantrach, A., Silvestri, F., McAuley, J.J., Baeza-Yates, R., Zia, L. (eds.) The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019. pp. 127–138. ACM (2019). https://doi.org/10.1145/3308558.3313472, https://doi.org/10.1145/3308558.3313472
11. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB J. **29**(2-3), 655–679 (2020). https://doi.org/10.1007/s00778-019-00558-9, https://doi.org/10.1007/s00778-019-00558-9
12. Buluç, A., Fineman, J.T., Frigo, M., Gilbert, J.R., Leiserson, C.E.: 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. pp. 233–244 (2009)
13. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. J. Comput. Syst. Sci. **64**(3), 443–465 (2002)
14. Casel, K., Schmid, M.L.: Fine-grained complexity of regular path queries. In: International Conference on Database Theory (ICDT). pp. 19:1–19:20 (2021)
15. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: SIGMOD 1987. pp. 323–330 (1987)
16. 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., van Rest, O., Voigt, H., Vrgoč, D., Wu, M.,

Zemke, F.: Graph pattern matching in GQL and SQL/PGQ. In: SIGMOD '22 (2022). https://doi.org/10.1145/3514221.3526057, https://doi.org/10.1145/3514221.3526057
17. Erling, O.: Virtuoso, a Hybrid RDBMS/Graph Column Store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012), http://sites.computer.org/debull/A12mar/vicol.pdf
18. Farías, B., Rojas, C., Vrgoč, D.: Millenniumdb path query challenge. In: AMW 2023 (2023)
19. Fionda, V., Pirrò, G., Gutiérrez, C.: Nautilod: A formal language for the web of data graph. ACM Trans. Web **9**(1), 5:1–5:43 (2015)
20. Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., Vrgoc, D.: Efficient enumeration algorithms for regular document spanners. ACM Trans. Database Syst. **45**(1), 3:1–3:42 (2020)
21. Foundation, T.W.: Wikidata:database download (2021), https://www.wikidata.org/wiki/Wikidata:Database_download
22. Francis, N., Gheerbrant, A., Guagliardo, P., Libkin, L., Marsault, V., Martens, W., Murlak, F., Peterfreund, L., Rogova, A., Vrgoč, D.: A researcher's digest of GQL (invited talk). In: ICDT 2023 (2023). https://doi.org/10.4230/LIPIcs.ICDT.2023.1, https://doi.org/10.4230/LIPIcs.ICDT.2023.1
23. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An Evolving Query Language for Property Graphs. In: SIGMOD 2018 (2018). https://doi.org/10.1145/3183713.3190657, https://doi.org/10.1145/3183713.3190657
24. Gubichev, A.: Query Processing and Optimization in Graph Databases. Ph.D. thesis, Technical University Munich (2015), https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20150625-1238730-1-7
25. Gubichev, A., Neumann, T.: Path query processing on very large RDF graphs. In: WebDB 2011 (2011), http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf
26. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation (2013), https://www.w3.org/TR/sparql11-query/
27. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71:1–71:37 (2022). https://doi.org/10.1145/3447772, https://doi.org/10.1145/3447772
28. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
29. Apache jena source code (2024), https://github.com/apache/jena
30. Jin, G., Feng, X., Chen, Z., Liu, C., Salihoglu, S.: Kùzu graph database management system. In: 13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023. www.cidrdb.org (2023), https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf
31. Jumper, J.e.: Highly accurate protein structure prediction with AlphaFold. Nature **596**(7873), 583–589 (Aug 2021)
32. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
33. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In: ISWC 2018 (2018)

34. Martens, W., Niewerth, M., Popp, T., Rojas, C., Vansummeren, S., Vrgoč, D.: Representing paths in graph database pattern matching. Proc. VLDB Endow. **16**(7), 1790–1803 (2023), https://www.vldb.org/pvldb/vol16/p1790-martens.pdf
35. Martens, W., Niewerth, M., Trautner, T.: A trichotomy for regular trail queries. In: International Symposium on Theoretical Aspects of Computer Science (STACS). pp. 7:1–7:16 (2020)
36. Martens, W., Trautner, T.: Evaluation and enumeration problems for regular path queries. In: International Conference on Database Theory (ICDT). pp. 19:1–19:21 (2018)
37. Martens, W., Trautner, T.: Bridging theory and practice with query log analysis. SIGMOD Rec. **48**(1), 6–13 (2019). https://doi.org/10.1145/3371316.3371319, https://doi.org/10.1145/3371316.3371319
38. Martens, W., Trautner, T.: Dichotomies for evaluating simple regular path queries. ACM Trans. Database Syst. **44**(4), 16:1–16:46 (2019). https://doi.org/10.1145/3331446, https://doi.org/10.1145/3331446
39. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. In: VLDB 1989. pp. 185–193 (1989)
40. Neo4j open source code (2024), https://github.com/neo4j/neo4j
41. Oracle: Oracle graph database. https://www.oracle.com/database/graph/
42. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. Semantic Web **12**(5), 711–740 (2021)
43. Sakarovitch, J.: Elements of automata theory. Cambridge University Press (2009)
44. Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W.G., Arenas, M., Besta, M., Boncz, P.A., Daudjee, K., Valle, E.D., Dumbrava, S., Hartig, O., Haslhofer, B., Hegeman, T., Hidders, J., Hose, K., Iamnitchi, A., Kalavri, V., Kapp, H., Martens, W., Özsu, M.T., Peukert, E., Plantikow, S., Ragab, M., Ripeanu, M., Salihoglu, S., Schulz, C., Selmer, P., Sequeda, J.F., Shinavier, J., Szárnyas, G., Tommasini, R., Tumeo, A., Uta, A., Varbanescu, A.L., Wu, H., Yakovets, N., Yan, D., Yoneki, E.: The future is big graphs: a community view on graph processing systems. Commun. ACM **64**(9), 62–71 (2021)
45. Savenkov, V., Mehmood, Q., Umbrich, J., Polleres, A.: Counting to k or how SPARQL1.1 property paths can be extended to top-k path queries. In: SEMANTiCS 2017 (2017). https://doi.org/10.1145/3132218.3132239, https://doi.org/10.1145/3132218.3132239
46. Team, J.: Jena TDB (2021), https://jena.apache.org/documentation/tdb/
47. Team, M.: Memgraph (2023), https://memgraph.com/
48. Team, T.: TigerGraph Documentation – version 3.1 (2021), https://docs.tigergraph.com/
49. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF Graph Database. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014), http://www.crcnetbase.com/doi/abs/10.1201/b16859-12
50. Vesoft Inc/Nebula: NebulaGraph (2023), https://www.nebula-graph.io/
51. Virtuoso open source code (2024), https://github.com/openlink/virtuoso-opensource
52. Vrandecic, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489, https://doi.org/10.1145/2629489
53. Vrgoč, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Aranda, C.B., Hogan, A., Navarro, G., Riveros, C., Romero, J.: Millenniumdb: A persistent, open-source,

graph database. CoRR **abs/2111.01540** (2021), https://arxiv.org/abs/2111.01540

54. Webber, J.: A programmatic introduction to Neo4j. In: Leavens, G.T. (ed.) SPLASH '12 (2012). https://doi.org/10.1145/2384716.2384777, https://doi.org/10.1145/2384716.2384777
55. ten Wolde, D., Singh, T., Szárnyas, G., Boncz, P.A.: Duckpgq: Efficient property graph queries in an analytical RDBMS. In: 13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023. www.cidrdb.org (2023), https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf
56. Yakovets, N., Godfrey, P., Gryz, J.: Query planning for evaluating SPARQL property paths. In: Özcan, F., Koutrika, G., Madden, S. (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1875–1889. ACM (2016). https://doi.org/10.1145/2882903.2882944, https://doi.org/10.1145/2882903.2882944

A Pipelined Execution

Here we present a pipelined version for each of the algorithms explained in sections 3 and 4. All these operators are implemented as standard linear iterators, providing the following three methods:

1. BEGIN$(G, query)$, which initializes our query, and positions itself just before the first output tuple, without returning anything.
2. NEXT$(G, query)$, which is called each time we wish to access the following tuple in the query answer. Notice that this implies that each algorithm is "paused" upon finding a solution, and then resumed when NEXT$(G, query)$ is called again. This is done in order to combine several operators in a pipelined fashion.
3. SEARCH(n, e, G), which allows to search inside the database G, and locate all neighbors of node n that are connected via an edge of type e. This method returns an iterator object, capable of sequentially retrieving the necessary data (neighbor nodes and the edges that connect them to n) from a list of matching tuples that is stored on the database index of choice (in our case, B+trees and CSRs).

The structure of the automaton is modeled as an array, where each element in the array represents an automaton state, and contains another array with all the outgoing transitions from said state. Each of these transitions store the destination state in the automaton and the edge label for the connection.

For the sake of simplicity, we omit implementations based on DFS, since they are very similar to the ones that use BFS. The only differences between the two versions, are the use of a stack (DFS) instead of a queue (BFS) for Open, and the need for DFS to store the iterator returned by SEARCH(n, e, G) and the currently explored transition of the automaton inside each search state, given that, unlike the BFS strategy, each search state will not necessarily be fully expanded before deciding to explore a different one.

A.1 ANY SHORTEST WALKS

As shown in Algorithm 4, the idea is to search for query solutions using a BFS traversal strategy, until one is found (line 20) or there are no more possible results (line 25). Most of the process occurs inside the expandAny auxiliary function, which is constructed in a way that allows the operator to "pause" the execution when returning a solution, and "resume" the search when Next is called again. To remember the state of the search each time this happens, we store the current transition and iterator inside variables. Since the complete expansion of a single search state can now take multiple calls to Next, whenever we access the top state from Open, we use the front() method to keep the state inside the queue (line 18), and only apply pop() after said state has been fully expanded (line 24).

Algorithm 4 Evaluation for a graph database G and an RPQ $query$ = ANY SHORTEST WALK $(v, \texttt{regex}, ?x)$, using database iterators.

```
1: function BEGIN(G, query)
2:     A ← Automaton(regex)                          ▷ q_0 initial state, F final states
3:     Open.init()
4:     Visited.init()
5:     ReachedFinal.init()
6:     startState ← (v, q_0, null, ⊥)
7:     Visited.push(startState)
8:     Open.push(startState)
9:     iter ← null                                   ▷ Index iterator for neighbors
10:    firstNext ← True                              ▷ First time calling NEXT

11: function NEXT(G, query)
12:    if firstNext then                             ▷ Check if initial state is solution
13:        firstNext ← False
14:        if v ∈ V and q_0 ∈ F then
15:            ReachedFinal.add(v)
16:            return v
17:    while Open ≠ ∅ do
18:        current ← Open.front()                    ▷ current = (n, q, edge, prev)
19:        reached ← EXPANDANY(current)
20:        if reached ≠ null then                    ▷ New solution found
21:            return GETPATH(reached, [])
22:        else                                      ▷ State was fully expanded
23:            iter ← null
24:            Open.pop()
25:    return null                                   ▷ No more solutions

26: function EXPANDANY(state = (n, q, edge, prev))
27:    if iter == null then                          ▷ First time state is explored
28:        if |A.transitions(q)| == 0 then
29:            return null
30:        else                                      ▷ Set the index iterator
31:            transitionIdx ← 0
32:            edgeType ← A.transitions(q)[transitionIdx].type
33:            iter ← Search(n,edgeType,G)
34:    while transitionIdx < |A.transitions(q)| do
35:        transition ← A.transitions(q)[transitionIdx]
36:        q' ← transition.to                        ▷ Next automaton state
37:        while iter.next() ≠ null do               ▷ iter = (n', edge')
38:            if (n', q', *, *) ∉ Visited then
39:                newState ← (n', q', edge', state)
40:                Visited.push(newState)
41:                Open.push(newState)
42:                if q' ∈ F and n' ∉ ReachedFinal then
43:                    ReachedFinal.add(n')
44:                    return newState
45:        transitionIdx++                           ▷ Next transition
46:        if transitionIdx < |A.transitions(q)| then
47:            edgeType ← A.transitions(q)[transitionIdx].type
48:            iter ← Search(n,edgeType,G)
49:    return null
```

A.2 ALL SHORTEST WALKS

Algorithm 5 follows the same structure as Algorithm 4, but extends it to handle the ALL SHORTEST WALK semantics, as seen in section 3. One important detail is the fact that we now use an additional queue-like structure, ReachedSolutions, to store a batch of paths found via the GETNEWPATHS function (line 24). This allows us to return a group of results one by one, each time NEXT is called (lines 13–14). The EXPANDALLSHORTEST auxiliary function is displayed separately in Algorithm 6, and follows the same logic as EXPANDANY, but adapted to the ALL SHORTEST case.

Algorithm 5 Evaluation algorithm for a graph database G and an RPQ $query =$ ALL SHORTEST WALK $(v, \texttt{regex}, ?x)$, using database iterators.

```
1:  function BEGIN(G, query)
2:      A ← Automaton(regex)                          ▷ q0 initial state, F final states
3:      Open.init()
4:      Visited.init()
5:      ReachedFinal.init()
6:      startState ← (v, q0, 0, ⊥)
7:      Visited.push(startState)
8:      Open.push(startState)
9:      iter ← null                                   ▷ Index iterator for neighbors
10:     ReachedSolutions.init()                       ▷ Queue of current solutions
11:     firstNext ← True                              ▷ First time calling NEXT

12: function NEXT(G, query)
13:     while ReachedSolutions ≠ ∅ do                 ▷ Enumerate solutions
14:         return ReachedSolutions.pop()
15:     if firstNext then                             ▷ Check if initial state is solution
16:         firstNext ← False
17:         if v ∈ V and q0 ∈ F then
18:             ReachedFinal.add(⟨v, 0⟩)
19:             return v
20:     while Open ≠ ∅ do                             ▷ current = (n, q, depth, prevList)
21:         current ← Open.front()
22:         reached ← EXPANDALLSHORTEST(current)
23:         if reached ≠ null then                    ▷ New solutions found
24:             ReachedSolutions ← GETNEWPATHS(reached)
25:             return ReachedSolutions.pop()
26:         else                                      ▷ State was fully expanded
27:             iter ← null
28:             Open.pop()
29:     return null                                   ▷ No more solutions
```

Algorithm 6 Auxiliary functions for ALL SHORTEST WALK evaluation, using database iterators.

```
1: function expandAllShortest(state = (n, q, depth, prevList))
2:     if iter == null then                                  ▷ First time state is explored
3:         if |A.transitions(q)| == 0 then
4:             return null
5:         else                                              ▷ Set the index iterator
6:             transitionIdx ← 0
7:             edgeType ← A.transitions(q)[transitionIdx].type
8:             iter ← Search(n,edgeType,G)
9:     while transitionIdx < |A.transitions(q)| do
10:        transition ← A.transitions(q)[transitionIdx]
11:        q' ← transition.to                                ▷ Next automaton state
12:        while iter.next() ≠ null do                       ▷ iter = (n', edge')
13:            if (n', q', *, *) ∈ Visited then
14:                (n', q', depth', prevList') ← Visited.get(n', q')
15:                if depth + 1 == depth' then
16:                    prevList'.add(⟨ state, edge'⟩)
17:                    if q' ∈ F then
18:                        shortest ← ReachedFinal.get(n').depth
19:                        if depth + 1 == shortest then
20:                            return (n', q', depth', prevList')
21:            else
22:                prevList.init()
23:                prevList.add(⟨ state, edge'⟩)
24:                newState ← (n', q', depth + 1, prevList)
25:                Visited.push(newState)
26:                Open.push(newState)
27:                if q' ∈ F then
28:                    if n' ∉ ReachedFinal then
29:                        ReachedFinal.add(⟨n', depth + 1⟩)
30:                        return newState
31:                    else
32:                        shortest ← ReachedFinal.get(n').depth
33:                        if depth + 1 == shortest then
34:                            return newState
35:        transitionIdx++                                   ▷ Next transition
36:        if transitionIdx < |A.transitions(q)| then
37:            edgeType ← A.transitions(q)[transitionIdx].type
38:            iter ← Search(n,edgeType,G)
39:    return null

40: function getNewPaths(state = (n, q, depth, prevList))
41:    if prevList == ⊥ then                                 ▷ Initial state
42:        return [v]
43:    newPrev ← prevList.back()                             ▷ Reconstruct last prev
44:    for prevPath ∈ getAllPaths(newPrev.state, []) do
45:        paths.add(prevPath.extend(n, newPrev.edge))
46:    return paths
```

A.3 ALL RESTRICTED PATHS

In the case of restrictor based semantics, Algorithm 7 computes ALL paths that satisfy a specific restrictor. Its auxiliary function is presented in Algorithm 8. For the sake of brevity, we omit the optional ALL SHORTEST selector for this algorithm, since it follows the same ideas as shown in Algorithm 5. The process is quite similar to that of Algorithm 4, but without discarding any visited states and instead checking if each explored path satisfies the restrictor of interest (line 13 in Algorithm 8).

Algorithm 7 Evaluation algorithm for a graph database G and an RPQ $query =$ ALL restrictor $(v, \texttt{regex}, ?x)$, using database iterators.

```
1:  function BEGIN(G, query)
2:      A ← Automaton(regex)                    ▷ q0 initial state, F final states
3:      Open.init()
4:      Visited.init()
5:      startState ← (v, q0, null, ⊥)
6:      Visited.push(startState)
7:      Open.push(startState)
8:      iter ← null                             ▷ Index iterator for neighbors
9:      firstNext ← True                        ▷ First time calling NEXT

10: function NEXT(G, query)
11:     if firstNext then                       ▷ Check if initial state is solution
12:         firstNext ← False
13:         if v ∈ V and q0 ∈ F then
14:             return v
15:     while Open ≠ ∅ do
16:         current ← Open.front()              ▷ current = (n, q, edge, prev)
17:         reached ← EXPANDALL(current)
18:         if reached ≠ null then              ▷ New solution found
19:             return GETPATH(reached, [])
20:         else                                ▷ State was fully expanded
21:             iter ← null
22:             Open.pop()
23:     return null                             ▷ No more solutions
```

Algorithm 8 The ExpandAll function of Algorithm 7.

```
1: function expandAll(state = (n, q, edge, prev))
2:     if iter == null then                          ▷ First time state is explored
3:         if |A.transitions(q)| == 0 then
4:             return null
5:         else                                      ▷ Set the index iterator
6:             transitionIdx ← 0
7:             edgeType ← A.transitions(q)[transitionIdx].type
8:             iter ← Search(n,edgeType,G)
9:     while transitionIdx < |A.transitions(q)| do
10:        transition ← A.transitions(q)[transitionIdx]
11:        q′ ← transition.to                        ▷ Next automaton state
12:        while iter.next() ≠ null do               ▷ iter = (n′, edge′)
13:            if isValid(state, iter, restrictor) then
14:                new ← (n′, q′, edge′, state)
15:                Visited.push(new)
16:                Open.push(new)
17:                if q′ ∈ F then
18:                    return new
19:        transitionIdx++                           ▷ Next transition
20:        if transitionIdx < |A.transitions(q)| then
21:            edgeType ← A.transitions(q)[transitionIdx].type
22:            iter ← Search(n,edgeType,G)
23:    return null
```

B Additional Experiments

To give some additional insight into the performance of our algorithms, and to be able to test with several contemporary property graph engines which do support returning paths, but could not handle the volume of data in the Wikidata experiment, we perform the following two sets of experiments:

- Pokec, which tests the effect of path length on performance over medium sized graphs; and
- Diamond, where we test the effect of having a large number of paths in the graph, even when the graph itself is rather small.

Next we describe each set of experiments in more detail.

(1) Pokec. This experiment uses the Pokec social network graph from SNAP [32]. Pokec is a Slovakian social network which records (directed) user connections, similar to the graph of Figure 1, but with a single type of edge label (we call it follows). The graph contains around 1.6 million nodes and 30 million edges. In our experiment we select the node that is median in terms of centrality,[1] and explore its influence network while also returning up to 100,000 paths that witness the connections. This means that from the selected node v we execute the RPQ and traverse follows-labeled edges from this node. We explore paths of length 1 through k, where k ranges from 1 to 12. Longer paths are uninteresting, since the graph's diameter is 11. We pair these queries with the path modes described in Section 2. Additionally, we also test the endpoint semantics, which simply looks to return all reachable nodes, as supported by SPARQL systems. The idea behind this experiment is to test what happens with query performance as we seek longer paths in a real-world graph of intermediate size.

(2) Diamond. In this experiment we test what happens when there is a large number of paths present in our graph. The database we use, taken from [34], is presented in Figure 3. The queries we consider look for paths between start and end using a-labeled edges. Notice that all such paths are, at the same time, shortest, trails and simple paths, and have length $2n$. Furthermore, there are 2^n such paths, while the graph only has $3n + 1$ nodes and $4n$ edges. We test our query with the path modes from Section 2, while scaling n (and thus path length) from 1 to 40. While returning all these paths is unfeasible for any algorithm, we test whether a portion of them (100,000 in our experiments) can be retrieved efficiently.

[1] Meaning that we computed the number of edges in which each node participates and selected one whose count is the median for the dataset.

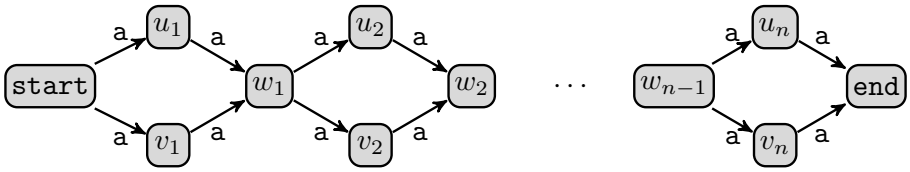


Fig. 3: Graph database with exponentially many paths.

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

- Neo4J version 4.4.12 [54] (Neo4j for short);
- NebulaGraph version 3.5.0 [50] (Nebula);
- Kuzu version 0.0.6 [30] (Kuzu);
- Jena TDB version 4.1.0 [46] (Jena);
- Blazegraph version 2.1.6 [49] (BlazeGraph); and
- Virtuoso version 7.2.6 [17] (Virtuoso).

From the aforementioned systems, Neo4j and Nebula use the ALL TRAIL semantics by default. Neo4j and Kuzu support ANY SHORTEST WALK and ALL SHORTEST WALK modes. Kuzu also supports ALL WALK, but to assure finite answers, all paths are limited to length at most 30. In terms of SPARQL systems (Jena, BlazeGraph, Virtuoso), these do not return paths, but do support arbitrary RPQs and according to the SPARQL standard [26], detect pairs of nodes connected by an arbitrary walk. Other systems we considered are DuckDB [55], Oracle Graph Database [41] and Tiger Graph [48], which support (parts of) SQL/PGQ. Unfortunately, [41] and [48] are commercial systems with limited free versions, while the SQL/PGQ module for DuckDB [55] is still in development.

How we ran the experiments. The 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 capacity. Note that this is a classical HDD, and not an SSD. Custom indexes for speeding up the queries were created for Neo4j, Nebula and Kuzu, and the four systems were run with the default settings and no limit on RAM usage. Jena, BlazeGraph, Virtuoso and PathFinder were assigned 64GB of RAM for buffering. Since we run large batches of queries, these are executed in succession, in order to simulate a realistic load to a database system. *All queries were run with a limit of 100,000 results and a timeout of 1 minute.*

B.1 Pokec: scaling the path length

Here we take a highly connected graph of medium size and test what happens if we ask for paths of increasing length. All tested systems easily loaded this data set. Given that SPARQL systems cannot return paths, we compare with them when the system is only asked to retrieve the reachable nodes (the ENDPOINTS experiment). Given that other systems only support TRAILS and WALKS, we retrieve paths according to these two modes. Our results are presented in Figure 4 and Figure 5.

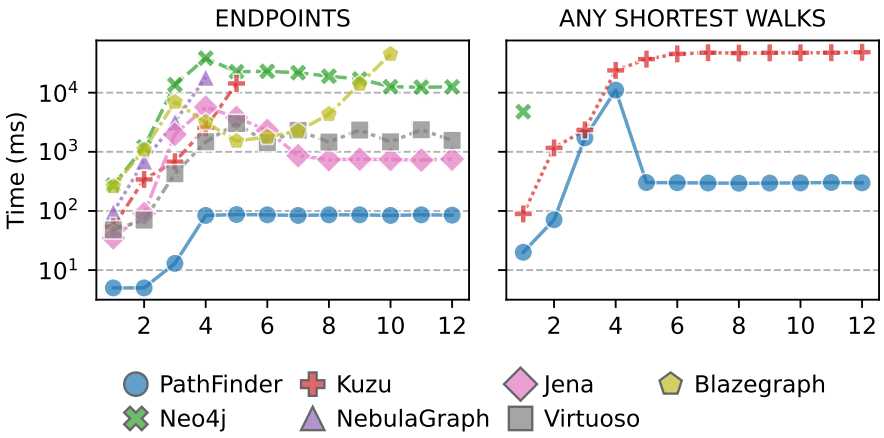


Fig. 4: Runtimes over the Pokec dataset.

ENDPOINTS. Results for retrieving reachable nodes is presented in Figure 4 (left). As we can see, SPARQL systems handle this use case relatively well (only BlazeGraph timed out for the largest path length), while Neo4j also shows decent performance with no timeouts. In contrast, Nebula and Kuzu start timing out for paths of length 5 and 6, respectively. PathFinder shows superior performance with a stable runtime. From length 4 onwards, PathFinder stabilizes due to pipelined execution.

WALKS. Results for ANY SHORTEST WALK are given in Figure 4 (right). As we can see, Kuzu has a highly performant algorithm that handles this use case well, while Neo4j times out rather quickly. PathFinder is the clear winner, with rather stable performance for longer lengths. We note that the spike in PathFinder for lengths 3 and 4 is due to loading the data from disk, which then stays in the buffer for longer length paths (recall that we run the queries one after another). The case of ALL SHORTEST WALK is presented in Figure 5 (right). The picture here is similar, with no system being able to handle paths of length 6 or more, while PathFinder scales very well. The data loading spike

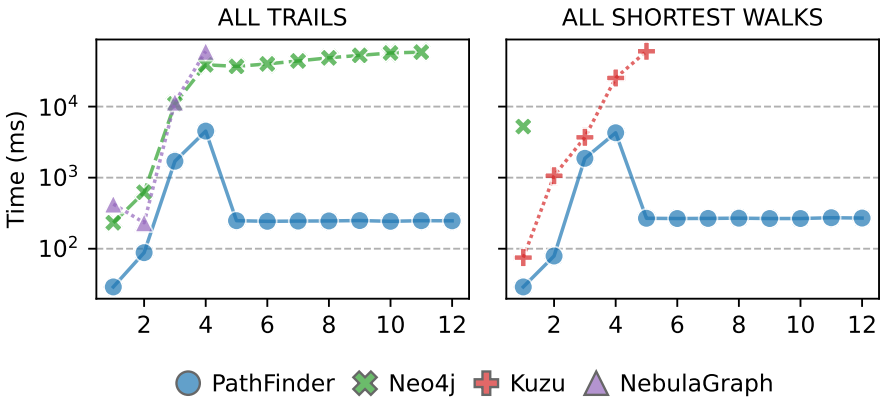


Fig. 5: Performance of graph engines in the Pokec dataset.

is again present for length 3 and 4, but it still results in fast performance. We also conducted an experiment where we look for paths of length precisely k, with $k = 1 \dots 12$, with identical results, so we omit the plot for this case. Overall, we can conclude that PathFinder offers stable performance, and unlike the other systems, does not get into issues as the allowed path length increases.

TRAILS. The results for ALL TRAIL is shown in Figure 5 (left). The performance of Neo4j is much better here than for SHORTEST WALKs, with timeouts occurring much later. On the other hand, Nebula could not handle length 5 paths. When it comes to PathFinder, the results show performance of the BFS version of Algorithm 3. For the DFS version (not shown in the figure) the picture is similar. As in other experiments, we see that PathFinder can handle the query load with no major issues.

B.2 Diamond: scaling the number of paths

Here we test the performance of the query looking for paths between the node start and the node end in the graph of Figure 3. We scale the size of the database by setting $n = 1, \dots, 40$. 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. To compare with other engines, we focus on the WALK restrictor and the TRAIL restrictor. All the other paths modes in PathFinder, which is the only system supporting them, have identical performance as in the TRAIL case, since they are all derivatives of Algorithm 3. Since SPARQL systems cannot return paths, we exclude them from this experiment.

SHORTEST WALKS. The runtimes for the ANY SHORTEST WALK and ALL SHORTEST WALK modes is presented in Figure 6. Here we compare with Neo4j and Kuzu, since Nebula only supports the TRAIL restrictor. Due to the small size of the graph, we run each query twice and report the second result. This

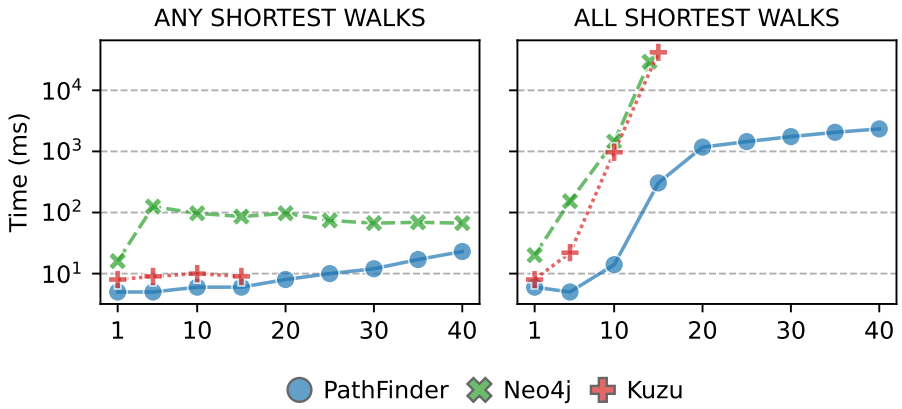


Fig. 6: WALK queries on the graph of Figure 3.

is due to minuscule runtimes which get heavily affected by initial data loading. As we can observe, for ANY SHORTEST WALK (Figure 6 (left)) all the engines perform well (we also pushed this experiment to $n = 1000$ with no issues). Notice that KUZU only works up to $n = 15$ since the longest path it supports is of length 30. In the case of ALL SHORTEST WALK (Figure 6 (right)), NEO4J times out for $n = 16$. Same as before, KUZU stops at $n = 15$ with a successful execution. Overall, PATHFINDER seems to have a linear time curve in this experiment, while the other engines grow exponentially, showing the full power of Algorithm 2 when returning 100,000 paths. We tried scaling to $n = 1000$ and the results were quite similar.

TRAILS. Apart from comparing with other systems, this experiment also allows to determine which traversal strategy (BFS or DFS) is better suited for Algorithm 3 in extreme cases such as the graph of Figure 3. We present the results for ANY TRAIL and TRAIL modes in Figure 7. Considering first the ANY TRAIL case, which is only supported by PATHFINDER, the BFS-based algorithm will time out already for $n = 26$, which is to be expected, since it will construct all paths of length $1, 2, \ldots, 25$, before considering the first path of length 26. In contrast, DFS will find the required paths rather fast. When it comes to the TRAIL mode, which retrieves *all* trails, the situation is rather similar. Here we also compare with other engines that find trails. As we can see, no engine apart from PATHFINDER-DFS could handle the entire query load as they all show an exponential performance curve. This illustrates that for a huge number of *trails*, DFS is the strategy of choice.

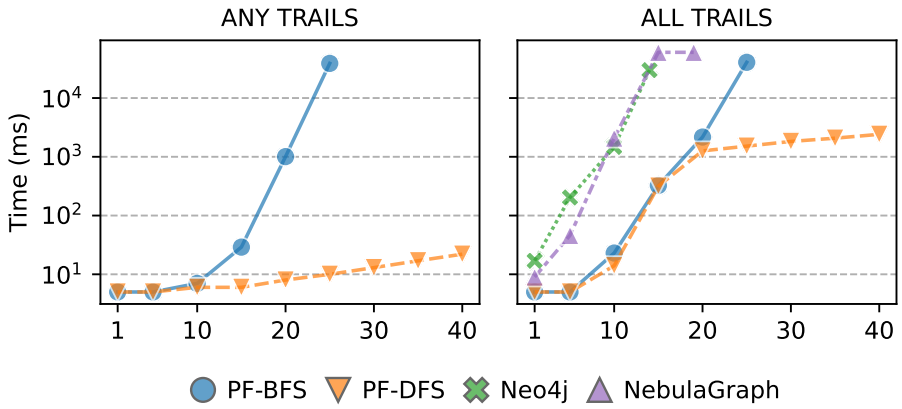


Fig. 7: TRAIL queries on the graph of Figure 3.

B.3 Conclusions

Based on our experiments, we believe that one can conclude that PathFinder offers a sound strategy for dealing with path queries. It is highly performant on all the query loads we considered, and runs faster than any other system in every scenario we tested. This is particularly true for the WALK semantics, which runs very fast and with few timeouts, even on huge datasets such as Wikidata. When it comes to TRAILS one has to be careful whether the BFS or DFS strategy is selected, with the former being a good candidate for highly connected graphs with few hops, and the latter being able to better handle a huge number of paths. Finally, we remark that PathFinder was tested only as a disk-based system that loads data into a main memory buffer as required by the queries. Our code [6] also includes an in-memory version of PathFinder, which uses the Compressed Sparse Row representation of graphs [12] in order to store the data in memory, and which runs about twice as fast as the results we presented (results not included for brevity).