A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases

Fred C. Santos, Umberto S. Costa, and Martin A. Musicante(✉)

Computer Science Department (DIMAp),
Federal University of Rio Grande do Norte, Natal, Brazil
freddcs@ppgsc.ufrn.br, {umberto,mam}@dimap.ufrn.br

Abstract. Many computing applications require processing of data that are directly collected from the Internet. In this context, the use of the Resource Description Framework (RDF) has became a common feature. The query and analysis of RDF data is paramount to explore the full potential of the data available on the Web. Query languages for RDF graph databases rely on the use of regular expressions to identify paths over the data. Some interesting queries, such as same-generation queries, cannot be expressed by regular expressions. We are interested in extending the expressiveness of queries over graph databases by using paths defined by context-free grammars. We introduce a new query algorithm to process context-free path queries over graph databases. Our approach is inspired by the LR(1) parsing techniques. A prototype was implemented and experiments were conducted to validate and compare the results of our algorithm with those obtained by similar approaches.

Keywords: Graph databases · Query answering
Context-free path queries

1 Introduction

Many computing applications require processing of data that are directly collected from the Internet. In this context, the use of the Resource Description Framework (RDF [3]) has became common. RDF documents define a set of triples (subject, predicate, object). The components of these triples are text or URIs used to identify resources over the Internet. These sets of triples may be seen as a graph database, where the predicate of each triple corresponds to an edge linking the subject and object. This format is well suited to represent data related to applications such as social networks or IoT [12].

The current standard to query graph databases is SPARQL [1]. The expressiveness of SPARQL is limited by the use of regular expressions to define query paths (called *Property Paths* in SPARQL). There are interesting queries that cannot be expressed by property paths. Examples of these are the *same-generation queries* [4], where the predicates linking nodes in the query to nodes in the

T. Mikkonen et al. (Eds.): ICWE 2018, LNCS 10845, pp. 225–233, 2018.
https://doi.org/10.1007/978-3-319-91662-0_17

answer is a string belonging to a context-free language. Some ongoing research initiatives define context-free path queries [7–9,13].

In this work, we contribute to the area by investigating how LR(1) parsing techniques [5] can be adapted to perform this class of queries. The main contributions of this paper are: (*i*) the design of an algorithm based on LR(1) parsing to recognize context-free paths in graphs; and (*ii*) the performance evaluation of our algorithm by means of experiments over ontologies and synthetic graphs.

This paper is organized as follows: Sect. 2 presents the concepts and definitions used in this work, a well as introduces our query evaluation algorithm; Sect. 3 is devoted to our experiments; and Sect. 4 gives some final remarks.

2 LR Queries over Graph Databases

A database is a collection of organized, related data which is used as a source to answer user queries or to facilitate other data processing activities. The basic database management problem is how to store and organize data efficiently to meet the data processing needs of the applications which use the data [4].

Graph databases are usually represented by using RDF (Resource Description Framework) [2], a W3C standard. Querying a graph database consists on looking for nodes of the graph under some search criteria. The standard query language for RDF databases is SPARQL [1], an SQL-inspired declarative query language. SPARQL allows the user to define paths inside the graph by using property paths. Evaluating a given property path R consists in finding all the pairs of nodes linked by paths belonging to the regular language generated by R.

Formally, a *Graph Database* is a set of triples in $V \times E \times V$, where V is a set of nodes and E is a set of edge labels. Given an initial node, the query processor looks for paths in the graph database. A path linking the nodes n and m of a graph database is defined as strings of edges linking n to m. In our context, these strings belong to the language generated by a context-free grammar [5]. Given a graph database D and a grammar G, a query Q_G is defined as a set of nodes of D. The nodes in Q_G will be used as starting points of G-generated paths in the graph database D. For any node n in Q_G, the evaluation of the query will look for those nodes m in the graph that are (*i*) linked to n by a path in D and (*ii*) the sequence of edges of the path forms a string generated by the grammar G. The answer to Q_G will be all the nodes of the graph D reachable by such paths.

LR parsing [5] is a well-known technique capable of parsing most current programming languages. Given a context-free grammar, LR parsers perform a reverse derivation for the input string. Intuitively, the parser *shifts* states (terminal symbols) onto a stack until forming the right-hand side of a production rule. Once a right-hand side is found, it is replaced (on the stack) by a state corresponding to the non-terminal symbol of the left-hand side of the rule. This operation is called *reduce*.

Tomita-Style Generalised LR Parsers. In [10,11], a bottom-up parsing algorithm is presented for general context-free grammars. The LR parsing process uses a

data structure called *Graph Structured Stack* (GSS) to represent several stacks. A GSS is a compact data structure that keeps track of multiple derivations without processing any part of the input twice. A GSS is formed by state nodes (states of the LR automaton) and symbol nodes (containing terminals and nonterminals of the grammar). State nodes are grouped into levels and the parser processes one level at each iteration. An initial level U_0 contains just the initial state of the automaton. Each subsequent level will be constructed by processing *shift* actions. Given a state node s_i at level U_k, the action "*shift* s_j" for a given input symbol a will create a new state node s_j at the GSS level U_{k+1} (Fig. 1).

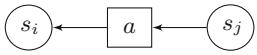

Fig. 1. Representation of a *shift* action in a GSS.

For any state node s_j at level U_k, a *reduction* by the rule $A \rightarrow \alpha$ at state s_j adds a new state node s_h at the same U_k level. This new node is connected to the ancestor s_i of s_j, such that, the distance between s_i and s_j in the GSS is $|\alpha|$ (the size of the right-hand side of the production rule). In this way, the state node s_j is located at level $U_{k-|\alpha|}$ (Fig. 2).

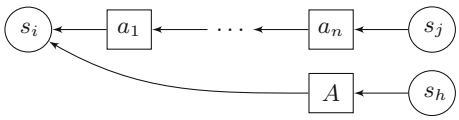

Fig. 2. Representation of a *reduce* action in a GSS.

The Proposed Method: GSSLR. Our method receives a data graph DG, a Context-Free Grammar G, a query Q of nodes of DG and calculates the set $Answers_G(Q)$. $Answers_G(Q)$ contains those nodes of DG that can be reached from nodes in Q through a path which is a word of the language generated by G. Our method is described by Algorithm 1 and uses a variant of GSS to encompass several derivations at a time: Our GSS contains *sd-nodes*, which associate states of LR automata with nodes of data graphs. First, the algorithm creates a parsing table for G (line 2) and initializes the GSS (line 3). The state s_0 is the start state of the LR automaton for G. The level U_0 of the GSS includes all the sd-nodes in $\{(a, s_0) \mid a \in Q\}$. The algorithm iterates over three main sets, namely *VisitedPairs*, *ReductionEdges* and *Answers* (defined at lines 4, 5 and 6 respectively):

- *VisitedPairs* contains the nodes of the GSS during the parsing.
- *ReductionEdges* is a set of triples (a, A, b), where a and b are vertices of the graph and A is a non-terminal symbol. These triples represent the existence of an A-generated path from vertex a to vertex b in the data graph. This set is produced by the algorithm during the parsing.
- *Answers* is the result of the query evaluation.

The main loop of Algorithm 1 (lines 8 to 43) iterates over (new) levels of the GSS. The loop exits when no new data is included in any of the sets *VisitedPairs* or *ReductionEdges*. Notice that the body of this main loop has three main parts, corresponding to the possible actions of the LR parser.

Lines 10 to 25 process all the possible reduce actions over the sd-nodes at the current GSS level. For each sd-node (a, s_i), and triple (a, t, b) in DG, the algorithm looks for reductions by t at state s_i. For each possible reduction, a new sd-node is created at the current level of the GSS. The new sd-node is linked to ancestor sd-nodes of the GSS, depending on the length of the right-hand side of the production rule used in the reduction. Each reduction may add a new triple to *ReductionEdges*. Intuitively, changes on *ReductionEdges* mean that the parser may discover new paths originated from derivations depending on the new reduction. This indicates that a new iteration needs to be performed.

Lines 26 to 32 produce new answers. Processing a query for a given origin vertex a, is to find all the paths $a\pi b$ generated by the grammar G. Any vertex of DG reachable at an accepting state of the parser is an answer to the query.

Lines 33 to 41 create the next level of the GSS. For each sd-node (a, s_i) at the current GSS level, the algorithm looks for shift actions at state s_i and outgoing edges of a in DG, creating new nodes at the next level of the GSS.

3 Experimental Results

In order to validate our proposal, we produced a prototype in Python. Our prototype was executed on an AMD Phenom II X4 B97 processor, with 7.3 GB of RAM, running Ubuntu 16.04 (x64) and Python 2.7. We also benefit of the speed gains provided by the PyPy Python compiler [6], which uses Just-in-Time (JIT) compiling techniques. The times reported are the average of five runs. Two experiments were performed. The first one evaluates the feasibility and efficacy of our method by comparing our results to those obtained in the related work [7,9,13]. The second experiment evaluates the scalability of our approach and used synthetic data to compare our results with those of [9].

$S \rightarrow subClassOf\ subClassOf^{-1}$
$S \rightarrow subClassOf\ S\ subClassOf^{-1}$
$S \rightarrow type\ type^{-1}$
$S \rightarrow type\ S\ type^{-1}$

(*a*)

$S \rightarrow B\ subClassOf^{-1}$
$B \rightarrow subClassOf\ B\ subClassOf^{-1}$
$B \rightarrow \lambda$

(*b*)

Fig. 3. Grammars for Queries Q_1 (*a*) and Q_2 (*b*).

Experiment 1: In the first experiment, two queries over a number of popular RDF ontologies are considered. Query Q_1 looks for all pairs of nodes at the same sub-class level of each ontology. Query Q_2 returns all pairs of nodes on adjacent sub-class levels. The grammar rules for queries Q_1 and Q_2 are given in Fig. 3.

Algorithm 1. *GSSLR* Query Processing Algorithm.

```
   input  : - a data graph DG ⊆ V × E × V;
            - a Context-Free Grammar G = (N, T, S, P);
            - a Context-Free Path Query Q ⊆ V.
   output: - Answers_G(Q).
 1 Function GrLR(DG, G, Q) : Answers_G(Q)
 2     (ParsingTable, s_0) ←CreateParsingTable(G)
 3     GSS ←CreateGSS(Q, s_0)
 4     VisitedPairs ← ∅
 5     ReductionEdges ← ∅
 6     Answers ← ∅
 7     level ← 0
 8     while TRUE do
 9         changed ←FALSE
           // processing reduces
10         PairsToProcess ←GSS_Pairs(GSS, level)
11         while PairsToProcess ≠ ∅ do
12             choose (a, s_i) ∈ PairsToProcess
13             PairsToProcess ← PairsToProcess\{(a, s_i)}
14             NextTerminals ← {terminal|(a, terminal, b) ∈ DG} ∪ {$}
15             for each terminal ∈ NextTerminals do
16                 for each ParsingTable[s_i][terminal] do
17                     if ParsingTable[s_i][terminal] = REDUCE A → α then
18                         Ancestors ← GSS_Up(GSS, (a, s_i), |α|)
19                         for each ((c, s_j) ∈ Ancestors) do
20                             GSSPair ← (a, ParsingTable[s_j][N])
21                             GSS_Insert_Pair(GSS, level, GSSPair)
22                             PairsToProcess ← PairsToProcess ∪ {GSSPair}
23                             if (c, N, a) ∉ ReductionEdges then
24                                 ReductionEdges ← ReductionEdges ∪ {(c, A, a)}
25                                 changed ← TRUE
           // processing accept states
26         for each (a, s_i) ∈ GSS_Pairs(GSS, level) do
27             if ParsingTable[s_i][$] = ACCEPT then
28                 Ancestors ← GSS_Up(GSS, (a, s_i), 1)
29                 for each (c, s_j) ∈ Ancestors do
30                     if (c, a) ∉ Answers then
31                         Answers ← Answers ∪ {(c, S, a)}
32                         changed ← TRUE
           // processing shifts
33         for each (a, s_i) ∈ GSS_Pairs(GSS, level) do
34             for each (a, terminal, b) ∈ DG do
35                 for each ParsingTable[s_i][terminal] do
36                     if ParsingTable[s_i][terminal] = SHIFT s_j then
37                         GSSPair ← (b, s_j)
38                         GSS_Insert_Pair(GSS, level + 1, GSSPair)
39                         if (b, s_j) ∉ VisitedPairs then
40                             VisitedPairs ← VisitedPairs ∪ {(b, s_j)}
41                             changed ← TRUE
           // has VisitedPairs or ReductionEdges changed at this level?
42         if not (changed) then break;
43         level ← level + 1
44     return Answers
```

As it is usual in RDF, the superscript "−1" indicates the inversed transversal of an edge of the graph. The RDF ontologies used in the experiments are *Skos*, *Generations*, *Travel*, *Univ-bench*, *Foaf*, *People-pets*, *Funding*, *Atom-primitive*, *Biomedical*, *Pizza* and *Wine*. These databases are used by [7,9,13] to perform the same experiment as ours. These ontologies can be found on the Web.

In Fig. 4 we compare our results to those in [7,9,13] for query Q_1. All the data correspond to average running times in milliseconds, as presented on those papers. GSSLR refers to our implementation.

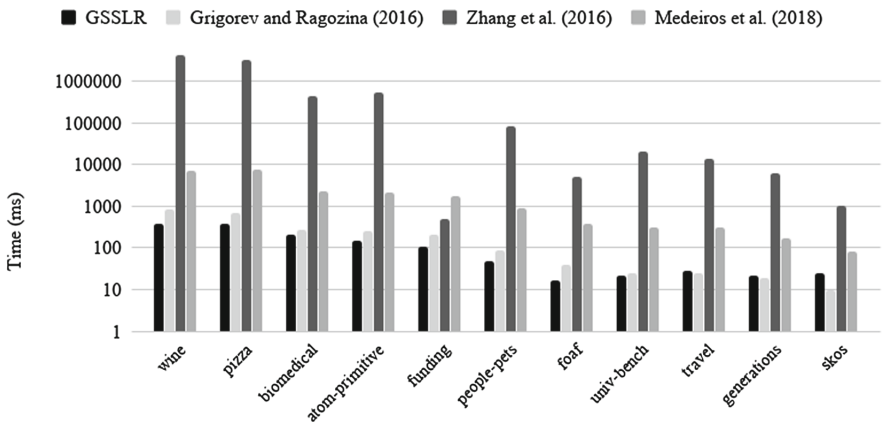

Fig. 4. Performance evaluation for Query Q_1 on RDF databases.

Due to to the differences in running environments, the comparison between these four implementations is not a fair one. However, it is useful as a general performance indication. The experiments of [13] were executed under Windows 7 on a Intel Core i5-760 2.80 GHz with 6 GB of RAM. Their implementation was written in Java 7. The results of [9] were obtained in a similar architecture as ours, but without the use of the JIT Python compiler. The running environment of [7] was an Intel Core i7-4790, 3.60 GHz CPU machine with 32 GB RAM running Windows 10 Pro. Their implementation was written in F#.

Figure 5 presents the results for query Q_2, under the same considerations as before. The comparison with the results obtained in [13] shows the expressive gain of our implementation when compared to theirs. The comparison of our implementation with that of [9] shows that our algorithm outperforms their implementation for the given ontologies and queries.

The results presented in [7] are close to ours. It is worth to notice that for both queries, our implementation is faster than [7] in all but three ontologies, even with a noticeable disadvantage in processing power (due to the hardware used in both experiments). This is an indication of the feasibility of our approach.

Experiment 2: This experiment consists on querying a number of synthetic graphs. The queries Q_3 and Q_4 are defined over the grammars presented in [7],

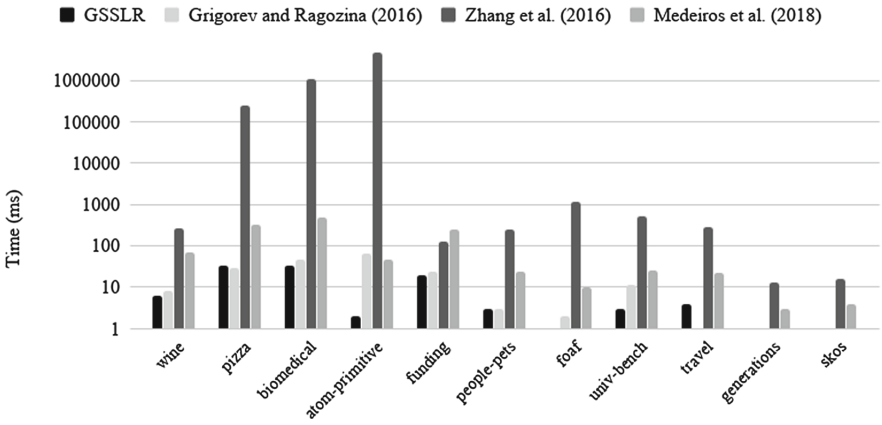

Fig. 5. Performance evaluation for Query Q_2 on RDF databases.

$$S \to \lambda$$
$$S \to a\ S\ b$$
$$S \to S\ S$$
$$(a)$$

$$S \to a\ S\ b\ S$$
$$S \to \lambda$$
$$(b)$$

Fig. 6. Grammars for queries Q_3 and Q_4.

which are reproduced in Fig. 6. Both grammars generate the same language, being grammar (a) ambiguous, while grammar (b) is unambiguous and LR(1).

This experiment uses graphs with the topology of complete binary trees. There are two terminals, a and b, linking data nodes. Query Q_3 consists on finding all the nodes linked to other nodes of the graph by paths generated by derivations of the symbol S of the grammar in Fig. 6(a). Query Q_4 is the same as Q_3, but considering the grammar in Fig. 6(b). As expected, both queries produce the same results.

In order to evaluate the scalability of our algorithm, both queries are evaluated on datasets of increasing size (in terms of the height of the binary trees). In Fig. 7 we compare our results with those of [9]. Notice that for the ambiguous grammar our results are close to those of [9]. For the unambiguous grammar, our algorithm presents a significant gain in performance. We can explain this gain by the fact that the number of derivations to be represented in the GSS directly impacts on the performance of our algorithm.

The overall results of the experiments suggest that it is viable to use our proposed algorithm to perform context-free queries on graphs on most of the real world scenarios. Due to the differences in the hardware used to execute the experiments, we cannot directly compare our results with the other proposals.

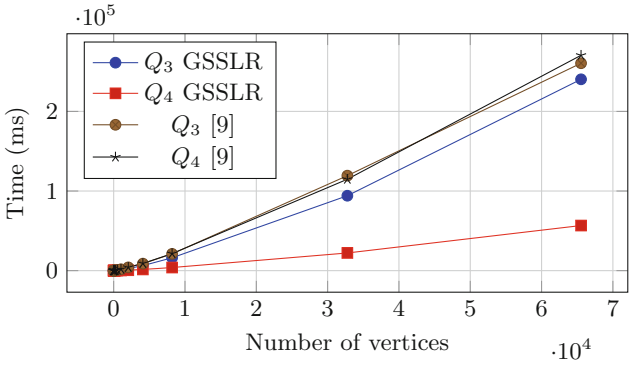

Fig. 7. Visualization of the binary tree experiment results.

4 Conclusion

We presented a new algorithm for the implementation of context-free path queries for graph databases. The proposed algorithm is inspired by the LR parsing technique [5] and uses a variant of the GSS introduced in [11] to encompass several derivations at a time. A Python prototype was implemented and experiments were conducted to validate and compare the results of our algorithm with those obtained by similar approaches. Two experiments and four queries were considered to evaluate our algorithm. In the first experiment, the ontologies used in [7,9,13] were used as databases. The main goal of this experiment was to investigate the feasibility of our method and compare our results with those reported in these works. In the second experiment, synthetic data of different sizes were used to investigate the scalability of our approach and compare it to [9]. Both experiments indicate that our algorithm performs as well as or better than similar approaches.

References

1. SPARQL 1.1 overview (2013). https://www.w3.org/TR/sparql11-query/. Accessed 13 Mar 2018
2. PRIMER rdf 1.1 primer (2014). https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/. Accessed 8 Feb 2017
3. RDF - semantics web standards (2014). https://www.w3.org/RDF/. Accessed 8 Feb 2017
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley Publishing Company Incorporated, Boston (2007)
6. Ancona, D., Bolz, C.F., Cuni, A., Rigo, A.: Automatic generation of JIT compilers for dynamic languages in .NET. Technical report, DISI, University of Genova and Heinrich-Heine-Universität Düsseldorf (2008)

7. Grigorev, S., Ragozina, A.: Context-free path querying with structural representation of result. In: Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia, CEE-SECR 2017, pp. 10:1–10:7. ACM, New York (2017)
8. Hellings, J.: Conjunctive context-free path queries. In: Proceedings of the 17th International Conference on Database Theory (ICDT), Athens, Greece, pp. 119–130, March 2014
9. Medeiros, C.M., Musicante, M.A., Costa, U.S.: Efficient evaluation of context-free path queries for graph databases. In: ACM SAC 2018: Symposium on Applied Computing. ACM, New York (2018). 8 pages
10. Scott, E., Johnstone, A., Hussain, S.S.: Tomita-style generalised LR parsers. Technical report, December 2000
11. Tomita, M.: An efficient augmented-context-free parsing algorithm. Comput. Linguist. **13**(1–2), 31–46 (1987)
12. Xia, F., Yang, L.T., Wang, L., Vinel, A.: Internet of things. Int. J. Commun. Syst. **25**(9), 1101 (2012)
13. Zhang, X., Feng, Z., Wang, X., Rao, G., Wu, W.: Context-free path queries on RDF graphs. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 632–648. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46523-4_38