One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries

Ekaterina Shemetova
katyacyfra@gmail.com
Saint Petersburg Academic University
St. Petersburg, Russia

Rustam Azimov
rustam.azimov19021995@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Egor Orachyov
egororachyov@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Ilya Epelbaum
iliyepelbaun@gmail.com
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Kronecker product based algorithm for context-free path querying (CFPQ) was recently proposed by Egor Orachev et. al. We reduce this algorithm to operation over Boolean matrices and extend it with a mechanism to extract all paths of interest. Also, we prove $O(n^3/\log n)$ time complexity of the proposed algorithm, where n is a number of vertices of the input graph. Thus we provide an alternative slightly subcubic algorithm for CFPQ which is based on linear algebra and on a classical graph-theoretic problem (incremental transitive closure), rather than the algorithm proposed by Swarat Chaudhuri. Our evaluation shows that the proposed solution is applicable for both RPQ and CFPQ over real-world graphs.

CCS CONCEPTS

• **Information systems** → **Graph-based database models**; **Query languages for non-relational engines**; • **Theory of computation** → **Grammars and context-free languages**; **Regular languages**; • **Mathematics of computing** → **Paths and connectivity problems**; *Graph algorithms.*

SIGMOD'21, ,

ACM ISBN XXX-X-XXXXX-XXX-X…$15.00

ACM Reference Format:
Ekaterina Shemetova, Rustam Azimov, Egor Orachyov, Ilya Epelbaum, and Semyon Grigorev. 2021. One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries. In . ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Language-constrained path querying [**?**] is a technique for graph navigation querying. This technique allows one to use formal languages as constraints on paths in edge-labeled graphs: path satisfies constraints if labels along it form a word from the specified language.

The utilization of regular languages as constraints, or *Regular Path Querying* (RPQ), is most well-studied and widespread. Different aspects of RPQs are actively studied in graph databases [**? ? ?**], while regular constraints are supported in such popular query languages as PGQL [**?**] and SPARQL[1] [**?**] (property paths). Nevertheless, there is certainly room for improvement of RPQ efficiency, and new solutions are being created [**? ?**].

At the same time, using more powerful languages, namely context-free languages, as constraints has gained popularity in the last few years. *Context-Free Path Querying* problem (CFPQ) was introduced by Mihalis Yannakakis in 1990 in [**?**]. Many algorithms were proposed since that time, but recently, Jochem Kuijpers et al. showed in [**?**] that state-of-the-art CFPQ algorithms are not performant enough for practical use. This motivates us to develop new algorithms for CFPQ.

[1]Specification of regular constraints in SPARQL property paths: https://www.w3.org/TR/sparql11-property-paths/. Access date: 07.07.2020.

One promising way to achieve high-performance solutions for graph analysis problems is to reduce them to linear algebra operations. This way, GraphBLAS [**?**] API, the description of basic linear algebra primitives, was proposed. Solutions that use libraries that implement this API, such as SuiteSparce [**?**] and CombBLAS [**?**], show that reduction to linear algebra is a way to utilize high-performance parallel and distributed computations for graph analysis.

Rustam Azimov shows in [**?**] how to reduce CFPQ to matrix multiplication. Later, it was shown in [**?**] and [**?**] that utilization of appropriate libraries for linear algebra for Azimov's algorithm implementation makes a practical solution for CFPQ. However Azimov's algorithm requires transforming the input grammar to Chomsky Normal Form, which leads to the grammar size increase, and hence worsens performance, especially for regular queries and complex context-free queries.

To solve these problems, an algorithm based on automata intersection was proposed [**?**]. This algorithm is based on linear algebra and does not require the transformation of the input grammar. We improve the algorithm in this work. We reduce the above mentioned solution to operations over Boolean matrices, thus simplifying its description and implementation. Also, we show that this algorithm is performant enough for regular queries, so it is a good candidate for integration with real-world query languages: one algorithm can be used to evaluate both regular and context-free queries.

Moreover, we show that this algorithm opens the way to tackle a long-standing problem about the existence of truly-subcubic $O(n^{3-\epsilon})$ CFPQ algorithm [**? ?**]. Currently, the best result is an $O(n^3/\log n)$ algorithm of Swarat Chaudhuri [**?**]. Also, there exist truly subcubic solutions which use fast matrix multiplication for some fixed subclasses of context-free languages [**?**]. Unfortunately, this solutions cannot be generalized to arbitrary CFPQs. In this work, we identify incremental transitive closure as a bottleneck on the way to achieve subcubic time complexity for CFPQ.

To sum up, we make the following contributions.

(1) We rethink and improve the CFPQ algorithm based on tensor-product proposed by Orachev et al. [**?**]. We reduce this algorithm to operations over Boolean matrices. As a result, all-path query semantics is handled, as opposed to the previous matrix-based solution which handles only the single-path semantics. Also, both regular and context-free grammars can be used as queries. We prove the correctness and time complexity for the proposed algorithm.
(2) We demonstrate the interconnection between CFPQ and incremental transitive closure. We show that incremental transitive closure is a bottleneck on the way to achieve faster CFPQ algorithm for general case of arbitrary graphs as well as for special families of graphs, such as planar graphs.
(3) We implement the described algorithm and evaluate it on real-world data. RPQ, CFPQ. Results show that !!!

2 PRELIMINARIES

In this section we introduce basic notation and definitions from graph theory and formal language theory.

2.1 Language-Constrained Path Querying Problem

We use a directed edge-labeled graph as a data model. To introduce the *Language-Constraint Path Querying Problem* [**?**] over directed edge-labeled graphs we first give both language and grammar definitions.

Definition 2.1. An *edge-labeled directed graph* $\mathcal{G}$ is a triple $\langle V, E, L\rangle$, where $V = \{0, \ldots, |V|-1\}$ is a finite set of vertices, $E \subseteq V \times L \times V$ is a finite set of edges and L is a finite set of edge labels.

The example of a graph which we use in the further examples is presented in Figure 1.

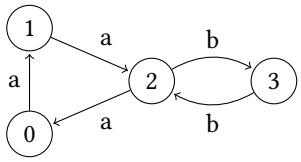


Figure 1: The example of input graph $\mathcal{G}$

Definition 2.2. An *adjacency matrix* for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L\rangle$ is a matrix M, which has size $|V| \times |V|$ and $M[i, j] = \{l \mid e = (i, l, j) \in E\}$.

Adjacency matrix M_2 of the graph $\mathcal{G}$ is

$$M_2 = \begin{pmatrix} . & \{a\} & . & . \\ . & . & \{a\} & . \\ \{a\} & . & . & \{b\} \\ . & . & \{b\} & . \end{pmatrix}.$$

Definition 2.3. The *Boolean matrices decomposition*, or *Boolean adjacency matrix*, for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L\rangle$ with adjacency matrix M is a set of matrices $\mathcal{M} = \{M^l \mid l \in L, M^l[i, j] = 1 \iff l \in M[i, j]\}$.

We use the decomposition of the adjacency matrix into a set of Boolean matrices. As an example, matrix M_2 can be

represented as a set of two Boolean matrices M_2^a and M_2^b:

$$M_2^a = \begin{pmatrix} . & 1 & . & . \\ . & . & 1 & . \\ 1 & . & . & . \\ . & . & . & . \end{pmatrix}, M_2^b = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & 1 \\ . & . & 1 & . \end{pmatrix}.$$

This way we reduce operations necessary for our algorithm from operations over custom semiring (over edge labels) to operations over a Boolean semiring with an *addition* $+$ as $\vee$ and a *multiplication* $\cdot$ as $\wedge$ over Boolean values.

We also use notation $\mathcal{M}(\mathcal{G})$ and $\mathcal{G}(\mathcal{M})$ to describe the Boolean decomposition matrices for some graph and the graph formed by its adjacency Boolean matrices.

Definition 2.4. A *path* π in the graph $\mathcal{G} = \langle V, E, L \rangle$ is a sequence $e_0, e_1, \ldots, e_{n-1}$, where $e_i = (v_i, l_i, u_i) \in E$ and for any e_i, e_{i+1}: $u_i = v_{i+1}$. We denote a path from v to u as $v \pi u$.

Definition 2.5. A *word formed by a path*

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

is a concatenation of labels along the path: $\omega(\pi) = l_0 l_1 \ldots l_{n-1}$.

Definition 2.6. A *language* $\mathcal{L}$ over a finite alphabet Σ is a subset of all possible sequences formed by symbols from the alphabet: $\mathcal{L}_\Sigma = \{\omega \mid \omega \in \Sigma^*\}$.

Now we are ready to introduce CFPQ problem for the given graph $\mathcal{G} = \langle V, E, L \rangle$ and the given language $\mathcal{L}$ with reachability and all-path semantics.

Definition 2.7. To evaluate context-free path query with reachability semantics is to construct a set of pairs of vertices (v_i, v_j) such that there exists a path $v_i \pi v_j$ in $\mathcal{G}$ which forms the word from the given language:

$$R = \{(v_i, v_j) \mid \exists \pi : v_i \pi v_j, \omega(\pi) \in \mathcal{L}\}$$

Definition 2.8. To evaluate context-free path query with all-path semantics is to construct a set of paths π in $\mathcal{G}$ which form the word from the given language:

$$\Pi = \{\pi \mid \omega(\pi) \in \mathcal{L}\}$$

Note that Π can be infinite, thus in practice we should provide a way to enumerate such paths with reasonable complexity, instead of explicit construction of the Π.

2.2 Regular Path Queries and Finite State Machine

In *Regular Path Querying* (RPQ) the language $\mathcal{L}$ is regular. This case is widespread [**?**] and well-studied. The most common way to specify regular languages is by *regular expressions*. We use the following definition of regular expressions.

Definition 2.9. A *regular expression* over the alphabet Σ is a finite combination of patterns, which can be defined as follows: $\varnothing$ (empty language), ε (empty string), $a_i \in \Sigma$ are regular expressions, and if R_1 and R_2 are regular expressions, then $R_1 \mid R_2$ (alternation), $R_1 \cdot R_2$ (concatenation), R_1^* (Kleene star) are also regular expressions.

For example, one can use regular expression $R_1 = ab^*$ to search for paths in the graph $\mathcal{G}$ (Figure 1). The expected query result is a set of paths which start with an a-labeled edge and contain zero or more b-labeled edges after that.

In this work we use the notion of *Finite-State Machine* (FSM) or *Finite-State Automaton* (FSA) for RPQs.

Definition 2.10. A *deterministic finite-state machine without ε-transitions* T is a tuple $\langle \Sigma, Q, Q_s, Q_f, \delta \rangle$, where Σ is an input alphabet, Q is a finite set of states, $Q_s \subseteq Q$ is a set of start (or initial) states, $Q_f \subseteq Q$ is a set of final states and $\delta : Q \times \Sigma \to Q$ is a transition function.

It is well known, that every regular expression can be converted to deterministic FSM without ε-transitions [**?**]. We use FSM as a representation of RPQ. FSM can be naturally represented by a directed edge-labeled graph: $V = Q$, $L = \Sigma$, $E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$, where some vertices have special markers to specify the start and final states. An example of the graph representation of FSM T_1 for the regular expression R_1 is presented in Figure 2.

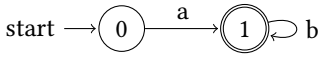


Figure 2: The example of graph representation of FSM for the regular expression ab^*

As a result, FSM also can be represented as a set of Boolean adjacency matrices $\mathcal{M}$ accompanied by the information about the start and final vertices. Such representation of T_1:

$$M^a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \; M^b = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Note, that an edge-labeled graph can be viewed as an FSM where edges represent transitions, and all vertices are both start and final at the same time. Thus RPQ evaluation is an intersection of two FSMs. The query result can also be represented as FSM, because regular languages are closed under intersection.

2.3 Context-Free Path Querying and Recursive State Machines

An even more general case than RPQ is a *Context-Free Path Querying Problem (CFPQ)*, where one can use context-free languages as constraints. These constraints are more expressive than the regular constraints. For example, a classic same-generation query can be expressed by a context-free language, but not a regular language.

Definition 2.11. A *context-free grammar* $G = \langle \Sigma, N, S, P \rangle$, where Σ is a finite set of terminals (or terminal alphabet), N is a finite set of nonterminals (or nonterminal alphabet), $S \in N$ is a start nonterminal, P is a finite set of productions (grammar rules) of form $N_i \to \alpha$ where $N_i \in N, \alpha \in (\Sigma \cup N)^*$.

Definition 2.12. The sequence $\omega_2 \in (\Sigma \cup N)^*$ is *derivable* from $\omega_1 \in (\Sigma \cup N)^*$ *in one derivation step*, or $\omega_1 \to \omega_2$, in the grammar $G = \langle \Sigma, N, S, P \rangle$ iff $\omega_1 = \alpha N_i \beta$, $\omega_2 = \alpha \gamma \beta$, and $N_i \to \gamma \in P$.

Definition 2.13. Context-free grammar $G = \langle \Sigma, N, S, P \rangle$ specifies a *context-free language*: $\mathcal{L}(G) = \{\omega \mid S \xrightarrow{*} \omega\}$, where $(\xrightarrow{*})$ denotes zero or more derivation steps $(\to)$.

For instance, a grammar $G_1 = \langle \{a, b\}, \{S\}, S, \{S \to a\, b;\ S \to a\ S\ b\} \rangle$ can be used to search for paths, which form words in the language $\mathcal{L}(G_1) = \{a^n b^n \mid n > 0\}$ in the graph $\mathcal{G}$ (fig. 1).

While a regular expression can be transformed to a FSM, a context-free grammar can be transformed to a *Recursive State Machine* (RSM) in the similar fashion. In our work we use the following definition of RSM based on [**?**].

Definition 2.14. A *recursive state machine* R over a finite alphabet Σ is defined as a tuple of elements $\langle M, m, \{C_i\}_{i \in M} \rangle$, where M is a finite set of labels of boxes, $m \in M$ is an initial box label, a $C_i = \langle \Sigma \cup M, Q_i, q_i^0, F_i, \delta_i \rangle$ is a *component state machine* or *box*, where:

- $\Sigma \cup M$ is a set of symbols, $\Sigma \cap M = \varnothing$
- Q_i is a finite set of states, where $Q_i \cap Q_j = \varnothing, \forall i \neq j$
- q_i^0 is an initial state for C_i
- $F_i \subseteq Q_i$ is a set of the final states for C_i
- $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$ is a transition function

RSM behaves as a set of finite state machines (or FSM). Each FSM is called a *box* or a *component state machine*. A box works similarly to the classical FSM, but it also handles additional *recursive calls* and employs an implicit *call stack* to *call* one component from another and then return execution flow back.

The execution of an RSM could be defined as a sequence of the configuration transitions, which are done while reading the input symbols. The pair (q_i, S), where q_i is a current state for box C_i and S is a stack of *return states*, describes an *execution configuration*.

The RSM execution starts from the configuration $(q_m^0, \langle\rangle)$. The following list of rules defines the machine transition from configuration (q_i, S) to (q', S') on some input symbol a:

- $(q_i^k, S) \rightsquigarrow (\delta_i(q_i^k, a), S)$
- $(q_i^k, S) \rightsquigarrow (q_j^0, \delta_i(q_i^k, j) \circ S)$
- $(q_j^k, q_i^t \circ S) \rightsquigarrow (q_i^t, S)$, where $q_j^k \in F_j$

An input word $a_1 \ldots a_n$ is accepted, if machine reaches configuration $(q, \langle\rangle)$, where $q \in F_m$. Note, that an RSM makes nondeterministic transitions and does not read the input character when it *calls* some component or *returns*.

According to [**?**], recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages [**?**], RSMs are equivalent to context-free languages. Thus RSMs suit to encode query grammars. Any CFG can be easily converted to an RSM with one box per nonterminal. The box which corresponds to a nonterminal A is constructed using the right-hand side of each rule for A.

An example of such RSM R constructed for the grammar G with rules $S \to aSb \mid ab$ is provided in Figure 3.

Since R is a set of FSMs, it can be represented as an adjacency matrix for the graph where vertices are states from $\bigcup_{i \in M} Q_i$ and edges are transitions between q_i^a and q_i^b with the label $l \in \Sigma \cup M$, if $\delta_i(q_i^a, l) = q_i^b$. Thus, similarly to an FSM, an RSM can be represented as a set of Boolean adjacency matrices.

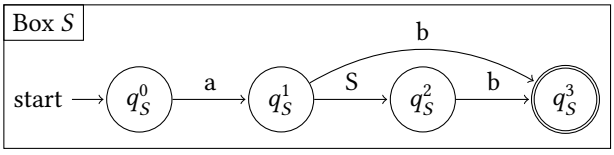


Figure 3: Recursive state machine R for grammar G

As an RPQ, a CFPQ is the intersection of the given context-free language and a FSM specified by the given graph. As far as every context-free language is closed under the intersection with regular languages, such intersection can be represented as an RSM. Also, an RSM can be viewed as an FSM over $\Sigma \cup N$. We use this point of view to propose a unified algorithm to evaluate both regular and context-free path queries with zero overhead for regular queries.

2.4 Graph Kronecker Product and Machines Intersection

In this section we introduce classical Kronecker product definition, describe graph Kronecker product and its relation to Boolean matrices algebra, and RSM and FSM intersection.

Definition 2.15. Given two matrices A and B of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively, with element-wise product operation $\cdot$, the *Kronecker product* of these two matrices is a new matrix $C = A \otimes B$ of size $m_1 * m_2 \times n_1 * n_2$ and $C[u * m_1 + v, n_1 * p + q] = A[u, p] \cdot B[v, q]$.

Definition 2.16. Given two edge-labeled directed graphs $\mathcal{G}_1 = \langle V_1, E_1, L_1 \rangle$ and $\mathcal{G}_2 = \langle V_2, E_2, L_2 \rangle$, the *Kronecker product* of these two graphs is a edge-labeled directed graph $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$, where $\mathcal{G} = \langle V, E, L \rangle$: $V = V_1 \times V_2$, $E = \{((u, v), l, (p, q)) \mid (u, l, p) \in E_1 \wedge (v, l, q) \in E_2\}$ and $L = L_1 \cap L_2$.

The Kronecker product for graphs produces a new graph with a property that if some path $(u, v)\pi(p, q)$ exists in the result graph then paths $u\pi_1 p$ and $v\pi_2 q$ exist in the input graphs, and $\omega(\pi) = \omega(\pi_1) = \omega(\pi_2)$. These paths π_1 and π_2 could be easily found from π by its definition.

The Kronecker product for directed graphs can be described as the Kronecker product of the corresponding adjacency matrices of graphs, what gives the following definition:

Definition 2.17. Given two adjacency matrices M_1 and M_2 of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively for some directed graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, the *Kronecker product* of these two adjacency matrices is the adjacency matrix M of some graph $\mathcal{G}$, where M has size $m_1 * m_2 \times n_1 * n_2$ and $M[u * m_1 + v, n_1 * p + q] = M_1[u, p] \cap M_2[v, q]$.

By the definition, the Kronecker product for adjacency matrices gives an adjacency matrix with the same set of edges as in the resulting graph in the Def. 2.16. Thus, $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$, where $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$.

Definition 2.18. Given two FSMs $T_1 = \langle \Sigma, Q^1, Q^1_S, S^1_F, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, Q^2_S, S^2_F, \delta^2 \rangle$, the *intersection* of these two machines is a new FSM $T = \langle \Sigma, Q, Q_S, S_F, \delta \rangle$, where $Q = Q^1 \times Q^2$, $Q_S = Q^1_S \times Q^2_S$, $Q_F = Q^1_F \times Q^2_F$, $\delta : Q \times \Sigma \to Q$ and $\delta(\langle q_1, q_2 \rangle, s) = \langle q'_1, q'_2 \rangle$, if $\delta(q_1, s) = q'_1$ and $\delta(q_2, s) = q'_2$.

According to [**?**] an FSM intersection defines the machine for which $L(T) = L(T_1) \cap L(T_2)$.

The most substantial part of intersection is the δ function construction for the new machine T. Using adjacency matrices decomposition for FSMs, we can reduce the intersection to the Kronecker product of such matrices over Boolean semiring at some extent, since the transition function δ of the machine T in matrix form is exactly the same as the product result. More precisely:

Definition 2.19. Given two adjacency matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ over Boolean semiring, the *Kronecker product* of these matrices is a new matrix $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$, where $\mathcal{M} = \{M^a_1 \otimes M^a_2 \mid a \in \Sigma\}$ and the element-wise operation is *and* over Boolean values.

Applying the Kronecker product theory for both the FSM and the edge-labeled directed graph, we can intersect these objects as shown in Def. 2.19, since the graph could be interpreted as an FSM with transitions matrix represented as the Boolean adjacency matrix.

In this work we show how to express RSM and FSM intersection in terms of the Kronecker product and transitive closure over Boolean semiring.

3 CONTEXT-FREE PATH QUERYING BY KRONECKER PRODUCT

In this section we introduce the algorithm for CFPQ which is based on Kronecker product of Boolean matrices. The algorithm solves all-pairs CFPQ in all-path semantics (according to Hellings [**?**]) and works in two steps.

(1) *Index creation.* In this step, the algorithm computes an index which contains information necessary to restore paths for given pairs of vertices. This index can be used to solve the reachability problem without extracting paths. Note that this index is finite even if the set of paths is infinite.
(2) *Paths extraction.* All paths for the given pair of vertices can be enumerated by using the index. Since the set of paths can be infinite, all paths cannot be enumerated explicitly, and advanced techniques such as lazy evaluation are required for the implementation. Nevertheless, a single path can always be extracted with standard techniques.

In the following subsections we describe these steps, prove correctness of the algorithm, and provide time complexity estimations.

3.1 Index Creation Algorithm

The *index creation* algorithm outputs the final adjacency matrix $\mathcal{M}_2$ for the input graph with all pairs of vertices which are reachable through some nonterminal in the input grammar G, as well as the index matrix C_3, which is to be used to extract paths in the *path extraction* algorithm.

The algorithm is based on the generalization of the FSM intersection for an RSM, and the edge-labeled directed input graph. Since the RSM is composed as a set of FSMs, it could be easily presented as an adjacency matrix for some graph over labels set $\Sigma \cup S$. As shown in the Def. 2.19, we can apply Kronecker product from Boolean matrices to *intersect* the RSM and the input graph to some extent. But the RSM contains the nonterminal symbols from N with additional *recursive calls* logic, which requires *transitive closure* step to extract such symbols.

The core idea of the algorithm comes from Kronecker product and transitive closure. The algorithm boils down to the iterative Kronecker product evaluation for the RSM adjacency matrix $\mathcal{M}_1$ and the input graph adjacency matrix $\mathcal{M}_2$, followed by transitive closure, extraction of nonterminals and updating the graph adjacency matrix $\mathcal{M}_2$. Listing 1 shows main steps of the algorithm.

3.1.1 Application of Dynamic Transitive Closure. It is easy to see that the most time-consuming steps of the algorithm are the Kronecker product and transitive closure computations. Note that the adjacency matrix $\mathcal{M}_2$ is always changed

Listing 1 Kronecker product based CFPQ using dynamic transitive closure

```
1:  function contextFreePathQuerying(G, 𝒢)
2:      R ← Recursive automata for G
3:      M_1 ← Adjacency matrix for R
4:      M_2 ← Adjacency matrix for 𝒢
5:      A_2 ← Adjacency matrix for 𝒢
6:      C_3 ← The empty matrix
7:      for s ∈ 0..dim(M_1) − 1 do
8:          for i ∈ 0..dim(M_2) − 1 do
9:              M_2[i, i] ← M_2[i, i] ∪ getNonterminals(R, s, s)
10:     while Matrix M_2 is changing do
11:         M'_3 ← M_1 ⊗ A_2
12:         A_2 ← The empty matrix of size n × n
13:         for M'_3[i, j] | M'_3[i, j] = 1 do
14:             C_3[i, j] ← 1
15:             C'_3 ← ⋃_(i,j) add(C_3, i, j) ▷ Updating the transitive closure
16:             C_3 ← C_3 + C'_3
17:         n ← dim(M_3)
18:         for (i, j) | C'_3[i, j] ≠ 0 do
19:             s, f ← getStates(C'_3, i, j)
20:             if getNonterminals(R, s, f) ≠ ∅ then
21:                 x, y ← getCoordinates(C'_3, i, j)
22:                 M_2[x, y] ← M_2[x, y] ∪ getNonterminals(R, s, f)
23:                 A_2[x, y] ← A_2[x, y] ∪ getNonterminals(R, s, f)
24:     return M_2, C_3
25: function getStates(C, i, j)
26:     r ← dim(M_1)                    ▷ M_1 is adjacency matrices for R
27:     return ⌊i/r⌋, ⌊j/r⌋
28: function getCoordinates(C, i, j)
29:     n ← dim(M_2)                    ▷ M_2 is adjacency matrices for 𝒢
30:     return i mod n, j mod n
```

incrementally i. e. elements (edges) are added to $\mathcal{M}_2$ (and are never deleted from it) at each iteration of the algorithm. So it is not necessary to recompute the whole product or transitive closure if an appropriate data structure is maintained.

To compute the Kronecker product, we employ the fact that it is left-distributive. Let $\mathcal{A}_2$ be a matrix with newly added elements and $\mathcal{B}_2$ be a matrix with the all previously found elements, such that $\mathcal{M}_2 = \mathcal{A}_2 + \mathcal{B}_2$. Then by the left-distributivity of the Kronecker product we have $\mathcal{M}_1 \otimes \mathcal{M}_2 = \mathcal{M}_1 \otimes (\mathcal{A}_2 + \mathcal{B}_2) = \mathcal{M}_1 \otimes \mathcal{A}_2 + \mathcal{M}_1 \otimes \mathcal{B}_2$. Note that $\mathcal{M}_1 \otimes \mathcal{B}_2$ is known and is already in the matrix $\mathcal{M}_3$ and its transitive closure also is already in the matrix C_3, because it has been calculated at the previous iterations, so it is left to update some elements of $\mathcal{M}_3$ by computing $\mathcal{M}_1 \otimes \mathcal{A}_2$.

The fast computation of transitive closure can be obtained by using incremental dynamic transitive closure technique. Now we describe the function *add* from Listing 1. Let C_3 be a transitive closure matrix of the graph G with n vertices. We use an approach by Ibaraki and Katoh [**?**] to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices, which become reachable after insertion of the certain edge. We have modified it to achive a logarithmic speed-up.

For each newly inserted edge (i, j) and every node $u \neq j$ of G such that $C_3[u, i] = 1$ and $C_3[u, j] = 0$, one needs to perform operation $C_3[u, v] = C_3[u, v] \wedge C_3[j, v]$ for every node v, where $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$ and $0 \wedge 1 = 1$. Notice that these operations are equivalent to the element-wise (Hadamard) product of two vectors of size n, where multiplication operation is denoted as $\wedge$. To check whether $C_3[u, i] = 1$ and $C_3[u, j] = 0$ one needs to multiply two vectors: the first vector represents reachability of the given vertex i from other vertices $\{u_1, u_2, ..., u_n\}$ of the graph and the second vector represents the same for the given vertex j. The operation $C_3[u, v] \wedge C_3[j, v]$ also can be reduced to the computation of the Hadamard product of two vectors of size n for the given u_k. The first vector contains the information whether vertices $\{v_1, v_2, ..., v_n\}$ of the graph are reachable from the given vertex u_k and the second vector represents the same for the given vertex j. The element-wise product of two vectors can be calculated naively in time $O(n)$. Thus, the time complexity of the transitive closure can be reduced by speeding up element-wise product of two vectors of size n.

To achieve logarithmic speed-up, we use the Four Russians' trick. First we split each vector into $n/\log n$ parts of size $\log n$. Then we create a table S such that $S(a, b) = a \wedge b$ where $a, b \in \{0, 1\}^{\log n}$. This takes time $O(n^2 \log n)$, since there are $2^{\log n} = n$ variants of Boolean vectors of size $\log n$ and hence n^2 possible pairs of vectors (a, b) in total, and each component takes $O(\log n)$ time. With table S, we can calculate product of two parts of size $\log n$ in constant time. There are $n/\log n$ such parts, so the element-wise product of two vectors of size n can be calculated in time $O(n/\log n)$ with $O(n^2 \log n)$ preprocessing.

Theorem 3.1. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. Let $\mathcal{M}_2$ be a resulting adjacency matrix after the execution of the algorithm in Listing 1. Then for any valid indices i, j and for each nonterminal $A \in N$ the following statement holds: the cell $M^A_{2,(k)}[i, j]$ contains $\{1\}$, iff there is a path $i\pi j$ in the graph $\mathcal{G}$ such that $A \xrightarrow{*} l(\pi)$.*

Proof. The main idea of the proof is to use induction on the height of the derivation tree obtained on each iteration. □

Theorem 3.2. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The algorithm from Listing 1 calculates resulting matrices $\mathcal{M}_2$ and C_3 in $O(n^3/\log n)$ time where $n = |V|$. Moreover, the maintaining of the dynamic transitive closure dominates the cost of the algorithm.*

Proof. Let $|\mathcal{A}|$ be a number of non-zero elements in a matrix $\mathcal{A}$. Consider the total time which is needed for computing the Kronecker products. The elements of the matrices

$\mathcal{A}_2^{(i)}$ are pairwise distinct on every i-th iteration of the algorithm therefore the total number of operations is

$$\sum_i T(\mathcal{M}_1 \otimes \mathcal{A}_2^{(i)}) = |\mathcal{M}_1| \otimes \sum_i |\mathcal{A}_2^{(i)}| = |\mathcal{M}_1| O(n^2).$$

Now we derive the time complexity of maintainig the dynamic transitive closure. Since C_3 has size of $O(n^2)$, no more than $O(n^2)$ edges will be added during all iterations of the algorithm. Checking condition whether $C_3[u, i] = 1$ and $C_3[u, j] = 0$ for every node $u \in V$ for each newly inserted edge (i, j) requires one multiplication of vectors per insertion, thus total time is $O(n^3/\log n)$. Note that after checking the condition, at least one element $C[u', j]$ changes value from 0 to 1 and then never becomes 0 for some u' and j. Therefore the operation $C_3[u', v] = C_3[u', v] \wedge C_3[j, v]$ for all $v \in V$ is executed at most once for every pair of vertices (u', j) during the entire computation implying that the total time is equal to $O(n^2 n/\log n) = O(n^3/\log n)$ (using multiplication of vectors).

The matrix C_3' contains only new elements, therefore C_3 can be updated directly using only $|C_3'|$ operations and hence $O(n^2)$ operations in total. The same holds for cycle in line 18 of the algorithm from Listing 1, because operations are performed only for non-zero elements of the matrix $|C_3'|$. Finally, the time complexity of the algorithm is $O(n^2) + O(n^2 \log n) + O(n^3/\log n) + O(n^2) + O(n^2) = O(n^3/\log n)$. □

3.1.2 Index creation for RPQ. In case of the RPQ, the main **while** loop takes only one iteration to actually append data. Since the input query is provided in the form of the regular expression, one can construct the corresponding RSM, which consists of the single *component state machine*. This CSM is built from the regular expression and is labeled as S, for example, which has no *recursive calls*. The adjacency matrix of the machine is build over Σ only. Therefore, calculating the Kronecker product, all relevant information is taken into account at the first iteration of the loop.

3.2 Paths Extraction Algorithm

After the index has been created, one can enumerate all paths between specified vertices. The index stores information about all reachable pairs for all nonterminals. Thus, the most natural way to use this index is to query paths between the specified vertices derivable from the specified nonterminal.

To do so, we provide a function GetPaths(v_s, v_f, N), where v_s is a start vertex of the graph, v_f — the final vertex, and N is a nonterminal. Implementation of this function is presented in Listing 2.

Paths extraction is implemented as three mutually recursive functions. The entry point is GetPaths(v_s, v_f, N). This function returns a set of the paths between v_s and v_f such that the word formed by a path is derivable from the nonterminal N.

Listing 2 Paths extraction algorithm

```
1:  C_3 ← result of index creation algorithm: final transitive closure
2:  M_1 ← the set of adjacency matrices of the input RSM
3:  M_2 ← the set of adjacency matrices of the final graph
4:  function GetPaths(v_s, v_f, N)
5:      q_N^0 ← Start state of automata for N
6:      F_N ← Final states of automata for N
7:      res ← ⋃_{f∈F_N} GetPathsInner((q_N, v_s), (f, v_f))
8:      return res
9:  function GetSubpaths((s_i, v_i), (s_j, v_j), (s_k, v_k))
10:     l ← {(v_i, t, v_k) | M_2^t[s_i, s_k] ∧ M_1^t[v_i, v_k]}
              ∪ ⋃_{N|M_2^N[s_i,s_k]} GetPaths(v_i, v_k, N)
              ∪ GetPathsInner((s_i, v_i), (s_k, v_k))
11:     r ← {(v_k, t, v_j) | M_2^t[s_k, s_j] ∧ M_1^t[v_k, v_j]}
              ∪ ⋃_{N|M_2^N[s_k,s_j]} GetPaths(v_k, v_j, N)
              ∪ GetPathsInner((s_k, v_k), (s_j, v_j))
12:     return l · r
13: function GetPathsInner((s_i, v_i), (s_j, v_j))
14:     parts ← {(s_k, v_k) | C_3[(s_i, v_i), (s_k, v_k)] = 1 ∧
        C_3[(s_k, v_k), (s_j, v_j)] = 1}
15:     return ⋃_{(s_k,v_k)∈parts} GetSubpaths((s_i, v_i), (s_j, v_j), (s_k, v_k))
```

To compute such paths, it is necessary to compute paths from vertices of the form (q_N^s, v_s) to vertices of the form (q_N^f, v_f) in the result of transitive closure, where q_N^s is an initial state of RSM for N and q_N^f is a final state. The function GetPathsInner$((s_i, v_i), (s_j, v_j))$ is used to do it. This function finds all possible vertices (s_k, v_k) which split a path from (s_i, v_i) to (s_j, v_j) into two subpaths. After that, function GetSubpaths$((s_i, v_i), (s_j, v_j), (s_k, v_k))$ computes the corresponding subpaths. Each subpath may be at least a single edge. If single-edge subpath is labeled by terminal then corresponding edge should be added to the result else (label is nonterminal) GetPaths should be used to restore paths. If subpath is longer then one edge, GetPaths should be used to restore paths.

It is assumed that the sets are computed lazily, so that to ensure termination in the case of an infinite number of paths. We also do not check paths for duplication manually, since they are assumed to be represented as sets.

4 IMPLEMENTATION DETAILS

Currenly, our goal is to evaluate the applicability of the proposed algorithm, thus we implemented its naive version. We compute the transitive closure from scratch on each iteration

and do not use any incremental techniques. In our implementation we use PyGraphBLAS[2] — a Python wrapper for SuiteSparse library [**?**][3]. SuiteSparse is a C implementation of GraphBLAS [**?**] standard which introduces linear algebra building blocks for implementation of graph analysis algorithms. Thus we provide a highly-optimized parallel CPU implementation of the naive version of the algorithm[4].

At present, we do not integrate with a graph database and a graph query language. We suppose that the input graph is stored in a file, while the query is expressed in terms of a context-free grammar and is also stored in file. As it was shown in [**?**], it is possible to integrate SuiteSparse based implementation in the RedisGraph database. Providing integration with a query language requires a lot of technical effort to extend the language. There are existing proposals, for example to extend the Cypher language[5].

Paths extraction is implemented in Python by using PyGraphBLAS. Since lazy evaluation is not natural for Python, we cap the maximal number of paths to extract in the implementation.

5 EVALUATION

We evaluate the implemented algorithm on both regular and context-free path queries in order to demonstrate applicability of the proposed solution. Namely, goals of the evaluation are following.

(1) Investigate the practical applicability of RPQ evaluation by the proposed algorithm.
(2) Compare Azimov's algorithm for reachability CFPQ and the proposed algorithm.
(3) Investigate the practical applicability of paths extraction algorithm for both regular and context-free queries.

For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7-6700 CPU, 3.4GHz, and DDR4 64Gb RAM. As far as we evaluate only algorithm execution time, we store each graph fully in RAM as its adjacency matrix in sparse format. Note, that graph loading time is not included in the result time of evaluation.

5.1 RPQ Evaluation

In oder to investigate applicability of the proposed algorithm for RPQ over real-world graphs we collect a set of real-world and synthetic graphs and evaluate queries generated by using the most popular templates for RPQs.

Graph	#V	#E
LUBM1k	120 926	484 646
LUBM3.5k	358 434	144 9711
LUBM5.9k	596 760	2 416 513
LUBM1M	1 188 340	4 820 728
LUBM1.7M	1 780 956	7 228 358
LUBM2.3M	2 308 385	9 369 511
Uniprotkb	6 442 630	24 465 430
Proteomes	4 834 262	12 366 973
Taxonomy	5 728 398	14 922 125
Geospecies	450 609	2 201 532
Mappingbased_properties	8 332 233	25 346 359

Table 1: Graphs for RPQ evaluation

5.1.1 Dataset. Brief description of collected graphs are presented in Table 1. Namely, the dataset consists of several parts. The first one is a set of LUBM graphs[6] [**?**] with a different number of vertices. The second one is a graphs from Uniprot database[7]: *proteomes*, *taxonomy* and *uniprotkb*. The last part is a RDF files *mappingbased_properties* from DBpedia[8] and *geospecies*[9]. These graphs represent data from different areas and they are frequently used for graph querying algorithms evaluation.

Queries for evaluation was generated by using templates of the most popular RPQs which are collected from [**?**] (Table 2) and [**?**] (some of complex queries from Table 5), and are presented in table 2. We generate 10 queries for each template and each graph using the most frequent relations from the given graph randomly[10]. For all LUBM graphs common set of queries was generated in order to investigate scalability of the proposed algorithm.

5.1.2 Results. For index creation average time of 5 runs is presented.

Index creation time for each query for LUBM graphs set is presented in figure 4. We can see, that query evaluation time depends on query: there are queries which evaluate less

[2]GitHub repository of PyGraphBLAS, a Python wrapper for GraphBLAS API: https://github.com/michelp/pygraphblas. Access date: 07.07.2020.
[3]Web page of SuiteSparse:GraphBLAS library: http://faculty.cse.tamu.edu/davis/GraphBLAS.html. Access date: 07.07.2020.
[4]Implementation of the described algorithm is published here: https://github.com/JetBrains-Research/CFPQ_PyAlgo. Access date: 07.07.2020.
[5]Cypher language extension proposal which introduces a syntax to express context-free queries: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. Access date: 07.07.2020.
[6]Lehigh University Benchmark (LUBM) web page: http://swat.cse.lehigh.edu/projects/lubm/. Access date: 07.07.2020.
[7]Universal Protein Resource (UniProt) web page: https://www.uniprot.org/. All files used for evaluation can be downloaded here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf/. Access date: 07.07.2020.
[8]DBpedia project web site: https://wiki.dbpedia.org/. Access date: 07.07.2020.
[9]The Geospecies RDF: https://old.datahub.io/dataset/geospecies. Access date: 07.07.2020.
[10]Used generator is available as part of CFPQ_data project: https://github.com/JetBrains-Research/CFPQ_Data/blob/master/tools/gen_RPQ/gen.py. Access data: 07.07.2020.

Name	Query	Name	Query
Q_1	a^*	Q_9^5	$(a \mid b \mid c \mid d \mid e)^+$
Q_2	$a \cdot b^*$	Q_{10}^2	$(a \mid b) \cdot c^*$
Q_3	$a \cdot b^* \cdot c^*$	Q_{10}^3	$(a \mid b \mid c) \cdot d^*$
Q_4^2	$(a \mid b)^*$	Q_{10}^4	$(a \mid b \mid c \mid d) \cdot e^*$
Q_4^3	$(a \mid b \mid c)^*$	Q_{10}^5	$(a \mid b \mid c \mid d \mid e) \cdot f^*$
Q_4^4	$(a \mid b \mid c \mid d)^*$	Q_{10}^2	$a \cdot b$
Q_4^5	$(a \mid b \mid c \mid d \mid e)^*$	Q_{11}^3	$a \cdot b \cdot c$
Q_5	$a \cdot b^* \cdot c$	Q_{11}^4	$a \cdot b \cdot c \cdot d$
Q_6	$a^* \cdot b^*$	Q_{11}^5	$a \cdot b \cdot c \cdot d \cdot f$
Q_7	$a \cdot b \cdot c^*$	Q_{12}	$(a \cdot b)^+ \mid (c \cdot d)^+$
Q_8	$a? \cdot b^*$	Q_{13}	$(a \cdot (b \cdot c)^*)^+ \mid (d \cdot f)^+$
Q_9^2	$(a \mid b)^+$	Q_{14}	$(a \cdot b \cdot (c \cdot d)^*)^+ \cdot (e \mid f)^*$
Q_9^3	$(a \mid b \mid c)^+$	Q_{15}	$(a \mid b)^+ \cdot (c \mid d)^+$
Q_9^4	$(a \mid b \mid c \mid d)^+$	Q_{16}	$a \cdot b \cdot (c \mid d \mid e)$

Table 2: Queries templates for RPQ evaluation

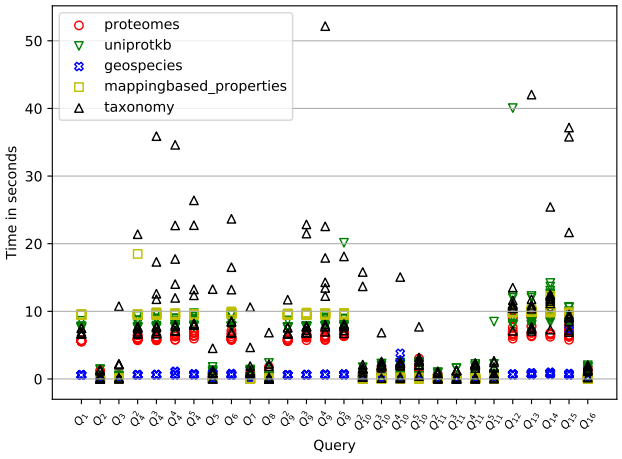


Figure 5: Index creation time for real-world RDFs

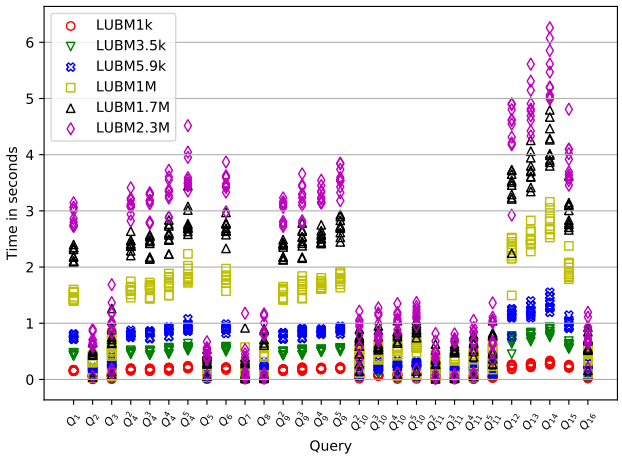


Figure 4: Index creation time for LUBM graphs

then 1 second even for biggest graph (Q_2, Q_5, Q_{11}^2, Q_{11}^3), while worst time is 6.26 seconds (Q_{14}). Anyway, we can argue that in this case our algorithm demonstrates reasonable time to be applied for real-world data analysis, because it is comparable with recent results on the same problem for LUBM querying by using distributed system over 10 nodes [**?**], while we use only one node. Note, that accurate comparison of different approaches is a huge interesting work for the future.

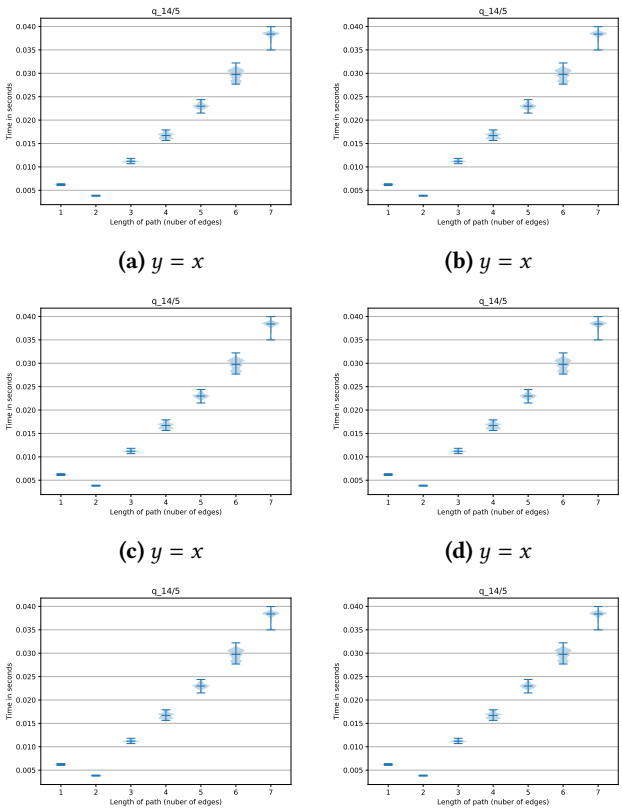
Index creation time for each query for for real-world graphs is presented in figure 5. We can see that query evaluation time depends on graph inner structure. First of all, in some cases handling of small graph requires more time, then handling bigger graph. For example, Q_{10}^4: querying the *geospecies* graph (450k vertices) in some cases requires more time than querying of *mappingbased_properties* (8.3M vertices) and *taxonomy* (5.7M vertices). On the other hand, *taxonomy* querying in relatively big number of cases requires significantly more time, than querying of other graphs, while *taxonomy* is not a bigger graph. Finally, we can see, that in big number of cases query execution time requires less then 10 seconds, even for big graph, and no queries which require more then 52.17 seconds.

Paths extraction was evaluated on cases with possible long paths. These cases were selected during index creation by using number of iterations in transitive closure evaluation. For each selected graph and query we measure paths extraction time for each reachable pair, index creation time is not included because exactly the same index, as calculated at the previous step, is used for paths extraction.

We evaluate two scenarios. The first one is a single path extraction. In this case results are represented as a dependency of extraction time on extracted path length. We can see linear !!!!

The second scenario is many paths extraction. Here we limit a number of path to extract by !!! In this case results are represented as a dependency of extraction time on number of extracted paths.

5.1.3 Conclusion. We can conclude that proposed algorithm is applicable for real-world data processing: the algorithm allows one both to solve reachability problem and to extract paths of interest in reasonable time even using naïve implementation. Paths extraction should be tuned for specific tasks. Detailed comparison with other algorithms is a work for future.



(a) $y = x$ **(b)** $y = x$

(c) $y = x$ **(d)** $y = x$

(e) $y = x$ **(f)** $y = x$

Figure 6: Single path extraction

5.2 CFPQ Evaluation

In order to investigate applicability of the proposed algorithm for CFPQ procesiing over real-world graphs we evaluate it on a number of classical cases and compare witn Azimov's algorithm. As far as currently only single path version of Azimov's algorithm exists, we use this version implemented by using PyGraphBLAS for comparison.

5.2.1 Dataset. We use CFPQ_Data[11] dataset for evaluation. Namely, we use relatevely big RDF files and respective same-generation queries G_1 (eq. 1) and G_2 (eq. 2) which are used in other works for CFPQ evaluation. Also we use *Geo* (eq. 3) query which was provided by J. Kyjpera et. al [**?**] for *geospecies* RDF.

$$S \rightarrow aSb \tag{1}$$

$$S \rightarrow aSb \tag{2}$$

[11] CFPQ_Data is a dataset for CFPQ evaluation which contains both synthetic and real-world data and queries https://github.com/JetBrains-Research/CFPQ_Data. Access date: 07.07.2020.

Graph	#V	#E
eclass_514en	120 926	484 646
enzyme	358 434	144 9711
geospecies	596 760	2 416 513
go	1 188 340	4 820 728
go-hierarchy	1 780 956	7 228 358
taxonomy	2 308 385	9 369 511
arch	6 442 630	24 465 430
crypto	6 442 630	24 465 430
drivers	6 442 630	24 465 430
fs	6 442 630	24 465 430

Table 3: Graphs for CFPQ evaluation

Table 4: RDFs querying time (measured in seconds)

Name	G_1		G_2		*Geo*		*MA*	
	Tns	Mtx	Tns	Mtx	Tns	Mtx	Tns	Mtx
eclass_514en	0.254	0.195	0.227	...	—	—	—	—
enzyme	0.035	0.029	0.036	...	—	—	—	—
geospecies	0.091	...	0.001	...	...	...	—	—
go-hierarchy	0.186	0.976	0.293	...	—	—	—	—
go	1.676	1.286	1.368	...	—	—	—	—
pathways	0.015	0.021	0.009	...	—	—	—	—
taxonomy	5.366	.....	3.282	...	—	—	—	—
arch	—	—	—	—	—	—	390.046	...
crypto	—	—	—	—	—	—	395.979	...
drivers	—	—	—	—	—	—	2114.156	...
fs	—	—	—	—	—	—	745.973	...

$$S \rightarrow aSb \tag{3}$$

$$S \rightarrow aSb \tag{4}$$

Additionally we evaluate our algorithm on memory aliases analysis problem: a classical problem, which can be reduced to CFPQ [**?**]. To do it we use some graphs builded for different parts of Linux OS kernel (*arch*, *crypto*, *drivers*, *fs*) and respective query *MA* (eq. 4) Detailed information about all used graphs are presented in table 3

5.2.2 Results. For index creation average time of 5 runs is presented. Results on index creation for both Azimov's algorithm (**Mtx**) and the proposed algortihm (**Tns**) are presented in table 4.

We can see that !!!!

Results on paths extraction is presented !!!!

5.2.3 Conclusion. We can conclude that !!!

To summirize averall evaluation, the proposed algorithm is applicable for both RPQ and CFPQ over real-world graphs. Thus, the proposed solution is an proctical unified algorithm for both RPQ and CFPQ evaluation.

6 RELATED WORK

Language constrained path querying is widely used in graph databases, static code analysis, and other areas. Both, RPQ and CFPQ (known as CFL reachability problem in static code analysis) are actively studied in the recent years.

There is a huge number of theoretical research on RPQ and its specific cases. RPQ with single-path semantics was investigated from the theoretical point of view by Barrett et al. [**?**]. In order to research practical limits and restrictions of RPQ, a number of high-performance RPQ algorithms were provided. For example, derivative-based solution provided by Maurizio Nolé and Carlo Sartiani which is implemented on the top of Pregel-based system [**?**], or solution of André Koschmieder et al. [**?**]. But only a limited number of practical solutions provide the ability to restore paths of interest. A recent work of Xin Wang et al. [**?**] provides a Pregel-based provenance-aware RPQ algorithm which utilizes a Glushkov's construction [**?**]. There is a lack of research of the applicability of linear algebra-based RPQ algorithms with paths-providing semantics.

On the other hand, many CFPQ algorithms with various properties were proposed recently. They employ the ideas of different parsing algorithms, such as CYK in works of Jelle Hellings [**?**] and Phillip Bradford [**?**], (G)LR and (G)LL in works of Ekaterina Verbitskaia et al. [**?**], Semyon Grigorev et al. [**?**], Fred Santos et al. [**?**], Ciro Medeiros et al. [**?**]. Unfortunately, none of them has better than cubic time complexity in terms of the input graph size. The algorithm of Azimov [**?**] is, best to our knowledge, the first algorithm for CFPQ based on linear algebra. It was shown by Arseniy Terekhov et al. [**?**] that this algorithm can be applied for real-world graph analysis problems, while Jochem Kuijpers et al. shows in [**?**] that other state-of-the-art CFPQ algorithms are not performant enough to handle real-world graphs.

It is improtant in both RPQ and CFPQ to be able to restore paths of interest. Some of the mentioned algorithms can solve only the reachability problem, while it may be important to provide at least one path which satisfies the query. While Arseniy Terekhov et al. [**?**] provide the first CFPQ algorithm with single path semantics based on linear algebra, Jelle Hellings in [**?**] provides the first theoretical investigation of this problem. He also provides an overview of the related works and shows that the problem is related to the string generation problem and respective results from the formal language theory. He concludes that both theoretical and empirical investigation of CFPQ with single-path and all-path semantics are at early stage. We agree with this point of view, and we only demonstrate applicability of our solution for paths extraction and do not investigate its properties in details.

Developing a truly subcubic CFPQ algorithm is a long-standing problem which is actively studied in both graph database and static code analysis communities. The question on the existence of a subcubic CFPQ algorithm was stated by Mihalis Yannakakis in 1990 in [**?**]. A bit later Thomas Reps proposed the CFL reachability as a framework for interprocedural static code analysis [**?**]. Melski and Reps gave a dynamic programming formulation of the problem which runs in $O(n^3)$ time [**?**]. The problem of the cubic bottleneck of context-free language reachability is also discussed by Heintze and McAllester [**?**], and Melski and Reps [**?**]. The slightly subcubic algorithm with $O(n^3/\log n)$ time complexity was provided by Swarat Chaudhuri in [**?**]. This result is inspired by recursive state machine reachability. The first truly subcubic algorithm with $O(n^\omega polylog(n))$ time complexity (ω is the best exponent for matrix multiplication) for an arbitrary graph and 1-Dyck language was provided by Phillip Bradford in [**?**], and Andreas Pavlogiannis and Anders Alnor Mathiasen in [**?**]. Other partial cases were investigated by Krishnendu Chatterjee et al. in [**?**], Qirun Zhang in [**?**].

The utilization of linear algebra is a promising way to high-performance graph analysis. There are many works on specific graph algorithm formulation in terms of linear algebra, for example, classical algorithms for transitive closure and all-pairs shortest paths. Recently this direction was summarized in GrpahBLAS API [**?**] which provides building blocks to develop a graph analysis algorithm in terms of linear algebra. There is a number of implementations of this API, such as SuiteSparse:GraphBLAS [**?**] or CombBLAS [**?**]. Approaches to evaluate different classes of queries in different systems based on linear algebra is being actively researched. This approach demonstrates significant performance improvement when applied for SPARQL queries evaluation [**? ?**] and for Datalog queries evaluation [**?**]. Finally, RedisGraph [**?**], a linear-algebra powered graph database, was created and it was shown that in some scenarios it outperforms many other graph databases.

7 CONCLUSION AND FUTURE WORK

In this work we present an improved version of the tensor-based algorithm for CFPQ: we reduce the algorithm to operations over Boolean matrices, and we provide the ability to extract all paths which satisfy the query. Moreover, the provided algorithm can handle grammars in EBNF, thus it does not require CNF transformation of the grammar and avoids grammar explosion. As a result, the algorithm demonstrates practical performance not only on CFPQ queries but also on RPQ ones, which is shown by our evaluation. Thus, we provide a universal linear algebra based algorithm for RPQ and CFPQ evaluation with all-paths semantics. Moreover our

algorithm opens a way to tackle the long-standing problem of subcubic CFPQ by reducing it to incremental transitive closure: incremental transitive closure with $O(n^{3-\varepsilon})$ total update time for n^2 updates, such that each update returns all of the new reachable pairs, implies $O(n^{3-\varepsilon})$ CFPQ algorithm. We prove $O(n^3/\log n)$ time complexity by providing $O(n^3/\log n)$ incremental transitive closure algorithm.

The first important task for future research is a detailed investigation of the paths extraction algorithm. Jelle Hellings in [**?**] provides a theoretical investigation of single-path extraction and shows that the problem is related to the formal language theory. Extraction of all paths is more complicated and should be investigated carefully in order to provide an optimal algorithm.

Recent hardness results for dynamic graph problems demonstrates that any further improvement for incremental transitive closure (and, hence, CFPQ) will imply a major breakthrough for other long-standing problems. For example, there is no incremental transitive closure algorithm with total update time $O(mn)^{1-\varepsilon}$ (n denotes the number of graph vertices, m is the number of graph edges) even with polynomial $poly(n)$ time preproccessing of the input graph assuming that the online matrix-vector (OMv) conjecture is true [**?**].

Thus, the first task for the future is to improve the logarithmic factor in the obtained bound. It is also interesting to improve bounds in partial cases for which dynamic transitive closure can be supported faster than in general case, for example, planar graphs [**?**], undirected graph and others. In the case of planarity, it is interesting to investigate properties of the input graph and grammar which allow us to preserve planarity during query evaluation.

From the practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work, we evaluate naive implementation in which transitive closure is recalculated from scratch on each iteration. It is shown in [**?**] that some advanced algorithms for dynamic transitive closure can be efficiently implemented. Can one of these algorithms be efficiently parallelized and utilized in the proposed algorithm?

Also, it is necessary to evaluate GPGPU-based implementation. Evaluation of Azimov's algorithm shows that it is possible to improve performance by using GPGPU because operations of linear algebra can be efficiently implemented on GPGPU [**? ?**]. Moreover, for practical reason, it is interesting to provide a multi-GPU version of the algorithm and to utilize unified memory, which is suitable for linear algebra based processing of out-of-GPGPU-memory data and traversing on large graphs [**? ?**].

In order to simplify the distributed processing of huge graphs, it may be necessary to investigate different formats for sparse matrices, such as HiCOO format [**?**]. Another interesting question in this direction is about utilization of virtualization techniques: should we implement distributed version of algorithm manually or it can be better to use CPU and RAM virtualization to get a virtual machine with huge amount of RAM and big number of computational cores. The experience of the Trinity project team shows that it can make sense [**?**].

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph[12] [**?**] is a suitable candidate for this purpose. This database uses SuiteSparse—an implementation of GraphBLAS standard—as a base for graph processing. This fact allowed to Arseny Terkhov et al. to integrate Azimov's algorithm to RedisGraph with minimal effort [**?**].

[12]RedisGraph is a graph database that is based on the Property Graph Model. Project web page: https://oss.redislabs.com/redisgraph/. Access date: 07.07.2020.