Path Results for Context-free Grammar Queries on Graphs (extended abstract)

Jelle Hellings
Hasselt University and Transnational University of Limburg
Martelarenlaan 42, 3500 Hasselt, Belgium
jelle.hellings@uhasselt.be

February 8, 2015

Abstract

Graph query languages often use regular expressions for navigation. Traditionally, these regular expressions evaluate to *binary relations* on nodes, where each node pair in the resulting relation is connected by a path whose labeling is accepted by the regular expression. Recent work generalized this usage of regular expressions to context-free grammars, while keeping efficient query evaluation algorithms under the traditional *relational* semantics.

We believe that the relational semantics is limiting: node relations only indicate that nodes in the graph are connected in some way, this without telling how these connections can be established and, hence, only providing limited insight in the structure of the graph. To address the limits of the relational semantics, we propose two *path-based* semantics for evaluating context-free grammars on graphs: evaluating context-free grammars to the *set of all paths* whose labeling is accepted by the context-free grammar, and to a *single path* whose labeling is accepted by the context-free grammar. For both path-based semantics we introduce query evaluation algorithms.

The algorithm we propose for the single path semantics guarantees that this path is the *shortest possible path*. As this shortest path algorithm places high demands on the hardware, we also propose a less demanding approximation algorithm that only guarantees that the length of the produced path is upper bounded. All proposed algorithms have polynomial complexity in terms of the size of the context-free grammar and the graph. We show practical viability of the algorithms via performance measurements on an implementation.

1 Introduction

Graph query languages usually have navigation-based operations at their core [12, 13, 21, 24]. In many cases, these navigational capabilities are provided by *regular expressions* over edge labels, whereby these regular expressions allow the specification of the edge-labeling of paths of interest. Well-known examples of query languages that use regular expressions for navigation are the *(conjunctive) regular path queries* [5, 12, 14]. In the search for greater expressive power, many extensions to the basic regular expressions have been proposed. These include *nested regular expressions* [6], which add conditionals, and *relational and rational relations* [4, 5], which add comparisons of (the edge-labeling of) paths. In practice, regular expressions, together with these extensions, cover a large part of the navigational capabilities of graph query languages such as XPath [7, 8] and SPARQL [25].

An alternative way to enhance the expressive power of query languages is by replacing the regular expressions with more powerful formal languages. Hellings [17] proposes such an approach by studying *context-free grammars* as the navigational component of query languages. The main difference between regular expressions and context-free grammars is the way in which they handle repetition: regular expressions only provide iteration while context-free grammars provide a limited form of recursion. The well-known *same generations*-query [1] is an example of a *context-free graph query* that depends on the recursive capabilities of context-free grammars:

Example 1. The context-free grammar

$$\begin{aligned} \mathtt{a} &\to \mathit{parentOf}\,\mathtt{a}\,\mathit{childOf}, \\ \mathtt{a} &\to \lambda, \end{aligned}$$

interpreted as a graph query on a social graph, evaluates to those people that are n-th generation descendants of a common ancestor.

Context-free grammars are not only more expressive than regular expressions, they can also be interpreted as a set of simple, yet powerful, Datalog rules [1] that can be written unambiguously without using variables. Other applications of context-free grammar-based query languages can be found for model checking [18] and for bio-informatics [23].

Traditionally, query languages using *context-free grammars* evaluate these context-free grammars on graphs to binary relations on nodes, where each node pair in the resulting node relation is connected by a path whose edge-labeling is accepted by the context-free grammar. We believe that this *relational* semantics limits the understanding of the graph data: node relations only indicate that nodes in the graph are connected in some way. To provide more insight in the structure of the graph, we propose stronger *path-based semantics*: answering queries with the *paths* whose edge-labeling matches the edge-labeling specified by the context-free grammars. These paths provide additional insight in the data being queried as they can help answer the question on *how nodes are connected*. When context-free grammars are used as a Datalog variant or when used in a model-checking context [18], these paths also provide proofs on why the expressed property does or does not hold.

For regular expressions, limited sketches of query evaluation algorithms with *path-based semantics* have been provided [5]. To the best of our knowledge, no such results are known for context-free grammar-based query languages. In this work, we study two path-based semantics: the *all-paths semantics*, whereby we answer a query with all paths matching the query, and the *single-path semantics*, whereby we answer a query with a single path. For the all-paths semantics we show that we can represent this, possible infinite, set of paths by a finite *graph-annotated* context-free grammar which can be constructed in $\mathcal{O}(nv + ne + (nv)^3)$, with n the number of non-terminals in the original context-free grammar, v the number of nodes in the original graph, and e the number of edges in the original graph.

For the single-path semantics we consider returning the *shortest such path*. These shortest paths can, for example, be used to calculate the *Erdős numbers*[1] of researchers in a social graph. When context-free grammars are used in a model-checking context [18], these paths provide a shortest proof on why the expressed property holds. We show that these shortest paths can be constructed in $\mathcal{O}(nv + ne + n^4v^5 + p)$, with n the number of non-terminals in the original context-free grammar, v the number of nodes in the original graph, e the number of edges in the original graph, and p the length of all the shortest paths produced. As a byproduct, we show that the shortest strings of a context-free grammar can be constructed in $\mathcal{O}((n\log(n) + p)n + s)$, with n the number of non-terminals in the context-free grammar, p the number of production rules, and s the length of all the shortest strings produced.

The provided algorithm for answering queries with the *shortest paths* places high demands on the hardware. As an alternative, we propose to answer queries with so-called *'acyclic' paths*. These 'acyclic' paths have less strict limits on their lengths, while still guaranteeing upper-bounded lengths. This allows for less-demanding query evaluation algorithms. We show that these 'acyclic' paths can be constructed in $\mathcal{O}(nv + ne + (nv)^3 + p)$, with n the number of non-terminals in the original grammar, v the number of nodes in the original graph, e the number of edges in the original graph, and p the length of all the 'acyclic' paths produced.

Besides proposing the path-based semantics and query evaluation algorithms for these semantics, we show practical viability of the algorithms by providing performance measurements on a

[1] See `http://en.wikipedia.org/wiki/erdos_number`.

C++ implementation. Our initial results show that the algorithms are usable in practice, while also indicating that the performance can benefit greatly from implementation decisions that result from prior knowledge of the graph data and the context-free grammars.

Organization

In Section 2, we present the necessary preliminaries used throughout this paper. In Section 3, we introduce path-based semantics of querying graphs using context-free grammars and introduce query evaluation algorithms for each of the introduced semantics. In Section 4 we present our results on implementations of the introduced query evaluation algorithms. In Section 5, we summarize our findings and discuss directions for future work.

2 Preliminaries

We assume a finite set of symbols Σ which we call the alphabet.

Definition 1. An *edge-labeled directed graph* (or *graph*) is a 3-tuple $G = (\Sigma, V, E)$, where Σ is the alphabet, V is a finite set of node labels, and $E \subseteq V \times \Sigma \times V$ is the labeled edge relation.

A *path* $\pi = n_1 e_1 \dots n_{i-1} e_{i-1} n_i$ in graph $G = (\Sigma, V, E)$, with $n_1, \dots, n_i \in V$ and $e_1, \dots, e_{i-1} \in E$, is a non-empty finite sequence that satisfies, for all $1 \leq j \leq i-1$, $e_j = (n_j, l_j, n_{j+1})$. The *length* of path π is defined by $|\pi| = i - 1$.[2] We write $n_1 \pi n_i$ to indicate that path π starts at node n_1 and ends at node n_i. The *trace* of path π is defined by $\text{trace}(\pi) = l_1 \dots l_i$. Observe that traces are strings over the alphabet Σ and that the trace of a path of length zero, a single node, is the empty string. We usually denote the empty string by λ.

By choosing nodes m and n as the initial and final state we can interpret graphs as *non-deterministic finite automata*. Using this interpretation, we define the language of graph G with respect to nodes m, n, denoted by $\mathcal{L}(G, m, n)$, as $\{\text{trace}(\pi) \mid m\pi n \textit{ is a path in } G\}$.

Given two strings $s = \sigma_1 \dots \sigma_i$ and $s' = \sigma'_1 \dots \sigma'_j$, we define the concatenation of these strings, denoted by $s \cdot s'$, as $\sigma_1 \dots \sigma_i \sigma'_1 \dots \sigma'_j$. Given two two paths $\pi = n_1 \dots n_{i-1} e_{i-1} n_i$ and $\pi' = n'_1 \dots n'_j$ with $n_i = n'_1$, we define the concatenation of these paths, denoted by $\pi \cdot \pi'$, as $n_1 \dots n_{i-1} e_{i-1} n'_1 \dots n'_j$.

Definition 2. A *context-free grammar* is a 3-tuple $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ where Σ is the alphabet, $\mathbf{N}$ is a finite set of non-terminals, and $\mathbf{P}$ is a finite set of production rules. Each production rule is of the form $\mathtt{a} \rightarrow \tau_1 \dots \tau_i$ where $\tau_1 \dots \tau_i$ is a string over $\Sigma \cup \mathbf{N}$.[3] We define the head of a production rule $(\mathtt{a} \rightarrow \tau_1 \dots \tau_i) \in \mathbf{P}$ by $\text{head}(\mathtt{a} \rightarrow \tau_1 \dots \tau_i) = \mathtt{a}$ and we write $\mathtt{a} \rightarrow \lambda$ when $\tau_1 \dots \tau_i$ is the empty string.

The production rules of a context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ describe rewrites of strings over $\Sigma \cup \mathbf{N}$ allowed by the context-free grammar: if we have a string $s = \sigma_1 \dots \sigma_j$, with $\sigma_k = \mathtt{a}$, for some $1 \leq k \leq j$ and $\mathtt{a} \in \mathbf{N}$, and there exists a production rule $\mathtt{a} \rightarrow \tau_1 \dots \tau_i$, then we can rewrite s into $\sigma_1 \dots \sigma_{k-1} \tau_1 \dots \tau_i \sigma_{k+1} \dots \sigma_j$. A string s derives string s' using production rules $\mathbf{P}$, denoted by $s \Longrightarrow_{\mathbf{P}} s'$, whenever s can be rewritten into s' by a finite number of rewrites using the production rules $\mathbf{P}$. We define the language of context-free grammar $\mathbf{C}$ with respect to non-terminal $\mathtt{a} \in \mathbf{N}$, denoted by $\mathcal{L}(\mathbf{C}, \mathtt{a})$, as $\mathcal{L}(\mathbf{C}, \mathtt{a}) = \{s \text{ a finite string over } \Sigma \mid \mathtt{a} \Longrightarrow_{\mathbf{P}} s\}$.

To simplify working with context-free grammars, we assume that all productions are of the form $\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c}$, $\mathtt{a} \rightarrow \sigma$, or $\mathtt{a} \rightarrow \lambda$, with $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathbf{N}$ and $\sigma \in \Sigma$. If these conditions hold, then the context-free grammar is said to be in *normal form*.[4] For context-free grammars $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$

[2] Usually, the length of a path is the length of the sequence of nodes. This work, however, focuses mainly on the labeling of the edges in paths, and, hence, we find it more practical to define the length of the path in terms of the sequence of edges.

[3] Usually, context-free grammars also have a designated start non-terminal $\mathtt{a} \in \mathbf{N}$. For our purposes, however, start non-terminals are not necessary.

[4] This is a less strict normal form than Chomsky's normal form [19]: we do not require that there is a start non-terminal $\mathtt{a}$ such that only $\mathtt{a} \Longrightarrow_{\mathbf{P}} \lambda$. For our purposes, however, this normal form suffices.

in normal form we usually specify the set of productions as the triple $(\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda)$, representing the productions of the form $\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c}$, $\mathtt{a} \rightarrow \sigma$, and $\mathtt{a} \rightarrow \lambda$, respectively. Using standard context-free grammar manipulation techniques [19] one can show that each context-free grammar can be rewritten into this normal form in polynomial time with respect to the size of the original context-free grammar.

3 Context-free Graph Queries

Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar and let $G = (\Sigma, V, E)$ be a graph. A *context-free graph query* is of the form $\mathtt{a}(G)$, with $\mathtt{a} \in \mathbf{N}$ a non-terminal. There are several possible semantics for context-free graph queries:

1. The *Boolean query semantics*, whereby $\mathtt{a}(G)$ evaluates to *true* if and only if there exists a path π in G with $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$.

2. The *relational query semantics*, whereby $\mathtt{a}(G)$ evaluates to the set of node pairs (m, n) for which we have $\exists \pi\, (m \pi n \wedge \text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a}))$.

3. The *all-paths semantics*, whereby $\mathtt{a}(G)$ evaluates to the set of paths π in G for which we have $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$.

4. The *single-path semantics*, whereby $\mathtt{a}(G)$ evaluates, for every pair of nodes m and n for which a path $m\pi' n$ in G with $\text{trace}(\pi') \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ exists, to a single path $m \pi n$ in G for which we have $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$.

Hellings [17] already provided algorithms for the efficient evaluation of con-text-free graph queries using the Boolean and relational semantics.

Proposition 1 (Hellings)**.** *Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar and let $G = (\Sigma, V, E)$ be a graph. All Boolean and relational context-free graph queries can be answered, simultaneously, in $\mathcal{O}(\mathbf{N}E + (\mathbf{N}V)^3)$.*[5]

In the following, we develop algorithms for query evaluation using the all-paths and single-path semantics.

3.1 The All-Paths Semantics

A context-free graph query evaluated over a cyclic graph can result in an infinite set of paths. Hence, the first challenge for query evaluation using the all-paths semantics is to represent the set of all paths in a finite structure.

We have already argued that we can interpret graphs as non-deterministic finite automata. It is well-known that the intersection of regular languages (as represented by non-deterministic automata) and context-free languages is itself a context-free language [3], and, thus, can be represented by a context-free grammar:

Proposition 2 (Bar-Hillel et al.)**.** *Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar, let $G = (\Sigma, V, E)$ be a graph, let $\mathtt{a} \in \mathbf{N}$ be a non-terminal, and let $m, n \in V$ be nodes. The set of strings $\mathcal{L}(G, m, n) \cap \mathcal{L}(\mathbf{C}, \mathtt{a})$ is expressible by a context-free grammar.*

The above guarantees that there is a finite representation of all traces of the paths in a graph matching a context-free graph query. There is, however, not a one-to-one mapping between traces and paths and, furthermore, we do not yet have means to construct this finite representation. To allow for a direct mapping of traces to paths, we propose to construct a context-free grammar whose non-terminals are annotated with node information.

[5] In $\mathcal{O}$-notations we use the notation S, for S a set, to indicate the size of set S, this instead of $|S|$.

Definition 3. The *graph-annotated grammar* of context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ and graph $G = (\Sigma, V, E)$ is a context-free grammar $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, \mathbf{P}_{[G]})$ satisfying the following properties (with $\mathbf{P} = (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda)$ and $\mathbf{P}_{[G]} = (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda)$):

1. Non-terminals in $\mathbf{N}_{[G]}$ are of the form $\mathtt{a}[m, n]$ with $\mathtt{a} \in \mathbf{N}$ and $m, n \in V$. We have $\mathtt{a}[m, n] \in \mathbf{N}_{[G]}$ if and only if $\mathcal{L}(G, m, n) \cap \mathcal{L}(\mathbf{C}, \mathtt{a}) \neq \emptyset$.
2. $(\mathtt{a}[m, n] \to \mathtt{b}[m, x_1]\, \mathtt{c}[x_2, n]) \in \mathbf{P}'_2$ if and only if $x_1 = x_2$, $\mathtt{b}[m, x_1], \mathtt{c}[x_2, n] \in \mathbf{N}_{[G]}$, and $(\mathtt{a} \to \mathtt{b}\,\mathtt{c}) \in \mathbf{P}_2$.
3. $(\mathtt{a}[m, n] \to \sigma) \in \mathbf{P}'_\Sigma$ if and only if $(m, \sigma, n) \in E$ and $(\mathtt{a} \to \sigma) \in \mathbf{P}_\Sigma$.
4. $(\mathtt{a}[m, n] \to \lambda) \in \mathbf{P}'_\lambda$ if and only if $m = n$ and $(\mathtt{a} \to \lambda) \in \mathbf{P}_\lambda$.

We say that a non-terminal $\mathtt{a}[m, n] \in \mathbf{N}_{[G]}$ can *derive* path $\pi = n_1 l_1 \ldots l_i n_{i+1}$ if it can derive the string $s = l_1 \ldots l_i$ such that, for each $1 \leq j \leq i$, the rewrite step producing l_j used a production rule of the form $(\mathtt{b}[n_j, n_{j+1}] \to l_j) \in \mathbf{P}'_\Sigma$.

Usually, we refer to the non-terminals in $\mathbf{N}_{[G]}$ as graph-annotated non-terminals and to the production rules in $\mathbf{P}_{[G]}$ as graph-annotated production rules.

Conceptually, the annotations in a graph-annotated grammar describe the mapping between the trace of a path, the first node on the path, and the last node on the path. Thereby the production rules in $\mathbf{P}'_\lambda$ describe paths of length zero (nodes), the production rules in $\mathbf{P}'_\Sigma$ describes paths of length one (edges), and the production rules in $\mathbf{P}'_2$ describes how to concatenate paths to form larger paths.

Lemma 1. *Let $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ be a context-free grammar, let $G = (\Sigma, V, E)$ be a graph, let $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda))$ be the graph-annotated grammar of $\mathbf{C}$ and G, let $m\pi n$ be a path in G, and let $\mathtt{a} \in \mathbf{N}$ be a non-terminal. We have $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ if and only if we can derive π from $\mathtt{a}[m, n] \in \mathbf{N}_{[G]}$.*

We illustrate the concept of a graph-annotated grammar with a small example:

Example 2. We have a small social network, as visualized in Figure 1, and Alice wants to know how she can contact Eve via friends, via friends of friends, and so on. Hence, she writes the context-free grammar with the following production rules:

$$\mathtt{c} \to \textit{friendOf}$$
$$\mathtt{c} \to \mathtt{c}\,\mathtt{c}$$

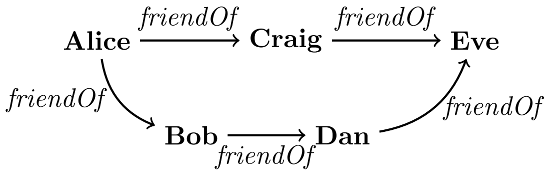

Figure 1: An example of an edge-labeled graph: a social network whereby persons (represented by nodes) can have *friendOf*-relations (represented by labeled edges).

The graph-annotated grammar of the context-free grammar and the graph visualized in Figure 1 has the following non-terminals:

$$\mathtt{c}[\text{Alice}, \text{Bob}], \mathtt{c}[\text{Alice}, \text{Craig}], \mathtt{c}[\text{Alice}, \text{Dan}],$$
$$\mathtt{c}[\text{Alice}, \text{Eve}], \mathtt{c}[\text{Bob}, \text{Dan}], \mathtt{c}[\text{Bob}, \text{Eve}],$$
$$\mathtt{c}[\text{Craig}, \text{Eve}], \mathtt{c}[\text{Dan}, \text{Eve}].$$

The production rules of the graph-annotated grammar consists of the production rules that correspond to *friendOf*-edges in the social network:

$$\begin{aligned}
\mathtt{c}[\text{Alice}, \text{Bob}] &\to \textit{friendOf}, & \mathtt{c}[\text{Alice}, \text{Craig}] &\to \textit{friendOf},\\
\mathtt{c}[\text{Bob}, \text{Dan}] &\to \textit{friendOf}, & \mathtt{c}[\text{Craig}, \text{Eve}] &\to \textit{friendOf},\\
\mathtt{c}[\text{Dan}, \text{Eve}] &\to \textit{friendOf}.
\end{aligned}$$

Furthermore, the following production rules of the graph-annotated grammar express the combination of paths in the social network to form bigger paths:

$$\begin{aligned}
\mathtt{c}[\text{Alice}, \text{Dan}] &\to \mathtt{c}[\text{Alice}, \text{Bob}]\ \mathtt{c}[\text{Bob}, \text{Dan}],\\
\mathtt{c}[\text{Alice}, \text{Eve}] &\to \mathtt{c}[\text{Alice}, \text{Bob}]\ \mathtt{c}[\text{Bob}, \text{Eve}],\\
\mathtt{c}[\text{Alice}, \text{Eve}] &\to \mathtt{c}[\text{Alice}, \text{Craig}]\ \mathtt{c}[\text{Craig}, \text{Eve}],\\
\mathtt{c}[\text{Alice}, \text{Eve}] &\to \mathtt{c}[\text{Alice}, \text{Dan}]\ \mathtt{c}[\text{Dan}, \text{Eve}],\\
\mathtt{c}[\text{Bob}, \text{Eve}] &\to \mathtt{c}[\text{Bob}, \text{Dan}]\ \mathtt{c}[\text{Dan}, \text{Eve}].
\end{aligned}$$

To produce a path from Alice to Eve, we can use this graph-annotated grammar:

$$\begin{aligned}
&\mathtt{c}[\text{Alice}, \text{Eve}]\\
\Longrightarrow&\{\text{Rewrite } \mathtt{c}[\text{Alice}, \text{Eve}] \to \mathtt{c}[\text{Alice}, \text{Bob}]\ \mathtt{c}[\text{Bob}, \text{Eve}]\}\\
&\mathtt{c}[\text{Alice}, \text{Bob}]\ \mathtt{c}[\text{Bob}, \text{Eve}]\\
\Longrightarrow&\{\text{Rewrite } \mathtt{c}[\text{Bob}, \text{Eve}] \to \mathtt{c}[\text{Bob}, \text{Dan}]\ \mathtt{c}[\text{Dan}, \text{Eve}]\}\\
&\mathtt{c}[\text{Alice}, \text{Bob}]\ \mathtt{c}[\text{Bob}, \text{Dan}]\ \mathtt{c}[\text{Dan}, \text{Eve}]\\
\Longrightarrow&\{\text{Rewrite } \mathtt{c}[\text{Alice}, \text{Bob}] \to \textit{friendOf}, \dots\}\\
&\textit{friendOf}\,\textit{friendOf}\,\textit{friendOf}.
\end{aligned}$$

Hence, there is a path from Alice to Eve of length three via Bob and Dan satisfying the conditions imposed by the context-free grammar.

As graph-annotated grammars are context-free grammars, one can use existing context-free enumeration techniques [10, 11, 20] to produce some of the paths represented by the graph-annotated grammar. The efficiency of these techniques depend on the size of the graph-annotated grammar. We observe the following worst-case bounds:

Lemma 2. *Let* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *be a context-free grammar, let* $G = (\Sigma, V, E)$ *be a graph, and let* $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda))$ *be the graph-annotated grammar of* $\mathbf{C}$ *and* G. *We have* $|\mathbf{N}_{[G]}| \leq |\mathbf{N}||V|^2$, $|\mathbf{P}'_2| \leq |\mathbf{P}_2||V|^3 \leq (|\mathbf{N}||V|)^3$, $|\mathbf{P}'_\Sigma| \leq |\mathbf{N}||E|$ *and* $|\mathbf{P}'_\Sigma| \leq |\mathbf{P}_\Sigma||V|^2$, *and* $|\mathbf{P}'_\lambda| = |\mathbf{P}_\lambda||V|$.

We prove that graph-annotated grammars exist for any combination of a context-free grammar and a graph by providing Algorithm 1 for the construction of these graph-anno-tated grammars.

Theorem 1. *Let* $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ *be a context-free grammar and let* $G = (\Sigma, V, E)$ *be a graph. The context-free grammar constructed by the invocation of Algorithm 1 is the graph-annotated grammar* $\mathbf{C}_{[G]}$.

Proof (sketch). We say that a derivation of path $m\pi n$ from $\mathtt{a}[m, n]$ depends on $\mathtt{d}[x, z]$ if, in a derivation $\mathtt{a} \Longrightarrow_{\mathbf{P}} \text{trace}(\pi)$, we have a step $\mathtt{a} \Longrightarrow_{\mathbf{P}} \tau_1 \dots \tau_i \mathtt{d} \tau_{i+1} \dots \tau_j$, with $\tau_1 \dots \tau_i$ and $\tau_{i+1} \dots \tau_j$ strings over $\Sigma \cup \mathbf{N}$, such that $\mathtt{d} \Longrightarrow_{\mathbf{P}} \text{trace}(x\pi' y)$, with π' a subsequence of π.

When arriving at the *while*-loop, Definition 3.3 and Definition 3.4 already hold. Termination of the *while*-loop will also guarantee that Definition 3.1 and Definition 3.2 hold as the *while*-loop maintains the following invariants:

Algorithm 1 CONSTRUCT-$\mathbf{C}_{[G]}$

```
Input: context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ with $\mathbf{P} = (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda)$ and graph $G = (\Sigma, V, E)$.
Output: the graph-annotated grammar $\mathbf{C}_{[G]}$.
$\mathbf{N}_{[G]}, \mathbf{P}_{[G]} := \emptyset, \emptyset$
for all $(\mathtt{a} \rightarrow \lambda) \in \mathbf{P}_\lambda$ and $n \in V$ do
    $\mathbf{N}_{[G]}, \mathbf{P}_{[G]} := \mathbf{N}_{[G]} \cup \{\mathtt{a}[n,n]\}, \mathbf{P}_{[G]} \cup \{(\mathtt{a}[n,n] \rightarrow \lambda)\}$
end for
for all $(\mathtt{a} \rightarrow \sigma) \in \mathbf{P}_\Sigma$ and $(m, \sigma, n) \in E$ do
    $\mathbf{N}_{[G]}, \mathbf{P}_{[G]} := \mathbf{N}_{[G]} \cup \{\mathtt{a}[m,n]\}, \mathbf{P}_{[G]} \cup \{(\mathtt{a}[m,n] \rightarrow \sigma)\}$
end for
$new := \mathbf{N}_{[G]}$
while $new \neq \emptyset$ do
    take and remove a $\mathtt{a}[m,n]$ from $new$
    for all $(\mathtt{c} \rightarrow \mathtt{a}\,\mathtt{b}) \in \mathbf{P}_2$ and $\mathtt{b}[n,o] \in \mathbf{N}_{[G]}$ do
        PRODUCE($\mathtt{c}[m,o] \rightarrow \mathtt{a}[m,n]\,\mathtt{b}[n,o]$)
    end for
    for all $(\mathtt{c} \rightarrow \mathtt{b}\,\mathtt{a}) \in \mathbf{P}_2$ and $\mathtt{b}[o,m] \in \mathbf{N}_{[G]}$ do
        PRODUCE($\mathtt{c}[o,n] \rightarrow \mathtt{b}[o,m]\,\mathtt{a}[m,n]$)
    end for
end while
return $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, \mathbf{P}_{[G]})$

Procedure PRODUCE($p = (\mathtt{d}[x,z] \rightarrow \mathtt{e}[x,y]\,\mathtt{f}[y,z])$):
$\mathbf{P}_{[G]} := \mathbf{P}_{[G]} \cup \{p\}$
if $\mathtt{d}[x,z] \notin \mathbf{N}_{[G]}$ then
    $new, \mathbf{N}_{[G]} := new \cup \{\mathtt{d}[x,z]\}, \mathbf{N}_{[G]} \cup \{\mathtt{d}[x,z]\}$
end if
```

(i) If there is a path $m\pi n$ in G and non-terminal $\mathtt{a} \in \mathbf{N}$ such that $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$, then we can either already derive π from $\mathtt{a}[m,n] \in \mathbf{N}_{[G]}$ or the derivation depends on another non-terminal $\mathtt{d}[x,z] \in$ *new*.

(ii) If we can derive path $m\pi n$ from $\mathtt{a}[m,n] \in \mathbf{N}_{[G]}$, then $\mathtt{a} \Longrightarrow_{\mathbf{P}} \text{trace}(\pi)$.

(iii) If $(\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c}) \in \mathbf{P}_2$ and $\mathtt{b}[m,x_1], \mathtt{c}[x_2,n] \in \mathbf{N}_{[G]} \setminus$ *new*, then $(\mathtt{a}[m,n] \rightarrow \mathtt{b}[m,x_1]\,\mathtt{c}[x_2,n]) \in \mathbf{P}_{[G]}$.

The only-if direction of Definition 3.2 follows directly from the algorithm. Algorithm 1 terminates as each possible non-terminal of the form $\mathtt{a}[m,n]$ is added to *new* at most once. □

Next we shall show that Algorithm 1 has polynomial running time, both in the terms of the context-free grammar and the graph.

Proposition 3. *The worst-case running time of Algorithm 1 on context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *and graph* $G = (\Sigma, V, E)$*, producing graph-annotated grammar* $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, \mathbf{P}_{[G]} = (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda))$*, is* $\mathcal{O}(\mathbf{P}'_\lambda + \mathbf{P}'_\Sigma + E + (\mathbf{N}V)^3)$ *using* $\mathcal{O}(\mathbf{C} + (\mathbf{N}V)^3)$ *memory and* $\mathcal{O}(\text{SCAN}(E))$ *IOs.*

Proof. We represent the context-free grammar $\mathbf{C}$ such that we can easily iterate over productions rules and/or check if production rules exist in constant time per production rule. We read the graph as a list of edges and we do so once. The production rules $\mathbf{P}'_\Sigma$ and $\mathbf{P}'_\lambda$ are produced exactly once, hence, we write them directly to a list. The production rules $\mathbf{P}'_2$ are each produced at most twice, hence, to assure we do not produce duplicates, we represent these production rules by a $(|\mathbf{N}||V|)^3$ Boolean matrix. The total cost for creating and maintaining these structures, representing the production rules of the graph-annotated grammar, is thus $\mathcal{O}(\mathbf{P}'_\Sigma + \mathbf{P}'_\lambda + (\mathbf{N}V)^3)$.

Variable *new* is a queue and each non-terminal from $\mathbf{N}_{[G]}$ will be added to and removed from this queue exactly once. To be able to check if a non-terminals from $\mathbf{N}_{[G]}$ exists, we use a $|\mathbf{N}||V|^2$ Boolean matrix. □

Proposition 3 uses a Boolean matrix to represent $\mathbf{P}'_2$. This choice guarantees strong worst-case upper bounds on the running time, this independent of the number of graph-annotated production rules that are actually placed in $\mathbf{P}'_2$. The consequences of this representation are high memory consumption and high construction costs for the Boolean matrix, even if the produced set $\mathbf{P}'_2$ has *low density*. Thereby the density of $\mathbf{P}'_2$ is an indication of the size of $\mathbf{P}'_2$ with respect to the worst-case size $(|\mathbf{N}||V|)^3$ provided by Lemma 2. We expect to see that when $\mathbf{P}'_2$ has *high density*, the Boolean matrix representation is preferable, while the construction of this Boolean matrix representation is an unnecessary waste of resources and time when $\mathbf{P}'_2$ has *low density*. Alternatively, we can choose for a representation that is expected to perform better when $\mathbf{P}'_2$ has low density:

Proposition 4. *The worst-case running time of Algorithm 1 on context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *and graph* $G = (\Sigma, V, E)$*, producing graph-annotated grammar* $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, \mathbf{P}_{[G]} = (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda))$*, is* $\mathcal{O}(\mathbf{P}'_\lambda + \mathbf{P}'_\Sigma + \mathbf{N}\,V^2 + \mathbf{P}_2\,V^3 + \mathbf{P}'_2 \log(\mathbf{P}'_2))$ *using* $\mathcal{O}(\mathbf{C}) + \mathbf{N}\,V^2$ bits *memory and* $\mathcal{O}(\textsc{scan}(E) + \textsc{scan}(\mathbf{N}_{[G]}) + \textsc{scan}(\mathbf{P}_{[G]}) + \textsc{sort}(\mathbf{P}'_2))$ *IOs.*

Proof. We write production rules directly to disk. Only the production rules $\mathbf{P}'_2$ can have duplicates, as each of the production rules in $\mathbf{P}'_2$ can be produced twice during execution of Algorithm 1. To remove these duplicates, we sort and scan the list representing $\mathbf{P}'_2$ once. We represent *new* as an external-memory queue. □

Other alternatives to the Boolean matrix representation of $\mathbf{P}'_2$ are set representations, for example by using standard *binary search trees* or *hash-tables*. In Section 4.1 we will investigate the effects of different choices for the representation of $\mathbf{P}'_2$ on the running time of Algorithm 1.

3.2 The Single-Path Semantics

In Section 3.1 we studied the all-paths semantics and showed how to construct graph-annotated grammars that describe all possible paths in a graph matching a context-free language. It is not hard to imagine that having access to a single path is sufficient in many practical situations. As the length of these paths is not necessarily upper bounded, a logical choice would be to choose a path that is as short as possible.

3.2.1 Deriving a Shortest Path

The graph-annotated grammars of Section 3.1 can be used to reduce the problem of finding the shortest path $m\pi n$ for a given context-free graph query $\mathtt{a}(G)$ to the problem of finding the shortest string derivable from $\mathtt{a}[m, n]$ in a graph-annotated grammar. In this context, we formally define shortest paths and shortest strings as follows:

Definition 4. Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar, let $G = (\Sigma, V, E)$ be a graph, let $\mathtt{a} \in \mathbf{N}$, and let $m, n \in V$ be nodes. String $s \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ is *a shortest string* for non-terminal $\mathtt{a}$ if no string $s' \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ with $|s'| < |s|$ exists. We denote the length of a shortest strings for non-terminal $\mathtt{a}$ by $\text{length}_\downarrow(\mathtt{a})$. Path $m\pi n$ in G with $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ is *a shortest path* for non-terminal $\mathtt{a}$ and nodes $m, n \in V$ if no path $m\pi' n$ with $\text{trace}(\pi') \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ and $|\pi'| < |\pi|$ exists.

Mclean et al. [22] already proved that the shortest strings producible by an arbitrary context-free grammar can be computed. Their results do, however, not give complexity results for an actual shortest string derivation algorithm. Towards such a shortest string derivation algorithm, we observe the following:

Lemma 3. *Let* $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ *be a context-free grammar. If a string* s *is a shortest string for non-terminal* $\mathtt{a} \in \mathbf{N}$*, then there is a derivation* $\mathtt{a} \Longrightarrow_{\mathbf{P}} s$ *that uses, per non-terminal* $\mathtt{b}$*, only a single production rule for rewriting* $\mathtt{b}$*.*

Observe that derivations that meet the conditions of Lem-ma 3 are fully deterministic, there are no choices in which rewrite steps one can make to rewrite a specific non-terminal and, hence, each non-terminal will be rewritten into a unique string over Σ. Such derivations leading to a string over Σ are also necessarily *acyclic*: no derivation steps will rewrite non-terminal $\mathsf{a} \in \mathbf{N}$ into a sequence that contains a (as such a derivation cannot terminate). Thus, a step one can take in the derivation of shortest strings is to derive an *acyclic* set of production rules.

Definition 5. Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar such that $\mathcal{L}(\mathbf{C}, \mathsf{a}) \neq \emptyset$ for every $\mathsf{a} \in \mathbf{N}$. Function $\mathbf{A} : \mathbf{N} \to \mathbf{P}$ is a *proper acyclic subset* of $\mathbf{P}$ when it satisfies the following conditions:

1. For each non-terminal $\mathsf{a} \in \mathbf{N}$, we have $\mathrm{head}(\mathbf{A}(\mathsf{a})) = \mathsf{a}$.
2. For each non-terminal $\mathsf{a} \in \mathbf{N}$ there exists a string s over Σ with $\mathsf{a} \Longrightarrow_{\mathrm{image}(\mathbf{A})} s$.[6]
3. The set of production rules $\mathrm{image}(\mathbf{A})$ is *acyclic*: no derivation exists that rewrites non-terminal $\mathsf{a} \in \mathbf{N}$ into a sequence that contains a while using only the production rules mapped to by $\mathbf{A}$.

Function $\mathbf{A}$ is a *minimizing acyclic subset* if, for every non-terminal $\mathsf{a} \in \mathbf{N}$ and string s over Σ with $\mathsf{a} \Longrightarrow_{\mathrm{image}(\mathbf{A})} s$, we have $|s| = \mathrm{length}_{\downarrow}(\mathsf{a})$.

Example 3. We consider the graph-annotated grammar constructed in Example 2. A proper acyclic subset of this graph-annotated grammar would be the following:

$$\begin{aligned}
\mathsf{c}[\text{Alice}, \text{Bob}] &\to \textit{friendOf}, & \mathsf{c}[\text{Alice}, \text{Craig}] &\to \textit{friendOf},\\
\mathsf{c}[\text{Bob}, \text{Dan}] &\to \textit{friendOf}, & \mathsf{c}[\text{Craig}, \text{Eve}] &\to \textit{friendOf},\\
\mathsf{c}[\text{Dan}, \text{Eve}] &\to \textit{friendOf}.
\end{aligned}$$

$$\begin{aligned}
\mathsf{c}[\text{Alice}, \text{Dan}] &\to \mathsf{c}[\text{Alice}, \text{Bob}]\,\mathsf{c}[\text{Bob}, \text{Dan}],\\
\mathsf{c}[\text{Alice}, \text{Eve}] &\to \mathsf{c}[\text{Alice}, \text{Bob}]\,\mathsf{c}[\text{Bob}, \text{Eve}],\\
\mathsf{c}[\text{Bob}, \text{Eve}] &\to \mathsf{c}[\text{Bob}, \text{Dan}]\,\mathsf{c}[\text{Dan}, \text{Eve}].
\end{aligned}$$

This is not a minimizing acyclic subset: the string produced for the non-terminal $\mathsf{c}[\text{Alice}, \text{Eve}]$ has length three, while a shorter string exists (via Craig). By replacing the production rule for $\mathsf{c}[\text{Alice}, \text{Eve}]$ by

$$\mathsf{c}[\text{Alice}, \text{Eve}] \to \mathsf{c}[\text{Alice}, \text{Craig}]\,\mathsf{c}[\text{Craig}, \text{Eve}]$$

we get a minimizing acyclic subset.

Due to the deterministic nature of any minimizing acyclic subset of a context-free grammar, we can use these minimizing acyclic subsets to construct a shortest string for every non-terminal in a context-free grammar. Algorithm 2 provides an efficient way to do so. Observe that Algorithm 2 can also be used to produce a single string by calling DP-TRAVERSE for a single non-terminal.

Proposition 5. *Algorithm 2 is correct and has a worst-case running time of $\mathcal{O}(\mathbf{N} + |S|)$, where $|S|$ is the total length of all produced strings or $\mathcal{O}(\mathbf{N} + |s|)$ when a single string s is produced.*

Proof (sketch). The algorithm follows a straightforward dynamic programming approach to construct all strings for all non-terminals. Outside the main loop, each production rule in $\mathrm{image}(\mathbf{A})$ forces the algorithm to consider each non-terminal at most twice. For producing all strings, we can simply represent strings by arrays. To achieve the claimed running time in terms of $|s|$ while producing a single string, it is necessary to use a representation wherein common substrings are shared. One approach would be to let M map to parts of doubly-linked lists, to concatenate these doubly-linked lists when possible, and to only copy a string s' when s' is repeated several times in the produced string s. □

[6] The *image of function f* with domain $\mathcal{D}$ is defined by $\mathrm{image}(f) = \{f(x) \mid x \in \mathcal{D}\}$.

Algorithm 2 PRODUCE-PATHS

```
Input: context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ and minimizing acyclic subset $\mathbf{A}$ of $\mathbf{P}$.
Output: a mapping from each non-terminal $\mathtt{a}$ to the string $s$ with $\mathtt{a} \Longrightarrow_{\text{image}(\mathbf{A})} s$.
$M := \emptyset$
for all $\mathtt{a} \in \mathbf{N}$ do
    DP-TRAVERSE($M$; $\mathtt{a}$)
end for
return $M$

Procedure DP-TRAVERSE($M$; $\mathtt{d}$):
if $\mathtt{d} \in M$ then
    return
else if $\mathbf{A}(\mathtt{d}) = (\mathtt{d} \to \lambda)$ then
    $M(\mathtt{d}) := \lambda$
else if $\mathbf{A}(\mathtt{d}) = (\mathtt{d} \to \sigma)$ with $\sigma \in \Sigma$ then
    $M(\mathtt{d}) := \sigma$
else
    let $\mathbf{A}(\mathtt{d}) = (\mathtt{d} \to \mathtt{e}\,\mathtt{f})$ with $\mathtt{e}, \mathtt{f} \in \mathbf{N}$
    DP-TRAVERSE($M$; $\mathtt{e}$)
    DP-TRAVERSE($M$; $\mathtt{f}$)
    $M(\mathtt{d}) := M(\mathtt{e}) \cdot M(\mathtt{f})$
end if
```

The worst-case runtime complexity of Algorithm 2 is dominated by the length of the produced strings. The restriction of the set of production rules to a proper acyclic subset allows us to provide the following worst-case upper bounds on the length of the produced strings:

Lemma 4. *Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a grammar and let function $\mathbf{A}$ be a proper acyclic subset of $\mathbf{P}$. The total length of all strings that can be produced using only the production rules* image($\mathbf{A}$) *is upper bounded by $2^{|\mathbf{N}|} - 1$ and the length of the longest such string is upper bounded by $2^{|\mathbf{N}|-1}$. These upper bounds are strict.*

Proof. Let $\mathbf{N} = \{\mathtt{a_1}, \dots, \mathtt{a_i}\}$. To maximize the length of all strings, we choose $\mathbf{A}(\mathtt{a_i}) = (\mathtt{a_i} \to \sigma)$, for some $\sigma \in \Sigma$, and $\mathbf{A}(\mathtt{a_j}) = (\mathtt{a_j} \to \mathtt{a_{j+1}}\,\mathtt{a_{j+1}})$, for all $1 \leq j < i$. Observe that these production rules maximize the number of rewrite steps, for each $1 \leq j \leq i$, to rewrite $\mathtt{a_j}$ into a string over Σ (under the restriction that the production rules are a proper acyclic subset). Using these definitions we have $\text{length}_{\downarrow}(\mathtt{a_j}) = 2^{i-j}$ for each $1 \leq j \leq i$. Hence, the total length of all strings produced by the production rules image($\mathbf{A}$) is

$$\sum_{1 \leq j \leq i} \text{length}_{\downarrow}(a_j) = \sum_{1 \leq j \leq i} 2^{i-j} = 2^i - 1 = 2^{|\mathbf{N}|} - 1$$

and the longest such string has length $2^{|\mathbf{N}|-1}$. □

We can only use Algorithm 2 to produce shortest strings when we can construct minimizing acyclic subset of the production rules of a grammar. We prove that we can construct minimizing acyclic subsets of the production rules of any context-free grammar by providing Algorithm 3.

Theorem 2. *Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar. The function $\mathbf{A}$ constructed by the invocation of Algorithm 3 is a minimizing acyclic subset of $\mathbf{P}$.*

Proof (sketch). Let $\mathtt{a} \in \mathbf{N}$. By $\mathit{ts}(\mathtt{a})$ we denote the second component of the priority of $\mathtt{a}$. The main *while*-loop maintains the following invariants:

(i) The production rules image($\mathbf{A}$) are acyclic: if $\mathtt{a} \in \mathit{cost}$ and $\mathbf{A}(\mathtt{a}) = (\mathtt{a} \to \mathtt{b}\,\mathtt{c})$, then $\langle \mathit{cost}(\mathtt{a}), \mathit{ts}(\mathtt{a}) \rangle > \langle \mathit{cost}(\mathtt{b}), \mathit{ts}(\mathtt{b}) \rangle$ and $\langle \mathit{cost}(\mathtt{a}), \mathit{ts}(\mathtt{a}) \rangle > \langle \mathit{cost}(\mathtt{c}), \mathit{ts}(\mathtt{c}) \rangle$.

Algorithm 3 MINIMIZING-ACYCLIC-SUBSET

```
Input: context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ with, for each $\mathtt{a} \in \mathbf{N}$, $\mathcal{L}(\mathbf{C}, \mathtt{a}) \neq \emptyset$.
Output: a minimizing acyclic subset $\mathbf{A}$ of $\mathbf{P}$.
$\mathit{cost}, \mathbf{A}, \mathit{ts} := \emptyset, \emptyset, 0$
$\mathit{new}$ is a min-priority queue
for all $(\mathtt{a} \rightarrow \lambda) \in \mathbf{P}_\lambda$ do
    $\mathit{cost}(\mathtt{a}), \mathbf{A}(\mathtt{a}) := 0, (\mathtt{a} \rightarrow \lambda)$
    $\mathit{new}, \mathit{ts} := \mathit{new} \cup \{\mathtt{a}$ with priority $\langle 0, \mathit{ts} \rangle\}, \mathit{ts} + 1$
end for
for all $(\mathtt{a} \rightarrow \sigma) \in \mathbf{P}_\Sigma$ do
    if $\mathtt{a} \notin \mathit{cost}$ then
        $\mathit{cost}(\mathtt{a}), \mathbf{A}(\mathtt{a}) := 1, (\mathtt{a} \rightarrow \sigma)$
        $\mathit{new}, \mathit{ts} := \mathit{new} \cup \{\mathtt{a}$ with priority $\langle 1, \mathit{ts} \rangle\}, \mathit{ts} + 1$
    end if
end for
while $\mathit{new} \neq \emptyset$ do
    take and remove $\mathtt{a}$ from $\mathit{new}$
    for all $(\mathtt{c} \rightarrow \mathtt{a}\,\mathtt{b}) \in \mathbf{P}_2$ and $\mathtt{b} \in \mathit{cost}$ do
        PRODUCE$(\mathtt{c} \rightarrow \mathtt{a}\,\mathtt{b})$
    end for
    for all $(\mathtt{c} \rightarrow \mathtt{b}\,\mathtt{a}) \in \mathbf{P}_2$ and $\mathtt{b} \in \mathit{cost}$ do
        PRODUCE$(\mathtt{c} \rightarrow \mathtt{b}\,\mathtt{a})$
    end for
end while
return $\mathbf{A}$

Procedure PRODUCE$(p = (\mathtt{d} \rightarrow \mathtt{e}\,\mathtt{f})$):
$c := \mathit{cost}(\mathtt{e}) + \mathit{cost}(\mathtt{f})$
if $\mathtt{d} \notin \mathit{cost}$ then
    $\mathit{cost}(\mathtt{d}), \mathbf{A}(\mathtt{d}) := c, p$
    $\mathit{new}, \mathit{ts} := \mathit{new} \cup \{\mathtt{d}$ with priority $\langle c, \mathit{ts} \rangle\}, \mathit{ts} + 1$
else if $\mathit{cost}(\mathtt{d}) > c$ then
    $\mathit{cost}(\mathtt{d}), \mathbf{A}(\mathtt{d}) := c, p$
    lower priority of $\mathtt{d}$ in $\mathit{new}$ to $\langle c, \mathit{ts} \rangle$
    $\mathit{ts} := \mathit{ts} + 1$
end if
```

(ii) If $\mathtt{a} \in \mathit{cost}$ and $\mathbf{A}(\mathtt{a}) = (\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c})$, then $\mathit{cost}(\mathtt{a}) \geq \mathit{cost}(\mathtt{b}) + \mathit{cost}(\mathtt{c})$.

(iii) If $\mathtt{a} \in \mathit{cost}$, then there exists a string s over Σ such that $\mathtt{a} \Longrightarrow_{\mathrm{image}(\mathbf{A})} s$ and $|s| \leq \mathit{cost}(\mathtt{a})$.

(iv) If $\mathtt{a} \in \mathit{cost} \setminus \mathit{new}$ and $\mathbf{A}(\mathtt{a}) = (\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c})$, then $\mathtt{b}, \mathtt{c} \in \mathit{cost} \setminus \mathit{new}$.

(v) Let $\mathtt{x}$ be the non-terminal with the min-priority in new. If $\mathrm{length}_\downarrow(\mathtt{a}) < \mathit{cost}(\mathtt{x})$ or $\mathtt{a} \in \mathit{cost} \setminus \mathit{new}$, then $\mathit{cost}(\mathtt{a}) = \mathrm{length}_\downarrow(\mathtt{a})$.

Initially all these invariants hold as no production rule from $\mathbf{P}_2$ has yet been assigned to $\mathbf{A}$. Invariant (i) and Invariant (ii) are maintained as $\mathit{cost}(\mathtt{a}) = \mathit{cost}(\mathtt{b}) + \mathit{cost}(\mathtt{c})$ and $\mathit{ts}(\mathtt{a}) > \max(\mathit{ts}(\mathtt{b}), \mathit{ts}(\mathtt{c}))$ when we assign $\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c}$ to $\mathbf{A}(\mathtt{a})$, the values $\mathit{ts}(\mathtt{b})$ and $\mathit{ts}(\mathtt{c})$ are only increased when $\mathit{cost}(\mathtt{b})$ and $\mathit{cost}(\mathtt{c})$ are decrease, and the values $\mathit{cost}(\mathtt{a})$ and $\mathit{ts}(\mathtt{a})$ are only changed when $\mathbf{A}(\mathtt{a})$ is changed. Assuming Invariant (iii) holds for $\mathtt{b}, \mathtt{c} \in \mathit{cost}$, it will also holds for $\mathtt{a}$ when we assign $\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c}$ to $\mathbf{A}(\mathtt{a})$ as we assign $\mathit{cost}(\mathtt{b}) + \mathit{cost}(\mathtt{c})$ to $\mathit{cost}(\mathtt{a})$ and the string over Σ derivable from $\mathtt{a}$ is the concatenation of the strings over Σ derivable from $\mathtt{b}$ and $\mathtt{c}$. Invariant (iv) follows from Invariant (i) as Invariant (i) guarantees that the priorities of non-terminals $\mathtt{b}$ and $\mathtt{c}$ are less than the priority of non-terminal $\mathtt{a}$.

Assume Invariant (v) holds before an iteration of the main *while*-loop. We prove that the invariant still holds after an iteration of the main *while*-loop that removes non-terminal $\mathtt{a}$ from *new*. Let s be a shortest string for $\mathtt{a}$ and let $p \in \mathbf{P}$ be the first production rule applied to $\mathtt{a}$ in the derivation $\mathtt{a} \Longrightarrow_{\mathbf{P}} s$. We have three cases: if $p \in \mathbf{P}_\Sigma$ or $p \in \mathbf{P}_\lambda$, then, due to the initialization steps of Algorithm 3 (the *for*-loops at Line 3 and Line 7), we must already have $\mathbf{A}(\mathtt{a}) = p$, and, hence, $\mathit{cost}(\mathtt{a}) = |s|$. The only case remaining is $p \in \mathbf{P}_2$. Assume $p = (\mathtt{a} \rightarrow \mathtt{b}\,\mathtt{c})$. We just removed $\mathtt{a}$, hence, by Invariant (iv), $\mathtt{b}, \mathtt{c} \notin \mathit{cost} \setminus \mathit{new}$ and, as Invariant (v) was valid before removing $\mathtt{a}$, we have $\mathit{cost}(\mathtt{b}) = \mathrm{length}_\downarrow(\mathtt{b})$ and $\mathit{cost}(\mathtt{c}) = \mathrm{length}_\downarrow(\mathtt{c})$ and, thus, $\mathit{cost}(\mathtt{a}) \geq \mathit{cost}(\mathtt{b}) + \mathit{cost}(\mathtt{c})$. Hence, during the iteration of the main *while*-loop that removed $\mathtt{b}$ or $\mathtt{c}$ (whichever came later) the procedure PRODUCE must have recognized that $\mathit{cost}(\mathtt{a})$ could be lowered. Hence, we must have $\mathit{cost}(\mathtt{a}) = \mathit{cost}(\mathtt{b}) + \mathit{cost}(\mathtt{c}) = \mathrm{length}_\downarrow(\mathtt{b}) + \mathrm{length}_\downarrow(\mathtt{c}) = \mathrm{length}_\downarrow(\mathtt{a})$. □

First, we provide the worst-case running time of Algorithm 3 when applied on arbitrary context-free grammars. Then, we investigate the worst-case running time of Algorithm 3 when applied to graph-annotated grammars. This directly leads to the worst-case running time for computing the minimizing acyclic subset from which we can produce shortest paths.

Proposition 6. *The worst-case running time of Algorithm 3 on context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *is* $\mathcal{O}(\mathbf{P}_\lambda + \mathbf{P}_\Sigma + (\mathbf{N}\log(\mathbf{N}) + \mathbf{P}_2)\mathbf{N})$ *using* $\mathcal{O}(\mathbf{C} + \mathbf{N}^2)$ *memory.*

Proof. We represent *costs* as an array holding $|\mathbf{N}|$ integers. The costs (used in *cost* and *new*) are integers in the range $0, \dots, 2^{|\mathbf{N}|} - 1$. Hence, operations on these integers will cost $I = \mathcal{O}(\log(2^{\mathbf{N}} - 1)) = \mathcal{O}(\mathbf{N})$. In the worst case $|\mathbf{N}| + 2|\mathbf{P}_2|$ increment operations on *ts* are performed, upper-bounding the maximum value stored in *ts* by $\mathcal{O}(\mathbf{N}^3)$, hence, operations on these integers cost at most $\mathcal{O}(\log(\mathbf{N}))$ and the total cost for operations on *ts* is $\mathcal{O}((\mathbf{N} + \mathbf{P}_2)\log(\mathbf{N}))$.

The initialization steps perform $\mathcal{O}(\mathbf{P}_\lambda + \mathbf{P}_\Sigma)$ 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 $2|\mathbf{P}_2|$ decrease key operations on priority queue *new*. Using a Fibonacci heap for these priorities, each insert and removal costs $\mathcal{O}(\log(e))$ heap operations in total, with e the number of elements in the heap, and each decrease key operation costs an amortized $\mathcal{O}(1)$ heap operations [9, 15]. Hence, the total cost for the operations on *new* is $\mathcal{O}((\mathbf{N}\log(\mathbf{N}) + 2\mathbf{P}_2)I) = \mathcal{O}((\mathbf{N}\log(\mathbf{N}) + \mathbf{P}_2)\mathbf{N})$. □

Using Lemma 2, Proposition 3, and Proposition 6, we can derive the following complexity results on the combination of Algorithm 1 and Algorithm 3:

Corollary 1. *Let* $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, \mathbf{P}_{[G]})$ *be the graph-annotated grammar of context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *and graph* $G = (\Sigma, V, E)$*. The minimizing acyclic subset of* $\mathbf{P}_{[G]}$ *can be constructed in* $\mathcal{O}(\mathbf{P}_\lambda V + \mathbf{N}E + (\mathbf{N}V^2\log(\mathbf{N}V) + \mathbf{P}_2 V^3)\log(I))$ *using* $\mathcal{O}(\mathbf{P}_\lambda V + \mathbf{N}E + \mathbf{P}_2 V^3 + \mathbf{N}V^2\log(I))$ *memory and* $\mathcal{O}(\textsc{scan}(E))$ *IOs, whereby* $\log(I) = \mathcal{O}(\mathbf{N}V^2)$ *is the number of bits needed to store the maximum possible length of a path defined by a proper acyclic subset.*

Combining Algorithm 1 and Algorithm 3 leads to an approach that has high memory usage. This is, to a large degree, the consequence of constructing and keeping the graph-annotated grammar in memory. We can also combine Algorithm 1 and Algorithm 3 into a single algorithm, Algorithm 4, that constructs the minimizing acyclic subset of the graph-annotated grammar without constructing the entire graph-annotated grammar, this by deriving the necessary parts of the graph-annotated grammar in-place.

Theorem 3. *Let* $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ *be a context-free grammar and let* $G = (\Sigma, V, E)$ *be a graph. The function* $\mathbf{A}$ *constructed by the invocation of Algorithm 4 is a minimizing acyclic subset of the production rules of the graph-annotated grammar of* $\mathbf{C}$ *and* G*.*

Proposition 7. *The worst-case running time of Algorithm 4 on context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ *and graph* $G = (\Sigma, V, E)$ *is*

$$\mathcal{O}(\mathbf{P}_\lambda V + \mathbf{N}E + (\mathbf{N}V^2\log(\mathbf{N}V) + \mathbf{P}_2 V^3)\log(I))$$

Algorithm 4 MINIMIZING-ACYCLIC-SUBSET*

```
Input: context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ with $\mathbf{P} = (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda)$ and graph $G = (\Sigma, V, E)$.
Output: a minimizing acyclic subset $\mathbf{A}$ of the productions of the graph-annotated grammar $\mathbf{C}_{[G]}$.
$cost, \mathbf{A}, ts := \emptyset, \emptyset, 0$
$new$ is a min-priority queue
for all $(\mathtt{a} \to \lambda) \in \mathbf{P}_\lambda$ and $n \in V$ do
    $cost(\mathtt{a}[n,n]), \mathbf{A}(\mathtt{a}[n,n]) := 0, (\mathtt{a}[n,n] \to \lambda)$
    $new, ts := new \cup \{\mathtt{a}[n,n]$ with priority $\langle 0, ts\rangle\}, ts + 1$
end for
for all $(\mathtt{a} \to \sigma) \in \mathbf{P}_\Sigma$ and $(m, \sigma, n) \in E$ do
    if $\mathtt{a}[m,n] \notin cost$ then
        $cost(\mathtt{a}[m,n]), \mathbf{A}(\mathtt{a}[m,n]) := 1, (\mathtt{a}[m,n] \to \sigma)$
        $new, ts := new \cup \{\mathtt{a}[m,n]$ with priority $\langle 1, ts\rangle\}, ts + 1$
    end if
end for
while $new \neq \emptyset$ do
    take and remove $\mathtt{a}[m,n]$ from $new$
    for all $(\mathtt{c} \to \mathtt{a}\,\mathtt{b}) \in \mathbf{P}_2$ and $\mathtt{b}[n,o] \in cost$ do
        PRODUCE$(\mathtt{c}[m,o] \to \mathtt{a}[m,n]\,\mathtt{b}[n,o])$
    end for
    for all $(\mathtt{c} \to \mathtt{b}\,\mathtt{a}) \in \mathbf{P}_2$ and $\mathtt{b}[o,m] \in cost$ do
        PRODUCE$(\mathtt{c}[o,n] \to \mathtt{b}[o,m]\,\mathtt{a}[m,n])$
    end for
end while
return $\mathbf{A}$

Procedure PRODUCE$(p = (\mathtt{d}[x,z] \to \mathtt{e}[x,y]\,\mathtt{f}[y,z]))$:
$c := cost(\mathtt{e}[x,y]) + cost(\mathtt{f}[y,z])$
if $\mathtt{d}[x,z] \notin cost$ then
    $cost(\mathtt{d}[x,z]), \mathbf{A}(\mathtt{d}[x,z]) := c, p$
    $new, ts := new \cup \{\mathtt{d}[x,z]$ with priority $\langle c, ts\rangle\}, ts + 1$
else if $cost(\mathtt{d}[x,z]) > c$ then
    $cost(\mathtt{d}[x,z]), \mathbf{A}(\mathtt{d}[x,z]) := c, p$
    lower priority of $\mathtt{d}[x,z]$ in $new$ to $\langle c, ts\rangle$
    $ts := ts + 1$
end if
```

using $\mathcal{O}(\mathbf{C} + \mathbf{N}\,V^2 \log(I))$ *memory and* $\mathcal{O}(\text{SCAN}(E))$ *IOs, whereby* $\log(I) = \mathcal{O}(\mathbf{N}\,V^2)$ *is the number of bits needed to store the maximum possible length of a path defined by a proper acyclic subset.*

Although this combined approach reduced the memory consumption significantly, the worst-case runtime complexity remains high. We observe that, for a large degree, this complexity stems from the worst-case upper bound I that we use on the size of shortest paths:

Proposition 8. *Let* $\mathbf{C}_{[G]} = (\Sigma, \mathbf{N}_{[G]}, (\mathbf{P}'_2, \mathbf{P}'_\Sigma, \mathbf{P}'_\lambda))$ *be the graph-annotated grammar of context-free grammar* $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ *and graph* $G = (\Sigma, V, E)$ *and let* $\mathtt{a}[m,n] \in \mathbf{N}_{[G]}$ *be a graph-annotated non-terminal. The worst-case length of a shortest path for* $\mathtt{a}[m,n]$ *is upper bounded by* $\mathcal{O}(2^{|\mathbf{N}||V|^2})$ *and lower bounded by* $\Omega(|V|2^{|\mathbf{N}|})$.

Proof (sketch). The worst-case upper bound is provided by Lemma 4. For the worst-case lower bound we consider context-free grammars such that $\mathtt{a} \in \mathbf{N}$ recognizes strings of length $y2^{|\mathbf{N}|}$, $1 \leq y$, and graphs consisting of a single closed path. We consider a path from node $n \in V$ to node n, any such path has length $x|V|$, $1 \leq x$. Hence, the shortest path $n\pi n$ with $\text{trace}(\pi) \in \mathcal{L}(\mathbf{C}, \mathtt{a})$ has length $\text{lcm}(|V|, 2^{|\mathbf{N}|})$ which is $|V|2^{|\mathbf{N}|}$ if $|V|$ is odd. □

Proposition 8 provides a worst-case upper bound, it does, however, not provide a strict worst-case upper bound. The construction of Lemma 4 can only be applied directly on graph-annotated grammars in corner cases (such as graphs with a single node and context-free grammars with a single non-terminal): Definition 3 puts severe restrictions on the graph-annotations of graph-annotated non-terminals that are allowed in graph-annotated production rules. Hence, one way to improve the worst-case runtime of Algorithm 4 is to provide better worst-case upper bounds on the length of shortest paths with respect to a context-free grammar and a graph:

Open Problem 1. Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar and let $G = (\Sigma, V, E)$ be a graph. What is the strict worst-case upper bound on the length of a shortest string in the graph-annotated grammar $\mathbf{C}_{[G]}$? Or, equivalently, what is the strict worst-case upper bound on the length of a shortest string in the intersection of the language of a context-free grammar with x non-terminals and the language of a (non-)deterministic automaton with y states?

3.2.2 Deriving an 'Acyclic' Path

Algorithm 4 performs a lot of bookkeeping to assure the construction of the shortest possible path for each graph-annotated non-terminal and this bookkeeping has a significant impact on the worst-case runtime complexity. When we loosen the requirement that the produced paths need to be the shortest possible paths, then we can also reduce the amount of bookkeeping.

As a minimal requirement, we want to assure that the produced paths have an upper-bounded length. We have already seen that this requirement is satisfied if we produce a proper acyclic subset. Hence, we opt to produce such a proper acyclic subset instead of a minimizing acyclic subset. As Algorithm 2 can be reused to produce paths from proper acyclic subsets, we only need an algorithm to construct proper acyclic subsets. We can construct such an algorithm by modifying Algorithm 1 such that the algorithm only keeps track of the first possible production rule for each graph-annotated non-terminal, resulting in Algorithm 5.

Theorem 4. *Let $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ be a context-free grammar and let $G = (\Sigma, V, E)$ be a graph. The function $\mathbf{A}$ constructed by the invocation of Algorithm 5 is a proper acyclic subset of the production rules of the graph-annotated grammar of $\mathbf{C}$ and G.*

Proof (sketch). The order in which Algorithm 5 assigns values $\mathbf{A}(\mathsf{a}[m, n])$, with $\mathsf{a}[m, n] \in \mathbf{N}_{[G]}$, guarantees that the production rules in image($\mathbf{A}$) are acyclic and that the derivation of $\mathsf{a}[m, n]$ terminates in a string over Σ. □

Proposition 9. *The worst-case running time of Algorithm 5 on context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda))$ and graph $G = (\Sigma, V, E)$ is $\mathcal{O}(\mathbf{P}_\lambda V + \mathbf{N}E + \mathbf{N}V^2 + \mathbf{P}_2 V^3)$ using $\mathcal{O}(\mathbf{C}) + \mathbf{N}V^2$ bits memory and $\mathcal{O}(\textsc{scan}(E) + \textsc{scan}(\mathbf{A}))$ IOs.*

The quality of the proper acyclic subset $\mathbf{A}$ produced by Algorithm 5, in terms of the length of the paths derivable from $\mathbf{A}$, depends completely on the ordering in which production rules are visited. This ordering in itself is mainly influenced by the operations of *new* (and the implementation details of the representation of the context-free grammar $\mathbf{C}$). Independent of these details, we can only guarantee the worst-case upper bounds of Lemma 4 on the length of the produced paths.

Open Problem 2. Can we manage the queue *new* in Algorithm 5 in such a way that we can guarantee the computation of *good* proper acyclic subsets, for example: the computation of proper acyclic subsets that can derive paths whose length is only a constant multiple of the shortest paths, this while not affecting the worst-case running time?

In Section 4.2 we will investigate the effects of different choices for the representation of the queue *new* on the size of the produced paths.

Algorithm 5 PROPER-ACYCLIC-SUBSET*

```
Input: context-free grammar $\mathbf{C} = (\Sigma, \mathbf{N}, \mathbf{P})$ with $\mathbf{P} = (\mathbf{P}_2, \mathbf{P}_\Sigma, \mathbf{P}_\lambda)$ and graph $G = (\Sigma, V, E)$.
Output: a proper acyclic subset $\mathbf{A}$ of the productions of the graph-annotated grammar $\mathbf{C}_{[G]}$.
$\mathbf{N}_{[G]}, \mathbf{A} := \emptyset, \emptyset$
for all $(\mathtt{a} \to \lambda) \in \mathbf{P}_\lambda$ and $n \in V$ do
    $\mathbf{N}_{[G]}, \mathbf{A}(\mathtt{a}[n,n]) := \mathbf{N}_{[G]} \cup \{\mathtt{a}[n,n]\}, (\mathtt{a}[n,n] \to \lambda)$
end for
for all $(\mathtt{a} \to \sigma) \in \mathbf{P}_\Sigma$ and $(m, \sigma, n) \in E$ do
    if $\mathtt{a}[m,n] \notin \mathbf{N}_{[G]}$ then
        $\mathbf{N}_{[G]}, \mathbf{A}(\mathtt{a}[m,n]) := \mathbf{N}_{[G]} \cup \{\mathtt{a}[m,n]\}, (\mathtt{a}[m,n] \to \sigma)$
    end if
end for
$new := \mathbf{N}_{[G]}$
while $new \neq \emptyset$ do
    take and remove a $\mathtt{a}[m,n]$ from $new$
    for all $(\mathtt{c} \to \mathtt{a}\,\mathtt{b}) \in \mathbf{P}_2$ and $\mathtt{b}[n,o] \in \mathbf{N}_{[G]}$ do
        PRODUCE($\mathtt{c}[m,o] \to \mathtt{a}[m,n]\,\mathtt{b}[n,o]$)
    end for
    for all $(\mathtt{c} \to \mathtt{b}\,\mathtt{a}) \in \mathbf{P}_2$ and $\mathtt{b}[o,m] \in \mathbf{N}_{[G]}$ do
        PRODUCE($\mathtt{c}[o,n] \to \mathtt{b}[o,m]\,\mathtt{a}[m,n]$)
    end for
end while
return $\mathbf{A}$

Procedure PRODUCE($p = (\mathtt{d}[x,z] \to \mathtt{e}[x,y]\,\mathtt{f}[y,z])$):
if $\mathtt{d}[x,z] \notin \mathbf{N}_{[G]}$ then
    $\mathbf{A}(\mathtt{d}[x,z]) \leftarrow p$
    $new, \mathbf{N}_{[G]} := new \cup \{\mathtt{d}[x,z]\}, \mathbf{N}_{[G]} \cup \{\mathtt{d}[x,z]\}$
end if
```

4 Experiments

To investigate the empirical behavior of our algorithms, we have performed two separate experiments on a C++ implementation of Algorithm 1, Algorithm 4, and Algorithm 5. These experiments were conducted on a standard desktop with an Intel Core i5-4670 processor and 16GB of main memory. During testing, all data structures were kept in main memory and we have monitored the tests to ensure that the runtime performance of the experiments was only bound by processor time (and not by other activities or memory exhaustion).

The first experiment focused on the *all-paths semantics* and Algorithm 1, and the second experiment focused on the *single-path semantics* and Algorithms 4 and 5.

4.1 The All-Paths Semantics

For the all-paths semantics we studied the running time behavior of Algorithm 1. To look into the impact of the density of the resulting set of production rules $\mathbf{P}'_2$, we tested with four different representations of $\mathbf{P}'_2$ and measured how these representations perform under various circumstances. The main representations of $\mathbf{P}'_2$ we tested were the *Boolean matrix* representation as described in Proposition 3 and the *(array based) list* representation as described in Proposition 4. Furthermore, we also tested with set representations of $\mathbf{P}'_2$ by using standard *binary search trees* and *hash-tables*.

We tested using two context-free grammars that where explicitly constructed to produce low and high density results with respect to the produced set of production rules $\mathbf{P}'_2$. The *'sparse' grammar* only recognizes edges and, hence, will only produce $\mathbf{P}'_2$ with a high density if we query a dense graph (in the number of edges). The production rules for this 'sparse' grammar are as follows: $\mathtt{a} \rightarrow \sigma, \mathtt{b} \rightarrow \lambda, \mathtt{b} \rightarrow \mathtt{b}\,\mathtt{a}, \mathtt{b} \rightarrow \mathtt{a}\,\mathtt{b}$. The *'dense' grammar* recognizes all non-empty paths, and, hence, will produce $\mathbf{P}_2'$ with a high density whenever the graph is (for a large part) strongly connected. The production rules for this 'dense' grammar are as follows: $\mathtt{a} \rightarrow \sigma, \mathtt{a} \rightarrow \mathtt{a}\,\mathtt{a}$.

We tested using three types of graphs. The *path graphs* consist of a single simple path: in this graph there are few edges and the nodes can only reach a low amount of other nodes. The *cycle graphs* consist of a single closed simple path: in this graph there are few edges, while the graph is strongly connected. The *full graphs* consist of an edge between every pair of distinct nodes: in this graph there are many edges and the graph is strongly connected.

In Figure 2 we have plotted the results for this experiment.

Analysis of the Results

Algorithm 1 terminated in a reasonable amount of time when we use the Boolean matrix representation or the list representation for $\mathbf{P}_2'$. We also see that the set representations for $\mathbf{P}_2'$ (using binary search trees or hash-tables) are not practical: they cannot handle $\mathbf{P}_2'$ with high density and when $\mathbf{P}_2'$ has low density, then the list representation still slightly outperforms the set representations. Moreover, the list representation has less memory overhead and can be pushed to external memory. We also see that the savings for choosing the list representation (when $\mathbf{P}_2'$ is expected to have low density) or the Boolean matrix representation (when $\mathbf{P}_2'$ is expected to have high density) can have a significant impact.

In practice, we expect that in most circumstances the produced $\mathbf{P}_2'$ has relatively low density: we expect practical context-free graph queries to be more selective and we expect that graphs are not as dense and structurally connected. In these settings, choosing for the list representation would be preferable. Moreover, by using the list representation, large parts of Algorithm 1 can be pushed to external memory, making Algorithm 1 capable of processing bigger graphs.

4.2 The Single-Path Semantics

For the single-path semantics we make a comparison between Algorithm 4 and Algorithm 5 in terms of running time and the quality of the produced paths. To look into the impact of different representations for the queue *new* used in Algorithm 5 (see also Open Problem 2), we tested Algorithm 5 with two distinct queue representations: a last-in first-out *stack* and a first-in first-out *double-ended queue*.

For this experiment, we used the context-free grammar of the *same generations*-query of Example 1. After putting this context-free grammar in normal form, we have the following production rules: $\mathtt{a} \rightarrow \sigma_1, \mathtt{b} \rightarrow \sigma_2, \mathtt{c} \rightarrow \lambda, \mathtt{c} \rightarrow \mathtt{a}\,\mathtt{d}, \mathtt{d} \rightarrow \mathtt{c}\,\mathtt{b}$. We evaluated this context-free grammar on a full graph wherein each edge has a probability of two in five to be labeled with σ_1 and, else, the edge is labeled with σ_2.

In Figure 3 we have plotted the results for this experiment.

Analysis of the Results

Both Algorithm 4 and Algorithm 5 terminated in reasonable amounts of time, this in all tests we performed. We also see that calculating the minimizing acyclic subset using Algorithm 4 is, unsurprisingly, the slowest approach and is always outperformed by calculating proper acyclic subsets via Algorithm 5. For Algorithm 5, we see that using a stack for *new* is consistently faster than using a double-ended queue. An explanation for this difference in running time, when using a stack or when using a double-ended queue, can be found in the memory-access patterns of the queue implementations: these are (always) cache-friendlier when using a stack.

When we look at the constructed proper acyclic subsets, we see that Algorithm 5, using a double-ended queue, produces results that are close to the optimal minimizing acyclic subset produced by Algorithm 4. We also observe that the proper acyclic subset produced by Algorithm 5, when using a stack, is of very low quality: the paths derivable from this proper acyclic subset are much longer than the optimal paths.

The difference in the quality of the proper acyclic subset produced when using a stack or a double-ended queue can be explained by considering the order in which new production rules in $\mathbf{P}_2'$ are explored and, hence, assigned to $\mathbf{A}$. When using a stack, Algorithm 5 produces the production rules in $\mathbf{P}_2'$ in a 'depth-first' manner by combining the most-recently produced non-terminal with other non-terminals. When using a double-ended queue, Algorithm 5 produces the production rules in $\mathbf{P}_2'$ in a 'breadth-first' manner by combining the least-recently produced non-terminal with other non-terminals. Thereby, the more-recently produced non-terminals are likely to have more complex and longer derivations with respect to the production rules that are already in image($\mathbf{A}$).

In practice, the differences between Algorithm 4 and Algorithm 5 in terms of running time can be bigger, especially when evaluating more complex context-free grammars on bigger graphs. In these settings choosing for Algorithm 5 can be preferable due to the lower running time and the ability to push large parts of Algorithm 5 to external memory, making Algorithm 5 capable of processing bigger graphs.

5 Conclusion

To provide more insight in the structure of graphs, we proposed path-based semantics for query languages that use context-free grammars for navigation. We studied two such path-based semantics, namely evaluating context-free grammars to the set of all paths whose labeling is accepted by the context-free grammar, and to a single path whose labeling is accepted by the context-free grammar. For both these semantics we introduced the necessary data structures to represent a (possible infinite) set of paths, algorithms to construct these data structures, and algorithms to derive paths from these data structures.

All our algorithms have worst-case polynomial complexity in terms of the size of the context-free grammar and the graph. We studied practical performance by means of performance measurements on a `C++` implementation. With our performance measurements, we also studied trade-offs that can be made while implementing the algorithms: for the all-paths semantics we showed a trade-off between memory consumption and (worst-case) running time and for the shortest-paths semantics we showed a trade-off between the length of the derived paths in one hand and the running time and memory consumption in the other hand. Based on the performance measurements, we believe that our algorithms not only have practical applications but are also practically usable. In conclusion, we believe that our work opens the door for further study of path-based evaluation of (context-free) graph queries.

Besides the open problems already stated in this work, other directions for future work can be found in improving the performance of the presented algorithms:

Open Problem 3. Can we use more efficient algorithms for the problems outlined in this paper? Can we, for example, use Boolean matrix multiplication techniques [16, 26]?

The query evaluation techniques presented in this work are all focused on answering a query for an entire graph. There are also plenty practical examples of queries that can be answered by only visiting a small localized part of the graph. Take, for example, social networks: users usually only want to query friends and family, and not the entire social network:

Open Problem 4. Can we write more efficient algorithms when evaluating queries for a specific non-terminal and a pair of nodes by, for example, using top-down techniques or using Datalog-inspired *magic set* evaluation techniques [1, 2]?

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, 1986.

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

[4] P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 115–124, 2012.

[5] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37(4):31:1–31:46, 2012.

[6] P. Barceló, J. Pérez, and J. L. Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management*, pages 180–195, 2012.

[7] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, and J. R. and Jérôme Siméon. XML path language (XPath) 2.0 (second edition). `http://www.w3.org/TR/2010/REC-xpath20-20101214/`. W3C Recommendation 14 December 2010 (Link errors corrected 3 January 2011).

[8] J. Clark and S. DeRose. XML path language (XPath) version 1.0. `http://www.w3.org/TR/1999/REC-xpath-19991116/`. W3C Recommendation 16 November 1999.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2009.

[10] P. Dömösi. Unusual algorithms for lexicographical enumeration. *Acta Cybernetica*, 14(3):461–468, 2000.

[11] Y. Dong. Linear algorithm for lexicographic enumeration of CFG parse trees. *Science in China Series F: Information Sciences*, 52(7):1177–1202, 2009.

[12] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, 1997.

[13] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. In *Proceedings of the 14th International Conference on Database Theory*, pages 197–207, 2011.

[14] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 139–148, 1998.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[16] D. Grune and C. J. H. Jacobs. *Parsing Techniques.* Monographs in Computer Science. Springer New York, 2008.

[17] J. Hellings. Conjunctive context-free path queries. In *ICDT*, pages 119–130, 2014.

[18] M. Lange. Model checking propositional dynamic logic with all extras. *Journal of Applied Logic*, 4(1):39–49, 2006.

[19] P. Linz. *An Introduction to Formal Languages and Automata, Fifth Edition.* Jones & Bartlett Publishers, 2012.

[20] E. Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–61, 1997.

[21] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *ACM SIGMOD Record*, 34(2):41–46, 2005.

[22] M. J. Mclean and 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*, pages 180–196. Springer Berlin Heidelberg, 1975.

[23] P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2), 2008.

[24] B. ten Cate. The expressivity of XPath with transitive closure. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 328–337. ACM, 2006.

[25] The W3C SPARQL Working Group. SPARQL 1.1 overview. `http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/`. W3C Recommendation 21 March 2013.

[26] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975.

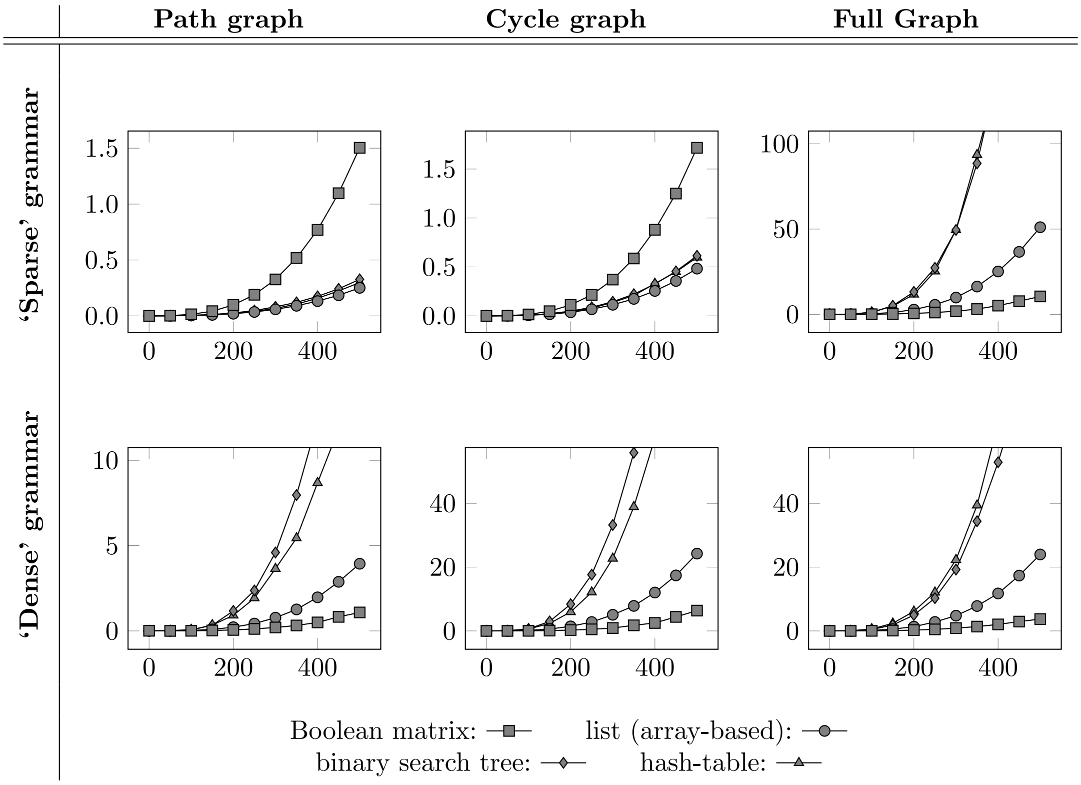

Figure 2: Relation between the size of a graph ($|V|$, horizontal) and the running time (s, vertical) of Algorithm 1 with respect to three types of graphs and two types of grammars. Each line indicates the running time while using a specific representation of $\mathbf{P}_2'$.

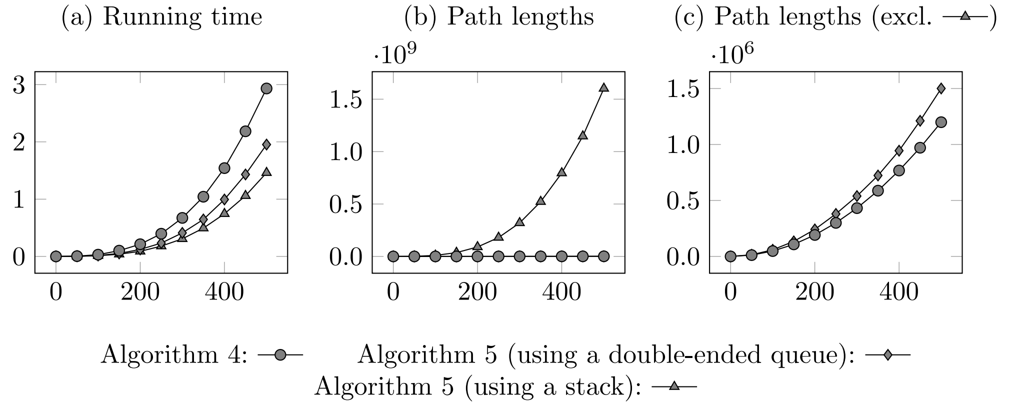

Figure 3: Measurements on the single-path query evaluation algorithms. Figure 3(a) shows the relation between the size of a graph ($|V|$, horizontal) and the running time (s, vertical). Figure 3(b) and Figure 3(c) show the relation between the size of a graph ($|V|$, horizontal) and the total length of all the produced paths (vertical); thereby Figure 3(b) included Algorithm 5 (using lifo queue) and Figure 3(c) excluded Algorithm 5 (using lifo queue).