Evaluation and Enumeration Problems for Regular Path Queries

Wim Martens
University of Bayreuth, Bayreuth, Germany
wim.martens@uni-bayreuth.de

Tina Trautner
University of Bayreuth, Bayreuth, Germany
tina.trautner@uni-bayreuth.de

Abstract

Regular path queries (RPQs) are a central component of graph databases. We investigate decision- and enumeration problems concerning the evaluation of RPQs under several semantics that have recently been considered: *arbitrary paths*, *shortest paths*, and *simple paths*.

Whereas arbitrary and shortest paths can be enumerated in polynomial delay, the situation is much more intricate for simple paths. For instance, already the question if a given graph contains a simple path of a certain length has cases with highly non-trivial solutions and cases that are long-standing open problems. We study RPQ evaluation for simple paths from a parameterized complexity perspective and define a class of *simple transitive expressions* that is prominent in practice and for which we can prove a dichotomy for the evaluation problem. We observe that, even though simple path semantics is intractable for RPQs in general, it is feasible for the vast majority of RPQs that are used in practice. At the heart of our study on simple paths is a result of independent interest: the two disjoint paths problem in directed graphs is W[1]-hard if parameterized by the length of one of the two paths.

2012 ACM Subject Classification Information systems → Query languages for non-relational engines, Theory of computation → Database query languages (principles), Theory of computation → Regular languages

Keywords and phrases Graph databases, regular path queries, regular languages, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.19

Acknowledgements We are grateful to Phokion Kolaitis for suggesting us to study enumeration problems on simple paths matching RPQs. We are also grateful to Holger Dell for pointing us to Theorem 9 and providing us with a proof sketch. We acknowledge the many useful comments of the anonymous reviewers for ICDT 2018 that helped us to significantly improve the structure and presentation of the paper.

1 Introduction

Regular path queries (RPQs) are a crucial feature of graph database query languages, since they allow us to pose queries about arbitrarily long paths in graphs. Essentially, RPQs are regular expressions that are matched against labeled directed paths in graph databases. Currently, the openCypher project [33] and the World Wide Web Consortium (W3C) [39] are considering how RPQ evaluation can be formally defined for the development of Neo4j's Cypher [31, 34] and SPARQL 1.1 [38], respectively. Several popular candidates that are being considered for the semantics of RPQs are *arbitrary paths*, *shortest paths*, and *simple paths* ([3, Section 4.4], [34]).

We briefly explain these semantics. Given a graph, an RPQ r considers directed paths for which the labels on the edges form a word in the language of r. We call such paths *candidate matches*. The different semantics restrict the kind of paths that *match* the RPQ, i.e., can be returned as answers. *Arbitrary paths* imposes no restriction and returns every candidate match. *Shortest paths*, on the other hand, only returns the shortest candidate matches and *simple paths* only returns candidate matches that do not have duplicate nodes.

Under *arbitrary paths*, the number of matches may be infinite if the graph is cyclic. This may pose a challenge for designing the query language, even if one does not choose to return all matching paths. Indeed, a well-known semantics of RPQs is to return *node pairs* (x, y) such that there exists a matching path from x to y. Under bag semantics for node pairs,[1] where each (x, y) is returned as often as the number of matches from x to y, one needs to deal with the case where this number is infinite.

Under *shortest paths* and *simple paths*, the number of matching paths is always finite, which simplifies the aforementioned design challenge. However, these two versions face other challenges. *Simple paths* may present complexity issues. Two fundamental problems are that

- counting the number of simple paths between two nodes is #P-complete [37] and
- deciding if there exists a simple path of even length between two given nodes is NP-complete [23].

Indeed, the first problem implies that evaluating the RPQ a^* under bag semantics is #P-complete and the second one implies that deciding if the RPQ $(aa)^*$ returns at least one answer is NP-complete.[2] *Shortest paths* does not have these complexity issues, but it is unclear if its semantics is very natural. For instance, under shortest paths semantics, if we ask how many paths exist from x to y, then this number may decrease if a new, shorter, path is added.[3] This may seem counter-intuitive to users.

Since it seems that there is no one-size-fits-all solution, the openCypher project team recently proposed to support several kinds of semantics for Cypher [34]. This situation motivated us to shed more light on evaluation of RPQs and enumerating the answers, focusing on the following aspects:

- Our goal is to better understand *enumerating the paths* that match RPQs. That is, we study problems where the task is to enumerate all matching paths without duplicates. We are interested in which situations it is possible to answer queries in *polynomial delay*, i.e., such that the time between consecutive answers is polynomial. To reach this goal, we must also improve our understanding for some decision problems related to RPQ evaluation.
- We take into account a recent study that investigated the structure of about 250K RPQs gathered from a wide range of SPARQL query logs [8]. It turns out that all these RPQs have a relatively simple structure, which is remarkable because their syntax is not restricted by the SPARQL recommendation.

Our contributions are the following.

1. We first observe that enumeration of arbitrary or shortest paths that match a given RPQ can be done in polynomial delay (Section 3).
2. We then turn to simple paths and study RPQ evaluation as a decision problem. This problem is challenging because it contains subproblems that are quite non-trivial. One

[1] SPARQL 1.1 uses such a bag semantics approach.

[2] It is also known that answering the RPQ a^*ba^* under simple path semantics is at least as difficult as the Two Disjoint Paths problem [29].

[3] Notice that each semantics only returns or counts the number of paths that match.

such subproblem is testing if there exists a directed simple path of length $\log n$ between two given nodes in a graph with n nodes, which was shown to be in PTIME by Alon et al., using their color coding technique [2]. The question if it can be decided in PTIME if there is a simple path of length $\log^2 n$ [2] is an open problem since two decades. Notice that these two problems are special cases of RPQ evaluation under simple path semantics (i.e., evaluate the RPQs $a^{\log n}$ and $a^{\log^2 n}$ in a graph where every edge has label a).

We therefore investigate RPQ evaluation from the angle of parameterized complexity (Section 4). We introduce the class of *simple transitive expressions (STEs)* that capture over 99% of the RPQs that were found in SPARQL query logs in a recent study [8]. We identify a property of STEs that we call *cuttability* and prove a dichotomy, showing that the parameterized complexity for evaluating STEs $\mathcal{R}$ is in FPT if $\mathcal{R}$ is cuttable and W[1]-hard otherwise. Examples of cuttable classes of expressions are $\{a^k a^* \mid k \in \mathbb{N}\}$ and $\{(a+b)^k a^* \mid k \in \mathbb{N}\}$. Examples of non-cuttable classes are $\{a^k b^* \mid k \in \mathbb{N}\}$, $\{a^k b a^* \mid k \in \mathbb{N}\}$, and $\{a^k (a+b)^* \mid k \in \mathbb{N}\}$.

3. At the core of the dichotomy are two results of independent interest (Section 5). The first is by the authors of [16], who showed that it can be decided in FPT if there is a simple path of length *at least* k between two nodes in a graph (Theorem 9). The second shows that the Two Disjoint Paths problem is W[1]-hard when parameterized by the length of one of the two paths (Theorem 11).
4. We then turn to enumeration of simple paths and prove that the dichotomy on STEs carries over to the enumeration setting. We also study the data complexity and show that Bagan et al.'s dichotomy for deciding the existence of a simple path that matches an RPQ [5] carries over to enumeration problems (Section 6).

Putting everything together, we see that, although simple path semantics leads to high complexity in general, its complexity for RPQs that have been found in SPARQL query logs is reasonable. We discuss this in the conclusions.

Related Work

RPQs on graph databases have been studied since the end of the 80's [10, 11, 40]. Given a graph database G, an RPQ r, and two nodes s and t, there are several natural fundamental problems associated to RPQ evaluation.

- The *decision problem*: Does r match a path from s to t in G?
- The *counting problem*: How many paths from s to t does r match?
- The *computation problem*: Compute the set of paths from s to t for which r matches.

The decision problem is well known to be tractable for arbitrary and shortest paths by standard automata techniques. Mendelzon and Wood [29] studied the problem for simple paths. They observed that the problem is NP-complete for a^*ba^* and $(aa)^*$. These two results heavily rely on the work of Fortune et al. [17], who showed NP-completeness of the two disjoint paths problem, and Lapaugh and Papadimitriou [23], who showed that the even length simple path problem is NP-complete.

Bagan et al. [5] provided a dichotomy for the *data complexity* of the decision problem. They defined a class $\mathsf{C}_{\text{tract}}$ such that the problem is in PTIME for each language in $\mathsf{C}_{\text{tract}}$ and NP-complete otherwise.

The *counting problem* for arbitrary paths is #P-complete in general [21].[4] However, if the RPQ is represented by a deterministic automaton (or even an unambiguous one), the counting problem is in PTIME [26], since it reduces to counting the number of paths in a graph. The complexity results for arbitrary paths can easily be extended to shortest paths. Indeed, all words have equal length in Kannan et al.'s #P-hardness proof [21] and the PTIME algorithm also works if we need to count the words of a given length n.

Concerning simple paths, we know from the classical result of Valiant [37] that counting the number of simple paths between two given nodes in a graph is #P-complete. This immediately implies that counting is already #P-hard for the RPQ a^*.

Concerning the *computation problem*, Ackermann and Shallit [1] proved that one can enumerate the words accepted by a given NFA in polynomial delay. This is easily extended to RPQ evaluation w.r.t. arbitrary paths and shortest paths, as we observe in Section 3. Concerning simple paths, Yen's algorithm [41] is a method to enumerate all simple paths between two given nodes in polynomial delay. We build on this result in Section 6.

Yen's algorithm was generalized by Lawler [24] and Murty [30] to a tool for designing general algorithms for enumeration problems. Lawler-Murty's procedure has been used for solving enumeration problems in databases in various contexts [18, 20, 22].

Further related work concerning RPQs on graph databases are studies about the complexity of SPARQL 1.1 property paths [4, 26], which are relevant because property paths extend RPQs. Their semantics is a mixture between arbitrary path and simple path semantics. The relative expressive power of graph query languages using transitive closures, data value comparisons, and branching was investigated in [25, 36]. Finally, we refer to [3, 6] for general overviews of the wide literature on graph databases.

2 Preliminaries

By Σ we always denote an *alphabet*, that is, a finite set. A (Σ-)*symbol* is an element of Σ. A *word* (over Σ) is a finite sequence $w = a_1 \cdots a_n$ of Σ-symbols. The *length* of w, denoted by $|w|$, is its number of symbols n. We denote the empty word by ε.

We assume familiarity with regular expressions and finite automata. The regular expressions we use in this paper are defined as follows: $\emptyset$, ε and every Σ-symbol is a regular expression; and when r and s are regular expressions, then (rs), $(r + s)$, $(r?)$, (r^*), and (r^+) are also regular expressions. From now on, we use the usual precedence rules to omit parentheses. The *size* $|r|$ of a regular expression is the number of occurrences of Σ-symbols in r. For example, $|aba^*| = 3$. We define the *language* $L(r)$ of r as usual. Since it is easy to test if $L(r) = \emptyset$ for a given expression r, we assume in this paper that $L(r) \neq \emptyset$ for all expressions, unless mentioned otherwise. For $n \in \mathbb{N}$, we use r^n to abbreviate the n-fold concatenation $r \cdots r$ of r. We abbreviate $(r?)^n$ by $r^{\leq n}$. In the context of graph databases, *regular path queries (RPQs)* are regular expressions that can be evaluated on graphs and return an output. In this paper, we will blur the distinction between them (language acceptors vs. queries) and use "regular expression" and RPQ as synonyms.

A *non-deterministic finite automaton (NFA)* N over Σ is a tuple $(Q, \Sigma, \Delta, Q_I, Q_F)$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta : Q \times \Sigma \times Q$ is the transition relation, $Q_I \subseteq Q$ is the set of initial states, and Q_F is the set of accepting states. By $\delta^*(w)$ we denote the set of states reachable by N after reading w, that is, $\delta^*(\varepsilon) = Q_I$ and, for every word w

[4] Kannan et al. proved that counting the number of words accepted by a non-deterministic automaton for a finite language is #P-complete. This result trivially extends to RPQ evaluation.

and symbol a, we define $\delta^*(wa) = \{\delta(q, a) \mid q \in \delta^*(w)\}$. The *size* of an NFA is $|Q|$, i.e., its number of states. We define the *language* $L(N)$ of N as usual.

2.1 Graph Databases

We use edge-labeled directed graphs as abstractions for graph databases. A graph G (with labels in Σ) will be denoted as $G = (V, E)$, where V is the finite set of *nodes* of G and $E \subseteq V \times \Sigma \times V$ is the set of *edges*. We say that edge $e = (u, a, v)$ goes *from node u to node v* and *has label a*. We use *a-edge* to refer to an edge with label a. Sometimes we write an edge as $(u, v) \in V \times V$ if the label does not matter. In this paper, we assume that graphs are directed, unless mentioned otherwise. Notice that our definition allows graphs to have self-loops and multi-edges. The *size* of a graph G, denoted by $|G|$ is $|V| + |E|$.

We assume familiarity with basic terminology on graphs. A *path* from node u to node v in G is a sequence $p = (v_0, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{n-1}, a_n, v_n)$ of edges in G such that $u = v_0$ and $v = v_n$. For $0 \leq i \leq n$, we denote by $p[i, i]$ (or $p[i]$) the node v_i and, for $0 \leq i < j \leq n$, we denote by $p[i, j]$ the subpath $(v_i, a_{i+1}, v_{i+1}) \dots (v_{j-1}, a_j, v_j)$. A path p is *simple* if all nodes $v_0, \dots, v_n$ are pairwise different.[5] The *length* of p, denoted $|p|$, is the number n of edges in p. By definition of paths, we consider two paths to be different if they are different sequences of edges. In particular, two paths going through the same nodes in the same order, but using different edge labels are different.

The set of *nodes of path p* is $V(p) = \{v_0, \dots, v_n\}$. The *word of p* is $a_1 \cdots a_n$ and is denoted by $\text{lab}(p)$. Path p *matches* a regular expression r (resp., NFA N) if $\text{lab}(p) \in L(r)$ (resp., $\text{lab}(p) \in L(N)$). The *concatenation* of paths $p_1 = (v_0, a_1, v_1) \cdots (v_{n-1}, a_n, v_n)$ and $p_2 = (v_n, a_{n+1}, v_{n+1}) \cdots (v_{n+m-1}, a_{n+m}, v_{n+m})$ is simply the concatenation $p_1 p_2$ of the two sequences.

We will often consider a graph $G = (V, E)$ together with a *source node s* and a *target node t*, for example, when considering paths from s to t. We denote such a graph with source s and target t as (G, s, t) and define its size $|(G, s, t)|$ as $|G|$.

The *product* of graph (G, s, t) and NFA $N = (Q, \Sigma, \Delta, Q_I, Q_F)$ is a graph (V', E') with $V' = (V \times Q)$ and $E' = \{((u_1, q_1), a, (u_2, q_2)) \mid (u_1, a, u_2) \in E \text{ and } (q_1, a, q_2) \in \Delta\}$. We denote this product by $(G, s, t) \times N$. Notice that simple paths in $(G, s, t) \times N$ may use nodes $(u, q_1) \neq (u, q_2)$ and may therefore correspond to non-simple paths in G.

2.2 Decision and Enumeration Problems

We consider the following problems, where G is always a graph, s and t are nodes in G, and r is an RPQ.

- Path: Given (G, s, t) and r, is there a path from s to t that matches r?
- SimPath: Given (G, s, t) and r, is there a simple path from s to t that matches r?

An *enumeration problem* P is a (partial) function that maps each input i to a finite or countably infinite set of *outputs for i*, denoted by $\mathsf{P}(i)$. Terminologically, we say that, given i, the task is to *enumerate* $\mathsf{P}(i)$. We consider the following enumeration problems:

- EnumPaths: Given (G, s, t) and r, enumerate the paths in G from s to t that match r.
- EnumShortPaths: Given (G, s, t) and r, enumerate the shortest paths in G from s to t that match r.

[5] We focus on *node-distinct paths* in this paper, but one can also consider *edge-distinct paths*. We come back to this in the conclusions.

- EnumSimPaths: Given (G, s, t) and r, enumerate the simple paths in G from s to t that match r.

An *enumeration algorithm* for P is an algorithm that, given input i, writes a sequence of answers to the output such that every answer in $\mathsf{P}(i)$ is written precisely once. If A is an enumeration algorithm for an enumeration problem P, we say that A runs in *polynomial delay* if the time before writing the first answer and the time between writing every two consecutive answers is polynomial in $|i|$.

For a class $\mathcal{R}$ of regular expressions, we denote by $\mathsf{Path}(\mathcal{R})$ the problem Path where we always assume that $r \in \mathcal{R}$. We use the same convention for all other decision- and enumeration problems. We assume familiarity with the notions *combined* and *data* complexity. In our decision problems, (G, s, t) is the data and r is the query.

3 Enumerating All Regular Paths and Shortest Regular Paths

It is well known that $\mathsf{Path}(\mathcal{R})$ is in PTIME for the complete class $\mathcal{R}$ of RPQs. Indeed, one only needs to construct the product of the graph and an NFA N for the RPQ and test if (t, q_f) is reachable from (s, q_0), where q_0 and q_f are an initial and an accepting state of N, respectively. We note that this favorable complexity carries over to EnumPaths and EnumShortPaths. At the core lies the following result by Ackerman and Shallit.

▶ **Theorem 1** (Theorem 3 in [1]). *Given an NFA N, enumerating the words in $L(N)$ can be done in polynomial delay.*

This result generalizes a result of Mäkinen [27], who proved that the words in $L(N)$ can be enumerated in polynomial delay if N is deterministic. Ackermann and Shallit genereralized his algorithm for nondeterministic N and proved that, for a given length n (which they call *cross-section*), the lexicographically smallest word in $L(N)$ can be found in time $O(|Q|^2 n^2)$ ([1], Theorem 1). They then prove that the set of all words of length n can be computed in time $O(|Q|^2 n^2 + |\Sigma||Q|^2 x)$, where x is the sum of the lengths of the words that were written to the output ([1], Theorem 2). A closer inspection of their algorithm actually shows that it has delay $O(|\Sigma||Q|^2|w|)$ where $|w|$ is the size of the next output. In fact, Ackermann and Shallit prove that the words in $L(N)$ can be enumerated in *radix order*.[6]

It is easy to extend the algorithm of Ackerman and Shallit to solve EnumPaths in polynomial delay as follows. We construct an NFA N_r for r and take the product with (G, s, t). The product automaton therefore has states (u, q) where u is a node from G and q a state from N_r. In the resulting automaton, we replace every transition $[(u_1, q_1), a, (u_2, q_2)]$ with $[(u_1, q_1), (u_1, a, u_2), (u_2, q_2)]$. Enumerating the words from the resulting automaton corresponds to enumerating the paths from s to t that match r. Using Theorem 1, we have the following corollary.

▶ **Corollary 2.** *EnumPaths and EnumShortPaths can be solved in polynomial delay.*

For completeness, we note that counting the number of paths from s to t that match a given regular expression r is #P-complete in general, even if G is acyclic, see [26, Theorem 4.8(1)] and [4, Theorem 6.1].[7] The same holds for counting the number of shortest paths, since all paths in the proof of [26, Theorem 4.8(1)] have equal length.

---

[6] That is, $w_1 < w_2$ in radix order if $|w_1| < |w_2|$ or $|w_1| = |w_2|$ and w_1 is lexicographically before w_2.

[7] Arenas et al. [4] actually prove that the problem is SPANL-complete. Although it is not known if SPANL = #P, they are equal under Cook reductions.

4 Deciding Existence of Simple Paths

We now turn to *simple paths*, which will require much more effort. First, we focus on the decision problem SimPath, where our main result will be a dichotomy for *simple transitive expressions (STEs)*, a very restricted class of RPQs.

We will investigate SimPath from a parametrized complexity perspective. The main reason is that the size of the regular expression has a drastic effect on the complexity of the problem. Indeed, if G is a graph with n nodes and only a-edges, then asking if there is a simple path that matches the expression a^{n-1} is the NP-complete Hamilton Path problem. On the other hand, Alon et al. [2] proved that SimPath for graphs with n nodes is in PTIME for the language $a^{\log n}$. It is open[8] since 1995 whether SimPath is in PTIME for $a^{\log^2 n}$ [2].

So, even very elementary RPQs of the form a^k can behave very differently depending on the relationship between k and the size of the graph. This motivates us to study the problem from the angle of parameterized complexity.

4.1 Parameterized Complexity

We first give a quick overview of some notions in parameterized complexity. We follow the exposition of Cygan et al. [12] and refer to their work for further details. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ where, as before, Σ is a fixed, finite alphabet. For an instance $(x, p) \in \Sigma^* \times \mathbb{N}$, we call p the *parameter*. The *size* $|(x, p)|$ of an instance (x, p) is defined as $|x| + p$. A parameterized problem L is called *fixed-parameter tractable* if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, p) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, p) \in L$ in time bounded by $f(p) \cdot |(x, p)|^c$. The complexity class containing exactly the fixed-parameter tractable problems is called FPT.

In the remainder of this section, we will study the parameterized complexity of SimPath. The instances (x, p) of this problem will always be such that x encodes the graph G and regular expression r, and the parameter p is $|r|$. For this reason, we overload notation and also denote the parameterized problem as SimPath.

4.2 Some Illustrations of the Dichotomy

Before we present our main dichotomy, we illustrate a few of its implications to give the reader some intuition about the result. In the following, we abbreviate the class of regular expressions $\{a^k \mid k \in \mathbb{N}\}$ simply by "a^k", and similar for $a^{\leq k}$, $a^k a^*$, $a^k b^*$, $a^k b a^*$, $b a^k a^*$, etc.

In the following Theorem, we consider problems SimPath($\mathcal{R}$), where $\mathcal{R}$ is one of the abovementioned classes.

▶ **Theorem 3.**
(a) *SimPath*(a^k), *SimPath*($a^{\leq k}$), *SimPath*($a^k a^*$), *and* *SimPath*($b a^k a^*$) *are in FPT.*
(b) *SimPath*($b^k a^*$) *and* *SimPath*($a^k b a^*$) *are W[1]-hard.*

[8] Recently, Björklund et al. [7] showed that, under the Exponential Time Hypothesis, there is no PTIME algorithm that can decide if there exists a simple path of length $\Omega(f(n) \log^2 n)$ between two nodes in a graph of size n for any nondecreasing polynomial time computable function f that tends to infinity.

Table 1 Structure of the 247,404 SPARQL property paths that were also used in the query logs investigated by Bonifati et al. [8]. The structure is sometimes in terms of a variable $\ell \in \mathbb{N}$, for which the second column indicated the values that were found in the logs. *Relative* indicates which percentage of the 247,404 property paths have this structure.

Expression Type	ℓ	*Relative*	*STE?*	*Expression Type*	ℓ	*Relative*	*STE?*
$(a_1 + \cdots + a_\ell)^*$	2–4	29.10%	yes	$a^*b?$		$< 0.01\%$	yes
$.$		25.48%	yes$^{(*)}$	abc^*		$< 0.01\%$	yes
a^*		19.66%	yes	$A_1 \cdots A_\ell$	2,6	$< 0.01\%$	yes
$a_1 \cdots a_\ell$	2–6	8.66%	yes	$.^*$		$< 0.01\%$	yes$^{(*)}$
a^*b		7.73%	yes	$(a_1 + a_2)?$		$< 0.01\%$	yes
$(a_1 + \cdots + a_\ell)$	1–6	6.61%	yes	$.?$		$< 0.01\%$	yes$^{(*)}$
$(a_1 + \cdots + a_\ell)^+$	1,2	1.54%	yes	$a^* + b$		$< 0.01\%$	no
$a_1?a_2? \cdots a_\ell?$	1–3,5	1.15%	yes	$a + b^+$		$< 0.01\%$	no
$a(b_1 + b_2)?$		0.01%	yes	$a^+ + b^+$		$< 0.01\%$	no
$a_1a_2? \cdots a_\ell?$	2,3	0.01%	yes	$(ab)^*$		$< 0.01\%$	no
$(ab^*) + c$		$< 0.01\%$	no				

We see that, even though all classes of regular expressions in Theorem 3 are similar, the complexities are drastically different (assuming FPT $\neq$ W[1]). The intuition is twofold:

1. "Short" paths can be dealt with using Color Coding [2] (or Bagan et al.'s extension thereof that incorporates finite regular languages [5, Theorem 6]). This explains why $\mathsf{SimPath}(a^k)$ and $\mathsf{SimPath}(a^{\leq k})$ are in FPT.
2. If paths can become arbitrarily long, the complexity depends on the interplay between the symbol in the *transitive closure* (which is always a here) and the rest. The intuition is that symbols that are "incompatible" with a (which is always b here) should only occur on positions that are a constant distance away from the beginning or end of words in the language. This explains why SimPath is in FPT for the classes a^ka^* (no incompatible symbols) and ba^ka^* (b is always on position one). Likewise, for the classes b^ka^* and a^kba^*, the symbol b can occur at positions arbitrarily far away from the beginning and end of words in the languages.

4.3 Dichotomy for Simple Transitive Expressions

We now aim at generalizing the results in Theorem 3 to more general RPQs which we call *simple transitive expressions (STEs)*. Although STEs are very restricted, we feel that they are relevant and important from a practical perspective since they constitute more than 99.99% of the *SPARQL property paths* found in query logs in an extensive recent study [8]. Notice that SPARQL property paths strictly extend RPQs. Their syntax is not restricted to a subset of regular expressions as, e.g., in Cypher patterns for "variable length pattern matching" [32, Section 3.2.7.7].

In the following definition, we use sets $A = \{a_1, \ldots, a_n\} \subseteq \Sigma$ to abbreviate expressions $(a_1 + \cdots + a_n)$. We allow $A = \emptyset$, in which case $L(A) = \emptyset$.

▶ **Definition 4.** An *atomic expression* is of the form $A \subseteq \Sigma$. A *bounded expression* is a regular expression of the form $A_1 \cdots A_k$ or $A_1? \cdots A_k?$, where $k \geq 0$ and each A_i is an atomic expression. Finally, a *simple transitive expression (STE)* is a regular expression

$$B_{\text{pre}} T^* B_{\text{suff}},$$

where B_{pre} and B_{suff} are bounded expressions and T is an atomic expression.

The central idea for STEs is that they can first perform some local navigation in B_{pre}, then an optional transitive part, followed by a second step of local navigation in B_{suff}. The local navigation steps allow to test paths of length exactly k or at most k, for some $k \in \mathbb{N}$. The transitive part is optional, since one can take $T = \emptyset$, so that T^* only matches ε.

We believe that STEs capture many RPQs that users ask in practice. Bonifati et al. [8] investigated the structure of 247,404 SPARQL property paths from query logs. Table 1 presents a classification of their raw data that facilitates comparison to RPQs. SPARQL property paths can express wildcard tests, which we denote by "." (similar to regexes). Furthermore, SPARQL uses reverse edges ("$\hat{}a$" means "follow an a-edge in reverse direction"), which we treat the same as a normal label test. Under *Expression Type*, the table summarizes which types of expressions are in Bonifati et al.'s data set, sometimes parameterized by a number ℓ for which the next column describes the values that were found. *Relative* describes which percentage of the 247,404 expressions fall into this expression type, and *STE?* indicates whether the expression is an STE. Here, we write "yes$^{(*)}$" to indicate that the expression is an STE if a wildcard is treated the same as a set of labels A. (Our algorithms indeed can be generalized to incorporate wildcards.)

In total, we saw that only 20 property paths are not STEs or trivially equivalent to an STE (by taking $T = \emptyset$ in the definition of STEs, for example).[9] For instance, the expression type $a_1 a_2? \cdots a_\ell?$ is equivalent to an STE where $B_{\text{pre}} = a_1$, $T = \emptyset$, and $B_{\text{suff}} = a_2? \cdots a_\ell?$. In summary, 99.992% of the property paths in Table 1 correspond to STEs.

We now define the notions that we need for the dichotomy.

▶ **Definition 5.** Let $r = B_{\text{pre}} T^* B_{\text{suff}}$ be an STE with $L(r) \neq \emptyset$. If $B_{\text{pre}} = A_1 \cdots A_{k_1}$, then the *left cut border* c_1 *of* r is the largest value such that $T \not\subseteq A_{c_1}$ if it exists and zero otherwise. If $B_{\text{pre}} = A_1? \cdots A_{k_1}?$, then the left cut border is zero. Symmetrically, if $B_{\text{suff}} = A'_{k_2} \cdots A'_1$, then the *right cut border* c_2 *of* r is the largest value such that $T \not\subseteq A'_{c_2}$ if it exists and zero otherwise. (Notice that the indices in B_{suff} are reversed.) If $B_{\text{suff}} = A'_{k_2}? \cdots A'_1?$, then the right cut border is zero.

We explain the intuition behind cut borders in Figure 1. For $c \in \mathbb{N}$, an expression is *c-bordered* if the maximum of its left and right cut borders is c. We call a class $\mathcal{R}$ of STEs *cuttable* if there exists a constant $c \in \mathbb{N}$ such that each expression in $\mathcal{R}$ is c'-bordered for some $c' \leq c$.

We can now prove a dichotomy on the complexity of $\mathsf{SimPath}(\mathcal{R})$ for classes of STEs $\mathcal{R}$, if $\mathcal{R}$ satisfies the following mild condition. We say that $\mathcal{R}$ *can be sampled* if there exists an algorithm that, given $k \in \mathbb{N}$, returns an expression in $\mathcal{R}$ that is k'-bordered with $k' \geq k$, and "no" if there is no such expression. We need the condition that $\mathcal{R}$ can be sampled to prove the W[1]-hardness. For this reason, this condition is no longer needed in Theorem 15.

▶ **Theorem 6.** *Let $\mathcal{R}$ be a class of STEs that can be sampled.*
(a) *If $\mathcal{R}$ is cuttable, then $\mathsf{SimPath}(\mathcal{R})$ is in FPT and*
(b) *otherwise, $\mathsf{SimPath}(\mathcal{R})$ is W[1]-hard.*

Proof idea. The main techniques will be presented in Section 5. We can attack case (a) using Theorem 7, Observation 8, and the techniques for proving Theorem 9. In short, if $\mathcal{R}$ is cuttable and we need to deal with arbitrarily long paths, then we can use exhaustive search to enumerate all possible pre- and suffixes of length at most c. We then use a variation of the representative sets technique [16] to obtain an FPT algorithm. In case (b), it is possible to adapt the reduction in the proof of Theorem 11. ◀

[9] In fact, *all* expressions except for $(ab)^*$ can be handled with the techniques we present here. For instance, the FPT algorithm can trivially be extended to unions of STEs by testing each STE separately. For the expression $(ab)^*$, even the data complexity of SimPath is NP-complete.

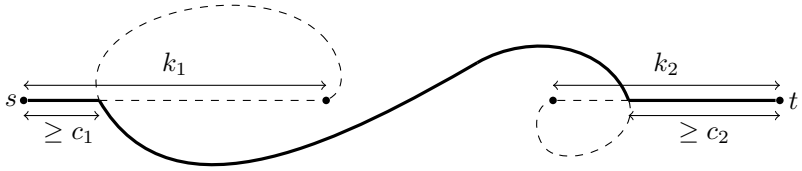

Figure 1 Assume $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ has left and right cut borders c_1 and c_2, respectively. Assume that an arbitrary path from s to t matches r such that its length k_1 prefix and length k_2 suffix are node-disjoint. If, after removing all loops, (1) the length c_1 prefix and length c_2 suffix are still the same and (2) the path still has length at least $k_1 + k_2$, then it matches r.

Notice that the difference between cuttable and non-cuttable classes of STEs can be quite subtle. For instance, $b^k a^*$ and $a^k(a+b)^*$ are non-cuttable, but $(a+b)^k a^*$ is cuttable. Looking back at Table 1, we see that abc^* is 2-bordered and all other STEs are either 0-bordered or 1-bordered. It therefore seems that cut borders in practice are small and over 99% of the expressions fall on the tractable side of Theorem 6.

5 Technical Core: Simple Paths With Length Constraints

In this section we investigate the parameterized complexity of problems that involve simple paths with length constraints. The problems we consider here are the core of the RPQ evaluation problems in Section 4.

5.1 One Path

We consider the following parameterized problems.

- PSimPath: Given an instance $((G, s, t), k)$ with parameter $k \in \mathbb{N}$, is there a simple path from s to t of length exactly k in G?
- PSimPath$^{\leq}$ and PSimPath$^{\geq}$: These two problems are defined analogously to PSimPath but ask if there is a simple path of length at most k and at least k, respectively.

These three problems are in FPT, but the techniques to obtain these results are quite different. For PSimPath, membership in FPT follows from the famous color coding technique [2].

▶ **Theorem 7** (Alon et al. [2]). *PSimPath is in FPT.*

PSimPath$^{\leq}$ is trivially in FPT because the shortest path problem is in PTIME.

▶ **Observation 8.** *PSimPath$^{\leq}$ is in PTIME (and therefore in FPT).*

Finally, PSimPath$^{\geq}$ can be shown to be in FPT by adapting methods from Fomin et al. [16]. They proved that finding simple cycles of length at least k is in FPT for cycles and discovered that their technique also works for paths [13]. The following theorem is therefore due to the authors of [16]. (Fomin et al. [16] already showed FPT membership for PSimPath$^{\geq}$ on *undirected* graphs, but the techniques needed on directed graphs are quite different.)

▶ **Theorem 9.** *(Similar to Theorem 5.3 in [16]) PSimPath$^{\geq}$ is in FPT.*

5.2 Two Disjoint Paths

We consider variants of the TwoDisjointPaths problem [17]. A *two-colored graph* is a directed graph in which every edge is labeled a or b. An *a-path* is a path consisting of only a-edges. We consider the following parameterized problems.

- PTwoDisjointPaths: Given a graph G, nodes s_1, t_1, s_2, t_2, and parameter $k \in \mathbb{N}$, are there simple paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 such that p_1 and p_2 are node-disjoint and p_1 has length k?
- PTwoColorDisjointPaths: Given a two-colored graph G, nodes s_a, t_a, s_b, t_b, and parameter $k \in \mathbb{N}$, is there a simple a-path p_a from s_a to t_a and a simple b-path p_b from s_b to t_b such that p_a and p_b are node-disjoint and p_a has length k?

It is well-known that TwoDisjointPaths, the non-parameterized version of PTwoDisjointPaths, is NP-complete [17]. In terms of parameterized complexity, Downey and Fellows [14] introduced the W-hierarchy, where FPT = W[0] and W[i] $\subseteq$ W[j] for all $i \leq j$. A famous complete problem for W[1] (under so-called *fpt-reductions*) is k-Clique with parameter k [15]. Therefore, k-Clique not being fixed-parameter tractable is equivalent to FPT $\neq$ W[1], which is a standard assumption in parameterized complexity.

Cai and Ye [9] proved that PTwoDisjointPaths is in FPT for *undirected graphs*, both for the cases where one wants node-disjoint or edge-disjoint paths. They left the cases for directed graphs as open problems [9, Problem 2]. We solve one of the cases by showing in Theorem 11 that PTwoDisjointPaths is W[1]-hard. We also prove that PTwoColorDisjointPaths is W[1]-hard – the proof for PTwoDisjointPaths relies on it.

▶ **Theorem 10.** *PTwoColorDisjointPaths is W[1]-hard.*

Proof idea. This result follows from a slight adaptation of a proof of Slivkins [35, Theorem 2.1]. Slivkins proved that k-Edge-Disjoint-Paths with parameter k is W[1]-hard in directed acyclic graphs. More precisely, given an instance of k-Clique, Slivkins constructs a DAG G and nodes s_i, t_i (with $1 \leq i \leq k$) and s_{ij}, t_{ij} (with $1 \leq i < j \leq k$) such that the input graph has a k-clique if and only if G has paths from each s_i to the corresponding t_i and from each s_{ij} to the corresponding t_{ij}, all edge-disjoint.

The main idea for our reduction is to take Slivkins' construction and

- connect each t_i to s_{i+1} with a b-edge;
- connect each t_{ik} to $s_{(i+1)(i+2)}$ with an a-edge;
- connect each t_{ij} with $i < j < k$ to $s_{i(j+1)}$ with an a-edge; and
- label all edges intended for "verifiers" with a and all edges intended for "selectors" with b. (Some edges in Slivkins' proof are intended for both verifiers and selectors. Here we can add two parallel edges, one labeled a and one labeled b.)

Then, it can be shown that the original instance is in k-Clique if and only if there exists an a-path from s_{12} to $t_{(k-1)k}$ and a b-path from s_1 to t_k. Moreover, the a-path, if it exists, has length $k' \in O(k^2)$. ◀

The two colors in the proof of Theorem 10 play a central role: since the a-path cannot use any b-edges and vice versa, we have much control over where the two paths can be. The following Theorem shows that the construction in Theorem 10 can be strengthened so that we do not need the two colors.

This is a non-trivial change. Very roughly, one can think of the graph G in the proof of Theorem 10 as a grid with k rows and n columns, where the a-edges are "vertical" and the b edges are "horizontal". (In reality, each coordinate in this grid is another gadget with $4k$ nodes.) The task for Theorem 11 is to change the proof so that paths with mixed a-edges and b-edges do not lead to a solution. For doing this, we use two ideas:

- We use the idea of "control nodes" by Grohe and Grüber [19, Lemma 16], who showed that Slivkins' construction can be used to show that k-Disjoint-Cycles is W[1]-hard.
- We replace each b-edge by a path p_b of length k', ensuring that each path that has p_b as subpath is too long.

▶ **Theorem 11.** *PTwoDisjointPaths is W[1]-hard.*

We provide a proof sketch in Appendix A.

For completeness, we mention the complexity of other variants of PTwoDisjointPaths, some of which can be shown by extending the technique from Theorem 11. We define TwoDisjointPaths$^{\leq}$ and TwoDisjointPaths$^{\geq}$ analogously to PTwoDisjointPaths by requiring that p_1 has length $\leq k$ and $\geq k$, respectively.

▶ **Theorem 12.**
- *TwoDisjointPaths$^{\leq}$ is W[1]-hard.*
- *TwoDisjointPaths$^{\geq}$ is NP-complete for every constant $k \in \mathbb{N}$ ([17]).*
- *PTwoColorDisjointPaths, PTwoDisjointPaths, and TwoDisjointPaths$^{\leq}$ are in W[P].*

Here, being in W[P] implies that the problems are in PTIME for each fixed k.

6 Enumerating Simple Regular Paths

We now turn to the question of enumerating simple paths with polynomial delay. A starting point is Yen's algorithm [41] for finding simple paths from a source s to target t. Yen's algorithm usually takes another parameter K and returns the K shortest simple paths, but we present a version here for enumerating all simple paths, see Algorithm 1.

We give a high-level explanation. First, observe that each shortest path in a graph is also a simple one. Therefore, the first solution is obtained by finding a shortest path p. The next shortest path must differ in some edge from p. So we search (if it exists), for all i, the shortest path that shares the first i edges with p, but not the $(i+1)$th edge. One of the shortest paths found this way is the next solution, which we again store in p. The next shortest path must again differ in some edge from the paths we already found. So we search again, for all i, for a shortest path that shares the first i edges with the new p, but not the $(i+1)$th edge. To avoid rediscovering an old path, we also forbid other edges to appear in the new path (lines 9–11). Correctness is proved in [41].

▶ **Theorem 13** (Implicit in [41])**.** *Given a graph G and nodes s, t, Algorithm 1 enumerates all simple paths from s to t in polynomial delay.*

Proof sketch. The original algorithm of Yen [41] finds, for a given G, s, t, and $K \in \mathbb{N}$ the K shortest simple paths from s to t in G. Its only difference to Algorithm 1 is that it stops when K paths are returned.

Yen does not prove that the algorithm has polynomial delay, but instead shows that the delay is $O(KN + N^3)$, where N is the number of nodes in G.[10] Unfortunately, K can be exponential in $|G|$ in general. However, the reason why the algorithm has K in the complexity is line 9, which iterates over all paths in A. If we do not store A as a linked list as in [41] but as a prefix tree of paths instead, the algorithm only needs $O(N^2)$ steps to complete the entire for-loop on line 9 (without any optimizations). Indeed, if paths p and p' share the first i edges, they will share a path of length i from the root node in the prefix tree. So we can find all forbidden $i+1$th edges by forbidding all edges that start at the end of this path. We therefore obtain delay $O(N^3)$ from Yen's analysis. ◀

[10] In [41], Section 5, he notes that computing path number k in the output costs, in his terminology, $O(KN)$ time in Step I(a) and $O(N^3)$ in Step I(b).

Algorithm 1 Yen's algorithm.

Input: Graph $G = (V, E)$, nodes s, t
Output: The simple paths from s to t in G

1: $A \leftarrow \emptyset$ ▷ A is the set of paths already written to output
2: $B \leftarrow \emptyset$ ▷ B is a set of paths from s to t
3: $p \leftarrow$ a shortest path from s to t in G
4: **while** $p \neq$ null **do** ▷ As long as we find a path p
5: **output** p
6: Add p to A
7: **for** $i = 1$ to $|p|$ **do**
8: $G' \leftarrow (V', E')$, where $V' = V \setminus V(p[0, i-1])$ and $E' = E \cap (V' \times V')$
9: **for** every path p_1 in A with $p_1[0, i-1] = p[0, i-1]$ **do**
10: Delete the edge $p_1[i-1, i]$ in G'
11: **end for** ▷ G' now no longer has paths already in A
12: Find a shortest path p_2 from $p[i, i]$ to t in G'
13: Add $p[0, i] \cdot p_2$ to B
14: **end for**
15: $p \leftarrow$ a shortest path in B ▷ $p \leftarrow$ null if $B = \emptyset$
16: Remove p from B
17: **end while**

6.1 Enumeration for Downward Closed Languages

Yen's algorithm immediately shows that EnumSimPaths can be solved in polynomial delay for languages that are closed under taking subsequences. Formally, we say that a language L is *downward closed* if, for every word $w = a_1 \cdots a_n \in L$ and every sequence $0 < i_1 < \cdots < i_k < n + 1$, we have that $a_{i_1} \cdots a_{i_k} \in L$.

▶ **Proposition 14.** *EnumSimPaths is in polynomial delay for regular expressions r such that $L(r)$ is downward closed.*

Proof sketch. Assume that (G, s, t) and r is an input for EnumSimPaths such that $L(r)$ is downward closed. Let $N = (Q, \Sigma, \delta, Q_I, Q_F)$ be an NFA for r.

We change Algorithm 1 as follows. In line 3, instead of finding a shortest path p in G, we first find a shortest path p in $(G, s, t) \times N$. We then replace every node of the form $(u, q) \in V \times Q$ in p by u.

In line 12 we need to find a shortest path in a product between $(G', p[i, i], t)$ and N. More precisely, let $J = \delta^*(\text{lab}(p[0, i]))$ and denote by N_J the NFA with initial state set J, that is, $(Q, \Sigma, \delta, J, Q_F)$. Then, in line 12 we first find a shortest path p_2 from any node in $\{(p[i, i], q_i) \mid q_i \in \delta^*(\text{lab}(p[0, i]))\}$ to any node in $\{(t, q_F) \mid q_F \in Q_F\}$ in $(G', p[i, i], t) \times N_J$. We then replace every node of the form $(u, q) \in V \times Q$ in p_2 by u. ◀

6.2 Enumeration for STEs

We show that Theorem 6(a) – the FPT part – can be extended to enumeration problems. We note that we do not need to show hardness, since the W[1]-hardness in Theorem 6(b) already holds for the decision problems.

To this end, a *parameterized enumeration problem* is defined analogously as an enumeration problem, but its input is of the form $(x, k) \in \Sigma^* \times \mathbb{N}$. It is in *FPT delay* if there exists an

algorithm that enumerates the output such that the time between two consecutive outputs is bounded by $f(k) \cdot |(x,k)|^c$ for a constant c and a computable function f. Notice that each problem in polynomial delay is also in FPT delay.

▶ Remark. Yen's algorithm makes two important calls to a black box algorithm for computing a shortest path, namely on lines 3 and 12. (There is another call to "shortest path" on line 15, but this one is only important for the ordering of the outputs and not for the correctness of the algorithm.) We can show that the algorithm is also correct if these two calls simply return a *simple path* instead of a shortest one. The main reason why this works is that no simple path from s to t is a subpath of another simple path from s to t. Therefore, we do not "lose" a path by enumerating them in this different order. Therefore, working on $(G,s,t) \times N$ as in the proof of Proposition 14, Yen's algorithm can be applied to any class of RPQs for which we can compute simple paths on lines 3 and 12 sufficiently efficiently.

Using this idea, we can show the following.

▶ **Theorem 15.** *Let $\mathcal{R}$ be a cuttable class of STEs. Then* EnumSimPaths($\mathcal{R}$) *is in FPT delay.*

To prove Theorem 15, we also need to show that the enumeration versions of PSimPath, PSimPath$^{\leq}$, and PSimPath$^{\geq}$ (from Section 5.1) are in FPT delay.

▶ **Theorem 16.** *PEnumSimPaths, PEnumSimPaths$^{\leq}$, and PEnumSimPaths$^{\geq}$ are in FPT delay.*

The proofs of the results in Theorem 16 are all along the same lines. We observe that the FPT algorithms for the decision versions of the problems can be trivially adjusted to also return a matching path if it exists. We also need to show that we can find simple paths matching *suffixes*[11] in the language (for the adapted line 12 of Yen's algorithm). This can also be done here, essentially because the suffixes of the languages we need to consider again can be solved with our FPT algorithms.

6.3 Data Complexity of Enumeration

Finally, we consider the *data complexity* of simple path enumeration. Bagan et al. [5] studied the data complexity of SimPath and discovered a dichotomy w.r.t. a class $\mathsf{C}_{\text{tract}}$ of regular languages.[12] More precisely, although SimPath(r) can be NP-complete in general, it is in PTIME if $L(r) \in \mathsf{C}_{\text{tract}}$ and NP-complete otherwise [5, Theorem 2]. Here, $\mathsf{C}_{\text{tract}}$ is defined as follows.

▶ **Definition 17** (Similar to [5], Theorem 4)**.** For $i \in \mathbb{N}$, we say that a regular language L *can be i-loop abbreviated* if, for all $w_\ell, w, w_r \in \Sigma^*$ and $w_1, w_2 \in \Sigma^+$ we have that, if $w_\ell w_1^i w w_2^i w_r \in L$, then $w_\ell w_1^i w_2^i w_r \in L$. We define $\mathsf{C}_{\text{tract}}$ as the set of regular languages L such that there exists an $i \in \mathbb{N}$ for which L can be i-loop abbreviated.

We show that Bagan et al.'s classification also leads to a dichotomy w.r.t. polynomial delay enumeration in terms of data complexity.

[11] More precisely, we need *language derivatives*, sometimes also called *Brzozowski derivatives*.

[12] They actually proved that there is a trichotomy: the third characterization is that SimPath is in AC^0 if $L(r)$ is finite.

▶ **Theorem 18.** *In terms of data complexity,*

(a) *EnumSimPaths*(r) *can be solved in polynomial delay if* $L(r) \in \mathsf{C}_{tract}$ *and*

(b) *SimPath*(r) *is NP-complete otherwise.*

Proof sketch. Part (b) is immediate from [5, Theorem 1]. For (a), our plan is to use Bagan et al.'s algorithm for simple paths (which we call BBG algorithm) as a subroutine in Yen's algorithm. We call BBG in lines 3 and 12, so that the algorithm receives

(i) a simple path from s to t that matches r in line 3 and

(ii) a simple path p_2 from $p[i]$ to t such that $p[0,i] \cdot p_2$ matches r in line 12,

respectively. Change (i) to Yen's algorithm is trivial. Change (ii) can be done by calling BBG with $(G', p[i], t)$ for the language of the automaton N_J in the proof of Proposition 14. ◀

The algorithm for Theorem 18(a) can even be adapted to output paths in increasing length or radix order.

▶ Remark (STEs versus $\mathsf{C}_{\mathsf{tract}}$). Notice that every STE is in $\mathsf{C}_{\mathsf{tract}}$. Therefore, the data complexity of their evaluation problem is in PTIME (and in polynomial delay for the enumeration version). Since $\mathsf{C}_{\mathsf{tract}}$ is a much bigger class than STEs, it is remarkable that, in Table 1, all expressions in $\mathsf{C}_{\mathsf{tract}}$ are *unions* of STEs.

7 Conclusions

Our main result shows a dichotomy on the parameterized complexity of evaluating *simple transitive expressions (STEs)*, which are a class of regular expressions powerful enough to capture over 99% of the RPQs occurring in a recent practical study [8].

The central property that we require for a class of expressions so that evaluation is in FPT is *cuttability*, i.e., constant *cut borders* (also see Figure 1). Looking at Table 1, we see that the cut borders for expressions in practice are indeed very small: it is one for a^*b, two for abc^*, and zero in all other cases.

Therefore, although the *simple path* semantics of RPQs is known to be hard in general, it seems that the RPQs that users actually ask are much less harmful. In fact, since the vast majority (over 99%) of expressions in Table 1 has cut borders of at most two, our FPT result in Theorem 6 implies that evaluation for this majority of expressions is in polynomial time combined complexity. Furthermore, matching paths can be enumerated in polynomial delay. (Recall that, if P $\neq$ NP, this is impossible even for fixed expressions: evaluation for a^*ba^* or $(aa)^*$ under simple path semantics is NP-complete.)

Finally, we note that this paper investigated evaluation for *node-distinct paths*. Preliminary work shows that our techniques can also be applied for *edge-distinct paths* [28]. More precisely, there is a similar dichotomy for edge-distinct paths, with subtle differences. For instance, evaluation for a^kb^* under edge-distinct path semantics is in FPT, whereas it is W[1]-hard under node-distinct (i.e., simple) path semantics.

We also noticed that our techniques extend beyond the class of STEs. For instance, we can also prove that, for every constant c and word w with $|w| = c$, the problem SimPath$(a^kw?a^*)$ with parameter k is in FPT. We believe that it would be very interesting to understand to which extent cuttability can be used to obtain FPT results for larger classes of RPQs (such as unions of STEs).

References

1 Margareta Ackerman and Jeffrey Shallit. Efficient enumeration of words in regular languages. *Theoretical Compututer Science (TCS)*, 410(37):3461–3470, 2009.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

3 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

4 Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *International Conference on World Wide Web (WWW)*, pages 629–638, 2012.

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

6 Pablo Barceló. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*, pages 175–188, 2013.

7 Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 222–233, 2004.

8 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proceedings of the VLDB Endowment (PVLDB)*, 11, 2017.

9 Leizhen Cai and Junjie Ye. Finding two edge-disjoint paths with length constraints. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 62–73, 2016.

10 Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.

11 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 323–330, 1987.

12 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

13 Holger Dell. Personal communication, 2017.

14 Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

15 Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theoretical Computer Science (TCS)*, 141(1):109–131, 1995.

16 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM*, 63(4):29:1–29:60, 2016.

17 Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science (TCS)*, 10(2):111–121, 1980.

18 Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):1028–1039, 2011.

19 Martin Grohe and Magdalena Grüber. Parameterized approximability of the disjoint cycle problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 363–374, 2007.

20 Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

21 Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. Counting and random generation of strings in regular languages. In *Symposium on Discrete Algorithms (SODA)*, pages 551–557, 1995.

22 Benny Kimelfeld and Yehoshua Sagiv. Extracting minimum-weight tree patterns from a schema with neighborhood constraints. In *International Conference on Database Theory (ICDT)*, pages 249–260, 2013.

23 Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

24 Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

25 Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

26 Katja Losemann and Wim Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems*, 38(4):24:1–24:39, 2013.

27 Erkki Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–62, 1997.

28 Wim Martens and Tina Trautner. Enumeration problems for regular path queries. *CoRR*, abs/1710.02317, 2017. URL: `https://arxiv.org/abs/1710.02317`.

29 Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 12 1995.

30 Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

31 Neo4j. Intro to cypher. `https://neo4j.com/developer/cypher-query-language/`, 2017.

32 Neo4j. The neo4j developer manual v3.3. `https://neo4j.com/docs/developer-manual/3.3/`, 2017.

33 Opencypher. `www.opencypher.org`. Visited on Sept. 14, 2017.

34 Stefan Plantikow, Mats Rydberg, and Petra Selmer. CIP2017-01-18 – configurable pattern matching semantics. `https://github.com/boggle/openCypher/blob/isomatch/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc`. Visited on Aug. 08, 2017.

35 Aleksandrs Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 24(1):146–157, 2010.

36 Dimitri Surinx, George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs using transitive closure. *Logic Journal of the IGPL*, 23(5):759–788, 2015.

37 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science (TCS)*, 8(2):189–201, 1979.

38 SPARQL 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, 2013. World Wide Web Consortium.

39 World wide web consortium. `www.w3.org`. Visited on Sept. 14, 2017.

40 Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990.

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

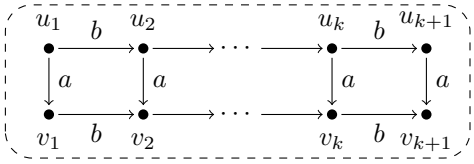

Figure 2 Internal structure of each of the gadgets $G_{i,j}$.

A Appendix

We present a proof sketch for Theorem 11. We will do an *fpt-reduction*, which we define next. If L and L' are two parameterized problems, an *fpt-reduction* from L to L' is an algorithm $\mathcal{R}$ that, given an instance (x, k) of L, outputs an instance (x', k') of L' such that

- (x, k) is a yes-instance of L if and only if (x', k') is a yes-instance of L',
- $k' \leq g(k)$ for some computable function g, and
- the running time of $\mathcal{R}$ is $f(k) \cdot |x|^{O(1)}$ for some computable function f.

▶ **Theorem 11.** *PTwoDisjointPaths is W[1]-hard.*

Proof sketch. We reduce from k-Clique, which is well known to be W[1]-complete [15, Corollary 3.2]. Let $G = (V, E)$ be an *undirected* graph and assume w.l.o.g. that $V = \{1, \ldots, n\}$.

The reduction consists of two steps. In the first step, we will construct a two-colored graph G', nodes s_a, t_a, s_b, t_b, and parameter $k' \in \Theta(k^2)$ such that G has a k-clique if and only if $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths. The graph G' will have $O(k^2 n)$ nodes. In the second step, we will construct a graph G'' and nodes s_1, t_1, s_2, t_2 such that $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths if and only if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths. The graph G'' will have $O(k^4 n)$ nodes.

We now explain the construction of G'. It contains kn gadgets $G_{i,j}$ with $i = 1, \ldots, k$ and $j = 1, \ldots, n$, each consisting of $2(k+1)$ nodes. Gadgets will be ordered in k rows, where row i has the gadgets $G_{i,1}, \ldots, G_{i,n}$. Furthermore, G' contains $k+1$ additional nodes $r_1, \ldots, r_{k+1}$ that link the rows together, and $k + 1 + k(k-1)/2$ control nodes $c_1, \ldots c_{k+1}$ and $c_{i_1 i_2}$ with $1 \leq i_1 < i_2 \leq k$ that will limit the number of disjoint paths from row $i-1$ to row i or from row i_1 to i_2, respectively. (To be fair, c_1 and c_{k+1} do not link rows together but just serve as start and end-nodes.) We define $s_a = c_1$, $t_a = c_{k+1}$, $s_b = r_1$, and $t_b = r_{k+1}$.

We will now explain how the nodes are connected in G'. We will denote by $u \xrightarrow{a} v$ that there is an a-edge from u to v (similar for b-edges). Each gadget contains a disjoint copy of $2(k+1)$ nodes which we call $u_1, u_2, \ldots, u_{k+1}$ and $v_1, v_2, \ldots, v_{k+1}$. To simplify notation, we sometimes give these nodes the same name (e.g., in Figures 3, 4, and 5), even though they are different. One such gadget is depicted in Figure 2. To avoid ambiguity, we may also refer to node u_ℓ in gadget $G_{i,j}$ by $G_{i,j}[u_\ell]$. Each gadget contains edges $u_\ell \xrightarrow{a} v_\ell$ (for every $\ell = 1, \ldots, k+1$) and $u_\ell \xrightarrow{b} u_{\ell+1}$ and $v_\ell \xrightarrow{b} v_{\ell+1}$ (for every $\ell = 1, \ldots, k$).

We now explain how the gadgets $G_{i,j}$ are connected within the same row, see Figure 3. In each row $i \in \{1, \ldots, k\}$, node r_i has two outgoing edges $r_i \xrightarrow{b} G_{i,1}[u_1]$ and $r_i \xrightarrow{b} G_{i,2}[v_1]$. We also have two incoming edges for r_{i+1}, namely $G_{i,n-1}[u_{k+1}] \xrightarrow{b} r_{i+1}$ and $G_{i,n}[v_{k+1}] \xrightarrow{b} r_{i+1}$. Furthermore, we have the edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+1}[u_1]$ and $G_{i,j}[v_{k+1}] \xrightarrow{b} G_{i,j+1}[v_1]$ for every $j = 1, \ldots, n-1$. We also add edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+2}[v_1]$ for every $j = 1, \ldots, n-2$.

We now explain how the gadgets $G_{i,j}$ are connected in different rows via the control nodes c_i and $c_{i_1 i_2}$ (Figure 4). We first consider the edges from row i to $i+1$. In each

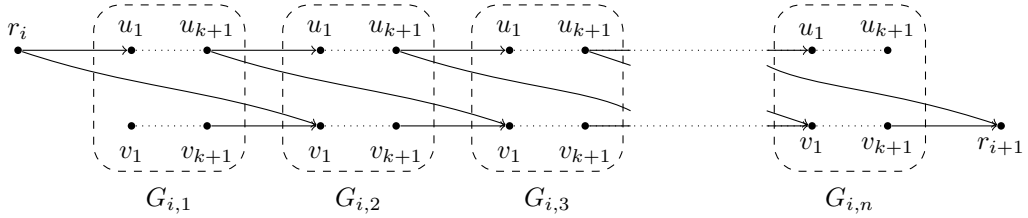

Figure 3 The b-edges in row i. The internal structure of the $G_{i,j}$ is as in Figure 2.

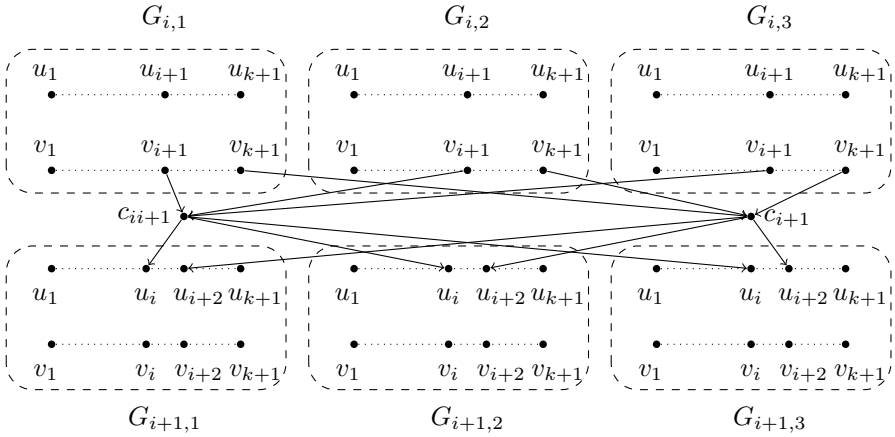

Figure 4 The a-edges from row i to row $i+1$. (We assume $n = 3$ in the picture).

row $i = 1, \ldots, k-1$, and every $j = 1, \ldots, n$, we add the edges $G_{i,j}[v_{k+1}] \xrightarrow{a} c_{i+1}$ and $c_{i+1} \xrightarrow{a} G_{i+1,j}[u_{i+2}]$. Furthermore, we add the edges $c_1 \xrightarrow{a} G_{1,j}[u_2]$ and $G_{k,j}[v_{k+1}] \xrightarrow{a} c_{k+1}$. We connect two rows i_1, i_2, with $1 \leq i_1 < i_2 \leq k$, by adding the edges $G_{i_1,j}[v_{i_2}] \xrightarrow{a} c_{i_1 i_2}$, and $c_{i_1 i_2} \xrightarrow{a} G_{i_2,j}[u_{i_1}]$ for all $j = 1, \ldots, n$.

The edges in G are modeled in G' by adding the edge $G_{i_2,x}[v_{i_1}] \xrightarrow{a} G_{i_1,y}[u_{i_2+1}]$ if and only if $1 \leq i_1 < i_2 \leq k$, $x \neq y$, and $(x, y) \in E$. This is illustrated in Figure 5.

Finally, we define $k' = k(k-1)/2 \cdot 5 + 3k$.

This concludes the construction of G'. We denote by G'_a the subgrapn of G' that contains only the a-edges.

▶ **Lemma 19.** *The graph G'_a has the following properties:*

(a) *G'_a is a DAG. Moreover, there is a strict total order $<_c$ on all control nodes C such that, for every path from a node $v \in C$ to another node $v' \in C$ where no intermediate vertex is in C, node v' is the successor of v in $<_c$.*

(b) *Each path in G'_a has length exactly k' if and only if it is from c_1 to c_{k+1}.*

(c) *Each path in G'_a of length k' visits all control nodes, i.e., it contains all c_i and $c_{i_1 i_2}$, with $i \in \{1, \ldots, k+1\}$ and $1 \leq i_1 < i_2 \leq k$.*

(d) *Each path in G'_a of length k' has at least one edge $u_\ell \xrightarrow{a} v_\ell$ in every row of G'_a.*

We omit the proof of Lemma 19 due to space constraints.

We prove that $(G, k) \in k$-Clique if and only if $(G', k') \in$ PTwoColorDisjointPaths. Let us first assume that the undirected graph G has a k-clique with nodes $\{n_1, \ldots, n_k\}$. Then an a-path can go from c_1 to c_{k+1} using only the gadgets G_{i,n_i} with $i = 1, \ldots, k$. The reason is that, since $(n_{i_1}, n_{i_2}) \in E$, the edges $G_{i_2,n_{i_2}}[v_{i_1}] \xrightarrow{a} G_{i_1,n_{i_1}}[u_{i_2+1}]$ exist for all $i_1 \leq i_2$. Due to Lemma 19(b), this path has exactly k' edges. The b-path, on the other hand, can go from

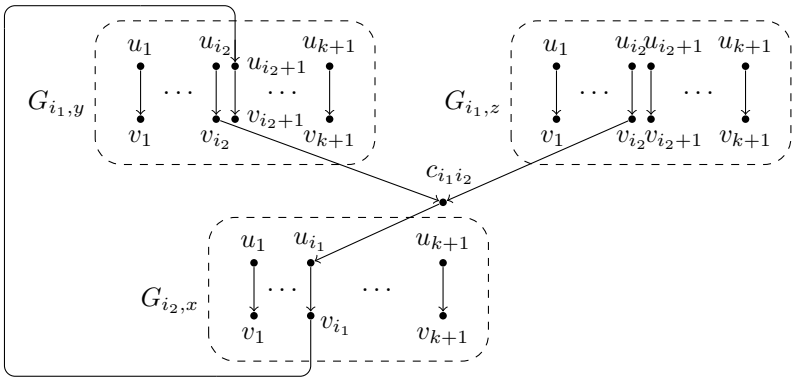

Figure 5 The a-edges in the gadgets and between gadgets $G_{i_1,y}$, $G_{i_1,z}$ and $G_{i_2,x}$, with $i_1 < i_2 - 1$, under the assumption that $(x, y) \in E$ and $(x, z) \notin E$.

r_1 to r_{k+1} and skip exactly G_{i,n_i} for all $i = 1, \ldots, k$ (using the diagonal edges in Figure 3). Since it skips these G_{i,n_i}, it is node-disjoint from the a-path and therefore we have a solution for PTwoColorDisjointPaths.

For the other direction let us assume that there exists a simple a-path p_a from c_1 to c_{k+1} and a simple b-path p_b from r_1 to r_{k+1} in G' such that p_a and p_b are node-disjoint and p_a has length k'. We show that G has a k-clique. Since every b-path from r_1 to r_{k+1} goes through each row, that is, from r_i to r_{i+1} for all $i = 1, \ldots, k$, this is also the case for p_b. By construction p_b must also skip exactly one gadget in each row, using the diagonal edges in Figure 3. Furthermore, for each gadget $G_{i,j}$ that p_b visits, it must be the case that it either visits all nodes $u_1, \ldots, u_{k+1}$ or all nodes $v_1, \ldots, v_{k+1}$. (This is immediate from Figure 2, showing all internal edges of a gadget.) Therefore, since p_a and p_b are node-disjoint, the p_a cannot visit any gadget $G_{i,j}$ already visited by p_b. Therefore, p_a, which goes from c_1 to c_{k+1}, can only do so through the k skipped gadgets, call them G_{i,n_i} for $i = 1, \ldots, k$. Recall that the edges between the gadgets $G_{i_2,n_{i_2}}$ and $G_{i_1,n_{i_1}}$ only exist if $(n_{i_1}, n_{i_2}) \in E$. As these edges are necessary for the existence of the a-path from c_1 to c_{k+1} that uses only the skipped gadgets, all nodes n_i must be pairwise adjacent in G. That is, they form a clique of size k in G. This completes the first step of the reduction.

We now explain the second and final step. We construct the graph G'' from G' by replacing each b-edge with a b-path of length k'. (Even though PTwoDisjointPaths does not care about a-edges or b-edges, we keep them to simplify the reasoning in the remainder of the proof.) We define $s_1 = s_a$, $t_1 = t_a$, $s_2 = s_b$, and $t_2 = t_b$.

▶ **Observation 20.** *In G'', we have that*

(a) *every path from c_1 to c_{k+1} has length at least k' and*

(b) *every path from c_1 to c_{k+1} has length exactly k' if and only if it is an a-path.*

We prove the observation using Lemma 19(b). For part (a) we have two cases. If a path from c_1 to c_{k+1} is an a-path, the result is immediate from Lemma 19(b). If it uses at least one b-edge, then it uses at least k' b-edges by construction. Thus, the path will have length at least k'.

For part (b), if a path from c_1 to c_{k+1} has length exactly k', it uses at least one a-edge since c_{k+1} only has incoming a-edges. If it used at least one b-edge, it would therefore use at least $k' + 1$ edges which contradicts that the length is k'. The converse direction is immediate from Lemma 19(b). This concludes the proof of Observation 20.

We show that $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths if and only if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths. If $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths, then we can use the corresponding paths in G'' (where we follow the longer b-paths in G'' instead of the b-edges in G').

Conversely, if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths, it follows from Observation 20 that p_1 can only use a-edges. We now show that the path p_2 from r_1 to r_{k+1} can only use b-edges, that is, we show that it cannot use a-edges. There are three types of a-edges in G'': (i) the ones from and to control nodes, (ii) "upward" edges that connect row i_2 to row i_1 with $i_1 < i_2$, and (iii) edges from u_ℓ to v_ℓ in one gadget.

Notice that, by construction, p_2 must visit nodes in row 1 and later also nodes in row k. To do so, p_2 cannot use edges from or to control nodes (type (i)), since, due to Lemma 19(c), p_1 already visits all control nodes. So p_2 cannot go from row i to a row j with $i < j$ via a-edges. This means that, if $i < j$, then p_2 can only go from row i to row j through r_{i+1} (and through nodes in row $i + 1$), since every remaining path from row i to a larger row goes through r_{i+1}. So, in order to go from row 1 to row k, path p_2 needs to visit all nodes $r_2, \ldots, r_k$, in that order. This means that it is also impossible for p_2 to use edges of type (ii). Indeed, if p_2 were to use an edge from row j to row i with $j > i$, then it would need to visit r_{i+1} a second time to arrive back in row j. Finally, if p_2 used an edge of type (iii) in row i, then, by construction, it would have to visit every gadget in this row. But since p_1 already uses at least one edge from u_ℓ to v_ℓ in each row, see Lemma 19(d), this means that p_2 cannot be node-disjoint with p_1. This completes the proof. ◀