PathFinder: A unified approach for handling paths in graph query languages

Anonymous Author(s)

ABSTRACT

Path queries are a core feature of modern graph query languages such as Cypher, SQL/PGQ, and GQL. These languages provide a rich set of features for matching paths, such as restricting to certain path modes (shortest, simple, trail) and constraining the edge labels along the path by a regular expression. In this paper we present PathFinder, a unifying approach for dealing with path queries in all these query languages. PathFinder leverages a compact representation of the (potentially exponential number of) paths that can match a given query, extends it with pipelined execution, and supports all commonly used path modes. In the paper we describe the algorithmic backbone of PathFinder, provide a reference implementation, and test it over a large set of real-world queries and datasets. Our results show that PathFinder exhibits very stable behavior, even on large data and complex queries, and its performance is an order of magnitude better than that of many modern graph engines.

KEYWORDS

regular path queries, GQL, SQL/PGQ

1 INTRODUCTION

Graph databases [4, 41] have gained significant popularity and are used in areas such as Knowledge Graphs [26], Biology [28], and The Semantic Web [5]. With the proliferation of different engines and query languages [38, 46–50, 58], there was an increasing need for a standard language for expressing graph queries. This led to several efforts such as SPARQL [25], a W3C standard for querying edge-labeled graphs, and SQL/PGQ [15], a recent ISO initiative to extend SQL with pattern matching for property graphs. Currently, ISO is still working on the Graph Query Language (GQL), which is planned to become a standard for querying property graphs [15]. Both SQL/PGQ and GQL are heavily influenced by Cypher [21].

Path queries are a core feature of all these graph query languages. In SPARQL these are supported through property paths, which are a variant of *regular path queries* (*RPQs*) that are well-studied in the literature [7, 12, 14, 36]. 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, it extracts all node pairs (`n1`, `n2`) such that there is a path in G linking `n1` to `n2`, and whose edge labels form a word that matches `regex`.

However, RPQs 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, it is desirable to also see the paths. This shortcoming of RPQs was recognized by the ISO standardization committee by making paths first-class citizens. Indeed, GQL and SQL/PGQ enrich RPQs with the ability to return the matching paths, filter on the type of such paths, limit the number of such paths, and provide many other features for path manipulation.

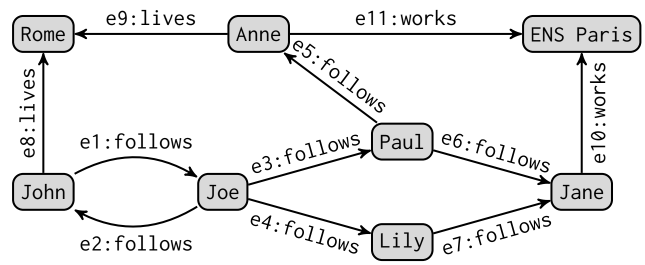


Figure 1: A sample graph database.

To illustrate these features consider the graph in Figure 1. Here we have node identifiers (such as Joe or Rome), edge identifiers (such as e1), and edge labels (for instance follows for e7).[1] A natural task would be to explore the influence network of, say, Joe, i.e., finding the people that Joe follows, then people they follow, and so on, transitively traversing follows-edges. This can be written as an RPQ of the form $(\text{Joe}, \texttt{follows}^+, ?x)$, signaling that we wish to start at Joe, and traverse any nonzero number of follows-edges. In this case, the query returns all the people in the database. 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 ensure the number of returned paths to be finite, GQL and SQL/PGQ [15] provide so-called *path modes*.

We give a few examples of path modes and provide comprehensive details later. Two common path modes 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, there is precisely one path that leaves Joe and comes back; namely, through the edge e2 and then back via e1.

Another common way to restrict paths is by selecting only the shortest of them, or even non-deterministically choosing a single shortest path. In GQL and SQL/PGQ, ANY SHORTEST will return a single shortest path for each different pair of endpoints in the answer, whereas ALL SHORTEST will return all such shortest paths. Consider the RPQ $(\text{Joe}, \texttt{follows}^+ \cdot \texttt{works}, ?x)$, which looks for the working place of people that Joe follows. In the graph of Figure 1, there are three shortest paths linking Joe to ENS Paris (traced by the edges e3 $\rightarrow$ e5 $\rightarrow$ e11, e4 $\rightarrow$ e7 $\rightarrow$ e10, and e3 $\rightarrow$ e6 $\rightarrow$ e10, respectively), and we might wish to retrieve all of them.

While GQL and SQL/PGQ recognize the need to allow full regular path queries *and* return paths at the same time, the support for these features in modern graph databases is lacking. In Table 1 we review path querying features in several leading graph systems. We see that, apart from SPARQL engines, which cannot return any paths, RPQs are only partially supported in property graph engines.

[1] We consider a slightly simplified model of property graphs in this paper to keep the discussion concise. Our results transfer verbatim to the real-world setting.

	RPQs	WALK	TRAIL	SIMPLE	ACYCLIC	SHORTEST
BlazeGraph [52]	✓	–	–	–	–	–
Jena [47]	✓	–	–	–	–	–
Virtuoso [16]	✓	–	–	–	–	–
Neo4j [58]	partial	✓	✓	–	–	✓
Nebula [55]	partial	partial	✓	–	✓	✓
Kuzu [27]	partial	✓	–	–	–	✓
DuckPGQ [50]	partial	✓	–	–	–	✓
PathFinder	✓	✓	✓	✓	✓	✓

Table 1: Support for path queries in graph database engines.

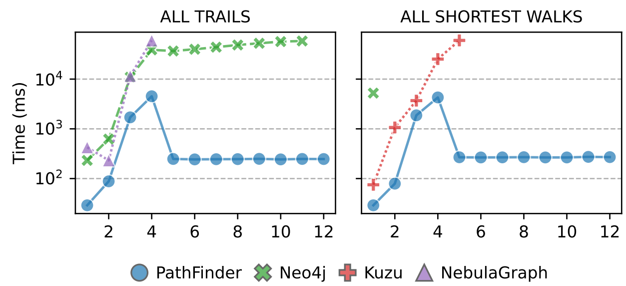


Figure 2: Performance of graph engines in the Pokec dataset.

More precisely, all engines under review only support regular expressions where the Kleene star is placed over a single edge label, or a disjunction of labels. Neo4j [58] is a bit more general, but does not support more complex expressions such as $(\texttt{follows} \cdot \texttt{knows})^*$. In brief, no engine supports all regular expressions in RPQs. Similarly, when it comes to returning different types of paths, engines usually pick a single path type to support. The most powerful in this sense is Nebula [55], which can return trails and acyclic paths, and can detect the presence of arbitrary paths (called "walks"), but cannot return walks. In general, none of the modern graph engines fully support RPQs with the path modes required by GQL and SQL/PGQ.

The lack of full support for RPQs is not surprising. In order to deal with them in a scalable fashion, one needs to find a way to cope with the potentially exponential number of paths that matches an RPQ in the data [33]. In fact, most engines suffer from performance issues when returning paths, especially as the length of paths grows. We illustrate this with a simple experiment on the Pokec dataset (1.6M nodes, 30M edges) from the SNAP graph collection [30], which contains a social network similar to the one in Figure 1. In our experiment we select the node that is median in terms of centrality,[2] and explore its influence network while also returning up to 100,000 paths that witness the connections. We do so for paths of length 1 to 12, since the diameter of our network is capped at 11. The experimental results are in Figure 2. On the left we consider ALL TRAILS and on the right ALL SHORTEST WALKS. No tested engine can finish the entire query load apart from PathFinder. Considering ALL TRAILS, Nebula starts timing out already for paths of length 5, while Neo4j works up to paths of length 11. Kuzu does not support returning trails. On the image on the right, Neo4j can only compute all shortest paths for length 1 paths, while Kuzu times out for paths of length 6 or more. We remark that Nebula can find all shortest trails, but it times out for all path lengths.

GQL and SQL/PGQ justify the need for efficient algorithms that can both support *all* RPQs, and *all* prescribed path modes, which are currently lacking. We aim to address this problem in the paper, which is quite fundamental to these standards, since RPQ query answering is at the core of path processing in the two standards.

Our contribution. We present PathFinder, the first system that supports *returning paths in graph query answers*, for *all regular path queries* and *all path modes prescribed by GQL and SQL/PGQ*:

- We enhance path multiset representations (PMRs) [33], a compact data structure for storing sets of paths, with the ability to return paths in a pipelined fashion. We go beyond the conceptual idea of PMRs and focus on returning answers (including paths) to the user (or next operator in the query plan) as quickly as possible.
- We describe PathFinder, the first system that supports all 15 different path modes described in the SQL/PGQ and GQL standards; all under pipelined evaluation.
- The paper provides explicit algorithms on how to compute succinct pipelined representations and return paths.
- Compared to other systems that partially support RPQs with path modes or even to systems that cannot return paths, we show in an extensive experimental study that PathFinder scales drastically better.

Organization. We define graph databases and formalize the RPQ fragment of GQL and SQL/PGQ in Section 2. Algorithms for the WALK mode are studied in Section 3, while TRAIL and SIMPLE are tackled in Section 4. We discuss the experimental results in Section 5. Related work is discussed in Section 6. We conclude in Section 7. Additional details, code, and proofs can be found at [6].

2 GRAPH DATABASES AND RPQS

In this section we define graph databases and regular path queries as supported by the GQL and SQL/PGQ standard.

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 [5, 15, 20, 33], we define graph databases as follows.

Definition 2.1. 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. Intuitively, $\rho(e) = (v_1, v_2)$ means that e is a directed edge going from v_1 to v_2.
- $\lambda : E \to$ Lab is a total function assigning a label to an edge.

Similarly as in [33], we use a simplified version of property graphs [15], 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 [39] and edge-labeled graphs [7, 36], which do not use explicit edge identifiers.

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

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.[3]
- 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 in GQL and SQL/PGQ. We study *regular path queries* (*RPQs* for short), which form the basis of navigation in GQL and SQL/PGQ [15]. 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 [15], 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:

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

Traditionally [5], 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 a path p from v to v', such that $\mathsf{lab}(p)$ is a word in the language of the regular expression regex.[4] Since the set of all such paths can be infinite [5], 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:

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

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

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

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

- $[\![\text{ANY restrictor } (v, \texttt{regex}, ?x)]\!]_G$ 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.
- $[\![\text{ANY SHORTEST restrictor } (v, \texttt{regex}, ?x)]\!]_G$ returns, for a 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 SHORTEST among all paths p' for which $(p', v') \in [\![q]\!]_G$.
- $[\![\text{ALL SHORTEST restrictor } (v, \texttt{regex}, ?x)]\!]_G$ 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 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 15 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.

Output-linear delay. Returning paths can result in a large number of outputs. Therefore, to measure the efficiency of our algorithms, we will use the paradigm of *enumeration algorithms* [8, 18, 33, 43]. Such algorithms work in two phases: a *pre-processing phase*, which constructs a data structure allowing us to output the solutions; and the *enumeration phase*, which enumerates these solutions *without repetitions*. The efficiency of an enumeration algorithm is measured by the complexity of the pre-processing phase, and the delay between any two outputs produced during the enumeration phase. We will say that an enumeration algorithm works with *output-linear delay*, when the delay is linear in the size of each output element. This means that the time needed to output a single path is linear in the number of nodes in the path, after which we immediately start to output the next path. Notice that this is optimal in a sense.

Regular expressions and automata. We assume basic familiarity with regular expressions and finite state automata [40]. If regex is a regular expression, we will denote by $\mathcal{L}(\texttt{regex})$ the language of 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. A regex can be converted into an equivalent NFA of size linear in $|\texttt{regex}|$ (see [40]). 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 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.
[4]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 for simplicity we consider the more limited case. We later comment on how to lift this restriction (see Section 5).

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:

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

Notice that here the selector part is obligatory, since the set of all walks can be infinite. We first construct a compact representation for all the paths in the answer [33] using the product construction [36, 40]. 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 $q = (v, \texttt{regex}, ?\texttt{x})$, the *product graph* is constructed by first converting the regular expression regex into an equivalent non-deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$. 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 regex. 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 $\mathcal{L}(\texttt{regex})$. 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).

It is important for efficiency that we only construct the subgraph of $G_\times$ that is needed for query answers. In [33] it is described how to efficiently construct the *trim* subgraph, which is the subgraph of $G_\times$ that has the nodes on a path from (v, q_0) to some node of the form (v', q_n) with $q_n \in F$. (Indeed, these are exactly the nodes of $G_\times$ that are useful for any answer of the WALK query [33].)[5]

We now take this a step further by only building $G_\times$ on-the-fly as we are exploring the product graph. Additionally, we show how the product graph can be used to return paths.

3.1 ANY (SHORTEST) WALKS

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 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 BFS and DFS 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). Query evaluation is presented in Algorithm 1.

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

```
function AnyWalk(G, query)
    𝒜 ← Automaton(regex)                 ▷ q0 initial state, F final states
    Open.init()
    Visited.init()
    ReachedFinal.init()
    startState ← (v, q0, null, ⊥)
    Visited.push(startState)
    Open.push(startState)
    if v ∈ V and q0 ∈ F then
        ReachedFinal.add(v)
        Solutions.add(startState)
    while Open ≠ ∅ do
        current ← Open.pop()             ▷ current = (n, q, edge, prev)
        for next = (n′, q′, edge′) ∈ Neighbors(current, G, 𝒜) do
            if (n′, q′, ∗, ∗) ∉ Visited then
                newState ← (n′, q′, edge′, current)
                Visited.push(newState)
                Open.push(newState)
                if q′ ∈ F and n′ ∉ ReachedFinal then
                    ReachedFinal.add(n′)
                    Solutions.add(newState)
    while Solutions ≠ ∅ do               ▷ Enumerate solutions
        sol = Solutions.pop()
        print(getPath(sol, []))          ▷ Retrieve paths

function getPath(state = (n, q, edge, prev), output)
    if prev == ⊥ then                    ▷ Initial state
        return [v] + output
    else                                 ▷ Recursive backtracking
        return getPath(prev, [edge, n] + output)
```

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 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.
- Solutions, which is a set containing (pointers to) search states in Visited that encode a solution path to be returned; and

[5] Importantly, the trim subgraph of $G_\times$ can be computed in optimal time, i.e., in the same time than it would take to decide *if there exists a path labeled with some word in* $\mathcal{L}(\texttt{regex})$ *from* u *to a given node* v', if the BMM conjecture holds [13].

- ReachedFinal, a set containing nodes we already returned as query answers, in case we re-discover them via a different end state (recall that an NFA can have several end states).

The algorithm can explore the product $G_\times$ 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, so we add it to Solutions (line 21). 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. Finally, the solutions are enumerated in lines 22–24 by traversing the pointers defined by search states stored in Visited. Here we use the function getPath which constructs the array representation of the path by prepending the edge and node of the current search state to the result constructed thus far before proceeding to the previous one. Notice that we take time proportional to each path to output it, thus achieving output-linear delay. 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. For DFS an arbitrary path is returned.

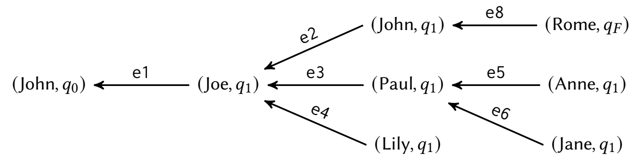


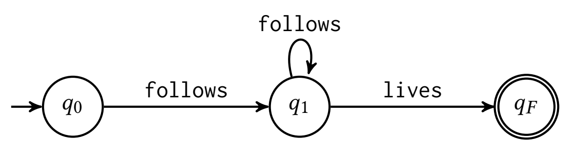
Figure 3: Visited after running Algorithm 1 in Example 3.1.

Pipelined execution. In principle, solutions can be returned as soon as they are detected (line 19). Freezing the state of the algorithm at this point, and continuing it on demand achieves pipelined-execution. An extended version of the paper available at [6] provides a detailed explanation of how the algorithm can be implemented using a linear iterator interface needed in the pipelined execution model. The same pattern holds for all the other algorithms presented in the remainder of the paper.

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

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

Namely, we wish to find places where people that John follows live. Looking at the graph in Figure 1, we see that Rome is such a place, and the shortest path reaching it starts with John, and loops back to him using the edges e1 and e2, before reaching Rome (via e8), as required. To compute the answer, Algorithm 1 first converts the regular expression $\texttt{follows}^+ \cdot \texttt{lives}$ 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 Solutions. 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 3. 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)). Since Solutions contains only (Rome, q_F), we enumerate a single path traced by the edges e1 $\rightarrow$ e2 $\rightarrow$ e8. □

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 $[\![q]\!]_G$ with $O(|\mathcal{A}| \cdot |G|)$.*

Notice that, since we only return a single answer, this also means that the algorithm runs in output-linear delay.

3.2 ALL SHORTEST WALKS

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

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

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: 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. We refer to Algorithm 2 for details.

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 to (n, q) from (v, q_0);
- *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 empty, 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 $G_\times$ of 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 14), we do not directly discard the new path, but instead choose to keep it if and only if it is also shortest (line 16). In this case, the *prevList* for (n', q') is extended by adding the new path (line 17). If a new pair (n', q') is discovered for the first time, a fresh *prevList* is created (lines 18–23). The second difference to Algorithm 1 lies in the fact that we now record solutions only after a state has been removed from Open (lines 10–12). 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., in line 13 we can test if $q' \in F$), in order to achieve pipelined execution.* Returning answers when popping from the queue in lines 10–12 has an added benefit that the paths are grouped for a pair (v, n) of connected nodes. Finally, we enumerate the solutions using getAllPaths, which traverses the DAG stored in

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

1: **function** AllShortestWalk(G, $query$)
2: $\quad\mathcal{A} \leftarrow$ Automaton(regex) ▷ *q_0 initial state, q_F final state*
3: $\quad$Open.init()
4: $\quad$Visited.init()
5: $\quad$**if** $v \in V$ **then**
6: $\quad\quad$startState $\leftarrow (v, q_0, 0, \perp)$
7: $\quad\quad$Visited.push(startState)
8: $\quad\quad$Open.push(startState)
9: $\quad$**while** Open $\neq \varnothing$ **do**
10: $\quad\quad$current $\leftarrow$ Open.pop() ▷ *current = $(n, q, depth, prevList)$*
11: $\quad\quad$**if** $q == q_F$ **then**
12: $\quad\quad\quad$Solutions.add(current)
13: $\quad\quad$**for** next = $(n', q', edge') \in$ Neighbors(current, G, $\mathcal{A}$) **do**
14: $\quad\quad\quad$**if** $(n', q', *, *) \in$ Visited **then**
15: $\quad\quad\quad\quad(n', q', depth', prevList') \leftarrow$ Visited.get(n', q')
16: $\quad\quad\quad\quad$**if** $depth + 1 == depth'$ **then**
17: $\quad\quad\quad\quad\quad prevList'$.add($\langle$ current, $edge'\rangle$)
18: $\quad\quad\quad$**else**
19: $\quad\quad\quad\quad prevList$.init()
20: $\quad\quad\quad\quad prevList$.add($\langle$ current, $edge'\rangle$)
21: $\quad\quad\quad\quad$newState $\leftarrow (n', q', depth + 1, prevList)$
22: $\quad\quad\quad\quad$Visited.push(newState)
23: $\quad\quad\quad\quad$Open.push(newState)
24: $\quad$**while** Solutions $\neq \emptyset$ **do** ▷ Enumerate solutions
25: $\quad\quad$sol = Solutions.pop()
26: $\quad\quad$getAllPaths(sol, []) ▷ Retrieve paths

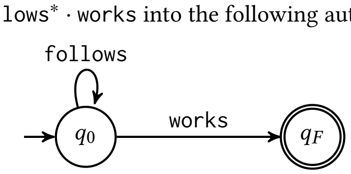
27: **function** getAllPaths(state = $(n, q, depth, prevList)$, output)
28: $\quad$**if** $prevList == \perp$ **then** ▷ *Initial state*
29: $\quad\quad$print($[v]$ + output)
30: $\quad$**for** prev = (prevState, $prevEdge$) $\in prevList$ **do**
31: $\quad\quad$getAllPaths(prevState, $[prevEdge, n]$ + output)

Visited in a depth-first manner. The method getAllPaths receives the current search state in Visited and the path reconstructed thus far (as an array output), and prepends the current node and edge to the current solution, continuing recursively until reaching v. 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

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

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 $\rightarrow$ e5 $\rightarrow$ e11, e4 $\rightarrow$ e7 $\rightarrow$ e10, and e3 $\rightarrow$ e6 $\rightarrow$ e10, respectively. To compute $[\![q]\!]_G$, Algorithm 2 converts the regular expression $\texttt{follows}^* \cdot \texttt{works}$ into the following automaton $\mathcal{A}$:



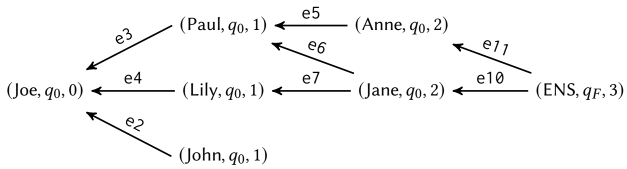


Figure 4: Visited after running Algorithm 2 in Example 3.3.

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 14–17 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 14–17 are triggered again. This time the depth test in line 16 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.

Figure 4 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.

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

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

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

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 14–17) 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 or trail between two nodes that conforms to a regular expression and is a simple path, acyclic path, or a trail is NP-complete [7, 9, 14, 34]. 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. Intuitively, we will be exploring an "unraveling" of 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, q) in $G_\times$, 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.
- Solutions, which is a set containing (pointers to) search states in Visited that encode a solution path to be returned.
- 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 record the solution. If the ALL SHORTEST selector is *not* present, we simply add the newly found solution (line 21). In the presence of the selector, we

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:      𝒜 ← Automaton(regex)      ▷ q0 initial state, F final states
3:      Open.init()
4:      Visited.init()
5:      ReachedFinal.init()
6:      startState ← (v, q0, 0, null, ⊥)
7:      Visited.push(startState)
8:      Open.push(startState)
9:      if v ∈ V and q0 ∈ F then
10:         ReachedFinal.add(⟨v, 0⟩)
11:         Solutions.add(startState)
12:     while Open ≠ ∅ do
13:         current ← Open.pop()  ▷ current = (n, q, depth, e, prev)
14:         for next = (n′, q′, edge′) ∈ Neighbors(current, G, 𝒜) do
15:             if isValid(current, next, restrictor) then
16:                 new ← (n′, q′, depth + 1, edge′, current)
17:                 Visited.push(new)
18:                 Open.push(new)
19:                 if q′ ∈ F then
20:                     if !(ALL SHORTEST) then
21:                         Solutions.add(new)
22:                     else if n′ ∉ ReachedFinal then
23:                         ReachedFinal.add(⟨n′, depth + 1⟩)
24:                         Solutions.add(new)
25:                     else
26:                         optimal ← ReachedFinal.get(n′).depth
27:                         if depth + 1 == optimal then
28:                             Solutions.add(new)
29:     while Solutions ≠ ∅ do                 ▷ Enumerate solutions
30:         sol = Solutions.pop()
31:         print(getPath(sol, []))            ▷ Retrieve paths
32: function isValid(state, next, restrictor)
33:     s ← state
34:     if s.node == v and s.prev ≠ ⊥ then return False
35:     while s ≠ ⊥ do
36:         if restrictor == ACYCLIC then
37:             if s.node == next.node then
38:                 return False
39:         else if restrictor == SIMPLE then
40:             if s.node == next.node and s.prev ≠ ⊥  then
41:                 return False
42:         else if restrictor == TRAIL then
43:             if s.edge == next.edge then
44:                 return False
45:         s ← s.prev
        return True
```

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 22–24). Upon discovering the same node again (lines 25–28), a new solution is added only if it is shortest. Once all paths have been explored (Open $= \emptyset$), we enumerate the solutions in lines 29–31 using the getPath function, which is identical to the one from Algorithm 1, but now uses extended search states. *For pipelined execution we would return results as soon as they pass the tests in lines 19–28, and resume the algorithm on demand.*

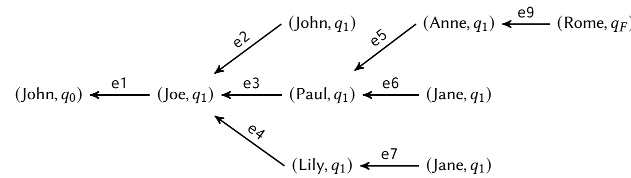


Figure 5: Visited after running Algorithm 3 in Example 4.1.

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

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

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 5. In Figure 5 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 5. 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 $(\text{Jane}, q_1, 3, \text{e7}, prev')$. □

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:

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

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

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 $O\big((|\mathcal{A}| \cdot |G|)^{|G|}\big)$.*

4.2 ANY and ANY SHORTEST

To treat queries of the form:

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

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 $\mathcal{L}(\texttt{regex})$. We would then replace lines 20–28 of Algorithm 3 with the following:

if $n' \notin$ ReachedFinal **then**
  ReachedFinal.add(n')
  Solutions.add(new)

Basically, when n' is discovered as a solution for the first time, we record 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 [14] 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:*

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

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

5 EXPERIMENTAL EVALUATION

We empirically evaluate PathFinder and show that the approach scales on a broad range of real-world and synthetic data sets and 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 see [6].

Implementation. PathFinder is implemented as the path processing engine of AnonymousDB, a recent graph database system developed by the authors[6]. AnonymousDB provides the infrastructure to process generic queries supported in GQL, and takes care of parsing, generation of execution plans, and data storage, while PathFinder executes path queries as described in the paper. The default data storage model 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 [12] and SPARQL property paths [25], 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. 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. The case when both ends are variables; namely, $(?x, \texttt{regex}, ?y)$, is supported by running the algorithm for every node that is a source of some edge labeled by a transition that leaves the initial state of the automaton for regex. As stated previously, *all the algorithms are implemented in a fully pipelined fashion, meaning that they can be paused as soon as a result is detected.* The source code can be obtained at [6].

[6]To preserve anonymity system name is not presented in the submission.

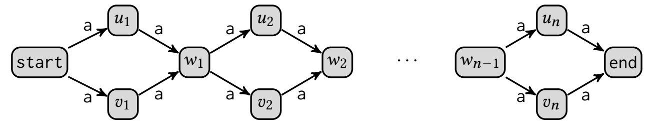


Figure 6: Graph database with exponentially many paths.

5.1 Experimental setup

We perform three sets of experiments:

- Pokec, which tests the effect of path length on performance;
- Wikidata, where we test the performance over a large real-world graph and user supplied queries; and
- Diamond, where we test the effect of having a large number of paths in the graph.

Next we describe each set of experiments in more detail.

(1) Pokec. This is an extension of the experiment described in the Introduction, using the Pokec social network graph from SNAP [30]. 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. As described, we fix a node that is median in terms of centrality, 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. 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) Wikidata. Here, we want to check performance over a large real-world graph. For this we use Wikidata [56] and queries from its public SPARQL query log [32]. 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 [19], 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 the Wikidata's public endpoint query log. We use the Paths query set, which contains 659 2RPQs patterns. From these, 592 have a fixed starting point or ending point (or both), while 67 have both endpoints free. We note that these queries require general regular expressions which cannot be expressed in some of the tested systems. The 659 patterns are then used in our tests under the restrictor and selector options described in Section 2.

(3) Diamond. In our final experiment we test what happens when there is a large number of paths present in our graph. The database we use, taken from [33], is presented in Figure 6. 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.

Tested systems. PathFinder supports both BFS traversal and DFS traversal (when paths need not be shortest). We denote the two versions as PathFinder-BFS and PathFinder-DFS, respectively. When there is only one algorithm (e.g. for all shortest walks), we simply write PathFinder. All the versions assume data to be stored on disk and being buffered into main memory as required.

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 [58] (Neo4j for short);
- NebulaGraph version 3.5.0 [55] (Nebula);
- Kuzu version 0.0.6 [27] (Kuzu);
- Jena TDB version 4.1.0 [47] (Jena);
- Blazegraph version 2.1.6 [52] (BlazeGraph); and
- Virtuoso version 7.2.6 [16] (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 [25], detect pairs of nodes connected by an arbitrary walk. A brief summary of supported features can be found in Table 1 in the Introduction. Other systems we considered are DuckDB [50], Oracle Graph Database [38] and Tiger Graph [49], which support (parts of) SQL/PGQ. Unfortunately, [38] and [49] are commercial systems with limited free versions, while the SQL/PGQ module for DuckDB [50] 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.*

5.2 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 7 and Figure 2 (from the Introduction).

ENDPOINTS. Results for retrieving reachable nodes is presented in Figure 7 (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

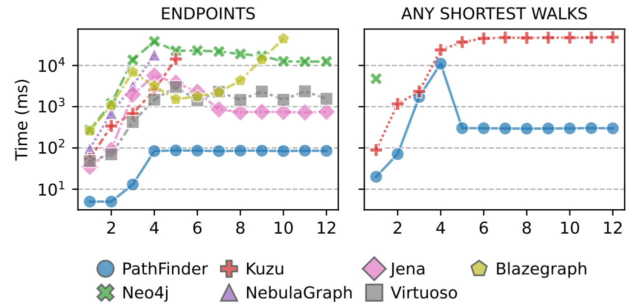


Figure 7: Runtimes over the Pokec dataset.

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 7 (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 was presented in Figure 2 (right) in the Introduction. 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 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 were shown in Figure 2 (left) in the Introduction. 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.

5.3 Wikidata: the effect of big graphs

Here we test whether evaluating path queries in big real-world graphs is feasible. We encountered significant issues when loading the data set into some engines. For Nebula we ran into the well documented storage issue [54] that we could not resolve, while Kuzu ran out of memory while loading the data set. We even tried splitting the data set into smaller chunks, with one file for each distinct edge label. In this case we only managed to load the ten biggest edge sets into Kuzu, but this amounted to less than a third of the total number of edges, so we excluded Kuzu from this experiment. Apart from PathFinder, systems that could load the data

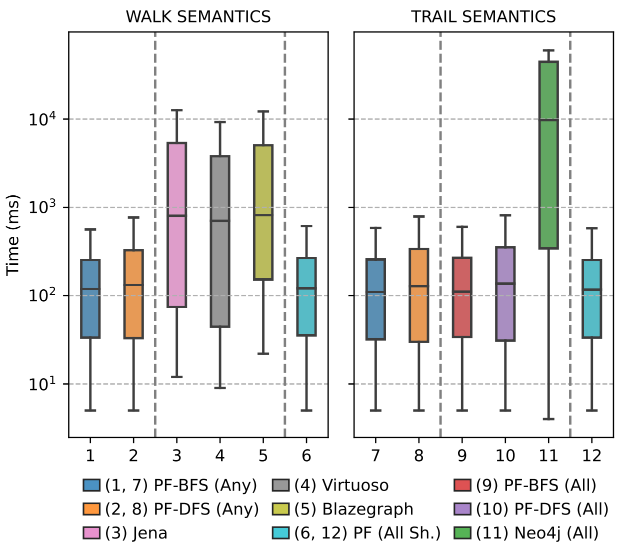


Figure 8: Runtimes for the Wikidata experiment.

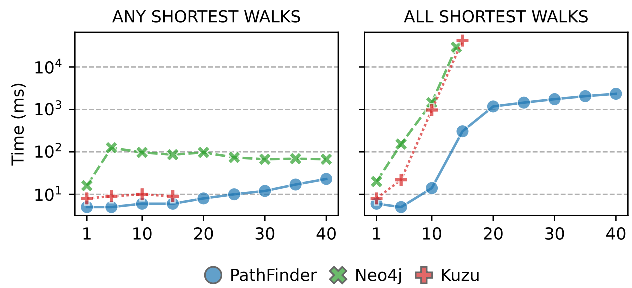


Figure 9: WALK queries on the graph of Figure 6.

were: Neo4j, BlazeGraph, Jena, and Virtuoso. Given that Neo4j supports the WALK and TRAIL restrictor, we ran our queries under these two path modes. Out of 659 queries, 20 could not be expressed in Neo4j since they were complex RPQs. Results are in Figure 8.

WALKS. In Figure 8 (left) we show the results for the WALK restrictor. Since we run 659 queries, we only present the box plots of our results. The first two columns represent the BFS and DFS version of Algorithm 1 in PathFinder, which corresponds to ANY (SHORTEST) WALK mode. Compared to SPARQL engines (the next three columns), which do not return paths, we can see a marked improvement. This is also reflected in the number of timeouts. Here PathFinder-BFS had 8 and PathFinder-DFS 9 timeouts. In contrast, Jena timed out 95 times, and BlazeGraph and Virtuoso 86 and 24 times, respectively. We remark that while Neo4j supports ANY SHORTEST WALK mode, it timed out in 657 out of 659 queries, so we do not include it in the graphs. The final column in Figure 8 (left) 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.

TRAILS. Results for the TRAIL semantics are shown in Figure 8 (right). The first two columns correspond to ANY SHORTEST TRAIL and ANY TRAIL in PathFinder. The performance here is almost identical to the ANY WALK case, with only 25 and 24 timeouts for the BFS and DFS version, respectively. The following three columns correspond to the ALL TRAIL path mode, supported by PathFinder-BFS, PathFinder-DFS, and Neo4j. Comparing PathFinder to Neo4j, we can see an order of magnitude improvement in performance. This is reflected in the number of timeouts with 134 for Neo4j, and only 9 for PathFinder-BFS and 10 for PathFinder-DFS. The final column corresponds to ALL SHORTEST TRAIL mode in PathFinder, which again shows similar performance to other TRAIL-based modes, with only 21 timeouts. Overall, we can see that PathFinder shows remarkably stable performance when returning trails. Interestingly, while the theoretical literature classifies the TRAIL mode as intractable [7], 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. This is most likely due to the fact that they can either detect 100,000 results rather fast, or because the data itself permits no further graph exploration. We remark that 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.

5.4 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 6. We scale the size of the database by setting $n = 1, \ldots, 40$. 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 9. 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 is due to minuscule runtimes which get heavily affected by initial data loading. As we can observe, for ANY SHORTEST WALK (Figure 9 (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 9 (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.

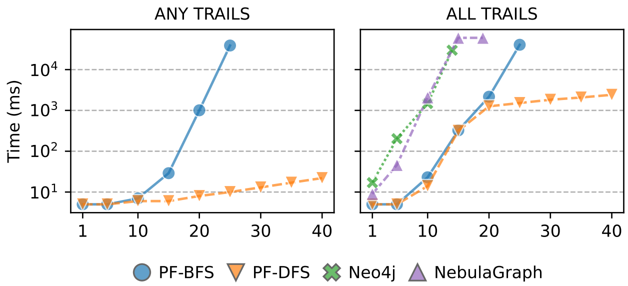


Figure 10: TRAIL queries on the graph of Figure 6.

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 6. We present the results for ANY TRAIL and TRAIL modes in Figure 10. 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.

5.5 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 [11] 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).

6 RELATED WORK

Our work builds on top of [33], where a method to construct a compact representation of a set of paths witnessing an RPQ query answer under *some* of the GQL path modes is presented. In contrast, our work focuses on *returning* paths to the user. Both [33] and our approach leverage the product construction [7] (Section 3), but we extend the approach of [33] in several important ways. First, we present algorithms that run on top of the product graph and return paths witnessing RPQ query answers in a pipelined fashion, instead of computing the entire result set as in [33]. Second, we support returning trails, simple paths, and acyclic paths, unlike in [33], and we also implement the nondeterministic ANY variant of all GQL path modes, unlike [33]. Third, our focus is on algorithms that are easily implementable (as witnessed by our complementary material [6]), whereas [33] mostly focuses on theoretical guarantees. In terms of other 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 [10, 17, 22, 23, 59], 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 [24] 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 [42], 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. Similar approach for top-k results is presented in [1], but not preferring shortest paths. Interesting optimizations for the base BFS-style algorithm are presented in [29, 31, 51], where vectorized execution of BFS with multiple starting points is explored. Incorporating these techniques to PathFinder is a promising direction for future work. Regarding the simple path semantics, [57] proposes a sampling-based algorithm which is an efficient approximate answering technique, but will not necessarily find *all* answers. There is a rich body of theoretical work on RPQ evaluation (see [7] for an overview and [9, 34, 35] for recent results on returning paths). Interestingly, in [13] Casel and Schmidt proved that, under the BMM conjecture, one requires $|G||q|$ time to *decide* if a given node pair is the answer of an RPQ q on G, essentially showing that the product construction used in Section 3 is optimal. Again, their focus is on returning nodes and not paths.

Reachability indices. There is extensive work on speeding-up graph traversal algorithms via reachability indexes [44, 45, 53, 60, 61]. Of these, [23] incorporated [44] to allow finding nodes reachable by paths conforming to an RPQ. In terms of indices developed for regular path queries, apart from some early theoretical work [37], there is also a recent proposal [62], that allows checking whether two nodes are connected by an RPQ-conforming path. In contrast, we design algorithms for retrieving paths over base data. However, using reachability indices to speed up our proposal is a promising direction of future work.

7 CONCLUSIONS AND LOOKING AHEAD

In this paper 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 *any* path mode prescribed by the GQL and SQL/PGQ query standards [15], and that our experimental evaluation shows the approach to be highly competitive and scalable on realistic workloads. In future work we would like to extend PathFinder to cover the GQL top-k semantics, allow combining different path modes within a single query, and support path features in GQL and SQL/PGQ that go beyond RPQs. Additionally, we plan to test how optimization techniques such as reachability indices [45, 62], or vectorized and parallel execution of BFS and DFS [29, 51] can speed up the algorithms we proposed.

REFERENCES

[1] Christian Aebeloe, Gabriela Montoya, Vinay Setty, and Katja Hose. 2018. Discovering Diversified Paths in Knowledge Bases. *Proc. VLDB Endow.* 11, 12 (2018), 2002–2005. https://doi.org/10.14778/3229863.3236245

[2] 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

[3] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. *WDBench Dataset Download*. https://doi.org/10.6084/m9.figshare.19599589

[4] 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*.

[5] 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

[6] Anonymous Authors. 2023. PathFinder: A unified approach for handling paths in graph query languages. https://github.com/AnonCSR/PathFinder

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

[8] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, Vol. 4207. Springer, 167–181.

[9] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Symposium on Principles of Database Systems (PODS)*. 261–272.

[10] 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

[11] 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.

[12] 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.

[13] Katrin Casel and Markus L. Schmid. 2021. Fine-Grained Complexity of Regular Path Queries. In *International Conference on Database Theory (ICDT)*. 19:1–19:20.

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

[15] 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

[16] 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

[17] 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.

[18] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* 45, 1 (2020), 3:1–3:42.

[19] The Wikimedia Foundation. 2021. Wikidata:Database download. https://www.wikidata.org/wiki/Wikidata:Database_download

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

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

[26] 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

[27] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. 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. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

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

[29] 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.

[30] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[31] Jia Li, Wenyue Zhao, Nikos Ntarmos, Yang Cao, and Peter Buneman. 2023. MITra: A Framework for Multi-Instance Graph Traversal. *Proc. VLDB Endow.* 16, 10 (2023), 2551–2564. https://www.vldb.org/pvldb/vol16/p2551-cao.pdf

[32] 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*.

[33] 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

[34] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*. 7:1–7:16.

[35] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *International Conference on Database Theory (ICDT)*. 19:1–19:21.

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

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

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

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

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

[41] 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.

[42] 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

[43] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, Wang-Chiew Tan, Giovanna Guerrini, Barbara Catania, and Anastasios Gounaris (Eds.). ACM, 10–20.

[44] 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.

[45] 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.

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

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

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

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

[50] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A. Boncz. 2023. 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. https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf

[51] 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.

- [52] 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
- [53] 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.
- [54] Vesoft Inc. 2023. NebulaGraph Documentation. https://docs.nebula-graph.io/3.5.0/20.appendix/0.FAQ/
- [55] Vesoft Inc/Nebula. 2023. NebulaGraph. https://www.nebula-graph.io/
- [56] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489
- [57] 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.
- [58] Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH '12*, Gary T. Leavens (Ed.). https://doi.org/10.1145/2384716.2384777
- [59] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1875–1889. https://doi.org/10.1145/2882903.2882944
- [60] 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.
- [61] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph reachability queries: A survey. *Managing and Mining Graph Data* (2010), 181–215.
- [62] 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

Here we provide a pipelined version for all of our algorithms.

B 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.

B.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).

B.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.

B.3 ALL RESTRICTED PATHS

In the case of restrictor based semantics, Algorithm 7 computes ALL paths that satisfy a specific restrictor. 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 36).

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

```
function Begin(G, query)
    𝒜 ← Automaton(regex)                  ▷ q0 initial state, F final states
    Open.init()
    Visited.init()
    ReachedFinal.init()
    startState ← (v, q0, null, ⊥)
    Visited.push(startState)
    Open.push(startState)
    iter ← null                           ▷ Index iterator for neighbors
    firstNext ← True                      ▷ First time calling Next

function Next(G, query)
    if firstNext then                     ▷ Check if initial state is solution
        firstNext ← False
        if v ∈ V and q0 ∈ F then
            ReachedFinal.add(v)
            return v
    while Open ≠ ∅ do
        current ← Open.front()            ▷ current = (n, q, edge, prev)
        reached ← expandAny(current)
        if reached ≠ null then            ▷ New solution found
            return getPath(reached, [])
        else                              ▷ State was fully expanded
            iter ← null
            Open.pop()
    return null                           ▷ No more solutions

function expandAny(state = (n, q, edge, prev))
    if iter == null then                  ▷ First time state is explored
        if |𝒜.transitions(q)| == 0 then
            return null
        else                              ▷ Set the index iterator
            transitionIdx ← 0
            edgeType ← 𝒜.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    while transitionIdx < |𝒜.transitions(q)| do
        transition ← 𝒜.transitions(q)[transitionIdx]
        q′ ← transition.to                ▷ Next automaton state
        while iter.next() ≠ null do       ▷ iter = (n′, edge′)
            if (n′, q′, ∗, ∗) ∉ Visited then
                newState ← (n′, q′, edge′, state)
                Visited.push(newState)
                Open.push(newState)
                if q′ ∈ F and n′ ∉ ReachedFinal then
                    ReachedFinal.add(n′)
                    return newState
        transitionIdx++                   ▷ Next transition
        if transitionIdx < |𝒜.transitions(q)| then
            edgeType ← 𝒜.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    return null
```

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

```
function Begin(G, query)
    𝒜 ← Automaton(regex)                  ▷ q0 initial state, F final states
    Open.init()
    Visited.init()
    ReachedFinal.init()
    startState ← (v, q0, 0, ⊥)
    Visited.push(startState)
    Open.push(startState)
    iter ← null                           ▷ Index iterator for neighbors
    ReachedSolutions.init()               ▷ Queue of current solutions
    firstNext ← True                      ▷ First time calling Next

function Next(G, query)
    while ReachedSolutions ≠ ∅ do         ▷ Enumerate solutions
        return ReachedSolutions.pop()
    if firstNext then                     ▷ Check if initial state is solution
        firstNext ← False
        if v ∈ V and q0 ∈ F then
            ReachedFinal.add(⟨v, 0⟩)
            return v
    while Open ≠ ∅ do                     ▷ current = (n, q, depth, prevList)
        current ← Open.front()
        reached ← expandAllShortest(current)
        if reached ≠ null then            ▷ New solutions found
            ReachedSolutions ← getNewPaths(reached)
            return ReachedSolutions.pop()
        else                              ▷ State was fully expanded
            iter ← null
            Open.pop()
    return null                           ▷ No more solutions
```

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

```
function expandAllShortest(state = (n, q, depth, prevList))
    if iter == null then                          ▷ First time state is explored
        if |A.transitions(q)| == 0 then
            return null
        else                                      ▷ Set the index iterator
            transitionIdx ← 0
            edgeType ← A.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    while transitionIdx < |A.transitions(q)| do
        transition ← A.transitions(q)[transitionIdx]
        q′ ← transition.to                        ▷ Next automaton state
        while iter.next() ≠ null do               ▷ iter = (n′, edge′)
            if (n′, q′, ∗, ∗) ∈ Visited then
                (n′, q′, depth′, prevList′) ← Visited.get(n′, q′)
                if depth + 1 == depth′ then
                    prevList′.add(⟨ state, edge′⟩)
                    if q′ ∈ F then
                        shortest ← ReachedFinal.get(n′).depth
                        if depth + 1 == shortest then
                            return (n′, q′, depth′, prevList′)
            else
                prevList.init()
                prevList.add(⟨ state, edge′⟩)
                newState ← (n′, q′, depth + 1, prevList)
                Visited.push(newState)
                Open.push(newState)
                if q′ ∈ F then
                    if n′ ∉ ReachedFinal then
                        ReachedFinal.add(⟨n′, depth + 1⟩)
                        return newState
                    else
                        shortest ← ReachedFinal.get(n′).depth
                        if depth + 1 == shortest then
                            return newState
        transitionIdx++                           ▷ Next transition
        if transitionIdx < |A.transitions(q)| then
            edgeType ← A.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    return null

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

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

```
function Begin(G, query)
    A ← Automaton(regex)              ▷ q0 initial state, F final states
    Open.init()
    Visited.init()
    startState ← (v, q0, null, ⊥)
    Visited.push(startState)
    Open.push(startState)
    iter ← null                       ▷ Index iterator for neighbors
    firstNext ← True                  ▷ First time calling Next

function Next(G, query)
    if firstNext then                 ▷ Check if initial state is solution
        firstNext ← False
        if v ∈ V and q0 ∈ F then
            return v
    while Open ≠ ∅ do
        current ← Open.front()        ▷ current = (n, q, edge, prev)
        reached ← expandAll(current)
        if reached ≠ null then        ▷ New solution found
            return getPath(reached, [])
        else                          ▷ State was fully expanded
            iter ← null
            Open.pop()
    return null                       ▷ No more solutions

function expandAll(state = (n, q, edge, prev))
    if iter == null then              ▷ First time state is explored
        if |A.transitions(q)| == 0 then
            return null
        else                          ▷ Set the index iterator
            transitionIdx ← 0
            edgeType ← A.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    while transitionIdx < |A.transitions(q)| do
        transition ← A.transitions(q)[transitionIdx]
        q′ ← transition.to            ▷ Next automaton state
        while iter.next() ≠ null do   ▷ iter = (n′, edge′)
            if isValid(state, iter, restrictor) then
                new ← (n′, q′, edge′, state)
                Visited.push(new)
                Open.push(new)
                if q′ ∈ F then
                    return new
        transitionIdx++               ▷ Next transition
        if transitionIdx < |A.transitions(q)| then
            edgeType ← A.transitions(q)[transitionIdx].type
            iter ← Search(n,edgeType,G)
    return null
```