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

Graph data model and graph data bases are very popular in many different areas such as bioinformatics, semantic web, social networks, etc. One of specific problems on graph DB is a path querying with constrains formulated in terms of formal grammars. There is a number of solutions, but building structural representation of query result which practical for answer understanding and exploration is still a problem. In this paper we propose graph parsing technique which allows to build such representation with respect to given grammar query for arbitrary context-free grammar and graph. Proposed algorithm is based on generalized top-down parsing algorithm which has cubic worst-case time complexity and linear for LL grammars.

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 satisfying specific constraints may be useful for graph structured data investigation and for relations between data items detection. One of specific problems is a path querying with constrains formulated in terms of formal grammars: 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 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 is used.

Existing solutions in DB area usually use such parsing algorithms as CYK or Earley(for example [5], [16]). These algorithms have nonlinear time complexity ($O(n^3)$ and $O(n^2)$ respectively), but there are such parsing algorithms as GLR and GLL, which have cubic worst-case complexity and linear for unambiguous grammars. Complexity is $O(n^3)$ in worst case and linear for unambiguous grammars, that better than complexity of CYK and Earley which are used as a base in other solutions (for example [5], [16]). This fact allows to demonstrate better performance in some cases, for example on linear subgraphs and unambiguous grammars. Also there is no need to transform the input grammar to Chomsky normal form which required for CYK which allows to avoid grammar size increasing, and to improve performance of parsing algorithm because it is sensitive to grammar size.

Despite the fact that there is a set of path querying solutions [16, 5, **?**], query result exploration is still a challenge [6]. Simplification of complex query debugging is also 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.

Graph parsing can be also used in dynamically generated strings or string-embedded languages processing. Regular approximation for value set of string variable can be represented as a finite automata. Moreover, if we want to check a correctness of dynamically generated strings, we should check that all generated strings (all paths from start states to final states in the given automata) are correct with respect to some context-free grammar. For example grammar of appropriate SQL dialect can be used for processing string-embedded SQL. There are some solutions that are addressed this problem: GLR-based checker of string-embedded SQL queries [2, 4]; relaxed parser of string-embedded languages [20] based on RNGLR parsing algorithm. The latest one allows to construct derivation forest for all correct paths in the input automata.

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

2. PRELIMINARIES

In this work we are focused on a 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.

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 is a set of vertices, $L \subseteq \Sigma$ is a set of edge's labels, and $E \subseteq V \times L \times V$. 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 edge's tag.

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

- $\oplus : L^+ \times L^+ \to L^+$ is a concatenation operation.

- Path p in graph M is a list of edges:

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

  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 allow to get a string produced by path.

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

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

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

Note that P can be an infinite set in some cases, and hence it cannot 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 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 to describe many different relations: n-th generation in parent-child, correct order of an open-close operations for resources, etc.

Let consider the graph $M = (\{0; 1; 2; 3\}, E, \{A; B\})$ is presented in figure 1 which 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 $S = s$.

Result of presented query for given graph is an infinite set of paths, hence it cannot be constructed explicitly. Moreover, for some tasks it can be necessary to get the structure of result with respect to given grammar. Further we show how to get finite representation of query result structure in terms of derivation in grammar.

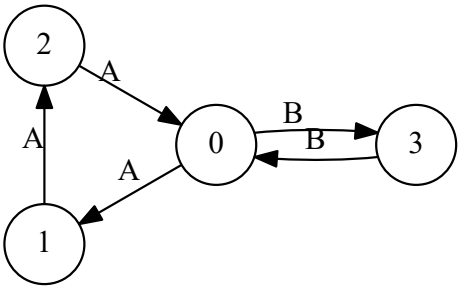


Figure 1: Input graph M

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

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

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 $n \to \alpha \cdot x\beta$ — grammar slot. Parsing may be described as related movement this pointers from initial position. Thus in each step we have two pointers and some possible cases to process them.

1. $n \to \alpha \cdot x\beta$ when x is a terminal and $x = input[i]$. In this case we should move both pointers to 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 save return address $n \to \alpha x \cdot \beta$ in stack and move pointer in grammar to position $x \to \cdot\gamma$.

3. $n \to \alpha\cdot$. This case means that we finish processing of nonterminal n. 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 check emptiness of input and report success or failure.

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 does 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.

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. It leads to infinite stack growth. Tomita-style graph structured stack (GSS) [18] allows to combine stacks to solve this problem. In GLL each GSS node contains a pair of position in input and grammar slot.

In order to provide termination and correctness we should avoid duplication of descriptors, and process GSS nodes correctly. It is necessary to use some additional sets for this.

- R — working set which contains descriptors to process. Algorithm finish when R is empty.
- U — all descriptors was created. Avoid duplication of these.
- P — popped nodes. Allows to process descriptors in arbitrary order without loosing new descriptors in step (3).

Instead of explicit code generation used in original paper, we use table version of GLL [**?**] and main control functions are different from original. Control functions of table based GLL are presented in listing 1. All other functions are the same as in original algorithm and their description can be found in original article [12] 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 [10] — is used to reduce space required for tree storage.

4.2 Shared packed parse forest

Binarized Shared Packed Parse Forest (SPPF) [15] allow to compress derivation trees with optimal reusing of common nodes and subtrees. Version of GLL which uses this structure for parsing forest representation achieve worst-case cubic space complexity [13].

Let we present an example of SPPF for the input sentence `"ABABAB"` and ambiguous grammar G_0 (pic 3).

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

Figure 3: Grammar G_0

There are two different leftmost derivations of given 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.

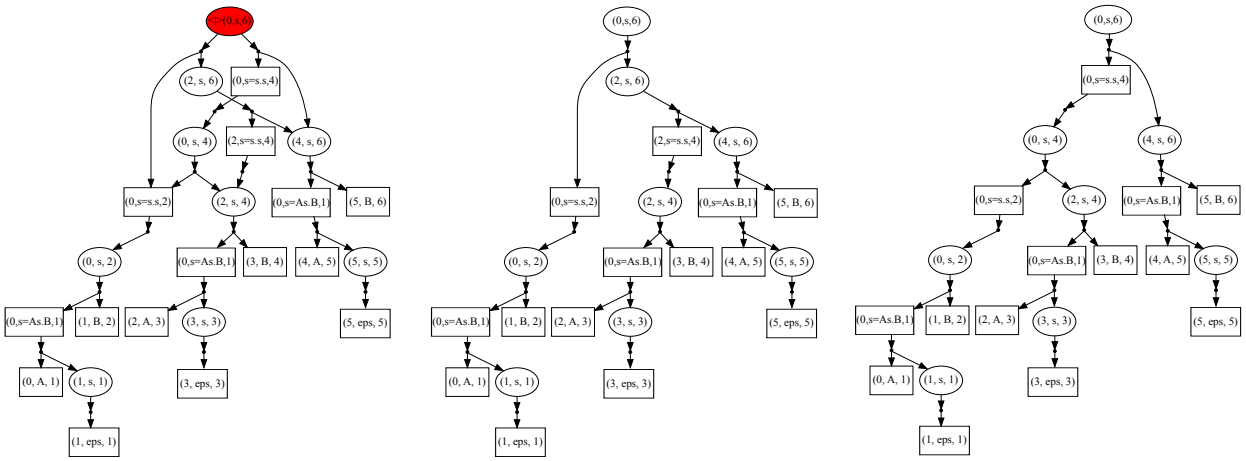
- Node with rectangle shape labeled with (i, T, j) is terminal node.

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
```

- Node with oval shape labeled with (i, N, j) is 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 ($\diamond$ (i, N, j)) for denote that there are more then one derivations from nonterminal N for substring from i to j.
- Packed 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 ($\diamond$ (i, N, j)).
- Node with rectangle shape and label $(N : \gamma\cdot, k)$ is an intermediate node.

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



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

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

Later 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 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 $V_f \subseteq V$, grammar G_1, it returns SPPF which contains all derivation trees for all paths p in M, such that $\Omega(p) \in L(G_1)$, and $p.start \in V_s, p.end \in V_f$.

First of all notice that input string for classical parser is a linear graph, and positions in input is vertices of this graph. This observation can be generalized to arbitrary graph with remark that in the position we have not only one next symbol, but set of labels of all outgoing edges for given vertex. Thus in order to use GLL for graph parsing we need to use graph vertices as position in input and modify **Processing** function to allow to process more then one "next symbol". Required modifications presented in listing 2. Small modification also required for initialization of R set: it is necessary to add not only one initial descriptor but set of descriptors for all vertices in V_s. All other functions can be reused from original algorithm without any changes.

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.

Note that termination of proposed algorithm is inherited from basic GLL algorithm. We are working with finite graphs, hence set of positions is finite, and tree construction is not changed. So, 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 [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.

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
```

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 us present a solution of task introduced in section 3: 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 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.

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 ancestor for all results, and it is vertex with $id = 0$.

Extensions stored in nodes allow us 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).

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 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 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

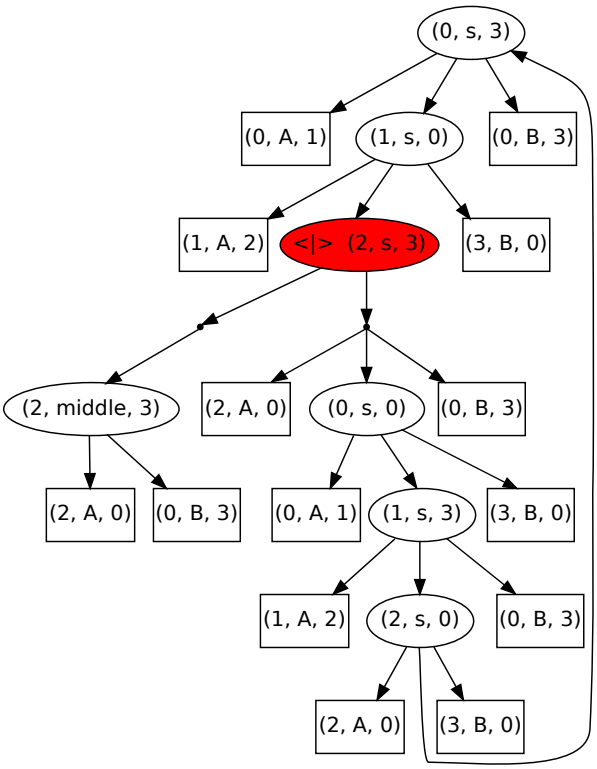


Figure 5: Result SPPF for input graph M(pic. 1) and query G_1(pic. 2)

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. 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 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.

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 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, 1]. 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

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)
```