# Context-Free Path Querying with Structural Representation of Result

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
semen.grigorev@jetbrains.com

Anastasiya Ragozina
Saint Petersburg State University
St. Petersburg, Russia
ragozina.anastasiya@gmail.com

## ABSTRACT

Graph data model and graph databases are popular in such areas 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 a grammar and paths querying is graph parsing with respect to the grammar. There are several solutions to it, but they are based mostly on CYK or Earley algorithms which impose some restrictions in comparison with other parsing techniques, and employing of advanced parsing techniques for graph parsing is still an open problem. In this paper we propose a graph parsing technique which is based on generalized top-down parsing algorithm (GLL) and allows one to build finite structural query result representation with respect to the given grammar in polynomial time and space for arbitrary context-free grammar and graph.

## CCS CONCEPTS

• **Information systems → Graph-based database models**; **Query languages for non-relational engines**; • **Theory of computation** → *Grammars and context-free languages*; • **Software and its engineering** → *Parsers*;

## KEYWORDS

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

**ACM Reference format:**
Semyon Grigorev and Anastasiya Ragozina. 2017. Context-Free Path Querying with Structural Representation of Result. In *Proceedings of Central and Eastern European Software Engineering Conference Russia, Saint-Petersburg, Russian Federation, October 20–21, 2017 (CEE-SECR '17),* 7 pages.
https://doi.org/10.1145/3166094.3166104

## 1 INTRODUCTION

Graph data model and graph data bases are very popular in such areas as bioinformatics, semantic web, and social networks. Hence, different graph structured data analysis problems are stated and appropriate solutions for these problems are required. One specific problem—path querying with constraints—is usually formulated in terms of formal grammars and is called formal language constrained path problem [? ].

Classical parsing techniques can be used to solve formal language constrained path problem and thus the more common problem—graph parsing. Graph parsing may be required in graph data base querying, formal verification, string-embedded language processing, and any other areas where graph structured data is used.

Existing solutions in context-free graph querying field usually employ such parsing algorithms as CYK or Earley (for example [? ? ? ]). These algorithms are simple, but impose restrictions. For example, CYK algorithm demands the input grammar to be transformed into Chomsky normal form causing performance issues, since parsing time depends on grammar size which significantly increases during the transformation. Such algorithms as GLR and GLL process arbitrary context-free grammars, thus performance may be improved by employing them for graph parsing problem. Other properties of parsing algorithms also affect performance: for example, in [? ] author expects that it is possible to improve evaluation of queries for a given pair of nodes by using top-down directed parsing algorithm. Both CYK and Earley parsing algorithms are bottom-up, CYK is undirected, and Earley-based implementation [? ] is known to have issues with cycles processing. In this paper we show this assumption is true. We also provide a positive answer to the question of applicability of advanced parsing techniques stated in [? ].

Even though a set of path querying solutions has been developed [? ? ? ? ], query result exploration is still a challenge [? ], as also there is a need for simplification of complex query debugging, especially for context-free queries. In [? ], annotated grammars are proposed as a possible solution: this representation is finite for any input data and contains information necessary for detailed result exploration. In the paper we propose the representation more native for grammar based analysis, provided by classical parsing techniques—derivation tree—which contains exhaustive information about parsed sentence structure in terms of specified grammar.

We make the following contributions in this paper.

(1) We propose the graph parsing algorithm based on the generalized top-down parsing algorithm—GLL [? ]—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 $O\left(|V|^3 * \max_{v \in V}\left(deg^+(v)\right)\right)$.
(2) We answer some questions on advanced parsing techniques applicability for graph processing stated in [? ].
(3) Proposed graph parsing algorithm constructs finite representation of parse forest containing derivation trees for all matched paths in graph. We show how this representation can be used for realistic problems solving.
(4) We have implemented the proposed algorithm and our evaluation shows that advanced parsing techniques increase performance (up to 1000 times in some cases) as compared to CYK-based implementation, proposed in [? ].

## 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 optimally located in RAM memory.

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, $\Sigma$ 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 the grammar $G$, and is a set of terminal strings derived from the 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 returns a tag of a given edge.
$$tag(e = (v_1, l, v_2), e \in E) = l$$
- $\oplus : L^+ \times L^+ \to L^+$ denotes a tag concatenation operation.
- $\Omega$ is a helper function which constructs a string produced by the given path. For every $p$ path in $M$
$$\Omega(p = e_0, e_1, \ldots, e_{n-1}) = tag(e_0) \oplus \cdots \oplus tag(e_{n-1}).$$

Using these definitions, we define 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 $Q(M, G)$ can be an infinite set, hence it cannot be represented explicitly. In order to solve this problem, we construct compact data structure representation which stores all elements of $Q(M, G)$ in finite space and from which one can extract any of them.

### 2.1 Generalized LL Parsing Algorithm

One of widely used classes of parsing algorithms is LL(k) [? ]—top-down algorithms which read input from left to right, build leftmost derivation, and use $k$ symbols for lookahead. LL(k) parser may be implemented as a deterministic pushdown automaton (DPDA). On the other hand, LL(k) parser may be implemented in recursive-descent manner: each rule transforms to function which can call functions for other rules in order specified by right hand side of corresponded rule. In this case, stack of DPDA is replaced with functions call stack.

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

(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 the stack and move grammar pointer to position $X \to \cdot\gamma$, where $X \to \gamma \in P$.
(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 a 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.

There can be several slots $X \to \cdot\gamma$ in the second case since the algorithm is nondeterministic, so some strategy to choose one of them for further parsing is needed. Lookahead is used to avoid nondeterminism in LL(k) algorithms, but this strategy is still not perfect because for some context-free languages deterministic choice is impossible even with infinite lookahead [? ]. On the contrary to LL(k), generalized LL does not choose at all, but instead handles all possible cases by means of 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 being constructed. Each descriptor fully describes parser state, thus instead of immediate processing of all cases, GLL stores all possible branches and process them sequentially in arbitrary order.

The stack in parsing process is used to store return information for the parser—a state to return to after current state processing is finished. As mentioned before, generalized parsers process all possible derivation branches and parser must store its own stack for every branch. Being done naively, it leads to an infinite stack growth. Tomita-style Graph Structured Stack (GSS) [? ] combines stacks resolving this problem.

Each GSS node contains a pair of a position in the 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. The following additional sets are used for these purposes.

- $R$—working set which contains descriptors to be processed. Algorithm terminates as soon as $R$ is empty.
- $U$—all created descriptors. New descriptor is added to $R$, only if it is not in $U$. This way each descriptor is processed only once which guarantees termination of the algorithm, since there is only finite number of possible descriptors.
- $P$—popped nodes. This set is necessary for correct processing of descriptors (and GSS nodes) in arbitrary order.

Instead of explicit code generation used in classical algorithm, we use table version of GLL [**?** ] in order to simplify adaptation to graph processing. As a result, main control function is different from the original one because it should process LL-like table instead of switching between generated parsing functions. Control functions of the table based GLL are presented in Algorithm 1. All other functions are the same as in the original algorithm and their descriptions can be found in the original article [**?** ].

There can exist several derivation trees for a string with respect to an ambiguous grammar. Generalized LL builds all such trees and compacts them into a special data structure Shared Packed Parse Forest [**?** ], which is described in the following section.

## 2.2 Shared Packed Parse Forest

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

Binarized SPPF can be represented as a graph in which each node has one of four types described below. We denote the start and the end positions of substring as $i$ and $j$ respectively, and we call tuple $(i, j)$ an *extension* of a 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 $(N \to \alpha, k)$. Subgraph with "root" in such node is one possible derivation from nonterminal $N$ in case when the parent is a nonterminal node labeled with $(\Leftrightarrow (i, N, j))$.

**Algorithm 1** Control functions of table version of GLL

```
function DISPATCHER( )
 if $R.Count \neq 0$ then
   $(L, v, i, cN) \leftarrow R.Get()$
   $cR \leftarrow dummy$
   $dispatch \leftarrow false$
 else
   $stop \leftarrow true$
function PROCESSING( )
 $dispatch \leftarrow true$
 switch $L$ do
   case $(X \to \alpha \cdot x\beta)$ where $x = input[i+1]$)
     if $cN = dummyAST$ then
       $cN \leftarrow$ GETNODET$(i)$
     else
       $cR \leftarrow$ GETNODET$(i)$
     $i \leftarrow i + 1$
     $L \leftarrow (X \to \alpha x \cdot \beta)$
     if $cR \neq dummy$ then
       $cN \leftarrow$ GETNODEP$(L, cN, cR)$
     $dispatch \leftarrow false$
   case $(X \to \alpha \cdot x\beta)$ where $x$ is nonterminal
     $v \leftarrow$ CREATE$((X \to \alpha x \cdot \beta), v, i, cN)$
     $slots \leftarrow pTable[x][input[i]]$
     for all $L \in slots$ do
       ADD$(L, v, i, dummy)$
   case $(X \to \alpha \cdot)$
     POP(v,i,cN)
   case $(S \to \alpha \cdot)$ when $S$ is start nonterminal
     final result processing and error notification
function CONTROL
 while not $stop$ do
   if $dispatch$ then
     DISPATCHER( )
   else
     PROCESSING( )
```

An example of SPPF is presented in figure 3a. We remove redundant intermediate and packed nodes from the SPPF to simplify it and to decrease the size of the structure.

## 3 GLL-BASED GRAPH PARSING ALGORITHM

In this section we propose GLL-based algorithm which can solve language constrained path problem. In detail, 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 $p.start \in V_s,\ p.end \in V_f$.

First of all, note that an input string for a 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 every 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 the input and modify behavior in case 1 from section 2.1 to process multiple "next symbols". Small modification is also required for initialization of $R$ set: the set of descriptors for all vertices in $V_s$ should be added to $R$, not only one initial descriptor. Also, in the case 4 (2.1): we should check that $i \in V_f$, not that $i = input.Length - 1$. All other steps (and correspondent functions) are reused from the original algorithm without any changes.

**Algorithm 2 Processing** function modified in order to process arbitrary directed graph

```
function PROCESSING( )
 $dispatch \leftarrow true$
 switch $L$ do
  case $(X \to \alpha \cdot x\beta)$ where $x$ is terminal
   for all $\{e | e \in input.outEdges(i), tag(e) = x\}$ do
    $new\_cN \leftarrow cN$
    if $new\_cN = dummyAST$ then
     $new\_cN \leftarrow$ GETNODET($e$)
    else
     $new\_cR \leftarrow$ GETNODET($e$)
    $L \leftarrow (X \to \alpha x \cdot \beta)$
    if $new\_cR \neq dummy$ then
     $new\_cN \leftarrow$ GETNODEP($L, new\_cN, new\_cR$)
    ADD($L, v, target(e), new\_cN$)
  case $(X \to \alpha \cdot x\beta)$ where $x$ is nonterminal
   $v \leftarrow$ CREATE($(X \to \alpha x \cdot \beta), v, i, cN$)
   $slots \leftarrow \bigcup_{e \in input.OutEdges(i)} pTable[x][e.Token]$
   for all $L \in slots$ do
    ADD($L, v, i, dummy$)
  case $(X \to \alpha \cdot)$
   POP($v, i, cN$)
  case _
   final result processing and error notification
```

Our solution handles arbitrary numbers of start and final vertices, which allows one to solve different kinds of problems arising in the field, namely querying of all paths in graph, all paths from specified vertex, all paths between specified vertices. As a result, we can use proposed algorithm for querying paths for two specified vertices, and thus we provide positive answer for a question stated in [**?** ].

The algorithm has the following properties. Detailed discussion of them presented in the next section.

- 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 $O\left(|V|^3 * \max_{v \in V}\left(deg^+(v)\right)\right)$.
- 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.

## 3.1 Complexity

Time complexity estimation in terms of input graph and grammar size is quite similar to the estimation of GLL complexity provided in [**?** ].

LEMMA 3.1. *For any descriptor $(L, u, i, w)$ either $w = \$$ or $w$ has extension $(j, i)$ where $u$ has index $j$.*

PROOF. Proof of this lemma is the same as provided for original GLL in [**?** ] because main function used for descriptors creation has not been changed. □

THEOREM 1. *The GSS generated by GLL-based graph parsing algorithm for grammar $G$ and input graph $M = (V, E, L)$ has at most $O(|V|)$ vertices and $O(|V|^2)$ edges.*

PROOF. Proof is the same as the proof of **Theorem 2** from [**?** ] because structure of GSS has not been changed. □

THEOREM 2. *The SPPF generated by GLL-based graph parsing algorithm on input graph $M = (V, E, L)$ has at most $O(|V|^3 + |E|)$ vertices and edges.*

PROOF. Let us estimate the number of nodes of each type.

- **Terminal nodes** are labeled with $(v_0, T, v_1)$, and such label can only be created if there is such $e \in E$ that $e = (v_0, T, v_1)$. Note, that there are no duplicate edges. Hence there are at most $|E|$ terminal nodes.
- $\varepsilon$**-nodes** are labeled with $(v, \varepsilon, v)$, hence there are at most $|V|$ of them.
- **Nonterminal nodes** have labels of form $(v_0, N, v_1)$, so there are at most $O(|V|^2)$ of them.
- **Intermediate nodes** have labels of form $(v_0, t, v_1)$, where $t$ is a grammar slot, so there are at most $O(|V|^2)$ of them.
- **Packed nodes** are children either of intermediate or nonterminal nodes and have label of form $(N \to \alpha \cdot \beta, v)$. There are at most $O(|V|^2)$ parents for packed nodes and each of them can have at most $O(|V|)$ children.

As a result, there are at most $O(|V|^3 + |E|)$ nodes in SPPF. The packed nodes have at most two children so there are at most $O(|V|^3 + |E|)$ edges which source is packed node. Nonterminal and intermediate nodes have at most $O(|V|)$ children and all of them are packed nodes. Thus there are at most $O(|V|^3)$ edges with source in nonterminal or intermediate nodes. As a result there are at most $O(|V|^3 + |E|)$ edges in SPPF. □

THEOREM 3. *The worst-case space complexity of GLL-based graph parsing algorithm for graph $M = (V, E, L)$ is $O(|V|^3 + |E|)$.*

Immediately follows from theorems 1 and 2.

THEOREM 4. *The worst-case runtime complexity of GLL-based graph parsing algorithm for graph $M = (V, E, L)$ is*

$$O\left(|V|^3 * \max_{v \in V}\left(deg^+(v)\right)\right).$$

PROOF. From Lemma 3.1, there are at most $O(|V|^2)$ descriptors. Complexity of all functions which were used in algorithm is the same as in proof of **Theorem 4** from [**?** ] except **Processing** function in which not a single next input token, but the whole set of outgoing edges, should be processed. Thus, for each descriptor at most

$$\max_{v \in V}\left(deg^+(v)\right)$$

edges are processed, where $deg^+(v)$ is outdegree of vertex $v$.

Thus, worst-case complexity of proposed algorithm is

$$O\left(V^3 * \max_{v \in V}\left(deg^+(v)\right)\right).$$

□

We can get estimations for linear input from theorem 4. For any $v \in V$, $deg^+(v) \leq 1$, thus $\max_{v \in V}(deg^+(v)) = 1$ and worst-case time complexity $O(|V|^3)$, as expected. For LL grammars and linear input complexity should be $O(|V|)$ for the same reason as for original GLL.

As discussed in [**?** ], special data structures, which are required for the basic algorithm, can be not rational for practical implementation, and it is necessary to find balance between performance, software complexity, and hardware resources. As a result, we can get slightly worse performance than theoretical estimation in practice.

Note that result SPPF contains only paths matched specified query, so 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. Also note that each specific path can be explored by linear SPPF traversal.

## 4 AN EXAMPLE

In this section we demonstrate our algorithm on a small problem. The query used in our example may seem too primitive, but keep in mind that it is a classical context-free *same-generation query* [**?** ], which is base for other context-free queries [**?** ].

Suppose that you are a student in 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. 1) 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').

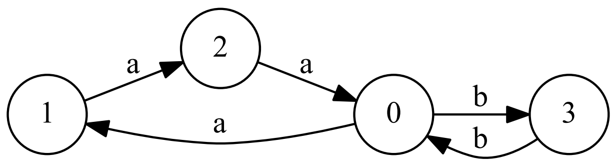


**Figure 1: An example: the map of School (input graph $M$)**

$$\begin{array}{rl} 0: & S \to a\ S\ b \\ 1: & S \to Middle \\ 2: & Middle \to a\ b \end{array}$$

**Figure 2: An example: grammar $G_1$ for language $L = \{a^n b^n; n \geq 1\}$ with additional marker for the middle of a path**

You want to find a path from your current position to the same floor in another tower. If only you have a map with all such paths. But orienteering is not your forte, so you prefer the structure of the paths be as simple as possible and all paths to have additional checkpoints to control your route.

It is evident that the simplest structure of required paths is $\{ab, aabb, aaabbb, \dots\}$. 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 1).

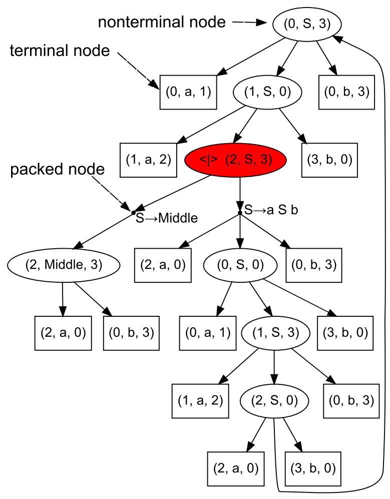
Unfortunately, language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is not regular which restricts the set of tools you can use. Another problem is the infinite size of solution. Being incapable to comprehend infinite set of paths, you want to obtain a finite map. Moreover, you want to know the structure of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

We are not aware of any existing tools which can solve this problem, except for the one presented in this paper. Let us show how one can get a map which helps to navigate in this strange School by using our tool.

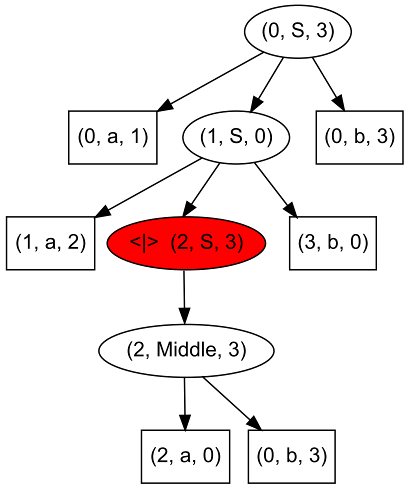
As the language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is context-free, it can be specified with context-free grammar. The fact that one language can be described with multiple grammars allows us to add checkpoints: additional nonterminals can mark required subpaths. In our case, desired checkpoint can be in the middle of the path. As a result, required language can be specified by the grammar $G_1$ presented in figure 2, where $N = \{s; Middle\}$, $\Sigma = \{a; b\}$, and $S$ is a start nonterminal.

Now, let us show that SPPF constructed by our algorithm is the desired map. SPPF for the example is presented in figure 3a. 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.

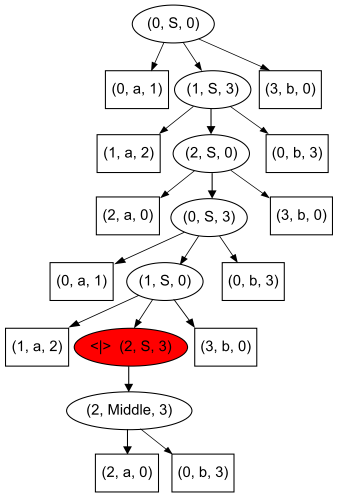
To illustrate how to use derivation structure, we can find a middle of any path. It is sufficient to find the nonterminal *Middle* in the SPPF for this purpose. So we can conclude



(a) Result SPPF



(b) Derivation tree for $p_0$



(c) Derivation tree for $p_1$

**Figure 3: SPPF and examples of trees for specific paths for data from example (fig 1 and fig 2). Redundant nodes are eliminated; terminal nodes are duplicated only for figure simplification. We use filled shape and label of form ($\diamondsuit$ $(i, N, j)$) for nonterminal node to denote that there are multiple derivations from nonterminal $N$ for substring $\omega[i..j-1]$.**

from examining the result that there is only one middle point for all possible paths and it is a vertex with $id = 0$.

Let us 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 such vertices: $(0, S, 0)$ and $(0, S, 3)$.) Then let us 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 3b and 3c respectively.

By this example we demonstrated that SPPF which was constructed by the described algorithm can be useful for query result investigation. Also, when explicit representation of matched subgraph is preferable, required subgraph may be extracted from SPPF trivially by its traversal.

## 5 EVALUATION

In order to estimate practical value of our algorithm, we applied it to the classical graph querying problem: navigation queries for ontologies. We used dataset from a paper [? ], but as our algorithm is aimed to process graphs, we first converted RDF files to edge-labeled directed graphs. This was done by creating a pair of edges $(o, p, s)$ and $(s, p^{-1}, o)$ for each triple $(o, p, s)$ in RDF file.

We perform two classical *same-generation queries* [? ], which are important parts of similarity query to biomedical databases [? ].

**Query 1** is based on the grammar for retrieving concepts on the same layer (presented in figure 4). For this query our algorithm demonstrates up to 1000 times better performance and provides identical results as compared to the presented in [? ] for $Q_1$.

$$\begin{array}{ll} 0: & S \to subClassOf^{-1}\ S\ subClassOf \\ 1: & S \to type^{-1}\ S\ type \\ 2: & S \to subClassOf^{-1}\ subClassOf \\ 3: & S \to type^{-1}\ type \end{array}$$

**Figure 4: Grammar for query 1**

**Query 2** is based on the grammar for retrieving concepts on the adjacent layers (presented in figure 5). Note that this query differs from the original query $Q_2$ from the paper [? ] in the following points. First of all, we count only triples for the nonterminal $S$ because only paths derived from it correspond to the paths between concepts on adjacent layers. Algorithm presented in [? ] returns triples for all nonterminals. Moreover, the grammar $\mathcal{G}_2$ presented in [? ], 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 the query to the description provided in the paper [? ]. Thus results of our query differs from results for $Q_2$ which can be found in [? ].

$$\begin{array}{ll} 0: & S \rightarrow B\ subClassOf \\ 0: & S \rightarrow subClassOf \\ 1: & B \rightarrow subClassOf^{-1}\ B\ subClassOf \\ 2: & B \rightarrow subClassOf^{-1}\ subClassOf \end{array}$$

**Figure 5: Grammar for query 2**

All tests were run on a x64-based PC with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 32 GB RAM. Results for both queries are presented in the 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 $(v_1, S, v_2)$.

**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 |

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

## 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 as well as for query debugging. Presented algorithm has been implemented in F# programming language [**?** ] and is available on GitHub: https://github.com/YaccConstructor/YaccConstructor.

We are working on improving performance by means of implementation of recently proposed modifications for the original GLL algorithm [**? ?** ]. Future direction of our research is a generalization of grammar factorization proposed in [**?** ] which may be practical for processing of regular queries which are common in real world applications.

## 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.