Context-Free Path Querying with Structural Representation of Result

Semyon Grigorev
and Anastasiya Ragozina

Saint Petersburg State University, 7/9 Universitetskaya nab.
St. Petersburg, 199034 Russia
semen.grigorev@jetbrains.com, ragozina.anastasiya@gmail.com

Abstract. Graph data model and graph databases are very popular in various areas such as bioinformatics, semantic web, and social networks. One specific problem in the area is a path querying with constraints formulated in terms of formal grammars. The query in this approach is written as grammar, and paths querying is graph parsing with respect to given grammar. There are several solutions to it, but they are based mostly on CYK or Earley algorithms which have some restrictions in comparison with another parsing techniques, but usage of advances parsing techniques for graph parsing is still an open problem. In this paper we propose a graph parsing technique which based on generalized top-down parsing algorithm (GLL) and allows one to build such representation with respect to given grammar in polynomial time and space for arbitrary context-free grammar and graph.

Keywords: Graph database, path query, graph parsing, context-free grammar, top-down parsing, GLL, LL

1 Introduction

Graph data model and graph data bases are very popular in various areas such as bioinformatics, semantic web, social networks, etc. Extraction of paths which satisfy specific constraints may be useful for investigation of graph structured data and for detection of relations between data items. One specific problem—path querying with constraints—is usually formulated in terms of formal grammars and is called formal language constrained path problem [4].

Classical parsing techniques can be used to solve formal language constrained path problem. It means that such technique can be used for more common problem—graph parsing. Graph parsing may be required in graph data base querying, formal verification, string-embedded language processing, and another areas where graph structured data is used.

Though constrains formulated in terms of regular languages is widely used, using of more powerful context-free grammars for graph querying is in active research: recent results is context-free extension for SPARQL [20] and research of Jelle Hellings ([9,8]).

Existing solutions in context-free graph querying field usually employ such parsing algorithms as CYK or Earley(for example [8,20,17]). These algorithms are pretty simple, and impose different restrictions. For example, in case of CYK, the input grammar should be transformed to Chomsky normal form (CNF) which leads to grammar size increase. Result is a performance decreasing, because parsing time significantly depends on grammar size. In case of using such parsing algorithms as GLR and GLL there is no need to transform a grammar to CNF for these algorithms. Thus, advanced parsing techniques allow us to improve performance of parsing in some cases, but graph processing with advanced parsing algorithms [7] is an open question [9].

Despite the fact that there is a set of path querying solutions [17,8,3,12], query result exploration is still a challenge [10], as also a simplification of complex query debugging. In [9] annotated grammar proposed as possible solution: this representation is finite for any input data and contains information which can be useful for detailed result exploration. At the same time, classical parsing techniques provide another representation—derivation tree—which contains exhaustive information about parsed sentence structure in terms of specified grammar, and looks more native for grammar based analysis. In this work we show that finite derivation forest may be constructed for arbitrary graph and context-free grammar.

We make the following contributions in this paper.

1. We propose a graph parsing algorithm based on generalized top-down parsing algorithm—GLL [13]—and provide its time and space complexity estimations. For graph $M = (V, E, L)$ space complexity is $O(|V|^3 + |E|)$ and time complexity is $O\left(|V|^3 * \max_{v \in V}(deg^+(v))\right)$. Thus we answer on first and second questions which are stated in [9]: yes, we can use advanced parsing techniques for graph parsing, and we can use top-down parsing algorithm for goal-oriented queries evaluation.
2. We propose a graph parsing algorithm which allows one to construct finite representation of parse forest which contains derivation trees for all matched paths in graph. we show how this representation can be used for realistic problems solving.
3. We implement proposed algorithm and in evaluation we show that advanced parsing techniques allow to increase performance (up to 1000 times in some cases) in comparison with CYK-based implementation, proposed in [20].

2 Preliminaries

In this work we are focused on the parsing algorithm, and not on the data representation, and we assume that whole input graph can be located in RAM memory in the optimal for our algorithm way.

We start by introduction of necessary definitions.

- Context-free grammar is a quadruple $G = (N, \Sigma, P, S)$, where N is a set of nonterminal symbols, Σ is a set of terminal symbols, $S \in N$ is a start nonterminal, and P is a set of productions.
- $\mathcal{L}(G)$ denotes a language specified by grammar G, and is a set of terminal strings derived from start nonterminal of G: $L(G) = \{\omega | S \Rightarrow_G^* \omega\}$.
- Directed graph is a triple $M = (V, E, L)$, where V is a set of vertices, $L \subseteq \Sigma$ is a set of labels, and a set of edges $E \subseteq V \times L \times V$. We assume that there are no parallel edges with equal labels: for every $e_1 = (v_1, l_1, v_2) \in E, e_2 = (u_1, l_2, u_2) \in E$ if $v_1 = u_1$ and $v_2 = u_2$ then $l_1 \neq l_2$.
- $tag : E \to L$ is a helper function which allows to get tag of edge.

$$tag(e = (v_1, l, v_2), e \in E) = l$$

- $\oplus : L^+ \times L^+ \to L^+$ denotes a tag concatenation operation.
- Ω is a helper function which constructs a string produced by the given path. For every p path in M

$$\begin{aligned}&\Omega(p = e_0, e_1, \dots, e_{n-1}) = \\ &tag(e_0) \oplus \dots \oplus tag(e_{n-1}).\end{aligned}$$

Using these definitions, we state the context-free language constrained path querying as, given a query in form of grammar G, to construct the set of paths

$$Q(M, G) = \{p | p \text{ is path in } M, \Omega(p) \in \mathcal{L}(G)\}.$$

Note that, in some cases, $Q(M, G)$ can be an infinite set, and hence it cannot be represented explicitly. In order to solve this problem, in this paper, we construct compact data structure representation which stores all elements of $Q(M, G)$ in finite amount of space and allows to extract any of them.

2.1 Generalized LL Parsing Algorithm

One of widely used classes of parsing algorithms is a LL(k) [7]—top-down algorithms which read input from left to right, build leftmost derivation, and use k symbols for lookahead. PDA or recursive-descent — stack

Classical LL algorithm operates with a pointer to input (position i) and with a grammar slot—pointer to grammar in form $N \to \alpha \cdot x\beta$. Parsing may be described as a transition of these pointers from the initial position ($i = 0$, $S \to \cdot\beta$, where S is start nonterminal) to the final ($i = input.Length$, $s \to \beta\cdot$). At every step, there are four possible cases in processing of these pointers.

1. $N \to \alpha \cdot x\beta$, when x is a terminal and $x = input[i]$. In this case both pointers should be moved to the right ($i \leftarrow i + 1$, $N \to \alpha x \cdot \beta$).
2. $N \to \alpha \cdot X\beta$, when X is nonterminal. In this case we push return address $N \to \alpha X \cdot \beta$ to stack and move pointer in grammar to position $X \to \cdot\gamma$.
3. $N \to \alpha\cdot$. This case means that processing of nonterminal N is finished. We should pop return address from stack and use it as new slot.

4. $S \to \alpha\cdot$, where S is a start nonterminal of grammar. In this case we should report success if $i = input.Length - 1$ or failure otherwise.

In the second case there can be several slots $X \to \cdot\gamma$ due to the fact that this algorithm is nondeterministic, so a strategy on how to choose one of them to continue parsing is needed. In LL(k) algorithm lookahead is used to avoid nondeterminism, but this strategy is still not good enough because there are context-free languages for which deterministic choice is impossible even for infinite lookahead [5]. On the contrary to LL(k), generalized LL does not choose at all, handling all possible variants by using descriptors mechanism. Descriptor is a quadruple (L, s, j, a) where L is a grammar slot, s is a stack node, j is a position in the input string, and a is a node of derivation tree. Each descriptor fully describe one parser state, thus instead of immediate processing of all variants, GLL store all possible branches and process them sequentially late.

The stack in parsing process is used to store return information for the parser—a name of function which will be called when current function finishes computation. As mentioned before, generalized parsers process all possible derivation branches and parser must store it's own stack for every branch. It leads to an infinite stack growth being done naively. Tomita-style graph structured stack (GSS) [19] combines stacks resolving this problem. Each GSS node contains a pair of position in input and a grammar slot in GLL.

In order to provide termination and correctness, we should avoid duplication of descriptors, and be able to process GSS nodes in arbitrary order. It is necessary to use the following additional sets for this.

- R—working set which contains descriptors to be processed. Algorithm terminates whenever R is empty.
- U—all created descriptors. Each time when we want to add a new descriptor to R, we try to find it in this set first. This way we process each descriptor only once which guarantee termination of parsing.
- P—popped nodes. Allows to process descriptors (and GSS nodes) in arbitrary order.

There can be more than one derivation tree of a string with relation to ambiguous grammar. Generalized LL build all such trees and compact them in a special data structure Shared Packed Parse Forest [11], which will be described in the following section.

2.2 Shared Packed Parse Forest

Binarized Shared Packed Parse Forest (SPPF) [16] compresses derivation trees optimally reusing common nodes and subtrees. Version of GLL which uses this structure for parsing forest representation achieves worst-case cubic space complexity [14].

Binarized SPPF can be represented as a graph in which each node has one of four types described below. Let i and j be the start and the end positions of substring, and let us call a tuple (i, j) an *extension* of node.

- **Terminal node** with label (i, T, j).
- **Nonterminal node** with label (i, N, j). This node denotes that there is at least one derivation for substring $\alpha = \omega[i..j-1]$ such that $N \Rightarrow^*_G \alpha, \alpha = \omega[i..j-1]$. All derivation trees for the given substring and nonterminal can be extracted from SPPF by left-to-right top-down graph traversal started from respective node.
- **Intermediate node**: a special kind of node used for binarization of SPPF. These nodes are labeled with (i, t, j), where t is a grammar slot.
- **Packed node** with label $(N \to \alpha \cdot \beta, k)$. Subgraph with "root" in such node is one variant of derivation from nonterminal N in case when the parent is a nonterminal node labeled with ($\Leftrightarrow$ (i, N, j)).

Example of SPPF presented in figure 2a. We remove redundant intermediate and packed nodes from the SPPF to simplify it and to decrease the size of structure.

3 GLL-based Graph Parsing Algorithm

In this section we present such modification of GLL algorithm, that for input graph M, set of start vertices $V_s \subseteq V$, set of final vertices $V_f \subseteq V$, and grammar G_1, it returns SPPF which contains all derivation trees for all paths p in M, such that $\Omega(p) \in L(G_1)$, and $p.start \in V_s,\ p.end \in V_f$. In other words, we propose GLL-based algorithm which can solve language constrained path problem.

First of all, notice that an input string for classical parser can be represented as a linear graph, and positions in the input are vertices of this graph. This observation can be generalized to arbitrary graph with remark that for a position there is a set of labels of all outgoing edges for given vertex instead of just one next symbol. Thus, in order to use GLL for graph parsing we need to use graph vertices as positions in input and modify step 1 from section 2.1 to process multiple "next symbols". Small modification is also required for initialization of R set: it is necessary to add not only one initial descriptor but the set of descriptors for all vertices in V_s. Also, we should slightly change step 4: we should check that $i \in V_f$, not $i = input.Length - 1$. All other steps (and correspondent functions) are reused from original algorithm without any changes.

Our solution handles arbitrary numbers of start and final vertices, which allows one to solve different kinds of problems arising in the field, namely all paths in graph, all paths from specified vertex, all paths between specified vertices. As a result, we can use proposed algorithm for goal-oriented querying, and it is a positive answer on corresponding question stated in [9]. Also SPPF represents a structure of paths in terms of grammar which provides exhaustive information about result.

Detailed discussion of some formal properties of proposed algorithm published in our report [6]. Here we present only main results.

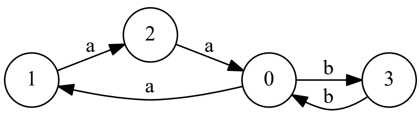
- Proposed algorithm terminates for arbitrary input data.

- The worst-case space complexity of proposed algorithm for graph $M = (V, E, L)$ is $O(|V|^3 + |E|)$.
- The worst-case runtime complexity of proposed algorithm for graph $M = (V, E, L)$ is $O\left(|V|^3 * \max\limits_{v \in V}(deg^+(v))\right)$.
- Result SPPF size is $O(|V'|^3 + |E'|)$ where $M' = (V', E', L')$ is a subgraph of input graph M which contains only matched paths.

4 An Example

Suppose that you are student in a School of Magic. It is your first day at School, so navigation in the building is a problem for you. Fortunately, you have a map of the building (fig. 1a) and additional knowledge about building construction:

- there are towers in the school (depicted as nodes of the graph in your map);
- towers can be connected by one-way galleries (represented as edges in your map);
- galleries have a “magic” property: you can start from any floor, but by following each gallery you either end up one floor above (edge label is ‘a’), or one floor below (edge label is ‘b’).



(a) The map of School (input graph M)

$0 : S \to a\ S\ b$
$1 : S \to Middle$
$2 : Middle \to a\ b$

(b) Grammar G_1 for language $L = \{a^n b^n; n \geq 1\}$ with additional marker for the middle of a path

Fig. 1: An example: input graph and grammar

You want to find a path from your current position to the same floor in another tower. Map with all such paths can help you. But orienteering is not your forte, so it would be great if the structure of the paths were as simple as possible and all paths had additional checkpoints to control your rout.

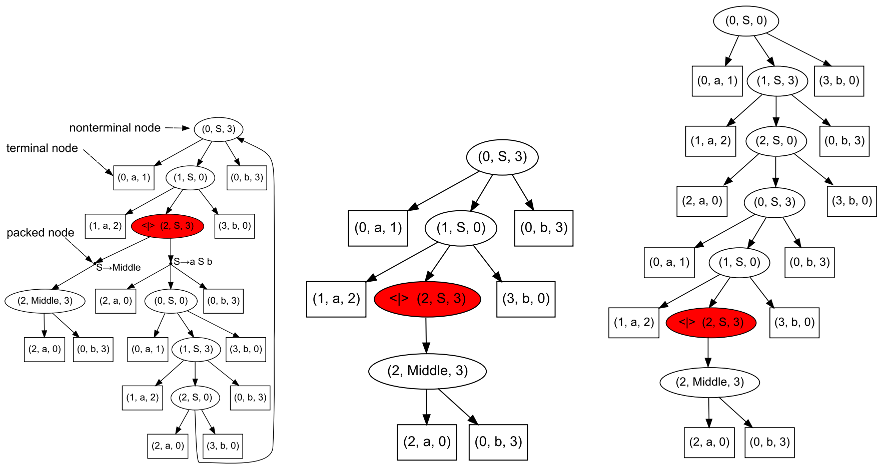
It is evident that the simplest structure of required paths is $\{ab, aabb, aaabbb, \ldots\}$. In terms of our definitions, it is necessary to find all paths p such that $\Omega(p) \in \{a^n b^n, n \geq 1\}$ in the graph $M = (\{0; 1; 2; 3\}, E, \{a; b\})$ (figure 1a).

Unfortunately, language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is not regular which restricts the set of tools you can use. Another problem is the infinite size of solution, but, being incapable to comprehend an infinite set of paths, you want to get a finite map. Moreover, you want to know structure of paths in terms of checkpoints.

We are not aware of any existing tools which can solve this problem, thus we have created such tool. Let us show how to get a map which helps to navigate in this strange School.

Fortunately, the language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is a context-free language and it can be specified with context-free grammar. The fact that one language can be described with multiple grammars allows to add checkpoints: additional nonterminals can mark required parts of sentences. In our case, desired checkpoint can be in the middle of the path. As a result, required language can be specified by the grammar G_1 presented in figure 1b, where $N = \{s; \mathit{Middle}\}$, $\Sigma = \{a; b\}$, and S is a start nonterminal.

Now, let us show that SPPF can be a solution for this problem. SPPF for data from example is presented in figure 2a. Each terminal node corresponds to the edge in the input graph: for each node with label (v_0, T, v_1) there is $e \in E : e = (v_0, T, v_1)$. Extensions stored in SPPF nodes allow us to check whether path from u to v exists and to extract it by SPPF traverse.



(a) Result SPPF (b) Derivation tree for p_0 (c) Derivation tree for p_1

Fig. 2: SPPF and examples of trees for specific paths for data from example (fig 1). Presented version does not contains redundant nodes, and we duplicate terminal nodes only for figure simplification. We use filled shape and label of form $(\lhd\!\mid\!\rhd\ (i, N, j))$ for nonterminal node to denote that there are multiple derivations from nonterminal N for substring $\omega[i..j-1]$.

As an example of derivation structure usage, we can find a middle of any path in example simply by finding correspondent nonterminal *Middle* in SPPF. So we can find out that there is only one (common) middle for all results, and it is a vertex with $id = 0$.

Lets find paths p_i such that $S \underset{G_1}{\Longrightarrow}^* \Omega(p_i)$ and p_i starts from the vertex 0. To do this, we should find vertices with label $(0, S, \_)$ in SPPF. (There are two vertices with such labels: $(0, S, 0)$ and $(0, S, 3)$.) Then let us to extract corresponded paths from SPPF. There is a cycle in SPPF in our example, so there are **at least** two different paths: $p_0 = \{(0, a, 1); (1, a, 2); (2, a, 0); (0, b, 3); (3, b, 0); (0, b, 3)\}$ and $p_1 = \{(0, a, 1); (1, a, 2); (2, a, 0); (0, a, 1); (1, a, 2); (2, a, 0); (0, b, 3); (3, b, 0); (0, b, 3); (3, b, 0); (0, b, 3); (3, b, 0)\}$. Trees for these paths are presented in figures 2b and 2c respectively.

We demonstrate that SPPF which was constructed by described algorithm can be useful for query result investigation. But in some cases explicit representation of matched subgraph is preferable, and required subgraph may be extracted from SPPF trivially by its traversal.

5 Evaluation

One of classical graph querying problems is a navigation queries for ontologies, and we apply our algorithm to this problem in order to estimate its practical value. We used dataset from paper [20]. Our algorithm is aimed to process graphs, so RDF files were converted to edge-labeled directed graph. For each triple (o, p, s) from RDF we added two edges: (o, p, s) and (s, p^{-1}, o).

We perform two classical *same-generation queries* [1], which, for example, is an important part of similarity query to biomedical databases [17].

Query 1 is based on the grammar for retrieving concepts on the same layer (presented in figure 3a). For this query our algorithm demonstrates up to 1000 times better performance and provides identical results as compared to the presented in [20] for Q_1.

Query 2 is based on the grammar for retrieving concepts on the adjacent layers (presented in figure 3b). Note that this query differs from the original query Q_2 from article [20] in the following details. First of all, we count only triples for nonterminal S because only paths derived from it correspond to paths between concepts on adjacent layers. Algorithm which is presented in [20] returns triples for all nonterminals. Moreover, grammar $\mathcal{G}_2$, which is presented in [20], describes paths not only between concepts on adjacent layers. For example, path "*subClassOf subClassOf*$^{-1}$" can be derived in $\mathcal{G}_2$, but it is a path between concepts on the same layer, not adjacent. We changed the grammar to fit a query to a description provided in paper [20]. Thus results of our query is different from results for Q_2 which provided in paper [20].

All tests were run on a x64-based PC with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 32 GB RAM. Results of both queries are presented in table 1, where #triples is a number of (o, p, s) triples in RDF file, and #results is a number of triples of form (S, v_1, v_2). In our approach result triples can be founded by filtering out all SPPF nonterminal nodes labeled by (v_1, S, v_2).

As a result, we conclude that our algorithm is fast enough to be applicable to some real-world problems.

$0 : S \rightarrow subClassOf^{-1}\ S\ subClassOf$
$1 : S \rightarrow type^{-1}\ S\ type$
$2 : S \rightarrow subClassOf^{-1}\ subClassOf$
$3 : S \rightarrow type^{-1}\ type$

(a) Grammar for query 1

$0 : S \rightarrow B\ subClassOf$
$1 : B \rightarrow subClassOf^{-1}\ B\ subClassOf$
$2 : B \rightarrow subClassOf^{-1}\ subClassOf$

(b) Grammar for query 2

Fig. 3: Grammars for evaluation

Table 1: Evaluation results for Query 1 and Query 2

Ontology	#triples	Query 1		Query 2	
		time(ms)	#results	time(ms)	#results
skos	252	10	810	1	1
generations	273	19	2164	1	0
travel	277	24	2499	1	63
univ-bench	293	25	2540	11	81
foaf	631	39	4118	2	10
people-pets	640	89	9472	3	37
funding	1086	212	17634	23	1158
atom-primitive	425	255	15454	66	122
biomedical-measure-primitive	459	261	15156	45	2871
pizza	1980	697	56195	29	1262
wine	1839	819	66572	8	133

6 Conclusion and Future Work

We propose GLL-based algorithm for context-free path querying which constructs finite structural representation of all paths satisfying given constraint. Provided data structure can be useful for result investigation and processing, and for query debugging. Presented algorithm has been implemented in F# programming language [18] and is available on GitHub:`https://github.com/YaccConstructor/YaccConstructor`.

We are working on performance improvement by implementation of recently proposed modifications in original GLL algorithm [15,2]. One direction of our research is generalization of grammar factorization proposed in [15] which may be useful for the processing of regular queries which are common in real world application.

Acknowledgments

We are grateful to the Jelle Hellings and Ekaterina Verbitskaia for their careful reading, pointing out some mistakes, and invaluable suggestions. This work is supported by grant from JetBrains Research

References

1. S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases, 1995.
2. A. Afroozeh and A. Izmaylova. Faster, practical gll parsing. In *International Conference on Compiler Construction*, pages 89–108. Springer, 2015.
3. P. Barceló, G. Fontaine, and A. W. Lin. Expressive path queries on graphs with data. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 71–85. Springer, 2013.
4. C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.
5. J. C. Beatty. Two iteration theorems for the ll (k) languages. *Theoretical Computer Science*, 12(2):193–228, 1980.
6. S. Grigorev and A. Ragozina. Context-free path querying with structural representation of result. *CoRR*, abs/1612.08872, 2016.
7. D. Grune and C. J. H. Jacobs. *Parsing Techniques (Monographs in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
8. J. Hellings. Conjunctive context-free path queries. 2014.
9. J. Hellings. Path results for context-free grammar queries on graphs. *CoRR*, abs/1502.02242, 2015.
10. P. Hofman and W. Martens. Separability by short subsequences and subwords. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
11. J. G. Rekers. *Parser generation for interactive environments.* PhD thesis, Citeseer, 1992.
12. J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, pages 1–53, 2015.
13. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
14. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
15. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
16. E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: a cubic tomita-style glr parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.
17. P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2):100, 2008.
18. D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0.* Springer, 2012.
19. M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'85, pages 756–764, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.
20. X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. Context-free path queries on rdf graphs. In *International Semantic Web Conference*, pages 632–648. Springer, 2016.