An Algorithm for Context-Free Path Queries over Graph Databases

Ciro M. Medeiros  
cirommed@ppgsc.ufrn.br  
Federal University of Rio Grande do Norte  
Natal, Brazil  
University of Orléans  
Orléans, France

Martin A. Musicante  
{mam,umberto}@dimap.ufrn.br  
Federal University of Rio Grande do Norte  
Natal, Brazil

ABSTRACT
Path queries are used to specify paths inside a data graph to match a given pattern. Query languages such as SPARQL usually include support for regular path patterns defined by means of regular expressions. Context-free path queries define a path whose language can be defined by a context-free grammar. This kind of query is interesting in practice in domains such as genetics, data science, or source code analysis. In this paper, we present a novel algorithm for context-free path query processing. Our algorithm works by looking for localized paths, allowing us to process subgraphs, in contrast to other approaches that have to process the whole graph. It also takes any context-free grammar as input, avoiding the use of normal forms that are more problematic in practice. The output of our algorithm provides enough information to reconstruct the paths matching the query. We prove the correctness of our approach and show its runtime and memory complexity. We show the viability of our approach by means of a prototype implemented in Go. We run experiments proposed in recent works, which include both synthetic and real RDF databases. Our algorithm shows some performance gains when compared with other algorithms implemented using single-thread programs.

CCS CONCEPTS
• Theory of computation → Grammars and context-free languages; Data structures design and analysis; Graph algorithms analysis; • Information systems → Graph-based database models; Resource Description Framework (RDF);

KEYWORDS
graph path queries, context-free grammars, RDF

1 INTRODUCTION
Processing a path query over a graph database consists of looking for pairs of vertices connected by a specified path inside the graph. The labels of the edges in a path form a string and, as such, can be specified by using grammars or other formal tools. Regular expressions have been widely used to define path queries. As regular languages belong to the most restricted class of formal languages, the expressivity of such queries is somehow limited. Recent studies have developed algorithms for supporting the use of context-free grammars in path queries to improve their expressiveness.

Graph databases can be represented as a set of triples. Each triple relates a subject and an object by means of a predicate. In Linked Data, the standard format for representing triples is the Resource Description Framework (RDF). SPARQL is the standard language for querying RDF graph databases. SPARQL supports the definition of paths using regular expressions over labels of edges in the graph.

Although regular expressions are widely used, some applications can benefit of more sophisticated queries that cannot be defined using regular expressions, but may be described by context-free grammars [1, 7]. Those applications include source code analysis [20], data science [11] and genetics [5].

In this paper we present a new approach that, while it is not based on a specific parsing technique, it uses annotations over grammar items to parse several paths at the same time, considering shared prefixes over these paths. Our main contributions are: (i) an algorithm for evaluation of context-free path queries that has useful features; (ii) an analysis of correctness, as well as time and space complexity for the algorithm; (iii) experimental results that demonstrate its applicability in different scenarios.

This paper is organized as follows. Section 2 briefly presents graphs and queries. Section 3 formalizes context-free path query evaluation, presents our algorithm for this purpose and states some important properties of it. Section 4 gathers related works from the past decade. Section 5 analyses the performance of our approach in a number of experiments. Section 6 concludes with some final remarks and points out for future research directions.

2 DATA GRAPHS AND QUERIES
In this section we present concepts related to graph databases and path queries. A graph database is a set of triples. A triple is denoted by \((s, p, o)\), where \(s\) is the subject, \(p\) is the predicate and \(o\) is the object. The subject and object represent entities, while the predicate states the relationship between them. Those concepts are formalized next.
Definition 2.1 (Graph). A graph is a set of triples in $V \times E \times V$, where $V$ is a set of vertices and $E$ is a set of edge labels. It is possible that $V \cap E \neq \emptyset$.

Triples from a graph can be organized to form paths. Those paths can be defined using context-free grammars [2].

Definition 2.2 (Paths and Traces). A path is a sequence of triples $(t_1, t_2, \ldots, t_k)$ from a given graph, where $t_i = (s_i, p_i, o_i)$, such that $o_i = s_{i+1}$. The trace of a path is the string formed by the concatenation of the edge labels $p$ from its triples. The set of paths between two vertices $x$ and $y$ is denoted by $\text{paths}(x, y)$. Notice that this includes the empty path between two nodes and itself. Given a set of paths $\Pi \subseteq \text{paths}(x, y)$, the set of traces defined by these paths is denoted as $\text{traces}(\Pi)$.

In the next section we discuss a new algorithm to implement context-free path queries over graph databases.

3 CONTEXT-FREE PATH QUERY EVALUATION

The next definition establishes the set of vertices that are reachable from a given vertex, by following a path represented by a string of (terminal and non-terminal) symbols of a grammar.

Definition 2.3 (Context-Free Path Query). Given a data graph $D$ and a context-free grammar $G = (\Sigma, \mathcal{P}, S)$, a context-free path query (CFPQ) $Q$ is a set of query pairs $(x, A)$ where $x$ is a vertex of the graph and $A$ a non-terminal symbol from a given grammar. The evaluation of a context-free path query $Q$ produces the set of all vertexes $y$ such that there exists a path from $x$ to $y$ whose trace is derivable by $A$, that is: $\text{Eval}(Q) = \{ y \mid \exists s : A \Rightarrow^* s \land s \in \text{traces}(\text{paths}(x, y)) \}$.

In the next section we discuss a new algorithm to implement context-free path queries over graph databases.

3.1 Our Algorithm

In this section we present our approach for the evaluation of CFPQs. Our algorithm receives a grammar, a data graph and a query, and follows context-free paths inside the data graph. The goal of the algorithm is to identify pairs of vertices linked by paths whose traces are strings generated by the grammar.

The following example illustrates the problem:

Example 3.3. Let us consider a grammar $G$ with production rules $P = \{ S \rightarrow a S b, S \rightarrow e \}$ and the data graph given in Figure 1.

Given the query $Q = \{(1, S), (3, S)\}$, our algorithm goes through paths starting at vertices 1 and 3 whose trace is generated by $S$. In this way all the production rules of $S$ will be investigated for paths starting at each of these vertices. For the above query $Q$, our algorithm will compute the sets of vertices $\{1, 3, 4\}$, reachable from node 1, and the set $\{3, 4\}$, reachable from node 3.

Our algorithm relies on two facts: (i) there may be several paths starting at a given node of the data graph; and (ii) for each of these paths, their trace may be derivable from a non-terminal of the grammar. It explores these two properties to parse all the paths from a given vertex, to discover which of them have traces derivable by a given non-terminal. The parsing of all these traces is performed in an incremental way. For each pair $(v, A)$ from the query $Q$, our algorithm identifies all the paths from $v$ whose traces are strings derivable from $A$.

In a traditional parsing setting, one may use the notion of grammar items to guide the parsing process. Grammar items use a dot on the right-hand side of a production rule to mark the progress of the parsing. Traditional parsing techniques are tailored to process one input string at a time. The information carried by the dot is related to the parsing progress. In our case, we also need to identify the vertices that form paths of the graph being parsed. Thus, we associate sets of vertices to the positions within the items, in the
An Algorithm for Context-Free Path Queries over Graph Databases

Definition 3.4 (Trace Item). Given a context-free grammar \( G = (N, \Sigma, P, S) \) and a data graph \( D \subseteq V \times E \times V \), a trace item is a pair formed by a production rule and a function associating a set of graph nodes to each position of the right-hand side of the rule. Formally, a trace item is defined as the pair \( (A \rightarrow a, f) \), where \( A \rightarrow a \in P \) and \( f : \{0, \ldots, |a|\} \rightarrow \mathcal{P}(V) \).

The trace item \( (A \rightarrow a_1, \ldots, a_n, f) \), where \( f = \{0 \mapsto C_0, \ldots, n \mapsto C_n\} \) will be noted as \( [A \rightarrow C_0 \ a_1 \ C_1 \ \ldots \ a_n \ C_n] \). The sets \( C_1, \ldots, C_n \) will be called position sets.

In general, given position sets \( C_1, C_2 \) and a grammar symbol \( a \), a sequence \( C_1 \ a \ C_2 \) on the right-hand side of an item indicates that each vertex in \( C_2 \) will be reached by an \( a \)-derivable path beginning at some vertex in \( C_1 \). For instance, the trace item \( [S \rightarrow \{1\} \ a \{2, 3\} \ S \{\} \ b \{\} ] \) in Example 3.3 indicates that the parsing process is in a stage where \( a \)-derivable paths linking vertex 1 to vertices 2 and 3 in the data graph have been identified.

Next, we present the intuitive idea of our algorithm. To solve a query \( Q \), our algorithm will start processing trace items obtained from the query pairs and rules of the grammar: for each query pair \( (v, A) \in Q \), we create one trace item for each production rule of \( A \) with \( v \) in its first position set. We will use special marks * and • respectively for unprocessed and processed vertices in position sets, to keep track of which vertices have already been processed\(^7\). Our algorithm will process trace items until there are no unprocessed vertices belonging to any position set.

The next example shows how to compute the answers for the given query, graph and grammar.

Example 3.5. Given the query \( Q = \{(1, S), (3, S)\} \) and data graph \( D \) and grammar \( G \) from Example 3.3, we start the parsing process by creating trace items. For each query pair \( (v, A) \in Q \), we create one trace item for each production rule of \( A \) with \( v \) in its first position set. For the query \( Q \) we build the trace items:

\[
\begin{align*}
[S & \rightarrow \{1\} \ a \{\} \ S \{\} \ b \{\} ] \quad (1) \\
[S & \rightarrow \{1\} ] \quad (2) \\
[S & \rightarrow \{3\} ] \\
[S & \rightarrow \{3\} ] \quad (4)
\end{align*}
\]

Our algorithm picks the unprocessed vertices in an arbitrary order. Let us start with vertex 1 from trace item (1). This vertex appears in a position set before the terminal symbol \( a \). We must walk from vertex 1 to all its neighbors linked by an \( a \)-labeled edge in \( D \). The neighbor vertices 2 and 3 must then be added to the next position set in the trace item. Doing so, our item will become \( [S \rightarrow \{1\} \ a \{2^*, 3^*\} \ S \{\} \ b \{\} ] \). Notice that vertex 1\(^*\) has changed to 1\(^\circ\) to signal that it has been processed. New vertices are added as unprocessed by using the mark *.

Now we may pick vertex 2 for the next step. This vertex is in a position set before the non-terminal symbol \( S \). That indicates that

\(^7\)We omit the * and • marks from vertices in position sets when such distinction is unnecessary.
I. In the first case (lines 6-10), given the trace item \( \alpha \) and rule \( \alpha \to \{ \alpha x \} \), the vertex \( \alpha \) belongs to the last position set of a trace item. The item \( \alpha \) is marked as visited (line 10).

II. In the second case of the main loop (lines 11 to 14) we identify that the vertex \( \alpha \) has already been processed, it is kept as processed in the position set. Otherwise, it is added as unprocessed.

The following data structures are manipulated during the algorithm’s execution:

I. A set of trace items, iteratively computed by the algorithm; 
II. A data graph containing the original data graph \( D \) incrementally augmented with new, non-terminal-labeled edges.

Let us now present our algorithm for processing context-free path queries (Algorithm 1). Our technique is based on the idea of building and updating a set of trace items. The input parameters of the algorithm are:

- A context-free grammar \( G = (N, \Sigma, P, S) \), defined by the user;
- An input graph \( G = V \times \Sigma \times V \) with edges restricted to the grammar alphabet;
- A set of query pairs \( Q \subseteq V \times N \). Each pair of the query set indicates a start vertex and non-terminal symbol used for recognizing paths.

Our algorithm uses the \( \sqcup \) operator to perform unions between sets of marked and unmarked vertices. This operator is defined as follows: given the position sets \( C \) and \( \{ x^o \} \), the union between them is defined as:

\[
C \sqcup \{ x^o \} = \begin{cases} 
C, & \text{if } x^o \in C \\
C \cup \{ x^o \}, & \text{otherwise}
\end{cases}
\]

That is, if the vertex \( x \) has already been processed, it is kept as processed in the position set. Otherwise, it is added as unprocessed.

### Algorithm 1: The Trace Item-based Algorithm

**Input:** \( G = (N, \Sigma, P, S), Q \subseteq V \times N, D = V \times \Sigma \times V \)  
**Output:** \( D' \subseteq V \times \Sigma \times V \)

```
function eval

I := \{ \{ A \to \{ w^o \} \alpha_1 \{ \} \ldots \alpha_n \{ \} \} | A \to \alpha_1 \ldots \alpha_n \in P \land (w, A) \in Q \}

D' := D

while \exists i, x s.t. i = \{ A \to \ldots \{ x^o, \ldots, \} \} \in I do

switch i

case i := \{ A \to \ldots \{ x^o, \ldots, \} C_k \ldots \} do

if \( \alpha_k \in \Sigma \lor \alpha_k \to \{ x \} . \ldots \} \in I \) then

\( C_k := C_k \sqcup \{ y^o \mid (x, \alpha_k, y) \in D' \} \)

else

\( I := I \cup \{ (\alpha_k \to \{ \} \ldots \} \mid \alpha_k \to \beta_1 \ldots \beta_n \in P \} \)

\( case i := \{ A \to \{ w \} \ldots \{ \} \} \in I do \)

\( D' := D' \cup \{ (w, A, x) \} \)

\( foreach \{ B \to \ldots \{ w^o \} \ldots \} \in I do \)

\( C := C \sqcup \{ x^o \} \)

mark(x, i)

return D'
```

Lines 2-3 initialize \( I \) and \( D' \). For each pair \( (w, A) \in Q \) and rule \( A \to \alpha_1 \ldots \alpha_n \in P \), the set \( I \) is initialized with trace items \( \{ A \to \{ w^o \} \alpha_1 \{ \} \ldots \alpha_n \{ \} \} \). The graph \( D' \) is initialized as a copy of the input graph \( D \). These steps prepare the algorithm to enter the main loop that processes unmarked vertices in items of \( I \). The main loop concludes when there are no such unmarked vertices.

The processing of unmarked vertices is divided into two cases:

I. In the first case (lines 6-10), given the trace item \( i = \{ A \to \alpha_1 \ldots \alpha_n \} \) and rule \( A \to \alpha_1 \ldots \alpha_n \in P \), we add all the vertices \( y^o \) such that there exists an edge \( (x, \alpha_k, y) \in D' \) (line 8).

II. In the second case of the main loop (lines 11 to 14) we identify that the vertex \( x \) belongs to the last position set \( C_{k-1} \) that is not the last position set of the item.

(1) If \( \alpha_k \in \Sigma \), we add to \( C_k \) all the vertices \( y^o \) such that there exists an edge \( (x, \alpha_k, y) \in D' \) (line 8).

(2) If \( \alpha_k \in N \) and \( \alpha_k \to \{ x \} . \ldots \} \in I \), we add all \( y^o \) to \( C_k \) such that there is an edge \( (x, \alpha_k, y) \in D' \) (this case is also treated by line 8).

(3) If \( \alpha_k \in N \) and there is no trace item \( \alpha_k \to \{ x \} . \ldots \} \), our algorithm initiates the search for \( \alpha_k \)-derivations beginning at \( x \). This is done by creating new trace items \( \alpha_k \to \{ x^o \} . \ldots \} \) and adding them to \( I \) (line 10).

We highlight three important features of Algorithm 1:
Directed path recognition – Trace items are computed in left-to-right order, starting at a given vertex. This allows Algorithm 1 to process only a reduced part of the graph if a subset of start vertices is provided. Moreover, since our approach works in a top-down fashion, as soon as a path fails to be recognized, it is put aside, avoiding the processing of many useless edges.

Use of grammars in any form – Many CFPQ evaluation engines [4, 7, 8] depend on the grammar being in Chomsky Normal Form (CNF). Knowing the form of the rules in the grammar simplifies processing and reduces runtime complexity. As a drawback, grammars converted to such normal form grow in number of rules, what may negatively impact the performance of those algorithms. Also, as the grammar has changed, it is harder to debug queries.

Path reconstruction – Trace items contain information for reconstructing paths from the original graph. This is useful, for instance, to extract all paths or the shortest one linking two given vertices, thus providing richer information than just reachable vertices.

In the next section we analyze the behaviour of our algorithm in terms of correctness, as well as runtime and memory complexity.

3.2 Correctness and Complexity

We start this section by presenting evidence that the proposed algorithm is correct. Afterward, we state runtime and space complexities. Due to space limitations, the formal proofs are provided in our online appendix[3].

Proposition 3.6. Let \( G = (N, \Sigma, P, S) \) be a grammar, \( D \subseteq V \times E \times V \) a data graph and a query pair \((w, A) \) \( \in Q \). Given \( A \rightarrow \{w|\alpha_1 C_1 \ldots \alpha_j C_j \ldots \} \in 1 \) computed by Algorithm 1, then for any vertex \( x \in V \) we have \( x \in C_i \iff x \in C_{G,D}(w, \alpha_1 \ldots \alpha_j) \).

The result graph \( D' \) is only updated at line 3, where it just copies the input graph \( D \), and at line 12, where it is increased with a new edge \((w, A, x)\) where \( w, A \) and \( x \) come from the generalized item \( i = A \rightarrow \{w|^*\} \ldots \{x|^*\} \ldots \). By Definition 3.1.2 we can conclude that line 3 is a valid step; however, for line 12 it depends on whether the generalized items \( i \in I \) were constructed correctly. That leads us to the next proposition.

Proposition 3.7. Algorithm 1 computes \( D' \) such that for all \((x, A) \in Q \) we have \( \forall y \in G_{D}(x, A) \iff (x, A, y) \in D' \).

Proof. This follows from Propositions 3.2 and 3.6. \( \square \)

As for the runtime and space complexity, we present the following propositions:

Proposition 3.8 (Worst-case Space Complexity). The worst-case space complexity of Algorithm 1 is \( O(|V|^2 \cdot |P| \cdot l) \).

Proposition 3.9 (Worst-case Runtime Complexity). The worst-case runtime complexity of Algorithm 1 is \( O(|V|^3 \cdot |P|^2 \cdot l^2) \).

The proof for these propositions is based on the finite number of elements in the sets the algorithm manipulates, which is proportional to \( |V|^2 \). For each element in those sets, the algorithm may run at most \(|V|\) operations. Therefore, we have quadratic and cubic space and runtime complexity, respectively.

4 RELATED WORK

Context-free path queries have received attention by the scientific community in the last few years.

In [7], the authors propose an algorithm to evaluate Context-Free Path Queries based on Earley’s and CYK parsing techniques. This algorithm receives a grammar in CNF and a data graph. The algorithm adds non-terminal-labelled edges to link nodes that are connected by a path generated by the non-terminal symbol. Regardless of the query, the algorithm processes the whole graph. After this step, atomic queries can be executed in constant time. The algorithm is \( \mathcal{O}(|N||E| + (|N||V|)^2) \), where \( N \) is the set of non-terminal symbols, \( V \) is the set of nodes of the graph and \( E \) is the set of edges.

The authors in [8] study two path-based semantics and provide algorithms for implementing them. For each pair of vertices connected by a path accepted by the given context-free grammar, evaluating a CFPQ under the all-paths semantics implies providing all paths linking them, whilst the single-path semantics implies providing only one of those paths. The algorithm implementing the all-paths semantics works by annotating grammar productions with graph vertices. It processes the entire graph for answering queries and has worst-case runtime complexity of \( \mathcal{O}(|P'_G| + |P'_X| + |E| + (|N||V|)^2) \), where \( P'_G \) is the set of empty production rules, \( P'_X \) is the set of unitary (terminal) production rules, \( E \) is the set of edges in the graph, \( N \) is the set of non-terminal symbols in the grammar and \( V \) is the set of vertices. The authors provide an algorithm for producing paths given an annotated grammar whose complexity is linear in terms of the length of the paths produced. Combining this algorithm with others that derive shortest strings or strings limited to a certain length, the authors implement the single-path semantics.

The authors in [6] propose an LL-based approach to recognize context-free paths in RDF graphs. The proposal uses the PLL [14] parsing technique to define an algorithm for querying data graphs with runtime complexity of \( \mathcal{O}(|V|^2 \max_{v \in V}(\text{deg}^+(v))) \), where \( V \) is the set of vertices and \( \text{deg}^+(v) \) is the outdegree of vertex \( v \). Notice that for complete graphs this runtime complexity reaches \( \mathcal{O}(|V|^4) \).

The algorithm in [4] uses a matrix representation of the graph where each cell contains the set of edges between two vertices, represented by line and column. The proposal is based on Valiant’s parsing algorithm [18]. Similarly to [7], the algorithm in [4] calculates all possible non-terminal labelled edges between nodes of the graph. The time complexity is \( \mathcal{O}(|V|^2 |N|^3 (\text{BMM}(V)) + \text{BMU}(V))) \), where \( V \) is the set of vertices, \( N \) is the set of non-terminal symbols and \( \text{BMM} \) and \( \text{BMU} \) refer to the number of elementary operations needed in the matrix multiplication. A subsequent paper [12] compares the performance of different implementations for this algorithm. Their results show the efficacy of using GPU for matrix multiplication.

The query algorithm in [13] is based on the LALR parsing [2]. The proposal extends Tomita’s algorithm and GSS data structure [17] to simultaneously discover context-free paths on a data graph. The proposed algorithm does not need to pre-process the whole graph to answer the query. The time complexity of this algorithm is given by \( \mathcal{O}(|V|^{4+k} \cdot |I|^{2+k}) \cdot (|E| \cdot |N|) \), where \( k \) is the maximum size of the right-hand side of the production rules in the grammar and \( l \) is the number of lines of the LALR(1) parsing table.
In [10], the authors propose a query processing algorithm based on the LL parsing technique [2]. For queries of the form \((x, S)\), where \(x\) is a vertex of the graph and \(S\) is a non-terminal symbol, the algorithm proceeds in a top-down manner, trying to discover \(S\)-generated paths from \(x\). The worst case runtime complexity of their algorithm is \(O(|V|^3 \cdot |P|)\), where \(P\) is the set of production rules of the grammar.

The authors in [9] compare the CFPQ evaluation methods in [4, 8, 13]. The authors perform experiments with several data sets, including real and synthetic ones. These methods were implemented in Java, using Neo4j’s graph store API to represent data and running a single execution thread. Because of that, the matrix-based algorithm, which should make use of the GPU for high performance computation, does not present a very competitive execution time in some cases. The paper focuses on the scalability of the three approaches and concludes that these methods are not yet adequate for processing large amounts of data. We expect to contribute towards that goal.

A novel approach [15] reduces CFPQ evaluation to solving systems of equations over real numbers. For linear grammars, the evaluation complexity is \(O((|V| + 2|o|)^2)\), where \(2|o|\) is the complexity of best algorithms for \(n \times n\)-matrix multiplication. For non-linear grammars, the author provides a solution based on Newton’s method for approximating the roots of non-linear functions. However, they do not provide a precise complexity for this solution.

Another recent work [16] presents a matrix-based CFPQ evaluation algorithm implementing single-path semantics. The authors introduce the concept of path indexes, which store information for extracting paths after processing. The algorithm is based on [4], having the same runtime complexity.

Our algorithm combines many positive aspects present in related works, including the capability of dealing with arbitrary context-free grammars, and the possibility of restricting processing to subgraphs, while maintaining the lowest complexity levels. In the next section, we demonstrate the applicability of our algorithm using the same experiments as in other works.

Table 2 summarizes the most important features of the Context-Free Path Query evaluation algorithms cited above.

| Algorithm          | Runtime     | Space     |
|--------------------|-------------|-----------|
| CYK-Earley’s [7]   | \(O(|V|^3)\) | \(O(|V|^3)\) |
| Annotated Grammar  [8] | \(O(|V|^3)\) | \(O(|V|^3)\) |
| GLL-based [6]      | \(O(|V|^3)\) | \(O(|V|^3)\) |
| Matrix [4]         | \(O(|V|^2 \cdot |N|^2)\) | \(O(|V|^2)\) |
| GSSLR [13]         | \(O(|V|^2 \cdot |P|^2 \cdot |K| \cdot |N|^2)\) | \(O(|V|^2)\) |
| LL [10]            | \(O(|V|^2 \cdot |P|^2)\) | \(O(|V|^2)\) |

Table 2: State-of-the-Art CFPQ Evaluation Techniques.

5 EXPERIMENTS

In this section we present some performance experiments to investigate the viability of our algorithm. We implemented a prototype using the Go programming language\(^1\) and compared it with the proposal in [10], implemented in Python. The experiments were performed on a Debian 8.11, 64GB RAM, Intel Xeon E312xx (Sandy Bridge) @ 2.195GHz, 64 bits. The results presented here are the average time and memory of 10 runs.

We performed the experiments described in [4, 6, 8–10, 19]. The queries used are defined as \(Q = \{(x, S) \mid x \in V\}\), where \(V\) is the set of vertices in the input graph and \(S\) is the start symbol of a grammar. This query makes the algorithm start at every vertex in the graph and follow all \(S\)-derivable paths to find the destination vertices. The grammars used are summarized in Table 3.

| #  | P | Description                                      |
|----|---|-------------------------------------------------|
| 1  | \(|S \rightarrow a S b \mid \epsilon\)| (Ambiguous) Generates balanced pairs of \(a\)'s and \(b\)'s [4, 6, 10, 19] |
| 2  | \(|S \rightarrow a S b S \mid \epsilon\)| Unambiguous grammar generating the same language of \(G_1\) [4, 6, 10, 19] |
| 3  | \(|A \rightarrow A A \mid a\)| Dense grammar recognizing the language \(a^+\) [8] |
| 4  | \(|B \rightarrow B A \mid A B \mid \epsilon, A \rightarrow a\)| Sparse grammar recognizing the language \(a^*\) [8] |
| 5  | \(|S \rightarrow sc S sc^{-1} \mid tS t^{-1}\)| Retrieves concepts on the same level of hierarchy of classes [4, 6, 10, 19] |
| 6  | \(|G \rightarrow B sc^{-1}, B \rightarrow sc B sc^{-1} \mid \epsilon\)| Retrieves concepts on adjacent levels of the hierarchy of classes [4, 6, 10, 19] |
| 7  | \(|G \rightarrow bt S bt^{-1}\)| Retrieves concepts on adjacent levels of hierarchy of species [9] |
| 8  | \(|G \rightarrow a S d \mid a X d, X \rightarrow b X c \mid \epsilon\)| Defines the language \(a^n b^m c^m d^n\) [9] |

Table 3: Summary of grammars used in the experiments.

The databases used in the experiments include both synthetic graphs and publicly available RDF ontologies. The synthetic graphs used in the experiments are described as follows. Graphs that have the form of a single straight path are called linear graphs. There are two kinds of them: the ones referred to as \(ab\)-lists [10], whose labels form an \(a^n b^m\) path; the ones called \(a\)-string graphs [8], that are linear graphs such that all their edges are labelled with the terminal \(a\). Graphs that form a circular path are called cycle graphs [8], and their edges are labeled with the terminal \(a\).

5.1 Synthetic Graphs

The synthetic graphs explore specific characteristics of the evaluation mechanisms: memory and runtime performance in worst-case or random scenarios; the influence of grammar ambiguity or density/sparsity; and scalability properties of our approach.

Dealing with Ambiguity in Grammars: The data presented in Figure 3a corresponds to the execution over \(ab\)-lists [10]. We used Grammars \(G_1\) and \(G_2\), which recognize the language of balanced \(a\)'s and \(b\)'s. We observe that our algorithm (labelled TI) presents very efficient performance, nearly following a linear behavior in this experiment. We also observe that it is not heavily affected by the grammar’s ambiguity. Our algorithm compares favorably with [10] (labelled LL) both in terms of memory and runtime.

Dense and Sparse Grammars: Figures 3b and 3c show the performance of our prototype over cycle and path graphs [8], respectively.

\(^1\)https://golang.org/
using Grammars $G_3$ and $G_4$. We observe that our algorithm presents very good performance when compared to [10], which does not scale well and quickly becomes impractical for larger graphs.

It is also important to remark that the form of the grammar’s production rules have an important influence on the time performance of the algorithms. For $a$-string and cycle graphs, sparse grammar $G_4$ has an advantage over the dense grammar $G_3$.

### 5.2 RDF Ontologies

For the next experiment we used a set of popular ontologies publicly available on the internet. This dataset and the grammars are the same used in previous works [4, 6, 10, 19]. The “geospecies” database and Grammar $G_7$ were used in [9].

Grammar $G_5$ retrieves concepts in the same level of the RDFS’ `subClassOf`/`type` hierarchy. The experiment consists of performing a same generation query [1]. For each vertex of the graph, the query looks for all vertices that are at the same level in the graph of the subclass/type hierarchy. Grammar $G_6$ retrieves concepts in adjacent levels of the RDFS’ `subClassOf` hierarchy.

Grammar $G_7$ retrieves concepts in the same level of the broader Transitive hierarchy. These edges are directed from child to parent, relating categories of species, families, orders, etc. This is a real example of application, where a CFPQ is used to identify the pairs relating categories of species, families, orders, etc. This is a real example of application, where a CFPQ is used to identify the pairs of vertices already in the graph. The edge labels are randomly chosen, being either $a$, $b$, $c$ or $d$. The probability for a vertex to be chosen is directly proportional to its degree at that moment, such that the higher the degree of the vertex, the higher is its probability of receiving new edges. The graphs used in this experiment as well as the algorithm’s runtime and amount of memory used are presented in Table 5.

As in the previous experiment, our algorithm (TI) compares favorably with that in [10]. The runtime numbers for TI are consistently better than those for LL. Memory consumption for TI and LL are in the same magnitude order. Notice that scalability in TI is better than in LL. This is evidenced by the existence of some cases that could not be treated by LL.

### 6 FINAL REMARKS

We presented an algorithm for the evaluation of context-free path queries for RDF databases, analysed its correctness, as well as its worst-case runtime and space complexity. We validated our work by using both synthetic and real-life examples, showing that our prototype outperforms another, recently published algorithm.

If we consider the runtimes presented in [9], we can see that our algorithm performs well for all databases, outperforming LL in most cases, both for runtime and memory consumption. Notice that our algorithm is now capable of dealing with the database “geospecies”, a large graph that could not be managed by LL. Thus, showing in practice that TI improved scalability when compared to LL.

### 5.3 Random Graphs

The next experiments were proposed by [9] and use random, synthetic graphs. We used a graph generator function based on the definition given by [3]. Given the size of the graph in number of vertices $n$ and a constant $k \leq n$, the generator function, denoted by $\mathcal{S}(n, k)$, starts with a clique of $k$ vertices. For each $v$ in the $n-k$ remaining vertices, the generator adds $k$ edges from $v$ to any vertices already in the graph. The edge labels are randomly chosen, being either $a$, $b$, $c$ or $d$. The probability for a vertex to be chosen is directly proportional to its degree at that moment, such that the higher the degree of the vertex, the higher is its probability of receiving new edges. The graphs used in this experiment as well as the algorithm’s runtime and amount of memory used are presented in Table 5.

In the data presented in Table 4, we can observe that our algorithm performs well for all databases, outperforming LL in most cases, both for runtime and memory consumption. Notice that our algorithm is now capable of dealing with the database “geospecies”, a large graph that could not be managed by LL. Thus, showing in practice that TI improved scalability when compared to LL.
being our memory consumption about 21% higher. Notice that these measures must be considered with caution, since different platforms were used in the experiments.

As future work, we intend to investigate a parallel version of our algorithm. This may improve its performance, since the treatment of unmarked vertices in position sets may be done in parallel. A better management of large graphs that do not fit in the memory is another desired improvement on our implementation.

We intend to work on the definition of benchmarking data sets for algorithms for evaluation of CFPQs. This will make possible to have more accurate data to compare the different algorithms that implement those queries. This benchmark should be based on real-life problems such as those used by [5, 11, 20].

7 ACKNOWLEDGEMENTS

This work is partly supported by INES grant CNPq/465614/2014-0. Work financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brasil (CAPES), Finance Code 001.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. 1995. Foundations of Databases. Addison-Wesley. https://books.google.com.br/books?id=H9QAAAAMAAJ
[2] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. 2007. Compilers: Principles, Techniques, and Tools. ADDISON WESLEY Publishing Company Incorporated. https://books.google.com.br/books?id=WnBPgAACAAJ
[3] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. Rev. Mod. Phys. 74 (Jan 2002), 47–97. Issue 1. https://doi.org/10.1103/RevModPhys.74.47
[4] Rustam Azimov and Semyon Grigorev. 2018. Context-Free Path Querying by Matrix Multiplication. In Proc. of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). 1–5.
[5] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In Web Engineering, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.
[6] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. Electronic Notes in Theoretical Computer Science 253, 7 (2010), 177 – 189. https://doi.org/10.1016/j.entcs.2010.08.041
[7] Jelle Hellings. 2014. Conjunctive Context-Free Path Queries. In Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 119–130. https://doi.org/10.5441/002/icdt.2014.15
[8] Jelle Hellings. 2015. Path Results for Context-Free Grammar Queries on Graphs. CoRR abs/1502.02242 (2015).
[9] Jochem Kuipers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In Proceedings of the 31st International Conference on Scientific and Statistical Database Management. ACM, 121–132.
[10] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2019. LL-based query answering over RDF databases. Journal of Computer Languages 51 (2019), 75 – 87. https://doi.org/10.1016/j.jola.2019.02.002
[11] Hui Mao and Anmol Deshpande. 2019. Understanding data science lifecycle provenance via graph segmentation and summarization. In 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 1710–1713.
[12] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbachev, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). 1–5.
[13] Masaru Tomita. 1985. Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell, MA, USA.
[14] Leslie G. Valiant. 1975. General Context-Free Recognition in Less Than Cubic Time. J. Comput. Syst. Sci. 10, 2 (1975), 308–315. https://doi.org/10.1016/S0022-0000(75)80046-8
[15] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In International Semantic Web Conference (1) (Lecture Notes in Computer Science), Vol. 9981. Springer, 632–648.
[16] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 197–208.