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 represented 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 querying, 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 approach 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 sensitive to grammar size. !! The algorithm allows to process ambiguous 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 sensitive to grammar size. !! ! , Graph parsing can be also used in string-embedded languages processing. Regular approximation for value set of string variable can by represented as directed graph of related finite automata. !!! "as directed graph of related fite automata."!

In order to check correctness 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 querying solutions [16, 5, **?**], query result exploration still a challenge [6]. Complex query debugging also is a problem. !!! ??! To solve these problems structural representation of query result can be useful, and classical parsing techniques allow to construct such representation: derivation tree contains full information about parsed sentence structure in terms of specified grammar.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Parsing technique allows to create structural representation of the query results. derivation tree contains full information about 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 generalized 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 a parsing algorithm, and not on the data representation, and we assume that full input graph can be located in RAM memory in 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 set of productions.
- $\mathcal{L}(G)$ 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 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 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 is a list of edges:
$$\begin{aligned} p &= (v_0, l_0, v_1), (v_1, l_1, v_2), \dots, (v_{n-1}, l_{n-1}, v_n) \\ &= e_0, e_1, \dots, e_{n-1} \end{aligned}$$
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.
$$\begin{aligned} &\Omega(p = e_0, e_1, \dots, e_{n-1}, p \in P) = \\ &tag(e_0) \oplus \dots \oplus tag(e_{n-1}). \end{aligned}$$

As a result we can define that context-free language constrained path querying means 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 compact data structure which stores all elements of P in finite space and allows to extract every of them. In this point our solution is slightly similar to subgraph querying proposed in article [16], but we also construct derivation forest for result subgraph.

3. MOTIVATING EXAMPLE

In this article we are discuss context-free constrained path querying, and one of well-known not regular but context-free language is an language
$$\mathcal{L} = \{A^n B^n; n \geq 1\} = \{AB; AABB; AAABBB; \dots\}$$
. This language is a subset of balanced brackets language 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.. each time when it opened it should be closed in future.

Let we have graph $M = (\{0; 1; 2; 3\}, E, \{A; B\})$ presented in figure 1 where labels represent one of mentioned relations. We want to find all paths p, such that $\Omega(p) \in \{AB; AABB; AAABBB; \dots\}$ or $\Omega(p) \in A^n B^n$ where $n \geq 1$. Required language can be specified by grammar G_1 presented in picture 2 where $N = \{s; middle\}$, $\Sigma = \{A; B\}$, and $S = s$.

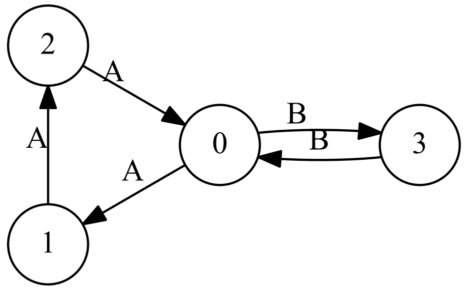


Figure 1: Input graph M

```
0: s = A s B
1: s = middle
2: middle = A B
```

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

Result of presented query for given graph is infinite set of path, hence it can not be constructed explicitly. More over, for some tasks it can be necessary to get structure of result w.r.t. given grammar. Further we show ho to get finite representation of query result structure in terms of derivation in grammar.

4. GRAPH PARSING ALGORITHM

We propose a context-free language constrained path problem solution which allows to create finite representation of parse forest which contains trees for all satisfied paths in graph. 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 arbitrary (including left-recursive and ambiguous) context-free grammars with worst-case cubic time complexity and linear for LL grammars..

4.1 Generalized LL Parsing Algorithm

In classical LL algorithm we have pointer in input and pointer in grammar of form $n \to \alpha \cdot \beta$ — grammar slot.

1.
2.
3.
4.

In case (2) we can use $FIRST$ set to choose single variant. But sometimes it is not possible to select only one path to continue parsing and it is do not allow to use LL parsing algorithm. Generalized LL algorithm handle all possible paths in this case. Instead of immediate processing of all variants GLL uses descriptors mechanism to store all possible branches and process them sequentially. Descriptor is a quadriple (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.

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, generalized parsers process all possible derivation branches and for every branch parser must store it's own stack. It leads to infinite stack grows. Tomita-style graph structured stack (GSS) [18] allows to combine stacks to solve this problem. In GLL each GSS node contains is a pair of position in input and grammar slot.

Detailed description of GLL parsing algorithm is available in this articele [?]. Pseudocode of stack an tree manipulation functions can be found in Appendix A.

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

Algorithm 1 Control functions

```
function DISPATCHER
    if R.Count ≠ 0 then
        (L, v, i, cN) ← R.Get()
        cR ← dummy
        dispatch ← false
    else
        stop ← true
function PROCESSING
    dispatch ← true
    switch L do
        case (X → α · xβ) where x = input[i + 1])
            if cN = dummyAST then
                cN ← getNodeT(i)
            else
                cR ← getNodeT(i)
            i ← i + 1
            L ← (X → αx · β)
            if cR ≠ dummy then
                cN ← getNodeP(L, cN, cR)
            dispatch ← false
        case (X → α · xβ) where x is nonterminal
            v ← create((X → αx · β), v, i, cN)
            slots ← pTable[x][input[i]]
            for all L ∈ slots do
                add(L,v,i,dummy)
        case (X → α·)
            pop(v,i,cN)
        case (S → α·) when S is start nonterminal
            final result processing and error notification
function CONTROL
    while not stop do
        if dispatch then
            dispatcher
        else
            processing
```

There are more than one tree for ambiguous grammar and generalized algorithms builds all derivation trees. Special data structure — SPPF — is used to reduce space required for tree storage.

4.2 Shared packed parse forest

Shared Packed Parse Forest (SPPF) [10] is a special 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 ambiguous grammar, can be compressed in one SPPF with optimal reusing of common parts. Binarized form of SPPF proposed in [15] and it allow to achieve worst-case cubic space complexity. GLL can use SPPF [13] for results representation achieve cubic space complexity with binarised version.

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

```
0: s = eps
1: s = A s B
2: s = s s
```

Figure 3: Grammar G_0

Let we parse the sentence `"ABABAB"`. There are two different leftmost derivations of this sentence in grammar G_0, hence 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 which described below with corresponded graphical notation.

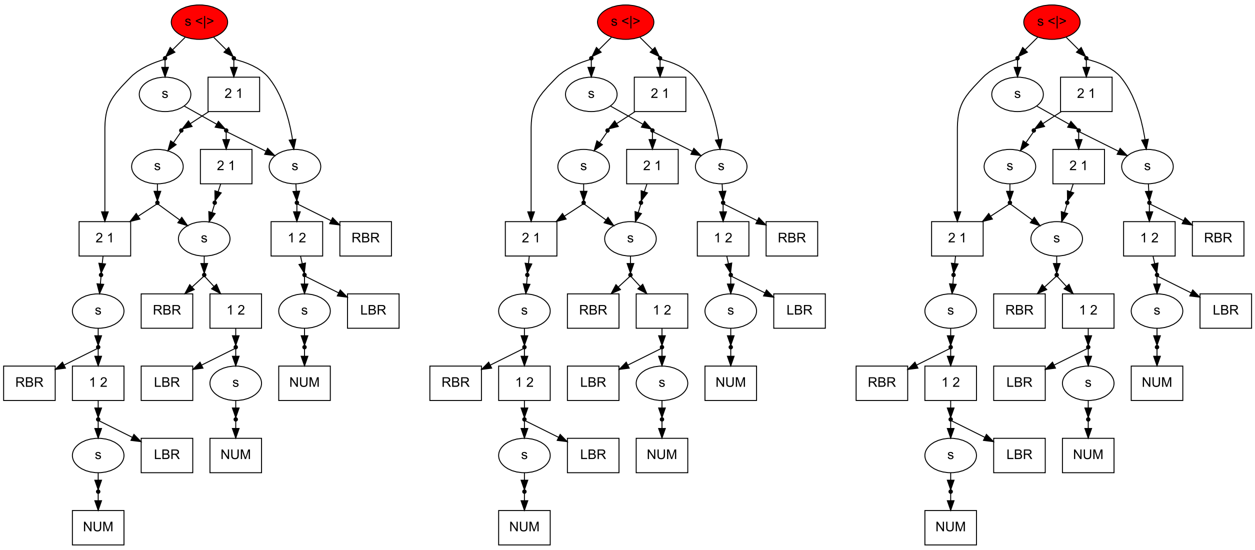
- Node with rectangle shape labeled with (i, T, j) is terminal node.
- Node with oval shape labeled with (i, N, j) is nonterminal node. This node denote that there is at least one derivation for substring α from position i to position j in input string ω such that $N \Rightarrow_G^* \alpha, \alpha = \omega[i..j-1]$. 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 $(\diamondsuit\ (i, N, j))$ for denote that there are more then one derivations from nonterminal N for substring from i to j.
- Intermediate node with label (i, t, j) where t is a grammar slot. We use dot shape for these nodes and omit label because it is important only for SPPF constriction. Subgraph with root in such node is one variant of derivation in case when parent is nonterminal node with label $(\diamondsuit\ (i, N, j))$.
- Node with rectangle shape and label $(N : \gamma\cdot, k)$ is a packed node.

One of nonterminal nodes can be marked as 'root' — node for start nonterminal. Tuple of positions (i, j) which represent start and end of substring is *extension* of node.

Further in our examples we will remove redundant intermediate 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



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

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

Algorithm 2 Control functions

function PROCESSING
  $dispatch \leftarrow true$
  switch L **do**
    case $(X \rightarrow \alpha \cdot x\beta)$ where $x = input[i+1]$)
      if $cN = dummyAST$ **then**
        $cN \leftarrow$ GETNODET(i)
      else
        $cR \leftarrow$ GETNODET(i)
      $i \leftarrow i+1$
      $L \leftarrow (X \rightarrow \alpha x \cdot \beta)$
      if $cR \neq dummy$ **then**
        $cN \leftarrow$ GETNODEP(L, cN, cR)
    case $(X \rightarrow \alpha \cdot x\beta)$ where x is nonterminal
      $v \leftarrow$ CREATE$((X \rightarrow \alpha x \cdot \beta), v, i, cN)$
      $slots \leftarrow \bigcup_{e \in input.OutEdges(i)} pTable[x][e.Token]$
      for all $L \in slots$ **do**
        ADD(L,v,i,dummy)
    case $(X \rightarrow \alpha \cdot)$
      POP(v,i,cN)
    case _
      final result processing and error notification

We implement some optimizations: [1]

We also use binarised SPPF for result representation which allow to simplify query debugging and result exploration. (!!!!!! ?!!!!!!!) 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 SPPF which contains all derivation trees for all paths p in M, such that $\Omega(p) \in L(G_1)$. 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 linear time.

A bit more on correctness.!!!!!

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].

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 [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 us estimate number of nodes of each type.

- Terminal nodes. Each of them has label of form (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 labeled 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.
- Indeterminate 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 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 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] achieving of theoretical complexity required special data structures which 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 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 contains obsolete nodes. Each terminal node corresponds with edge in the input graph: for each node with label (v_0, T, v_1) there is $e \in E : e = (v_0, T, v_1)$. We duplicate terminal nodes only for figure simplification.

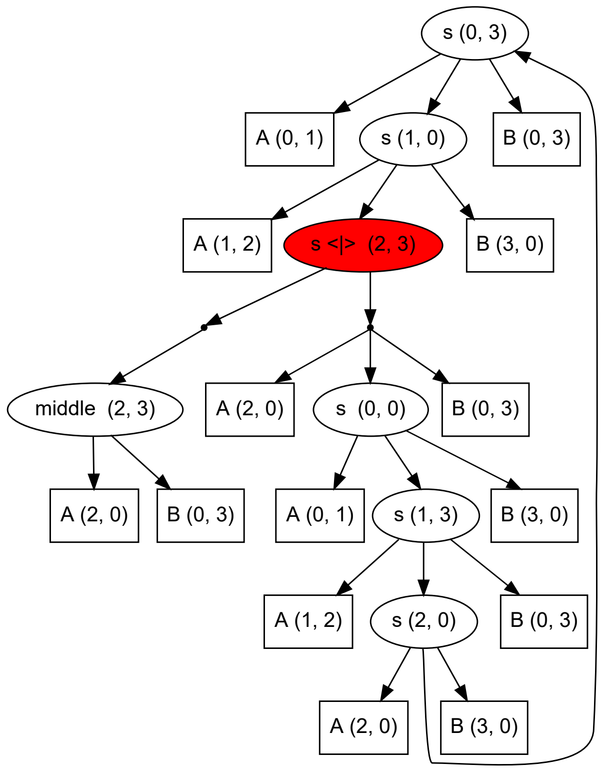


Figure 5: Result SPPF for input graph M(pic. 1) and query G_1(pic. 2)

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 traverse 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 required 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 required subgraph 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 worst case of time and space complexity can be achieved.

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

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

Figure 6: Unambiguous grammar G_2 for balanced brackets

For input we use complete graphs where for each terminal symbol there is edge between every 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 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 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 $R^2 = 1$.

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

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

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 assembly querying, 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.

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

- R — working set. Descriptors to process.
- U — all descriptors was created. Avoid duplication.
- P — popped nodes. Allows to process

Algorithm 3 Single vertex processing

```
function ADD(L, v, i, a)
    if (L, v, i, a) ∉ U then
        U.add(L, v, i, a)
        R.add(L, v, i, a)
function POP(v, i, z)
    if v ≠ v0 then
        P.add(v, z)
        for all (a, u) ∈ v.outEdges do
            y ← GETNODEP(v.L, a, z)
            ADD(v.L, u, i, y)
function CREATE(L, v, i, a)
    if (L, i) ∉ GSS.nodes then
        GSS.nodes.add(L, i)
    u ← GSS.NODES.GET(L, i)
    if (u, a, v) ∉ GSS.edges then
        GSS.edges.add(u, a, v)
        for all (u, z) ∈ P do
            y ← GETNODEP(L, a, z)
            (_, _, k) ← z.lbl
            ADD(L, v, k, y)
    return u
```

Algorithm 4 Single vertex processing

```
function GETNODET(x, i)
    if x = ε then
        h ← i
    else
        h ← i + 1
    if (x, i, h) ∉ SPPF.nodes then
        SPPF.nodes.add(x, i, h)
    return SPPF.nodes.get(x, i, h)
function GETNODEP((X → ω1 · ω2), a, z)
    if ω1 is terminal or non-nullable nonterminal and ω2 ≠ ε then return z
    else
        if ω2 = ε then
            t ← X
        else
            h ← (→ ω1 · ω2)
        (q, k, i) ← z.lbl
        if a ≠ dummy then
            (s, j, k) ← a.lbl
            y ← findOrCreate SPPF.nodes (n.lbl = (t, i, j))
            if y does not have a child with label (X → ω1 · ω2) then
                y' ← newPackedNode(a, z)
                y.chld.add y'
                return y
            else
                y ← findOrCreate SPPF.nodes (n.lbl = (t, k, i))
                if y does not have a child with label (X → ω1 · ω2) then
                    y' ← newPackedNode(z)
                    y.chld.add y'
                    return y
    return SPPF.nodes.get(x, i, h)
```