Санкт-Петербургский государственный университет

ПОГОЖЕЛЬСКАЯ Влада Владимировна

Выпускная квалификационная работа

Реализация и экспериментальное исследование алгоритма поиска путей с контекстно-свободными ограничениями в графовой базе данных Neo4j

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2018 «Программная инженерия»*

Научный руководитель:
доцент кафедры информатики, к.ф.-м.н., Григорьев С.В.

Рецензент:
Программист ООО "Интеллиджей Лабс" Вербицкая Е.А.

Санкт-Петербург
2022

Saint Petersburg State University

Vlada Pogozhelskaya

Bachelor's Thesis

Implementation and experimental study of GLL algorithm with Neo4j graph database

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2018 «Software Engineering»*

Scientific supervisor:
C.Sc., docent Semyon Grigorev

Reviewer:
Software Engineer at IntelliJ Labs Co. Ltd Ekaterina Verbitskaia

Saint Petersburg
2022

Contents

Introduction

Graph-structured data models are widely used in many scientific application domains [17] such as social network analytics [3], biological knowledge graph management [16], and static code analysis [8]. One of the main advantages of this model over the relational data model is that obtaining information about the relationship between objects is very fast. The relationships between nodes are not calculated during query execution but stored in the model itself. One of the most common tasks associated with analyzing data represented by graphs is searching for paths. In graph databases queries are used for paths analysis. The natural way of specifying it is to impose restrictions on paths between vertices.

One can express such queries by defining formal grammar over an alphabet of edge labels. A path belongs to the language specified by the formal grammar if the language contains the word obtained by concatenating the edge labels of the given path [21]. Most often, with this approach, regular grammar is taken. For example, one of the most common graph databases Neo4j uses a declarative language called Cypher as a query language. It supports path constraints in terms of regular languages. It is noteworthy that Cypher supports regular constraints only partially, and at this moment its query language is quite limited and it does not provide sufficient expressive power in a variety of domains. One way to extend the expressiveness of queries is by using constraints in terms of context-free languages. Context-free path queries strictly extend the expressive power of regular path queries for sophisticated graph analytics and thus this is suitable a wider class of problems. One of these problems is the same-generation query. In bioinformatics, the aim of such queries is to determine paths between species in a genealogical database where the species are at the same level in the species-subspecies hierarchy. The query is expressible in terms of context-free languages, but there is no way to express it in terms of regular expressions [18].

Despite the fact that the problem of context-free paths querying is well studied and a lot of algorithms were proposed [2, 5, 12, 15], there are still a number of problems associated with its applicability in the analysis of real data. The most critical problems are the poor performance of existing algorithms on real-world data and bad integration with real-world graph databases and graph analysis systems [7]. These problems hinder the adoption of CFPQ.

The problem with the performance of CFPQ algorithms in real-world scenarios was pointed out by Jochem Kuijpers [7] as a result of an attempt to extend the Neo4j graph database with CFPQ. The number of the state of the art methods for CFPQ processing were selected [18, 11, 14, 6]. The authors implemented them using Neo4j as a graph storage and evaluated them. The results of comparison of measured performance showed that these solutions are not able to cope with large graphs as found in practice.

One of the implemented methods uses matrix representation to get the information about reachability in a given graph. This method was proposed by Rustam Azimov in [6] and is based on matrix operations. Since the performance problem was pointed out, it was shown that this algorithm demonstrates good performance enough. Moreover, the matrix-based CFPQ algorithm has become the base for the first full-stack support of CFPQ by extending the RedisGraph graph database.

Matrix representation is not the only one way to express CFPQ algorithms. Moreover, research shows that basic parsing algorithms that accept string and grammar can be straightforwardly generalized for graph input [9]. For example, there are efforts that describe how the CFPQ problem can be solved by using a generalized LL analyzer or a generalized LR analyzer. It is worth noting that such an approach allows getting information not only about reachability in the graph but also solves all paths problem [13]. However, such practical cases of getting information about reachability and paths are not studied enough.

The combination of these factors motivates further research and implementation of new solutions. On the other hand, there exists the recent development of the CFPQ problem [1] integrated with the Neo4j graph database[1]. This is the adaptation of the classic Generlized LL parsing al-

[1] Github repository of modificated GLL algorithm: https://github.com/YaccConstructor/iguana, last

gorithm for execution of context-free queries on graphs. It is important to note that the resulting algorithm supports the entire class of context-free languages. The modified GLL algorithm, just like the original one, returns information not only about reachability between vertices, but also information for reconstructing the paths themselves. A special data structure is used for this purpose called Shared Packed Parse Forest (SPPF). However, this data structure consumes a significant amount of resources and, as a result, leads to worse performance of the entire algorithm. In practice, the limitations on processor resources are quite significant, and the paths themselves are not always required: it would be enough to obtain only information about their existence. What is more, there is no investigation on providing parallel solutions based not on linear-algebra-oriented algorithms. In the multi- and many-core world and the big data era, it is important to provide a parallel solution for CFPQ.

Thus, the aim of this work is to provide an implementation of the GLL algorithm to graph handling for both the `all paths` and the `reachability` scenarios. Then, it is planned in this work to make an evaluation of the proposed solution to investigate whether significant performance improvements could be achieved.

accessed: 05/05/2022

1 Problem statement

The aim of the work is to improve the implementation of the GLL-based CFPQ algorithm and to measure the performance of provided implementation on real-world graphs. In order to achieve the aim, the following objectives have been set.

- To analyze and refactor the code of the current implementation of the GLL-based CFPQ algorithm for identifying and eliminating performance problems.

- To provide an ability to solve both, the `reachability CFPQ problem` and `all paths CFPQ problem`.

- To evaluate the resulting algorithm on real-world graphs and to compare it with the existing one.

2 Related work & background

This section includes basic notation and definitions in graph theory and formal language theory which are used in this work. Also, the further description of both the theoretical part of the GLL-based CFPQ algorithm and its implementation are provided.

2.1 Basic Definitions of Formal Languages

In this work, the context-free grammars are used as path constraints, thus context-free languages and grammars are defined in this subsection.

A *context-free grammar* is a tuple $G = \langle N, \Sigma, P, S \rangle$, where

- N is a finite set of nonterminals
- Σ is a finite set of terminals, $N \cap \Sigma = \varnothing$
- P is a finite set of productions of the form $A \to \alpha$, where $A \in N,\ \alpha \in (N \cup \Sigma)^*$
- $S \in N$.

□

We use the conventional notation $A \Rightarrow^* w$ to denote, that a word $w \in \Sigma^*$ can be derived from a non-terminal A using some sequence of production rules from P.

A *context-free language* is a language generated by a con-text-free grammar G:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

2.2 Basic Definitions of Graph Theory

In a simplified way, the Neo4j graph database uses a labeled directed graph as a data model. It can be defined as follows.

Labeled directed graph is a tuple $D = \langle V, E, T \rangle$, where

- V is a finite set of vertices. For simplicity, we assume that the vertices are natural numbers from 0 to $|V| - 1$.
- T is a set of labels on edges.
- $E \subseteq V \times T \times V$ is a set of edges.

□

Path π in the graph $D = \langle V, E, T\rangle$ is a finite sequence of edges $(e_0, e_1, ..., e_{n-1})$, where $\forall\ j,\ 0 \leq j \leq n - 1 : e_j = (v_j, t_j, v_{j+1}) \in E$.

We denote the set of all paths in the graph D as $\pi(D)$. □

2.3 Context-free Path Querying

Now, we can define context-free path querying problems. Let be:

- a context-free grammar $G = \langle N, \Sigma, P, S\rangle$;
- a directed graph $D = \langle V, E, T\rangle$, where V is the set of vertices of the graph, $E \subseteq V \times T \times V$ is the set of edges, $T \subseteq \Sigma$ is the set of labels on edges, where each label is a terminal symbol of the grammar G;
- a set of start vertices $V_S \subseteq V$ and final vertices $V_F \subseteq V$.

Consider a path in the graph D:

$$\pi = (e_0, e_1, \cdots, e_{n-1}),$$

where $e_k = (v_k, t_k, v_{k+1}),\ \forall\ k,\ 0 \leq k \leq n - 1\ e_k \in E$. To path in the graph the word $l(\pi) = t_0 t_1 \cdots t_{n_1}$ is associated — the concatenation of the labels on the edges of this path.

In the introduced notation, the following problems can be formulated.

- **The problem of a path querying in a graph with context-free constraints** consists in finding all paths in the graph such that $l(\pi) \in L(G)$ and $v_0 \in V_S,\ v_n \in V_F$.

- **The problem of reachability in a graph with context-free constraints** consists in finding a set of pairs of vertices for which there is a path with a beginning and an end at these vertices, such that the word composed of labels of the edges of the path belongs to the given language: $\{(v_i, v_j) \mid \exists\ l(\pi) \in L(G) \text{ and } v_0 \in V_S,\ v_n \in V_F\}$.

It should be noted that it is often necessary to identify complex dependencies in a graph data model. So, according to the context and application area, both variants of the above problems are of practical importance.

For each problem there are two variants of set of starting vertices: the set may consist of all vertices of a graph or may consist only a particular vertices of interest. The first variant is called all-pairs context-free path querying problem and the second is called a multiple-source (and a single-source as a partial case) context-free path querying problem.

2.4 Generalized LL Parsing Algorithm

One of the common parsing techniques is the LL(k) algorithm [10], that performs top-down analysis with a lookahead. It means that the decision about which production of the grammar should be applied is based on looking at the k following character from the current one. To choose the right production rule at this step algorithm supports a parsing table, where the information for parsing the current non-terminal is stored. However, it can be applied only to a subset of the context-free grammars class and does not support ambiguous context-free grammars or grammars with left recursion in derivation.

Top-down analysis algorithms are relatively easier to implement and debug, because it fully matches the structure of the grammar. For this reason, to extend the parsing power of above-mentioned technique there was proposed [19] the generalized LL (GLL) algorithm. Also GLL can handle ambiguous grammars. In case of LL(k) algorithm may arise the situation when it is impossible to determine which production should be applied in the current state of the parsing process [4]. To solve this issue the GLL algorithm maintains a queue of descriptors. Each descriptor is a structure that describes the current state of the analyzer. Thus, using a queue of descriptors allows one to consider all possible transitions during the operation of the parser.

The parsing table for the generalized GLL algorithm can store multiple alternatives for parsing the current non-terminal. In this case, descriptor duplication can occur. For efficient storage and reuse of many different descriptors, GLL uses a specific structure — Graph Structured Stack (GSS) [22].

To represent the result, GLL provides the Shared Packed Parse Forest (SPPF) structure [20], which contains all derivation trees for all paths satisfying the specified language.

2.5 GLL-based CFPQ Algorithm

As it was showed, classical GLL parsing technique can be used to solve context-free language constrained path problem. It means that such technique can be used to proceed graph input. Previously, the algorithm was generalized from linear input to graph processing, as was described in [9].

To do this, the following modifications were proposed.

- A query has became a triple: a set of initial vertices, a set of final vertices, and a grammar.
- An initial set of descriptors must include all the start vertices of the graph.
- At the step of transition to the next character, it is necessary to support all possible transition options that correspond to all outgoing edges of the vertex.
- If parsing is completed, it is necessary to check whether the final vertex in the parsing belongs to the set of final vertices of the graph.

The described principles of the generalized GLL algorithm are important for understanding the features of its implementation, which will be described below.

The implementation of the algorithm is based on the Iguana project which is written in Java. This library provides the modified GLL algorithm. The advantage of Iguana project is that it uses a more efficient GSS for GLL parsing. In addition, it does not affect the worst-case cubic run time and space complexities of GLL parsing.

Under this work, it is important to pay attention to the following changes that were made to the workflow of the GLL algorithm to unable graph processing.

- In order to support graph processing, the abstraction of an input data was changed. The new implementation of the *Input* interface has been added. Now it is represented as a graph adjacency list, a set of start and final vertices of the resulting paths.

- There can be multiple start vertices for a graph input, unlike a linear input. So, also the initialization of the descriptor queue was modified. In case of processing a descriptor with slot $(N \to \alpha.x\beta)$, where x is a terminal, the nextSymbols method was used. It took an index i in the input string and returns an index j such that the substring of the input string from i to $j - 1$ matches x. Thus, j is the index in the input string from which the parsing should continue by going to the slot $(N \to \alpha x.\beta)$. Considering the graph input there can be several similar positions. Therefore, the signature of this method has been changed. Now it returns a list of identifiers.

As far as the original GLL is aimed to handle arbitrary context-free grammars, this solution can handle arbitrary grammars too. It makes the solution less restrictive with regard to a query specification language, thus being more user-friendly.

3 Algorithm modification

This section will outline some important problems of current implementation and proposed solutions.

3.1 Research motivation

There were experiments of GLL-based CFPQ algorithm implementation carried out on real graphs. The experiment analysis showed that the algorithm demonstrated a good performance in most cases. It means that it is the right way in solving the problem of searching for paths in a graph with context-free constraints.

However, sometimes in the single source scenario an unexpected deterioration in the behavior of the resulting solution was revealed. Since the cause of performance problems remained unclear, as part of this work, it was decided to repeat experiments on a wider set of queries. These experiments of the extended GLL algorithm showed that queries for some start vertex sets take an abnormally long time to complete compared to other queries of the same type.

To sum up, despite of the efficiency showed by the algorithm, the behavioral problems call into a question its applicability in practice in the form in which it exists.

3.2 Problems and its solution

This subsection describes the changes made to the algorithm implementation to improve its performance and to eliminate the identified problems.

First of all, the solution was profiled with Java Flight Recorder (JFR). The results showed that the largest amount of CPU time is spent in the main class method `Neo4jGraphInput - nextSymbols`. This method is used to map the current input to a grammar terminal. It takes a vertex and returns a list of labels (symbols) on the outgoing edges. At the same time, in order to obtain these labels, it is necessary to reach out to the Neo4j database. The Native Java API provides a convenient way to do this: to get an iterator over the set of outgoing edges using the `getRelationships.iterator()` method. However, in the current implementation, almost all CPU time is spent on calculations within the database.

It should be noted that after the input matching with the terminal, it is possible that not all labels will be used in the further execution of the algorithm. Saving all the data leads to a huge overhead (up to a heap overflow) in case when degree of the vertex is very big, and most of the labels are discarded after matching. Thus, it is necessary to optimize the transfer of labels from the database to further processing. The following solution was proposed.

A common and reasonable solution to this problem is using the Stream API. Stream in Java is a sequence of elements supporting sequential and parallel aggregate operations. In other words, it is not a data store, but an interface to the source, from where elements are taken only when they are needed. One scenario for using threads as the return type of a method is as follows. In the called method, one must specify the processing of objects using one or more intermediate operations, and in the calling method, the final operation. The `nextSymbols` method has been rewritten to accommodate this scenario for all input types. Now the data source is the Neo4j database, the intermediate operation is filtering edges by labels, and the final operation is getting labels from the stream returned by the `nextSymbols` method. Thus, in this method, stream processing of data was provided.

The modified algorithm was tested and re-profiled. The profiling results confirmed that the changes made to the `nextSymbols` method were correct and, thus, the problem of the algorithm's slowing was eliminated. Moreover, it will be demonstrated in Experimental Study section that the optimizations generally affected the speed of the algorithm in a positive direction.

3.3 Functionality extension

This section describes the changes that have been made to the algorithm for solving the reachability CFPQ problem.

Extended GLL algorithm returns SPPF. It contains all derivations trees for all paths which satisfy to language constraints. So there is provided a natural solution for the all paths CFPQ problem. But in practice, the restrictions on processor resources are very significant, while restoring the paths themselves is not always required. Often it is enough to obtain only information about its existence. Accordingly, there was added a switch that allows one to not to create SPPF in case when only reachability information. SPPF is needed only for paths reconstruction, so if one wants to get only reachable pairs, SPPF construction can be omitted, which leads to performance improvements and memory consumption decreasing.

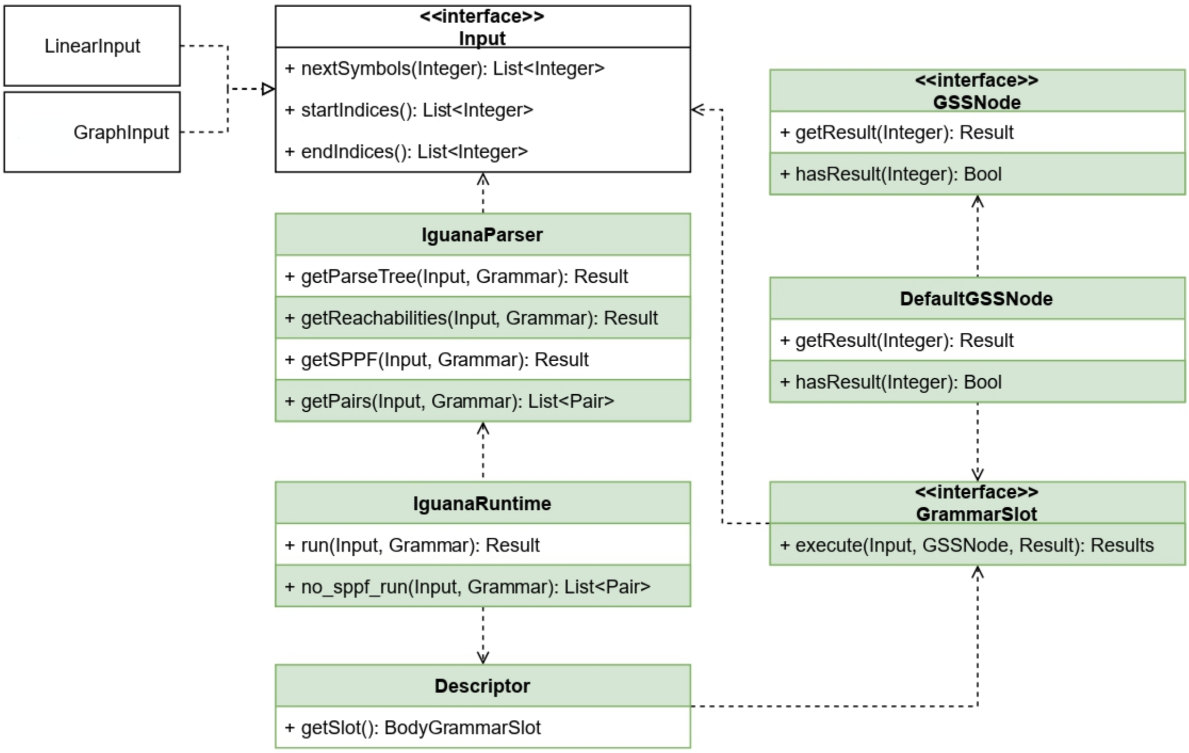

LinearInput
GraphInput
<<interface>>
Input
+ nextSymbols(Integer): List<Integer>
+ startIndices(): List<Integer>
+ endIndices(): List<Integer>
<<interface>>
GSSNode
+ getResult(Integer): Result
+ hasResult(Integer): Bool
IguanaParser
+ getParseTree(Input, Grammar): Result
+ getReachabilities(Input, Grammar): Result
+ getSPPF(Input, Grammar): Result
+ getPairs(Input, Grammar): List<Pair>
DefaultGSSNode
+ getResult(Integer): Result
+ hasResult(Integer): Bool
IguanaRuntime
+ run(Input, Grammar): Result
+ no_sppf_run(Input, Grammar): List<Pair>
<<interface>>
GrammarSlot
+ execute(Input, GSSNode, Result): Results
Descriptor
+ getSlot(): BodyGrammarSlot

Figure 1: Architecture of the solution

The main parts of the solution are presented in figure 1. To handle the scenario when SPPF construction is omitted almost all architecture elements were changed. The green color in this diagram marks the classes and methods that have been changed at this stage of the work. `Iguana Runtime` takes an `Input` and a context-free `Grammar` to produce the `Result`. The `Result` is a key entity which should or should not contain SPPF information. `Input` abstracts a data structure with the ability to get the next symbols for the given position. One example of `Input` is `Graph`, in which the position is a vertex, and the next symbols are the labels of its outgoing edges. There was provided to use different forms of graph representation. Communication with the database is done using the Neo4j Native Java API. The way to create a database was changed. Now an embedded database is used. It means that it runs inside of the application and it is not used as an external service as it was earlier. Worth noting, the architecture is extensible, and one can provide their own implementation of `Graph` to enable context-free path querying for a new graph storage.

4 Experimental study

To assess the applicability of the proposed solution the number of scenarios with real-world graphs and queries were evaluated. In this section, the evaluation of the resulting algorithm is described.

4.1 Hardware

All experiments were run on server with following characteristics.

- Operating system Ubuntu 18.04.
- Processor Intel Core i7-6700 CPU, 3.40GHz, 4 threads (hyper-threading is turn off).
- DDR4 64Gb RAM.
- Java HotSpot(TM) 64-Bit server virtual machine(build 15.0.2+7-27, mixed mode, sharing). JVM was configured to use 55Gb of heap memory.
- Neo4j version 4.0.3. Almost all configurations of Neo4j are default.

4.2 Datasets

The experimental study was carried out on the graphs from CFPQ_Data[2] data set. There was selected a number of graphs related to RDF analysis, as well as a number of graphs extracted from the Linux sources which related to static code analysis problems. A detailed description of the graphs, namely the number of vertices and edges and the number of edges labeled by tags used in queries, is in table 1 and table 2.

All queries used in the evaluation are variants of *same-generation query*. For the *RDF* graphs were used the same queries as in the other works for CFPQ algorithms evaluation: G_1 (1), G_2 (2), and Geo (3). The queries are

[2] CFPQ_Data is a public set of graphs and grammars for CFPQ algorithms evaluation. GitHub repository: https://github.com/JetBrains-Research/CFPQ_Data. Accessed: 05/05/2022.

Graph name	$\|V\|$	$\|E\|$	#subClassOf	#type	#broaderTransitive
Core	1 323	2 752	178	0.	0
Pathways	6 238	12 363	3 117	3 118	0
Go hierarchy	45 007	490 109	490 109	0	0
Enzyme	48 815	86 543	8 163	14 989	8 156
Eclass	239 111	360 248	90 962	72 517	0
Geospecies	450 609	2 201 532	0	89 065	20 867
Go	582 929	1 437 437	94 514	226 481	0
Taxonomy	5 728 398	14 922 125	2 112 637	2 508 635	0

Table 1: Graphs for evaluation: number of vertices and edges, and number of edges with specific label

Graph name	$\|V\|$	$\|E\|$	#a	#d
Apache	1 721 418	1 510 411	362 799	1 147 612
Block	3 423 234	2 951 393	669 238	2 282 155
Fs	4 177 416	3 609 373	824 430	2 784 943
Ipc	3 401 022	2 931 498	664 151	2 267 347
Lib	3 401 355	2 931 880	664 311	2 267 569
Mm	2 538 243	2 191 079	498 918	1 692 161
Net	4 039 470	3 500 141	807 162	2 692 979
Postgre	5 203 419	4 678 543	1 209 597	3 468 946
Security	3 479 982	3 003 326	683 339	2 319 987
Sound	3 528 861	3 049 732	697 159	2 352 573
Init	2 446 224	2 112 809	481 994	1 630 815
Arch	3 448 422	2 970 242	6 712 95	2 298 947
Crypto	3 464 970	2 988 387	678 408	2 309 979
Drivers	4 273 803	3 707 769	858 568	2 849 201
Kernel	11 254 434	9 484 213	1 981 258	7 502 955

Table 2: Graphs for evaluation: number of vertices and edges, and number of edges with specific label

expressed as context-free grammars where S is a nonterminal, *subClassOf, type, broaderTransitive,* $\overline{subClassOf}$, $\overline{type}$, $\overline{broaderTransitive}$ are terminals or the labels of edges. The inverse of an x relation and the respective edge is denoted as $\overline{x}$.

$$\begin{aligned} S \rightarrow & \overline{subClassOf}\ \ S\ \ subClassOf \mid \overline{type}\ \ S\ \ type \\ & \mid \overline{subClassOf}\ \ subClassOf \mid \overline{type}\ type \end{aligned} \tag{1}$$

$$S \to \overline{subClassOf} \;\; S \;\; subClassOf \,|\, subClassOf \tag{2}$$

$$\begin{aligned} S \to & broaderTransitive \;\; S \;\; \overline{broaderTransitive} \\ & |\; broaderTransitive \;\; \overline{broaderTransitive} \end{aligned} \tag{3}$$

For *Program analysis* graphs was used a *PointsTo* query (4) which describes a points-to analysis [23]. Here M and V are nonterminals, M is start nonterminal, $\{a, d, \overline{a}, \overline{d}\}$ are terminals.

$$\begin{aligned} M &\to \overline{d} \; V \; d \\ V &\to (M? \; \overline{a})^* \; M? \; (a \; M?)^* \end{aligned} \tag{4}$$

4.3 The statement of the experiments

To show the usefulness of the proposed implementation the results of measurements were considered in comparison with the original implementation of the algorithm. The following chunks were taken as start sets:

$$\forall r \in R = \{1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 50,\ 100,\ 500,\ 1000,\ 5000,\ 10000\}$$

$$V(r) = V_1 \sqcup V_2 \sqcup \cdots \sqcup V_m,\ \forall 1 \le i < m : |V_i| = r,\ |V_m| \le r.$$

The entire set of graph vertices was considered as the final vertices in all experiments.

Then, the proposed solution was evaluated on the following scenarios: `all-pairs reachability`, `single source reachability`, and `single source all paths`. The `all-pairs all paths` scenario was omitted because its impracticality: the detailed analysis is often required only for paths within a specific subgraph, not the entire graph.

For `all-pairs reachability` scenario there were 5 iterations for each query. Two out of five iterations were held to warm up Java machine. Time results of others were used in the evaluation analysis. For `single source reachability` and `single source all-paths` scenarios for each graph there were created randomized sets of vertices. These sets were used as the start sets for the corresponding graphs in all appropriate experiments.

Only algorithm run time was measured. The time spent creating the database, as well as the grammar loading time, was not taken into account. To check the correctness of the solution and to force the result stream, the number of reachable pairs for each query was computed.

To demonstrate the applicability of proposed solution, results for graphs related to static code analysis were compared to results of Azimov's CFPQ algorithm based on matrix operations. The implementation from CFPQ_PyAlgo repository[3] was taken as the implementation of the matrix CFPQ algorithm. This library contains the implementation for both scenarios, all pairs and single source. To perform matrix operations pygraphblas[4] was used. Pygraphblas is a python wrapper over the SuiteSparse library, which contains a set of sparse matrix algorithms to provide graph processing in terms of linear algebra.

4.4 Evaluation Results

First of all, it is necessary to assure that the provided modifications are useful in context of algorithm applicability. To show the difference between the proposed implementation and the original implementation of the algorithm the Enzyme graph was used.

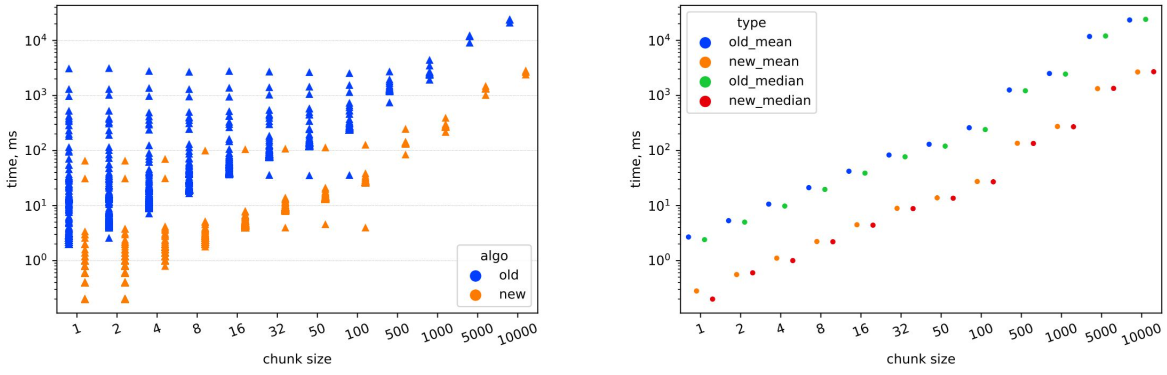
Below in fig.2-4 comparative results of time measurements are shown.

Similarly, the median and mean query time for each size of the set of start vertices are additionally allocated. These results show that the improved version of the algorithm not only spends an order of magnitude less time resources, but also is more stable on average on real-world graphs. This is especially noticeable in Fig.3 which demonstrates the median and average time. For the modified algorithm these two indicators are almost equal.

The resulting solution was evaluated on the same hardware. Since, after the optimizations, the amount of consumed processor resources significantly decreased for both algorithms, it became possible to include the whole RDF graphs dataset for experimental study. The results of the all-pairs reacha-

[3] CFPQ_PyAlgo repository: https://github.com/JetBrains-Research/CFPQ_PyAlgo, accessed: 05/05/2022

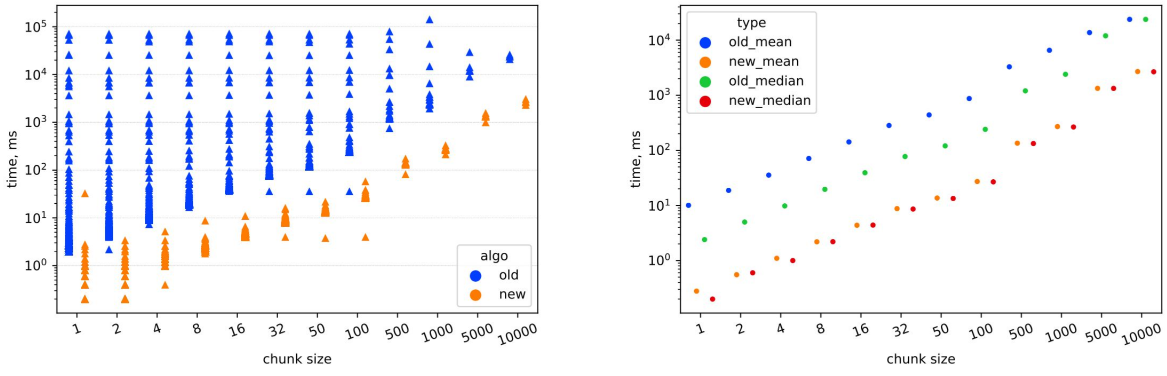
[4] pygraphblas repository: https://github.com/Graphegon/pygraphblas, accessed: 05/05/2022



(a) Query time

(b) Median and mean query time

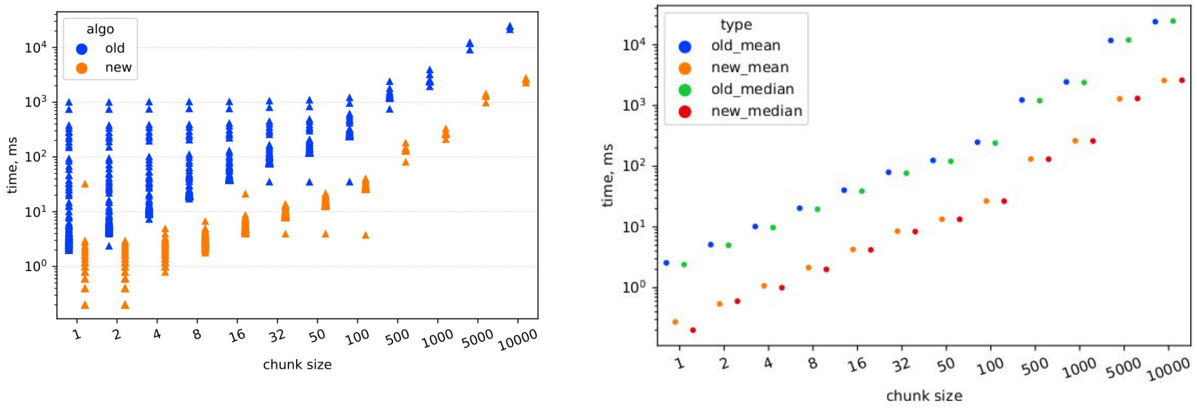
Figure 2: Grammar G_1 and Enzyme



(a) Query time

(b) Median and mean time

Figure 3: Grammar G_2 and Enzyme



(a) Query time

(b) Median and mean

Figure 4: Grammar *Geo* and Enzyme

bility queries evaluation are presented in tables 3 – 5.

The results show that query evaluation time depends not only on a graph

Graph name	G_1		G_2	
	time (sec)	#answer	time (sec)	#answer
Core	0,02	204	0,01	214
Pathways	0,07	884	0,04	3117
Go hierarchy	3,68	588 976	5,4	738 937
Enzyme	0,22	396	0,17	8163
Eclass	1,5	90 994	0,98	96 163
Geospecies	2,87	85	2,65	0
Go	5,56	640 316	4,2	659 501
Taxonomy	45,47	151 706	36,07	2 112 637

Table 3: Single thread all-pairs reachability performance results for RDFs: time in seconds, #**answer** is a number of reachable pairs

Graph name	*Geo*	
	time (sec)	#answer
Enzyme	5,7	14 267 542
Geospecies	145,8	226 669 749

Table 4: Single thread all-pairs reachability performance results for RDFs: time in seconds, #**answer** is a number of reachable pairs

size or its sparsity, but also on an inner structure of the graph. For example, the relatively small graph Go hierarchy fully consists of edges used in queries G_1 and G_2, thus evaluation time for these queries is significantly bigger than for some bigger but more sparse graphs, for example, for Eclass graph. Note that the size of the answer is not a good metric, because, for example, answers for Geo query, and Enzyme and Geospecies graphs, are calculated faster than the answers for Go hierarchy. The creation of relevant metrics for CFPQ queries evaluation time prediction is a challenging problem by itself and should be tackled in the future.

The important results are demonstrated in Table 4. Previous similar solution required more then 6900 seconds to evaluate the the Geo query for the Geospecies graph [7]. Thus it shows that the performance of CFPQ for Neo4j was significantly improved.

The results for graphs related to static code analysis on all-pairs scenario are presented in Table 5. The sign '–' in cells means that the respective query and graph require a considerable amount of memory during algorithm

Graph name	*PointsTo*		
	time (sec)	matrix time (sec)	#answer
Apache	–	536,7	92 806 768
Block	113,01	123,88	53 514 095
Fs	167,73	105,72	9 646 475
Ipc	109,43	79,52	5 249 389
Lib	111,09	121,79	5 276 303
Mm	77,92	84,15	3 990 305
Net	160,64	206,29	8 833 403
Postgre	–	969,88	90 661 446
Security	115,75	181,7	5 593 387
Sound	120,14	133,64	6 085 269
Init	87,25	45,84	3 783 769
Arch	130,77	119,92	5 339 563
Crypto	128,8	122,09	5 428 237
Drivers	371,18	279,39	18 825 025
Kernel	614,05	378,05	16 747 731

Table 5: Single thread all-pairs reachability performance results: time in seconds, #**answer** is a number of reachable pairs

execution that leads to unpredictable time to get the result. Although other queries related to code analysis can be evaluated in reasonable time even for relatively big graphs. Moreover, in some cases GLL-based CFPQ algorithm demonstrates comparable and even better time results than matrix based CFPQ algorithm. For example, Security graph is processed approximately one and a half times faster with GLL-based CFPQ algorithm than with matrix based CFPQ algorithm. Thus, CFPQ can be used to improve Neo4j-based code analysis systems.

The another important scenario is the case when the start set is a single vertex. The results of the `single source reachablity` and `all paths reachability` queries related to RDF analysis are presented in figures 5–7.

The go hierarchy graph is omitted because it fully connected with *subClassOf* relation and GLL-based CFPQ algorithm does not handle well the SPPF construction for such specific case because of memory limits.

Firstly, it should be noted that the single-source queries are reasonably fast in both the `reachability` and the `all paths` cases: median time is

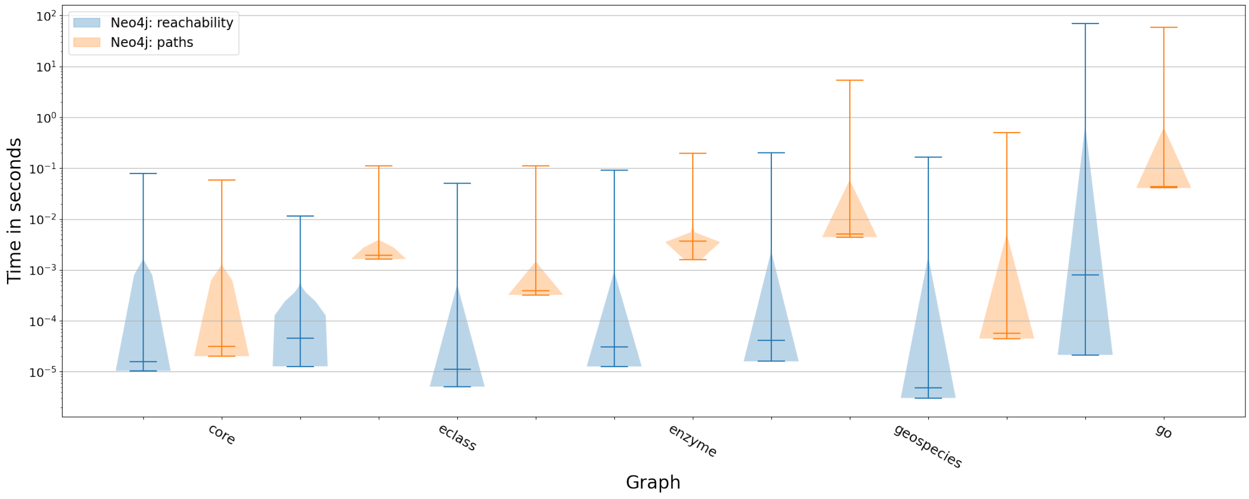


Figure 5: Single source CFPQ results for queries related to RDF analysis and G_1

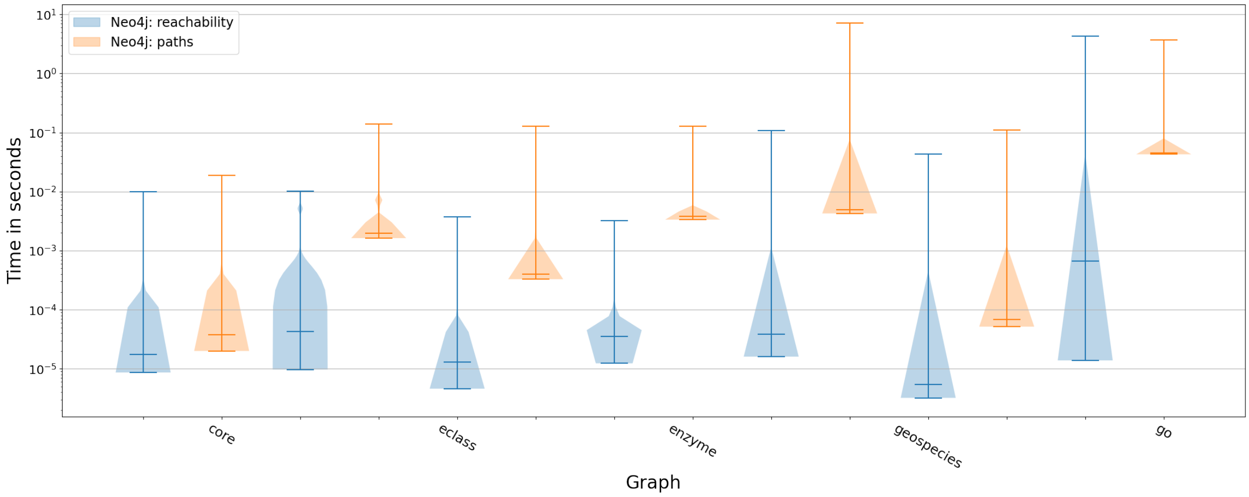


Figure 6: Single source CFPQ results for queries related to RDF analysis and G_2

less than 10^{-4} seconds for reachability queries, and is less than 10^{-1} seconds for `all paths` queries. Even for queries G_1 and G_2, which return relatively small answers, time required for the all paths queries is bigger than for reachability queries. Also there are some ”heavy” cases when both scenarios require relatively a big amount of time to get the result. Moreover, for such cases the time for `reachability` and `all-paths` is comparable in contradiction to others. It should be due to the fact that often in single source scenario the memory allocation for SPPF construction require relatively significant amount of time. Thus, there is a noticeable difference between

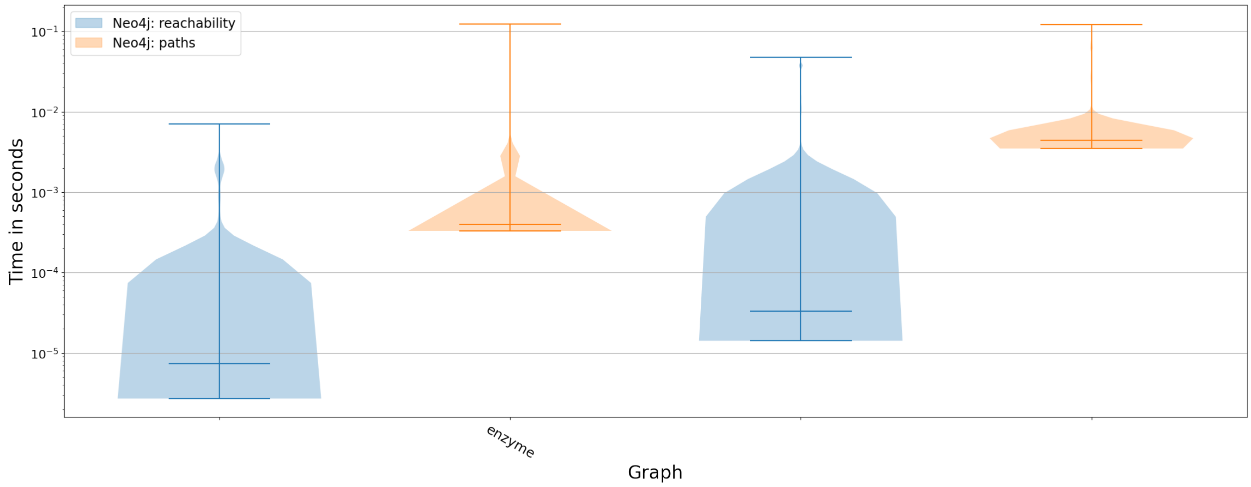


Figure 7: Single source CFPQ results for queries related to RDF analysis and Geo

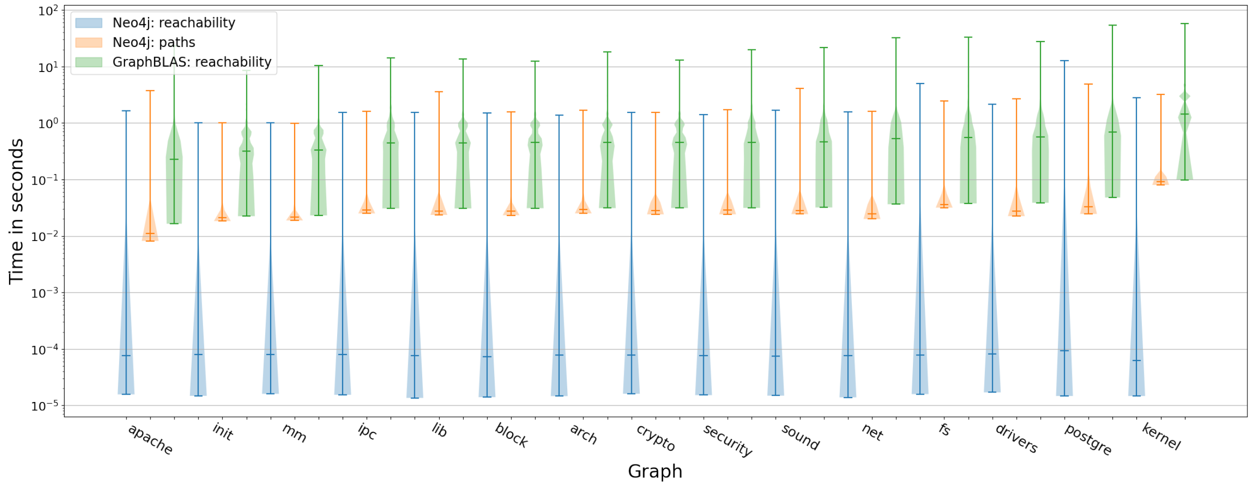


Figure 8: Single source CFPQ results for queries related to static code analysis

`reachability` and `all-paths` scenarios.

The results of the `single source reachablity` and `all paths` queries related to static code analysis are presented in figure 8. Here is also the comparison with matrix based CFPQ algorithm. This is perhaps the most significant result, which shows that as a result of the optimizations, both the for paths and the calculation of the fact of reachability are orders of magnitude faster than the execution of the same queries using the matrix algorithm.

It is noticeable that the execution results are similar to results of the previous experiment. The execution time for all graphs except Postgre graph is less than 10^2 seconds. Moreover, median time is less than 10^{-4} seconds for all processing sets.

To conclude, the analysis of the obtained results showed that the modified GLL algorithm can be effectively used on real-world graphs to solve the both problems: reachability and all-paths context-free path querying problems. The obtained results make relevant further research aimed at both improving this algorithm and implementation, and its full integration into the Neo4j graph database.

Conclusion

The following results have been obtained.

- The performance problems in the implementation of the GLL-based CFPQ algorithm were eliminated.
- The implementation of GLL-based CFPQ algorithm was extended with ability to solve both, `the reachability` and `the all paths CFPQ problem`.
- The resulting algorithm implementation was evaluated on two sets of real-world graphs: a number of graphs related to RDF analysis and a number of graph related to static code analysis problem for both the `all pairs` and the `multiple sources` scenarios. The evaluation shows that the proposed algorithm is more than 45 times faster than the previous solution for Neo4j.

References

[1] Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // Compiler Construction / Ed. by Björn Franke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. — P. 89–108.

[2] Azimov Rustam, Grigorev Semyon. Context-Free Path Querying by Matrix Multiplication. — GRADES-NDA '18. — New York, NY, USA : Association for Computing Machinery, 2018. — 10 p. — URL: https://doi.org/10.1145/3210259.3210264.

[3] Barceló Baeza Pablo. Querying Graph Databases // Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. — PODS '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 175–188. — URL: https://doi.org/10.1145/2463664.2465216.

[4] Beatty John C. Iteration Theorems for LL(k) Languages (Extended Abstract) // Proceedings of the Ninth Annual ACM Symposium on Theory of Computing. — STOC '77. — New York, NY, USA : Association for Computing Machinery, 1977. — P. 122–131. — URL: https://doi.org/10.1145/800105.803402.

[5] Zhang Xiaowang, Feng Zhiyong, Wang Xin et al. Context-Free Path Queries on RDF Graphs. — 2016. — 1506.00743.

[6] Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication / Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, Semyon Grigorev // Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA'20. — New York, NY, USA : Association for Computing Machinery, 2020. — 12 p. — URL: https://doi.org/10.1145/3398682.3399163.

[7] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker // Proceedings of the 31st International Conference on Scientific and Statistical Database Management. — SSDBM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 121–132. — URL: https://doi.org/10.1145/3335783.3335791.

[8] Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis / Qirun Zhang, Michael R. Lyu, Hao Yuan, Zhendong Su // SIGPLAN Not. — 2013. — Vol. 48, no. 6. — P. 435–446. — URL: https://doi.org/10.1145/2499370.2462159.

[9] Grigorev Semyon, Ragozina Anastasiya. Context-Free Path Querying with Structural Representation of Result // Proceedings of the 13th Central amp; Eastern European Software Engineering Conference in Russia. — CEE-SECR '17. — New York, NY, USA : Association for Computing Machinery, 2017. — 7 p. — URL: https://doi.org/10.1145/3166094.3166104.

[10] Grune Dick, Jacobs Ceriel J. H. Parsing Techniques (Monographs in Computer Science). — Berlin, Heidelberg : Springer-Verlag, 2006. — ISBN: 038720248X.

[11] Hellings Jelle. Conjunctive context-free path queries // Proceedings of ICDT'14. — 2014. — P. 119–130.

[12] Hellings Jelle. Querying for Paths in Graphs using Context-Free Path Queries. — 2016. — 1502.02242.

[13] Hellings Jelle. Explaining Results of Path Queries on Graphs // Software Foundations for Data Interoperability and Large Scale Graph Data Analytics / Ed. by Lu Qin, Wenjie Zhang, Ying Zhang et al. — Cham : Springer International Publishing, 2020. — P. 84–98.

[14] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. Efficient Evaluation of Context-Free Path Queries for Graph Databases // Pro-

ceedings of the 33rd Annual ACM Symposium on Applied Computing. — SAC '18. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 1230–1237. — URL: https://doi.org/10.1145/3167132.3167265.

[15] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. An Algorithm for Context-Free Path Queries over Graph Databases. — 2020. — 2004.03477.

[16] Quantifying variances in comparative RNA secondary structure prediction / James Anderson, Adám Novák, Zsuzsanna Sükösd et al. // BMC bioinformatics. — 2013. — 05. — Vol. 14. — P. 149.

[17] Robinson I., Webber J., Eifrem E. Graph Databases: New Opportunities for Connected Data. — O'Reilly Media, 2015. — ISBN: 9781491930847.

[18] Santos Fred C., Costa Umberto S., Musicante Martin A. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases // Web Engineering / Ed. by Tommi Mikkonen, Ralf Klamma, Juan Hernández. — Cham : Springer International Publishing, 2018. — P. 225–233.

[19] Scott Elizabeth, Johnstone Adrian. GLL Parsing // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189. — Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009). URL: https://www.sciencedirect.com/science/article/pii/S1571066110001209.

[20] Scott Elizabeth, Johnstone Adrian. GLL parse-tree generation // Science of Computer Programming. — 2013. — Vol. 78, no. 10. — P. 1828–1844. — Special section on Language Descriptions Tools and Applications (LDTA'08 '09) Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011). URL: https://www.sciencedirect.com/science/article/pii/S0167642312000627.

[21] Shemetova Ekaterina, Grigorev Semyon. Path querying on acyclic graphs using Boolean grammars // Proceedings of the Institute for System Programming of RAS. — 2019. — 10. — Vol. 31. — P. 211–226.

[22] Tomita Masaru. An Efficient Context-Free Parsing Algorithm for Natural Languages // Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.

[23] Zheng Xin, Rugina Radu. Demand-driven Alias Analysis for C // Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '08. — New York, NY, USA : ACM, 2008. — P. 197–208. — URL: http://doi.acm.org/10.1145/1328438.1328464.