Distinct Shortest Walk Enumeration for RPQs

CLAIRE DAVID, LIGM, Université Gustave Eiffel, CNRS, France
NADIME FRANCIS, LIGM, Université Gustave Eiffel, CNRS, France
VICTOR MARSAULT, LIGM, Université Gustave Eiffel, CNRS, France

We consider the Distinct Shortest Walks problem. Given two vertices s and t of a graph database $\mathcal{D}$ and a regular path query, we want to enumerate all walks of minimal length from s to t that carry a label that conforms to the query. Usual theoretical solutions turn out to be inefficient when applied to graph models that are closer to real-life systems, in particular because edges may carry multiple labels. Indeed, known algorithms may repeat the same answer exponentially many times. We propose an efficient algorithm for graph databases with multiple labels. The preprocessing runs in $\mathbf{O}(|\mathcal{D}| \times |\mathcal{A}|)$ and the delay between two consecutive outputs is in $\mathbf{O}(\lambda \times |\mathcal{A}|)$, where $\mathcal{A}$ is a nondeterministic automaton representing the query and λ is the minimal length. The algorithm can handle ε-transitions in $\mathcal{A}$ or queries given as regular expressions at no additional cost.

CCS Concepts: • **Information systems → Graph-based database models**; • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Regular languages**; **Database query processing and optimization (theory)**.

Additional Key Words and Phrases: graph databases, regular path queries, all shortest walks, enumeration complexity, deduplication

ACM Reference Format:
Claire David, Nadime Francis, and Victor Marsault. 2024. Distinct Shortest Walk Enumeration for RPQs. *Proc. ACM Manag. Data* 2, N2 (PODS), Article 100 (May 2024), 22 pages. https://doi.org/10.1145/3651601

1 INTRODUCTION

Regular Path Queries (RPQs, [4, 8]) are the building block of most query languages over graph databases. Formally, an RPQ is defined by a regular expression R and is said to *match* any walk in the database which carries a label that conforms to R. During query processing, one fundamental task consists in producing all matching walks of minimal length that start and end at given vertices. In particular, this task is at the heart of the *all-shortest-walks semantics*, one of the most widespread semantics in practice. For instance, it is the semantics of GSQL [9, 23] (TigerGraph), and of the theoretical language G-Core [3]. All-shortest-walks semantics is also supported by PGQL [16] (Oracle), and by GQL [12, 13]. The latter is particularly relevant, as it has been designed to become the standard query language for graph databases.

In theoretical settings, this task is often considered to pose little challenge. Indeed, graph databases are usually abstracted as single-labeled graphs $\mathcal{D}$ (that is, graphs whose edges carry exactly one

Authors' addresses: Claire David, LIGM, Université Gustave Eiffel, CNRS, Bâtiment Copernic, 5 boulevard Descartes, 77420, Champs-sur-Marne, France, claire.david@univ-eiffel.fr; Nadime Francis, LIGM, Université Gustave Eiffel, CNRS, Bâtiment Copernic, 5 boulevard Descartes, 77420, Champs-sur-Marne, France, nadime.francis@univ-eiffel.fr; Victor Marsault, LIGM, Université Gustave Eiffel, CNRS, Bâtiment Copernic, 5 boulevard Descartes, 77420, Champs-sur-Marne, France, victor.marsault@univ-eiffel.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/5-ART100
https://doi.org/10.1145/3651601

label from a finite alphabet), while queries are given as a deterministic finite automaton $\mathcal{A}$. In that case, finding shortest matching walks can be done as follows: construct the product graph $\mathcal{D} \times \mathcal{A}$, identify vertices that correspond to initial and final states of the automaton, discard all labels, and then simply run any well-known algorithm for finding shortest paths in an unlabeled graph.

However, this approach does not apply to real-life scenarios. Queries are typically given by the user as a regular expression, which does not translate to a deterministic automaton without a possible exponential increase in size. More importantly, real-life systems allow edges to carry multiple labels, either natively (as in GQL), or as a theoretical abstraction of boolean tests on data values. These two features lead to *nondeterminism* both *in the query* and *in the data*. Thus, our goal is to efficiently solve the problem below.

Distinct Shortest Walks
• Inputs: A multi-labeled multi-edge database $\mathcal{D}$, and two vertices s, t in $\mathcal{D}$.
• Query: A nondeterministic finite automaton $\mathcal{A}$.
• Output: All shortest walks from s to t that match $\mathcal{A}$, without duplicates.

Remark that this problem asks for a variable amount of outputs. In such settings, the standard approach consists in distinguishing between *preprocessing* (time before the first output) and *delay* (time between two consecutive outputs). This is known as *enumeration complexity*; see [21] for more details.

The main challenge is the handling of duplicates. Indeed, when either the query $\mathcal{A}$ or the database $\mathcal{D}$ allows nondeterminism, a single walk w of $\mathcal{D}$ might correspond to exponentially many walks in $\mathcal{D} \times \mathcal{A}$. In that case, naively enumerating shortest walks in $\mathcal{D} \times \mathcal{A}$ would return an exponential number of copies of w. One could ensure that each walk is returned only once by storing all outputted walks. In the worst case, this approach requires exponential space and leads to an exponential delay since the algorithm might find all copies of the same walk before discovering a new one.

Quite surprisingly, this problem has received little attention in the literature. In [18], Martens and Trautner[1] use a prior result due to Ackermann and Shallit [1] to show that it can be enumerated with polynomial delay. The construction is given in more details in [20]. Note that although they consider the case of single-labeled databases, their result can be adapted to multi-labeled graph databases at no additional cost. Proving precise complexity upper bounds was not the main concern of [18, 20], and [1] did not use the enumeration complexity framework. Thus, the resulting algorithm only achieves a polynomial delay bound. The technical report [14] translates [1] into the enumeration complexity framework, which leads to the following, more precise, statement.

Theorem 1 ([18] combined with [14]). *Let $\mathcal{A}$ be a nondeterministic automaton, $\mathcal{D}$ a database, and s, t two vertices in $\mathcal{D}$. Distinct Shortest Walks($\mathcal{D}$,$\mathcal{A}$,s,t) can be enumerated with delay in $\mathbf{O}(|\mathcal{D}| \times |\Delta| \times \lambda)$ after a preprocessing in $\mathbf{O}(|Q|^2 \times |V|^2 \times \lambda + |\Delta| \times |\mathcal{D}| \times \lambda)$, where λ is the length of a shortest walk, Q is the state set of $\mathcal{A}$, and Δ is the transition table of $\mathcal{A}$.*

Other recent articles [11, 17] have been devoted to query evaluation under all-shortest-walks semantics, which once again highlights its significance as a core task in graph data management. However, in both cases, graphs are assumed to be single-labeled and the query is assumed to be given as an unambiguous automaton. These assumptions prevent all form of nondeterminism from the product graph $\mathcal{D} \times \mathcal{A}$ in the following sense: if a walk w of $\mathcal{D}$ matches $\mathcal{A}$, there is only one

[1]Remark: Trautner is now known as Popp.

witness of this fact in $\mathcal{D} \times \mathcal{A}$. In that case, the usual approach produces no duplicates. This leads to an algorithm with $\mathbf{O}(|\mathcal{D}| \times |\mathcal{A}|)$ preprocessing and $\mathbf{O}(\lambda)$ delay.

Contributions. We propose an efficient algorithm for Distinct Shortest Walks. As stated in Theorem 2 below, the algorithm yields very satisfactory complexity bounds. In particular, the delay between two consecutive outputs during the enumeration phase *does not depend* on the size of the input database, and the preprocessing phase is only linear in the size of the database.

Theorem 2. *Let $\mathcal{A}$ be a nondeterministic automaton, $\mathcal{D}$ be a database, and s, t be two vertices in $\mathcal{D}$. Distinct Shortest Walks($\mathcal{D}$,$\mathcal{A}$,s,t) can be enumerated with delay in $\mathbf{O}(\lambda \times |\mathcal{A}|)$ after a preprocessing in $\mathbf{O}(|\mathcal{D}| \times |\mathcal{A}|)$, where λ is the length of a shortest walk.*

Our algorithm has an additional $|\mathcal{A}|$ factor in the delay when compared to the simpler setting, as an extra cost for handling nondeterminism in the query and in the data. Note that it takes linear time to check whether a given automaton $\mathcal{A}$ is deterministic and a given database $\mathcal{D}$ is single-labeled. Thus, detecting that the input lies in the more favourable setting and running the more efficient algorithm instead can be done at no additional cost.

The main idea behind the algorithm is as follows. The enumeration phase consists in a depth-first traversal of the set of answers, represented as a backward-search tree rooted at the target t. Since all branches of $\mathcal{T}$ have the same length λ, this ensures the required bound on the delay between two consecutive outputs. Additionally, all walks are represented at most once in $\mathcal{T}$, which ensures that no output is returned twice. Ideally, the preprocessing phase would explicitly compute $\mathcal{T}$. This, however, can take an exponential time and space in the worst case scenario. To circumvent this issue, the algorithm will instead annotate the database with a limited amount of information that allows recomputing $\mathcal{T}$ *on the fly* during the enumeration phase. This annotation makes use of carefully chosen data structures to ensure that recomputing the next edge of $\mathcal{T}$ *does not depend* on the size[2] of $\mathcal{D}$ and is only linear in $\mathcal{A}$.

Outline. Most of the article is devoted to describing the algorithm and proving its properties. After necessary preliminaries, Section 3 gives the algorithm in pseudocode and introduces the necessary tools to prove its correctness. Section 4 establishes the complexity bounds that were claimed in Theorem 2. Finally, Section 5 discusses several extensions of the algorithm that do not affect its complexity. In particular, we show that the algorithm can easily be adapted in case the query is given as a regular expression or as an automaton with spontaneous ε-transitions.

2 PRELIMINARIES

2.1 Sets, lists and queues

First, we explicitly state our assumptions about the data structures that are used to represent collections of elements. Indeed, the efficiency of the algorithm hinges in part on carefully choosing the data structure that is used at each step, as some structures are more efficient for traversal, ordered insertion, or copy. In particular, we use the following structures:

Maps and sets. A *map* $S : X \to Y$ is implemented as an array of elements of $Y \cup \{\perp\}$ of size $|X|$ hence:

- Creating and initializing a new map takes time $\mathbf{O}(|X|)$.
- Assigning an image y to x or removing x takes time $\mathbf{O}(1)$. This is denoted as $S[x] \leftarrow y$ or $S[x] \leftarrow \perp$.
- Browsing a map takes time $\mathbf{O}(|X|)$.

[2]It does not even depend on the degree of the current vertex.

A *set* over domain X is simply a map : $X \to \{\top, \bot\}$.

Lists. Our lists can be modified in only one way: append an element at the head. In particular, elements cannot be deleted or replaced. Such immutable singly-linked lists enjoy the following operations.

- Creating a new empty list takes time $\mathbf{O}(1)$.
- Appending in the head takes time $\mathbf{O}(1)$.
- Copying a list amounts to copying the head pointer, hence takes time $\mathbf{O}(1)$.

Restartable queues. Our algorithm requires queues that may be restarted. They are implemented as linked lists with three pointers: start, end, and current. Queues feature the following operations, which all take time $\mathbf{O}(1)$.

- Creation of an empty queue.
- Enqueue, that adds an element at the end of the queue.
- Advance, that moves current to the next element.
- Peek, that retrieves the element pointed by current.
- Restart, that moves current to the start.

2.2 Graph databases

In this document, we model graph databases as multi-labeled, multi-edge directed graphs, and simply refer to them as *databases* for short. Databases are formally defined below. Figure 1 gives an example of such a database.

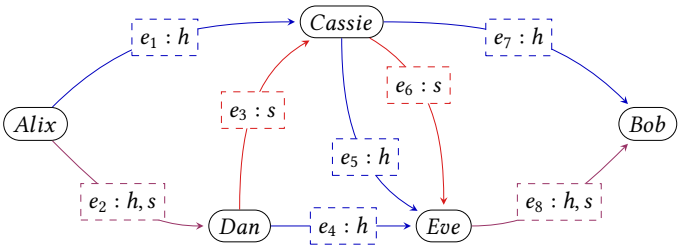


Fig. 1. A (multi-edge multi-labeled graph) database $\mathcal{D}$

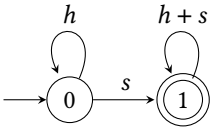


Fig. 2. An automaton $\mathcal{A}$ with $L(\mathcal{A}) = h^*s(h+s)^*$

Definition 3. *A* database $\mathcal{D}$ *is a tuple* $(\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ *where:* Σ *is a finite set of symbols, or* labels*;* V *is a finite set of* vertices*;* E *is a finite set of* edges*;* $\textsc{Src} : E \to V$ *is the* source *function;* $\textsc{Tgt} : E \to V$ *is the* target *function; and* $\textsc{Lbl} : E \to 2^{\Sigma}$ *is the* labelling *function.*

Since this article establishes precise complexity bounds, we need to make our assumptions about the memory representation of databases explicit. We assume the following:

- Every label takes space $\mathbf{O}(1)$ and equality of two labels can be checked in time $\mathbf{O}(1)$.
- Every vertex v provides the following in time $\mathbf{O}(1)$.
  - $\textsc{In}(v)$: an array of pointers to the edges ending in v
  - $\textsc{InDeg}(v)$: the in-degree of v, that is $|\textsc{In}(v)|$
  - $\textsc{Out}(v)$: an array of pointers to the edges starting in v
  - $\textsc{OutDeg}(v)$: the out-degree of v, that is $|\textsc{Out}(v)|$
- Every edge e provides the following in time $\mathbf{O}(1)$.
  - $\textsc{Src}(e)$: a pointer to the source vertex of e
  - $\textsc{Tgt}(e)$: a pointer to the target vertex of e

- $\textsc{Lbl}(e)$: the set of labels carried by e. We assume that iterating through $\textsc{Lbl}(e)$ is done in time $\mathbf{O}\big(|\,\textsc{Lbl}(e)|\big)$.
- $\textsc{TgtIdx}(e)$: the position of e in $\textsc{In}(\textsc{Tgt}(e))$, that is $e = \textsc{In}(\textsc{Tgt}(e))[\textsc{TgtIdx}(e)]$.

Altogether, the size of $\mathcal{D} = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ satisfies:

$$|\mathcal{D}| \in \mathbf{O}\left(|V| + |E| + \sum_{e \in E} |\,\textsc{Lbl}(e)|\right)$$

Remark 4. *The only nonstandard item in the memory representation of databases is the function* TgtIdx. *Note that it may be precomputed in time* $\mathbf{O}\big(|V| + |E|\big)$ *if it is not natively provided by the database. Thus, this assumption does not change the complexity bounds promised in Theorem 2.*

Definition 5. *A* walk w *in a database* $\mathcal{D}$ *is a nonempty finite sequence of alternating vertices and edges of the form* $w = \langle v_0, e_0, v_1, \ldots, e_{k-1}, v_k\rangle$ *where* $k \geq 0$, $v_0, \ldots, v_k \in V$, $e_0, \ldots, e_{k-1} \in E$, *such that:*

$$\forall i, 0 \leq i < k, \quad \textsc{Src}(e_i) = v_i \quad \textit{and} \quad \textsc{Tgt}(e_i) = v_{i+1}$$

We call k *the* length *of* w *and denote it by* $\textsc{Len}(w)$. *We extend the functions* Src, Tgt *and* Lbl *to the walks in* $\mathcal{D}$ *as follows. For each walk* $w = \langle v_0, e_0, v_1, \ldots, e_{k-1}, v_k\rangle$ *in* D, $\textsc{Src}(w) = v_0$, $\textsc{Tgt}(w) = v_k$, *and* $\textsc{Lbl}(w) = \left\{ a_0 a_1 \cdots a_{k-1} \;\middle|\; \forall i, 0 \leq i < k,\ a_i \in \textsc{Lbl}(e_i) \right\}$. *Finally,* $s \xrightarrow{w} t$ *means that* $\textsc{Src}(w) = s$ *and* $\textsc{Tgt}(w) = t$.

We say that two walks w, w' concatenate *if* $\textsc{Tgt}(w) = \textsc{Src}(w')$, *in which case we define their* concatenation *as usual, and denote it by* $w \cdot w'$, *or simply* ww' *for short.*

For ease of notation, we will sometimes omit vertices from walks, as they are implicitly defined by the adjacent edges: for instance, in the database of Figure 1, $\langle e_1, e_7\rangle$ *is short for* $\langle Alix, e_1, Cassie, e_7, Bob\rangle$. *Similarly, for a walk* w *and an edge* e, *we write* $w \cdot e$ *as a shorthand for* $w \cdot \langle \textsc{Src}(e), e, \textsc{Tgt}(e)\rangle$.

We use $\textsc{Walks}(\mathcal{D})$ *to denote the (possibly infinite) set of all walks in* $\mathcal{D}$, *and* $\textsc{Walks}^{\leq \ell}(\mathcal{D})$ *to denote the restriction of* $\textsc{Walks}(\mathcal{D})$ *to walks of length at most* ℓ.

2.3 Automata and queries

Definition 6. *A nondeterministic* automaton $\mathcal{A}$ *is a 5-tuple* $(\Sigma, Q, \Delta, I, F)$ *where* Σ *is a finite set of symbols,* Q *is a finite set of* states, $I \subseteq Q$ *is the set of* initial *states,* $\Delta \subseteq Q \times \Sigma \times Q$ *is the set of* transitions *and* $F \subseteq Q$ *is the set of* final *states.*

As usual, we extend Δ *into a relation over* $Q \times \Sigma^* \times Q$ *as follows: for every* $q \in Q$, $(q, \varepsilon, q) \in \Delta$; *and for every* $q, q', q'' \in Q$ *and every* $x, y \in \Sigma^*$, *if* $(q, x, q') \in \Delta$ *and* $(q', y, q'') \in \Delta$ *then* $(q, xy, q'') \in \Delta$. *We write* $\Delta(q, u)$ *and* $\Delta^{-1}(u, q')$ *as shorthands for the sets* $\Delta(q, u) = \{q' \mid (q, u, q') \in \Delta\}$ *and* $\Delta^{-1}(u, q') = \{q \mid (q, u, q') \in \Delta\}$, *respectively.*

We denote by $L(\mathcal{A})$ *the* language *of* $\mathcal{A}$, *defined as follows.*

$$L(\mathcal{A}) = \left\{ x \in \Sigma^* \;\middle|\; \exists i \in I,\ \exists f \in F,\ (i, x, f) \in \Delta \right\}$$

As for databases, we explicitly state our assumptions about the memory representation of automata.

- States are encoded as integers, that is $Q = \{0, \ldots, |Q| - 1\}$.
- Given a state q and a label a, one may access $\Delta(q, a)$ in time $\mathbf{O}(1)$, and $\Delta(q, a)$ is a (non-repeating) list of states.

Altogether, the size of $\mathcal{A}$ is $|\Sigma| + |Q| + |\Delta| + |I| + |F|$, which is in $\mathbf{O}\big(|\Sigma| + |Q| + |\Delta|\big)$.

Definition 7. *A* Regular Path Query *(RPQ) is defined by a regular language, given as an automaton* $\mathcal{A}$. *Given an RPQ* $\mathcal{A}$ *and a walk* w *in a database* $\mathcal{D}$, *we say that* $\mathcal{A}$ matches w *(or equivalently, that* w *matches* $\mathcal{A}$*) if* $L(\mathcal{A}) \cap \textsc{Lbl}(w) \neq \emptyset$. *We use* $\textsc{Match}(\mathcal{A}, \mathcal{D})$ *to denote the (possibly infinite) set of walks that match* $\mathcal{A}$.

2.4 Distinct shortest walks

Given a query $\mathcal{A}$, a database $\mathcal{D}$, and two vertices s and t of $\mathcal{D}$, our goal is to enumerate the set of all shortest walks from s to t that match $\mathcal{A}$. This set is written $[\![\mathcal{A}]\!](\mathcal{D}, s, t)$ and defined as follows.

Definition 8. *Let $\mathcal{A}$ be an automaton, $\mathcal{D}$ a database and s, t two vertices of $\mathcal{D}$. We define*

$$[\![\mathcal{A}]\!](\mathcal{D}, s, t) = \{\, w \in \textsc{Match}(\mathcal{A}, \mathcal{D}) \mid \textsc{Len}(w) = \lambda \,\},$$

where $\lambda = \min\{\, \textsc{Len}(w) \mid w \in \textsc{Match}(\mathcal{A}, \mathcal{D}), s \xrightarrow{w} t \,\}$.

We can now restate the main problem as:

Distinct Shortest Walks
• Inputs: A multi-labeled multi-edge database $\mathcal{D}$, and two vertices s, t in $\mathcal{D}$.
• Query: A nondeterministic finite automaton $\mathcal{A}$.
• Output: Enumerate $[\![\mathcal{A}]\!](\mathcal{D}, s, t)$, without duplicates.

Example 9. *In the graph database $\mathcal{D}$ of Figure 1, vertices represent people and edges represent bank transfers. Transfers can have up to two labels: h for "high value" and s for "suspicious".*

Assume that we are searching for fraudulent behavior. We want to find sequences of transfers from Alix to Bob that contain only high value or suspicious transfers, with at least one of them being suspicious. This corresponds to computing $[\![\mathcal{A}]\!](\mathcal{D}, Alix, Bob)$ with $\mathcal{A}$ being the two-state automaton that captures $h^*s(h+s)^*$ (Fig. 2).

We can now remark the following:

- *The shortest walk from Alix to Bob, $\langle e_1, e_7 \rangle$, is of length 2. However, it does not match $\mathcal{A}$, as $hh \notin L(\mathcal{A})$.*
- *Four walks of minimal length 3 match $\mathcal{A}$: $w_1 = \langle e_1, e_5, e_8 \rangle$, $w_2 = \langle e_1, e_6, e_8 \rangle$, $w_3 = \langle e_2, e_3, e_7 \rangle$ and $w_4 = \langle e_2, e_4, e_8 \rangle$.*
- *w_4 carries* three *labels that belong to $L(\mathcal{A})$: shh, hhs and shs. It is still returned only* once. *The same holds for w_2 and w_3, which carry* two *suitable labels.*
- *Even though they visit the same vertices, w_1 and w_2* are not *the same walk, and are* both *returned. Indeed, e_5 and e_6 do not represent the same transfer. In the database, they might have different amounts, dates, operating banks...*
- *The walk $w_5 = \langle e_2, e_3, e_6, e_8 \rangle$ matches $\mathcal{A}$, but is not returned, as it is not of minimal length.*

3 THE ALGORITHM

In this section, we describe the main algorithm and introduce the necessary tools to prove its correctness. The pseudocode of the main function Main and its subfunctions Annotate, Trim and Enumerate are given in Figure 3-4, page 7-8. This section provides the main ideas and formal statements of all steps of the proofs. Due to space constraints, the technical details of the proofs have been moved to Appendix B.

We fix a graph database $\mathcal{D} = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$, a query $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$, and two vertices s and t of $\mathcal{D}$. We assume that there exists at least one walk from s to t that matches $\mathcal{A}$ and we let λ be the minimal length of any such walk.

The Main function just calls the other three functions. The first two functions, Annotate and Trim, correspond to the preprocessing phase. Figure 5 gives the annotations L_u, B_u and C_u of the preprocessing for Example 9. The TgtIdx of edges are written at the head of arrows. The last function, Enumerate, corresponds to the enumeration phase. The following sections detail each function.

Fig. 3. Pseudocode of the main algorithm 1/2

Inputs of the algorithm, also used as global variables:

- Automaton $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$
- Database $\mathcal{D} = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}\ \textsc{Lbl})$
- Source vertex $s \in V$
- Target vertex $t \in V$

```
1.  function Main()
2.      B, L, λ ← Annotate()
3.      C ← Trim(B)
4.      S ← {q | L_t[q] = λ}
5.      Enumerate(C, λ, ⟨t⟩, S ∩ F)

6.  function Annotate()
7.      ℓ ← 0
8.      current ← empty list
9.      next ← empty list
10.     for each vertex u in V
11.         B_u ← new Map: Q → {0, ..., InDeg(u) − 1} → List[Q],
                  fully initialised with empty lists
12.         L_u ← new empty Map : Q → ℕ
13.     for each state p in I
14.         L_s[p] ← ℓ
15.         add (s, p) to next

16.     stop ← ⊥
17.     while next is not empty and stop = ⊥
18.         ℓ ← ℓ + 1
19.         current ← next
20.         next ← empty list
21.         for each (v, q) ∈ current          // NB: (v, q) ∈ current ⟺ L_v[q] = ℓ − 1.
22.             for each e ∈ Out(v)
23.                 u ← Tgt(e)
24.                 for each p ∈ Δ(q, Lbl(e))
25.                     if p ∉ Dom(L_u)        // First time state p is reached at vertex u
26.                         L_u[p] ← ℓ
27.                         add (u, p) to next
28.                         if u = t and p ∈ F        // First time a final state is reached at vertex t
29.                             stop ← ⊤
30.                         add q to B_u[p][TgtIdx(e)]
31.                     else if L_u[p] = ℓ        // We found another walk of length ℓ that reaches
                                                     state p at vertex u.
32.                         add q to B_u[p][TgtIdx(e)]
33.     return (B, L, ℓ)
```

Fig. 4. Pseudocode of the main algorithm 2/2

```
34. function Trim(Annotation B)
35.     for each u ∈ V
36.         C_u ← new Map Q → Queue[In(u) × List[Q]], initialised with empty queues
37.         for each p ∈ Q
38.             for each e ∈ In(u)          // Recall that In(u) is sorted in increasing TgtIdx order
39.                 if B_u[p][TgtIdx(e)] is not empty
40.                     enqueue (e, B_u[p][TgtIdx(e)]) in C_u[p]
41.         // Note that, by construction, for all u, p, the queue C_u[p] is sorted by first component: the
               pairs (e, _) are in increasing order of TgtIdx(e)
42.     return C


43. function Enumerate (Trimmed annotation C, Integer ℓ, Walk w, State set S)
44.     u ← Src(w)
45.     if ℓ = 0          // Remark that ℓ = 0 implies u = s and S ⊆ I. No need to verify it.
46.         output w
47.     else
48.         while ⊤
49.             e_min ← nil
50.             for each p ∈ S
51.                 if C_u[p] is not empty
52.                     (e, X) ← peek C_u[p]
53.                     if e_min=nil or TgtIdx(e)< TgtIdx(e_min)
54.                         e_min ← e
55.             if e_min = nil          // All queues are exhausted. It never happens on the first iteration.
56.                 for each p ∈ S
57.                     restart C_u[p]
58.                 return
59.             S′ ← new empty subset of Q
60.             for each p ∈ S
61.                 if C_u[p] is not empty
62.                     (e, X) ← peek C_u[p]
63.                     if e = e_min
64.                         for each q ∈ X
65.                             S′[q] ← ⊤
66.                         advance C_u[p]
67.             Enumerate(C, ℓ − 1, e_min · w, S′)
```

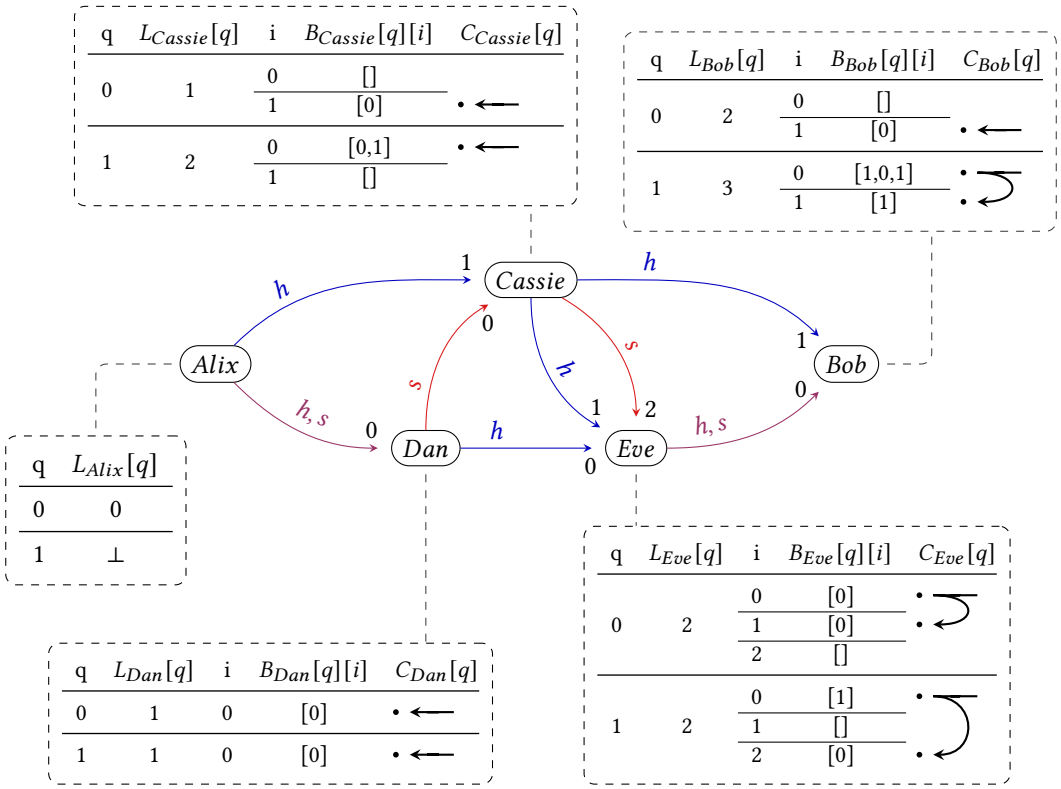


Fig. 5. Annotation of $\mathcal{D}$ (Fig. 1) when queried by automaton $\mathcal{A}$ (Fig. 2) from *Alix* to *Bob*

3.1 Annotate

Annotate is the main preprocessing function. Intuitively, its purpose is to precompute, for all vertex u of $\mathcal{D}$ and state $p \in Q$, the shortest walks w that start in s, end in u and carry a label that reaches p in $\mathcal{A}$. However, representing this set of walks explicitly could take exponential time and space. Instead, Annotate will annotate each vertex u with two maps, L_u and B_u.The map L_u ("length") records the length of the shortest walks that can reach the states of $\mathcal{A}$ at u. The map B_u ("back") records the last edge used along said walks, in order to be able to reconstruct them backwards.

Formally, if w is a shortest walk from s to u with a label that reaches p, it will be reflected as follows in the two maps:

- $L_u[p] = \textsc{Len}(w)$. L_u is a partial map: if no such w exists, $L_u[p]$ is undefined.
- $q \in B_u[p][\textsc{TgtIdx}(e)]$, where e is the last edge of w and $p \in \Delta(q, \textsc{Lbl}(e))$, that is, q is a possible predecessor state to p along the walk. $B_u[p][\textsc{TgtIdx}(e)]$ is a list, and thus may contain duplicates. However, its total size will never exceed $\sum_{a\in\Sigma} |\Delta^{-1}(a, p)|$.

Annotate populates the maps by essentially performing a breadth-first traversal of the product graph $\mathcal{D} \times \mathcal{A}$. It stops at the end of step λ, where λ is the length of a shortest walk from s to t that matches $\mathcal{A}$. Indeed, it will be the first iteration of the traversal in which the target vertex t is reached together with a final state of $\mathcal{A}$.

The following lemma ensures the correctness of Annotate.

Lemma 10. *At the end of* Annotate*, the following properties hold for all $p, q \in Q$, $u \in V$, $e \in E$ and $i \leq \textsc{InDeg}(u)$:*

(1) $L_u[p] = \min \left\{ \textsc{Len}(w) \,\middle|\, w \in \textsc{Walks}^{\leq\lambda}(\mathcal{D}), s \xrightarrow{w} u \text{ and } p \in \Delta(I, \textsc{Lbl}(w)) \right\}$

(2) $q \in B_u[p][i]$ *if and only if there exists a walk w from s to u of the form $w = w' \cdot e$ such that:*
- $\textsc{Len}(w) = L_u[p]$
- $\textsc{TgtIdx}(e) = i$
- $q \in \Delta(I, \textsc{Lbl}(w'))$
- $p \in \Delta(q, \textsc{Lbl}(e))$

(3) $B_u[p][\textsc{TgtIdx}(e)]$ *is of size at most* $\sum_{a\in\Sigma} |\Delta^{-1}(a, p)|$.

3.2 Trim

Trim is the second step of the preprocessing phase. It translates each map $B_u[p]$ into a queue $C_u[p]$, essentially by removing empty $B_u[p][\textsc{TgtIdx}(e)]$. During the enumeration phase, this allows to efficiently iterate over all edges e such that $B_u[p][\textsc{TgtIdx}(e)]$ is nonempty. Indeed, browsing $B_u[p]$ directly would increase the delay by a factor d, the maximal in-degree of $\mathcal{D}$.

Formally, $C_u[p]$ is a queue of pairs (e, X), where $e \in \textsc{In}(u)$ and X is a nonempty list over Q. The correspondence between C_u and B_u is formally stated in the lemma below, whose proof immediately follows from the pseudocode of Trim.

Lemma 11. *At the end of* Trim*, the following properties hold for all $p \in Q$, $u \in V$, $e, e' \in \textsc{In}(u)$ and lists X, X' over Q:*

(1) $(e, X) \in C_u[p]$ *if and only if* $X = B_u[p][\textsc{TgtIdx}(e)]$ *and* $X \neq \emptyset$.

(2) $C_u[p]$ *is sorted in increasing* $\textsc{TgtIdx}(e)$ *order: if (e, X) appears before (e', X') in $C_u[p]$, then* $\textsc{TgtIdx}(e) < \textsc{TgtIdx}(e')$. *In particular, $e \neq e'$.*

(3) If $(e, X) \in C_u[p]$, then X is of size at most $\sum_{a\in\Sigma} |\Delta^{-1}(a, p)|$.

3.3 Enumerate

Finally, Enumerate makes use of the precomputed structures C_u to handle the enumeration phase. In order to prove its correctness, we formally define the backward-search tree of the set of answers and show how it relates to C_u.

Definition 12. *The backward-search tree $\mathcal{T}$ of $[\![A]\!](\mathcal{D}, s, t)$ is the tree defined as follows:*

(1) The nodes of $\mathcal{T}$ are the suffixes of walks in $[\![A]\!](\mathcal{D}, s, t)$: a walk w is a node of $\mathcal{T}$ if and only if there exists a walk w' such that $w' \cdot w \in [\![A]\!](\mathcal{D}, s, t)$.

(2) The root of $\mathcal{T}$ is the walk $\langle t \rangle$.

(3) The children of a node w are all the walks of the form $e \cdot w$ that are nodes in $\mathcal{T}$.

(4) The children of a node w are ordered according to the target index of their first edge, that is, if $w_1 = e_1 \cdot w$ and $w_2 = e_2 \cdot w$ are two distinct children of w, then w_1 appears before w_2 if and only if $\textsc{TgtIdx}(e_1) < \textsc{TgtIdx}(e_2)$.

Remark 13. *The definition of $\mathcal{T}$ implies the following properties:*

- *In item 4, it cannot be that* $\textsc{TgtIdx}(e_1) = \textsc{TgtIdx}(e_2)$*, as e_1 and e_2 are distinct and have the same target.*
- *All branches of $\mathcal{T}$ from the root to a leaf are of length λ.*
- *The leaves of $\mathcal{T}$ are precisely the walks of $[\![A]\!](\mathcal{D}, s, t)$.*

Intuitively, Enumerate performs a depth-first traversal of $\mathcal{T}$ and outputs precisely the walks that are found at the leaves. The tree will be constructed on the fly, together with a certificate

that witnesses that each branch of the tree does indeed correspond to a walk in $[\![A]\!](\mathcal{D}, s, t)$. This certificate consists in a subset $S(w)$ of Q attached to each node w of $\mathcal{T}$ that guarantees the existence of at least one accepting run of $\mathcal{A}$ over w. It is defined as follows:

Definition 14. *For each node w in $\mathcal{T}$, we denote by $S(w)$ the following set.*

$$S(w) = \left\{ q \in Q \;\middle|\; \begin{array}{l} \exists w_q \in \textsc{Walks}(\mathcal{D}) \\ \bullet\ (w_q \cdot w) \in [\![A]\!](\mathcal{D}, s, t) \\ \bullet\ q \in \left(\Delta(I, \textsc{Lbl}(w_q)) \cap \Delta^{-1}(\textsc{Lbl}(w), F)\right) \end{array} \right\}$$

Remark that the definition immediately implies that, for a node w in $\mathcal{T}$, $S(w) \neq \emptyset$.

Intuitively, $\mathcal{T}$ and $S(\cdot)$ can be reconstructed from C as follows. Assume that $\mathcal{T}$ and $S(\cdot)$ have already been constructed up to some node w. All $p \in S(w)$ are states that can be reached at w and are useful in at least one accepting run that reaches t. Then, computing the children of w in $\mathcal{T}$ (and $S(\cdot)$ for them) amounts to looking for predecessor edges and states that reach p at w. This is precisely the information provided by C, as formally stated below.

Lemma 15. *Let $e \cdot w$ be a node of $\mathcal{T}$ for some edge e and walk w. Let $u = \textsc{Tgt}(e)$. For every p, we denote by X_p the unique*[3] *list of states such that $(e, X_p) \in C_u[p]$ if such a list exists, or $X_p = \emptyset$ otherwise. Then, the following holds.*

$$S(e \cdot w) = \bigcup_{p \in S(w)} set(X_p)$$

We can now explain Enumerate more precisely. Start from the walk $\langle t \rangle$ and $S(\langle t \rangle)$ that is explicitly computed in Main as the set of final states of $\mathcal{A}$ that can be reached after a walk of length λ.

The goal of Enumerate is to make a depth-first traversal of $\mathcal{T}$ while rebuilding the tree on the fly. When called on a walk w, with $u = \textsc{Src}(w)$, and the state set $S(w)$, Enumerate makes use of C_u to construct the children $e \cdot w$ of w in $\mathcal{T}$, along with their certificate $S(e \cdot w)$. Indeed, Lemma 15 states that $S(e \cdot w)$ can be built by looking for the pairs (e, X) in each $C_u[p]$ for $p \in S(w)$. We then proceed with the depth-first traversal by starting over with $w = e \cdot w$ and $S = S(e \cdot w)$.

There is one technical hurdle to overcome before making a recursive call for an edge e: in order to avoid doing two calls for the same e, we have to make sure that we collect all occurrences of pairs (e, X) that appear in any $C_u[p]$ for $p \in S(w)$. However, this cannot be done by browsing each $C_u[p]$ entirely, otherwise the delay would depend on (the maximal in-degree of) $\mathcal{D}$. This issue is solved by the fact that $C_u[p]$ is sorted in TgtIdx order, as stated in Lemma 11. Thus, to find the pairs (e, X) that correspond to the first child of w in $\mathcal{T}$, we only have to search in the head of each $C_u[p]$.

In the pseudocode of Enumerate, lines 49-58 look at the head of each $C_u[p]$ to find the minimal edge e that has not yet been found. Then lines 59-66 correspond to collecting all (e, X) in the head of each $C_u[p]$ for the found edge e. Line 58 corresponds to the case where all $C_u[p]$ have been exhausted.

Finally, Enumerate keeps track, in variable ℓ, of the remaining distance to the leaves of $\mathcal{T}$, and outputs each time it reaches $\ell = 0$.

The correctness of Enumerate comes from this final lemma:

Lemma 16. *The tree of recursive calls to Enumerate is isomorphic to $\mathcal{T}$ in the following sense: Enumerate(C, ℓ, w, S) is called exactly once per node w in $\mathcal{T}$. Moreover, the parameters of this call satisfy $\ell = \lambda - \textsc{Len}(w)$ and $S = S(w)$.*

[3]There can be at most one, due to Lemma 11, item 2.

4 COMPLEXITY ANALYSIS

In Section 4.1, we show that the algorithm meets the complexity bounds claimed in Theorem 2 and in Section 4.2, we discuss its memory usage.

4.1 Time complexity of the algorithm

Annotate. Creating and fully initializing a map B_u for some u takes time $\textsc{InDeg}(u) \times |Q|$. Thus, creating all the maps takes

$$\mathbf{O}\left(\sum_{u\in V} \textsc{InDeg}(u) \times |Q|\right) = \mathbf{O}\big(|E| \times |Q|\big) \tag{1}$$

The remainder of the initialization is negligible.

The main loop (starting at line 16) is essentially a breadth-first traversal of $\mathcal{D} \times \mathcal{A}$, hence it is not surprising that it runs in $\mathbf{O}\big(|E| \times |\Delta|\big)$. Indeed, each pair $(v, q) \in V \times Q$ is visited at most once. Then, for each such (v, q), we visit each outgoing edge e of v (line 22), for which we then visit each outgoing transition of q (line 24). Thus, the elementary instructions, at lines 25-32, are executed at most $|E| \times |\Delta|$ times. Thus, the total runtime of $\textsc{Annotate}$ is $\mathbf{O}\big(|E| \times |\Delta|\big)$.

Trim. Creating and initializing a map C_u for some u takes time $\mathbf{O}\big(|Q|\big)$. Thus, creating all maps takes time $\mathbf{O}\big(|V| \times |Q|\big)$. The remainder of the function consists in three nested for-loops that simply visit each edge-state pair exactly once. Thus, the total runtime of $\textsc{Trim}$ is $\mathbf{O}\big(|E| \times |Q|\big)$.

Enumerate. As shown in Lemma 16, $\textsc{Enumerate}$ performs a depth-first traversal of $\mathcal{T}$ and makes exactly one recursive call per node of the tree. Thus, a leaf of the tree is reached and an output is produced at most every λ recursive calls. Within $\textsc{Enumerate}$, the time between two consecutive recursive calls (excluding time spent in the recursive calls themselves) is dominated by the two nested for-loops (lines 59-66). Lemma 11 states that, for a fixed e, the size of $(e, X_p) \in C_u[p]$ is bounded by $\sum_{a\in\Sigma} |\Delta^{-1}(a, p)|$. Thus, browsing all such X_p for all p in Q takes time $\mathbf{O}\big(|\Delta|\big)$. Hence, $\textsc{Enumerate}$ produces an output at most every $\mathbf{O}\big(\lambda \times |\Delta|\big)$ steps.

Total. The preprocessing phase ($\textsc{Annotate}$ and $\textsc{Trim}$) takes time $\mathbf{O}\big(|E| \times |\Delta|\big)$ and the delay between two consecutive outputs during $\textsc{Enumerate}$ is $\mathbf{O}\big(\lambda \times |\Delta|\big)$, which proves Theorem 2.

4.2 Memory usage

Sometimes, a polynomial delay algorithm might end up using exponential space. Indeed, when the enumeration procedure is allowed to update the precomputed data structure, it could become arbitrarily large after arbitrarily many answers have been outputted. Our algorithm avoids this pitfall.

$\textsc{Remark}$ 17. *Throughout the enumeration, the total memory usage of the algorithm never exceeds* $\mathbf{O}\big(|E| \times |\Delta|\big)$. *Remark that the space needed to store the walk of length* λ *that is being outputted is also in* $\mathbf{O}\big(|E| \times |\Delta|\big)$, *and therefore negligible.*

A stricter class of enumeration algorithms, called *memoryless* [21], forbids any kind of modification to the precomputed structures during the enumeration. Formally, in a memoryless enumeration algorithm, the $(i + 1)$-th output is computed directly from the i-th output and the precomputed data structures.

Our algorithm is *not* memoryless, as the efficiency of $\textsc{Enumerate}$ hinges on reading C_u in the order it was initially created. Thus, enumeration cannot be readily resumed from any given output. We can adapt $\textsc{Enumerate}$ to the memoryless framework as follows. Given an answer w, we perform a computation that is *guided* by w: instead of searching for the minimum edge at lines 49-58, we

set e_{min} to the last edge e of w, advance all queues to the first e' with $\textsc{TgtIdx}(e') \geq \textsc{TgtIdx}(e)$, remove the last edge of w and do the recursive call. Once this computation reaches $\ell = 0$, we skip outputting w. Then, all queues have been set to the correct position, and the algorithm resumes as usual to produce the next output.

Unfortunately, this guided execution costs more than the normal delay: advancing all queues to match e costs $\mathbf{O}\big(|Q| \times \textsc{InDeg}(u)\big)$, where $u = \textsc{Tgt}(e)$. This leads to an algorithm with memoryless delay in $\mathbf{O}\big(d \times \lambda \times |\Delta|\big)$, where d is the maximum in-degree of $\mathcal{D}$.

The d factor can actually be avoided by using a more involved data structure to represent C_u that allows resuming from a given e in constant time. Formally, C_u will be a copy of A_u in which each cell also contains a pointer to the next non-empty cell. It can be computed in time $\mathbf{O}\big(|E| \times |Q|\big)$ as follows:

```
68. function ResumableTrim(Annotation B)
69.     for each u ∈ V
70.         C_u ← new Map Q → {0, ..., InDeg(u) − 1} → List[Q × ℕ]
71.         for each p ∈ Q
72.             next ← nil
73.             for each i ∈ {InDeg(u) − 1, ..., 0}        // We enumerate i in reverse order
74.                 C_u[p][i] ← (B_u[p][i], next)
75.                 if B_u[p][i] is not empty
76.                     next ← i
77.     return C
```

It remains to replace Enumerate by a new function NextOutput that acts as its counterpart in the memoryless setting: given the precomputed data structure C and a previous output w, $\textsc{NextOutput}(C, w)$ performs a guided run as described previously and then produces the next output. Writing this function poses no conceptual challenge, but requires some technical care while traversing C. Altogether, this leads to the following theorem:

Theorem 18. *Distinct Shortest Walks can be enumerated with a* memoryless *algorithm with preprocessing time in* $\mathbf{O}\big(|\mathcal{D}| \times |\mathcal{A}|\big)$ *and a delay in* $\mathbf{O}\big(\lambda \times |\mathcal{A}|\big)$*, where* λ *is the length of a shortest walk.*

5 EXTENSIONS

5.1 Handling spontaneous transitions

Note that Definition 6 disallows ε-transitions. An automaton $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$ allows ε-transitions when $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

Handling spontaneous transitions requires some light editing of Annotate and does not impact the other subfunctions. It essentially consists in eliminating ε-transitions *on the fly*: each time a new state p is reached at a vertex u, we also add each state r that can be reached from p by following one or more ε-transitions. Formally, it amounts to replacing the innermost loop of Annotate (lines 25–32) with a call to $\textsc{PossiblyVisit}(u, p, e)$, given below.

The test on line 80 will be true at most once per pair $(u, p) \in Q \times V$. Thus, at the end Annotate, there will have been at most $|V| \times |\Delta_\varepsilon|$ laps in the for-loop on line 86, where Δ_ε is the set of spontaneous transitions in $\mathcal{A}$. Therefore, this modification does not change the time complexity of Annotate.

```
78. function PossiblyVisit(Vertex u ∈ V, State p ∈ Q, Edge e ∈ E)
79.     // We use next, stop, L_u, B_u, ℓ, t and A from Annotate as global variables
80.     if p ∉ Dom(L_u)          // First time state p is reached at vertex u
81.         L_u[p] ← ℓ
82.         add (u, p) to next
83.         if u = t and p ∈ F          // First time a final state is reached at vertex t
84.             stop ← ⊤
85.         add q to B_u[p][TgtIdx(e)]
86.         for each r ∈ Δ(p, ε)
87.             PossiblyVisit(u, r, e)
88.     else if L_u[p] = ℓ          // We found another walk of length ℓ that reaches state p at vertex u
89.         add q to B_u[p][TgtIdx(e)]
```

5.2 Query given as a regular expression

In real-life scenarios, the query is usually given in some query language that more closely resembles regular expressions, rather than automata. Given a regular expression R, we define $[\![R]\!](\mathcal{D}, s, t)$ as $[\![\mathcal{A}]\!](\mathcal{D}, s, t)$ for any automaton $\mathcal{A}$ that describes the same language as R. We can now formally define the corresponding computational problem and establish the corresponding complexity bounds.

Distinct Shortest Walks (Regexp variant)
• Inputs: A multi-labeled multi-edge database $\mathcal{D}$, and two vertices s, t in $\mathcal{D}$. • Query: A regular expression R. • Output: Enumerate $[\![R]\!](D, s, t)$, without duplicates.

The usual approach for handling a regular expression R consists in translating it first to some equivalent automaton $\mathcal{A}$. We know from Section 5.1 that our algorithm works on automata with ε-transitions at no additional cost. Thus, we can readily use Thompson construction (Theorem 19 below) which, combined with Theorem 2, immediately leads to Corollary 20, stated afterwards.

Theorem 19 (Thompson, 1968). *Given a regular expression R, one may build in time $\mathbf{O}(|R|)$ an equivalent automaton with ε-transitions $\mathcal{A}$ with $\mathbf{O}(|R|)$ states and $\mathbf{O}(|R|)$ transitions.*

Corollary 20. *When the query is given as a regular expression R, Distinct Shortest Walks can be enumerated with a preprocessing time in $\mathbf{O}(|R| \times |\mathcal{D}|)$ and a delay in $\mathbf{O}(\lambda \times |R|)$, where λ is the length of a shortest walk.*

A more common translation from regular expressions to automata consists in using Glushkov construction [7]. The produced NFA has no ε-transitions, but may have up to $\mathbf{O}(|R|^2)$ transitions. In our case, this would yield weaker complexity bounds: $\mathbf{O}(|R|^2 \times |\mathcal{D}|)$ preprocessing time and $\mathbf{O}(\lambda \times |R|^2)$ delay.

5.3 Adaptation to related problems

In this section, we discuss several problems that are related to Distinct Shortest Walks and briefly explain how our algorithm can be adapted to address them.

One source to many targets. In Distinct Shortest Walks, both source and target vertices s and t are given as part of the input. Another version of the problem only fixes s, and then asks for shortest walks from s to t for a subset of (or possibly all) the vertices t in $\mathcal{D}$. This problem can be

solved at no additional cost: start by running Annotate and add wanted targets to a queue the first time they are annotated with a final state of $\mathcal{A}$. This step stops when no new pair (v, q) can be discovered, and thus has the same worst-case complexity as the main algorithm. Then, for each queued target, collect the set of final states that have been reached with a walk of minimal length, and run the enumeration step as usual.

Distinct Cheapest Walks. In this scenario, in addition to their labels, edges of $\mathcal{D}$ carry a positive value, called *cost*. Instead of looking for shortest walks from s to t, this problem asks for *cheapest* walks, that is, walks that minimize the sum of costs along their edges. Our algorithm can be adapted to this setting by replacing the breadth-first traversal in Annotate with a cheapest-first traversal, as is done in Dijkstra's algorithm. In that case, the preprocessing time complexity becomes $\mathbf{O}\big(|\mathcal{D}| \times |\mathcal{A}| + |V| \times |Q| \times \big(\log(|V|) + \log(|Q|)\big)\big)$ using standard techniques ([15]) and the delay is unchanged.

Shortest Walks with Multiplicities. This version of the problem asks to return all shortest walks w from s to t together with their *multiplicity*, that is, the number of different accepting runs of $\mathcal{A}$ over $\textsc{Lbl}(w)$. Theoretically, one could rerun $\mathcal{A}$ on w when it is output, and simply count the runs. Indeed, this would cost $\mathbf{O}\big(\lambda \times |\mathcal{A}|\big)$, and would not change the delay. That being said, our algorithm essentially runs $\mathcal{A}$ over w along the recursive calls to Enumerate. Hence, it can be easily adapted to keep track of the number of times each state has been produced along the walk.

6 PERSPECTIVES

We have proposed an algorithm for solving Distinct Shortest Walks that achieves a $\mathbf{O}\big(\lambda \times |\mathcal{A}|\big)$ delay after a preprocessing time in $\mathbf{O}\big(|\mathcal{D}| \times |\mathcal{A}|\big)$. In this final section, we briefly discuss lower bounds and potential leads to improve our upper bounds.

It is unlikely that the preprocessing time of our algorithm can be improved by a polynomial factor. Indeed, several results [5, 10] show that, under the Strong Exponential Time Hypothesis, deciding whether a word w matches a regular expression R cannot be done in $\mathbf{O}\big(|w|^{1-\varepsilon} \times |R|\big)$ nor $\mathbf{O}\big(|w| \times |R|^{1-\varepsilon}\big)$ for any $\varepsilon > 0$. Deciding whether Distinct Shortest Walk has at least one output subsumes this problem. Hence, under SETH, the preprocessing time of Distinct Shortest Walk cannot belong to $\mathbf{O}\big(|\mathcal{D}|^{1-\varepsilon} \times |R|\big)$ nor $\mathbf{O}\big(|\mathcal{D}| \times |R|^{1-\varepsilon}\big)$, for any $\varepsilon > 0$. However, it might be possible to reduce it by a polylogarithmic factor. Indeed, Myers [19] provided an algorithm in $\mathbf{O}\Big(\frac{|R| \times |w|}{\log |w|} + |w|\Big)$ for the former problem. It was later improved by Bille and Thorup[6] to run in $\mathbf{O}\Big(\frac{|R| \times |w|}{\log^{1.5} |w|} + |w|\Big)$. These results provide interesting leads for improving our algorithm.

A significant part in the delay comes from the time taken to actually write down the output in full. However, it is likely that most walks have large parts in common, especially if the set of answers is larger than the size of the database. In that case, one may significantly decrease the delay by outputting only the difference with the previous output. In that case, the order in which the walks are produced is crucial. In a recent article [2], Amarilli and Monet showed how to find efficient orders for enumerating a regular language given as an automaton. Using similar techniques might allow to significantly reduce the (amortized) delay of our algorithm.

ACKNOWLEDGMENTS

We warmly thank Laurent Bulteau for explaining to us that the algorithm we were using for sorting did not actually involve sorting. We also thank Etienne Collet for implementing early versions of the algorithm described here. The authors acknowledge the support of *Agence Nationale de la Recherche* under grant *QUID*, code ANR-18-CE40-0031.

REFERENCES

[1] Margareta Ackerman and Jeffrey Shallit. 2009. Efficient enumeration of words in regular languages. *Theoretical Computer Science* 410, 37 (2009), 3461–3470. https://doi.org/10.1016/j.tcs.2009.03.018 Implementation and Application of Automata (CIAA 2007).

[2] Antoine Amarilli and Mikaël Monet. 2023. Enumerating Regular Languages with Bounded Delay. In *40th International Symposium on Theoretical Aspects of Computer Science (STACS'23) (LIPIcs, Vol. 254)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:18. https://doi.org/10.4230/LIPIcs.STACS.2023.8

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. ACM, 1421–1432.

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

[5] Arturs Backurs and Piotr Indyk. 2016. Which Regular Expression Patterns Are Hard to Match?. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 457–466. https://doi.org/10.1109/FOCS.2016.56

[6] Philip Bille and Mikkel Thorup. 2009. *Faster Regular Expression Matching*. Lecture Notes in Computer Science, Vol. 5555. Springer, Berlin, Heidelberg, 171–182. https://doi.org/10.1007/978-3-642-02927-1_16

[7] Anna Brügemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science* 120 (1993), 197–213.

[8] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *SIGMOD'87*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM, 323–330. https://doi.org/10.1145/38713.38749

[9] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. http://arxiv.org/abs/1901.08248 Preprint arXiv:1901.08248.

[10] Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. 2023. On the Complexity of String Matching for Graphs. *ACM Trans. Algorithms* 19, 3, Article 21 (apr 2023), 25 pages. https://doi.org/10.1145/3588334

[11] Benjamín Farías, Carlos Rojas, and Domagoj Vrgoč. 2023. Evaluating Regular Path Queries in GQL and SQL/PGQ: How Far Can The Classical Algorithms Take Us? (2023). https://doi.org/10.48550/arXiv.2306.02194 Preprint.

[12] International Organization for Standardization. 2024. *GQL*. Standard under development ISO/IEC DIS 39075. https://www.iso.org/standard/76120.html To appear.

[13] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. GPC: A Pattern Calculus for Property Graphs. In *PODS'23*. https://arxiv.org/abs/2210.16580

[14] Nadime Francis and Victor Marsault. 2023. Enumerating regular languages in radix order : Revisiting the Ackerman-Shallit algorithm. ArXiv:2310.13309.

[15] M.L. Fredman and R.E. Tarjan. 1984. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. (1984), 338–346. https://doi.org/10.1109/SFCS.1984.715934

[16] Property Graph Query Language. 2021. PGQL 2.0 Specification. https://pgql-lang.org/spec/2.0/

[17] Wim Martens, Matthias Niewerth, Tina Popp, Carlos Rojas, Stijn Vansummeren, and Domagoj Vrgoč. 2023. Representing Paths in Graph Database Pattern Matching. In *VLDB'23*, Vol. 16. 14 pages. https://doi.org/10.14778/3587136.3587151

[18] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria (LIPIcs, Vol. 98)*, Benny Kimelfeld and Yael Amsterdamer (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:21. https://doi.org/10.4230/LIPIcs.ICDT.2018.19

[19] Gene Myers. 1992. A Four Russians algorithm for regular expression pattern matching. *J. ACM* 39, 2 (April 1992), 432–448. https://doi.org/10.1145/128749.128755

[20] Tina Popp. 2022. *Evaluation and Enumeration of Regular Simple Path and Trail Queries*. Ph. D. Dissertation. Bayreuth. https://epub.uni-bayreuth.de/6606/

[21] Yann Strozecki. 2021. *Enumeration Complexity: Incremental Time, Delay and Space*. Université de Versailles – Saint-Quentin-en-Yvelines. Habilitation thesis.

[22] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (jun 1968), 419–422. https://doi.org/10.1145/363347.363387

[23] TigerGraph. 2023. GSQL Language Reference (version 3.9). https://docs.tigergraph.com/gsql-ref/3.9/intro/

A REDUCTION FROM DISTINCT SHORTEST WALKS TO ALL SHORTEST WORDS

Appendix shows how existing work can be used to solve Distinct Shortest Walks, albeit with a worse complexity than what we achieve in this paper. More precisely, we explain how the problem

reduces to All Shortest Words, a tasks which consists in enumerating all words of minimal length that are accepted by a given NFA in lexicographic order. Formally, All Shortest Words is defined as follows:

All Shortest Words
• Input: A nondeterministic automaton $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$. • Output: Enumerate all shortest words in $L(\mathcal{A})$, without duplicates and in lexicographic order.

The best known algorithm to solve All Shortest Words is given in [1]. However, Ackerman and Shallit did not explicitly write their algorithm in the enumeration complexity framework. This is done in the technical report [14], which lead to the the following theorem.

Theorem 21. *All Shortest Words can be enumerated with* $\mathbf{O}\big(\lambda \times |\Delta| + \lambda \times |Q|^2\big)$ *preprocessing and* $\mathbf{O}\big(\lambda \times |\Delta|\big)$ *delay, where* λ *is the length of any shortest word in* $L(\mathcal{A})$.

We show how Distinct Shortest Walks reduces to All Shortest Words. This reduction follows very closely the ideas in [18]. The authors were mostly interested in showing that the problem can be enumerated with polynomial delay, and hence did not look at the precise complexity. Moreover, our data model slightly differs from theirs. Thus, for the sake of completeness, we chose to rewrite their proof directly on our model, and then give the corresponding complexity. This leads to Theorem 1 given in the introduction:

Theorem 1 ([18] combined with [14]). *Let* $\mathcal{A}$ *be a nondeterministic automaton,* $\mathcal{D}$ *a database, and* s, t *two vertices in* $\mathcal{D}$. *Distinct Shortest Walks*$(\mathcal{D},\mathcal{A},s,t)$ *can be enumerated with delay in* $\mathbf{O}\big(|\mathcal{D}| \times |\Delta| \times \lambda\big)$ *after a preprocessing in* $\mathbf{O}\big(|Q|^2 \times |V|^2 \times \lambda + |\Delta| \times |\mathcal{D}| \times \lambda\big)$, *where* λ *is the length of a shortest walk,* Q *is the state set of* $\mathcal{A}$, *and* Δ *is the transition table of* $\mathcal{A}$.

Let $\mathcal{D} = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ be a database, $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$ an automaton, and s, t two vertices in $\mathcal{D}$.

We define a new automaton $\mathcal{A}' = (\Sigma', Q', \Delta', I', F')$ as follows:

- $\Sigma' = E$;
- $Q' = V \times Q$;
- $\Delta' = \left\{ (v_1, q_1), e, (v_2, q_2) \;\middle|\; \begin{array}{l} \textsc{Src}(e) = v_1, \textsc{Tgt}(e) = v_2 \\ \exists a \in \textsc{Lbl}(e), (q_1, a, q_2) \in \Delta \end{array} \right\}$
- $I' = \{ s \} \times I$
- $F' = \{ t \} \times F$

We conclude the reduction by remarking that there is a one-to-one mapping from words in $L(\mathcal{A}')$ to walks from s to t that match $\mathcal{A}$. After each output of the algorithm for All Shortest Words, we need to reconstruct the corresponding path. This is done in time $\mathbf{O}(\lambda)$ by simply retrieving the source and target vertices of each edge. Hence, this is negligible when compared to the delay, which is $\mathbf{O}\big(\lambda \times |\Delta'|\big) = \mathbf{O}\big(\lambda \times |E| \times |\Delta|\big)$.

Constructing $\mathcal{A}'$ takes time $\mathbf{O}\big(|V| \times |Q| + |\Delta| \times |E|\big)$. Again, this is negligible when compared to the delay of All Shortest Words, in $\mathbf{O}\big(\lambda \times \Delta' + \lambda \times |Q'|^2\big)$. Altogether, the delay is in

$$\mathbf{O}\big(\lambda \times |\Delta| \times |E| + \lambda \times |V|^2 \times |Q|^2\big) .$$

B PROOFS

Appendix B contains the proofs of all lemmas in the body of the article.

Recall that, in all statements, we have a fixed automaton $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$, a database $\mathcal{D} = (\Sigma, V, E, \textsc{Src}, \textsc{Tgt}, \textsc{Lbl})$ and two vertices s and t of $\mathcal{D}$. Moreover, we assume that there exists at least one walk from s to t that matches $\mathcal{A}$, and let λ denote the minimal length of any such walk.

B.1 Proof of Lemma 10

Lemma 10. *At the end of Annotate, the following properties hold for all $p, q \in Q$, $u \in V$ and $e \in E$:*

(1) $L_u[p] = \min(\{\ \textsc{Len}(w) \mid w \in \textsc{Walks}^{\leq\lambda}(\mathcal{D}), s \xrightarrow{w} u \text{ and } p \in \Delta(I, \textsc{Lbl}(w))\ \})$

(2) $q \in B_u[p][i]$ *if and only if there exists a walk $w = w' \cdot e$ from s to u such that:*

- $\textsc{Len}(w) = L_u[p]$
- $\textsc{TgtIdx}(e) = i$
- $q \in \Delta(I, \textsc{Lbl}(w'))$
- $p \in \Delta(q, \textsc{Lbl}(e))$

(3) $B_u[p][\textsc{TgtIdx}(e)]$ *is of size at most* $\sum_{a\in\Sigma} |\Delta^{-1}(a, p)|$.

Proof. The proof of 3 comes from the fact that each pair (v, q) is explored at most once during Annotate at line 21. Thus, for $e \in \textsc{Out}(v)$ and $a \in \textsc{Lbl}(e)$, each element $p \in \Delta(q, a)$ adds at most one item in $B_u[p][\textsc{TgtIdx}(e)]$ at line 30 or line 32, which yields the required bound.

The proof of 1 and 2 are more involved, and require a stronger auxiliary statement.

For $\ell > 0$, let $L_u^{(\ell)}$, $B_u^{(\ell)}$ and $\text{next}^{(\ell)}$ respectively denote the contents of L_u, B_u and next at the end of the ℓ-th step of Annotate, with $L_u^{(0)}$, $B_u^{(0)}$ and $\text{next}^{(0)}$ being their respective contents after the initialization, at line 16.

We prove the following properties by induction over $\ell \leq \lambda$, which immediately imply the required properties when applied with $\ell = \lambda$.

(1) $L_u^{(\ell)}[p] = \min(\{\ \textsc{Len}(w) \mid w \in \textsc{Walks}^{\leq\ell}(\mathcal{D}), s \xrightarrow{w} u \text{ and } p \in \Delta(I, \textsc{Lbl}(w))\ \})$

(2) $q \in B_u^{(\ell)}[p][i]$ if and only if there exists a walk $w = w' \cdot e$ from s to u such that:

- $\textsc{Len}(w) = L_u^{(\ell)}[p]$
- $\textsc{TgtIdx}(e) = i$
- $q \in \Delta(I, \textsc{Lbl}(w'))$
- $p \in \Delta(q, \textsc{Lbl}(e))$

($\star$) $(u, p) \in \text{next}^{(\ell)}$ if and only if $L_u^{(\ell)}[p] = \ell$

Initialization step: it is immediate that the three properties hold for $\ell = 0$, since $B_u^{(0)}$ is empty for each vertex u and $L_s^{(0)}$ correctly reflects the fact that the only walk of length 0 starting at s has an empty label and ends in s.

Induction step: assume that the three properties hold for some $\ell \geq 0$ and that there is an $(\ell + 1)$-th step. In other words, the algorithm did not end at step ℓ, ie. $\ell < \lambda$.

$\Rightarrow$: (1) Assume that $L_u^{(\ell+1)}[p] = k$ for some u, p and k. Remark that once a key-value pair is added to L_u, the algorithm never replaces or removes it. Thus, there are two cases:

- Case 1: at step ℓ, we already have $L_u^{(\ell)}[p] = k$. In that case, the induction hypothesis immediately gives the desired property for $L_u^{(\ell+1)}[p] = k$.
- Case 2: at step ℓ, $L_u^{(\ell)}[p]$ is undefined. In that case, it is defined during step $\ell + 1$, after the test at line 25 and $k = \ell + 1$. At this point in the algorithm, we know that there exist $(v, q) \in \text{next}^{(\ell)}$ and $e \in \textsc{Out}(v)$ such that $u = \textsc{Tgt}(e)$ and $p \in \Delta(q, \textsc{Lbl}(e))$.
  Since $(v, q) \in \text{next}^{(\ell)}$, the induction hypothesis implies that $L_v[q] = \ell$. Thus there exists a walk w going from s to v with $q \in \Delta(I, \textsc{Lbl}(w))$. Moreover, w is of length ℓ, which is minimal among all walks with the same endpoints that can reach state q at v.

Thus, $w \cdot e$ is a walk from s to u with $p \in \Delta(I, \textsc{Lbl}(w \cdot e))$. It is of length $\ell+1$, which is indeed minimal, otherwise the induction hypothesis would not allow $L_u^{(\ell)}[p]$ to be undefined.

(2) Similarly, assume that $q \in B_u^{(\ell+1)}[p][\textsc{TgtIdx}(e)]$ for some u, p, q and e. Then, either it was already true at step ℓ and the induction hypothesis immediately gives the result, or q was appended during step $\ell+1$, at line 30 or at line 32. In both cases, this only happens if $L_u^{(\ell+1)}[p]$ is defined, so that the previous reasoning applies and yields the required walk.

($\star$) Finally, assume that $(u, p) \in \text{next}^{(\ell+1)}$. Once again, this can only happen when $L_u^{(\ell+1)}[p]$ is defined, and the previous reasoning yields $L_u^{(\ell+1)}[p] = \ell+1$.

$\Leftarrow$: Assume that there exists a walk $w \in \textsc{Walks}^{\leq \ell+1}(\mathcal{D})$ such that $s \xrightarrow{w} u$ and $p \in \Delta(I, \textsc{Lbl}(w))$ for some vertex u and state p. Moreover, assume that w is of minimal length k among such walks.

- Case 1: $k < \ell+1$. Then $w \in \textsc{Walks}^{\leq \ell}(\mathcal{D})$. In that case, the induction hypothesis yields $L_u^{(\ell)}[p] = k$. Since the algorithm never removes a key-value pair in L, we immediately get $L_u^{(\ell+1)}[p] = k$. Similarly, if the conditions of (2) are satisfied for w and some w', e and p, then the induction hypothesis yields $q \in B_u^{(\ell)}[p][\textsc{TgtIdx}(e)]$, from which we get $q \in B_u^{(\ell+1)}[p][\textsc{TgtIdx}(e)]$. Finally, there is nothing to prove for $\text{next}^{(\ell+1)}$, since $k < \ell+1$.
- Case 2: $k = \ell+1$. In that case, there exists w', a vertex v and an edge $e \in \textsc{Out}(v)$ such that $w = s \xrightarrow{w'} v \xrightarrow{e} u$. Since $p \in \Delta(I, \textsc{Lbl}(w))$, there exists $q \in \Delta(I, \textsc{Lbl}(w'))$ such that $p \in \Delta(q, \textsc{Lbl}(e))$.
  Remark that w' is necessarily of minimal length ℓ among walks going from s to v that can reach state q, otherwise w would not be of minimal length. Thus, the induction hypothesis yields $L_v^{(\ell)}[q] = \ell$ and $(v, q) \in \text{next}^{(\ell)}$. This means that, at step $\ell+1$ of Annotate, (v, q) is added to current. It simply remains to check that e and p satisfy all the conditions so that $L_u^{(\ell+1)}[p] = \ell+1$, $q \in B_u^{(\ell+1)}[p][\textsc{TgtIdx}(e)]$ and (u, p) is added to $\text{next}^{(\ell+1)}$. The only hurdle is to prove that, during this step, either $L_u[p] = \ell+1$ or $L_u[p]$ is undefined, otherwise the induction hypothesis would once again contradict the minimality of w. □

B.2 Proof of Lemma 11

Lemma 11. *At the end of Trim, the following properties hold for all $p \in Q$, $u \in V$, $e, e' \in \textsc{In}(u)$ and lists X, X' over Q:*

(1) $(e, X) \in C_u[p]$ if and only if $X = B_u[p][\textsc{TgtIdx}(e)]$ and $X \neq \emptyset$.
(2) $C_u[p]$ is sorted in increasing $\textsc{TgtIdx}(e)$ order, that is, if (e, X) appears before (e', X') in $C_u[p]$, then $\textsc{TgtIdx}(e) < \textsc{TgtIdx}(e')$. In particular, $e \neq e'$.
(3) If $(e, X) \in C_u[p]$, then X is of size at most $\sum_{a \in \Sigma} |\Delta^{-1}(a, p)|$.

Proof. The proof of (1) immediately follows from the pseudocode of Trim. Indeed, Trim browses all B_u exhaustively[4] and enqueues precisely the pairs $(e, B_u[p][\textsc{TgtIdx}(e)])$ for which $B_u[p][\textsc{TgtIdx}(e)]$ is not empty.

Moreover, B_u is explored in the same order as $\textsc{In}(u)$ (at line 38), thus pairs (e, X) are enqueued in C_u in the same order as e appears in $\textsc{In}(u)$. Thus, (2) follows from the definition of $\textsc{TgtIdx}(e)$, which is precisely the position where e appears in $\textsc{In}(u)$.

Finally, (3) immediately follows from (1) together with Lemma 10, item 3. □

[4]Indeed, $B_u[p]$ ranges over $\textsc{In}(u)$

B.3 Proof of Lemma 15

Lemma 15. *Let $e \cdot w$ be a node of $\mathcal{T}$ for some edge e and walk w. Let $u = \textsc{Tgt}(e)$. For every p, we denote by X_p the unique list of states such that $(e, X_p) \in C_u[p]$ if such a list exists, or $X_p = \emptyset$ otherwise. Then, the following holds.*

$$S(e \cdot w) = \bigcup_{p \in S(w)} set(X_p)$$

Proof. Let e, u, w be defined as in the statement of the lemma. Let $v = \textsc{Src}(e)$.

$\subseteq$: Let $q \in S(e \cdot w)$. By definition of S, there exists a walk w_q such that $w_q \cdot e \cdot w \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$, with $q \in \Delta(I, \textsc{Lbl}(w_q))$ and $q \in \Delta^{-1}(\textsc{Lbl}(e \cdot w), F)$.

Remark that w_q is a walk from s to v with $q \in \Delta(I, \textsc{Lbl}(w_q))$. Moreover, w_q is of minimal length among such walks, otherwise $w_q \cdot e \cdot w \notin [\![\mathcal{A}]\!](\mathcal{D}, s, t)$.

We know $\Delta(q, \textsc{Lbl}(e)) \neq \emptyset$, otherwise we could not have $q \in \Delta^{-1}(\textsc{Lbl}(e \cdot w), F)$. Thus, let $p \in \Delta(q, \textsc{Lbl}(e))$. Then $w_q \cdot e$ is a walk from s to u with $p \in \Delta(I, \textsc{Lbl}(w_q))$, and it is of minimal length among such walks, for similar reasons.

We can apply Lemma 10 to $w_q \cdot e$. Thus, $q \in B_u[p][\textsc{TgtIdx}(e)]$.

Then, Lemma 11 provides a set X_p such that $q \in X_p$ and $(e, X_p) \in C_u[p]$.

It remains to remark that $p \in S(w)$. Indeed, in the definition of S, we can choose the walk $w_p = w_q \cdot e$ as a witness.

$\supseteq$: Let $q \in X_p$ for some $p \in S(w)$.

Since $p \in S(w)$, it means that there exists a walk $s \xrightarrow{w_p} u$ that reaches p at u, such that $w_p \cdot w \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$, with $p \in \Delta^{-1}(\textsc{Lbl}(w), F)$. As for the direct inclusion, this implies that w_p is of minimal length among such walks.

Since $q \in X_p$, Lemma 11 implies that $q \in B_u[p][\textsc{TgtIdx}(e)]$.

Thus, Lemma 10 shows that there exists a walk $w_2 = w_2' \cdot e$ from s to u with $q \in \Delta(I, \textsc{Lbl}(w_2'))$ and $p \in \Delta(q, \textsc{Lbl}(e))$. Moreover, w_2 is of minimal length among walks that reach p at u. Since $p \in \Delta^{-1}(\textsc{Lbl}(w), F)$, we deduce that $w_2 \cdot w$ reaches a final state at t. Additionally, $\textsc{Len}(w_p) = \textsc{Len}(w_2)$ (as they both have minimal length), thus $w_2 \cdot w \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$.

Then, we simply remark that w_2' is a suitable witness to show that $q \in S(e \cdot w)$ in the definition of S. □

B.4 Proof of Lemma 16

The proof of Lemma 16 requires an additional result:

Lemma 22. *Let w_1 be a node of $\mathcal{T}$ and w_2 be a strict descendant of w_1 such that $\textsc{Src}(w_1) = \textsc{Src}(w_2)$. Then $S(w_1) \cap S(w_2) = \emptyset$.*

Proof. Let w_1 and w_2 be defined as in the statement of the lemma. Since w_2 is a descendant of w_1, by definition of $\mathcal{T}$, there exists a walk w_2' with $\textsc{Len}(w_2') \geq 1$ such that $w_2 = w_2'.w_1$

By contradiction, assume that there exists $q \in S(w_1) \cap S(w_2)$. Then, we know that there exists two walks w_{1q} and w_{2q} such that:

- $w_{1q} \cdot w_1 \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$, with $q \in \Delta(I, \textsc{Lbl}(w_{1q})$ and $q \in \Delta^{-1}(\textsc{Lbl}(w_1), F)$.
- $w_{2q} \cdot w_2 \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$, with $q \in \Delta(I, \textsc{Lbl}(w_{2q})$ and $q \in \Delta^{-1}(\textsc{Lbl}(w_2), F)$.

Now, remark that w_{2q} and w_1 concatenate. Indeed, $\textsc{Tgt}(w_{2q}) = \textsc{Src}(w_2) = \textsc{Src}(w_1)$. Thus, $s \xrightarrow{w_{2q} \cdot w_1} t$. Moreover, $w_{2q} \cdot w_1$ reaches a final state at t, because $q \in \Delta(I, \textsc{Lbl}(w_2))$ and $q \in \Delta^{-1}(\textsc{Lbl}(w_1), F)$, which means that $w_{2q} \cdot w_1$ matches $\mathcal{A}$.

However, $w_{2q} \cdot w_1$ is shorter than $w_{2q} \cdot w_2$. Indeed, $\textsc{Len}(w_{2q} \cdot w_2) = \textsc{Len}(w_{2q} \cdot w_1) + \textsc{Len}(w'_2)$ and $\textsc{Len}(w'_2) \geq 1$. This is a contradiction with $w_{2q} \cdot w_2 \in [\![\mathcal{A}]\!](\mathcal{D}, s, t)$. □

We are now ready to prove Lemma 16.

Lemma 16. *The tree of recursive calls to* Enumerate *is isomorphic to $\mathcal{T}$ in the following sense:* Enumerate*(C, ℓ, w, S) is called exactly once per node w in $\mathcal{T}$. Moreover, the parameters of this call satisfy $\ell = \lambda - \textsc{Len}(w)$ and $S = S(w)$.*

Proof. We prove the result by induction over the tree of recursive calls to Enumerate.

Initialization step. The first call, in Main, is $\textsc{Enumerate}(C, \lambda, \langle t \rangle, S_t)$, where

$$S_t = \{\, q \mid L_t[q] = \lambda \,\} \cap F$$

As stated in the lemma, the walk $\langle t \rangle$ corresponds to the root of $\mathcal{T}$ and we have $\lambda = \lambda - \textsc{Len}(\langle t \rangle)$, since $\langle t \rangle$ is a walk of length 0. It remains to show that $S_t = S(\langle t \rangle)$. Indeed, we have the following equivalences:

$$\begin{aligned} q \in S_t &\Leftrightarrow L_t[q] = \lambda \text{ and } q \in F \\ \text{(Lemma 10)} &\Leftrightarrow \exists w_q, s \xrightarrow{w_q} t, q \in \Delta(I, \textsc{Lbl}(w_q)) \cap F \text{ and } \textsc{Len}(w_q) = \lambda \\ &\Leftrightarrow \exists w_q \in [\![\mathcal{A}]\!](\mathcal{D}, s, t) \text{ and } q \in \Delta(I, \textsc{Lbl}(w_q)) \cap F \\ \text{(with } w_q = w_q \cdot \langle t \rangle) &\Leftrightarrow q \in S(\langle t \rangle) \end{aligned}$$

Induction step. Assume that the property holds for $\textsc{Enumerate}(C, \ell, w, S)$ and all its ancestors in the tree of recursive calls of Enumerate. We now have to show that it holds for its recursive calls.

- Case 1: $\ell = 0$. In that case, Enumerate stops without making recursive calls, at line 46. It remains to show that w has no child in $\mathcal{T}$. Indeed, since $\ell = 0$, the induction hypothesis yields $\textsc{Len}(w) = \lambda$ and we know from the definition of $\mathcal{T}$ that the nodes w of length λ are precisely at the leaves.
- Case 2: $\ell > 0$. This case requires some care, as the same structure C is shared between all calls to Enumerate. Hence, we first have to make sure that previous or concurrent calls will not interfere with the execution of the current call. This is stated in the following claim:

  Claim 16.1. *Calls to* Enumerate *do not make concurrent access to the same data structures, in the following sense:*
  - *At the beginning of a call to* Enumerate*, all queues $C_u[p]$ that will be read during this call are on their starting position.*
  - *If a call to* Enumerate *reads or advances a queue $C_u[p]$, then none of its ancestors reads nor advances the same queue.*
  - *At the end of a call to* Enumerate*, all queues $C_u[p]$ that have been advanced during this call have been restarted.*

  Claim 16.1 (up to the current call) immediately follows from the induction hypothesis together with Lemma 22 and the fact that Enumerate restarts used queues before returning, on line 57.

  We can now reason about the current call $\textsc{Enumerate}(C, \ell, w, S)$. Let $u = \textsc{Src}(w)$, as set at line 44. First, the induction hypothesis yields $S = S(w)$. Thus, from the claim, we deduce that, at the beginning, all queues $C_u[p]$ for $p \in S(w)$ are on their starting position. Hence, the loop at lines 49-58 computes e_{min} as the least e (in TgtIdx order) such that

$(e, X) \in C_u[p]$ for some X and $p \in S(w)$. Indeed, we know from Lemma 11 that $C_u[p]$ is sorted, hence the least e can only appear in the head of the queues. Additionally, it cannot be that all queues are empty. Indeed, w must have a child in $\mathcal{T}$ (otherwise $\textsc{Len}(w) = \lambda$ and $\ell = 0$). Thus, $S(w') \neq \emptyset$ and Lemma 15 ensures that at least one queue is not empty.
Next, the loop at lines 59-66 computes the union of all X such that $(e_{min}, X) \in C_u[p]$ for some $p \in S(w)$, once again thanks to the fact that $C_u[p]$ is sorted. From Lemma 15, we know that this is precisely $S(e_{min} \cdot w)$. Thus, the parameters of the first recursive call at line 67 correctly correspond to the first child of w in $\mathcal{T}$.
For the subsequent calls, simply remark that the loop only advanced the queues that had e_{min} in the head. Thus, we can repeat the same reasoning when e_{min} finds the second least element in $C_u[p]$ for $p \in S(w)$, and so on, until all queues are exhausted. In the end, $e_{min} = nil$, Enumerate restarts all used queues and returns. □

Received December 2023; revised February 2024; accepted March 2024