Generalized LL parsing for context-free constrained path search problem

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

Aaaabstract is very abstract.... word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10 word1 word2 word3 word4 word5 word6 word7 word8 word9 word10

1. INTRODUCTION

Graph data model and graph data bases are very popular in many different areas such as bioinformatic, semantic web, social networks etc. Extraction of paths satisfying specific constraints may be useful for graph structured data investigation and for relations between data items detection. Path querying with constrains formulated in terms of formal grammars is a specific problem named formal language constrained path problem [3] and research in this area is still actual [8]. !! Information from different areas such as bioinformatic, semantic web, social networks can be represened in graph model. Moreover, there are data graph data bases. (?) One of the common graph problem is paths extraction from graph. Paths must satisfy specific constrains and the search must use a reasonable time. (?) !!!

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

The moste solution in DB area use such parsing algorithms as CYK or Earley. In string-embedded languages analysis (RN)GLR is used. It has better time complexity. !!! Parsing technique are used in DB previously. Usually this aproach use CYK or Earley algorithms. (WHAT PROBLEM and why string embedded lang?) ? , , , , . !!! ? Complexity is $O(n^3)$ in worst case and linear for unambiguous grammars, that better than complexity of CYK and Earley which used as base in other solutions (for example [5], [16]). This fact allows to demonstrate better performance on linear subgraphs and unambiguous grammars. Also it is not necessary to transform input grammar to CNF which required for CYK which allows to avoid grammar size increasing. It is important because real performance of parsing algorithm is sensetive to grammar size. !!! The algorithm allows to process ambigious grammar and it is not necessary to transform grammar to CNF which increases grammar size. It is important because real performance of parsing algorithm is sensetive to grammar size. !! ! , Graph parsing can be also used in string-embedded languages processing. Regular approximation for value set of string variable can by represened as directed graph of related fite automata. !!! ”as directed graph of related fite automata.”!

In order to check corectness or safety (sql injections)... all generated strings (all paths from start states to final states) are correct w.r.t some context-free grammar. For example grammar of one of SQL dialects. GLR-based for string-embedded SQL checking [2, 4]. Solution based on RNGLR [11] for relaxed parsing of string-embedded languages [20] which allow to find all path between two specified vertices.

Despite of the fact that there is set of path quering solutions [16, 5, **?**], query result exploration still a challenge [6]. Complex query debugging also is a problem. !!! ??! To solve these problems structural represenatation of query result can be useful, and classical parsing thechniques allow to construct such representation: derivation tree contains full information abaut parsed sentence structure in terms of specified grammar.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Parsing technique allows to create structural representation of the query results. derivation tree contains full information abaut parsed sentence structure in terms of specified grammar. It simplify debugging process. !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

In this paper, we propose graph parsing technique which allows to construct structural representation of query result with relation to grammar query or derivation of result.

Proposed algorithm is based on generalised top-down parsing algorithm — GLL. LL parsers are easier than LR parser, is more natural and so on. !!!!!!!!!!!!!!!

2. PRELIMINARIES

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

Also we need to introduce some definitions.

- Context-free grammar $G = (N, \Sigma, P, S)$ 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 productions set.
- $\mathcal{L}(G)$ is a language specified by grammar G.
- Directed graph $M = (V, E, L)$ where V — vertices set, $L \subseteq \Sigma$ — edge labels set, $E \subseteq V \times L \times V$. We assume that there are no parallel edges with equal labebs: 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 for edge's tag calculation .

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

- $\oplus : L^+ \times L^+ \to L^+$ is a concatenation operation.
- Path p in graph M.

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

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 for calculation string produced by path.

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

As a result we can define that context-free language constrained path querying meens that we get query as grammar G and result of this query is a set of paths

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

For some graphs and some queries P can be infinite set, and it can not be explicitly represented. In order to solve this problem, in this paper, we will construct not explicit representation of P but compact data structure which store all elements of P in finite space and allow to extract any of them. In this point our solution is slightly similar to subgraph quering proposed in article [16], but we are also construct derivetion forest for result subgraph.

3. MOTIVATING EXAMPLE

In this article we are discuss context-free constrained path quering, and one of well-known not regular but context-free language is an language

$$\mathcal{L} = \{A^n B^n; n \geq 1\} = \{AB; AABB; AAABBB; \ldots\}$$

. This language is a subset of balanced brackets langage and in practice may be used for description many different relations: n-th generation in parent-child, any open should be closed in correct order, etc..

As a motivation of context-free constraints importance let we introduce the next example. Let we have graph $M = (\{0; 1; 2; 3\}, E, \{A; B\})$ presented in figure 1 where labels represent next relations:

each time when it opend it should be closed in future.

Suppose for each $n \geq 1$ we want to find all n-th generation friends with a common ancestor. In the other worlds, we wath to find all paths p, such that $\Omega(p) \in \{AB; AABB; AAABBB; \ldots\}$ or $\Omega(p) = A^n B^n$ where $n \geq 1$. This constraint can not be specified with regular language as far as $L = \{A^n B^n; n \geq 1\}$ is not regular but context free. Required language can be specified by grammar G_1 presented in picture 2 where $N = \{s; middle\}$, $\Sigma = \{A; B\}$, and $S = s$.

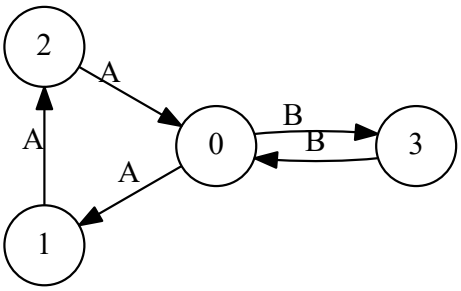


Figure 1: Input graph M

```
0: s = L s R
1: s = middle
2: middle = L R
```

Figure 2: Grammar G_1 for language $L = \{L^n R^n; n \geq 1\}$

Result is infinite for this query, and we can not... Also we want to know, who is common accesor. Futher we show ho to solve it.

4. GRAPH PARSING ALGORITHM

We propose a context-free language constrained path problem solution which allows to find all paths in graph. Paths are satisfied specified arbitrary context-free grammar. The algorithm constructs implicit representation of result. The results are represented with parsing forest of all possible parsing trees. Finite representation of result set with structure related to 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) [12, 1] parsing algorithm which allows to process ambiguous context-free grammars.

4.1 Generalized LL Parsing Algorithm

States

-
-
-
-

Sets

-
-
-
-

Generalized LL (GLL) is generalized top-down parsing algorithm which handle all context-free grammars (including left recursive) with worst-case cubic time complexity and linear for LL grammars. GLL is native for grammar, can be simple created !!!!!!! and debugging. Generalised algorithms look through all possible derivation in grammar for input. If current parsing branch is wrong the analisys (!??) process cotinious with others. GLL uses descriptors mechanism to store all parsing branches. Descriptors are four (!!????!!) elements which fully (??!!!) describes current parser state. Descriptor is a quadriple (L, s, j, a) where L is a line label, s is a stack node, j is a position in the input, and a is a node of derivation tree. GLL parsers, like recursive descent parsers, consist of functions for every nonterminal and one dispatching function. Every function has label and a function gets control from another function due the call by the name. Process of analisys consist of calling function and tarts from the function for start nonterminal. !!!!!!!!!!!!!!! , !!!!!!!!! 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 stop work. As previously mentioned, generalised parsers process all possible derivation branches. For every branch parser must store it's own stack. It leads to OOM. !!!! Graph structured stack (GSS) [18] is used to solve this problem. GSS allows to combine stacks to prevent duplication. Stacks with common part are combined to one graph structure and it stores only one node for evey analysis branch except whole stack. It allows to reduce () memory significantly. In GLL each GSS node contains pair — position in input and grammmar slot. Grammar slot is like LR-slot. Slot is a grammar rule and position in it (for example !!!!).

The next part of the descriptor is a tree node. Parsers build derivation tree using input and grammar. There are more than one tree for ambigious grammar and generalised algorithms builds all derivation trees. Special data structure – SPPF – is used to reduce space required for tree storage.

We use table version [**?**] instead of code generation.

4.2 Shared packed parse forest

Shared Packed Parse Forest (SPPF) [10] is a spetial data structure for derivation forest compact representation which allow to reuse common nodes and subtrees. As a result multiple derivation trees, which can be produced in case of ambiguos grammar, can be compressed in one SPPF with optimal reusing of common parts. Binarized form of SPPF proposed in [15] and it allow to achive worst-case cubic space complexity. GLL can use SPPF [13] for results representation achive cubic space complexity with binarised version.

Let we present an example of SPPF for ambiguos grammar G_0 (pic 3).

Here `N` is token for number, `L` and `R` are tokens for '(' and ')' respectively.

Let we parse the sentence `"()()()"`. There are two diferent lefmost derivations of this sentence in grammar G_0 ($\rightarrow^n$ denote an application of production with nimber n):

Algorithm 1 Control functions

```
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 _
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
```

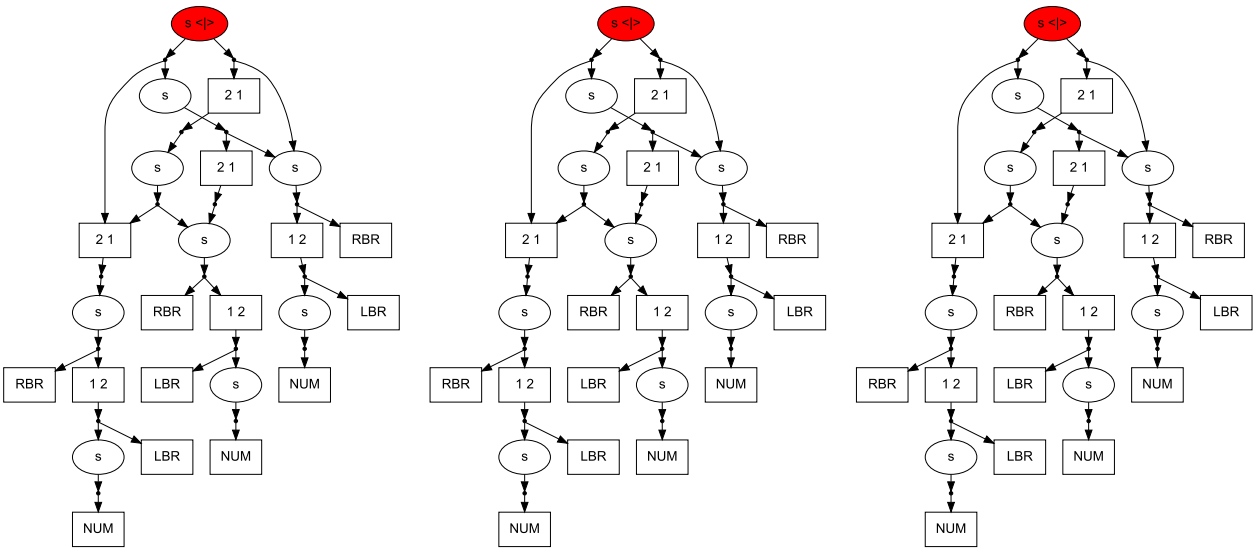
1. $s \rightarrow^2 ss \rightarrow^2 sss \rightarrow^1 LsRss \rightarrow^0 LNRss \rightarrow^1 LNRLsRs \rightarrow^1 LNRLsRs \rightarrow^0 LNRLNRs \rightarrow^1 LNRLNRLsR \rightarrow^0 LNRLNRLNR$
2. $s \rightarrow^2 ss \rightarrow^1 LsRs \rightarrow^0 LNRs \rightarrow^2 LNRss \rightarrow^1 LNRLsRs \rightarrow^1 LNRLsRs \rightarrow^0 LNRLNRs \rightarrow^1 LNRLNRLsR \rightarrow^0 LNRLNRLNR$

As far as there are tho different derivations, SPPF should contains two different trees and it is presented in figure 4: result SPPF(fig. 4a) and trees for derivation 1(fig. 4b) and derivation 2(fig. 4c) respectively.

Binarised SPPF can be represented as a graph where each node has one of four types:

- terminal node with label (i, T, j);
- nonterminal node with label (i, N, j);
- intermidiate node with label (t, i, j) where t is a grammar slot;
- packed node with label $(N : \gamma\cdot, k)$;

, and one of nodes can be marked as 'root' — node for start nonterminal.



(a) SPPF (b) Tree for derivation 1 (c) Tree for derivation 2

Figure 4: SPPF for sentence `"(1)(2)(3)"` and grammar G_0

```
0: s = NUM
1: s = L s R
2: s = s s
```

Figure 3: Grammar G_0

Further in our examples we will remove redudant intermidiate and packed nodes from SPPF to simplify it and decrease size of structure.

4.3 GLL-based graph parsing

In order to use GLL for graph parsing we need only use graph vertices as position in input. After that we should modify **Processing** function such that

We implement some optimizations: [1]

We also use binarised SPPF for result representation whish allow to simplify query debugging and result explortion. (!!!!!! ?!!!!!!!) In our case more then one root may be specified. For example, look at picture!!!! We

$\mathbb{P} : G, M, StartVset, FinalVSet \rightarrow SPPF$ In details, main function input is graph M, set of start vertices $V_s \subseteq V$, set of final vertices $V_f \subseteq V$, grammar G_1. Output is Shared Packed Parse Forest (SPPF) [10] — finite data structure which contains all derivation trees for all paths in M, $\Omega(p) \in L(G_1)$ and allows to reconstruct any of paths implicitly. As far as we can specify sets of start and final vertices, our solution can find 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 allow to get more useful information about result. Binarized SPPF is at most cubic in terms of result size. Any path can be extracted in the linear time.

A bit more on corectnes.!!!!!

4.4 Complexity

Time complexity estimation in terms of input graph and grammar size is pretty similar to estimation of GLL complexity provided in [13].

Algorithm 2 Control functions

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

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 riginal GLL in [13] because main function used for descriptor creation are the same as original one. □

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 the same as the proof of **Theorem 2** from [13].

□

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 we estimate number of nodes of each type.

- Terminal nodes. Each of them has label of form (T, v_0, v_1), and such lable 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 tere are at most $|E|$ terminal nodes.
- ε nodes labled with (ε, v, v), hence there are at most $|E|$ of these.
- Nonterminal nodes have label of form (N, v_0, v_1), so there are at most $O(|V|^2)$ of these.
- Intermideate nodes have label of form (t, v_0, v_1), where t is grammar slot, so there are at most $O(|V|^2)$ of these.
- Packed nodes are children of intermediate or nonterminal nodes and have label of form (t, v) where t is a grammar slot $N : \alpha \cdot \beta$. 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 requerd 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 are the same as in proof of **Theorem 4** from [13] except *processing* function where we should process not one next input token, but all 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.

So, 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 $v \in V deg^+(v) \leq 1$, so $\max_{v \in V}(deg^+(v)) = 1$ and we get $O(|V|^3)$. For LL grammars and linear input complexity should be $O(|V|)$ for the same reason as for original GLL.

As discussed in [7] achiving of theoretical complexity reqired special datastructures which can be irrational for practice implementation and it is necessary to finde 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 we introduce the next example. Grammar G_1 is a query and we want to find all paths in graph M (presented in picture 1) matched this query. Result SPPF for this input is presented in picture 5. Note that presented version does not contein obsolete nodes.

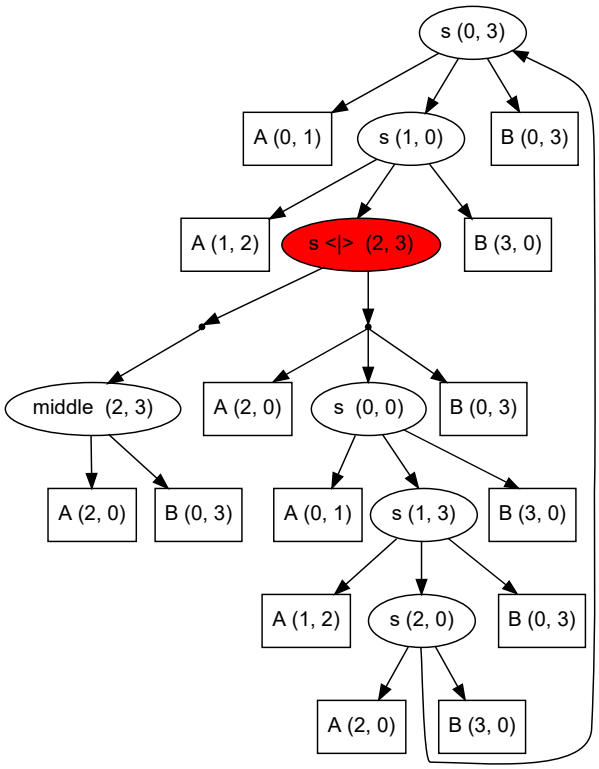


Figure 5: Result SPPF for input graph M(pic. 1) and query G_1(pic. 2)

We use next markers for nodes which similar to original SPPF but have some additional information in order to relation with graph.

- Node with rectangle shape labeled with (v_0, T, v_1) is terminal node. Each terminal node corresponds with edge in the input graph: for each node with label

(v_0, T, v_1) there is $e \in E : e = (v_0, T, v_1)$. Duplication of terminal nodes is only for figure simplification.

- Node with oval shape labeled with (v_0, nt, v_1) is nonterminal node. This node denote that there is at least one path p from vertex v_0 to vertex v_1 in input graph M such that $nt \Rightarrow^*_G \Omega(p)$. All paths matched this condition can be extracted from SPPF by left-to-right top-down graph traversal started from respective node.
- Filled node with oval shape labeled with $(< | > (v_0, nt, v_1))$ is nonterminal node denote that there are more then one path p from v_0 to v_1 such that $nt \Rightarrow^*_G \Omega(p)$.
- Node with dot shape is used for representation of derivation variants. Subgraph with root in one such node is one variant of derivation. Parent of such nodes is always node with label $(\lhd\rhd (v_0, nt, v_1))$.
- v_0 and v_1 are left and right extensions of node respectively.

As an example of derivation structure usage we can find 'middle' of any path in example above simply by finding corresponded nonterminal *middle* in SPPF. So we can found that there is only one common ancestor for all results, and it is vertex with $id = 0$.

Extensions stored in nodes allow to check whether path from u to v exists, and extract it. To extract specified path we need only travers SPPF, and it can be done in linear time (in terms of SPPF size).

Let for example we want to 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 reqired label: $(0, s, 0)$ and $(0, s, 3)$. Next step let we try to extract corresponded paths from SPPF. In our example there is 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 constructed by described algorithm can be useful for query result investigation. But in some cases explicit representation of matched subgraph may be preferred, and requred subgraph may be extracted from SPPF trivially by its traversal.

5. EVALUATION

We use two grammars for balanced brakets — ambiguos grammar G_0 3 and unambiguos grammar G_2 6 — in order to investigate performance and grammar ambiguity correlation.

```
0: s = L s R s
1: s = eps
```

Figure 6: Unambiguos grammar G_2 for balanced brackets

For input we use complete graphs where for each termial symbol there is edge between two vertices labeled with it. Note that we use only terminal symbols for edges labels. 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 quering in terms of required resources because there are correct path between any two vertices and result set is infinite.

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 mesurament results presented in figure 7. As far as for complete graph $M = (V, E, L)$

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

where Σ is terminals of input grammar, we should get time complexity at most $O(|V|^4)$ and space complexity at most $O(|V|^3)$. For time mesurement results we have that all two curves can be fit with polinomial function of degree 4 to a high level of confidence with R^2.

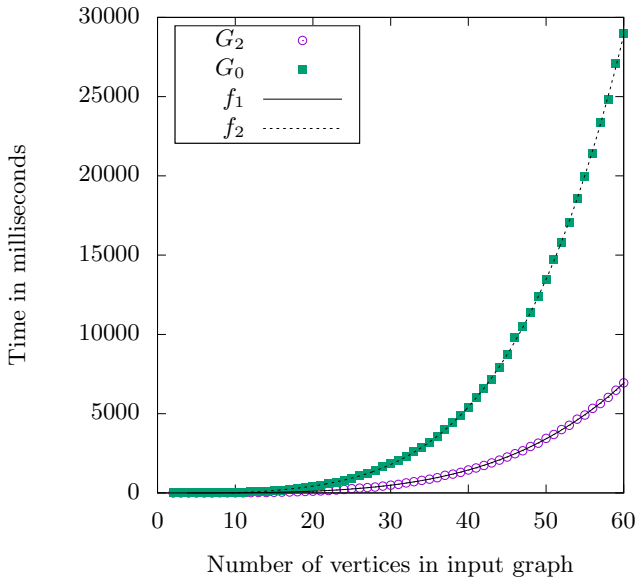


Figure 7: Performance on complete graphs for grmmars G_0 and G_2
$f_1(x) = 0.000495989 * x^4 + 0.001252184 * x^3 + 0.068491746 * x^2 - 0.306749160 * x$; $R^2 = 0.99996$
$f_2(x) = 0.003368883 * x^4 - 0.114919298 * x^3 + 3.161793404 * x^2 - 22.549491142 * x$; $R^2 = 0.99995$

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 $R^2 = 1$.

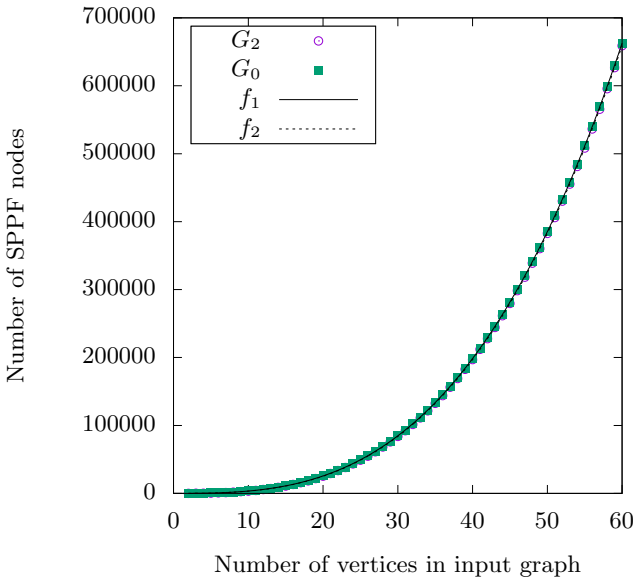


Figure 8: SPPF size on complete graph for grmmars G_0 and G_2 an complete graphs
$f_1(x) = 3.000047 * x^3 + 3.994579 * x^2 + 4.191568 * x;\ R^2 = 1$
$f_2(x) = 3.000050 * x^3 + 2.994338 * x^2 + 4.196472 * x;\ R^2 = 1$

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 query debugging. Presented algorithm implemented in F# [17] and available on GitHub:https://github.com/YaccConstructor/YaccConstructor.

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

Also we are working on performance improvement by implementation of recently proposed modifications in original GLL algorithm [14]. One of direction of our research is generalization of grammar factorization proposed in [14] which may be useful for regular query processing.

We are working on utilisation of GPGPU and multicore CPU power for graph parsing problem with Valiant [19] algorithm modification proposed by Alexander Okhotin [9]. One of possible benefit is ability to process more expressive queries because modification proposed by Alexander Okhotin extended to support boolean grammars.

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

[6] 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.

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

[8] J. A. Miller, L. Ramaswamy, K. J. Kochut, and A. Fard. Research directions for big data graph analytics. In *2015 IEEE International Congress on Big Data*, pages 785–794. IEEE, 2015.

[9] A. Okhotin. Parsing by matrix multiplication generalized to boolean grammars. *Theoretical Computer Science*, 516:101–120, 2014.

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

[11] E. Scott and A. Johnstone. Right nulled glr parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):577–618, 2006.

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

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

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

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

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

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

[18] 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.

[19] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–315, 1975.

[20] 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

Algorithm 3 Single vertex processing

```
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 ≠ v_0 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 Single vertex processing

```
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 → ω_1 · ω_2), a, z)
9:     if ω_1 is terminal or non-nullable nonterminal and ω_2 ≠ ε then return z
10:    else
11:        if ω_2 = ε then
12:            t ← X
13:        else
14:            h ← (→ ω_1 · ω_2)
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 → ω_1 · ω_2) 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 → ω_1 · ω_2) then
26:                    y' ← newPackedNode(z)
27:                    y.chld.add y'
28:                    return y
       return SPPF.nodes.get(x, i, h)
```