???

Semyon Grigorev
Saint Petersburg State University
7/9 Universitetskaya nab.
St. Petersburg, 199034 Russia
semen.grigorev@jetbrains.com

Anastasiya Ragozina
Saint Petersburg State University
7/9 Universitetskaya nab.
St. Petersburg, 199034 Russia
ragozina.anastasiya@gmail.com

ABSTRACT

Graph data model and graph databases are very popular in many different areas such as bioinformatics, semantic web, social networks, etc. One of specific problems of graph databases is a path querying with constrains formulated in terms of formal grammars. There are several solutions, but providing structural representation of query result which is practical for answer investigation and exploration is still a problem. In this paper we propose graph parsing technique which allows us to build such representation with respect to given grammar query for arbitrary context-free grammar and graph. Proposed algorithm is based on generalized LL parsing algorithm while previous solutions based mostly on CYK or Earley algorithms.

1. INTRODUCTION

Graph data model and graph data bases are very popular in many different 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 [3].

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.

Existing solutions in databases field usually use such parsing algorithms as CYK or Earley(for example [6], [14]). These algorithms have nonlinear time complexity for unambiguous grammars ($O(n^3)$ and $O(n^2)$ respectively). Moreover the input grammar should be transformed to Chomsky normal form (CNF) in case of CYK which leads to grammar size increasing and . To solve these problems we can use such parsing algorithms as GLR and GLL which have cubic worst-case complexity and linear for unambiguous grammars. Also there is no need to transfor grammar to CNF for these algorithms. These facts allow us to improve performance of parsing in some cases.

Despite the fact that there is a set of path querying solutions [14, 6, **?**], query result exploration is still a challenge [7]. Simplification of complex query debugging is also a problem. Structural representation of query result can be used to solve these problems, and classical parsing techniques provide such representation—derivation tree—which contains exhaustive information about parsed sentence structure in terms of specified grammar.

Graph parsing can also be used to analyze dynamically generated strings or string-embedded languages. String variable in a program may gets more then one value in run time. For static analysis we can use regular approximation for value set of string variable which can be represented as a finite automaton. Moreover, if we want to check a syntactic correctness of dynamically generated strings, we should check that all generated strings (all paths from start states to final states in the given automaton) are correct with respect to given context-free grammar. There are some solutions to this problem: GLR-based checker of string-embedded SQL queries [2, 4]; parser of string-embedded languages [17] based on RNGLR parsing algorithm. RNGLR-based algorithm allows to construct derivation forest for all correct paths in the input automaton.

In this paper we propose graph parsing technique which allows to construct structural representation of query result with respect to given grammar. This structure can be useful for query debugging and exploration. Proposed algorithm is based on generalized top-down parsing algorithm—GLL [10]—which has cubic worst-case time complexity and linear for LL grammars on linear input.

2. PRELIMINARIES

In this work we are focused on the parsing algorithm, and not on the data representation, and we assume that the 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 nonterminl 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 edge's labels, and 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.
- Path p in graph M is a list of incident edges:

$$p = (v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_{n-1}, v_n)$$
$$= e_0, e_1, \ldots, e_{n-1}$$

  where $v_i \in V$, $e_i \in E$, $e_i = (v_i, l_i, v_{i+1})$, $l_i \in L$, $|p| = n, n \geq 1$.

- Set of paths $P = \{p : p \text{ path in } M\}$ where M is a directed graph.
- $\Omega : p \to L^+$ is a helper function which constructs a string produced by the given path.

$$\Omega(p = e_0, e_1, \ldots, e_{n-1}, p \in P) =$$
$$tag(e_0) \oplus \cdots \oplus tag(e_{n-1}).$$

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

$$P = \{p | \Omega(p) \in \mathcal{L}(G)\}.$$

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

3. MOTIVATING EXAMPLE

Let suppose that you are student of School of Magic. It is your first day in the School, so navigation in the building is a problem for you. Fortunately you have a map of building (fig 1) and some additional knowledges on building properties:

- there are some towers in the school (nodes of the graph in your map);
- all floors of some towers connected by directed galleries (edges in your map);
- each gallery has a "magic" property: start floor is aways one more (edge label is 'b') or one less (edge label is 'b') then end floor.

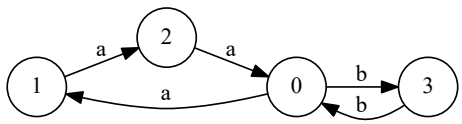


Figure 1: The map of School (input graph M)

And now you want to find the path from your current position to the same floor of another tower. Map with all such paths can help you. But orienting is not your forte, so it would be great if all structure of paths will be ass simple as possible and all paths will have checkpoints to control your rout.

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

First problem is that language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is not a regular and this fact restricts the set of tools which you can use. Another problem is that solution in presented case is an infinite set of path, and you want to get a finite map without any magic. Moreover, you want to know a structure of paths with checkpoints added.

We do not know any existing tools which can help to solve your problem and we create a new one. Let us to show how to get the map which help to orient in this strange School.

Fortunately the language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is a context-free language and can be specified by context-free grammar. The fact that one language can be described with more then one grammar allow you to add checkpoints: you can use additional nonterminals for "marking" required parts of sentence. In our case the good checkpoint is a middle of path. As a result, required language can be specified by grammar G_1 presented in picture 2 where $N = \{s; Middle\}$, $\Sigma = \{a; b\}$, and S is a start nonterminal.

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

Figure 2: Grammar G_1 for language $L = \{a^n b^n; n \geq 1\}$ with additional marker for path's middle

Algorithm of map construction is presented below.

4. GRAPH PARSING ALGORITHM

We propose graph parsing algorithm which allows to construct finite representation of parse forest which contains trees for all matched paths in graph. Finite representation of result set with structure related to the specified grammar may be useful not only for results understanding and processing but also for query debugging especially for complex queries.

Our solution is based on generalized LL (GLL) [10, 1] parsing algorithm which allows to process arbitrary (including left-recursive and ambiguous) context-free grammars with worst-case cubic time complexity and linear for LL grammars on linear input.

4.1 Generalized LL Parsing Algorithm

In classical LL algorithm we have a pointer to input (position i) and a pointer to grammar in form $N \to \alpha \cdot x\beta$ — grammar slot. The parsing may be described as a movement of these pointers from initial position ($i = 0$, $S \to \cdot\beta$, where S is start nonterminal) to 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: $i = i + 1$, new slot $N \to \alpha x \cdot \beta$.

2. $N \to \alpha \cdot X\beta$ when X is nonterminal. In this case we should 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 in other case.

In the second case here can be several slots $X \to \cdot\gamma$, so some strategy on how to choose one of them to continue parsing is needed. In LL(k) algorithm lookahead is used, but this strategy is still not good enough. On the contraty to LL(k), generalized LL doesn't choose at all, handling all possible variants. Note, that instead of immediate processing of all variants, GLL uses descriptors mechanism to store all possible branches and process them sequentially. 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, and a is a node of derivation tree.

The stack in parsing process is used to store return information for the parser — a name of function which would be called when current function will finish computation. As mentioned before, generalized parsers process all possible derivation branches and for every branch parser must store it's own stack. In naive approach it leads to infinite stack growth. Tomita-style graph structured stack (GSS) [16] allows to combine stacks to solve this problem. In GLL each GSS node contains a pair of position in input and a grammar slot.

In order to provide termination and correctness we should avoid duplication of descriptors, and allow to process GSS nodes in arbitrary order. It is necessary to use some 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 new descriptor to R, we try to find it in this set first. This way we process each descriptor only once.
- P — popped nodes. Allows to process descriptors (and GSS nodes) in arbitrary order.

Instead of explicit code generation used in classical algorithm, we use table version of GLL [5] 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 listing 1. All other functions are the same as in the original algorithm and their descriptions can be found in the original article [10] or in Appendix A.

There are more than one tree for ambiguous grammar and generalized algorithms builds all derivation trees. Special data structure — Shared Packed Parse Forest [9] — is used to reduce space required for tree storage.

4.2 Shared Packed Parse Forest

Binarized Shared Packed Parse Forest (SPPF) [13] allows to compress derivation trees with optimal reusing of common nodes and subtrees. Version of GLL which uses this

Algorithm 1 Control functions of table version of GLL

```
1: function DISPATCHER( )
2:   if R.Count ≠ 0 then
3:     (L, v, i, cN) ← R.Get()
4:     cR ← dummy
5:     dispatch ← false
6:   else
7:     stop ← true
8: function PROCESSING( )
9:   dispatch ← true
10:  switch L do
11:    case (X → α · xβ) where x = input[i + 1])
12:      if cN = dummyAST then
13:        cN ← GETNODET(i)
14:      else
15:        cR ← GETNODET(i)
16:      i ← i + 1
17:      L ← (X → αx · β)
18:      if cR ≠ dummy then
19:        cN ← GETNODEP(L, cN, cR)
20:      dispatch ← false
21:    case (X → α · xβ) where x is nonterminal
22:      v ← CREATE((X → αx · β), v, i, cN)
23:      slots ← pTable[x][input[i]]
24:      for all L ∈ slots do
25:        ADD(L,v,i,dummy)
26:    case (X → α·)
27:      POP(v,i,cN)
28:    case (S → α·) when S is start nonterminal
29:      final result processing and error notification
30: function CONTROL
31:   while not stop do
32:     if dispatch then
33:       DISPATCHER( )
34:     else
35:       PROCESSING( )
```

structure for parsing forest representation achieves worst-case cubic space complexity [11].

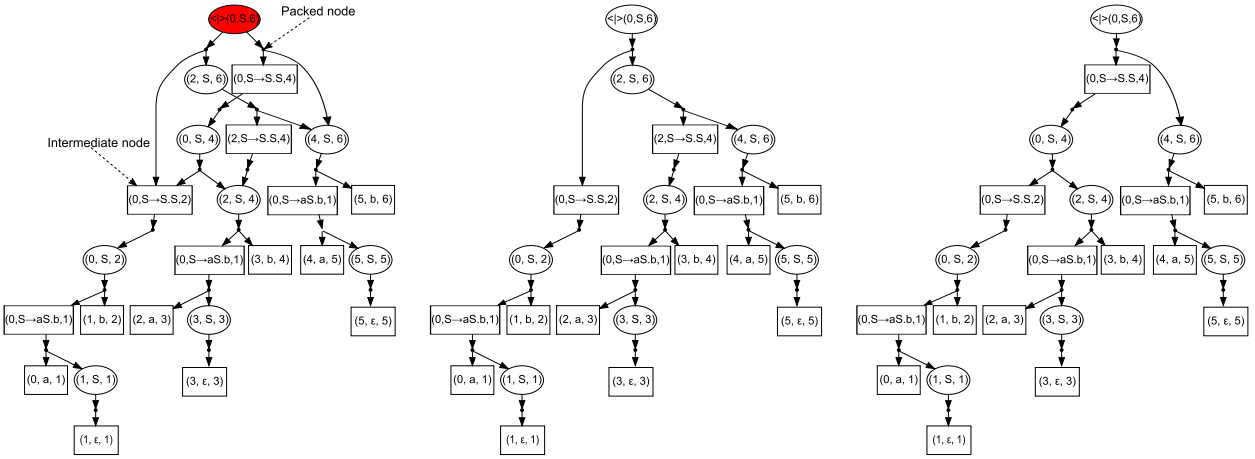
Let us present an example of SPPF for the input sentence `"abababa"` and ambiguous grammar G_0 (fig 3).

$$\begin{aligned} 0: &\quad S \to \varepsilon \\ 1: &\quad S \to b\ S\ b \\ 2: &\quad S \to S\ S \end{aligned}$$

Figure 3: Grammar G_0

There are two different leftmost derivations of given sentence in grammar G_0, hence SPPF should contain two different derivation trees. Resulting SPPF(fig. 4a) and two trees extracted from it (fig. 4b and fig. 4c) are presented in figure 4.

Binarised SPPF can be represented as a graph where each node has one of four types which described below with correspondent graphical notation. Let i and j be start and end positions of substring, and tuple of positions (i, j) — an *extension* of node.



(a) SPPF (b) First derivation tree (c) Second derivation tree

Figure 4: SPPF for sentence `"ababab"` and grammar G_0

- Node of rectangle shape labeled with (i, T, j) is a terminal node.
- Node of oval shape labeled with (i, N, j) is a nonterminal node. This node denotes that there is at least one derivation for substring α from position i to position j in input string ω such that $N \Rightarrow^*_G \alpha, \alpha = \omega[i..j-1]$. All derivation trees for given substring and nonterminal can be extracted from SPPF by left-to-right top-down graph traversal started from respective node. We use filled nonterminal node labeled with $(\diamond\!\!\!\!\!\!\diamond\ (i, N, j))$ to denote that there are more then one derivations from nonterminal N for substring from i to j.
- Node of rectangle shape and label (i, t, j) where t is a grammar slot is an intermediate node: a special node which allows to construct binary version of SPPF.
- Packed node with label $(N \to \alpha \cdot \beta, k)$. In our pictures we use dot shape for these nodes and omit labels because they are important only on SPPF constriction stage. Subgraph with "root" in such node is one variant of derivation from nonterminal N in case when parent is nonterminal node with label $(\diamond\!\!\!\!\!\!\diamond\ (i, N, j))$.

Later in our examples we will remove redundant intermediate and packed nodes from the SPPF to simplify it and to decrease the size of structure.

4.3 GLL-based Graph Parsing

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 input string for classical parser can be represented as a linear graph, and positions in 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 **Processing** function to allow to process multiple "next symbols". Required modifications are presented in listing 2(line 5 and 17). 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. All other functions can be reused from original algorithm without any changes.

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

```
1: function PROCESSING( )
2:   dispatch ← true
3:   switch L do
4:     case (X → α · xβ) where x is terminal
5:       for all {e|e ∈ input.outEdges(i), tag(e) = x} do
6:         new_cN ← cN
7:         if new_cN = dummyAST then
8:           new_cN ← GETNODET(e)
9:         else
10:          new_cR ← GETNODET(e)
11:        L ← (X → αx · β)
12:        if new_cR ≠ dummy then
13:          new_cN ← GETNODEP(L, new_cN, new_cR)
14:        ADD(L,v,target(e),new_cN)
15:    case (X → α · xβ) where x is nonterminal
16:      v ← CREATE((X → αx · β), v, i, cN)
17:      slots ← ⋃_{e∈input.OutEdges(i)} pTable[x][e.Token]
18:      for all L ∈ slots do
19:        ADD(L,v,i,dummy)
20:    case (X → α·)
21:      POP(v,i,cN)
22:    case _
23:      final result processing and error notification
```

Note that our solution handles arbitrary numbers of start and final vertices, which allows to solve different kinds of problems arraising in the field , namely all paths in graph,

all paths from specified vertex, all paths between specified vertices. Also SPPF represents a structure of paths in terms of derivation which allows to get more useful information about result.

Note that termination of proposed algorithm is inherited from the basic GLL algorithm. We are process finite graphs, hence the set of positions is finite, and tree construction is not changed. As a result, total number of descriptors is finite, and any of them cannot be added in R twice, hence main loop is finite.

4.4 Complexity

Time complexity estimation in terms of input graph and grammar size is pretty similar to estimation of GLL complexity provided in [11].

LEMMA 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 [11] because main function used for descriptors creation was not changed. □

THEOREM 1. *The GSS generated by GLL-based graph parsing algorithm for grammar G on 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 [11] because structure of GSS was not 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.** Each of them is labeled with (T, v_0, v_1), and such label can be created only 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.
- ε **nodes** are labeled with (v, ε, 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.
- **Indeterminate 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 of intermediate or nonterminal nodes and have label of form (t, v) where t is a grammar slot $N \to \alpha \cdot \beta$. 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 with source in 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 space complexity of GLL-based graph parsing algorithm for graph $M = (V, E, L)$ is at most $O(|V|^3 + |E|)$.*

PROOF. From theorems 1 and 2 we have that space required for main data structures is at most $O(|V|^3 + |E|)$.

□

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

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

PROOF. From Lemma 1 we get that there are at most $O(|V|^2)$ descriptors. Complexity of all functions which used in algorithm are the same as in proof of **Theorem 4** from [11] except **Processing** function where we should process not single next input token, but the whole set of outgoing edges. Thus, for each descriptor we should examine at most

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

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

Thus, worst-case complexity of proposed algorithm is

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

□

From theorem 4 we can get estimations for linear input and for LL grammars: for any $v \in V$ it is true, that $deg^+(v) \leq 1$, so $\max_{v \in V}(deg^+(v)) = 1$ and we get $O(|V|^3)$, as expected. For LL grammars and linear input complexity should be $O(|V|)$ for the same reason as for original GLL.

As discussed in [8], special data structures, which requred for basic algirithm, can be irrational for practice implementation and it is necessary to find balance between performance, software complexity, and hardware resources. As a result in practice we can get slightly worse performance than theoretical estimation.

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 with linear SPPF traversal.

4.5 Example

Let us present a solution of motivating example: grammar G_1 is a query and we want to find all paths in graph M (presented in picture 1) matching this query. Result SPPF for this input is presented in figure 5. Note that presented version does not contains redundant nodes. Each terminal node corresponds to the edge in the input graph: for each node with label (v_0, T, v_1) there is $e \in E : e = (v_0, T, v_1)$. We duplicate terminal nodes only for figure simplification.

As an example of derivation structure usage we can find middle of any path in example above 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 $id = 0$.

Extensions stored in nodes allow us to check whether path from u to v exists and to extract it. To extract path we need

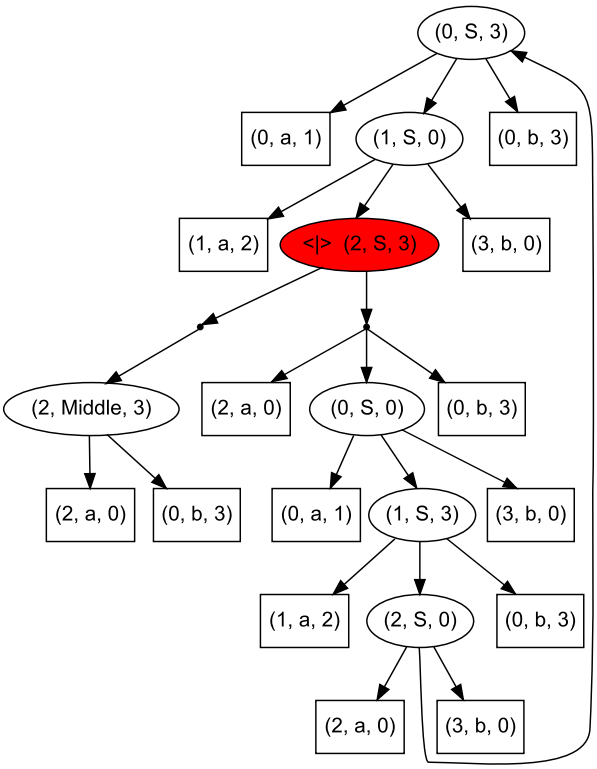


Figure 5: Result SPPF for input graph M(fig. 1) and query G_1(fig. 2)

only to traverse SPPF, and it can be done in polinomial time (in terms of SPPF size).

Lets find paths satisfying specified in G_1 constraints from vertex 0. To do this, we should find vertices with label $(0, S, \_)$ in SPPF. We can see that there are two vertices with label matched this pattern: $(0, S, 0)$ and $(0, S, 3)$. At the next step let us to extract corresponded paths from SPPF. In our example there is a cycle in SPPF 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

$$p_1 = \{(0, a, 1); (1, a, 2); (2, a, 0); (0, a, 1); (1, a, 2); (2, a, 0); \\ (0, b, 3); (3, b, 0); (0, b, 3); (3, b, 0); (0, b, 3); (3, b, 0)\}.$$

Thus 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 that may be extracted from SPPF trivially by its traversal.

5. EVALUATION

In this section we show that performance of implemented algorithm is in good agreement with theoretical estimations, and that the worst case of time and space complexity can be achieved.

We use two grammars for balanced brackets — ambiguous grammar G_0(fig. 3) and unambiguous grammar G_2(fig. 6) — in order to investigate performance and grammar ambiguity correlation.

$$0: \quad S \to a\ S\ b\ S$$
$$1: \quad S \to \varepsilon$$

Figure 6: Unambiguous grammar G_2 for balanced brackets

For input we use complete graphs where for each terminal symbol there is an edge between every two vertices labeled with it. Note that we use only terminal symbols for edges labels. The task we solve in our experiments is to find all paths from all vertices to all vertixes satisfied specified query. Such designed input looks hard for querying in terms of required resources because there are correct path between any two vertices and result set is infinite.

For complete graph $M = (V, E, L)$ we get

$$\max_{v \in V} \left(deg^+(v)\right) = (|V| - 1) * |\Sigma|$$

where Σ is terminals of input grammar, hence we should get time complexity at most $O(|V|^4)$ and space complexity at most $O(|V|^3)$.

All tests were performed on a PC with following characteristics:

- OS Name: Microsoft Windows 10 Pro
- System Type: x64-based PC
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Core(s), 4 Logical Processor(s)
- RAM: 32 GB

Performance measurement results are presented in figure 7. For time measurement results we have that all two curves can be fit with polynomial function of degree 4 to a high level of confidence with R^2.

Also we present SPPF size in terms of nodes for both G_0 and G_2 grammars 8. As we expected, all two curves are cubic to a high level of confidence with $R^2 = 1$.

6. CONCLUSION AND FUTURE WORK

We propose GLL-based algorithm for context-free path querying which construct 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# [15] and is available on GitHub:https://github.com/YaccConstructor/YaccConstructor.

In order to estimate practical value of proposed algorithm we should perform evaluation on a real dataset and real queries. One possible application of our algorithm is metagenomical assembly querying, and we are currently working on this topic.

We are also working on performance improvement by implementation of recently proposed modifications in original GLL algorithm [12, 1]. One direction of our research is generalization of grammar factorization proposed in [12] which may be useful for the processing of regular queries which are common in real world application.

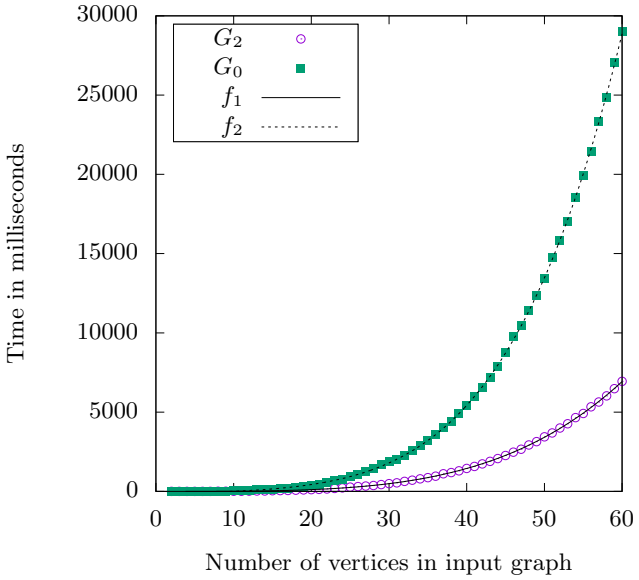


Figure 7: Performance on complete graphs for grammar G_0 and G_2
$f_1(x) = 0.000495989 * x^4 + 0.001252184 * x^3 + 0.068491746 * x^2 - 0.306749160 * x$; $R^2 = 0.99996$
$f_2(x) = 0.003368883 * x^4 - 0.114919298 * x^3 + 3.161793404 * x^2 - 22.549491142 * x$; $R^2 = 0.99995$

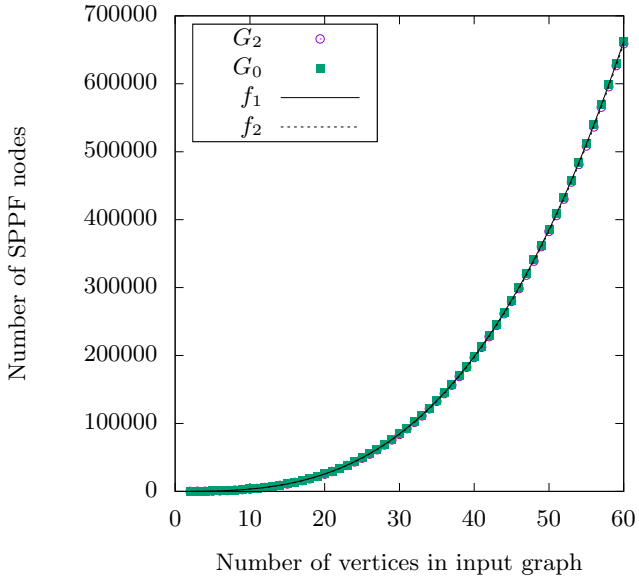


Figure 8: SPPF size on complete graph for grammars G_0 and G_2 an complete graphs
$f_1(x) = 3.000047 * x^3 + 3.994579 * x^2 + 4.191568 * x$; $R^2 = 1$
$f_2(x) = 3.000050 * x^3 + 2.994338 * x^2 + 4.196472 * x$; $R^2 = 1$

7. REFERENCES

[1] A. Afroozeh and A. Izmaylova. Faster, practical gll parsing. In *International Conference on Compiler Construction*, pages 89–108. Springer, 2015.

[2] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene. An interactive tool for analyzing embedded sql queries. In *Asian Symposium on Programming Languages and Systems*, pages 131–138. Springer, 2010.

[3] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[4] A. Breslav, A. Annamaa, and V. Vene. Using abstract lexical analysis and parsing to detect errors in string-embedded dsl statements. In *Proceedings of the 22nd Nordic Workshop on Programming Theory*, pages 20–22, 2010.

[5] S. V. Grigorev and A. K. Ragozina. Generalized table-based ll-parsing. *Sistemy i Sredstva Informatiki [Systems and Means of Informatics]*, 25(1):89–107, 2015.

[6] J. Hellings. Conjunctive context-free path queries. 2014.

[7] 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.

[8] A. Johnstone and E. Scott. Modelling gll parser implementations. In *International Conference on Software Language Engineering*, pages 42–61. Springer Berlin Heidelberg, 2010.

[9] J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.

[10] E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

[11] E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.

[12] E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.

[13] E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: a cubic tomita-style glr parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.

[14] P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2):100, 2008.

[15] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Springer, 2012.

[16] 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.

[17] E. Verbitskaia, S. Grigorev, and D. Avdyukhin. Relaxed parsing of regular approximations of string-embedded languages. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 291–302. Springer International Publishing, 2015.

APPENDIX

A. GLL PSEUDOCODE

Main functions of GLL parsing algorithms:

- stack and descriptors manipulation functions 3;
- SPPF construction functions 4.
- R—working set which contains descriptors to process.
- U—all descriptors was created.
- P—popped nodes.

Algorithm 3 Stack and descriptors manipulation functions

```
1: function ADD(L, v, i, a)
2:   if (L, v, i, a) ∉ U then
3:     U.add(L, v, i, a)
4:     R.add(L, v, i, a)
5: function POP(v, i, z)
6:   if v ≠ v0 then
7:     P.add(v, z)
8:     for all (a, u) ∈ v.outEdges do
9:       y ← getNodeP(v.L, a, z)
10:      add(v.L, u, i, y)
11: function CREATE(L, v, i, a)
12:   if (L, i) ∉ GSS.nodes then
13:     GSS.nodes.add(L, i)
14:   u ← GSS.nodes.get(L, i)
15:   if (u, a, v) ∉ GSS.edges then
16:     GSS.edges.add(u, a, v)
17:     for all (u, z) ∈ P do
18:       y ← getNodeP(L, a, z)
19:       (_, _, k) ← z.lbl
20:       add(L, v, k, y)
      return u
```

Algorithm 4 SPPF construction functions

```
1: function getNodeT(x, i)
2:   if x = ε then
3:     h ← i
4:   else
5:     h ← i + 1
6:   if (x, i, h) ∉ SPPF.nodes then
7:     SPPF.nodes.add(x, i, h)
     return SPPF.nodes.get(x, i, h)
8: function getNodeP((X → ω1 · ω2), a, z)
9:   if ω1 is terminal or non-nullable nonterminal and ω2 ≠
   ε then return z
10:  else
11:    if ω2 = ε then
12:      t ← X
13:    else
14:      h ← (→ ω1 · ω2)
15:    (q, k, i) ← z.lbl
16:    if a ≠ dummy then
17:      (s, j, k) ← a.lbl
18:      y ← findOrCreate SPPF.nodes (n.lbl = (t, i, j))
19:      if y does not have a child with label (X → ω1 · ω2)
   then
20:        y′ ← newPackedNode(a, z)
21:        y.chld.add y′
22:        return y
23:      else
24:        y ← findOrCreate SPPF.nodes (n.lbl = (t, k, i))
25:        if y does not have a child with label (X → ω1 · ω2)
   then
26:          y′ ← newPackedNode(z)
27:          y.chld.add y′
28:          return y
     return SPPF.nodes.get(x, i, h)
```