Explaining Results of Path Queries on Graphs: Single-Path Results for Context-Free Path Queries*

Jelle Hellings[a]

[a]*Department of Computing and Software, McMaster University, 1280 Main St. W., Hamilton, ON L8S 4L7, Canada*

Abstract

Many graph query languages use, at their core, *path queries* that yield node pairs (m, n) that are connected by a path of interest. For the end-user, such node pairs only give limited insight as to *why* this result is obtained, as the pair does not directly identify the underlying path of interest.

In this paper, we propose the *single-path semantics* to address this limitation of path queries. Under single-path semantics, path queries evaluate to a single path connecting nodes m and n and that satisfies the conditions of the query. To put our proposal in practice, we provide an efficient algorithm for evaluating *context-free path queries* using the single-path semantics. Additionally, we perform a short evaluation of our techniques that shows that the single-path semantics is practically feasible, even when query results grow large.

In addition, we explore the formal relationship between the single-path semantics we propose the problem of finding the *shortest string* in the intersection of a regular language (representing a graph) and a context-free language (representing a path query). As our formal results show, there is a distinction between the complexity of the single-path semantics for queries that use a single edge label and queries that use multiple edge labels: for queries that use a single edge label, the length of the shortest path is *linearly upper bounded* by the number of nodes in the graph; whereas for queries that use multiple edge labels, the length of the shortest path has a worst-case *quadratic lower bound*.

Keywords: Graphs, Path Results, Path Queries, Shortest Paths, Context-Free Grammars

1. Introduction

The graph data model is one of the most versatile and natural data models in use: graph-structured data is everywhere and examples can be found in

*An extended abstract of this work was presented at the 2nd International Workshop on Large Scale Graph Data Analytics [1].

Email address: jhellings@mcmaster.ca (Jelle Hellings)

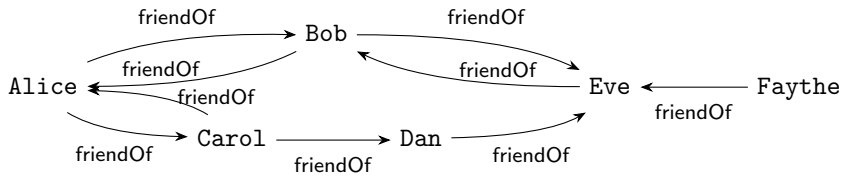

Figure 1: A typical example of graph data: a social network relating people.

family trees, social networks, process models, gene networks, XML data, and RDF data [2–6]. As an example, consider the small social network visualized in Figure 1 in which nodes represent people and edges represent the relationships between people.

A central step in the analysis of such graph data is the ability to *query* the data for relationships of interest. For this purpose, many different query languages have been developed, including XPath for querying XML data [4, 7, 8], SPARQL for querying RDF data [6, 9], the graph query languages GXPath [10], Cypher [11], and Gremlin [12], and formal verification languages such as PDL, KAT, CTL, and LTL [5, 13, 14]. At their core, these graph query languages depend on *path queries* that can be used to express *indirect relationships* that can be derived from the data [3]. Examples of such path queries are the well-known regular path queries [15] and the context-free path queries [14, 16–18]. Unfortunately, path queries are typically evaluated to only a set of node pairs (m, n) that are connected by a path of interest, which gives little insight in the way pairs (m, n) are obtained, limiting their capabilities for graph analytics.

Example 1. Let $\mathfrak{G}$ be the social network visualized in Figure 1. The path query $\mathsf{indirectFriendOf} = \mathsf{friendOf}^+$, expressed by a regular expression, will return the derived relationship summarized in Figure 2. This relationship holds people-pairs (m, n) such that m is a friend of n, or a friend-of-a-friend of n, or a friend-of-a-friend-of-a-friend of n, and so on.

The pair $(\texttt{Alice}, \texttt{Eve})$ is in the result of evaluating $\mathsf{indirectFriendOf}$ on graph $\mathfrak{G}$. Unfortunately, Alice cannot use this result to determine whom of her friends can help her to get in contact with Eve, and Alice will have to further analyze the underlying graph data.

Example 1 illustrates the need to answer path queries with the underlying paths of interest inspected by these queries. To address this need, we propose the *single-path semantics* for evaluating path queries: using the single-path semantics, a path query will evaluate to a shortest path connecting node pair (m, n) (for each node pair in the query result).

Example 2. Consider the setting of Example 1. Evaluation of $\mathsf{indirectFriendOf}$ using the single path semantics can result in the path "`Alice` $\mathsf{friendOf}$ `Bob` $\mathsf{friendOf}$ `Eve`", from which Alice can derive a way to contact Eve. As the single path semantics requires a single and shortest path between node pairs, the path "`Alice` $\mathsf{friendOf}$ `Carol` $\mathsf{friendOf}$ `Dan` $\mathsf{friendOf}$ `Eve`" cannot be in the output.

indirectFriendOf *part one*	
Alice	Alice
Alice	Bob
Alice	Carol
Alice	Eve
Alice	Dan
Bob	Alice
Bob	Bob
Bob	Carol
Bob	Eve
Bob	Dan

indirectFriendOf *part two*	
Carol	Alice
Carol	Bob
Carol	Carol
Carol	Eve
Carol	Dan
Dan	Alice
Dan	Bob
Dan	Carol
Dan	Eve
Dan	Dan

indirectFriendOf *part three*	
Eve	Alice
Eve	Bob
Eve	Carol
Eve	Eve
Eve	Dan
Faythe	Alice
Faythe	Bob
Faythe	Carol
Faythe	Eve
Faythe	Dan

Figure 2: The table representing the result of evaluating the query indirectFriendOf using traditional *relational* semantics.

Although regular path queries such as the one in Example 1 are frequently used in graph querying, they are limited in their expressive power [17]. As we shall illustrate next, the usefulness of a *single and shortest path semantics* extends beyond the limited expressive power of regular path queries, however.

Example 3. Consider a family tree with parentOf and childOf edges. The *context-free path query*

$$\text{Q} \rightarrow \text{parentOf childOf}; \qquad \text{Q} \rightarrow \text{parentOf Q childOf},$$

which cannot be expressed by an equivalent regular path query, yields pairs of family members that are both n-th generation descendant of a common ancestor. For example, this query will return pairs of siblings, first cousins, second cousins, and so on. The query will not return any other pairs of family members, however.

Simply evaluating the query Q does not tell whether a resulting pair represents a pair of siblings, first cousins, second cousins, and so on. To determine this, we need the distance between the pair and their closest common ancestor. When evaluating the above query with a *single and shortest path* semantics, one can directly determine this closest common ancestor and, hence, determine whether a pair represents a sibling pair (path of two edges), a first cousin pair (path of four edges), a second cousin pair (path of six edges), and so on.

The need for the single-path semantics extends beyond the above toy examples. Not only can single-path semantics provide more relevant information to end-users, the single-path semantics can also aid in graph analytics and data exploration, and can be used to provide *data provenance* for traditional path queries [19] by providing paths that show why a path query includes a certain node pair in its output. Furthermore, in the large-scale graph data setting in which many complex path queries are evaluated, there is a need for tools to support query debugging [20], for which the single-path semantics can also be of use.

In this paper, we deal with the issues outlined by proposing the *single-path semantics*. We focus our study on the context-free path queries, as these are a particular powerful type of path queries that can express all typical regular path queries (e.g. [15, 17]). Furthermore, the context-free path queries also have applications in model checking [14], bio-informatics [18], and parser construction [21]. Our contributions are as follows;

1. We formalize the *single-path semantics*.
2. We introduce *graph-annotated grammars* as a finite representation of all paths in a graph that satisfy the conditions of a context-free path query.
3. We propose the MinimizeSet algorithm that can derive the shortest path between a node pair represented by a graph-annotated grammar by finding shortest strings in the graph-annotated grammar.
4. We propose the MinimizeSetGG algorithm that provides a direct and efficient evaluation of context-free path queries on graphs using our single-path semantics.
5. We provide an analysis of the worst-case length of the paths returned by MinimizeSetGG: we prove an interesting formal distinction between path queries that use a single edge label and queries that use multiple edge labels. Specifically, for a graph with n nodes and a query using a context-free grammar $\mathcal{C}$ in Chomsky Normal Form with m non-terminals, we show:
   (a) if either the graph or the query uses only a single edge label, then the worst-case length of the shortest path is *linear* in the size of the graph: the worst-case length is lower bounded by $n2^{m-1}$ and upper bounded by $n2^{2m}$;
   (b) otherwise, the worst-case lower bound on the length of the shortest path is *quadratic* in the size of the graph: the worst-case length is lower bounded by $(n^2 2^m)/32$ (this even if the grammar $\mathcal{C}$ is deterministic and only inspects two edge labels).
6. We implement MinimizeSetGG and evaluate the performance of MinimizeSetGG in practice.

Our results show that MinimizeSetGG can easily answer queries whose results contain tens-of-millions of paths, even if these paths have considerable lengths.

2. Preliminaries

First, we introduce the terminology and notation used throughout this paper.

Strings. Let Σ be a set of symbols that we refer to as the *alphabet*. We call a sequence $s = \sigma_1 \dots \sigma_n$ of symbols, $\sigma_1, \dots, \sigma_n \in \Sigma$, a string over Σ. We write $|s| = n$ to denote the length of s. The empty string is denoted by ϵ and we usually treat individual symbols as strings of length one. The concatenation of two strings s_1 and s_2 is denoted by $s_1 \circ s_2$. We denote the set of all strings over Σ by Σ^*. A *language* over Σ is a (possibly infinite) set of strings over Σ.

Edge-labeled graphs. A *graph* is a triple $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$, in which $\mathcal{V}$ is a finite set of nodes, Σ is a finite set of alphabet symbols used as edge labels, and $\delta \subseteq \mathcal{V} \times \Sigma \times \mathcal{V}$ is a finite set of labeled edges. To simplify presentation, we assume that $\mathcal{V}$ and Σ do not overlap ($\mathcal{V} \cap \Sigma = \emptyset$). A *path* in $\mathfrak{G}$ is a sequence $\pi = n_1\ \sigma_1\ n_2 \ldots n_{u-1}\ \sigma_{u-1}\ n_u$ such that, for every $n_i\ \sigma_i\ n_{i+1}$ in the sequence, $1 \leq i < u$, we have $(n_i, \sigma_i, n_{i+1}) \in \delta$. We write $n_1 \pi n_u$ to indicate that π is a path starting at node n_1 and ending at node n_u. We write $|\pi| = u - 1$ to denote the length of π and we write $\text{trace}(\pi) = \sigma_1 \ldots \sigma_{u-1}$ to denote the trace of π, the string represented by the sequence of edge labels in π.

Finite automaton. Let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph and $m, n \in \mathcal{V}$. We interpret the triple $(\mathfrak{G}, m, n)$ as a finite automata with initial state m and final state n. The language of this finite automata is defined by

$$\mathcal{L}(\mathfrak{G}; m, n) = \{\text{trace}(\pi) \mid m \pi n \text{ is a path in } \mathfrak{G}\}.$$

Context-free grammars. A *grammar* is a triple $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$, in which $\mathcal{N}$ is a set of non-terminals, Σ is a finite set of alphabet symbols, and $\mathcal{P}$ is a set of production rules. We require that $\mathcal{N}$ and Σ do not overlap ($\mathcal{N} \cap \Sigma = \emptyset$). To simplify presentation, we assume that grammars are in Chomsky Normal Form [22], and we exclude the derivation of the empty string ϵ. All results in this work can be generalized to the setting where context-free languages can define ϵ. Hence, the set of production rules, $\mathcal{P}$, consists of production rules of the form $\text{A} \mapsto \text{B C}$ or $\text{A} \mapsto \sigma$, in which $\text{A}, \text{B}, \text{C} \in \mathcal{N}$ and $\sigma \in \Sigma$.

Each non-terminal in $\mathcal{N}$ represents a language over Σ: the production rules in $\mathcal{P}$ describe how to produce strings out of non-terminals via rewrite steps. To illustrate this, consider a string $s = s_1 \circ \text{A} \circ s_2$ in which $s_1, s_2 \in (\mathcal{N} \cup \Sigma)^*$ and $\text{A} \in \mathcal{N}$. If there exists a production rule $(\text{A} \mapsto s') \in \mathcal{P}$, then s can be rewritten into $s_1 \circ s' \circ s_2$ by applying the rewrite $\text{A} \mapsto s'$. We write $s \rightarrow^*_{\mathcal{P}} s'$ if s can be rewritten into s' by a finite number of rewrites using production rules in $\mathcal{P}$. The *language* of non-terminal $\text{A} \in \mathcal{N}$ is defined by $\mathcal{L}(\mathcal{C}; \text{A}) = \{s \mid s \in \Sigma^* \wedge \text{A} \rightarrow^*_{\mathcal{P}} s\}$.

Example 4. Consider the grammar $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ in which $\mathcal{N} = \{\text{A}\}$, $\Sigma = \{\mathsf{friendOf}\}$, and $\mathcal{P} = \{\text{A} \rightarrow \mathsf{friendOf}, \text{A} \rightarrow \text{A A}\}$. The language $\mathcal{L}(\mathcal{C}; \text{A})$ is equivalent to the language of the regular expression $\mathsf{indirectFriendOf}$ of Example 1.

Path queries. A *path query* q is specified by a language $\mathcal{L}$ that contains all traces of paths of interest, e.g., via a regular expression (RPQs) or via a context-free grammar (CFPQs). Given a graph $\mathfrak{G}$, we write $[\![q]\!]_{\mathfrak{G}}$ to denote the evaluation of query q on graph $\mathfrak{G}$ using the standard *relational semantics* [10, 15, 17, 23–25]: we have

$$[\![q]\!]_{\mathfrak{G}} = \{(m, n) \mid \exists \text{ path } m \pi n \text{ in } \mathfrak{G} \text{ with } \text{trace}(\pi) \in \mathcal{L}\}.$$

Hence, $[\![q]\!]_{\mathfrak{G}}$ is the set of all node pairs in $\mathfrak{G}$ that are connected by paths whose trace is in $\mathcal{L}$.

Example 5. In Example 1, the query indirectFriendOf was expressed by the regular expression $\mathsf{friendOf}^+$ with language

$$\mathcal{L} = \{\mathsf{friendOf}, \mathsf{friendOf} \circ \mathsf{friendOf}, \mathsf{friendOf} \circ \mathsf{friendOf} \circ \mathsf{friendOf}, \dots\}.$$

We have $(\texttt{Alice}, \texttt{Eve}) \in [\![\mathsf{indirectFriendOf}]\!]_{\mathfrak{G}}$, with $\mathfrak{G}$ the graph in Figure 1, as there exists a path $\pi =$ "Alice friendOf Bob friendOf Eve" with $\text{trace}(\pi) \in \mathcal{L}$. In Example 4, we have shown how to express the above language $\mathcal{L}$ by a context-free grammar $\mathcal{C}$. Hence, we have $[\![\mathsf{indirectFriendOf}]\!]_{\mathfrak{G}} = [\![\mathcal{C}]\!]_{\mathfrak{G}}$.

Remark 1. Generalizations of the regular path queries and context-free graph queries can support traversing edges in both directions, e.g., taking the *inverse* of the parentOf edge would allow one to interpret the edge as a childOf edge. This is typically done by adding, for every edge label σ, symbols of the form σ^{-1} to the alphabet and by interpreting these symbols σ^{-1} as the inverse of the edges labeled σ [23–25]. As each inversed edge symbol can be interpreted as another edge relation, the results in this paper generalize to the setting with inversed edges. Hence, to simplify notation, we did not included inversed edges in our formalism.

3. The single-path semantics

In Section 2, we already introduced the typical *relational semantics* of path queries. Unfortunately, the step toward *path-based semantics*—in which a path query yields paths $m\pi n$ instead of node pairs (m, n)—is not straightforward. Even in basic situations, the resulting set of paths can already be unbounded in size, making it impossible to simply evaluate to such a set:

Example 6. Consider Example 1 and the graph $\mathfrak{G}$ visualized in Figure 1. This graph is cyclic, as there is a path "Alice friendOf Carol friendOf Dan friendOf Eve friendOf Bob friendOf Alice". Hence, we can make paths of arbitrary lengths that match the query indirectFriendOf, and the set of all paths matching the query is unbounded in size.

One way to assure that the set of paths considered is finite is by restricting the types of paths to *simple paths* (paths without node repetition) or a *simple walk* (paths without edge repetitions). Such a restriction of the types of path used to evaluate path queries changes the semantics of the path queries drastically, however. Moreover, it is well-known that such restrictions make query evaluation prohibitively expensive [3, 26–28]. Hence, restricting the types of paths used to evaluate path queries defeats the purpose of path-based semantics as a data provenance and debugging tool for normal path queries.

As an alternative, we consider restricting the number of paths in the result, e.g., to a single path per node pair. This restriction will also assure a finite result. As Example 6 already shows, individual paths in such a finite result set can still have a practically unbounded length. To address this, we choose to return a single as-short-as-possible path for each node-pair (m, n):

Definition 3.1. Let q be a path query specified by language $\mathcal{L}$ and let $\mathfrak{G}$ be a graph. The evaluation of q on $\mathfrak{G}$ using the *single-path semantics*, denoted by $\text{single}(q|_{\mathfrak{G}})$, yields, for every $(m,n) \in [\![q]\!]_{\mathfrak{G}}$, a single shortest path $m\pi n$ in $\mathfrak{G}$ such that $\text{trace}(\pi) \in \mathcal{L}$. (Hence, for every other path $m\pi' n$ in $\mathfrak{G}$ with $\text{trace}(\pi') \in \mathcal{L}$, we have $|\pi| \leq |\pi'|$).

Next, we work toward evaluating context-free path queries using the single-path semantics in three steps. First, in Section 3.1, we show that all paths of interest of a context-free path query can be represented by a grammar. Then, in Section 3.2, we propose MinimizeSet, an algorithm for computing a shortest string in a context-free language. Finally, in Section 3.3, we combine these results and show how to evaluating context-free path queries using the single-path semantics.

3.1. Representing the paths of interest of a path query

Let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph and $(m,n) \in \mathcal{V}$ a pair of nodes. There is a close correspondence between labeled graphs and finite automata and we can easily interpret $(\mathfrak{G}, m, n)$ as a finite automaton with initial state m and final state n. The language of this finite automata is $\mathcal{L}(\mathfrak{G}; m, n) = \{\text{trace}(\pi) \mid m\pi n \text{ is a path in } \mathfrak{G}\}$. It is well-known that the intersection of a finite automaton and a grammar can be represented by another context-free grammar:

Lemma 3.2 (Bar-Hillel et al. [29])**.** *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, let $\text{A} \in \mathcal{N}$, and let $m, n \in \mathcal{V}$. The language $\mathcal{L}(\mathcal{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$ can be represented by a grammar.*

Lemma 3.2 guarantees that there is a finite representation of the set of all strings $\mathcal{L}(\mathcal{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$, each such string $s \in \mathcal{L}(\mathcal{C}; \text{A})$ representing the trace $s = \text{trace}(\pi)$ of a path $m\pi n$ in $\mathfrak{G}$. There can be several paths with the same trace, however. Hence, Lemma 3.2 does not guarantee that we can effectively derive the underlying paths π matching the strings s represented by $\mathcal{L}(\mathcal{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$. To improve on the guarantees of Lemma 3.2, we show the existence of *graph-annotated grammars* that can directly represent the paths π:

Definition 3.3. Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar and let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph. We denote triples $(\text{A}, m, n) \in \mathcal{N} \times \mathcal{V}^2$ by $\text{A}|_{mn}$. The *annotated grammar* over $(\mathcal{C}, \mathfrak{G})$ is the grammar $\mathcal{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$ in which

1. $\mathcal{N}|_{\mathfrak{G}} = \{\text{A}|_{mn} \in \mathcal{N} \times \mathcal{V}^2 \mid \mathcal{L}(\mathcal{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n) \neq \emptyset\}$;
2. $\mathcal{P}|_{\mathfrak{G}} = P_\Sigma \cup P_\mathcal{N}$ with

$$P_\Sigma = \{\text{A}|_{mn} \mapsto \sigma \mid (m, \sigma, n) \in \delta \wedge (\text{A} \mapsto \sigma) \in \mathcal{P} \wedge \text{A}|_{mn} \in \mathcal{N}|_{\mathfrak{G}}\};$$
$$P_\mathcal{N} = \{\text{A}|_{mn} \mapsto \text{B}|_{mo}\ \text{C}|_{on} \mid (\text{A} \mapsto \text{B C}) \in \mathcal{P} \wedge \text{A}|_{mn}, \text{B}|_{mo}, \text{C}|_{on} \in \mathcal{N}|_{\mathfrak{G}}\}.$$

The notation $\text{A}|_{mn}$ denotes a node-annotated non-terminal: any string produced from rewriting this non-terminal is a trace of a path $m\pi n$. As rewriting $\text{A}|_{mn}$ eventually leads to rewrite steps using production rules in P_Σ, which represent single edges in $\mathfrak{G}$, the path π can be derived by keeping track of these node-annotations. As $|\mathcal{N}|_{\mathfrak{G}}| \leq |\mathcal{N}||\mathcal{V}|^2$, $|P_\Sigma| \leq |\mathcal{P}||\delta|$, and $|P_\mathcal{N}| \leq |\mathcal{P}||\mathcal{V}|^3$, an annotated grammar has a size bounded by the size of the graph and the grammar.

Example 7. Let $\mathfrak{G}$ be the graph visualized in Figure 1 and $\mathcal{C}$ the grammar of Example 4. We construct the annotated grammar $\mathcal{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$. For brevity, we refer to each person by the first letter of their name. We have

$$\mathcal{N}|_{\mathfrak{G}} = \{\mathrm{Q}|_{mn} \mid m, n \in \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}\} \cup \{\mathrm{Q}|_{\mathrm{F}n} \mid n \in \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}\}.$$

We have $\mathcal{P}|_{\mathfrak{G}} = P_\Sigma \cup P_\mathcal{N}$, in which P_Σ represents all edges in $\mathfrak{G}$ and $P_\mathcal{N}$ represents all ways in which paths in $\mathfrak{G}$ can be combined. E.g., $\mathrm{Q}|_{\mathtt{AB}} \in P_\Sigma$, as $(\mathtt{Alice}, \mathsf{friendOf}, \mathtt{Bob})$ is and edge in $\mathfrak{G}$, and $(\mathrm{Q}|_{\mathtt{AB}} \mapsto \mathrm{Q}|_{\mathtt{AD}}\ \mathrm{Q}|_{\mathtt{DB}}) \in P_\mathcal{N}$, as there is a $\mathsf{friendOf}$-labeled path from $\mathtt{Alice}$ to $\mathtt{Dan}$ and another $\mathsf{friendOf}$-labeled path from $\mathtt{Dan}$ to $\mathtt{Bob}$. To produce a path from Alice to Eve, we use $\mathcal{C}|_{\mathfrak{G}}$:

$$\begin{aligned}
\mathrm{Q}|_{\mathtt{AliceEve}} &\to^*_{\mathcal{P}|_{\mathfrak{G}}} \{\text{Rewrite } \mathrm{Q}|_{\mathtt{AliceEve}} \mapsto \mathrm{Q}|_{\mathtt{AliceCarol}}\ \mathrm{Q}|_{\mathtt{CarolEve}}\} \\
\mathrm{Q}|_{\mathtt{AliceCarol}}\ \mathrm{Q}|_{\mathtt{CarolEve}} &\to^*_{\mathcal{P}|_{\mathfrak{G}}} \{\text{Rewrite } \mathrm{Q}|_{\mathtt{CarolEve}} \mapsto \mathrm{Q}|_{\mathtt{CarolDan}}\ \mathrm{Q}|_{\mathtt{DanEve}}\} \\
\mathrm{Q}|_{\mathtt{AliceCarol}}\ \mathrm{Q}|_{\mathtt{CarolDan}}\ \mathrm{Q}|_{\mathtt{DanEve}} &\to^*_{\mathcal{P}|_{\mathfrak{G}}} \{\text{Rewrite } \mathrm{Q}|_{\mathtt{AliceCarol}} \mapsto \mathsf{friendOf}, \dots\} \\
&\qquad\qquad \mathsf{friendOf} \circ \mathsf{friendOf} \circ \mathsf{friendOf}.
\end{aligned}$$

The annotations in each node-annotated non-terminal carry information that can be used to map strings in $\mathcal{C}|_{\mathfrak{G}}$ to paths in the underlying graph $\mathfrak{G}$: in this rewrite, we derived a path from $\mathtt{Alice}$ to $\mathtt{Eve}$ in graph $\mathfrak{G}$ of length three, namely the path "$\mathtt{Alice}$ $\mathsf{friendOf}$ $\mathtt{Carol}$ $\mathsf{friendOf}$ $\mathtt{Dan}$ $\mathsf{friendOf}$ $\mathtt{Eve}$".

Using induction, we can prove that graph-annotated grammars can always be used as illustrated in Example 7:

Proposition 3.4. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, let $\mathcal{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$ be the annotated grammar over $(\mathcal{C}, \mathfrak{G})$, let $m\pi n$ be a path in $\mathfrak{G}$, and let $\mathrm{A} \in \mathcal{N}$ be a non-terminal. We have* $\mathrm{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \mathrm{A})$ *if and only if we can derive π from $\mathrm{A}|_{mn} \in \mathcal{N}|_{\mathfrak{G}}$.*

Annotated grammars can, on their own, be used in *interactive* data exploration tools in which users can explorer the query results by zooming in on certain paths in the dataset, e.g., for graph analysis and query debugging.

3.2. Deriving shortest strings of a grammar

Let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, and $\mathrm{A}|_{mn} \in \mathcal{N}|_{\mathfrak{G}}$ be a graph-annotated non-terminal. As we argued in Section 3.1, $\mathcal{C}|_{\mathfrak{G}}$ is itself a grammar and any derivation $\mathrm{A}|_{mn} \to^*_{\mathcal{P}|_{\mathfrak{G}}} s$ of a string s over Σ carries enough information to derive the underlying path $m\pi n$ in $\mathfrak{G}$ with $\mathrm{trace}(\pi) = s$. Hence, if we have an efficient algorithm to derive shortest strings in a grammar, then we can apply that algorithm to the grammar $\mathcal{N}|_{\mathfrak{G}}$ to answer context-free path queries using the single-path semantics of Definition 3.1.

Next, we propose an efficient way to compute a shortest string in the language defined by a grammar. Mclean et al. [30] already proved that a shortest string can be computed given a grammar. In their proofs, Mclean et al. [30] used a rather brute-force algorithm for computing these shortest strings without

indications on whether the algorithm could be implemented efficiently, however. To make the step toward an efficient algorithm to compute shortest strings, we first introduce rewrites using *simple* production rules:

Definition 3.5. Let $\mathcal{P}$ be a set of production rules. We define $\text{heads}(\mathcal{P}) = \{\text{A} \mid (\text{A} \mapsto s) \in \mathcal{P}\}$ and we define the set of non-terminals derivable from A using the production rules in $\mathcal{P}$ by $\langle \text{A} \rangle_{\mathcal{P}} = \{\text{B} \in \mathcal{N} \mid \exists s_1 \exists s_2 \ \text{A} \to_{\mathcal{P}}^{+} s_1 \circ \text{B} \circ s_2\}$.

A set of production rules $\mathcal{P}$ is *non-recursive* if, for every $\text{A} \in \text{heads}(\mathcal{P})$, we have $\text{A} \notin \langle \text{A} \rangle_{\mathcal{P}}$. A set of production rules $\mathcal{P}$ is *deterministic* if, for every $\text{A} \in \text{heads}(\mathcal{P})$, there exists exactly one production rule $(\text{A} \mapsto s) \in \mathcal{P}$. Finally, a set of production rules $\mathcal{P}$ is *effective* if $\text{A} \in \text{heads}(\mathcal{P})$ implies that there exists a string $s \in \Sigma^*$ such that $\text{A} \to_{\mathcal{P}}^{*} s$. We refer to a set of production rules that is non-recursive, deterministic, and effective as *simple*.

A deterministic set of production rules $\mathcal{P}$ over alphabet Σ can only rewrite every non-terminal $\text{A} \in \text{heads}(\mathcal{P})$ into *exactly* one final string $\text{ustring}_{\mathcal{P}}(\text{A})$ over $(\mathcal{N} \cup \Sigma)$ (after exhausting all rewrite options). It is straightforward to prove that $\text{ustring}_{\mathcal{P}}(\text{A})$ exists and is unique: the rules $\mathcal{P}$ do not provide any choice in how any non-terminal encountered during rewriting is rewritten. If, in addition, $\mathcal{P}$ is a simple set of production rules, then it is guaranteed that the unique string $\text{ustring}_{\mathcal{P}}(\text{A})$ is a string over Σ.

Example 8. Consider Example 7. For brevity, we restrict ourselves to Alice, Carol, Dan, and Eve. With respect to these four people, the following set of production rules in the annotated grammar is deterministic non-recursive:

$$\text{Q}|_{\texttt{AC}} \mapsto \mathsf{friendOf}, \quad \text{Q}|_{\texttt{CA}} \mapsto \mathsf{friendOf}, \quad \text{Q}|_{\texttt{CD}} \mapsto \mathsf{friendOf}, \quad \text{Q}|_{\texttt{DE}} \mapsto \mathsf{friendOf},$$
$$\text{Q}|_{\texttt{AD}} \mapsto \text{Q}|_{\texttt{AC}} \ \text{Q}|_{\texttt{CD}}, \quad \text{Q}|_{\texttt{AE}} \mapsto \text{Q}|_{\texttt{AD}} \ \text{Q}|_{\texttt{DE}}, \quad \text{Q}|_{\texttt{CE}} \mapsto \text{Q}|_{\texttt{CA}} \ \text{Q}|_{\texttt{AE}}.$$

As the above rules are deterministic non-recursive, each non-terminal can be rewritten into one unique string. For example,

$$\text{Q}|_{\texttt{AD}} \to^{*}_{\text{Q}|_{\texttt{AD}} \mapsto \text{Q}|_{\texttt{AC}} \ \text{Q}|_{\texttt{CD}}} \text{Q}|_{\texttt{AC}} \ \text{Q}|_{\texttt{CD}} \to^{*}_{\text{Q}|_{\texttt{AC}} \mapsto \mathsf{friendOf}, \text{Q}|_{\texttt{CD}} \mapsto \mathsf{friendOf}} \mathsf{friendOf} \circ \mathsf{friendOf}.$$

As simple production rules define a unique way to derive specific strings, we can use simple production rules derived from a grammar to represent the derivation of the shortest strings in that grammar:

Lemma 3.6. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar. There exists a simple set of production rules $\mathcal{P}' \subseteq \mathcal{P}$ such that, for every non-terminal $\text{A} \in \mathcal{N}$ with $\mathcal{L}(\mathcal{C}; \text{A}) \neq \emptyset$, $\text{ustring}_{\mathcal{P}'}(\text{A})$ is a shortest string in $\mathcal{L}(\mathcal{C}; \text{A})$.*

We say that a set of production rules that satisfies the conditions of Lemma 3.6 is *minimizing*. Not every simple set of production rules is minimizing, however:

Example 9. Consider Example 8. The provided simple set of production rules $\mathcal{P}'$ is not minimizing: we have $|\text{ustring}_{\mathcal{P}'}(\text{Q}|_{\texttt{CE}})| = 4$, while a shorter string of length two exists. By replacing the production rule for $\text{Q}|_{\texttt{CE}}$ in $\mathcal{P}'$ by $\text{Q}|_{\texttt{CE}} \mapsto \text{Q}|_{\texttt{CD}} \ \text{Q}|_{\texttt{DE}}$, we obtain a minimizing set of production rules.

Using a minimizing set of production rules, it is straightforward to produce shortest strings for $\textsc{a} \in \text{heads}(\mathcal{P})$. Moreover, the way to obtain these shortest strings, by rewriting A, also provides complete information on how these shortest strings can be obtained from the original grammar. Next, we propose the MinimizeSet algorithm to construct a minimizing set of production rules. The pseudo-code of this algorithm can be found in Figure 3, *left*.

The MinimizeSet algorithm works rather intuitively. Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar. Production rules of the form $(\textsc{a} \to \sigma) \in \mathcal{P}$, $\sigma \in \Sigma$, produce the shortest possible strings: if $(\textsc{a} \to \sigma) \in \mathcal{P}$, then σ is a shortest string in $\mathcal{L}(\mathcal{C}; \textsc{a})$. If such productions rules exist for A, then we choose one of them for the minimizing set of production rules (Line 3). Next, we process non-terminals A for which we have determined the length *cost*[A] of the shortest strings in $\mathcal{L}(\mathcal{C}; \textsc{a})$. We do so on increasing string length by using a min-priority queue *new* (Line 7). We process A by checking, for each production rule $(\textsc{c} \mapsto \textsc{a}\ \textsc{b}) \in \mathcal{P}$ or $(\textsc{c} \mapsto \textsc{b}\ \textsc{a}) \in \mathcal{P}$, whether using this production rule will allow us to rewrite C into a shorter string than the currently-found string with length *cost*[C] (Line 10 and Line 12). We do so by rewriting—in this production rule—A to a string of length *cost*[A] (Line 14). Next, we illustrate the working of MinimizeSet with a small example.

Example 10. The following grammar $\mathcal{C}$ describes elementary math expressions:

$$\begin{aligned} &\textsc{ts} \mapsto \textsc{n}\ \textsc{at}; && \textsc{ts} \mapsto \textsc{ts}\ \textsc{at}; \\ &\textsc{n} \mapsto \sigma,\ \sigma \in \{\text{`0'}, \ldots, \text{`9'}\}; && \textsc{n} \mapsto \textsc{n}\ \textsc{n}; \\ &\textsc{op} \mapsto \sigma,\ \sigma \in \{\text{`+'}, \text{`-'}\}; && \textsc{at} \mapsto \textsc{op}\ \textsc{n}. \end{aligned}$$

Now consider the execution of MinimizeSet($\mathcal{C}$). After completing the loop at Line 3, we have

$$\begin{aligned} \mathcal{P}' &= \{\textsc{n} \mapsto (\textsc{n} \mapsto \text{“0”}), \textsc{op} \mapsto (\textsc{n} \mapsto \text{“+”})\}; \\ \mathit{cost} &= \{\textsc{n} \mapsto 1, \textsc{op} \mapsto 1\}; \\ \mathit{new} &= \{\textsc{n}, \textsc{op}\}. \end{aligned}$$

Next, the algorithm starts the loop at Line 7 by removing N from *new*. During the loops at Line 10 and 12, the algorithm can consider three production rules that involve N:

1. The rule $\textsc{ts} \mapsto \textsc{n}\ \textsc{at}$ cannot yet be considered, as $\textsc{at} \notin \mathit{cost}$.
2. The rule $\textsc{n} \mapsto \textsc{n}\ \textsc{n}$ produces a string “00” of length two for non-terminal N. For N, we already have a string of length one and, hence, the procedure produce does nothing.
3. The rule $\textsc{at} \mapsto \textsc{op}\ \textsc{n}$ produces a string “+0” for non-terminal AT. For non-terminal AT, we do not yet have a string. Hence, the if-case of the procedure produce at Line 15 adds $\textsc{at} \mapsto (\textsc{at} \mapsto \textsc{op}\ \textsc{n})$ to $\mathcal{P}'$, sets $\mathit{cost}[\textsc{at}] = 2$, and adds AT to *new*.

Next, the algorithm starts the second round of the loop at Line 7 by removing OP from *new*. This round, the algorithm can only consider the production rule

Algorithm MinimizeSet($\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$)**:**

1: $\mathcal{P}'$, $cost$:= empty mapping, empty mapping.
2: new is a min-priority queue.
3: **for all** $(\text{A} \mapsto \sigma) \in \mathcal{P}$ **do**
4: **if** $\text{A} \notin cost$ **then**
5: $cost[\text{A}], \mathcal{P}'[\text{A}] := 1, (\text{A} \mapsto \sigma)$.
6: add A to new with priority 1.
7: **while** $new \neq \emptyset$ **do**
8: Take A with minimum priority in new.
9: Remove A from new.
10: **for all** $(\text{C} \mapsto \text{A B}) \in \mathcal{P}$ with $\text{B} \in cost$ **do**
11: produce($\text{C} \mapsto \text{A B}$).
12: **for all** $(\text{C} \mapsto \text{B A}) \in \mathcal{P}$ with $\text{B} \in cost$ **do**
13: produce($\text{C} \mapsto \text{B A}$).
14: **return** $\{\mathcal{P}'[\text{A}] \mid \text{A} \in \mathcal{P}'\}$.

Procedure produce($\text{D} \mapsto \text{E F}$)**:**

15: **if** $\text{D} \notin cost$ **then**
16: $cost[\text{D}] := cost[\text{E}] + cost[\text{F}]$.
17: $\mathcal{P}'[\text{D}] := \text{D} \mapsto \text{E F}$.
18: Add D to new with priority $cost[\text{E}] + cost[\text{F}]$.
19: **else if** $cost[\text{D}] > cost[\text{E}] + cost[\text{F}]$ **then**
20: $cost[\text{D}] := cost[\text{E}] + cost[\text{F}]$.
21: $\mathcal{P}'[\text{D}] := \text{D} \mapsto \text{E F}$.
22: Lower priority of $\text{D} \in new$ to $cost[\text{E}] + cost[\text{F}]$.

Algorithm MinimizeSetGG($\mathcal{C}$, $\mathfrak{G}$)**:**

1: $\mathcal{P}'$, $cost$:= empty mapping, empty mapping.
2: new is a min-priority queue.
3: **for all** $(\text{A} \mapsto \sigma) \in \mathcal{P}$ and $(m, \sigma, n) \in \delta$ **do**
4: **if** $\text{A}|_{mn} \notin cost$ **then**
5: $cost[\text{A}|_{mn}], \mathcal{P}'[\text{A}|_{mn}] := 1, (\text{A}|_{mn} \mapsto \sigma)$.
6: Add $\text{A}|_{mn}$ to new with priority 1.
7: **while** $new \neq \emptyset$ **do**
8: Take $\text{A}|_{mn}$ with minimum priority in new.
9: Remove $\text{A}|_{mn}$ from new.
10: **for all** $(\text{C} \mapsto \text{A B}) \in \mathcal{P}$ with $\text{B}|_{no} \in cost$ **do**
11: produceGG($\text{C}|_{mo} \mapsto \text{A}|_{mn}\ \text{B}|_{no}$).
12: **for all** $(\text{C} \mapsto \text{B A}) \in \mathcal{P}$ with $\text{B}|_{om} \in cost$ **do**
13: produceGG($\text{C}|_{on} \mapsto \text{B}|_{om}\ \text{A}|_{mn}$).
14: **return** $\{\mathcal{P}'[\text{A}|_{mn}] \mid \text{A}|_{mn} \in \mathcal{P}'\}$.

Procedure produceGG($\text{D}|_{uw} \mapsto \text{E}|_{uv}\ \text{F}|_{vw}$)**:**

15: **if** $\text{D}|_{uw} \notin cost$ **then**
16: $cost[\text{D}|_{uw}] := cost[\text{E}|_{uv}] + cost[\text{F}|_{vw}]$.
17: $\mathcal{P}'[\text{D}|_{uw}] := \text{D}|_{uw} \mapsto \text{E}|_{uv}\ \text{F}|_{vw}$.
18: Add $\text{D}|_{uw}$ to new with priority $cost[\text{E}|_{uv}] + cost[\text{F}|_{vw}]$.
19: **else if** $cost[\text{D}|_{uw}] > cost[\text{E}|_{uv}] + cost[\text{F}|_{vw}]$ **then**
20: $cost[\text{D}|_{uw}] := cost[\text{E}|_{uv}] + cost[\text{F}|_{vw}]$.
21: $\mathcal{P}'[\text{D}|_{uw}] := \text{D}|_{uw} \mapsto \text{E}|_{uv}\ \text{F}|_{vw}$
22: Lower priority of $\text{D}|_{uw} \in new$ to $cost[\text{E}|_{uv}] + cost[\text{F}|_{vw}]$.

Figure 3: On the *left*, the MinimizeSet algorithm that constructs a minimizing set of production rules for the grammar $\mathcal{C}$. On the *right*, the MinimizeSetGG algorithm that constructs a minimizing set of production rules for the annotated grammar $\mathcal{C}|_{\mathfrak{G}}$, of which only the necessary parts are implicitly constructed.

AT $\mapsto$ OP N that involves OP. This production rule produces the string "+0" of length two for non-terminal AT. For AT, we already derived this string of length two and, hence, the procedure PRODUCE does nothing.

Next, the algorithm starts the third round of the loop at Line 7 by removing AT from new. This round, the algorithm can consider two production rules involving AT.

1. The rule TS $\mapsto$ N AT produces a string "0+0" for non-terminal TS. For non-terminal TS, we do not yet have a string. Hence, the if-case of the procedure PRODUCE at Line 15 adds TS $\mapsto$ (TS $\mapsto$ N AT) to $\mathcal{P}'$, sets $cost[\text{TS}] = 3$, and adds TS to new.
2. The rule TS $\mapsto$ TS AT produces a string "0+0+0" of length five for non-terminal TS. For TS, we just produced a string of length three and, hence, the procedure PRODUCE does nothing.

In the fourth and final round of the loop at Line 7, the algorithm removes TS from new. This round, the algorithm can only consider the rule TS $\mapsto$ TS AT. As we have seen in the third round, this rule produces a string "0+0+0" of length five for non-terminal TS and, hence, the procedure PRODUCE does nothing.

Theorem 3.7. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar. Execution of* MinimizeSet*($\mathcal{C}$) yields a minimizing set of production rules $\mathcal{P}'$ for $\mathcal{C}$ in $\mathcal{O}(|\mathcal{N}|(|\mathcal{N}| \log |\mathcal{N}| + |\mathcal{P}|))$. Using $\mathcal{P}'$, a set R of shortest strings s_{A} in $\mathcal{L}(\mathcal{C}; \text{A})$, $\text{A} \in \mathcal{N}$, can be constructed in $\mathcal{O}(L)$, in which $L = \sum\{|s_{\text{A}}| \mid s_{\text{A}} \in R\}$ is the total length of these shortest strings.*

Proof (sketch). In the following, we write slength(A) = $|s_{\mathrm{A}}|$, $\mathrm{A} \in \mathcal{N}$, to denote the length of any shortest string for non-terminal $\mathrm{A} \in \mathcal{N}$. The main *while*-loop maintains the following invariants:

1. The set $\{\mathcal{P}'[\mathrm{A}] \mid \mathrm{A} \in \mathcal{P}'\}$ is simple.
2. If $\mathrm{A} \in \mathcal{P}'$ and $\mathcal{P}'[\mathrm{A}] = (\mathrm{A} \mapsto \mathrm{B}\ \mathrm{C})$, then $cost[\mathrm{A}] \geq cost[\mathrm{B}] + cost[\mathrm{C}]$.
3. If $\mathrm{A} \in \mathcal{P}'$, then slength($\mathrm{A}$) $\leq cost[\mathrm{A}]$.
4. Let μ be the priority of the last element removed from *new*. No new element is inserted in *new* with priority less than or equal to μ.
5. Let μ be the priority of the last element removed from *new*. For every $\mathrm{A} \in \mathcal{N}$ with slength(A) $\leq \mu$, we have slength(A) = $cost[\mathrm{A}]$.
6. For every $\mathrm{A} \in \mathcal{N}$, we have either
   (i) A cannot be rewritten into a string over Σ; or
   (ii) slength(A) = $cost[\mathrm{A}]$; or
   (iii) slength(A) < $cost[\mathrm{A}]$ and A can be rewritten into a string $\mathrm{A}_1 \circ \ldots \circ \mathrm{A}_i$ of non-terminals such that $\mathrm{A}_1 \circ \ldots \circ \mathrm{A}_i$ can be rewritten into a shortest string for A; slength(A_j) = $cost[\mathrm{A}_j]$ for all $j, 1 \leq j \leq i$, and at-least one of $\mathrm{A}_1, \ldots, \mathrm{A}_i$ is in *new*.

As each non-terminal is added to *new* at most once, the MinimizeSet algorithm terminates. At termination, Invariants 1–6 guarantee that the resulting set of production rules is minimizing.

To obtain the stated complexity, we represent *costs* as an array holding $|\mathcal{N}|$ integers. The costs used in *cost* and *new* are integers in the range $1, \ldots, 2^{|\mathcal{N}|-1}$, which we can represent using $\log(2^{|\mathcal{N}|}) = |\mathcal{N}|$ bits. The initialization steps perform $\mathcal{O}(|\mathcal{P}|)$ steps. The *while*-loop will, in the worst case, visit every non-terminal once. For each of these non-terminals, one insertion into and one removal from the priority queue *new* is performed. The inner *for*-loops will visit every production rule twice, causing at most $2|\mathcal{P}|$ decrease key operations on priority queue *new*. When using a Fibonacci heap for a priority queue holding at most e elements, each insert and removal costs $\mathcal{O}(\log e)$ and each decrease key operation costs an amortized $\mathcal{O}(1)$ heap operations [31]. Hence, a total of $\mathcal{O}(|\mathcal{N}| \log |\mathcal{N}| + |\mathcal{P}|)$ heap operations are performed. Taking the size of the integers representing priorities into account, the heap operations cost $\mathcal{O}(|\mathcal{N}|(|\mathcal{N}| \log |\mathcal{N}| + |\mathcal{P}|))$. □

3.3. Deriving shortest paths for path query results

Using Proposition 3.4 and Theorem 3.7, we can already answer context-free path queries under the single-path semantics, this by applying MinimizeSet on a graph-annotated grammar.

Corollary 3.8. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, and $\mathrm{A} \in \mathcal{N}$ a context-free path query. We can evaluate* single($\mathrm{A}|_{\mathfrak{G}}$) *using* MinimizeSet *in $\mathcal{O}(|\mathcal{N}||\mathcal{V}|^2(|\mathcal{N}||\mathcal{V}|^2 \log(|\mathcal{N}||\mathcal{V}|^2) + |\mathcal{P}|(|\mathcal{V}|^3 + |\delta|)) + L)$ in which $L = \sum\{|\pi| \mid \pi \in \mathrm{single}(q|_{\mathfrak{G}})\}$ is total length of the shortest paths in the result.*

Unfortunately, using MinimizeSet has high overheads due to the explicit construction and storing of the graph-annotated grammar. Luckily, during the execution of MinimizeSet, the relevant parts of $\mathcal{C}|_{\mathfrak{G}}$ can be implicitly derived from $\mathcal{C}$ and $\mathfrak{G}$. We obtain the MinimizeSetGG algorithm by integrating these implicit derivation steps into MinimizeSet. The resulting pseudo-code can be found in Figure 3, *right*. We conclude:

Theorem 3.9. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, and $\mathrm{A} \in \mathcal{N}$ a context-free path query. We can evaluate* $\mathrm{single}(\mathrm{A}|_{\mathfrak{G}})$ *using* MinimizeSetGG *in $\mathcal{O}(|\mathcal{N}||\mathcal{V}|^2(|\mathcal{N}||\mathcal{V}|^2 \log(|\mathcal{N}||\mathcal{V}|^2) + |\mathcal{P}|(|\mathcal{V}|^3 + |\delta|)) + L)$, in which $L = \sum\{|\pi| \mid \pi \in \mathrm{single}(q|_{\mathfrak{G}})\}$ is total length of the shortest paths in the result.*

Proof (sketch). In the following, we write $\mathrm{plength}(\mathrm{A}|_{mn}) = |m\pi n|$ for non-terminal $\mathrm{A} \in \mathcal{N}$ and nodes $m, n \in \mathcal{V}$ to denote the length of any shortest path $m\pi n$ with $\mathrm{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \mathrm{A})$. The main *while*-loop maintains the following invariants:

1. The set $\{\mathcal{P}'[\mathrm{A}|_{mn}] \mid \mathrm{A}|_{mn} \in \mathcal{P}'\}$ is simple.
2. If $\mathrm{A}|_{mn} \in \mathcal{P}'$ and $\mathcal{P}'[\mathrm{A}|_{mn}] = (\mathrm{A}|_{mn} \mapsto \mathrm{B}|_{mo}\ \mathrm{C}|_{on}$, then $\mathit{cost}[\mathrm{A}|_{mn}] \geq \mathit{cost}[\mathrm{B}|_{mo}] + \mathit{cost}[\mathrm{C}|_{on}]$.
3. If $\mathrm{A}|_{mn} \in \mathcal{P}'$, then $\mathrm{plength}(\mathrm{A}|_{mn}) \leq \mathit{cost}[\mathrm{A}|_{mn}]$.
4. Let μ be the priority of the last element removed from *new*. No new element is inserted in *new* with priority less than or equal to μ.
5. Let μ be the priority of the last element removed from *new*. For every $\mathrm{A}|_{mn} \in \mathcal{N}|_{\mathfrak{G}}$ with $\mathrm{plength}(\mathrm{A}|_{mn}) \leq \mu$, we have $\mathrm{plength}(\mathrm{A}|_{mm}) = \mathit{cost}[\mathrm{A}|_{mn}]$.
6. For every $\mathrm{A} \in \mathcal{N}$ and every $m, n \in \mathcal{V}$, we have either
   (i) $a|_{mn} \notin \mathcal{N}|_{\mathfrak{G}}$; or
   (ii) $\mathrm{plength}(\mathrm{A}|_{mn}) = \mathit{cost}[\mathrm{A}|_{mn}]$; or
   (iii) $\mathrm{plength}(\mathrm{A}|_{mn}) < \mathit{cost}[\mathrm{A}|_{mn}]$ and $a|_{mn}$ can be rewritten using the production rules $\mathcal{P}|_{\mathfrak{G}}$ of the graph-annotated grammar $\mathcal{C}|_{\mathfrak{G}}$ into a string $\mathrm{A}_1|_{m_1n_1} \circ \ldots \circ \mathrm{A}_i|_{m_in_i}$ of non-terminals such that $\mathrm{A}_1|_{m_1n_1} \circ \ldots \circ \mathrm{A}_i|_{m_in_i}$ can itself be rewritten using the production rules $\mathcal{P}|_{\mathfrak{G}}$ into a shortest path for $\mathrm{A}|_{mn}$; $\mathrm{plength}(\mathrm{A}_j|_{m_jn_j}) = \mathit{cost}[\mathrm{A}_j|_{m_jn_j}]$ for all $j, 1 \leq j \leq n$, and at-least one of $\mathrm{A}_1|_{m_1n_1}, \ldots, \mathrm{A}_j|_{m_jn_j}$ is in *new*.

As each node-annotated non-terminal is added to *new* at most once, the MinimizeSetGG algorithm terminates. At termination, Invariants 1–6 guarantee that the resulting set of production rules is minimizing. To obtain the stated complexity, we refer to the proof of Theorem 3.7. □

4. On the worst-case length of the shortest paths

Let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph with $m, n \in \mathcal{V}$, let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar with $\mathrm{A} \in \mathcal{N}$, and consider the path query specified by the language $\mathcal{L}(\mathcal{C}; \mathrm{A})$. As outlined in Section 3.1, the length of any shortest path $m\pi n$ in graph $\mathfrak{G}$ with $\mathrm{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \mathrm{A})$ is equivalent to the length of the string $\mathrm{trace}(\pi)$ and

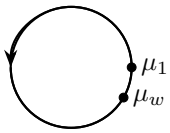

Figure 4: A directed cyclic graph with $|\mathcal{V}| = w$ nodes. In this graph, the shortest path from μ_1 to μ_w visits every other node in the graph and, hence, has length $w - 1$.

trace(π) must be a shortest string in the language $\mathcal{L}(\mathcal{C}|_{\mathfrak{G}}; \mathrm{A}|_{mn})$. In Section 3.3, we introduced the MinimizeSetGG algorithm to efficiently compute the string trace(π) and path π. Next, we will study worst-case bounds on the length of π, thereby bounding the complexity of writing out π after running MinimizeSetGG. By definition, the language $\mathcal{L}(\mathcal{C}|_{\mathfrak{G}}; \mathrm{A}|_{mn})$ is equivalent to the language $\mathcal{L}(\mathcal{C}; \mathrm{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$. Hence, determining worst-case bounds on the length of π is equivalent to the following formal language problem:

Problem 1. What is the worst-case length of the shortest string in the language $\mathcal{L}(\mathcal{C}; \mathrm{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$?

Next, we proceed in four steps to gain more insight into this formal language problem and, hence, into worst-case bounds on the length of shortest paths. First, we look at the worst-case length of shortest strings in languages with *regular representations* and *context-free representations*. Second, we look at the worst-case length of shortest paths for path queries that can be represented using a *finite automaton*. Third, we look at the worst-case length of shortest paths for queries represented by grammars over a *singleton alphabet*. Fourth and finally, we look at the worst-case length of shortest paths for queries represented by all other *grammars*.

4.1. The worst-case length of shortest strings

For normal regular languages represented by finite automata and for context-free languages represented by grammars, the worst-case length of the shortest string in these languages is well-known [32]. First, the case for regular languages represented by finite automata:

Lemma 4.1. *Let* $(\mathfrak{G} = (\mathcal{V}, \Sigma, \delta), m, n)$ *be a finite automaton and let* $s \in \mathcal{L}(\mathfrak{G}; m, n)$ *be a shortest string. We have* $|s| \leq |\mathcal{V}| - 1$.

Proof (sketch). The longest string must be the trace of a path $m\pi n$ in $\mathfrak{G}$. As this path has minimal length, we know it does not have cycles. Hence, to make this path as long as possible, it has to include every node exactly once. To do so, we construct a *cyclic* σ-labeled graph $\mathfrak{G}$. Let $\mathcal{V} = \{\mu_1, \ldots, \mu_{|\mathcal{V}|}\}$ and $\delta = \{(\mu_i, \sigma, \mu_{(i+1) \bmod |\mathcal{V}|}) \mid 1 \leq i \leq |\mathcal{V}|\}$. We have sketched this graph in Figure 4. We choose $m = \mu_1$ and $n = \mu_{|\mathcal{V}|}$ and the shortest string in $\mathcal{L}(\mathfrak{G}; m, n)$ must have length $|s| = |\mathcal{V}| - 1$. □

Next, the case for context-free languages represented by grammars:

Lemma 4.2. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar with $\mathrm{A} \in \mathcal{N}$ and let $s \in \mathcal{L}(\mathcal{C}; \mathrm{A})$ be a shortest string. We have $|s| \leq 2^{|\mathcal{N}|-1}$. In the worst case, we have $|s| = 2^{|\mathcal{N}|-1}$*

Proof (sketch). Let $\mathcal{C} = (\mathcal{N}, \{\sigma\}, \mathcal{P})$ be the grammar with $\mathcal{N} = \{\mathrm{A}_1, \ldots, \mathrm{A}_{|\mathcal{N}|}\}$ and $\mathcal{P} = \{\mathrm{A}_1 \mapsto \sigma\} \cup \{\mathrm{A}_j \mapsto \mathrm{A}_{j-1}\ \mathrm{A}_{j-1} \mid 1 < j \leq |\mathcal{N}|\}$. Using these production rules, each A_j, $1 \leq j \leq |\mathcal{N}|$, is rewritten into a sequence of 2^{j-1} σ symbols. Hence, the shortest and only string in $\mathcal{L}(\mathcal{C}; \mathrm{A}_j)$ has length 2^{j-1}. By choosing $\mathrm{A} = \mathrm{A}_{|\mathcal{N}|}$, we obtain $|s| = 2^{|\mathcal{N}|-1}$. Finally, using an inductive argument on the number of non-terminals $|\mathcal{N}|$, one can show that any derivable string either has length at-most $2^{|\mathcal{N}|-1}$ or, if it is longer, is not a shortest string. □

4.2. Shortest paths and regular path queries

Next, we take a look at simple context-free grammars: those grammars that describe a *regular* language (and, hence, could have been represented by a finite automaton). We have

Proposition 4.3. *Let $\mathcal{L}$ be a regular language, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, let $m, n \in \mathcal{V}$, and let $m\pi n$ be the shortest path from m to n in $\mathfrak{G}$ with $\mathrm{trace}(\pi) \in \mathcal{L}$. There exists a constant c, depending only on $\mathcal{L}$, such that $|\pi| \leq c|\mathcal{V}|$.*

Proof. As $\mathcal{L}$ is a regular language, it can be represented via a finite automaton in which all transitions use symbols from Σ [22]. Consider such an automaton with c states. We can construct the language $\mathcal{L} \cap \mathcal{L}(\mathfrak{G}; m, n)$ using the well-known product construction for the intersection of two finite automata [22]. The resulting automaton has at-most $c|\mathcal{V}|$ states. Using Lemma 4.1, we conclude that the shortest path between any two states in the resulting intersection automaton that represent nodes m and n in $\mathfrak{G}$ will have worst-case length $c|\mathcal{V}|$, proving the upper bound. □

4.3. Shortest paths and queries that use a single edge label

Next, we consider context-free grammars over a singleton alphabet ($|\Sigma| = 1$), which represent context-free graph queries that use a single edge label. In this case, we can use the result that all context-free grammars over a singleton alphabet represent regular languages [33]:

Theorem 4.4. *Let $\mathcal{C} = (\mathcal{N}, \Sigma', \mathcal{P})$ be a grammar with $|\Sigma'| = 1$, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph with $\Sigma' \subseteq \Sigma$, let $\mathrm{A} \in \mathcal{N}$, let $m, n \in \mathcal{V}$, and let $m\pi n$ be the shortest path from m to n in $\mathfrak{G}$ with $\mathrm{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \mathrm{A})$. In the worst-case, there exists a constant c, $2^{|\mathcal{N}|-1} \leq c \leq 2^{2|\mathcal{N}|-1}$, such that $|\pi| \leq c|\mathcal{V}|$.*

Proof. First, we prove the lower bound. Without loss of generality, we assume $\Sigma' = \{\sigma\}$. Now let $\mathcal{C}' = (\mathcal{N}, \{\sigma\}, \mathcal{P}')$ be the grammar that produces strings $s = \sigma \ldots \sigma$ of length $|s| = k2^{|\mathcal{N}|-1}, k \geq 1$. We obtain this grammar $\mathcal{C}'$ by adding the production rule $\mathrm{A}_{|\mathcal{N}|} \mapsto \mathrm{A}_{|\mathcal{N}|}\ \mathrm{A}_{|\mathcal{N}|}$ to the grammar $\mathcal{C}$ used in the proof of Lemma 4.2. Note that $\mathcal{L}(\mathcal{C}; \mathrm{A}) = \{\sigma^{2^{|\mathcal{N}|-1}}\}$. By construction, we have

$$\mathcal{L}(\mathcal{C}'; \mathrm{A}_{|\mathcal{N}|}) = \{s \in \Sigma'^* \mid |s| = k2^{|\mathcal{N}|-1} \wedge k \geq 1\}.$$

Let $\mathfrak{G} = (\mathcal{V}, \{\sigma\}, \delta)$ be a σ-labeled cycle as used in the proof of Lemma 4.1. We have

$$\mathcal{L}(\mathfrak{G}; n_1, n_1) = \{s \in {\Sigma'}^* \mid |s| = k|\mathcal{V}| \wedge k \geq 0\}.$$

The shortest string $s \in (\mathcal{L}(\mathcal{C}'; \mathrm{A}_{|\mathcal{N}|}) \cap \mathcal{L}(\mathfrak{G}; n_1, n_1))$ must have length $|s| = \mathrm{lcm}(2^{|\mathcal{N}|-1}, |\mathcal{V}|)$. If $|\mathcal{V}|$ is odd, then $|\mathcal{V}|$ and $2^{|\mathcal{N}|-1}$ are coprime, implying that $|s| = 2^{|\mathcal{N}|-1}|\mathcal{V}|$, proving the lower bound on c.

The upper bound is proven using a result of Pighizzini et al. [33]: Pighizzini et al. show that, for each $\mathrm{A} \in \mathcal{N}$, there exists a finite automaton with language $\mathcal{L}(\mathcal{C}; \mathrm{A})$ and at most $2^{2|\mathcal{N}|-1} + 1$ states. Given such an automaton with at most $2^{2|\mathcal{N}|-1} + 1$ states, we apply the construction of the proof of Proposition 4.3, proving the upper bound on c. □

4.4. Shortest paths and context-free path queries

Finally, we consider context-free grammars over arbitrary alphabets, which represent context-free graph queries that use multiple edge labels. As we shall show, the worst-case bounds provided by Theorem 4.4 cannot be generalized to arbitrary alphabets, which we show next.

Theorem 4.5. *Let $\mathcal{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar with $|\Sigma| > 1$, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, let $\mathrm{A} \in \mathcal{N}$, let $m, n \in \mathcal{V}$, and let $m\pi n$ be the shortest path from m to n in $\mathfrak{G}$ with $\mathrm{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \mathrm{A})$. In the worst-case, we have $2^{|\mathcal{N}|}|\mathcal{V}|^2 \leq 32|\pi|$, even if $|\Sigma| = 2$ and $\mathcal{C}$ represents a deterministic context-free language.*[1]

Proof. Consider the language $\mathcal{L} = \{{\sigma_1}^k {\sigma_2}^k \mid k \geq 1\}$ over $\Sigma = \{\sigma_1, \sigma_2\}$. This language is deterministic context-free, but not regular [22]. Let $\mathcal{C} = (\mathcal{N}, \{\sigma_1, \sigma_2\}, \mathcal{P})$ be the grammar with

$$\begin{aligned}
\mathcal{N} &= \{\mathrm{A}_1, \ldots, \mathrm{A}_{|\mathcal{N}|-4}, \mathrm{B}, \mathrm{B}', \mathrm{B}_1, \mathrm{B}_2\};\\
\mathcal{P} &= \{\mathrm{A}_1 \mapsto \mathrm{B}\ \mathrm{B}\} \cup\\
&\quad \{\mathrm{A}_j \mapsto \mathrm{A}_{j-1}\ \mathrm{A}_{j-1} \mid 1 < j \leq |\mathcal{N}| - 4\} \cup\\
&\quad \{\mathrm{B} \mapsto \mathrm{B}_1\ \mathrm{B}_2, \mathrm{B} \mapsto \mathrm{B}_1\ \mathrm{B}', \mathrm{B}' \mapsto \mathrm{B}\ \mathrm{B}_2, \mathrm{B}_1 \mapsto \sigma_1, \mathrm{B}_2 \mapsto \sigma_2\}.
\end{aligned}$$

We have $\mathcal{L}(\mathcal{C}; \mathrm{B}) = \mathcal{L}$ and each A_j, $1 \leq j \leq |\mathcal{N}| - 4$, will be rewritten into a string of exactly 2^j B non-terminals. Hence, $\mathcal{L}(\mathcal{C}; \mathrm{A}_j) = \mathcal{L}^{2^j} = \{{\sigma_1}^k {\sigma_2}^k \mid k \geq 1\}^{2^j}$. Furthermore, $\mathcal{L}(\mathcal{C}; \mathrm{A}_j)$ is deterministic.

Let $\mathfrak{G} = (\mathcal{V}, \{\sigma_1, \sigma_2\}, \delta)$ be a double-cyclic graph that consists of two cycles that share a single common node c. The first cycle represents a single cyclic path $c\sigma_1 n_1 \sigma_1 \ldots \sigma_1 n_{u-1} \sigma_1 c$ of u edges labeled σ_1. The second cycle forms a cyclic path $c\sigma_2 m_1 \sigma_2 \ldots \sigma_2 m_{v-1} \sigma_2 c$ of v edges labeled σ_2. The resulting graph is visualized in Figure 5. Next, we formalize this double-cyclic graph. We choose

[1] Note that a *deterministic context-free language* is a language $\mathcal{L}$ for which a deterministic pushdown automaton exists that accepts $\mathcal{L}$ [22].

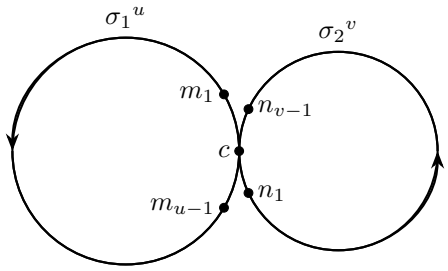

Figure 5: The double-cyclic graph consists of two cycles that share a single common node c. The cycle on the left forms a cyclic path $c\sigma_1 n_1 \sigma_1 \dots \sigma_1 n_{u-1} \sigma_1 c$ of u edges labeled σ_1. The cycle on the right forms a cyclic path $c\sigma_2 m_1 \sigma_2 \dots \sigma_2 m_{v-1} \sigma_2 c$ of v edges labeled σ_2.

k, $k \geq 1$, and $|\mathcal{V}| = u + v - 1$ with $u = 2^k + 1$ and $v = 2^k$. Hence,

$$\begin{aligned}
\mathcal{V} &= \{c, m_1, \dots, m_{u-1}, n_1, \dots n_{v-1}\};\\
\delta &= \{(c, \sigma_1, m_1), (m_{u-1}, \sigma_1, c)\} \cup\\
&\quad \{(c, \sigma_2, n_1), (n_{v-1}, \sigma_2, c)\} \cup\\
&\quad \{(m_i, \sigma_1, m_{i+1}) \mid 1 \leq i < u - 1\} \cup\\
&\quad \{(n_i, \sigma_2, n_{i+1}) \mid 1 \leq i < v - 1\}.
\end{aligned}$$

Note that the language $\mathcal{L}(\mathfrak{G}; c, c)$ is the regular language expressed by the regular expression $(\sigma_1{}^u + \sigma_2{}^v)^*$. Let $c\pi c$ be the shortest path in $\mathfrak{G}$ with $\text{trace}(\pi) \in \mathcal{L}(\mathcal{C}; \text{B})$. Due to the definition of B, we must have $\text{trace}(\pi) = s_1 \circ s_2$ with, for some $x \geq 1$, $s_1 = \sigma_1{}^x$ and $s_2 = \sigma_2{}^x$. As only node c has outgoing edges labeled with σ_1 and σ_2, s_1 is the trace of a path $c\pi_1 c$ in the σ_1-labeled cycle. Likewise, s_2 is the trace of a path $c\pi_2 c$ in the σ_2-labeled cycle. Consequently, we have $|\pi_1| = x = y_1 u$ and $|\pi_2| = x = y_2 v$, $y_1 \geq 1$ and $y_2 \geq 1$. As π is the shortest path, we must have $x = \text{lcm}(u, v)$. As $u = 2^k + 1$ and $v = 2^k$ are coprime, we have $\text{lcm}(u, v) = uv$. As $u = v + 1$ and $v = |\mathcal{V}|/2$, we conclude

$$|\pi| = 2x = 2uv = 2(v+1)v \geq 2v^2 = 2\left(\frac{|\mathcal{V}|}{2}\right)^2 = \frac{|\mathcal{V}|^2}{2}.$$

Let $c\pi_j c$, $1 \leq j \leq |\mathcal{N}| - 4$, be the shortest path in $\mathfrak{G}$ with $\text{trace}(\pi_j) \in \mathcal{L}(\mathcal{C}; \text{A}_j)$. Recall that every A_j will be rewritten in a string of exactly 2^j B non-terminals. As only node c has outgoing edges labeled with σ_1 and σ_2, each of these 2^j B non-terminals will represent the path $c\pi c$. Hence,

$$|\pi_j| = 2^j(2uv) \geq 2^j \left(\frac{|\mathcal{V}|^2}{2}\right).$$

To complete the proof, we only need to consider the longest of the paths π_j, the path $\pi_{|\mathcal{N}|-4}$. We have

$$|\pi_{|\mathcal{N}|-4}| \geq 2^{|\mathcal{N}|-4}\left(\frac{|\mathcal{V}|^2}{2}\right) = \frac{2^{|\mathcal{N}|}}{16}\frac{|\mathcal{V}|^2}{2} = \frac{2^{|\mathcal{N}|}|\mathcal{V}|^2}{32}. \qquad \square$$

The above theorem already holds for the class of non-regular deterministic context-free languages. More generally, the grammar $\mathcal{C}$ used in the proof of Theorem 4.5 can be expressed in all typical *simple* non-regular fragments of the context-free grammars (e.g., the simple grammars, LL(k) grammars, LR(k) grammars, and so on). Hence, the quadratic lower bound of Theorem 4.5, in terms of the number of nodes in the graph, applies to all these classes of *non-regular* context-free grammars.

Theorem 4.4 and Theorem 4.5 consider context-free path queries over alphabets with one or two symbols. These theorems show that moving from singleton alphabets to alphabets with two symbols starkly increases the worst-case lower bound (in terms of the number of nodes in the graph) on the length of shortest paths when evaluating such queries. Hence, it is only natural to ask whether moving to even larger alphabets further increases this worst-case lower bound. Next, we shall argue that this is not the case. Consider any non-singleton alphabet Σ'. Using a straightforward binary encoding, one can encode every symbol $\sigma \in \Sigma'$ by a unique string in Σ^* of length at-most $c = \lceil \log(|\Sigma|) \rceil$. For example, we can represent the symbols $\Sigma = \{\text{`a'}, \text{`1'}, \text{`}\pi\text{'}, \text{`}\circ\text{'}\}$ in the $|\Sigma| = 4$-symbol alphabet Σ by the strings over the two-symbol alphabet $\Sigma_2 = \{\text{`0'}, \text{`1'}\}$ via the mapping $\{\text{`a'} \mapsto \text{“00”}, \text{`1'} \mapsto \text{“01”}, \text{`}\pi\text{'} \mapsto \text{“10”}, \text{`}\circ\text{'} \mapsto \text{“11”}\}$. As regular languages and context-free languages are closed under homomorphisms [22], we can apply this binary encoding to graphs and grammars. This encoding will only increase the length of paths by a constant factor dependent only on c.

4.5. A distinction result

In Table 1, we have summarized our results. As one can see from the table, our results showcase a distinction: the shortest path is *linearly upper bounded* by the size of the graph when the query can be expressed as a regular language after restricting the query to only those symbols it has in common with the graph (e.g., when the query is a regular language or a context-free language with only one symbol in common with the graph); whereas the shortest path is *quadratically lower bounded* by the size of the graph for arbitrary context-free grammars that use several symbols ($|\Sigma| > 1$), even if the context-free grammar is deterministic and uses only two symbols. Furthermore, based on the results in this paper, this distinction can be extended to the intersection of multiple context-free languages. To show the distinction for the intersection of multiple context-free languages, we introduce the following technical results:

Lemma 4.6. *Let $\mathcal{C}_i = (\mathcal{N}_i, \Sigma_i, \mathcal{P}_i)$, $1 \leq i \leq n$, be n context-free grammars and consider the language $\mathcal{L} = \mathcal{L}(\mathcal{C}_1; \mathrm{A}_1) \cap \ldots \cap \mathcal{L}(\mathcal{C}_n; \mathrm{A}_n)$. If $|\Sigma_j| = 1$ for any $j, 1 \leq i \leq n$, then the worst-case upper bound on the length of the shortest strings in $\mathcal{L}$ is $2^{2(|\mathcal{N}_1|+\cdots+|\mathcal{N}_n|)}$.*

Proof. Without loss of generality, we can assume that $|\Sigma_1| = 1$. As we are only interested in the intersection $\mathcal{L}$, we can restrict all other grammars $\mathcal{C}_j$, $2 \leq j \leq n$, to the alphabet Σ_1 by removing production rules of the form $(\mathrm{A} \mapsto \sigma) \in \mathcal{P}_j$ with $\sigma \in (\Sigma_j \setminus \Sigma_1)$. We write ${\mathcal{C}_j}'$, $2 \leq j \leq n$, to denote this restriction of $\mathcal{C}_j$ to alphabet

Σ_1. By Pighizzini et al. [33] (using the same arguments as in Theorem 4.4), we conclude that the language $\mathcal{L}(\mathcal{C}_1; \mathrm{A}_1)$ can be expressed by a finite automaton with at-most $2^{2|\mathcal{N}_1|}$ states and that the language $\mathcal{L}(\mathcal{C}_j{}'; \mathrm{A}_j)$, $2 \leq j \leq n$, can be expressed by a finite automaton with at-most $2^{2|\mathcal{N}_j|}$ states. We can compute the intersection of these finite automata using the product construction, which will yield a finite automaton with at most $2^{2|\mathcal{N}_1|} \times \cdots \times 2^{2|\mathcal{N}_n|} = 2^{2(|\mathcal{N}_1|+\cdots+|\mathcal{N}_n|)}$ states. Hence, by Lemma 4.1, the length of the shortest string in $\mathcal{L}$ is upper-bounded by $2^{2(|\mathcal{N}_1|+\cdots+|\mathcal{N}_n|)}$. □

Due to Lemma 4.6, the intersection $\mathcal{L}$ of context-free languages of which at-least one language is defined over a singleton alphabet is itself a regular language and, hence, there exists a well-defined worst-case upper bound on the length of the shortest string in $\mathcal{L}$. This is not the case for the general intersection of context-free languages:

Lemma 4.7. *Let $\mathcal{L}$ be the intersection of two context-free languages. There is no worst-case upper bound on the length of the shortest string in $\mathcal{L}$.*

Proof. Let $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$ such that $\mathcal{L}_1$ and $\mathcal{L}_2$ are context-free languages. We prove this lemma by contradiction. Assume one can determine a worst-case upper bound c on the length of the shortest string in $\mathcal{L}$. Note that $\mathcal{L} \neq \emptyset$ if and only if there exists a shortest string $s \in \mathcal{L}$ with $|s| \leq c$. There are only a finite number of strings with length up-to c over the alphabets of the context-free grammars that define $\mathcal{L}_1$ and $\mathcal{L}_2$. Hence, we can enumerate all strings s' of length at-most c and test whether $s' \in \mathcal{L}_1$ and $s' \in \mathcal{L}_2$ to eventually find whether there is a shortest string $s \in \mathcal{L}$ of length at-most c and, hence, to determine whether $\mathcal{L} = \emptyset$. We conclude that if we can determine a worst-case upper bound c, then we can determine whether $\mathcal{L} = \emptyset$, which is a well-known undecidable problem [22], a contradiction. □

The distinction observed here is surprising: the worst-case length of a shortest string in just a regular language, just a context-free language, or the intersection of two regular languages (which is equivalent to the worst-case length of a shortest path resulting from a regular path query) are all *independent* of the size of the alphabet Σ. The size of the alphabet only plays a role when one looks at the worst-case length of the shortest strings of languages that are the intersection of a context-free language with either a regular or a context-free language.

5. Empirical Evaluation

To show that the path-based semantics for context-free path are viable in practice, we implemented the MinimizeSetGG algorithm of Figure 3, *right*, in C++20. Remember that the MinimizeSetGG algorithm produces a set of simple production rules for each graph-annotated non-terminal. Hence, given a graph-annotated non-terminal, there is a deterministic and unique derivation leading toward a single shortest path. To construct these shortest paths in our experiments, we also implemented this straightforward path construction

Language	$\lvert\Sigma\rvert = 1$	$\lvert\Sigma\rvert > 1$	Rationale
Regular	n		Lemma 4.1
Context-Free	$2^m - 1$		Lemma 4.2
Regular ∩ Regular	$n_1 n_2$		Proposition 4.3
Regular ∩ Det. Context-Free	$\mathcal{O}(n \cdot 2^{2m-1})$	$\Omega(n^2 2^m)$, $\mathcal{O}(2^{n^2 m})$	Theorems 4.4 and 4.5
Regular ∩ Context-Free	$\mathcal{O}(n \cdot 2^{2m-1})$	$\Omega(n^2 2^m)$, $\mathcal{O}(2^{n^2 m})$	Theorems 4.4 and 4.5
Context-Free ∩ Context-Free	$\mathcal{O}(2^{2(m_1+m_2)})$	unbounded	Lemmas 4.6 and 4.7

Table 1: The worst-case length of a shortest string in a language. For the complexity bounds in the second and third column, we represent *regular languages* by finite automata with n states and represent *context-free languages* by grammars with m non-terminals. In the case of two regular languages, their representations have n_1 and n_2 states, respectively. In the case of two context-free languages, their representations have m_1 and m_2 non-terminals, respectively. As a finite automaton with n states is a graph with n nodes, the results for intersections with regular languages translates directly to bounds on the length of shortest paths in a graph with n nodes.

algorithm in `C++20`. The complexity of this path construction algorithm to derive a single path π is $\Theta(|\pi|)$. Open-source code of the full `C++20` implementation of the data structures, algorithms, and supporting tooling used can be found at `https://www.jhellings.nl/projects/cfpqpaths/`. Using this implementation, we ran three different experiments to study the behavior of the MinimizeSetGG algorithm. The programs were compiled and run on a workstation with an Intel Core i7-8700 CPU, running at a maximum of 4.6 GHz, and with 16 GiB of main memory. In each of our experiments, we test with synthetic graphs that are designed specifically to test the worst-case behavior known for these algorithms, e.g., due to the size of generated paths (Theorem 4.5) or due to the size of the result sets. Next, we detail each of the experiments and their results.

5.1. Cost of the single-path semantics

As the first experiment, we study the cost of evaluating context-free path queries using the single-path semantics. To put MinimizeSetGG to the test, we run these experiments with the grammar

$$\mathrm{Q} \mapsto \mathrm{A}\ \mathrm{Q}', \qquad \mathrm{Q}' \mapsto \mathrm{Q}\ \mathrm{B}, \qquad \mathrm{Q} \mapsto \mathrm{A}\ \mathrm{B}, \qquad \mathrm{A} \mapsto \sigma_1, \qquad \mathrm{B} \mapsto \sigma_2,$$

which is context-free and cannot be expressed by a regular language and we use the double-cyclic graphs of Figure 5 with $u = v + 1$. As proven in Theorem 4.5, this combination of query and graphs produces very large paths that have a length that is *quadratic* in the size of the graph. Furthermore, this combination of graph and query are the worst known combination with respect to the lengths of the paths produced. The results of the experiment can be found in Figure 6.

As is clear from the results, single-path evaluation is practically feasible: the query Q, evaluated on a double-cyclic graph of 9750 nodes, yields a set of 47 550 751 distinct paths, of which the longest (non-simple) path has 47 541 001 edges, and the average path has 23 770 501 edges. Hence, the query result is

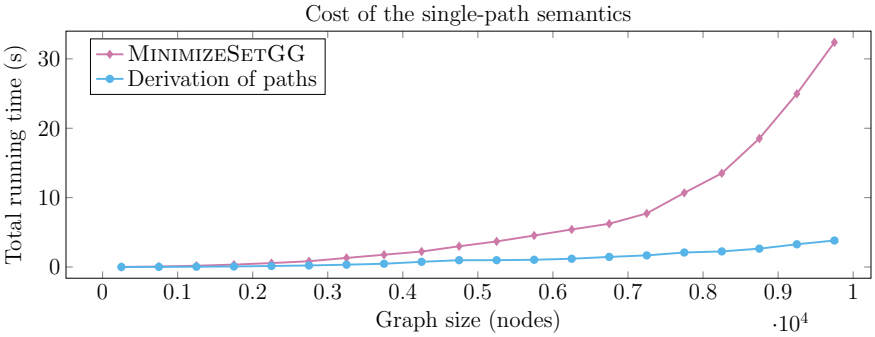

Figure 6: Measurements for the *cost of the single-path semantics* experiment. In this experiment, we measured the performance of MinimizeSetGG as a function of the graph size and we detail both the running time of MinimizeSetGG and of the subsequent path derivation algorithm. As the graph and query are based on the worst-case complexity results of Theorem 4.5, this experiment uses the most complex combination of graph and query known to us.

large. Still, MinimizeSetGG finished in only 32 s and the longest path was constructed in 3.8 s. Hence, even for queries and graphs that produce very large results, the query costs are reasonable.

5.2. Grammars: bounded vs. unbounded

In the second experiment, we take a more in-depth look of the cost of context-free path query evaluation. In practice, many path queries are *bounded* in the sense that only paths of a limited length are inspected in the graph. For example, to ask for friends-of-friends in a social network, one only has to inspect paths of length two. Some path queries, however, are *unbounded*, as context-free path queries can use recursion. This is of use, e.g., to query for pairs of indirect friends (Example 1). As unbounded queries can yield much larger result sets than bounded queries, we inspect the impact of the type of queries on the running time of MinimizeSetGG. For this experiment, we use the queries P_1 (bounded) and P_2 (unbounded):

$$\mathrm{P}_1 \mapsto \mathrm{S}\ \mathrm{B} \qquad \mathrm{B} \mapsto \mathrm{S}\ \mathrm{S} \qquad \mathrm{S} \mapsto \sigma;$$
$$\mathrm{P}_2 \mapsto \mathrm{S}\ \mathrm{P}_2 \qquad \mathrm{P}_2 \mapsto \sigma \qquad \mathrm{S} \mapsto \sigma.$$

The language described by P_1 is $\mathcal{L}_1 = \{\sigma\sigma\sigma\}$, and the language described by P_2 is $\mathcal{L}_2 = \{\sigma^k \mid k \geq 1\}$. We use the cycle graphs with w nodes of Figure 4, on which query P_1 will evaluate to a very sparse result set of w paths, whereas query P_2 will evaluate to a very dense result set of w^2 paths (which is the maximum number of paths in the result set). We measured the running time of MinimizeSetGG for both queries. The results of the experiment can be found in Figure 8.

As is clear from the results, the performance of single-path evaluation depends largely on the size of the query result. On the one hand, query evaluation for P_1, a bounded query yielding a small result set, finished within a second for

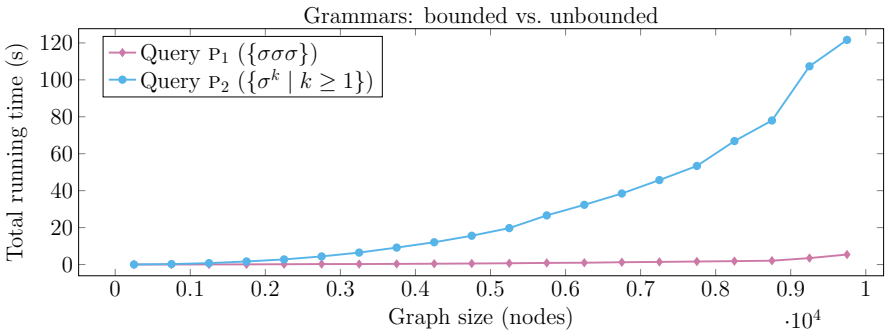

Figure 7: Measurements for the *grammars: bounded vs. unbounded* experiment. In this experiment, we measured the performance of MinimizeSetGG as a function of the graph size. Specifically, we compare the performance for evaluating a *bounded query* P_1 that only inspects paths of length up-to-three and an *unbounded query* P_2 that inspects paths of arbitrary lengths.

all graphs with up-to-6250 nodes and finished within six seconds on all graphs. On the other hand, query evaluation for P_2, an unbounded query yielding large result sets, produced a result set of 95 072 250 paths on the largest graph and did so in 121 s. We notice that the size of the result set is the limiting factor here: in the previous experiment, we already demonstrated that MinimizeSetGG can easily deal with very large paths constructed by complex context-free path queries.

5.3. Grammars: unambiguous vs. ambiguous

In the third and final experiment, we look at the impact of the design of context-free path queries on the cost of their evaluation. This experiment is inspired by well-known results from parsing and compiler construction (see, e.g., [21]): for grammars that are deterministic and unambiguous[2], e.g., LL(k) or LR(k) grammars, simple high-performance parsers with a linear running time exist. For non-deterministic and for ambiguous grammars, such high-performance parsers do not exist, however. The MinimizeSetGG algorithm we propose works on all grammars, even grammars that are non-deterministic and ambiguous. This raises the question whether the type of grammars impacts the overall performance. To answer this question, we construct two equivalent queries Q_1 (unambiguous) and Q_2 (ambiguous):

$$\begin{array}{lll} \text{Q}_1 \mapsto \text{S}\ \text{Q}_1 & \text{Q}_1 \mapsto \sigma & \text{S} \mapsto \sigma; \\ \text{Q}_2 \mapsto \text{Q}_2\ \text{Q}_2 & \text{Q}_2 \mapsto \sigma. & \end{array}$$

Both queries specify the language $\mathcal{L} = \{\sigma^k \mid k \geq 1\}$. As in the previous experiment, we use the cycle graphs with w nodes of Figure 4. On these cycle

[2] A grammar $\mathcal{C}$ with language $\mathcal{L}$ is said to be ambiguous if there exists a string $s \in \mathcal{L}$ for which multiple derivation trees exist based on the production rules of grammar $\mathcal{C}$ [22].

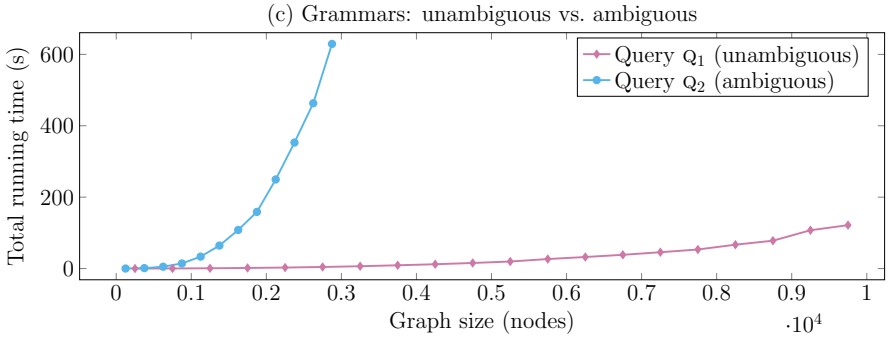

Figure 8: Measurements for the *grammars: unambiguous vs. ambiguous* experiment. In this experiment, we measured the performance of MinimizeSetGG as a function of the graph size. Specifically, we compare the performance for evaluating an *unambiguous query* Q_1 that has unique derivations for any string it describes and an *ambiguous* P_2 that has multiple different derivations for the strings it describes.

graphs, both queries will evaluate to very dense results sets. We measured the running time of MinimizeSetGG for both queries. The results of the experiment can be found in Figure 8.

As is clear from the results, evaluation of the unambiguous query Q_1 is magnitudes faster than evaluation of the ambiguous query Q_2, even though MinimizeSetGG does not yet optimize for deterministic or unambiguous grammars. The reason for this is simple: for any shortest path in the graph, Q_2 has many different ways to derive the trace of this path, whereas Q_1 only has a single derivation. Consequently, MinimizeSetGG will have to inspect many more choices while evaluating Q_2.

This experiment shows that MinimizeSetGG already benefits from simple unambiguous queries. Still, we believe that specialized algorithms that *only* operate on deterministic and unambiguous grammars can be further optimized, a direction we leave open for future work.

6. Related Work

There is an abundant literature on graph queries, formal languages, and context-free grammars. There is only limited work toward answering graph queries with paths, however. Likewise, there is only limited work on the related problem of deriving shortest strings from grammars. Next, a brief overview.

6.1. Path-based semantics

As stated before, path-based semantics have only gained limited attention. For the regular expressions, Barceló et al. [34] introduced the extended regular path queries that have path variables for output. The main focus of Barceló et al. is, however, on the use of path variables for expressivity purposes, and path-based results are only studied in limited details. Recent work by Hofman et al. [20]

provides an alternative to use path-based query semantics for debugging: to gain more insight in the behavior of regular path queries with respect to the expected behavior, Hofman et al. propose a technique based on separability. Although this approach addresses query debugging, it does not lift the other limitations of the traditional query semantics used to evaluate path queries. Several recent works [35–37] have observed the lack of alternatives to the traditional relational semantics and study the theoretical properties of subgraph-based and path-based semantics when evaluating graph queries. None of these works look at languages with the expressive power of the context-free path queries, implement evaluation algorithm, or provide practical evaluations of their approaches, however.

In practical graph database systems, path-based results can already be used in some limited settings [3]. E.g., SPARQL can return RDF graphs via `CONSTRUCT` queries [9], which can be used to encode fixed-size paths; whereas Gremlin can enumerate graph traversal steps (which can encode paths) via the `.path()` step, which comes at prohibitive high costs [12]. In the setting of model checking using CTL [5], path-based query semantics are widely used to produce witnesses and counterexamples that show why the graph does or does not meet the conditions expressed by the CTL formulae. Unfortunately, model checking languages lack the expressive power found in most path query languages used to query graph databases. This sharply contrasts our work, as we show that path-based results are viable both in theory and in practice, this even for complex context-free path queries. Hence, to the best of our knowledge, our work is the first to systematically formalize and study path results for complex graph query languages.

Barrett et al. [27] studies variations of the single-path semantics we propose in this work. They do so from a complexity-theoretical standpoint, however, by classifying the complexity of query evaluation using variations of our single-path semantics. E.g., they show that the single-path semantics is feasible for regular path queries and context-free path queries, but becomes unfeasible when only simple paths are to be returned. As their focus is on classifying the complexity of evaluation, Barret et al. do not provide practical algorithms for the evaluation of path queries using the single-path semantics. We improve on this work by providing the algorithm MinimizeSetGG, an efficient algorithm for evaluating context-free path queries on graphs using the single-path semantics.

6.2. Deriving shortest strings

In this work we have shown that the evaluation of context-free path queries on graphs using the single-path semantics can be reduced to the derivation of a shortest string from a grammar. Mclean et al. [30] proved that such a shortest string could be computed effective given a grammar, but failed to give a practical algorithm for doing so. We improve on these results by providing the algorithm MinimizeSet, an efficient algorithm for computing the shortest string in a grammar. Other works, e.g. [38–41], provide ways to enumerate strings in a grammar, but these algorithms cannot effectively be used to quickly find the shortest such string.

6.3. Worst-case bounds on the length of shortest strings

Problem 1, which we studied in Section 4 of this paper, is closely related to the study of the *rational index* of a context-free language [42, 43]: given a context-free language $\mathcal{L}$, the rational index $\rho_{\mathcal{L}}(n)$ is the worst-case length of the shortest string in the intersection of any automaton with at-most n states and context-free language $\mathcal{L}$. Our Problem 1 generalizes the rational index of a given context-free language [42, 43], as we try to determine the worst-case rational index for *all* context-free languages that can be represented by a grammar in Chomsky Normal Form with at-most m non-terminals.

7. Conclusions and Future Work

To address the limitations of the traditional semantics for evaluating path queries, such as the regular path queries and the context-free path queries, we proposed the single-path semantics. This path-based semantics is not only useful for end-users, but also enables new directions in the design of graph query languages and enables new tools for graph analytics, data exploration, data provenance, and debugging of complex path queries. To show the practical viability of the single-path semantics, we also propose algorithms that evaluate context-free path queries using the single-path semantics. Our results are promising: our experimental evaluation shows that queries can be evaluated using the single-path semantics with little effort, even in cases where the path-based query results are very large. Furthermore, we studied the problem of determining worst-case length of the shortest paths found when evaluating queries using the single-path semantics.

Based on our results, we see several avenues for the further study of evaluating queries with path-based semantics:

1. The algorithms in our paper are *bottom-up* and are tuned toward evaluating a query over the entire graph. In this case, the bounded vs. unbounded experiment of Section 5.2 has already shown that queries can be evaluated efficiently using the single-path semantics *if* the size of the result set is limited. In many practical applications on *huge graphs*, the end-user is only interested in a part of the graph, e.g., paths that originate or end at a certain node n_c. Hence, the expected result sets in these applications are limited. The bottom-up algorithms we present evaluate qeries over the entire graph, however, due to which they cannot take advantage of the expected limited size of practical query result sets on huge graphs (e.g., by only focusing on the subgraph close to node n_c). To provide practical single-path evaluation performance for such applications, we are interested in the development of *top-down* and *goal-oriented* algorithms.
2. Our measurements showed that the cost of evaluating a context-free path query depends heavily on the structure of the grammar used by the query: evaluating different grammars that express the same query can have widely different costs. This raises an interesting query optimization question: can we automatically optimize grammars to reduce the cost of evaluation?

3. Furthermore, it is open whether simpler, more efficient, query evaluation algorithms exist for restricted classes of context-free grammars (e.g., deterministic grammars or unambiguous grammars [21]). It is not directly clear if such algorithms exist: deterministic and unambiguous grammars will still face ambiguity and non-deterministic choices in their evaluation on graphs, as complex graphs can have many paths with the same traces.

References

[1] J. Hellings, Explaining results of path queries on graphs, in: Software Foundations for Data Interoperability and Large Scale Graph Data Analytics, Springer, 2020, pp. 84–98. doi:10.1007/978-3-030-61133-0_7.

[2] U. Alon, An Introduction to Systems Biology: Design Principles of Biological Circuits, Chapman and Hall/CRC, 2006.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, D. Vrgoč, Foundations of modern query languages for graph databases, ACM Comput. Surv. 50 (2017) 68:1–68:40.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan, Extensible Markup Language (XML) 1.1 (Second Edition), Technical Report, W3C, 2006. http://www.w3.org/TR/2006/REC-xml11-20060816.

[5] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, The MIT Press, 1999.

[6] G. Schreiber, Y. Raimond, RDF 1.1 Primer, Technical Report, W3C, 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624.

[7] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon, XML Path Language (XPath) 2.0 (Second Edition), Technical Report, W3C, 2010. http://www.w3.org/TR/2010/REC-xpath20-20101214/.

[8] J. Clark, S. DeRose, XML Path Language (XPath) Version 1.0, Technical Report, W3C, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116/.

[9] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, Technical Report, W3C, 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321.

[10] L. Libkin, W. Martens, D. Vrgoč, Querying graph databases with XPath, in: Proceedings of the 16th International Conference on Database Theory, ACM, 2013, pp. 129–140.

[11] I. Robinson, J. Webber, E. Eifrem, Graph Databases: New Opportunities for Connected Data, 2nd ed., O'Reilly Media, Inc., 2015.

[12] M. A. Rodriguez, The gremlin graph traversal machine and language (invited talk), in: Proceedings of the 15th Symposium on Database Programming Languages, ACM, New York, NY, USA, 2015, pp. 1–10.

[13] D. Kozen, Kleene algebra with tests, ACM Trans. Program. Lang. Syst. 19 (1997) 427–443.

[14] M. Lange, Model checking propositional dynamic logic with all extras, J. Appl. Log. 4 (2006) 39–49.

[15] P. Barceló, Querying graph databases, in: Proceedings of the 32nd Symposium on Principles of Database Systems, ACM, 2013, pp. 175–188.

[16] D. Harel, A. Pnueli, J. Stavi, Propositional dynamic logic of nonregular programs, J. Comput. Syst. Sci. 26 (1983) 222–243.

[17] J. Hellings, Conjunctive context-free path queries, in: Proceedings of the 17th International Conference on Database Theory (ICDT 2014), 2014, pp. 119–130.

[18] P. Sevon, L. Eronen, Subgraph queries by context-free grammars, J. Integr. Bioinform. 5 (2008) 157–172.

[19] J. Cheney, L. Chiticariu, W.-C. Tan, Provenance in databases: Why, how, and where, Foundations and Trends in Databases 1 (2009) 379–474.

[20] P. Hofman, W. Martens, Separability by short subsequences and subwords, in: 18th International Conference on Database Theory, volume 31 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 230–246.

[21] D. Grune, C. J. Jacobs, Parsing Techniques: A Practical Guide, 2nd ed., Springer-Verlag New York, 2008.

[22] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3th edition, Pearson, 2007.

[23] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Y. Vardi, Containment of conjunctive regular path queries with inverse, in: Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann Publishers Inc., 2000, pp. 176–185.

[24] G. H. L. Fletcher, M. Gyssens, D. Leinders, D. Surinx, J. Van den Bussche, D. Van Gucht, S. Vansummeren, Y. Wu, Relative expressive power of navigational querying on graphs, Information Sciences 298 (2015) 390–406.

[25] D. Surinx, G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, Y. Wu, Relative expressive power of navigational querying on graphs using transitive closure, Logic Journal of the IGPL 23 (2015) 759–788.

[26] M. Arenas, S. Conca, J. Pérez, Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard, in: Proceedings of the 21st International Conference on World Wide Web, ACM, 2012, pp. 629–638.

[27] C. Barrett, R. Jacob, M. Marathe, Formal-language-constrained path problems, SIAM J. Comput. 30 (2000) 809–837.

[28] K. Losemann, W. Martens, The complexity of evaluating path expressions in SPARQL, in: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2012, pp. 101–112.

[29] Y. Bar-Hillel, M. A. Perles, E. Shamir, On formal properties of simple phrase structure grammars, Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung 14 (1961) 143–172.

[30] M. J. Mclean, D. B. Johnston, An algorithm for finding the shortest terminal strings which can be produced from non-terminals in context-free grammars, in: Combinatorial Mathematics III, volume 452 of *Lecture Notes in Mathematics*, Springer Berlin Heidelberg, 1975, pp. 180–196.

[31] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (1987) 596–615.

[32] L. Alpoge, T. Ang, L. Schaeffer, J. Shallit, Decidability and shortest strings in formal languages, in: Descriptional Complexity of Formal Systems, Springer, 2011, pp. 55–67.

[33] G. Pighizzini, J. Shallit, M. Wang, Unary context-free grammars and pushdown automata, descriptional complexity and auxiliary space lower bounds, J. Comput. Syst. Sci. 65 (2002) 393–414.

[34] P. Barceló, L. Libkin, A. W. Lin, P. T. Wood, Expressive languages for path queries over graph-structured data, ACM Trans. Database Syst. 37 (2012) 31:1–31:46.

[35] R. García, R. Angles, An algebra for path manipulation in graph databases, in: Advances in Databases and Information Systems, Springer International Publishing, 2022, pp. 61–74.

[36] W. Martens, M. Niewerth, T. Popp, C. Rojas, S. Vansummeren, D. Vrgoč, Representing paths in graph database pattern matching, Proc. VLDB Endow. 16 (2023).

[37] T. Cucumides, J. Reutter, D. Vrgoč, Size bounds and algorithms for conjunctive regular path queries, in: 26th International Conference on Database Theory (ICDT 2023), volume 255 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl, 2023, pp. 13:1–13:17. doi:10.4230/LIPIcs.ICDT.2023.13.

[38] P. Dömösi, Unusual algorithms for lexicographical enumeration, Acta Cybern. 14 (2000) 461–468.

[39] C. C. Florêncio, J. Daenen, J. Ramon, J. V. den Bussche, D. V. Dyck, Naive infinite enumeration of context-free languages in incremental polynomial time, J. Univers. Comput. Sci. 21 (2015) 891–911.

[40] Y. Dong, Linear algorithm for lexicographic enumeration of CFG parse trees, Science in China Series F: Information Sciences 52 (2009) 1177–1202.

[41] E. Mäkinen, On lexicographic enumeration of regular and context-free languages, Acta Cybern. 13 (1997) 55–61.

[42] L. Boasson, B. Courcelle, M. Nivat, The rational index: A complexity measure for languages, SIAM Journal on Computing 10 (1981) 284–296.

[43] J.-L. Deleage, L. Pierre, The rational index of the Dyck language $D'_1{}^*$, Theoretical Computer Science 47 (1986) 335–343.