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

Jelle Hellings

Department of Computer Science, University of California, Davis
Davis, CA 95616-8562, USA
jhellings@ucdavis.edu

Abstract. Many graph query languages use, at their core, path queries that yield node pairs that are connected by a path of interest. For the end-user, such node pairs only give limited insight as to *why* this query result is obtained, as the pair does not directly identify the underlying path of interest. To address this limitation of path queries, we propose the *single-path semantics*, which evaluates path queries to, for each node pair (m, n), a single path from m to n satisfying the conditions of the query. To put our proposal in practice, we provide an efficient algorithm for evaluating *context-free path queries*, a particular powerful type of 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.

Keywords: Graph Queries · Path Results · Context-Free Path Queries

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 family trees, social networks, process models, gene networks, XML data, and RDF data [1,2,9,12,30]. As an example, consider the small social network visualized in Figure 1 in which nodes represent peoples 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 [8,9,11],

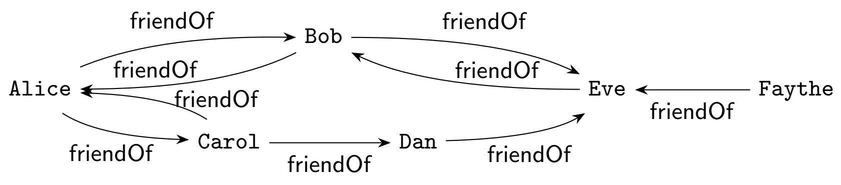

Fig. 1. A typical example of graph data: a social network relating peoples.

SPARQL for querying RDF data [19,30], the graph query languages GXPath [24], Cypher [28], and Gremlin [29], and formal verification languages such as PDL, KAT, CTL, and LTL [12,22,23]. 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 [2]. Examples of such path queries are the well-known regular path queries [5] and the context-free path queries [18,20,23,31]. 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 between 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 $(\mathtt{Alice}, \mathtt{Eve})$ 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 "$\mathtt{Alice}\ \mathsf{friendOf}\ \mathtt{Bob}\ \mathsf{friendOf}\ \mathtt{Eve}$", 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 "$\mathtt{Alice}\ \mathsf{friendOf}\ \mathtt{Carol}\ \mathsf{friendOf}\ \mathtt{Dan}\ \mathsf{friendOf}\ \mathtt{Eve}$" cannot be in the output.

The need for the single-path semantics extends beyond the above toy example. 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 [10], this 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 [21], 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 cannot only express all typical path queries (e.g. [5,20]), but also have applications in model checking [23], bio-informatics [31], and parser construction [17]. In specific, we formalize the *single-path semantics*, introduce the algorithm MinimizeSetGG that provides efficient evaluation of context-free path queries on graphs using the single-path semantics, and evaluate the performance of MinimizeSetGG in practice. Our results show promise, as MinimizeSetGG can easily answer queries whose results contains 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.

Let Σ be a set of symbols. 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.

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 $\pi = m_1\ \sigma_1\ m_2 \dots m_{n-1}\ \sigma_{n-1}\ m_n$ such that, for every $m_i\ \sigma_i\ m_{i+1}$ in the sequence, $1 \leq i < n$, we have $(m_i, \sigma_i, m_{i+1}) \in \delta$. We write $m_1 \pi m_n$ to indicate that π is a path starting at node m_1 and ending at node m_n. We write $|\pi| = n - 1$ to denote the length of π and we write $\text{trace}(\pi) = \sigma_1 \dots \sigma_{n-1}$ to denote the trace of π, the string represented by the sequence of edge labels in π.

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). The evaluation of q on graph $\mathfrak{G}$ using the standard *relational semantics* simply consists of all node pairs that are connected by paths whose trace is in $\mathcal{L}$. To denote the evaluation of q on $\mathfrak{G}$, we write $[\![q]\!]_{\mathfrak{G}}$, and we have $[\![q]\!]_{\mathfrak{G}} = \{(m, n) \mid \exists \text{ path } m\pi n \text{ in } \mathfrak{G} \text{ with } \text{trace}(\pi) \in \mathcal{L}\}$.

Example 3. In Example 1, the query $\mathsf{indirectFriendOf}$ was expressed by the regular expression $\mathsf{friendOf}^+$. This regular expression represents the 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 $\pi =$ "$\texttt{Alice}\ \mathsf{friendOf}\ \mathsf{Bob}\ \mathsf{friendOf}\ \texttt{Eve}$" with $\text{trace}(\pi) \in \mathcal{L}$.

A *grammar* is a triple $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$, 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 ($\mathcal{N} \cap \Sigma = \emptyset$). 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 we can rewrite s into $s_1 \circ s' \circ s_2$ by applying the rewrite $\text{A} \mapsto s'$. We write $s \to_{\mathcal{P}}^* s'$ if s can be rewritten into s' using production rules in $\mathcal{P}$, and we write $s \to_{\mathcal{P}}^{+} s'$ if $s \to_{\mathcal{P}}^{*} s'$ and at least one rewrite step is necessary to rewrite s into s'.

The *language* of non-terminal $\textsc{a} \in \mathcal{N}$ is defined by $\mathcal{L}(\mathscr{C}; \textsc{a}) = \{s \in \Sigma^* \mid \textsc{a} \to_{\mathcal{P}}^{*} s\}$. Given a grammar with non-terminal $\textsc{a}$, we simply write $\textsc{a}$ to denote the path query based on the language $\mathcal{L}(\mathscr{C}; \textsc{a})$.

Example 4. Consider the grammar $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ in which $\mathcal{N} = \{\textsc{a}\}$, $\Sigma = \{\mathsf{friendOf}\}$, and $\mathcal{P} = \{\textsc{a} \to \mathsf{friendOf}, \textsc{a} \to \textsc{a}\ \textsc{a}\}$. The language $\mathcal{L}(\mathscr{C}; \textsc{a})$ is equivalent to the language $\mathcal{L}$ of Example 3. Hence, we have $(\mathtt{Alice}, \mathtt{Eve}) \in [\![\textsc{a}]\!]_{\mathfrak{G}}$, in which $\mathfrak{G}$ is the graph visualized in Figure 1.

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 5. 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.

Restricting the paths considered in the evaluation, e.g., to simple paths, assures that the set of paths considered is finite. Unfortunately, changing the paths considered during evaluation defeats the purpose of path-based semantics as a data provenance and debugging tool for normal path queries. Moreover, it is well-known that such restrictions make query evaluation prohibitive expensive [2,3,7,25]. Restricting the number of paths in the result, e.g., to a single path per node pair, will also assure a finite result. Unfortunately, as Example 5 already shows, individual paths in such a finite result set can still have a practically unbounded length. To address these issues, we choose to return a single as-short-as-possible path for each node-pair (m, n):

Definition 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* $\mathrm{single}(q|_{\mathfrak{G}})$*, yields, for every $(m, n) \in [\![q]\!]_{\mathfrak{G}}$, a single shortest path $m\pi n$ in $\mathfrak{G}$ such that* $\mathrm{trace}(\pi) \in \mathcal{L}$*. (Hence, for every other path $m\pi' n$ in $\mathfrak{G}$ with* $\mathrm{trace}(\pi') \in \mathcal{L}$*, we have $|\pi| \leq |\pi'|$).*

Toward evaluating context-free path queries using the single-path semantics, we proceed 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 automata 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 1 (Bar-Hillel et al. [4]). *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ 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 $\mathcal{L}(\mathscr{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$ can be represented by a grammar.*

Lemma 1 guarantees that there is a finite representation of the set of all strings in $\mathcal{L}(\mathscr{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n)$, each such string representing the trace of a path $m\pi n$ in $\mathfrak{G}$ with $\text{trace}(\pi) \in \mathcal{L}(\mathscr{C}; \text{A})$. Unfortunately, there can be several paths with the same trace, complicating the derivation of the underlying paths. To improve on this, we show the existence of *graph-annotated grammars* that directly represent the set of paths instead of their traces:

Definition 2. *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ 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 $\text{A}|_{mn}$. An* annotated grammar *over $(\mathscr{C}, \mathfrak{G})$ is a grammar $\mathscr{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$ in which*

1. *$\mathcal{N}|_{\mathfrak{G}} = \{\text{A}|_{mn} \in \mathcal{N} \times \mathcal{V}^2 \mid \mathcal{L}(\mathscr{C}; \text{A}) \cap \mathcal{L}(\mathfrak{G}; m, n) \neq \emptyset\}$;*
2. *$\mathcal{P}|_{\mathfrak{G}} = P_\Sigma \cup P_\mathcal{N}$ with $P_\Sigma = \{\text{A}|_{mn} \mapsto \sigma \mid (m, \sigma, n) \in \delta \wedge (\text{A} \mapsto \sigma) \in \mathcal{P}\}$ and $P_\mathcal{N} = \{\text{A}|_{mn} \mapsto \text{B}|_{mo}\ \text{C}|_{on} \mid (\text{A} \mapsto \text{B C}) \in \mathcal{P}\}$.*

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. Notice that $|\mathcal{N}|_{\mathfrak{G}}| \leq |\mathcal{N}||\mathcal{V}|^2$, $|P_\Sigma| \leq |\mathcal{P}||\delta|$, and $|P_\mathcal{N}| \leq |\mathcal{P}||\mathcal{V}|^3$. We illustrate these annotated grammars with an example:

Example 6. Let $\mathfrak{G}$ be the graph visualized in Figure 1 and $\mathscr{C}$ the grammar of Example 4. We construct the annotated grammar $\mathscr{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$. For brevity, we refer to each person by the first letter of their name. We have

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

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., $\text{Q}|_{\texttt{AB}} \in P_\Sigma$, as $(\texttt{Alice}, \mathsf{friendOf}, \texttt{Bob})$ is and edge in $\mathfrak{G}$, and $(\text{Q}|_{\texttt{AB}} \mapsto \text{Q}|_{\texttt{AD}}\ \text{Q}|_{\texttt{DB}}) \in P_\mathcal{N}$, as there is a $\mathsf{friendOf}$-labeled path from `Alice` to `Dan` and another $\mathsf{friendOf}$-labeled path from `Dan` to `Bob`. To produce a path from Alice to Eve, we use $\mathscr{C}|_{\mathfrak{G}}$:

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

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

Using induction, we can prove that graph-annotated grammars can always be used as illustrated in Example 6:

Proposition 1. *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar, let $\mathfrak{G} = (\mathcal{V}, \Sigma, \delta)$ be a graph, let $\mathscr{C}|_{\mathfrak{G}} = (\mathcal{N}|_{\mathfrak{G}}, \Sigma, \mathcal{P}|_{\mathfrak{G}})$ be the annotated grammar over $(\mathscr{C}, \mathfrak{G})$, let $m\pi n$ be a path in $\mathfrak{G}$, and let $\mathrm{A} \in \mathcal{N}$ be a non-terminal. We have* $\mathrm{trace}(\pi) \in \mathcal{L}(\mathscr{C}; \mathrm{A})$ *if and only if we can derive π from $\mathrm{A}|_{mn} \in \mathcal{N}|_{\mathfrak{G}}$.*

We note that 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

Next, we propose an efficient way to compute a shortest string in the language defined by a grammar. Mclean et al. [27] already proved that a shortest string can be computed effective given a grammar, but did not provide a practical algorithm for computing shortest strings. Toward such an algorithm, we introduce rewrites using *simple* production rules:

Definition 3. *Let $\mathcal{P}$ be a set of production rules. We define* $\mathrm{heads}(\mathcal{P}) = \{\mathrm{A} \mid (\mathrm{A} \mapsto s) \in \mathcal{P}\}$ *and we define the set of non-terminals derivable from* A *using the production rules in $\mathcal{P}$ by $\langle \mathrm{A} \rangle_{\mathcal{P}} = \{\mathrm{B} \in \mathcal{N} \mid \exists s_1 \exists s_2\ \mathrm{A} \rightarrow^{+}_{\mathcal{P}} s_1 \circ \mathrm{B} \circ s_2\}$.*

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

A simple set of production rules $\mathcal{P}$ over alphabet Σ can be used to rewrite every non-terminal $\mathrm{A} \in \mathrm{heads}(\mathcal{P})$ into a unique string $\mathrm{ustring}_{\mathcal{P}}(\mathrm{A})$ over Σ in a straightforward manner, as $\mathcal{P}$ does not provide any choices during such a rewrite.

Example 7. Consider Example 6. 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:

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

We can use simple production rules derived from a grammar to represent shortest strings in that grammar:

Lemma 2. *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ 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 $\mathcal{L}(\mathscr{C}; \text{A}) \neq \emptyset$, $\text{ustring}_{\mathcal{P}'}(\text{A})$ is a shortest string in $\mathcal{L}(\mathscr{C}; \text{A})$.*

We say that a set of production rules that satisfies the conditions of Lemma 2 is *minimizing*. Unfortunately, not every simple set of production rules is minimizing:

Example 8. Consider Example 7. 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 $\text{A} \in \text{heads}(\mathcal{P})$. 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 2, *left*.

The MinimizeSet algorithm works rather intuitively. Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be a grammar. Production rules of the form $(\text{A} \to \sigma) \in \mathcal{P}$, $\sigma \in \Sigma$, produce the shortest possible strings: if $(\text{A} \to \sigma) \in \mathcal{P}$, then σ is a shortest string in $\mathcal{L}(\mathscr{C}; \text{A})$. 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 $\mathit{cost}(\text{A})$ of the shortest strings in $\mathcal{L}(\mathscr{C}; \text{A})$. 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 $(\text{C} \mapsto \text{A B}) \in \mathcal{P}$ or $(\text{C} \mapsto \text{B A}) \in \mathcal{P}$, whether using this production rule will allow us to rewrite C into a shorter string than the currently-found string with length $\mathit{cost}[\text{C}]$ (Line 10 and Line 12). We do so by rewriting—in this production rule—A to a string of length $\mathit{cost}(\text{A})$ (Line 14).

Theorem 1. *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ be grammar. Execution of* MinimizeSet*($\mathscr{C}$) yields a minimizing set of production rules $\mathcal{P}'$ for $\mathscr{C}$ in $\mathcal{O}(|\mathcal{N}|(|\mathcal{N}| \log|\mathcal{N}| + |\mathcal{P}|))$. Using $\mathcal{P}'$, a set R of shortest strings s_{A} in $\mathcal{L}(\mathscr{C}; \text{A})$, $\text{A} \in \mathcal{N}$, 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). The main *while*-loop maintains the following invariants:

1. The set $\{\mathcal{P}'[\text{A}] \mid \text{A} \in \mathcal{P}'\}$ is simple.
2. If $\text{A} \in \mathcal{P}'$ and $\mathcal{P}'[\text{A}] = (\text{A} \mapsto \text{B C})$, then $\mathit{cost}[\text{A}] \geq \mathit{cost}[\text{B}] + \mathit{cost}[\text{C}]$, $\mathit{cost}[\text{A}] > \mathit{cost}[\text{B}]$, and $\mathit{cost}[\text{A}] > \mathit{cost}[\text{C}]$.
3. If $\text{A} \in \mathcal{P}'$ and s is a shortest string in $\mathcal{L}(\mathscr{C}; \text{A})$, then $|s| \leq \mathit{cost}[\text{A}]$.
4. Let m be the priority of the last element removed from *new*. No new element is inserted in *new* with priority less than or equal to m.
5. Let m be the priority of the last element removed from *new*. For every $\text{A} \in \mathcal{N}$ and every shortest string s in $\mathcal{L}(\mathscr{C}; \text{A})$ with $|s| \leq m$, we have $\mathit{cost}[\text{A}] = |s|$.

As each non-terminal is added to *new* at most once, the MinimizeSet algorithm terminates. At termination, Invariants 1–5 guarantee that the resulting set of production rules is minimizing.

```
Algorithm MinimizeSet($\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$):
$\mathcal{P}', \mathit{cost}$ := empty mapping, empty mapping.
$\mathit{new}$ is a min-priority queue.
for all $(\textsc{a} \mapsto \sigma) \in \mathcal{P}$ do
    if $\textsc{a} \notin \mathit{cost}$ then
        $\mathit{cost}[\textsc{a}], \mathcal{P}'[\textsc{a}] := 1, (\textsc{a} \mapsto \sigma)$.
        add $\textsc{a}$ to $\mathit{new}$ with priority 1.
while $\mathit{new} \neq \emptyset$ do
    Take $\textsc{a}$ with minimum priority in $\mathit{new}$.
    Remove A from $\mathit{new}$.
    for all $(\textsc{c} \mapsto \textsc{a}\ \textsc{b}) \in \mathcal{P}$ with $\textsc{b} \in \mathit{cost}$ do
        produce($\textsc{c} \mapsto \textsc{a}\ \textsc{b}$).
    for all $(\textsc{c} \mapsto \textsc{b}\ \textsc{a}) \in \mathcal{P}$ with $\textsc{b} \in \mathit{cost}$ do
        produce($\textsc{c} \mapsto \textsc{b}\ \textsc{a}$).
return $\{\mathcal{P}'[\textsc{a}] \mid \textsc{a} \in \mathcal{P}'\}$.

Procedure produce($\textsc{d} \mapsto \textsc{e}\ \textsc{f}$):
if $\textsc{d} \notin \mathit{cost}$ then
    $\mathit{cost}[\textsc{d}] := \mathit{cost}[\textsc{e}] + \mathit{cost}[\textsc{f}]$.
    $\mathcal{P}'[\textsc{d}] := \textsc{d} \mapsto \textsc{e}\ \textsc{f}$.
    Add $\textsc{d}$ to $\mathit{new}$ with priority $\mathit{cost}[\textsc{e}] + \mathit{cost}[\textsc{f}]$.
else if $\mathit{cost}[\textsc{d}] > \mathit{cost}[\textsc{e}] + \mathit{cost}[\textsc{f}]$ then
    $\mathit{cost}[\textsc{d}] := \mathit{cost}[\textsc{e}] + \mathit{cost}[\textsc{f}]$.
    $\mathcal{P}'[\textsc{d}] := \textsc{d} \mapsto \textsc{e}\ \textsc{f}$.
    Lower priority of $\textsc{d} \in \mathit{new}$ to $\mathit{cost}[\textsc{e}] + \mathit{cost}[\textsc{f}]$.
```

```
Algorithm MinimizeSetGG($\mathscr{C}$, $\mathfrak{G}$):
$\mathcal{P}', \mathit{cost}$ := empty mapping, empty mapping.
$\mathit{new}$ is a min-priority queue.
for all $(\textsc{a} \mapsto \sigma) \in \mathcal{P}$ and $(m, \sigma, n) \in \delta$ do
    if $\textsc{a}|_{mn} \notin \mathit{cost}$ then
        $\mathit{cost}[\textsc{a}|_{mn}], \mathcal{P}'[\textsc{a}|_{mn}] := 1, (\textsc{a}|_{mn} \mapsto \sigma)$.
        Add $\textsc{a}|_{mn}$ to $\mathit{new}$ with priority 1.
while $\mathit{new} \neq \emptyset$ do
    Take $\textsc{a}|_{mn}$ with minimum priority in $\mathit{new}$.
    Remove $\mathrm{A}|_{mn}$ from $\mathit{new}$.
    for all $(\textsc{c} \mapsto \textsc{a}\ \textsc{b}) \in \mathcal{P}$ with $\textsc{b}|_{no} \in \mathit{cost}$ do
        produceGG($\textsc{c}|_{mo} \mapsto \textsc{a}|_{mn}\ \textsc{b}|_{no}$).
    for all $(\textsc{c} \mapsto \textsc{b}\ \textsc{a}) \in \mathcal{P}$ with $\textsc{b}|_{om} \in \mathit{cost}$ do
        produceGG($\textsc{c}|_{on} \mapsto \textsc{b}|_{om}\ \textsc{a}|_{mn}$).
return $\{\mathcal{P}'[\textsc{a}|_{mn}] \mid \textsc{a}|_{mn} \in \mathcal{P}'\}$.

Procedure produceGG($\textsc{d}|_{uw} \mapsto \textsc{e}|_{uv}\ \textsc{f}|_{vw}$):
if $\textsc{d}|_{uw} \notin \mathit{cost}$ then
    $\mathit{cost}[\textsc{d}|_{uw}] := \mathit{cost}[\textsc{e}|_{uv}] + \mathit{cost}[\textsc{f}|_{vw}]$.
    $\mathcal{P}'[\textsc{d}|_{uw}] := \textsc{d}|_{uw} \mapsto \textsc{e}|_{uv}\ \textsc{f}|_{vw}$.
    Add $\textsc{d}|_{uw}$ to $\mathit{new}$ with priority $\mathit{cost}[\textsc{e}|_{uv}] + \mathit{cost}[\textsc{f}|_{vw}]$.
else if $\mathit{cost}[\textsc{d}|_{uw}] > \mathit{cost}[\textsc{e}|_{uv}] + \mathit{cost}[\textsc{f}|_{vw}]$ then
    $\mathit{cost}[\textsc{d}|_{uw}] := \mathit{cost}[\textsc{e}|_{uv}] + \mathit{cost}[\textsc{f}|_{vw}]$.
    $\mathcal{P}'[\textsc{d}|_{uw}] := \textsc{d}|_{uw} \mapsto \textsc{e}|_{uv}\ \textsc{f}|_{vw}$
    Lower priority of $\textsc{d}|_{uw} \in \mathit{new}$ to $\mathit{cost}[\textsc{e}|_{uv}] + \mathit{cost}[\textsc{f}|_{vw}]$.
```

Fig. 2. On the *left*, the MinimizeSet algorithm that constructs a minimizing set of production rules for the grammar $\mathscr{C}$. On the *right*, the MinimizeSetGG algorithm that constructs a minimizing set of production rules for the annotated grammar $\mathscr{C}|_{\mathfrak{G}}$, of which only the necessary parts are implicitly constructed.

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 $1, \dots, 2^{|\mathcal{N}|-1}$, 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 [16]. 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 the above results, we can already answer context-free path queries under the single-path semantics, this by applying MinimizeSet on an annotated grammar. Unfortunately, this approach has high overhead due to the explicit construction and storing of the annotated grammar. Luckily, during the execution of MinimizeSet, the relevant parts of $\mathscr{C}|_{\mathfrak{G}}$ can be implicitly derived from $\mathscr{C}$ and $\mathfrak{G}$. We obtain the MinimizeSetGG algorithm by integrate these implicit derivation steps into MinimizeSet. The resulting pseudo-code can be found in Figure 2, *right*. We conclude:

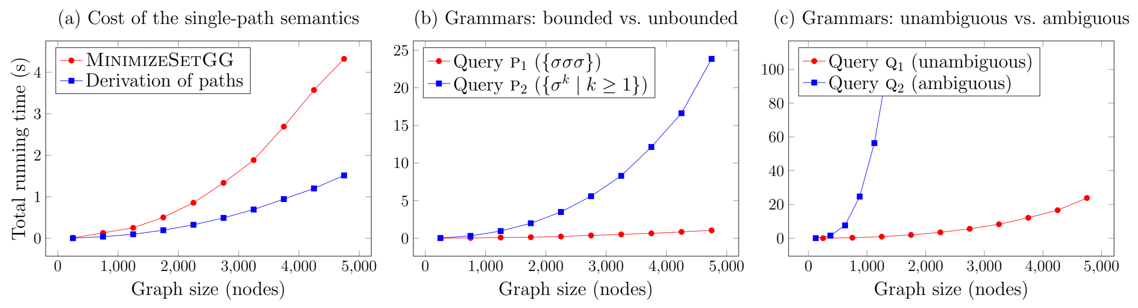

Fig. 3. Measurements on the performance of MinimizeSetGG.

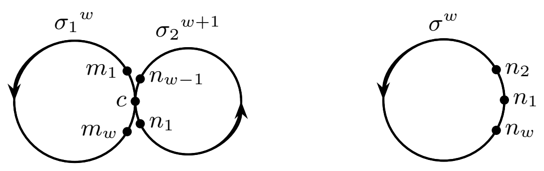

Fig. 4. On the *left*, the double-cyclic graph: two cycles, one having $w - 1$ edges labeled with σ_1, and one having w edges labeled with σ_2. The two cycles are connected via a shared node c. On the *right*, the cyclic graph having w nodes labeled with σ.

Theorem 2. *Let $\mathscr{C} = (\mathcal{N}, \Sigma, \mathcal{P})$ 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.*

4 Empirical Evaluation

To show that the path-based semantics for context-free path are viable in practice, we implemented the MinimizeSetGG algorithm of Figure 2, *right*, and the straightforward path derivation algorithm in C++14. Open-source code of the full C++14 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 i5-4670 CPU, running at a maximum of 3.8 GHz, and with 16 GiB of main memory. In each of our experiments, we test with synthetic graphs that are designed specifically to test extreme-case behavior of the algorithms. Visualizations of these graphs can be found in Figure 4.

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. We use the double-cyclic graphs of Figure 4, as this combination of query and graphs produces very large path. More specifically, we have proven that, in this case, the longest shortest paths have a size that is *quadratic* in the size of the graph, whereas for all single-symbol grammars and regular grammars the maximum size is only *linear* in the size of the graph (details omitted due to space limitations). The results of the experiment can be found in Figure 3(a). As is clear from the results, single-path evaluation is practically feasible: the query $\textsc{q}$, evaluated on a double-cyclic graph of 4750 nodes, yields a set of $11 \cdot 10^6$ distinct paths, of which the longest (non-simple) path has $11 \cdot 10^6$ edges, and the average path has $5.6 \cdot 10^6$ edges. Hence, the query result is large. Still, MinimizeSetGG finished in only 4.3 s and the longest path was constructed in 1.5 s. Hence, even for queries and graphs that produce very large results, the query costs are reasonable.

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. E.g., 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 $\textsc{p}_1$ (bounded) and $\textsc{p}_2$ (unbounded):

$$\begin{aligned} &\textsc{p}_1 \mapsto \textsc{s}\ \textsc{b} && \textsc{b} \mapsto \textsc{s}\ \textsc{s} && \textsc{s} \mapsto \sigma; \\ &\textsc{p}_2 \mapsto \textsc{s}\ \textsc{p}_2 && \textsc{p}_2 \mapsto \sigma && \textsc{s} \mapsto \sigma. \end{aligned}$$

The language described by $\textsc{p}_1$ is $\mathcal{L}_1 = \{\sigma\sigma\sigma\}$, and the language described by $\textsc{p}_2$ is $\mathcal{L}_2 = \{\sigma^k \mid k \geq 1\}$. We use the cycle graphs with w nodes of Figure 4, on which query $\textsc{p}_1$ will evaluate to a very sparse result set of w paths, whereas query $\textsc{p}_2$ will evaluate to a very dense result set of w^2 paths. We measured the running time of MinimizeSetGG for both queries. The results of the experiment can be found in Figure 3(b). 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 $\textsc{p}_1$, a bounded query yielding a small result set, finished within a second on all graphs. On the other hand, query evaluation for $\textsc{p}_2$, an unbounded query yielding large result sets, produced a result set $22 \cdot 10^6$ paths on the larges graph and did so in 22 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.

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., [17]): for grammars that are deterministic and unambiguous, 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 cycle graphs. On these cycle 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 3(c). 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. Still, we believe that further optimizations for deterministic and unambiguous grammars are possible, a direction we leave open for future work.

5 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, we give a brief overview.

As stated before, path-based semantics have only gained limited attention. For the regular expressions, Barceló et al. [6] 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. [21] 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. In practical graph database systems, path-based results can already be used in some limited settings [2]. E.g., SPARQL can return RDF graphs via `CONSTRUCT` queries [19], 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 [29]. In the setting of model checking using CTL [12], 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. [7] 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.

Finally, 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. [27] 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. [13,15,14,26], provide ways to enumerate strings in a grammar, but these algorithms cannot effectively be used to quickly find the shortest such string.

6 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 initial 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. Based on our initial 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 many practical applications, the end-user is

only interested in a part of the graph, e.g., paths that originate or end at a certain node. 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 [17]). 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. Alon, U.: An Introduction to Systems Biology: Design Principles of Biological Circuits. Chapman and Hall/CRC (2006)
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017)
3. Arenas, M., Conca, S., Pérez, J.: 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. pp. 629–638. ACM (2012)
4. Bar-Hillel, Y., Perles, M.A., Shamir, E.: On formal properties of simple phrase structure grammars. Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung **14**, 143–172 (1961)
5. Barceló, P.: Querying graph databases. In: Proceedings of the 32nd Symposium on Principles of Database Systems. pp. 175–188. ACM (2013)
6. Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. **37**(4), 31:1–31:46 (2012)
7. Barrett, C., Jacob, R., Marathe, M.: Formal-language-constrained path problems. SIAM J. Comput. **30**(3), 809–837 (2000)
8. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML path language (XPath) 2.0 (second edition). Tech. rep., W3C (2010), http://www.w3.org/TR/2010/REC-xpath20-20101214/
9. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible markup language (XML) 1.1 (second edition). Tech. rep., W3C (2006), http://www.w3.org/TR/2006/REC-xml11-20060816
10. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: Why, how, and where. Foundations and Trends in Databases **1**(4), 379–474 (2009)
11. Clark, J., DeRose, S.: XML path language (XPath) version 1.0. Tech. rep., W3C (1999), http://www.w3.org/TR/1999/REC-xpath-19991116/
12. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
13. Dömösi, P.: Unusual algorithms for lexicographical enumeration. Acta Cybern. **14**(3), 461–468 (2000)

14. Dong, Y.: Linear algorithm for lexicographic enumeration of CFG parse trees. Science in China Series F: Information Sciences **52**(7), 1177–1202 (2009)
15. Florêncio, C.C., Daenen, J., Ramon, J., den Bussche, J.V., Dyck, D.V.: Naive infinite enumeration of context-free languages in incremental polynomial time. J. Univers. Comput. Sci. **21**(7), 891–911 (2015)
16. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
17. Grune, D., Jacobs, C.J.: Parsing Techniques: A Practical Guide. Springer-Verlag New York, 2nd edn. (2008)
18. Harel, D., Pnueli, A., Stavi, J.: Propositional dynamic logic of nonregular programs. J. Comput. Syst. Sci. **26**(2), 222–243 (1983)
19. Harris, S., Seaborne, A.: SPARQL 1.1 query language. Tech. rep., W3C (2013), http://www.w3.org/TR/2013/REC-sparql11-query-20130321
20. Hellings, J.: Conjunctive context-free path queries. In: Proceedings of the 17th International Conference on Database Theory (ICDT 2014). pp. 119–130 (2014)
21. Hofman, P., Martens, W.: Separability by short subsequences and subwords. In: 18th International Conference on Database Theory. Leibniz International Proceedings in Informatics (LIPIcs), vol. 31, pp. 230–246. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015)
22. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
23. Lange, M.: Model checking propositional dynamic logic with all extras. J. Appl. Log. **4**(1), 39–49 (2006)
24. Libkin, L., Martens, W., Vrgoč, D.: Querying graph databases with XPath. In: Proceedings of the 16th International Conference on Database Theory. pp. 129–140. ACM (2013)
25. Losemann, K., Martens, W.: The complexity of evaluating path expressions in SPARQL. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 101–112. ACM (2012)
26. Mäkinen, E.: On lexicographic enumeration of regular and context-free languages. Acta Cybern. **13**(1), 55–61 (1997)
27. Mclean, M.J., Johnston, D.B.: An algorithm for finding the shortest terminal strings which can be produced from non-terminals in context-free grammars. In: Combinatorial Mathematics III, Lecture Notes in Mathematics, vol. 452, pp. 180–196. Springer Berlin Heidelberg (1975)
28. Robinson, I., Webber, J., Eifrem, E.: Graph Databases: New Opportunities for Connected Data. O'Reilly Media, Inc., 2nd edn. (2015)
29. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages. pp. 1–10. ACM, New York, NY, USA (2015)
30. Schreiber, G., Raimond, Y.: RDF 1.1 primer. Tech. rep., W3C (2014), http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624
31. Sevon, P., Eronen, L.: Subgraph queries by context-free grammars. J. Integr. Bioinform. **5**(2), 157–172 (2008)