A trichotomy for regular simple path queries on graphs

Guillaume Bagan, Angela Bonifati, Benoit Groz

Journal of Computer and System Sciences 108 (2020) 29–48

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

A trichotomy for regular simple path queries on graphs ☆

Guillaume Bagan [a,*], Angela Bonifati [a], Benoit Groz [b]

[a] Université Lyon 1, LIRIS UMR CNRS 5205, F-69622, Lyon, France
[b] Université Paris Sud, LRI UMR CNRS 8623, F-91405, Orsay, France

ARTICLE INFO

Article history:
Received 3 June 2018
Received in revised form 23 June 2019
Accepted 19 August 2019
Available online 20 September 2019

Keywords:
Graphs
Paths
Regular simple paths
Complexity
Regular languages
Automata

ABSTRACT

We focus on the computational complexity of regular simple path queries (RSPQs). We consider the following problem RSPQ(L) for a regular language L: given an edge-labeled digraph G and two nodes x and y, is there a simple path from x to y that forms a word belonging to L? We fully characterize the frontier between tractability and intractability for RSPQ(L). More precisely, we prove RSPQ(L) is either AC^0, NL-complete or NP-complete depending on the language L. We also provide a simple characterization of the tractable fragment in terms of regular expressions. Finally, we also discuss the complexity of deciding whether a language L belongs to the fragment above. We consider several alternative representations of L: DFAs, NFAs or regular expressions, and prove that this problem is NL-complete for the first representation and PSPACE-complete for the other two.

1. Introduction

Graph databases have been investigated starting from the late 80s and are now again in vogue due to their wide application scenarios, ranging from social networks to biological and scientific databases (see [2] for a survey). Regular path queries (RPQs) are one of the most notable classes of queries on graph databases. They allow to retrieve pairs of nodes connected by a path, where the path is described through a regular expression. Such regular path queries are computable in time polynomial in both query and data complexity (combined complexity). In this paper, we investigate the computational complexity of regular simple path queries (RSPQs), a variant of RPQ in which the path connecting the pair has to be simple, i.e., does not have repeated vertices. Given an edge-labeled graph G and a regular language L, an RSPQ selects pairs of vertices connected by a simple path whose edge labels form a word in L.

The evaluation of RSPQs is NP-complete even for fixed basic languages such as $(aa)^*$ or a^*ba^* [20], in sharp contrast with RPQs. RSPQs are desirable in many application scenarios [17,23,6,15,13,30], such as transportation problems, VLSI design, metabolic networks, DNA matching and routing in wireless networks. Additionally, regular simple paths have been recently considered in SPARQL 1.1 queries exhibiting property paths. In particular, recent studies on the complexity of property paths in SPARQL [3,18] have highlighted the hardness of the semantics proposed by W3C to evaluate such paths in RDF graphs. Roughly speaking, according to the semantics considered in [18], the evaluation of expressions under Kleene-star closure should return a simple path, whereas the evaluation of the remaining expressions allows to traverse the same vertex multiple times. As such, the semantics studied in [18] is an hybrid between regular paths and regular simple paths semantics. RPQs have been recently found in practice within real-world SPARQL query logs, such as DBPedia and Wikidata

☆ This article is an extended version of [5].

* Corresponding author.
E-mail addresses: guillaume.bagan@liris.cnrs.fr (G. Bagan), angela.bonifati@univ-lyon1.fr (A. Bonifati), benoit.groz@lri.fr (B. Groz).

https://doi.org/10.1016/j.jcss.2019.08.006
0022-0000/

query logs [8,7,9], showing thus the increasing interest of users manipulating real-world graph datasets with these queries. In particular, the majority of RPQs analyzed in large corpuses [8,9] belong to the tractable fragment $\mathcal{C}_{tract}$ studied in this paper, further confirming the applicability and impact of our theoretical investigation.

Contributions. In this paper, we address the long standing open question [20,6] of exactly characterizing the maximal class of regular languages for which RSPQs are tractable. By "tractable" we mean computable in time polynomial in the size of the graph. Precisely, we establish a comprehensive classification of the complexity of RSPQs for a fixed regular language L. A first step towards this important issue has been made in [20]. They exhibit a tractable fragment: the class of languages closed under taking subword. However, their fragment is not maximal.

Our contributions can be detailed as follows. We introduce a class of languages, named $\mathcal{C}_{tract}$, for which RSPQs can be evaluated in polynomial time (data complexity), and even in NL. We then show that RSPQ evaluation is NP-complete for every regular language that does not belong to $\mathcal{C}_{tract}$. Consequently, the maximal tractable fragment $\mathcal{C}_{tract}$ characterizes the frontier between tractability and intractability for this problem, under the hypothesis NL $\neq$ NP. We further refine our results to show the following trichotomy: the evaluation of RSPQ problem is either AC^0, NL-complete or NP-complete. We note that we consider the language L as a fixed parameter, thus our results characterize the *data* complexity of RSPQ evaluation.

We also discuss the complexity of deciding, given a language L, whether the RSPQ problem for L is tractable. We consider several alternative representations of L: DFAs, NFAs or regular expressions. We prove that this problem of deciding tractability is NL-complete for the first representation and PSPACE-complete for the two others.

Next, we give a characterization of the tractable fragment $\mathcal{C}_{tract}$ for edge-labeled graphs in terms of regular expressions. We also show that $\mathcal{C}_{tract}$ is closed by union and intersection and show that languages in $\mathcal{C}_{tract}$ are aperiodic, i.e., can be expressed by first-order formulas [29].

We conclude with some minor results that identify further cases where RSPQs admit efficient solutions. We thus prove that RSPQs are FPT for the class of finite languages. Furthermore, we prove that the problem is also FPT for the class of all regular languages when the parameter is the size of the path. Finally, we prove that the problem RSPQ is polynomial w.r.t. combined complexity on graphs of bounded directed treewidth. This is actually a straightforward generalization of a result of [14].

A preliminary version of the present article has appeared in [5] without proofs of the main results. Here we provide detailed proofs.

Related work. A few papers deal with RSPQs or are related to them. Lapaugh et al. [16] prove that finding simple paths of even length is polynomial for non directed graphs and NP-complete for directed graphs. This study has been extended in [4] by considering paths of length i mod k. Similarly, finding k disjoint paths with extremities given as input is polynomial for non directed graphs [26] and NP-complete for directed graphs [10]. Using these results, Mendelzon and Wood prove that evaluating an RSPQ on an edge-labeled directed graph is NP-hard even for fixed languages [20]. However, they show that the problem can be decided in polynomial time for subword-closed languages. They also show that the problem becomes polynomial under some restrictions on the size of cycles of both graph and automaton. A subsequent paper [22] proves the polynomiality for the class of outerplanar graphs. Barrett et al. [6] extend this result, proving that the regular simple path problem is polynomial w.r.t. combined complexity for graphs of bounded treewidth. Barrett et al. [6] also show that the problem is NP-complete for the class of grid graphs even when the language is fixed.

We already mentioned that the complexity of evaluating SPARQL property paths has been investigated in previous studies [18,3]. As highlighted above, the semantics of SPARQL property paths blends the arbitrary paths and simple paths semantics. Losemann and Martens [18] and Arenas et al. [3] focus on the complexity of evaluating such property paths. They show that the evaluation is NP-complete in several cases, and exhibit cases in which it is polynomial. More precisely, Losemann and Martens [18] classify several fragments of property paths with respect to their complexity. Both the semantics and the query fragments are different from the ones in our paper. Counting the number of paths that match a regular expression (which is permitted for instance in SPARQL 1.1) is hard in many cases [18,3]. Recently, Martens and Trautner [19] studied the decision- and enumeration problems concerning the evaluation of RPQs by considering several semantics: arbitrary paths, shortest paths, and simple paths. While we prove here a trichotomy for the data complexity of the decision problem, they focus on the data complexity of the polynomial delay enumeration problem.

Section 2 introduces and illustrates the problem. We then establish our trichotomy. We first show in section 3 why languages outside our tractable fragment have NP-hard complexity. Section 4 discusses some properties of the tractable fragment, and shows how those properties (about strongly connected components) lead to a polynomial evaluation algorithm. And finally, Section 5 details our algorithm to deal with the tractable languages.

2. Preliminaries

Let $[n]$ denote the set of integers $\{1, \ldots, n\}$ and $[n, m]$ denote the set of integers $\{n, \ldots, m\}$.

2.1. Complexity

A TM refers to a Turing Machine and a NDTM refers to a non deterministic Turing Machine. AC^0, L, NL, P, NP, PSPACE refer to the classical classes of complexity [24]. The relations $AC^0 \subseteq L \subset NL \subseteq P \subseteq NP \subseteq PSPACE$ between these classes are well known.

FL is the class of functions computable by a deterministic log-space transducer. The class L^{NL} is the set of decision problems computable by a deterministic log-space algorithm with an oracle in NL. The class $\mathrm{FL}^{\mathrm{NL}}$ is the set of functions computable by a deterministic log-space transducer with an oracle in NL. The class $\mathrm{FL}^{\mathrm{NL}}(\mathrm{NL})$ is the set of problems reducible to a problem in NL by a function in $\mathrm{FL}^{\mathrm{NL}}$.

Lemma 1. *[24,11]* $\mathrm{NL} = \mathrm{L}^{\mathrm{NL}} = \mathrm{FL}^{\mathrm{NL}}(\mathrm{NL})$.

$\mathrm{NL}^{\mathrm{NL}}$ is the class of decision problems computable by a non deterministic log-space algorithm with an oracle in NL. The next lemma is true only if we make some restrictions on the oracle machine model (see [27]): the TM must write on the oracle tape deterministically i.e. it works deterministically as soon as it starts to write on the tape until it calls the oracle. The oracle tape is erased at the end of each call.

Lemma 2. *[11]* $\mathrm{NL} = \mathrm{NL}^{\mathrm{NL}}$.

2.2. Graphs

In our paper, we essentially consider db-graphs. A db-graph is a tuple $G = (V, \Sigma, E)$ where V is a set of vertices, Σ is a set of labels and $E \subseteq V \times \Sigma \times V$ is a set of edges labeled by symbols of Σ. Given a set $S \subseteq V$, $G[S]$ the induced subgraph of G by S is $(S, \Sigma, E \cap S \times \Sigma \times S)$. A path p of a db-graph G from x to y is a sequence $(v_1 = x, a_1, \ldots, v_m, a_m, v_{m+1} = y)$ such that for each $i \in [m+1]$, v_i is a vertex in G and for each $i \in [m]$, (v_i, a_i, v_{i+1}) is an edge in G. A path p is simple if all vertices v_i in p are distinct. Given a language $L \subseteq \Sigma^*$, p is L-labeled if $a_1 \ldots a_m \in L$. Given a subset $S \subseteq V$, p is S-restricted if every intermediate vertex of p belongs to S. Given a simple path p and two vertices x and y in p, $p[x, y]$ denotes the subpath of p from x to y.

2.3. Languages and automata

Let L be a regular language. Given a word w and a language L, $w^{-1}L = \{w' : ww' \in L\}$. We denote by $A_L = (Q_L, i_L, F_L, \Delta_L)$ the minimal DFA for L, and by M_L the number of states $M_L = |Q_L|$ in A_L. Whenever the language is obvious from context, we drop the subscript and write M instead of M_L. We assume that A_L is complete i.e. Δ_L is a total function, so that in general A_L may have a sink state. For any state $q \in Q$ and word $w \in \Sigma^*$, $\Delta_L(q, w)$ denotes the state obtained when reading w from q. For any state $q \in Q$ and set of words $S \subseteq \Sigma^*$, $\Delta_L(q, S)$ denotes the set of states q' such that there exists $w \in S$ with $q' = \Delta_L(q, w)$. Finally, $\mathscr{L}_q$ denotes the set of all words accepted from q. For every state q we denote by $Loop(q)$ the set of all non empty words that allow to loop on q: $Loop(q) = \{w \in \Sigma^+ \mid \Delta_L(q, w) = q\}$. We say that a state q' is reachable from a state q if $q' \in \Delta_L(q, \Sigma^*)$. A (strongly connected) component of A_L is a maximal set of states that are pairwise reachable.

The run of L (or A_L) over a path $p = (v_1, a_1, \ldots, a_m, v_{m+1})$ is the mapping $\rho : \{v_1, \ldots, v_{m+1}\} \to Q_L$ such that: $\rho(v_1) = i_L$ and $\rho(v_{i+1}) = \Delta_L(i_L, a_1 \ldots a_i)$ for every $i \in [m]$. There are many characterizations of aperiodic languages [29]. A language L is aperiodic if and only if it satisfies $\Delta_L(q, w^{M+1}) = \Delta_L(q, w^M)$ for every state q and word w. Intuitively, that means that for any state q_0 and a word w, in the infinite sequence $(q_0, q_1, q_2, \ldots)$ with $q_{i+1} = \Delta_L(q, w)$ for any integer i, there is an integer M such that $q^{M+1} = q^M$.

2.4. Regular simple paths

Given a regular language L, we define the following problem:

RSPQ(L)
Input: A db-graph $G = (V, \Sigma, E)$, and two vertices $x, y \in V$
Question: Is there a simple L-labeled path from x to y?

For this problem, L is fixed, so we focus on data complexity. Notice that the representation of L does not matter here. Although we consider the Boolean version of the problem, namely deciding the existence of a path, our algorithms actually also return a simple L-labeled path.

The main problem that we address in this paper is to distinguish cases when RSPQ(L) is tractable (i.e. decidable in polynomial time) and when it is not (i.e. NP-hard).

2.5. The class of tractable languages

We recall that M refers to the size of Q_L, here and henceforth. We next introduce the class $\mathcal{C}_{tract}$ of languages. We will prove that it is exactly the class of regular languages for which RSPQ(L) is tractable.

Definition 1. A regular language L belongs to the class $\mathcal{C}_{tract}$ if the following property is satisfied: for all pairs of states $q_1, q_2 \in Q_L$ and all words w with $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$, $q_2 \in \Delta_L(q_1, \Sigma^*)$ and $w \in Loop(q_2)$, it holds that $w^M \mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$.

This definition is merely a technical definition for $\mathcal{C}_{tract}$, but we will provide in Theorem 6 a more intuitive characterizations of the class.

Example 1. As an introductory example, consider the language $L = a^*(bb^+ + \epsilon)c^*$. We observe that this language belongs to our class $\mathcal{C}_{tract}$. We wish to decide RSPQ(L), i.e., whether there exists a simple path from x to y labeled by L, given two vertices x, y of a db-graph G. It is not absolutely trivial that RSPQ(L) can be solved efficiently: RSPQ(a^*bc^*) has indeed been proved NP-complete. Yet we outline below a polynomial algorithm for L.

We distinguish two cases: there is a simple L-labeled path from x to y if and only if one of the following cases holds:

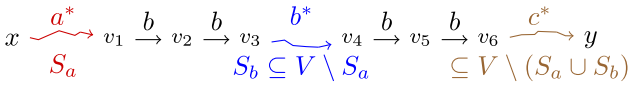
1: there exists a simple $a^*b^kc^*$-labeled path from x to y for some $k \in \{0, 2, 3\}$
2: case 1 does not hold and there exists a simple $a^*b^4b^*c^*$-labeled path from x to y.

The first case is the easiest to check. We first check whether y can be reached from x by a (non-necessarily simple) a^*c^*-labeled path. If we find one, we obtain a simple a^*c^*-labeled path by eliminating its loops. Assume now there is no a^*c^*-labeled path from x to y. We then check as follows if there exists a simple $a^*b^kc^*$-labeled path from x to y for some $k \in \{2, 3\}$: we try every possible assignment for the k middle b-edges. For each combination of k b-edges, we check if the initial b-edge can be reached from x through an a^*-labeled path (avoiding the vertices of the other b edges), and check if the final b-edge can lead to y through some c^*-labeled path (avoiding the vertices of the other b edges). In the resulting $a^*b^kc^*$-labeled path the a^*-labeled prefix and c^*-labeled suffix cannot intersect (we assumed there is no a^*c^* path). Consequently we obtain a simple $a^*b^kc^*$-labeled path by eliminating its loops. As the number of possible assignments for k edges ($k \leq 3$) is polynomial, we have proved that we can find out in polynomial time whether case 1 holds.

Let us now assume w.l.o.g. that there is no $a^*b^kc^*$-labeled path from x to y for $k \in \{0, 2, 3\}$. We can show that in this second case there exists a simple L-labeled path from x to y if and only if there exist six vertices $v_1, v_2, v_3, v_4, v_5, v_6$, two integers l_a, l_b and two sets S_a, S_b satisfying all following conditions:

- the vertices $v_1, \ldots, v_6$ are all distinct except that v_3 may equal v_4.
- there is a b-labeled edge from v_1 to v_2, from v_2 to v_3, from v_4 to v_5, and from v_5 to v_6.
- there is an a^*-labeled path from x to v_1 avoiding all other v_is ($i > 1$). The shortest possible such path has length l_a.
- S_a is the set of all vertices reachable from x through an a^*-labeled path of length at most l_a that avoids all v_is ($i > 1$).
- there is a b^*-labeled path from v_3 to v_4 of which all vertices (but the first and last) avoid S_a and the v_is. The shortest possible such path has length l_b.
- S_b is the set of all vertices reachable from v_3 through any b^*-labeled path of length at most l_b that avoids S_a and all other v_is ($i \neq 4$).
- there is a c^*-labeled path from v_6 to y of which all vertices (but the first) avoid S_a and S_b and all other v_is ($i < 6$).

The figure below summarizes all these conditions.



These conditions can clearly be verified in time polynomial in G. It is relatively clear also that the path constructed above is an L-labeled simple path from x to y, so the conditions are sufficient to obtain an L-labeled simple path. To prove that our procedure is correct, we only have to prove that reciprocally, if there exists a simple L-labeled path we can find one satisfying our restrictions (the conditions above involving the v_i, S_a, S_b).

For every shortest L-labeled simple path p from x to y, let $v_1, \ldots, v_6$ denote the vertices that delimit the first and last two b-edges of p. We next show that those vertices satisfy the conditions above. The last vertex of p that belongs to S_a cannot occur after v_3 in p. Otherwise, we could obtain a simple path p' by replacing the prefix of p up to v with a shorter path through S_a. This resulting path p' would still be L-labeled by definition of the v_i, which contradicts the minimality of p. A similar argument shows that the last occurrence of a vertex from S_b cannot occur after v_6. We conclude that the paths connecting v_3 to v_4 in p (resp. v_6 to y) exclude respectively all vertices from S_a (resp. $S_a \cup S_b$). As a consequence, the path from x to v_1 will only feature vertices from S_a by minimality of p, which proves that vertices $v_1, \ldots, v_6$ satisfy the conditions above.

The crux of our approach is to construct the a^*, b^* and c^* subpaths independently, lest we enumerate exponentially many paths. This is why we require that the b^* subpath avoids S_a: this condition is stronger than necessary to guarantee the first two subpaths do not intersect, but the stronger requirement allows us to build the two subpaths independently, as S_a is a superset of the vertices on the subpath from x to v_1. Our algorithm for tractable instances will generalize this idea.

3. Hard languages for RSPQ

This section is devoted to the proof of a hardness result: RSPQ(L) is NP-hard for every regular language L that does not belong to $\mathcal{C}_{tract}$. The first step toward that proof lies in the following characterization of $\mathcal{C}_{tract}$.

Definition 2 *(Witness of hardness).* Let L be a regular language. A *witness for hardness* of L is a tuple $(w_l, w_m, w_r, w_1, w_2)$ where $w_l, w_r \in \Sigma^*$ and $w_m, w_1, w_2 \in \Sigma^+$ satisfying

- $w_l w_1^* w_m w_2^* w_r \subseteq L$
- $w_l(w_1 + w_2)^* w_r \cap L = \emptyset$.

Lemma 3. *Let L be a regular language that does not belong to $\mathcal{C}_{tract}$. Then, L admits a witness for hardness.*

Proof. Let L be a regular language that does not belong to $\mathcal{C}_{tract}$. For commodity, we distinguish two cases, depending on whether L satisfies or not the following property: $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$ for every $q_1, q_2 \in Q_L$ such that $q_2 \in \Delta_L(q_1, \Sigma^*)$ and $Loop(q_1) \cap Loop(q_2) \neq \emptyset$ (Property $\mathcal{P}$).

Let L be a language that does not satisfy Property $\mathcal{P}$, there exist q, q_2, w_m, w, w_r such that $\Delta_L(q, w_m) = q_2$, $w \in Loop(q) \cap Loop(q_2)$, and $w_r \in \mathscr{L}_{q_2} \setminus \mathscr{L}_q$. Let w_l such that $\Delta(i_L, w_l) = q$. Then $w_l, w_m, w_r, w_1 = w_2 = w$ is a witness for hardness.

We next plan to exhibit a witness for hardness for the case where L satisfies Property $\mathcal{P}$, but we first prove that every language satisfying property $\mathcal{P}$ (whether in $\mathcal{C}_{tract}$ or not) is aperiodic. Let L be a language satisfying Property $\mathcal{P}$, $q \in Q_L$ and w a word in Σ^+. Let also q' denote the state $q' = \Delta_L(q, w^M)$. We denote by q'' the state $\Delta_L(q', w)$. We want to prove that $q' = q''$. By the pigeonhole principle there exists some $k_0 < k_1 \leq M$ such that $\Delta_L(q, w^{k_0}) = \Delta_L(q, w^{k_1})$. We then have $\Delta_L(q', w^k) = q'$ for $k = k_1 - k_0$. Then q' and q'' both loop on w^k, so that $\mathscr{L}_{q'} = \mathscr{L}_{q''}$ by definition of $\mathcal{P}$, hence $q' = q''$ by minimality. Consequently, L is aperiodic.

Let L be a language that satisfies Property $\mathcal{P}$ (and so in particular is aperiodic), but that does not belong to $\mathcal{C}_{tract}$. By definition of $\mathcal{C}_{tract}$ there exist states q, q_2 and words w_l, w_1, w_2, w_m, w'_r such that $\Delta(i_L, w_l) = q$, $w_1 \in Loop(q)$, $w_2 \in Loop(q_2)$, $\Delta_L(q, w_m) = q_2$, $w'_r \in \mathscr{L}_{q_2}$ and $w_2^M w'_r \notin \mathscr{L}_q$. W.l.o.g. we can suppose that $w_1 = (w'_1)^M$ for some word w'_1. We then claim that $\mathscr{L}_{q'} \subseteq \mathscr{L}_q$ for every q' in $\Delta_L(q, \Sigma^* w_1)$. Indeed, for every $q' \in \Delta_L(q, \Sigma^* w_1)$, there exists some $k > 0$ such that $\Delta_L(q', w_1^k) = q'$, hence q' loops over w_1 by aperiodicity of L. We thus have $w_1 \in Loop(q) \cap Loop(q')$ and therefore $\mathscr{L}_{q'} \subseteq \mathscr{L}_q$ due to Property $\mathcal{P}$.

Let $w_r = w_2^M w'_r$. By definition, $w_m w_2^* w_r \subseteq \mathscr{L}_q$ because $w_r \in \mathscr{L}_{q_2}$. We now prove that $(w_1 + w_2)^* w_r \cap \mathscr{L}_q = \emptyset$, because any word in $(w_1 + w_2)^* w_r$ can be decomposed into uv with $u \in \epsilon + (w_1 + w_2)^* w_1$ and $v \in (w_2)^* w_r$. We recall that $w_r = w_2^M w'_r \notin \mathscr{L}_q$ and L is aperiodic, so that $v \notin \mathscr{L}_q$. Furthermore, we have just proved that $q' = \Delta_L(q, u)$ satisfies $\mathscr{L}_{q'} \subseteq \mathscr{L}_q$. Consequently, $v \notin \mathscr{L}_{q'}$ and $uv \notin \mathscr{L}_q$. Thus, w_l, w_m, w_r, w_1, and w_2 provide a witness for hardness, which concludes the proof of Lemma 3. □

We can now prove our hardness result, by reduction from Vertex-Disjoint-Path, a problem also used in [20] to prove hardness in the particular case of a^*ba^*.

Vertex-Disjoint-Path

Input: A directed graph $G = (V, E)$, four vertices $x_1, y_1, x_2, y_2 \in V$
Question: Are there two disjoint paths, one from x_1 to y_1 and the other from x_2 to y_2?

Lemma 4. *Let L be a regular language that does not belong to $\mathcal{C}_{tract}$. Then, RSPQ(L) is* NP*-hard.*

Proof. Let $L \notin \mathcal{C}_{tract}$. We exhibit a reduction from the Vertex-Disjoint-Path problem to RSPQ(L). According to Lemma 3, L admits a witness for hardness w_l, w_m, w_r, w_1, w_2. By definition we get $w_l(w_1 + w_2)^* w_r \cap L = \emptyset$ and $w_l w_1^* w_m w_2^* w_r \subseteq L$.

We build from G a db-graph G' whose edges are labeled by non empty words. This is actually a generalization of db-graphs. Nevertheless, by adding intermediate vertices, an edge labeled by a word w can be replaced with a path whose edges form the word w.

G' is constructed as follows. The vertices of G' are the same as the vertices of G. For each edge (v_1, v_2) in G, we add two edges (v_1, w_1, v_2) and (v_1, w_2, v_2). Moreover, we add two new vertices x, y and three edges (x, w_l, x_1), (y_1, w_m, x_2) and (y_2, w_r, y). We next prove that RSPQ(L) returns True for (G', x, y) iff Vertex-Disjoint-Path returns True for (G, x_1, y_1, x_2, y_2).

Assume there is a simple L-labeled path p from x to y in G'. By definition of G', this path necessarily goes through the edge (y_1, w_m, x_2) since $w_l(w_1 + w_2)^* w_r \cap L = \emptyset$. Since p is simple, the subpaths from x_1 to y_1 and x_2 to y_2 are disjoint, hence Vertex-Disjoint-Path returns True for (G, x_1, y_1, x_2, y_2). Reciprocally, if Vertex-Disjoint-Path returns True for (G, x_1, y_1, x_2, y_2), there exist disjoint paths from x_1 to y_1 and from x_2 to y_2. By definition these two paths match a word in $(w_1 + w_2)^*$. We can then obtain two disjoint simple paths, one from x_1 to y_1 matching a word in w_1^* and one from x_2 to y_2 matching a word in w_2^*. To obtain those paths we keep the vertices as the original paths, eliminate the loops if

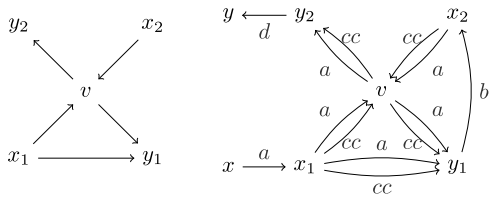


Fig. 1. Reduction for $L = a^*b(cc)^*d$.

there are any, and switch w_1 and w_2 edges where needed: we can always replace a w_1 edge with a w_2 by construction of G since every pair of vertices is connected by both types of edges or none. Concatenating the edge (x, w_l, x_1) with the first path, the edge (y_1, w_m, x_2), the second path and the edge (y_2, w_r, y) provides a simple L-labeled path p from x to y, which concludes our proof. We illustrate in Fig. 1 the reduction for $L = a^*b(cc)^*d$, on an instance (G, x_1, y_1, x_2, y_2), choosing $w_l = w_1 = a$, $w_m = b$, $w_2 = cc$, and $w_r = d$. □

This concludes our proof that languages outside $\mathcal{C}_{tract}$ are intractable. After this negative result, we now focus on the positive result, namely that languages in $\mathcal{C}_{tract}$ admit efficient algorithms.

4. Properties of languages in $\mathcal{C}_{tract}$

The main result of this paper is that for every $L \in \mathcal{C}_{tract}$, RSPQ($L$) $\in$ NL. The algorithm to evaluate efficiently RSPQ(L) exploits a particular kind of pumping argument between strongly connected components of the automaton. This pumping argument proves that if we build carefully a path using the usual reachability algorithm inside the strongly connected components, then we need not care about possible intersections between subpaths corresponding to different components. In this section, we introduce and prove this pumping argument in Lemma 11 through a serie of technical lemmas about the structure of automata that recognize $\mathcal{C}_{tract}$ languages.

4.1. Alternative characterization of $\mathcal{C}_{tract}$

To begin with, we prove that every language from $\mathcal{C}_{tract}$ is aperiodic and deduce an alternative characterization of $\mathcal{C}_{tract}$.

Lemma 5. *Let L be a regular language in $\mathcal{C}_{tract}$. Then L is aperiodic.*

Proof. In the proof of Lemma 3 we defined a property $\mathcal{P}$ and showed that languages satisfying property $\mathcal{P}$ are aperiodic. We show that every $L \in \mathcal{C}_{tract}$ satisfies property $\mathcal{P}$. Let $L \in \mathcal{C}_{tract}$, $q_1, q_2 \in Q_L$ and w satisfy $q_2 \in \Delta_L(q_1, \Sigma^*)$ and $w \in Loop(q_1) \cap Loop(q_2)$. By definition of $\mathcal{C}_{tract}$, $w^M \mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$, hence $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$ because $w \in Loop(q_1)$. □

We then exploit this aperiodicity property to establish the following characterization of $\mathcal{C}_{tract}$, which strengthens the requirements from Definition 1 on the loops of A_L.

Lemma 6. *Let L be a regular language. Then, L belongs to $\mathcal{C}_{tract}$ iff for every pair of states $q_1, q_2 \in Q_L$ such that $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$ and $q_2 \in \Delta_L(q_1, \Sigma^*)$, the following statement holds: $(Loop(q_2))^M \mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$.*

Proof. The (if) implication is immediate by Definition 1. Let us now prove the (only if) implication. Assume $L \in \mathcal{C}_{tract}$. Let $q_1', q_2' \in Q_L$ satisfy $Loop(q_1') \neq \emptyset$, $Loop(q_2') \neq \emptyset$, $q_2' \in \Delta_L(q_1', \Sigma^*)$, and let $w \in Loop(q_2')$. Let also q_3 denote the state $\Delta_L(q_1', w^M)$. Then Definition 1 implies $w^M \mathscr{L}_{q_2'} \subseteq \mathscr{L}_{q_1'}$. Thus, $\mathscr{L}_{q_2'} \subseteq \mathscr{L}_{q_3}$. The crux of the proof is to choose carefully q_1', q_2' and w to exploit the constraints on $\mathscr{L}_{q_3}$.

Let q_1, q_2 be two states such that $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$ and $q_2 \in \Delta_L(q_1, \Sigma^*)$. Let $(v_1, \ldots, v_M)$ be a sequence of words in $(Loop(q_2))^M$ and $q_3 = \Delta_L(q_1, v_1 \ldots v_M)$. We wish to prove $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_3}$.

For some i, j, $0 \leq i < j \leq M$, we get $\Delta_L(q_1, v_1 \ldots v_i) = \Delta_L(q_1, v_1 \ldots v_j)$, using the convention $\Delta_L(q_1, v_1 \ldots v_i) = q_1$ for $i = 0$. Let $u_1 = v_1 \ldots v_i$, $u_2 = v_{i+1} \ldots v_j$ and $u_3 = v_{j+1} \ldots v_M$. Let $q_4 = \Delta_L(q_1, u_1)$. We claim that $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_4}$. The result then follows from $\mathscr{L}_{q_2} = u_3^{-1}\mathscr{L}_{q_2} \subseteq u_3^{-1}\mathscr{L}_{q_4} = \mathscr{L}_{q_3}$. To prove the claim, let $w = u_1 u_2^M$ and $q_5 = \Delta_L(q_1, w^M)$. As $\Delta_L(q_1, w^M) = q_5$ and $w \in Loop(q_2)$, we get $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_5}$ through Definition 1 with q_1, q_2 and w. Furthermore, u_2 belongs to $Loop(q_5)$ because L is aperiodic. To conclude the proof, we observe that $\mathscr{L}_{q_5} \subseteq \mathscr{L}_{q_4}$, by Definition 1 with q_5, q_4 and u_2, and because $\Delta_L(q_4, u_2^M) = q_4$ and $u_2 \in Loop(q_5)$.[1] □

[1] This last application of Definition 1 corresponds actually to observing that every language in $\mathcal{C}_{tract}$ satisfies property $\mathcal{P}$ from Lemma 3.

4.2. Technical lemmas on the components of A_L

In this section, we show properties about the components of $\mathcal{C}_{tract}$ languages. Notice that states in a component are mutually reachable, but not reachable from states in other components that they can reach themselves. From now on, and until the end of the section, we fix a language $L \in \mathcal{C}_{tract}$. We introduce in Lemmas 9 and 11 the pumping argument that we exploit in the algorithm to compute a simple path. In the other lemmas we prove auxiliary results, based on the decomposition of the automaton in strongly connected components. We prove that components of languages in $\mathcal{C}_{tract}$ are very particular, in the sense that every word staying long enough in the component is synchronizing. A preliminary lemma shows that two distinct states q_1 and q_2 in the same component cannot loop on the same word.

Lemma 7. *Let q_1 and q_2 be two states belonging to the same component of A_L. If $Loop(q_1) \cap Loop(q_2) \neq \emptyset$, then $q_1 = q_2$.*

Proof. Let q_1, q_2 as above, and let w a word in $Loop(q_1) \cap Loop(q_2)$. According to Definition 1, $w^M \mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$, hence $\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$ since $w \in Loop(q_1)$. By symmetry, $\mathscr{L}_{q_2} = \mathscr{L}_{q_1}$, which implies $q_2 = q_1$. □

The next two lemmas characterize the internal language of a component.

Lemma 8. *Let C be a component of A_L, $q_1, q_2 \in C$ and $a \in \Sigma$. Then $\Delta_L(q_1, a) \in C$ iff $\Delta_L(q_2, a) \in C$.*

Proof. Let $q_1 \neq q_2$ two states in the same component C. Let a satisfy $\Delta_L(q_1, a) \in C$. Let also $w \in Loop(q_1) \cap a\Sigma^*$ and $q_3 = \Delta_L(q_2, w^M)$: a and w necessarily exist because C is the strongly connected component of q_1 and q_2. We next prove that q_3 belongs to C: by our definition of C, this implies $\Delta_L(q_2, a) \in C$. As L is aperiodic, $w \in Loop(q_3)$, and consequently, $w^M \mathscr{L}_{q_3} \subseteq \mathscr{L}_{q_1}$ by Definition 1. Furthermore, $w^M \mathscr{L}_{q_1} \subseteq \mathscr{L}_{q_2}$ also by Definition 1. Hence $\mathscr{L}_{q_3} \subseteq \mathscr{L}_{q_1}$ and $\mathscr{L}_{q_1} \subseteq (w^M)^{-1} \mathscr{L}_{q_2} = \mathscr{L}_{q_3}$. Thus, $\mathscr{L}_{q_1} = \mathscr{L}_{q_3}$ and, by minimality of A_L, $q_1 = q_3$, so that $q_3 \in C$. □

Notation 1. We denote the internal alphabet of a component C of A_L by $\Sigma_C = \{a \in \Sigma : \exists q_1, q_2 \in C. \Delta_L(q_1, a) = q_2\}$.

As a direct consequence of Lemma 8 we get:

Lemma 9. *Let C be a component of A_L, $q \in C$ and $w \in \Sigma^*$. Then $\Delta_L(q, w) \in C$ iff $w \in (\Sigma_C)^*$.*

Finally, we prove that inside a component, every word with length at least M^2 is synchronizing. This result is the core of our pumping argument between strongly connected components as exposed in Lemma 11.

Lemma 10. *Let C be a component of A_L, Σ_C be the internal alphabet of C, q_1, q_2 be two states of C and $w \in (\Sigma_C)^{M^2}$. Then, $\Delta_L(q_1, w) = \Delta_L(q_2, w)$.*

Proof. Assume that $w = a_1 \dots a_{M^2}$. For each i from 0 to M^2 and $\alpha = 1, 2$, let $q_{\alpha,i} = \Delta_L(q_\alpha, a_1 \dots a_i)$. Since there are at most M^2 distinct pairs $(q_{1,i}, q_{2,i})$, there exist i, j, with $i < j$ such that $q_{1,i} = q_{1,j}$ and $q_{2,i} = q_{2,j}$. By Lemma 9, $q_{1,i}, q_{2,i} \in C$. Let $w' = a_{i+1} \dots a_j$. We have $w' \in Loop(q_{1,i}) \cap Loop(q_{2,i})$, hence $q_{1,i} = q_{2,i}$ by Lemma 7. As a consequence, $\Delta_L(q_1, w) = \Delta_L(q_2, w)$. □

Notice that the above lemma still holds for $w \in (\Sigma_C)^{M^2} \Sigma_C^*$. Here and thereafter, we fix the constant $N = 2M^2$.

Lemma 11. *Let q_1, q_2 be two states such that $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$, and $q_2 \in \Delta_L(q_1, \Sigma^*)$. Let C be the component that contains q_2 and Σ_C be the internal alphabet of C. Then, $\mathscr{L}_{q_2} \cap (\Sigma_C)^N \Sigma^* \subseteq \mathscr{L}_{q_1}$.*

Proof. Let $w \in \mathscr{L}_{q_2} \cap (\Sigma_C)^N \Sigma^*$. There are some words $u, v \in (\Sigma_C)^{M^2}$, $w' \in \Sigma^*$ such that $w = uvw'$. By Lemma 9 and the Pigeonhole Principle, there exist a state $q_3 \in C$ and $M + 1$ non-empty words $v_1, \dots, v_{M+1}$ such that $v = v_1 \dots v_{M+1}$ and $\Delta_L(q_2, uv_1 \dots v_i) = q_3$ for every $i \in [M]$. Therefore, $w \in uv_1(Loop(q_3))^{M-1} v_{M+1} w'$. By Lemma 10, $\Delta_L(q_3, uv_1) = \Delta_L(q_2, uv_1) = q_3$. Thus, w belongs to both $(Loop(q_3))^M v_{M+1} w'$ and $\mathscr{L}_{q_3}$. By Lemma 6, $w \in \mathscr{L}_{q_1}$. □

Or main result focuses on data complexity and therefore assumes the language (hence N) is constant. Yet the complexity will be exponential in N therefore we next prove, for the sake of efficiency, that we can take $N = M$ in Lemma 11.

Lemma 12. *Let q_1, q_2 be two states such that $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$, and $q_2 \in \Delta_L(q_1, \Sigma^*)$. Let C be the component that contains q_2 and Σ_C be the internal alphabet of C. Then, $\mathscr{L}_{q_2} \cap (\Sigma_C)^M \Sigma^* \subseteq \mathscr{L}_{q_1}$.*

Proof. Lemma 10 shows that all words of length at least M^2 are synchronizing inside a component. A straightforward partition-refinement argument [25] shows that for every k, k' and every k-states DFA, if words of length k' are synchronizing, then words of length k are already synchronizing. This shows that words of length $|C|$ are synchronizing inside a component of A_L, hence it is enough to assume $w \in (\Sigma_C)^{|C|}$ in Lemma 10.

With the assumptions of Lemma 11, let $w = uv \in \mathscr{L}_{q_2}$ such that $u \in (\Sigma_C)^M$. As u synchronizes C, $\Delta(q_2, u^{M+1}v) = \Delta(q_2, uv) \in F$. If $\Delta(q_1, u) \in C$ then $\Delta(q_1, u^{M+1}v) \in F$ since $L \in \mathcal{C}_{tract}$. Furthermore, $\Delta(q_1, uv) = \Delta(q_1, u^{M+1}v)$ as u synchronizes C. As a consequence, $\Delta(q_1, uv) \in F$.

Otherwise the automaton avoids C while reading u from q_1 and as a consequence there exist C', u_1, u_2, u_3 satisfying the following 4 conditions: $u = u_1u_2u_3$, $\Delta(q, u_1) \in C'$, $u_2 \in (\Sigma_{C'})^{|C'|}$ and $|u_3| \geq |C|$. Then $\Delta(q_2, u_1u_2^N u_3 v) = \Delta(q_2, uv) \in F$ because u_3 synchronizes C. By Lemma 11, this implies $\Delta(q_1, u_1u_2^N u_3 v) \in F$. As u_2 synchronizes C', $\Delta(q_1, uv) = \Delta(q_1, u_1u_2^N u_3 v)$ and therefore belongs to F. In both cases, $w \in \mathscr{L}_{q_1}$, which concludes our proof. □

5. Computing RSPQ(L) for L in $\mathcal{C}_{tract}$

In this section, we describe a polynomial algorithm that computes RSPQ(L) when L belongs to $\mathcal{C}_{tract}$. A (non-necessarily simple) L-labeled path between two points could be computed incrementally using dynamic programming: we only need to record the last vertex in the (partial) path together with the corresponding state. But this approach is not adequate to build a *simple* path, as we need to memorize all the vertices in the path to check the absence of loops. The approach may thus lead to consider an exponential number of paths.

Nevertheless, we will show that in the case where L belongs to $\mathcal{C}_{tract}$, we can identify a constant number of vertices that suffice to check if the path is (or can be transformed into) a simple path labeled with L. Section 5.1 defines path summaries that record these "critical" vertices, while Section 5.2 explains how we can restore any shortest simple L-labeled path from its summary. Finally, Section 5 presents our algorithm for RSPQ, which enumerates all candidate summaries in logarithmic space, until it finds a candidate summary from which it can restore a simple L-labeled path.

5.1. Defining summaries

Roughly speaking, the idea of a summary is to keep only a bounded number of vertices of p, that depends only on L. Notice that in a run of A_L over p, states of the same component appear consecutively. Indeed, if one leaves a component of A_L, one cannot re-enter it later. Inspired by Lemma 12, we will only record the first and the M last vertices having their state in C, for each component C of A_L. When the number of such vertices is greater than $M+1$, we replace the path between the first vertex and the M last ones by a cut symbol cut_C. This symbol intuitively represents a Σ_C^*-labeled path that has been cut from the path. More formally, a summary is defined as follows.

Definition 3 *(Long run components).* Let $p = (v_1, a_1, \ldots, a_m, v_{m+1})$ be a path and let ρ be the run of L over p. A long run component of p is a component C of A_L such that there are at least $M+2$ vertices v in p such that $\rho(v) \in C$. We denote by $C_1, \ldots, C_l$ the long run components of p (the sequence is sorted by order of appearance in p). For each integer $i \in [l]$, Σ_{C_i} is the internal alphabet of C_i, $left_i$ is the first vertex v_j of p such that $\rho(v_j) \in C_i$ and $right_i$ is the last vertex v_j of p such that $\rho(v_j), \ldots, \rho(v_{j+M}) \in C_i$.

Definition 4 *(Cut symbols and summary).* We introduce a new "cut" symbol cut_C for each component C of A_L. The set of all cut symbols is denoted by *Cuts*. Let $p = (v_1, a_1, \ldots, a_m, v_{m+1})$, ρ a run over p, and $(C_i, \Sigma_{C_i}, left_i, right_i)_{i \in [l]}$ be as stated in Definition 3. The summary S of the path p (w.r.t. A_L) is the sequence obtained from p by replacing, for each $i \in [l]$ the subpath $p[left_i, right_i]$ by the sequence $(left_i, cut_{C_i}, right_i)$.

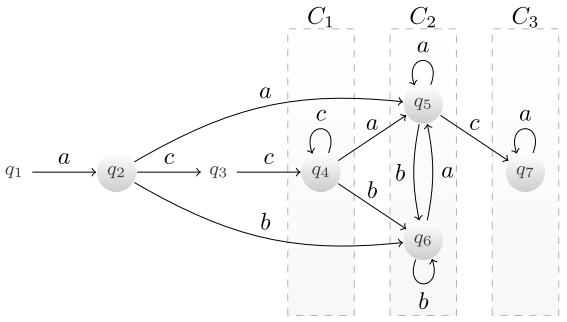
Example 2 depicts a path together with its summary.

Example 2. Fig. 2a represents the minimal DFA for $L = a(c^{\geq 2} + \epsilon)(a+b)^*(ac)?a^*$ (we did not represent the sink state). This automaton can loop in three strongly connected components: $C_1 = \{q_4\}$, $C_2 = \{q_5, q_6\}$, and $C_3 = \{q_7\}$. The accepting states are q_2, q_4, q_5, q_6, and q_7. For this automaton, we observe that Lemma 12 still holds after replacing exponent M with 2. Consequently, we shall pretend that $M = 2$ when defining the long run components and summary in our example. This means we only store the first and last 3 (instead of 8) vertices of each component, which will simplify our illustrative example.

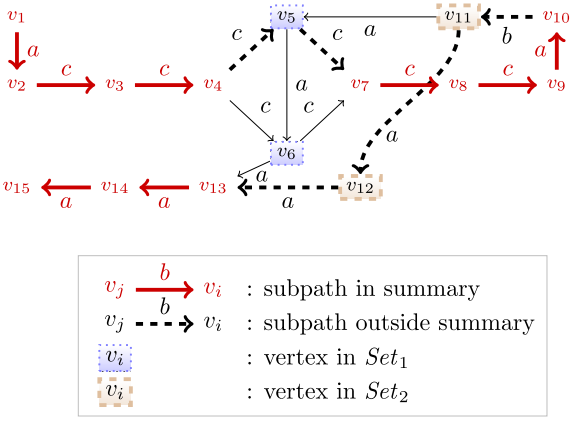
Let us consider the path p_1 illustrated in Fig. 2b with thick edges. Fig. 2c details the run over this path.

We observe that p_1 is a simple L-labeled path. The summary S of p_1 is obtained by removing the second (resp. second and third) vertex with state in C_1 (resp. C_2). The vertices on a white background in Fig. 2c are thus eliminated from the summary and replaced with their respective cut symbol cut_{C_1} and cut_{C_2}:

$$S = (v_1, a, v_2, c, v_3, c, v_4, cut_{C_1}, v_7, c, v_8, c, v_9, a, v_{10}, cut_{C_2}, v_{13}, a, v_{14}, a, v_{15}).$$



(a) Minimal DFA for $L = a(c^{\geq 2} + \epsilon)(a + b)^*(ac)?a^*$



(b) A graph. A shortest simple L-labeled path p_1 (thick) from v_1 to v_{15} is illustrated with thick edges. Dashed edges are eliminated in its summary (see Example 2).

v_1	v_2	v_3	v_4	v_5	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
q_1	q_2	q_3	q_4	q_4	q_4	q_4	q_4	q_5	q_6	q_5	q_5	q_5	q_5
			run in C_1					run in C_2					

(c) Run of A over p_1.

Fig. 2. Components, summaries and safe completions.

We also observe that in a summary, all cut symbols are clearly distinct by definition of strongly connected components. A summary therefore contains at most $M(M+2) = O(M^2)$ elements (vertices, labels and cut symbols), which is constant if L is fixed. As a consequence, each summary can be represented with a logarithmic number of bits.

We next define a candidate summary as an alternative sequence of vertices and symbols or cut symbols of the form above.

Definition 5 *(Candidate summary).* We define as a candidate summary S any sequence of vertices and labels of the form above; $S = (v_1, \alpha_1, \ldots, \alpha_m, v_{m+1})$ where $\alpha_i \in \Sigma \cup Cuts$ for every $i \in [m]$, all cut symbols are distinct, and $m \leq M$. Similarly to Definition 3, we denote by $cut_{C_1}, \ldots, cut_{C_l}$ the sequence of cut symbols appearing in S. Furthermore we define, for each $i \in [l]$, $left_i$ (resp. $right_i$) as the vertex at left (resp. right) of cut_{C_i} in p.

Every summary is clearly a candidate summary but a candidate summary needs not be the summary of any L-labeled path, let alone a simple one.

Definition 6. A path p obtained by replacing each subsequence $(left_i, cut_{C_i}, right_i)$ with a simple $\Sigma^*_{C_i}$-labeled path from $left_i$ to $right_i$ is called a *completion* of the candidate summary S.

By definition, completion and summary are inverse operations in the following sense:

Lemma 13. *Let S be the summary of an L-labeled path p and let p' be a completion of S. Then, p' is an L-labeled path with summary S.*

For each candidate summary S that happens to be a summary, we could compute in NL a completion of S. For instance, this can be done by repeatedly using a reachability oracle. By definition, this completion p is an L-labeled path from a summary S. However, the path p is not necessarily simple, even if S is the summary of a simple path. The reason is that the paths $(p_i)_{i\in[l]}$ we have built between each $left_i$ and $right_i$ are not necessarily disjoint.

5.2. Safe completions

The completion of a summary needs not be a simple path. We therefore define in this section *safe* completions such that safe completions are always simple L-labeled paths, and reciprocally every shortest simple L-labeled paths is a safe completion of its summary.

For that purpose, we will define *local domains* $Set_1, \ldots, Set_l$ which are disjoint sets of vertices from G. For each $i \in [l]$, the safe completions require the path p_i between $left_i$ and $right_i$ to be Set_i-restricted. Consequently, these paths will be disjoint. To guarantee that reciprocally we can find a safe completion for the summary of a shortest simple L-labeled path, we define Set_i to the set of vertices that might occur on a *shortest* $\Sigma^*_{C_i}$-labeled path from $left_i$ to $right_i$ that avoids all Set_j $(j < i)$:

Definition 7 *(Local domains).* Let S be a candidate summary. We denote by $(C_i, left_i, right_i)_{i\in[l]}$ its components as stated in Definition 5, and denote by $V(S)$ the set of vertices appearing in S. We define the local domains Set_i recursively for each i from 1 to l. The set Set_i is defined as a subset of $V_i = V \setminus (V(S) \cup \bigcup_{j<i} Set_j)$, as follows. If there is no V_i-restricted $\Sigma^*_{C_i}$-labeled simple path p from $left_i$ to $right_i$, then $Set_i = \emptyset$. Otherwise, we denote by k_i the length of the shortest such path and define Set_i as the set of vertices y in V_i that can be reached from $left_i$ by a V_i-restricted $\Sigma^*_{C_i}$-labeled path p of length at most $k_i - 1$.

The following lemma is a direct consequence of Definition 7.

Lemma 14. *Let S be a candidate summary. Then all sets $V(S)$, $(Set_i)_{i\in[l]}$ from Definition 7 are disjoint.*

Definition 8 *(Safe completion).* Let p be a path with label in L and summary S. We qualify p as *safe* completion of summary S if the following two conditions are satisfied: (a) all vertices appearing in S are distinct and (b) for every $i \in [l]$, the path $p[left_i, right_i]$ is simple and Set_i-restricted.

By the definition and Lemma 14, a safe completion is necessarily simple.

Lemma 15. *Let S be a candidate summary. Every safe completion of S is a simple L-labeled path.*

Example 3. The path p_1 defined in Example 2 is a safe completion, since $Set_1 = \{v_5, v_6\}$ and $Set_2 = \{v_{11}, v_{12}\}$, as illustrated in Fig. 2b. The definition of safe completions guarantees the paths replacing cut_{C_1} and cut_{C_2} are disjoint. Indeed we can check that $\{v_5\} \cap \{v_{11}, v_{12}\} = \emptyset$.

Being a safe completion is clearly more restrictive than being a simple path. However, it turns out that shortest simple paths are safe completions, as shown below. That means that the existence of a simple path is equivalent to the existence of a safe completion.

Lemma 16. *Let (G, x, y) be a RSPQ(L) instance. Every shortest simple L-labeled path from x to y is a safe completion of its summary.*

Proof. Let $p = (v_1, a_1, \ldots, a_m, v_{m+1})$ be a shortest simple L-labeled path from x to y. Assume that p is not a safe completion. That means there is some i_0 and vertex v between $left_{i_0}$ and $right_{i_0}$ such that $v \notin Set_{i_0}$. Equivalently, if p is not a safe completion, then there exists some i and some vertex v on the path that satisfy one of the following two properties $\mathcal{P}_1(i)$ or $\mathcal{P}_2(i)$: (1) v satisfies $\mathcal{P}_1(i)$ if there is $j > i$ such that v is visited between $left_j$ and $right_j$, and $v \in Set_i$ (2) v satisfies $\mathcal{P}_2(i)$ if no vertex v' satisfies $P_1(i)$, v is visited between $left_i$ and $right_i$, and $v \notin Set_i$. Intuitively, case (2) covers the case where the path between $left_i$ and $right_i$ stays within V_i but fails to stay within Set_i.

We choose a minimal such i. For each of the two cases, we will construct a path p' shorter than p from x to y. We then prove that p' is an L-labeled simple path, which contradicts our assumption that p is the shortest such path.

Case (1): let v be a vertex satisfying property $\mathcal{P}_1(i)$. Then, by definition of Set_i, there is a Set_i-restricted $\Sigma^*_{C_i}$-labeled simple path sp from $left_i$ to v that is shorter than the subpath $p[left_i, right_i]$ and, consequently, shorter than $p[left_i, v]$. Let p' be the path obtained from p by replacing $p[left_i, v]$ with sp. This path p' is shorter than p. For the remainder of the proof we assume that v is the last visited vertex in p satisfying property $\mathcal{P}_1(i)$ and we define p' accordingly.

Case (2): let v' be a vertex satisfying property $\mathcal{P}_2(i)$. There is a Set_i-restricted L-labeled simple path sp between $left_i$ and $right_i$ that is shorter than $p[left_i, right_i]$. We choose p' as the path obtained from p by replacing $p[left_i, right_i]$ with sp. Furthermore, for homogeneity of the proof, we define $j = i$ and $v = right_i$ (v is not the vertex satisfying property $\mathcal{P}_2(i)$).

The remainder of the proof is common to the two cases. We need to prove that p' is a simple L-labeled path. We first prove that p' is an L-labeled path. Let ρ' be the run of L over p'. Let w be the word formed by the labels of the subpath $p[v, y]$. We know that $w \in \mathscr{L}_{\rho(v)}$ since p is an L-labeled path. We will show using Lemma 11 that $w \in \mathscr{L}_{\rho'(v)}$. By definition, $\rho'(\mathit{left}_i)$ belongs to the component C_i and the path $s = p'[\mathit{left}_i, v]$ is $\Sigma_{C_i}^*$-labeled, hence $\rho'(v) \in C_i$ by Lemma 9. Furthermore $\rho(v) \in C_j$ and C_j is reachable from C_i. In addition, by definition of a summary, there are at least $M+1$ vertices v'' of p after v (including v) such that $\rho(v'') \in C_j$, and therefore the M labels following vertex v in p belong to Σ_{C_j} by Lemma 9 again. The definition of components C_i, C_j guarantees that q_1, q_2 admit loops. We have thus proved that $q_1 = \rho'(v)$ and $q_2 = \rho(v)$ meet all requirements for Lemma 12, which implies $w \in \mathscr{L}_{\rho'(v)}$. Consequently, p' is an L-labeled path.

We now prove that p' is simple. Since p is simple, it suffices to prove that the vertices in sp (between left_i and v) are disjoint with other vertices in p'. Notice that all intermediate vertices of sp belong to Set_i. By minimality of i, for all $i' < i$, the vertices between $\mathit{left}_{i'}$ and $\mathit{right}_{i'}$ belong to $\mathit{Set}_{i'}$ and, since $\mathit{Set}_{i'}$ and Set_i are disjoint (Lemma 14), do not belong to Set_i. Consequently, there is no vertex v' before left_i such that v' belong to Set_i. By construction, in the two cases (1) and (2), there is no vertex v' after v such that v' belongs to Set_i. This concludes the proof. □

We observe that many of our summary and completion definitions were actually tailored for this Lemma 16: our proof by contradiction explains for instance why our definition of Set_i guarantees the existence of a safe completion. And our application of Lemma 12 in that proof explains why our summaries feature $M+2$ vertices of each component.

We next show how a summary admitting a safe completion can be completed in logarithmic space into a simple path.

5.3. An algorithm for RSPQ via safe completions

We next introduce Algorithm 1, show that it computes simple L-labeled path from x to y, and analyze its complexity.

Algorithm 1 Algorithm for RSPQ(L).

```
(* Fixed parameter: A_L.      Input: graph G, vertices x, y. *)
 1: for all candidate summary S do
 2:     if SAFE-COMPLETION(S)≠∅ then
 3:         return SAFE-COMPLETION(S)

 4: procedure SAFE-COMPLETION(S)
    (* Denoting by (C_i, Σ_{C_i}, left_i, right_i)_{i∈[l]} the components of S *)
 5:     if the vertices of S are not distinct then
 6:         return ∅
 7:     for all i ∈ [l] do
 8:         Compute a shortest Set_i-restricted Σ*_{C_i}-labeled path from left_i to right_i
 9:         if there is such a path then
10:             replace in S the sequence (left_i, cut_{C_i}, right_i) by that path
11:         else return ∅
12:     if the resulting path p is L-labeled and has summary S then
13:         return p
14:     else return ∅
```

Theorem 1. *Algorithm 1 returns a simple L-labeled path from x to y if there is one. The algorithm can be implemented in* NL.

We first prove that local domains Set_i can be computed in logarithmic space.

Lemma 17. *Let L be a fixed language in $\mathcal{C}_{tract}$. The following problem P_{Set} is in* NL. *Given an instance (G, x, y) of RSPQ(L), a candidate summary S, a vertex z and an integer i, decide whether $z \in \mathit{Set}_i$.*

Proof. For each $k \geq 0$, let P_{Set}^k be the decision problem P_{Set} with the restriction $i \leq k$: $(G, x, y, S, z, i) \in P_{Set}^k$ iff $(G, x, y, S, z, i) \in P_{Set}$ and $i \leq k$.

Clearly, the number l of cuts in a summary S as in Definition 4, is bounded by the number K of strongly connected components of L. Consequently, $P_{Set} = P_{Set}^K$. Notice that K is a constant that does not depend on the instance. Let us prove, that $P_{Set}^k \in$ NL for each $k \in [0, K]$. The proof is by induction on k. If $k = 0$, P_{Set}^0 always returns False because Set_i is not defined for $i = 0$. So P_{Set}^0 is trivially in NL. Assume, by induction, that $P_{Set}^k \in$ NL. It suffices to show that there is an NL-algorithm for P_{Set}^{k+1} using P_{Set}^k as oracle. Since $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$ (Lemma 2), this implies that $P_{Set}^{k+1} \in$ NL. Let (G, x, y, S, z, i) be an instance of P_{Set}^{k+1}. If $i \leq k$, we return the same answer as the oracle P_{Set}^k. If $i = k+1$, using the definition and notations of Set_{k+1}, the problem essentially boils down to computing the distances between the vertices left_{k+1} and z on one hand, left_{k+1} and right_{k+1} on the other hand in the graph $G' = (V', E')$ where $V' = V_{k+1} \cup \{\mathit{left}_{k+1}, \mathit{right}_{k+1}\}$ and E' is the set of $\Sigma_{C_{k+1}}$-labeled edges of $G[V']$. In order to remain within logarithmic space, the graph G' is not stored in memory but simulated. The distance computation can thus clearly be performed in non deterministic log-space using the oracle P_{Set}^k. □

Lemma 18. *Let L be a fixed language in $\mathcal{C}_{tract}$, (G, x, y) an instance of RSPQ(L) and S a summary. Procedure* SAFE-COMPLETION*(S) from Algorithm 1 returns in* NL *a safe completion p from x to y if there is any, and returns $\emptyset$ otherwise.*

Proof. The correctness of the procedure is immediate as it matches the definition of safe completions. We still have to check the complexity. To achieve logarithmic space, we do not store the Set_i in memory: we only need to check on-the-fly if a given vertex belongs those sets, using Lemma 17. This proves the algorithm can easily be implemented in NL. □

We can now prove Theorem 1. By Lemmas 15 and 16, there exists a safe completion from x to y if and only if there is a simple path from x to y. Lemma 16 implies that Algorithm 1 identifies a safe completion if there is one, so the algorithm is correct. The complexity is in NL as candidate summaries have constant size and therefore can be enumerated in NL, whereas completions are computed in NL according to Lemma 16. This concludes the proof of Theorem 1.

Notice that we can easily adapt Algorithm 1 so that it outputs a shortest path for positive instances. The main theorem summarizes our results, combining Lemma 4 with Theorem 1.

Theorem 2. *Let L be a regular language. Then, RSPQ(L) is in* NL *if $L \in \mathcal{C}_{tract}$ and is* NP*-complete otherwise.*

5.4. Towards a complete classification

We have partitioned RSPQ(L) problems into NL and NP-complete problems. We next refine the classification within the class of NL problems.

Lemma 19. *For every regular language L, RSPQ(L) $\in$ AC^0 if L is finite, otherwise RSPQ(L) is* NL*-hard.*

The proof is based on a reduction from the following NL-complete problem [24].

Reachability
Input: A directed graph G and two vertices x, y in G
Question: Is there a path from x to y?

Proof. (Membership) For a finite language L, RSPQ(L) can easily be defined by a first-order formula, more precisely a disjunction of conjunctive formulas. Consequently, RSPQ(L) $\in$ AC^0 [12].

(Hardness) We exhibit a reduction from Reachability. Let L be an infinite regular language. By the Pumping Lemma, there exist non empty words u, v, w such that $uv^*w \subseteq L$. We build a db-graph G' from G by first labeling every edge of G with v, and then adding two vertices x' and y' with edges (x', u, x) and (y, w, y'). There is a (not necessarily simple) path from x to y in G iff there is an L-labeled simple path from x' to y' in G'. Consequently, RSPQ(L) is NL-hard. □

Our results so far can be summarized in the following trichotomy which refines Theorem 2.

Theorem 3. *Let L be a regular language. The complexity of RSPQ(L) can be determined as follows.*

1. *L is finite: RSPQ(L) $\in$ AC^0;*
2. *$L \in \mathcal{C}_{tract}$ and L is infinite: RSPQ(L) is* NL*-complete;*
3. *$L \notin \mathcal{C}_{tract}$: RSPQ($L$) is* NP*-complete.*

6. Variations

In this Section, we analyze the computational complexity of variations of the main problem: allowing ϵ-edges, finding shortest paths in weighted graphs and finding paths that minimize repetitions of vertices.

6.1. Db-graphs with ϵ-edges

We observed in Lemma 4 that RSPQ(L) does not become harder if we allow edges labeled by non empty words. However, we have not discussed yet the complexity of RSPQ(L) when we allow edges labeled by ϵ. We name RSPQ$^\epsilon(L)$ this variation of RSPQ(L). We show that the class of tractable languages RSPQ$^\epsilon(L)$ corresponds to the languages closed under subwords, and consequently is a strict subset of $\mathcal{C}_{tract}$.

Theorem 4. *Let L be a regular language. The complexity of RSPQ$^\epsilon(L)$ can be determined as follows.*

1. *L is empty: RSPQ$^\epsilon(L)$ is trivial i.e. does not contain positive instances;*

2. *L is non empty and closed under subwords: RSPQ$^\epsilon$(L) is* NL*-complete;*
3. *L is not closed under subwords: RSPQ$^\epsilon$(L) is* NP*-complete.*

Proof. 1) is straightforward.

2) The membership in NL is straightforward: for any subword closed language L, there exists a simple L-labeled path from x to y in G if and only if there exists an L-labeled path (simple or not) from x to y in G. This can obviously be checked in NL. We next prove that RSPQ$^\epsilon(L)$ is NL-hard. Since L is not empty, let w be a word in L. The reduction is identical to the reduction of Lemma 19. We choose $u = v = \epsilon$ and w as defined above.

3) Since L is not closed under subwords, there exist words u, v, w such that $uvw \in L$ and $uw \notin L$. The reduction is identical to the reduction of Lemma 4. We choose $w_1 = w_2 = \epsilon$, $w_l = u$, $w_m = v$ and $w_r = w$. □

6.2. Shortest path in weighted graphs

In this section, we consider the following problem:

weighted-RPQ(L, C)
Input: A db-graph G, two vertices x and y and a positive weight function W over the edges of G
Objective: Find a simple path L-labeled path p from x to y with minimal weight $W(p)$ if such a path exists

The problem can be solved with a straightforward generalization of Algorithm 1. Edges of weight 0 are the only ones that require special care. Indeed, in the proof of Lemma 16, we start from a shortest path p and build a path p' shorter than p with the goal of showing a contradiction. When both p and p' can have weight 0, there may be no contradiction. To overcome this problem, one can for instance define weights over a product order to count the number of edges with weight 0: if the weights are originally defined over some ordered set W, we assign to each edge e the weight $(w, 0)$ if e has weight $w > 0$, and $(w, 1)$ otherwise. The weights of the edges along a path are summed componentwise, and paths are compared based on a lexicographic order. This means that the number of edges with weight 0 along a path is solely used to break ties.

6.3. Paths that minimize repetitions of vertices

We next show that our technique and results extend to generalizations of the simple path problem that do not forbid but only limit repetitions of vertices along the path.

Definition 9. Let p be a path and $v_1, \ldots, v_m$ be the sequence of vertices appearing in p. We define $rep(p)$ as the multiset keeping only vertices v_i for which there is v_j, $j < i$ such that $v_i = v_j$. We denote by $nbrep(p)$ the cardinality of $rep(p)$.

Clearly p is a simple path if and only if $nbrep(p) = 0$. We consider the following problem:

rep-RPQ(L)
Input: A db-graph G, two vertices x and y and an integer $k \geq 0$
Question: Is there an L-labeled path p from x to y such that $nbrep(p) \leq k$?

Lemma 20. *Let $L \in \mathcal{C}_{tract}$. Then there is a constant K_L such that for every db-graph G and L-labeled path p from x to y in G, there exists an L-labeled path p' from x to y with $nbrep(p') \leq K_L$.*

Proof. Let p be an L-labeled path from x to y that minimizes $nbrep(p)$. We will prove that all occurrences of a given vertex v appear in the summary S except at most one. This is sufficient for the proof since the size of the summary S is bounded by a constant. Assume, for the sake of contradiction, that p is of the form $(v_1, a_1, \ldots a_l, v_{l+1})$ and contains two occurrences v_i, v_j of the same vertex such that v_i and v_j are not in the summary of p. Let $p' = (v_1, \ldots v_i, a_j, v_{j+1}, \ldots, v_{l+1})$. Then, $nbrep(p') < nbrep(p)$, and p' is an L-labeled path by Lemma 11. □

Theorem 5. *Let $L \in \mathcal{C}_{tract}$. Then, rep-RPQ$(L) \in$* NL*.*

Proof. Given a db-graph G and a multiset A of vertices in G, we define the graph $c(G, A)$ as follows. We start from G and for each vertex x in A we add a new vertex in G which is a copy of x i.e. has the same incoming and outgoing edges as x has.

By construction, we have the following property: there exists an L-labeled simple path from x to y in $c(G, A)$ iff there exists an L-labeled path p from x to y in G such that $rep(p) \subseteq A$. Consequently, it suffices to enumerate all multisets A of

Table 1
A family of regular expressions for $\mathcal{C}_{tract}$.

$$\begin{aligned} \Psi_{seq} &::= w + \epsilon \mid A^{\geq k} + \epsilon \mid \Psi_{seq}\Psi_{seq} \\ \Psi_{tr} &::= w\Psi_{seq}w' \mid \Psi_{tr} + \Psi_{tr} \\ &\quad \text{where } w, w' \in \Sigma^*, A \subseteq \Sigma \end{aligned}$$

at most k vertices and check if, for one of them, $c(G, A), x, y$ is a positive instance of RSPQ(L). Thanks to Lemma 20, we do not have to consider multisets of more than K_L vertices.

Moreover, it is easily seen that c is a function in FL. Thus, we obtain the following algorithm for rep-RPQ(L).

Algorithm 2 Algorithm for rep-RPQ(L).

```
for each multiset A of min(k, K_L) vertices do
    if c(G, A), x, y is a positive instance of RSPQ(L) then
        return true
return false
```

□

Other constraints on repetitions may be worth mentioning. It is obvious that Lemma 20 and therefore also Theorem 5 still hold if we define $nbrep(p)$ as the maximal number of occurrences of a vertex in p. Or if we define $nbrep(p)$ as the number of distinct vertices that are repeated (one or several times) in p.

7. Characterization by regular expressions

In this section, we propose two characterizations of $\mathcal{C}_{tract}$ languages. The first in terms of regular expressions and the second in terms of a pumping property. Unlike the other properties discussed before on the minimal DFA of L, the pumping property is expressed directly on the language L. The languages in $\mathcal{C}_{tract}$ are exactly those that can be expressed with an expression in the fragment Ψ_{tr} defined in Table 1. This fragment enforces restrictions on the concatenation of subexpressions: roughly speaking, only expressions of the form $e + \epsilon$ can be concatenated.

Example 4. For instance, the expression $a^*(b^{\geq 2} + \epsilon)c^*$ investigated in Example 1 belongs to the fragment Ψ_{tr} (using notation $c^* = c^{\geq 0}$). Expression $a^*ba^* + (a + b)^*$, on the opposite, does not, but is clearly equivalent to $(a + b)^{\geq 0}$, which does. The following theorem, however, implies that a^*ba^* is not equivalent to any expression from Ψ_{tr}.

Theorem 6. *Let L be a regular language. The three following statements are equivalent:*

1. *A language L belongs to $\mathcal{C}_{tract}$.*
2. *L is recognized by a regular expression in Ψ_{tr}.*
3. *There is an integer $i \geq 0$ such that for all words $w_l, w_m, w_r \in \Sigma^*$ and all non empty words $w_1, w_2 \in \Sigma^+$, $w_l w_1^i w_m w_2^i w_r \in L$ implies $w_l w_1^i w_2^i w_r \in L$.*

Proof. $1 \Rightarrow 2$) We next outline an algorithm to build the regular expression e from A_L. Let $C_1, \ldots, C_l$ be the strongly connected components of L in some topological order. For every $k \in \{0, \ldots, l\}$ and every sequence $1 \leq j_1 < \cdots < j_k \leq l$, we denote by $L[j_1, \ldots, j_k]$ the set of all words from L that stay for at least $2M$ steps in each component $C_{j_1}, \ldots, C_{j_k}$, and stay for at most $2M - 1$ steps after entering in each other component.

Clearly, L is the union of all $L[j_1, \ldots, j_k]$ over all sequences $j_1, \ldots, j_k$. We next show how to build an expression for $L[j_1, \ldots, j_k]$. We denote by $S_1, \ldots, S_k$ the components $C_{j_1}, \ldots, C_{j_k}$ and by $\Sigma_1, \ldots, \Sigma_k$ their alphabet. For any component S_i and state q in S_i, we can easily build an expression H_q with language: $L(H_q) = \{w \in (\Sigma_i)^M \mid \exists q_0 \in S_i, \Delta_L(q_0, w) = q\}$. The rationale for this definition is that words of $(\Sigma_i)^M$ are synchronizing within S_i according to Lemma 12: for all $q, q_1 \in S_i$ and $w \in L(H_q)$, we have $\Delta_L(q_1, w) = q$.

Let $i < k$ and $q \in S_i$. We build an expression W_q for the set of all words w that lead from q to some state of S_{i+1} while respecting the sequence of components. In other words, a word $w = a_1 \ldots a_m$ belongs to $L(W_q)$ iff when we denote by q_j the state $\Delta_L(q, a_1 \ldots a_j)$, the sequence $q_1 \ldots q_m$ satisfies the following properties[2]:

- $q_1 \notin S_i$
- q_m is the first state of the sequence that belongs to S_{i+1}

[2] We require somewhat arbitrarily that the first letter of w lets quit S_i, while the last letter of w lets enter S_{i+1} (i.e., is not in Σ_{i+1}).

- there are at most $2M$ states q_j in the same component of A_L.

Similarly, for $i = k$, we build for any state $q \in S_k$ an expression W_q for the set of all words w that lead from q to some final state while respecting the sequence of components, i.e., satisfying conditions similar to the above ones except that q_m belongs to F_L instead of S_{i+1}. $L(W_q)$ is a finite set of words having length at most $2M^2$.

If i_L belongs to S_1, we define the expression e_{init} as ϵ, otherwise e_{init} is the set of all words that lead from i_L to some state in S_1 while respecting the sequence of components. Rephrased differently, a word $w = a_1 \dots a_m$ belongs to $L(e_{\text{init}})$ iff, when we denote by q_j the state $\Delta_L(i_L, a_1 \dots a_j)$, the sequence $q_1 \dots q_m$ satisfies the following properties:

- q_m is the first state of the sequence that belongs to S_1,
- there are at most $2M$ states q_j in the same component of A_L.

e_{init} is a finite set of words having length at most $2M^2$.

Claim 1: *The expression e'_0 defined by the following equations represents the language $L[j_1, \dots, j_k]$*

$$e'_k = (\Sigma_k)^{\geq M} \cdot \Big(\bigcup_{q \in S_k} H_q \cdot W_q\Big)$$

$$e'_i = (\Sigma_i)^{\geq M} \cdot \Big(\bigcup_{q \in S_i} H_q \cdot W_q\Big) \cdot e'_{i+1} \text{ for all } 1 \leq i < k$$

$$e'_0 = e_{\text{init}} \cdot (\Sigma_1)^{\geq M} \cdot \Big(\bigcup_{q \in S_1} H_q \cdot W_q\Big) \cdot e'_1$$

The language of e'_0 clearly contains $L[j_1, \dots, j_k]$. The converse inclusion follows from Lemma 12, which concludes the proof of Claim 1. We now define the expressions $e_0, \dots, e_k$ recursively as follows (with i ranging from 1 to $k-1$ included, in the second equation):

$$e_k = ((\Sigma_k)^{\geq M} + \epsilon) \cdot \Big(\bigcup_{q \in S_k} H_q \cdot W_q\Big)$$

$$e_i = ((\Sigma_i)^{\geq M} + \epsilon) \cdot \Big(\bigcup_{q \in S_i} H_q \cdot W_q + \epsilon\Big) \cdot e_{i+1}$$

$$e_0 = e_{\text{init}} \cdot ((\Sigma_1)^{\geq M} + \epsilon) \cdot \Big(\bigcup_{q \in S_1} H_q \cdot W_q + \epsilon\Big) \cdot e_1$$

Claim 2: *The language of e_0 contains $L[j_1, \dots, j_k]$ and is contained in L.*

The language of e_0 clearly contains the language of e'_0, hence $L[j_1, \dots, j_k]$ by Claim 1. Let $w \in L(e_0)$. There exist $u_0, v_0, u_1, v_1 \dots, u_n$, and v_n such that

- $w = u_0 v_0 u_1 v_1 \dots u_n v_n$
- $u_0 \in L(e_{\text{init}} \cdot ((\Sigma_1)^{\geq M} + \epsilon))$
- $v_n \in L(\bigcup_{q \in S_k} H_q \cdot W_q)$
- for each $0 \leq i \leq n-1$, $v_i \in L(\bigcup_{q \in S_i} H_q \cdot W_q + \epsilon)$
- for each $1 \leq i \leq n$, $u_i \in L((\Sigma_i)^{\geq M} + \epsilon)$.

Let w' be the word obtained from w by replacing every v_i equal to ϵ with an arbitrary word from $L(\bigcup_{q \in S_i} H_q \cdot W_q)$, and every e'_i equal to ϵ with an arbitrary word from $L((\Sigma_i)^{\geq M})$. Then w' belongs to $L(e'_0)$ and in particular to L. Consequently, w also belongs to L by repeated applications of Lemma 12. As $L(W_q)$ and $L(H_q)$ are finite sets of words for every state q, e_0 belongs to the fragment, which concludes the proof of Claim 2.

$2 \Rightarrow 3$) It is easily seen that languages that satisfy Statement (3) are closed by union. Consequently, we consider a regular expression $\varphi \in \Psi_{tr}$ of the form $\varphi_1 \cdot \dots \cdot \varphi_l$ where φ_1 and φ_l are words and φ_i is a Ψ_{seq}-term for every $i \in \{2, \dots, l-1\}$. For each $i \in [l]$, we denote by L_i the language recognized by φ_i. Let M be the size of φ, defined as the number of states in its Glushkov automaton (i.e., the number of symbols in the expressions when terms of the form $A^{\geq k}$ are expanded). Let u, v, w, w_1, w_2 be words with w_1 and w_2 non empty such that $u w_1^M v w_2^M w \in L$. We want to prove that $u w_1^M w_2^M w \in L$. Using the usual pumping argument for automata, applied to the Glushkov automaton of the expression, one shows easily that there is some term φ_i of the form $A^{\geq n} + \epsilon$ such that $u w_1^M \in L_1 \dots L_i$ (and $w_1^M v w_2^M w \in A^{\geq n} \cdot L_{i+1} \dots L_l$). Notice that φ_i cannot be of the form $w + \epsilon$ because w_1^M necessarily "ends" inside a Kleene-star subexpression by definition of M. Similarly, there is some term φ_j, $j \geq i$ of the form $B^{\geq m} + \epsilon$ such that $u w_1^M v w_2^M \in L_1 \dots L_j$ and $w_2^M w \in L_j \dots L_l$. Thus,

$uw_1^M w_2^M w \in L_1 \dots L_i \cdot L_j \dots L_l$. Furthermore, $L_1 \dots L_i \cdot L_j \dots L_l \subseteq L$: if $i = j$, then $L_i L_j \subseteq L_j$ due to the form of L_i, and if $i < j$, then $\epsilon \in L_{i+1} L_{j-1}$ because every intermediate term can be skipped. As a consequence, $uw_1^M w_2^M w \in L$.

$3 \Rightarrow 1$) Assume that L satisfies Statement (3). Let i be as stated in (3). Let $q_1, q_2 \in Q_L$ such that $Loop(q_1) \neq \emptyset$, $Loop(q_2) \neq \emptyset$ and $q_2 \in \Delta_L(q_1, \Sigma^*)$. Let $w_1, w_2, w_l, w_m, w_r \in \Sigma^*$ such that $w_1 \in Loop(q_1)$, $w_2 \in Loop(q_2)$, $\Delta_L(i_L, w_l) = q_1$, $\Delta_L(q_1, w_m) = q_2$ and $w_r \in \mathscr{L}_{q_2}$. We need to prove that $w_2^M w_r \in \mathscr{L}_{q_1}$. Let $w = w_l w_1^M w_m w_2^M w_r \in L$. We consider two cases: $i \leq M$ and $i > M$.
Case $i \leq M$: $w = w_l w_1^{M-i} w_1^i w_m w_2^i w_2^{M-i} w_r$. By hypothesis, we then have $w_l w_1^{M-i} w_1^i w_2^i w_2^{M-i} w_r = w_l w_1^M w_2^M w_r \in L$. Consequently, $w_2^M w_r \in \mathscr{L}_{q_1}$.
Case $i > M$: By the classical pumping lemma, there exist integers $k, k' > 0$ such that for every $j, j' \geq 0$, the word $w_l w_1^{M+kj} w_m w_2^{M+k'j'} w_r$ belongs to L. Consequently, for every $i_1, i_2 \geq 0$, we can find a word $w' \in w_1^* w_m w_2^*$ such that $w_l w_1^{i+i_1} w' w_2^{i+i_2} w_r \in L$, which by hypothesis implies $w_l w_1^{i+i_1} w_2^{i+i_2} w_r \in L$. Observe that $\Delta(i_L, w_l w_1^{i+i_1}) = q_1$, so that $w_2^{i+i_2} w_r \in \mathscr{L}_{q_1}$ for every $i_2 \geq 0$. By the usual pumping argument, $w_2^M w_r \in \mathscr{L}_{q_1}$. □

The two characterizations of $\mathcal{C}_{tract}$ in Theorem 6 imply the following results.

Corollary 1. *$\mathcal{C}_{tract}$ is closed under intersection, union and word reversal.*

The closure under intersection and union of $\mathcal{C}_{tract}$ is a consequence of the second statement of Theorem 6 and the closure under word reversal is a consequence of the third statement of the same theorem.

An NFA A is pseudo-acyclic if every loop in A is a self-loop (i.e. a transition from a state to itself). Since in the third characterization of $\mathcal{C}_{tract}$ in Theorem 6 expressions under a Kleene star are unions of symbols, we obtain the following result.

Corollary 2. *Let $L \in \mathcal{C}_{tract}$. Then, A is recognizable by a pseudo-acyclic NFA.*

The converse is not true since a^*ba^* is not in $\mathcal{C}_{tract}$ but recognizable by a pseudo-acyclic NFA.

8. Recognition of tractable languages

The following theorem establishes the complexity of deciding if RSPQ(L) is tractable (i.e. deciding if RSPQ(L) can be computed in polynomial time). We consider different representations of L (DFAs, NFAs and regular expressions).

Theorem 7. *Testing whether a regular language L belongs to $\mathcal{C}_{tract}$ is:*

1. NL*-complete if L is given by a DFA;*
2. PSPACE*-complete if L is given by an NFA (resp. a regular expression).*

Before proving this theorem, we need some useful lemmas.

Lemma 21 *(Folklore). There is an* L *transducer that, given two DFAs A_1 and A_2 that respectively recognize the languages L_1 and L_2, returns a DFA that recognizes the language $L_1 \cap L_2$ (resp. $L_1 \cup L_2$, $L_1 \setminus L_2$).*

Proof. The classical construction by product of the two DFAs can be done by an L transducer. □

The next lemma permits to consider only minimal DFAs.

Lemma 22. *Let $\mathcal{L}$ be a class of regular languages, such that we can test in* NL *whether a minimal DFA recognizes a language in $\mathcal{L}$.*

1. *testing whether a DFA recognizes a language in $\mathcal{L}$ is in* NL*;*
2. *testing whether an NFA recognizes a language in $\mathcal{L}$ is in* PSPACE*.*

Proof. 1) We will exhibit an NL transducer t that given a DFA A_L returns a minimal DFA A'_L equivalent to A_L. We construct t as the composition of two transducers t_1 and t_2, where t_1 removes from A_L all states non reachable from the initial state and t_2 merges all Nerode-equivalent states. This composition of two NL transducers is an NL transducer [24].

The log-space algorithm for t_1 uses an oracle for the problem Reachability and the log-space algorithm for t_2 uses an oracle for state equivalence. This problem of deciding given q_1 and q_2, whether $\mathscr{L}_{q_1} = \mathscr{L}_{q_2}$, is actually in NL since it can be reduced to Emptiness using L transducers of Lemma 21.

2) We determinize the automaton and then apply an NL algorithm $\mathcal{A}$ to recognize $\mathcal{L}$ on the deterministic automaton thus obtained. To achieve the polynomial bound on space, the deterministic automaton is actually simulated on the fly instead of stored in memory. □

Lemma 23. *There is an* L *transducer that given a minimal DFA A_L and a state q of A_L computes a DFA that recognizes the language $Loop(q)^M\mathscr{L}_q$.*

Proof. Let $L' = Loop(q)^M\mathscr{L}_q$. We construct a DFA $A_{L'} = (Q_{L'}, i_{L'}, F_{L'}, \Delta_{L'})$ as follows. This construction can clearly be computed by an L transducer.

- $Q_{L'} = Q_L \times [0, M]$;
- $i_{L'} = (q, 0)$;
- $F_{L'} = F_L \times \{M\}$;
- Let $(q_1, i) \in Q_{L'}$ and $a \in \Sigma$. Then $\Delta_{L'}((q_1, i), a) = (\Delta_L(q_1, a), i + 1)$ if $q_1 = q$ and $i < M$. Otherwise, $\Delta_{L'}((q_1, i), a) = (\Delta_L(q_1, a), i)$.

Let us prove that $A_{L'}$ recognizes the language L'. Let $w \in L' = Loop(q)^M\mathscr{L}_q$. We decompose w as $w = w_1w_2$ where $w_1 \in Loop(q)^M$ and w_2 is accepted by A_L starting from q. By construction $\Delta_{L'}((q, 0), w_1) = (q, M)$ and w_2 is accepted by $A_{L'}$ starting from (q, M). Thus, w is accepted by $A_{L'}$.

Let w be a word accepted by $A_{L'}$. By construction, we easily see that w is accepted by A_L starting from q. Additionally, the run of w over A_L contains at least $M + 1$ occurrences of the state q. Consequently, $w = w_1w_2$ where $w_1 \in Loop(q)^M$ and w_2 is accepted by A_L starting from q. Thus $w \in L'$. □

Lemma 24. *We can decide in* NL *whether $Loop(q_2)^M\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$ given a minimal DFA together with two states q_1 and q_2.*

Proof. Combining Lemmas 21 and 23, we obtain an L transducer that, given a minimal DFA A_L and two states q_1, q_2, returns a DFA that recognizes the language $Loop(q_2)^M\mathscr{L}_{q_2} \setminus \mathscr{L}_{q_1}$. We thus reduce our problem to the Emptiness problem that is in NL. □

With the help of those lemmas we finally prove Theorem 7.

Proof. 1) Membership: let A_L be a minimal DFA. The proof is based on the characterization of Lemma 6: for each pair of states q_1, q_2, we check in NL if they admit a loop and if q_2 is accessible from q_1. Then we check $Loop(q_2)^M\mathscr{L}_{q_2} \subseteq \mathscr{L}_{q_1}$, still in NL according to Lemma 24. We thus check in NL whether $L \in \mathcal{C}_{tract}$ when L is given by a minimal DFA which, together with Lemma 22, concludes our proof.

Hardness: we exhibit a reduction from Emptiness. Let $L \subseteq \Sigma^*$ and $L' = 1^+L1^+$ where $1 \notin \Sigma$. Moreover, we assume that $\epsilon \notin L$. Emptiness clearly remains NL-complete with this restriction. Our reduction maps a DFA A_L that recognizes L to a DFA $A_{L'}$ that recognizes L'. First, we prove that $L = \emptyset$ iff $L' \in \mathcal{C}_{tract}$. If $L = \emptyset$ then $L' = \emptyset \in \mathcal{C}_{tract}$. Conversely, assume that $L' \in \mathcal{C}_{tract}$, and let $w \in \Sigma^+$. By construction, $1^i1^i \notin L'$ for every $i \geq 0$. As a consequence, Theorem 6 (using $w_l = w_r = \epsilon$) yields some $i \geq 0$ such that $1^iw1^i \notin L'$. Thus, $w \notin L$, hence L is empty.

It remains to prove that the reduction can be computed with an L transducer. We construct $A_{L'}$ from A_L as follows. We add a state $i_{L'}$ that will be the initial state of $A_{L'}$ and a state $f_{L'}$ that will be the unique final state of $A_{L'}$. $\Delta_{L'}$ is the extension of Δ_L defined as follows:

- $\Delta_{L'}(i_{L'}, 1) = i_{L'}$ and $\Delta_{L'}(i_{L'}, a) = i_L$ for every symbol $a \in \Sigma$.
- For every final state $q \in F_L$, $\Delta_{L'}(q, 1) = f_{L'}$, and $\Delta_{L'}(f_{L'}, 1) = f_{L'}$.

2) The membership is a direct consequence of (1) and Lemma 22. To prove the hardness result we will exhibit a reduction from Universality. Consider the alphabet $\Sigma = \{0, 1, a, b\}$ and two languages $L \subseteq \{0, 1\}^*$ and $L' = (0 + 1)^*a^*ba^* + La^*$. Let us prove that $L = \{0, 1\}^*$ iff $L' \in \mathcal{C}_{tract}$. Assume that $L = \{0, 1\}^*$. Then $L' = (0+1)^*a^*ba^* + (0+1)^*a^* = (0+1)^*a^*(b+\epsilon)a^* \in \mathcal{C}_{tract}$ according to Theorem 6. Conversely, assume $L' \in \mathcal{C}_{tract}$. Let us prove that every word $w \in \{0, 1\}^*$ belongs to L. By definition of L', $wa^Mba^M \in L'$ for every $M \geq 0$. Consequently, $wa^Ma^M \in L'$ for some $M \geq 0$ since $L' \in \mathcal{C}_{tract}$. Thus $w \in L$. □

9. Other minor results

9.1. Parametrized complexity

The next section focuses on the parametrized complexity of the RSPQ problem.

para-RSPQ
Input: a db-graph graph $G = (V, \Sigma, E)$,
a regular language L given by an NFA A_L with q states
two vertices x and y
Parameter: q
Question: Is there a simple L-path from x to y in G?

Our initial goal was to determine the parametrized complexity para-RSPQ($\mathcal{C}_{tract}$) when L is restricted to $\mathcal{C}_{tract}$. Unfortunately, we could only partially reach this goal. We first address the parametrized complexity of RSPQs when the parameter is the size of the path.

k-RSPQ
Input: a db-graph graph $G = (V, \Sigma, E)$,
a regular language L given by an NFA A_L with q states,
two vertices x and y
an integer $k \geq 0$
Parameter: k
Question: Is there a simple L-labeled path of size at most k from x to y in G?

Theorem 8. *k-RSPQ is* FPT. *More precisely, the problem is solvable in time* $2^{O(k)}q \cdot |G| \cdot \log |G|$.

The proof is based on the Color Coding method [1]. Let V be a finite set. A k-coloring of V is a function $c : V \to [k]$. A set $S \subseteq V$ is colorful for c if $c(x) = c(y) \Rightarrow x = y$ for every $x, y \in S$. The crux of our proof is the following result by Alon et al.:

Theorem 9 *([1]). Given* $k, n \geq 0$ *and a set* V *of* n *elements, one can compute in time* $2^{O(k)}|V| \log |V|$ *a set of* $l \in 2^{O(k)} \log |V|$ *k-coloring functions* $c_1, \ldots c_l$ *such that every set* S *of* V *of size* k *is colorful for at least one* c_i $(i \in [l])$.

To exploit this result when building incrementally a simple path from x to y we shall record for each coloring the subset of colors used so far, instead of recording the set of nodes visited. This reduces the number of combinations considered to $2^{O(k)} \log |V|$.

Proof of Theorem 8. Let G, A_L, k be an instance of k-RSPQ. We compute l k-coloring functions $c_1, \ldots c_l$ as stated in Theorem 9. Let c be one of these functions. We will show how to decide if there is a colorful L-labeled path from x to y in G for c. To this purpose, we define a function $f : V \times Q_L \times \mathcal{P}([k]) \to \{0, 1\}$ such that $f(v, q, S) = 1$ if and only if there exists a colorful path p starting from x that uses only colors of S and such that $\Delta_L(i_Q, w) = q$ where w is the label of p. The function can be computed by dynamic programming using the following equation.

- $f(x, i_Q, \{c(x)\}) = 1$
- $f(v, q, S) = 1$ if there is a subset $S' \subsetneq S$ such that $f(v, q, S') = 1$;
- $f(v, q, S) = 1$ if $c(v) \in S$ and there is a vertex v', a state q' and a label a such that $f(v', q', S \setminus c(v)) = 1$, $(v', a, v) \in E$ and $q \in \Delta_L(q', a)$;
- $f(v, q, S) = 0$ otherwise.

This function can be computed in time $O(2^k \cdot |A_L| \cdot |G|)$. We compute f for every function c_i, $i \in [l]$ where $l \in 2^{O(k)} \log |V|$. Clearly there is a simple path of length at most k from x to y if and only if there are $i \in [l]$, $S \subseteq [k]$ and $q \in F_L$ such that $f(y, q, S) = 1$ for coloring function c_i. Consequently, k-RSPQ can be solved in time $2^{O(k)}|A_L| \cdot |G| \cdot \log |G|$. □

As a consequence of this theorem we get:

Corollary 3. *para-RSPQ restricted to finite languages is* FPT.

9.2. Directed treewidth

Directed treewidth is a notion introduced in [14]. It measures in some sense how close a digraph is to a directed acyclic graph and is based on the notion of arboreal decomposition. Johnson et al. [14] present a general method to design polynomial algorithms on graphs of bounded directed treewidth. Like most algorithms exploiting treewidth, this method leverages a dynamic programming approach on the decomposition tree. They apply this method to show that testing the

existence of a Hamiltonian path is polynomial on such classes of graphs. Here, we extend this result to show that the regular simple path problem is also computable in polynomial time for the same classes.

It has been observed in the literature that RSPQ has polynomial combined complexity on two interesting classes of graphs: graphs of bounded treewidth [6], and DAGs [20]. The result for DAGs is immediate indeed, as every path in a DAG is simple. The next theorem generalizes both these two results.

Theorem 10. *Let L be a regular language, $k \geq 0$ and $\mathcal{G}$ be a class of db-graphs with directed treewidth at most k. Then, RSPQ(L) restricted to the class $\mathcal{G}$ is polynomial if an arboreal decomposition of the graph is given as input. Furthermore, it is also polynomial if L is a part of the problem.*

Proof. The proof is a straightforward adaptation of the proof proposed in [14] for the Hamiltonian Path problem. Since they use a dynamic approach, they consider a more general problem: given a digraph G and a sequence of k tuples $(v_i, n_i, v'_i)_{i\in[k]}$, are there k disjoint simple paths $p_1, \dots p_k$ such that p_i is a path of size n_i from v_i to v'_i for every $i \in [k]$?

We extend the problem as follows: given a db-graph G, a regular language L and a sequence of k tuples $(v_i, n_i, v'_i, q_i, q'_i)_{i\in[k]}$, are there k words $w_1, \dots w_k$ and k disjoint simple paths $p_1, \dots p_k$ such that p_i is a w_i-labeled path of size n_i from v_i to v'_i and $\Delta_L(q_i, w_i) = q'_i$ for every $i \in [k]$? Therefore, their proof can easily be adapted to this new problem. □

10. Future work

We now pinpoint some directions for future work.

- As an extension of our work, we can consider context-free languages. It seems to be difficult to obtain useful results, since we can easily prove that distinguishing polynomial and NP-hard instances is undecidable if P ≠ NP.
- What becomes tractable under restrictions to the graph such as planar digraphs or undirected graphs? Notice that both disjoint paths and even path problems are polynomial in these cases [16,21,26,28].
- Study the parametrized complexity of RSPQ for languages L represented by NFAs, the parameter is the number of states of the NFA (or L is represented by a regular expression and the parameter is the size of the regular expression). Corollary 3 proves that the complexity is FPT when L is restricted to be finite.

Declaration of competing interest

The authors declare that they have no known compe5ng financial interests or personal rela5onships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers and Tina Trautner for their insightful comments on the manuscript.

References

[1] N. Alon, R. Yuster, U. Zwick, Color-coding, J. ACM 42 (4) (1995) 844–856.
[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J.L. Reutter, D. Vrgoc, Foundations of modern query languages for graph databases, ACM Comput. Surv. 50 (5) (2017) 68:1–68:40.
[3] M. Arenas, S. Conca, J. Pérez, Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard, in: WWW, 2012, pp. 629–638.
[4] E.M. Arkin, C.H. Papadimitriou, M. Yannakakis, Modularity of cycles and paths in graphs, J. ACM 38 (2) (1991) 255–274.
[5] G. Bagan, A. Bonifati, B. Groz, A trichotomy for regular simple path queries on graphs, in: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2013, pp. 261–272.
[6] C.L. Barrett, R. Jacob, M.V. Marathe, Formal-language-constrained path problems, SIAM J. Comput. 30 (3) (2000) 809–837.
[7] A. Bielefeldt, J. Gonsior, M. Krötzsch, Practical linked data access via SPARQL: the case of Wikidata, in: Proceedings of LDOW Workshop, 2018.
[8] A. Bonifati, W. Martens, T. Timm, An analytical study of large SPARQL query logs, VLDB J. 11 (2) (2017) 149–161.
[9] A. Bonifati, W. Martens, T. Timm, Navigating the maze of Wikidata query logs, in: The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13–17, 2019, 2019, pp. 127–138.
[10] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.
[11] N. Immerman, Nondeterministic space is closed under complementation, SIAM J. Comput. 17 (5) (1988) 935–938.
[12] N. Immerman, Descriptive Complexity, Springer, 1999.
[13] R. Jin, H. Hong, H. Wang, N. Ruan, Y. Xiang, Computing label-constraint reachability in graph databases, in: SIGMOD Conference, 2010, pp. 123–134.
[14] T. Johnson, N. Robertson, P.D. Seymour, R. Thomas, Directed tree-width, J. Comb. Theory, Ser. B 82 (1) (2001) 138–154.
[15] A. Koschmieder, U. Leser, Regular path queries on large graphs, in: SSDBM, 2012, pp. 177–194.
[16] A.S. Lapaugh, C.H. Papadimitriou, The even-path problem for graphs and digraphs, Networks 14 (4) (1984) 507–513.
[17] U. Leser, A query language for biological networks, in: ECCB/JBI, 2005, p. 39.
[18] K. Losemann, W. Martens, The complexity of regular expressions and property paths in SPARQL, ACM Trans. Database Syst. (TODS) 38 (4) (2013) 24.
[19] W. Martens, T. Trautner, Evaluation and enumeration problems for regular path queries, in: ICDT, in: LIPIcs, vol. 98, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018, pp. 19:1–19:21.

[20] A.O. Mendelzon, P.T. Wood, Finding regular simple paths in graph databases, SIAM J. Comput. 24 (6) (1995) 1235–1258.
[21] Z.P. Nedev, Finding an even simple path in a directed planar graph, SIAM J. Comput. 29 (1999) 685–695.
[22] Z.P. Nedev, P.T. Wood, A polynomial-time algorithm for finding regular simple paths in outerplanar graphs, J. Algorithms 35 (2) (2000) 235–259.
[23] F. Olken, Graph data management for molecular biology, Omics. J. Integr. Biol. 7 (1) (2003) 75–78.
[24] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
[25] M. Perles, M. Rabin, E. Shamir, The theory of definite automata, IEEE Trans. Electron. Comput. EC-12 (3) (June 1963) 233–243.
[26] N. Robertson, P.D. Seymour, Graph minors XIII. The disjoint paths problem, J. Comb. Theory, Ser. B 63 (1) (1995) 65–110.
[27] W.L. Ruzzo, J. Simon, M. Tompa, Space-bounded hierarchies and probabilistic computations, J. Comput. Syst. Sci. 28 (2) (1984) 216–230.
[28] A. Schrijver, Finding k disjoint paths in a directed planar graph, SIAM J. Comput. 23 (4) (1994) 780–788.
[29] M.P. Schützenberger, On finite monoids having only trivial subgroups, Inf. Control 8 (2) (1965) 190–194.
[30] C.B. Ward, N.M. Wiegand, Complexity results on labeled shortest path problems from wireless routing metrics, Comput. Netw. 54 (2) (2010) 208–217.