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 Orachev
egor.orachev@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

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

Semyon Grigorev
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com
Saint Petersburg State University
St. Petersburg, Russia
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 with 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 way to construct a slightly subcubic algorithm for CFPQ which is based on linear algebra and on a classical graph-theoretic problem (incremental transitive closure), rather than the way proposed by Swarat Chaudhuri. Our evaluation shows that this algorithm is a good candidate to be a universal algorithm for both regular and context-free path querying.

ACM ISBN XXX-X-XXXXX-XXX-X…$15.00

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.*

ACM Reference Format:
Ekaterina Shemetova, Rustam Azimov, Egor Orachev, 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, 18 pages.

1 INTRODUCTION

Language-constrained path querying [5] is one of the techniques 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 widely spread. Different aspects of RPQs are actively studied in graph databases [2, 4, 33], and support of regular constraints is implemented in such popular query languages as PGQL [49], or SPARQL[1] [30] (property paths). Even that, the improvement of RPQ algorithms efficiency on huge graphs is an

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

actual problem nowadays, and new algorithms and solutions are being created [38, 51].

At the same time, the utilization of more powerful languages, namely context-free languages, gains popularity in the last few years. *Context-Free Path Querying* problem (CFPQ) was introduced by Yannakakis in 1990 in [52]. A number of different algorithms were proposed since that time, but recently, in [31] Kuijpers et al. show that state-of-the-art CFPQ algorithms are not performant enough to be used in practice. This fact motivates us to find new algorithms for CFPQ.

One of the promising ways to achieve high-performance solutions for graph analysis problems is to reduce graph problems to linear algebra operations. This way, the description of basic linear algebra primitives, the GraphBLAS [28] API, was proposed. Solutions that use libraries that implement this API, such as SuiteSparce [14] and CombBLAS [8], show that reduction to linear algebra is a way to utilize high-performance parallel and distributed computations for graph analysis.

Azimov in [3] shows how to reduce CFPQ to matrix multiplication. Late, in [37] and [47], it was shown that utilization of appropriate libraries for linear algebra for Azimov's algorithm implementation allows one to get 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, recently, an algorithm based on automata intersection was proposed [39]. This algorithm is based on linear algebra and does not require the input grammar transformation. In this work, we improve it. First of all, we reduce the above mentioned solution to operations over Boolean matrices, thus simplify 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: we can use one algorithm to evaluate both regular and context-free queries.

Moreover, we show that this algorithm opens the way to attack a long-standing problem whether there is a truly-subcubic $O(n^{3-\epsilon})$ CFPQ algorithm [11, 52]. The best-known result is an $O(n^3/\log n)$ algorithm of Chaudhuri [11]. Also, there are truly subcubic solutions using fast matrix multiplication for some fixed subclasses of context-free languages. For example, there is a truly subcubic $O(n^{3-\epsilon})$ time algorithm for 1-Dyck language proposed by Bradford [7]. Unfortunately, this solution cannot be generalized to arbitrary CFPQs. So, in our knowledge, there is no truly subcubic algorithm for CFPQs. In this work, we show that incremental transitive closure is a bottleneck on the way to get subcubic time complexity for CFPQ.

To summarize, we make the following contributions in this paper.

(1) We give an upper bound on the complexity of the CFPQ algorithm in dependence on the size of the query (context-free grammar). We show that althouth the proposed algorithm has cubic complexity in terms of the grammar size, which is comparable with the state-of-the-art CFPQ solutions, the algorithm does not require transforming the input grammar to Chomsky Normal Form as other solutions (which leads to the at least quadratic blow-up in grammar size), and, thus, it is faster in terms of the grammar size.
(2) We implement ... index creation with dynamic transitive closure... Our results show that ...
(3) Path extraction ...

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* [5] 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
- 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.

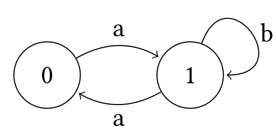


Figure 1: The example of input graph $\mathcal{G}$

Definition 2.2. An *adjacency matrix* for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L\rangle$ is a matrix M, where:

- M has size $|V| \times |V|$
- $M[i, j] = \{l \mid e = (i, l, j) \in E\}$

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

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

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 $\mathcal{M} = \{M^l \mid l \in L, M^l[i,j] = 1 \iff l \in M[i,j]\}$.

In our work 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 as presented in Figure 2.

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

Figure 2: The representation of the matrix M_2 as a set of Boolean matrices

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) is regular expression
- ε (empty string) is regular expression
- $a_i \in \Sigma$ is regular expression
- 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,
- $\delta : Q \times \Sigma \to Q$ is a transition function.

It is well known, that every regular expression can be converted to deterministic FSM without ε-transitions [25]. 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 3.

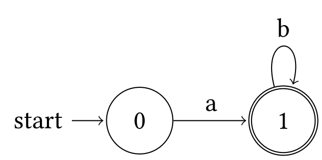


Figure 3: The example of graph representation of FSM for the regular expression ab^*

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 is shown in Figure 4.

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

Figure 4: The representation of the FSM T_1 as a set of Boolean matrices

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 $N_i \in N$, $\alpha \in (\Sigma \cup N)^*$.

The *size of a grammar* $|G|$ is the sum of the sizes of its production rules $|P|$, where the *size of a rule* is one plus the length of its right-hand side. Consider a context-free grammar in *extended Backus-Naur form (EBNF)*. It is a combination of standard Backus-Naur Form with an addition of the constructs often found in regular expressions like quantifiers, and some other extensions. Thus, if its production rule contains a regular expression then the size of such a rule is equal to the sum of the sizes of production rules obtained by the conversion of the regular expression directly to the corresponding linear grammar with the eliminating ε-rules.

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 [1].

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.
- Set of *component state machines* or *boxes*, where $C_i = (\Sigma \cup M, Q_i, q_i^0, F_i, \delta_i)$:
  - $\Sigma \cup M$ 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 is a set of final states for C_i, where $F_i \subseteq Q_i$
  - $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$ 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*. The *size of RSM* $|R|$ is the sum of the number of states in all boxes. 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 $a_1 \dots a_n$ 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 [1], recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages [25], RSMs are equivalent to context-free languages. Thus RSMs suit to encode query grammars. Any CFG $G = \langle \Sigma, N, S, P \rangle$ can be easily converted to an RSM R_G 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. It is easy to see that such a conversion implies that $|R_G| = |G| = |P|$.

An example of such RSM R constructed for the grammar G with rules $S \rightarrow aSb \mid ab$ is provided in Figure 5. For a given example of the grammar and the RSM consider the following sequence of the machine configuration transitions, in case, where one want to determine, if input word *aabb* belongs to the language $L(G)$. The RSM execution starts from configuration $(q_S^0, \langle\rangle)$, reads symbols a and goes to $(q_S^1, \langle\rangle)$. Then, in the nondeterministic manner it tries to read b but fails, and in the same time tries to derive S and goes to configuration $(q_S^0, \langle q_S^2 \rangle)$, where q_S^2 is *return* state. Then machine reads a and goes to $(q_S^1, \langle q_S^2 \rangle)$. In this case, in the nondeterministic choice it fails to derive S, but successfully reads b and goes to configuration $(q_S^3, \langle q_S^2 \rangle)$. Since q_S^3 is final state for the box S, the RSM tries to make *return* and goes to $(q_S^2, \langle\rangle)$. Then it reads b and transits to $(q_S^3, \langle\rangle)$. Since $q_S^3 \in F_S$ and the *return* stack is empty, the machine accepts the input sequence *aabb*.

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 $\delta_i(q_i^a, l) = q_i^b$. An example of such adjacency matrix M_R for the machine R is provided in Section 3.3.

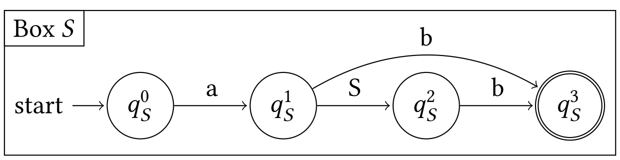


Figure 5: The recursive state machine R for grammar G

Similarly to an FSM, an RSM can be represented as a graph and, hence, as a set of Boolean adjacency matrices. For our example, M_1 is:

$$M_1 = \begin{pmatrix} . & \{a\} & . & . \\ . & . & \{S\} & \{b\} \\ . & . & . & \{b\} \\ . & . & . & . \end{pmatrix}$$

Matrix M_1 can be represented as a set of Boolean matrices as follows:

$$M_1^S = \begin{pmatrix} . & . & . & . \\ . & . & 1 & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, \; M_1^a = \begin{pmatrix} . & 1 & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, \; M_1^b = \begin{pmatrix} . & . & . & . \\ . & . & . & 1 \\ . & . & . & 1 \\ . & . & . & . \end{pmatrix}$$

Similarly to 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 $\Sigma \cup N$. In this work 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 $C[u * m_2 + v, n_2 * p + q] = A[u, p] \cdot B[v, q]$.

It is worth mention, that the Kronecker product produces blocked matrix C, with total number of the blocks $m_1 * n_1$, where each block has size $m_2 * n_2$ and is defined as $A[i, j] \cdot B$.

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 $\mathcal{G} = \langle V, E, L \rangle$:

- $V = V_1 \times V_2$
- $E = \{((u, v), l, (p, q)) \mid (u, l, p) \in E_1 \wedge (v, l, q) \in E_2\}$
- $L = L_1 \cap L_2$

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 $M[u * m_2 + v, n_2 * p + q] = M_1[u, p] \cap M_2[v, q]$.

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 $T_1 = \langle \Sigma, Q^1, Q_S^1, Q_F^1, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, Q_S^2, Q_F^2, \delta^2 \rangle$, the *intersection* of these two machines is a new FSM $T = \langle \Sigma, Q, Q_S, Q_F, \delta \rangle$, where:

- $Q = Q^1 \times Q^2$
- $Q_S = Q_S^1 \times Q_S^2$
- $Q_F = Q_F^1 \times Q_F^2$

- $\delta : Q \times \Sigma \to Q$, $\delta(\langle q_1, q_2 \rangle, s) = \langle q_1', q_2' \rangle$, if $\delta(q_1, s) = q_1'$ and $\delta(q_2, s) = q_2'$

According to [25] 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 $\mathcal{M} = \{M_1^a \otimes M_2^a \mid a \in \Sigma\}$ 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 [22]) 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. For the first step we firstly introduce naïve algortihm. After that we show how to achieve cubic time complexity by using dynamic transitive closure algorithm and shave off a logarithmic factor to achive the best known time complexity for CFPQ.

After that we provide step-by-step example of query evaluation by using the proposed algorithm.

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 $\mathcal{M}_2$.

3.1.1 Naïve Version. Listing 1 shows main steps of the algorithm. The algorithm accepts context-free grammar $G = (\Sigma, N, P)$ and graph $\mathcal{G} = (V, E, L)$ as an input. Start nonterminal is not required here, since the algorithm allows to query paths, which are reachable via some nonterminal from N. An RSM R is created from the grammar G. Note, that R must have no ε-transitions. $\mathcal{M}_1$ and $\mathcal{M}_2$ are the Boolean adjacency matrices for the machine R and the graph $\mathcal{G}$ correspondingly.

Then for each vertex i of the graph $\mathcal{G}$, the algorithm adds loops with non-terminals, which allows deriving ε-word. Here the following rule is implied: each vertex of the graph is reachable by itself through an ε-transition. Since the machine R does not have any ε-transitions, the ε-word could be derived only if a state s in the box B of the R is both initial and final. This data is queried by the *getNonterminals* function for each state s.

The algorithm terminates when the matrix $\mathcal{M}_2$ stops changing. Kronecker product of matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ is evaluated for each iteration. The result is stored in $\mathcal{M}_3$ as a Boolean matrix. Since we are interested only in the reachability of some vertices, there is no need to store a separate Boolean matrix for each label from $\Sigma \cup N$. Therefore, we can collapse it into single Boolean matrix M_3', what is done in the next step. This Boolean matrix could be interpreted as an adjacency matrix for some directed graph without labels with the same set of the vertices, as in the graph formed by $\mathcal{M}_3$. From that point of view the matrix M_3' has the following property from its definition: if some vertices connected by some path in the graph $\mathcal{G}(M_3')$ then these vertices are connected by one or many paths in the graph $\mathcal{G}(\mathcal{M}_3)$.

For the given M_3' a C_3 transitive closure matrix is evaluated by the corresponding function call. Then the algorithm iterates over cells of the C_3. For the pair of indices (i, j), it computes s and f — the initial and final states in the recursive automata R which relate to the concrete $C_3[i, j]$ of the closure matrix. If the given s and f belong to the same box B of R, $s = q_B^0$, and $f \in F_B$, then $getNonterminals$ returns the respective nonterminal. Then for each such nonterminal the respective matrix of the graph adjacency matrix $\mathcal{M}_2$ is updated and a new edge as a Boolean value in the appropriate cell is added.

The functions $getStates$ and $getCoordinates$ (see Listing 2) are used to map indices between Kronecker product arguments and the result matrix. The Implementation appeals to the blocked structure of the matrix C_3, where each block corresponds to some automata and graph edge.

The algorithm returns the computed path extraction index C_3 and the updated matrix $\mathcal{M}_2$, which contains the initial graph $\mathcal{G}$ data as well as data for nonterminals from N. If a cell $M_2^S[i, j]$ for any valid indices i and j and $S \in N$ contains $\{1\}$, then vertex j is reachable from vertex i in grammar G for nonterminal S.

Listing 1 Kronecker product based CFPQ

```
function contextFreePathQuerying(G, 𝒢)
    R ← Recursive automata for G
    M1 ← Boolean adjacency matrix for R
    M2 ← Boolean adjacency matrix for 𝒢
    C3 ← The empty matrix
    for s ∈ 0..dim(M1) − 1 do
        for S ∈ getNonterminals(R, s, s) do
            for i ∈ 0..dim(M2) − 1 do
                M2^S[i, i] ← {1}
    while Matrix M2 is changing do
        M3 ← M1 ⊗ M2                          ▷ Evaluate Kronecker product
        M3' ← ⋁_{M3^a ∈ M3} M3^a              ▷ Collapse to Boolean matrix
        C3 ← transitiveClosure(M3')
        n ← dim(M3)                           ▷ Matrix M3 size = n × n
        for (i, j) ∈ [0..n − 1] × [0..n − 1] do
            if C3[i, j] then
                s, f ← getStates(C3, i, j)
                x, y ← getCoordinates(C3, i, j)
                for S ∈ getNonterminals(R, s, f) do
                    M2^S[x, y] ← {1}
    return M2, C3
```

Listing 2 Help functions for Kronecker product based CFPQ

```
function getStates(C, i, j)
    r ← dim(M1)                  ▷ M1 is adjacency matrices for R
    return ⌊i/r⌋, ⌊j/r⌋
function getCoordinates(C, i, j)
    n ← dim(M2)                  ▷ M2 is adjacency matrices for 𝒢
    return i mod n, j mod 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 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 labeled as the 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.

Lemma 3.1. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. Let $\mathcal{M}_{2,(k)}$ be an adjacency matrix $\mathcal{M}_2$ after the execution of some iteration $k \geq 0$ of the algorithm in Listing 1. Then for any valid indices i, j and for each nonterminal $A \in N$ such that cell $M_{2,(k)}^A[i, j]$ contains $\{1\}$, the following statement holds: in the graph $\mathcal{G}$ $\exists i\pi j : A \xrightarrow{*} l(\pi)$.*

Proof. (Proof by induction)

Basis: For $k = 0$ and the statement of the lemma holds, since $\mathcal{M}_{2,(0)} = \mathcal{M}_2$, where $\mathcal{M}_2$ is adjacency matrix of the graph $\mathcal{G}$. The nonterminals, which allow to derive ε-word, are also added at algorithm preprocessing step, since each vertex of the graph is reachable by itself through an ε-transition.

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$, where $p \geq 1$.

For the algorithm iteration p the Kronecker product $\mathcal{M}_3$, M_3' and transitive closure C_3 are evaluated as described in the algorithm. By the properties of this operations, some edge $e = ((s, i), (f, j))$ exists in the directed graph, represented by adjacency matrix C_3, if and only if $\exists s\pi' f$ in the RSM graph, represented by matrix $\mathcal{M}_1$, and $\exists i\pi j$ in graph, represented by $\mathcal{M}_{2,(p-1)}$. Concatenated symbols along the path π' form some derivation string v, composed from terminals and non-terminals, where $v \xrightarrow{*} l(\pi)$ by the inductive assumption.

The new $\{1\}$ will be added to the cell $M_{2,(k)}^A[i, j]$ only if s and f are initial and final states of some box of the RSM corresponding to the non-terminal A. In this case, the grammar G has the derivation rule $A \rightarrow v$, and by the inductive assumption $v \xrightarrow{*} l(\pi)$. Therefore, $A \xrightarrow{*} l(\pi)$ and this completes the proof of the lemma.

□

Lemma 3.2. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. Let $\mathcal{M}_{2,(k)}$ be an adjacency matrix $\mathcal{M}_2$ after the execution of some iteration $k \geq 0$ of the algorithm in Listing 1. For any path $i\pi j$ in the graph $\mathcal{G}$ with word $l = l(\pi)$ if exists the derivation tree of l from the nonterminal A of the grammar G with the height $h \leq k + 1$, then $M^A_{2,(k)}[i, j]$ contains $\{1\}$.*

Proof. (Proof by induction)

Basis: Show that statement of the lemma holds for the $k = 0$. Matrix $\mathcal{M}_{2,(0)} = \mathcal{M}_2$ and edges of the graph $\mathcal{G}$ contains only labels from L. Since the derivation tree of height $h = k + 1 = 1$ contains only one non-terminal A as a root and only symbols from $\Sigma \cup \varepsilon$ as leafs, for all paths, which form a word with derivation tree of the height $h = 1$, the corresponding nonterminals will be added to the $M^A_{2,(0)}[i, j]$ via preprocessing step. Thus, the lemma statement holds for the $k = 1$.

Inductive step: Assume that the statement of the lemma hold for any $k \leq (p - 1)$ and show that it also holds for $k = p$, where $p \geq 2$.

For the algorithm iteration p the Kronecker product $\mathcal{M}_3$, M'_3 and transitive closure C_3 are evaluated as described in the algorithm. By the properties of this operations, some edge $e = ((s, i), (f, j))$ exists in the directed graph, represented by adjacency matrix C_3, if and only if $\exists s\pi' f$ in the RSM graph, represented by matrix $\mathcal{M}_1$, and $\exists i\pi j$ in graph, represented by $\mathcal{M}_{2,(p-1)}$.

For any path $i\pi j$, such that exist derivation tree of height $h < p + 1$ for the word $l(\pi)$ with root non-terminal A, the cell $M^A_{2,(p)}[i, j]$ contains $\{1\}$ by inductive assumption.

Suppose, that exists derivation tree T of height $h = p + 1$ with the root non-terminal A for the path $i\pi j$. The tree T is formed as $A \longrightarrow a_1..a_d, d \geq 1$ where $\forall x \in [1..d]$ a_x is subtree of height $h_x \leq p$ for the sub-path $i_x\pi_x j_x$. By inductive hypothesis, there exists path π_x for each derivation sub-tree, such that $i = i_1\pi_1 i_2..i_d\pi_d j_d = j$ and concatenation of these paths forms $i\pi j$, and the root nonterminals of this sub-trees are included in the matrix $M_{2,(p-1)}$.

Therefore, vertices i_x $\forall x \in [1..d]$ form path in the graph, represented by matrix $\mathcal{M}_{2,(p-1)}$, with complete set of labels. Thus, new $\{1\}$ will be added to the cell $M^A_{2,(p)}[i, j]$ corresponding to the vertices i and j and nonterminal A. This completes the proof of the lemma.

□

Theorem 3.3. *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 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 $\{1\}$, if and only if there is a path $i\pi j$ in the graph $\mathcal{G}$ such that $A \xrightarrow{*} l(\pi)$.*

Proof. This theorem is a consequence of the Lemma 3.1 and Lemma 3.2.

□

Theorem 3.4. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The algorithm in Listing 1 terminates in finite number of steps.*

Proof. The main *while-loop* in the algorithm is executed while graph adjacency matrix $\mathcal{M}_2$ is changing. Since the algorithm only adds the edges with non-terminals from N, the maximum required number of iterations is $|N| \times |V| \times |V|$, where each component has finite size. This completes the proof of the theorem.

□

3.1.3 Application of Dynamic Transitive Closure. In this subsection we show how to reduce the time complexity of the algorithm in Listing 1 by avoiding redundant calculations.

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 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. We use an approach by Ibaraki and Katoh [26] 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 (see Figure 6 for details). The algorithm is presented in Listing 3 (we have slightly modified it to efficiently track new elements of the matrix C_3).

Final version of the modified algorithm from Listing 1 is shown in Listing 4.

Theorem 3.5. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The algorithm from Listing 4 calculates a result matrices $\mathcal{M}_2$ and C_3 in $|P|^3 n^3$ time where $n = |V|$.*

Listing 3 The dynamic transitive closure procedure

function ADD(C_3, i, j)
  $n \leftarrow$ Number of rows in C_3
  $C_3' \leftarrow$ Empty matrix of size $n \times n$
  for $u \neq 0 \in$ checkCondition(C_3, i, j) **do**
    newReachablePairs(C_3, C_3', u, j)
  return C_3'
function CHECKCONDITION(C_3, i, j)
  $A \leftarrow$ Empty array of size n
  for $u \in 0 \ldots n \mid u \neq j$ **do** ▷ $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0; 0 \wedge 1 = 1$
    $A[u] = C_3[u, j] \wedge C_3[u, i]$
  return A
function NEWREACHABLEPAIRS(C_3, C_3', u, j)
  $C_3'[u, v] = C_3[u, v] \wedge C_3[j, v]$ ▷ $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0; 0 \wedge 1 = 1$

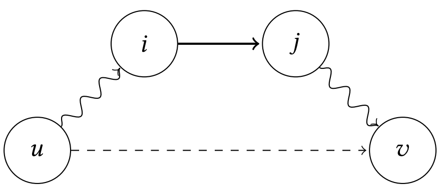


Figure 6: The vertex j become reachable from the vertex u after the addition of edge (i, j). Then the vertex v is reachable from u after inserting the edge (i, j) if v is reachable from j.

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 we have $\sum_i T(\mathcal{M}_1 \otimes \mathcal{A}_2^{(i)}) = |\mathcal{M}_1| \sum_i |\mathcal{A}_2^{(i)}| = (|N| + |\Sigma|)|P|^2 \sum_i |\mathcal{A}_2^{(i)}| \leq (|N| + |\Sigma|)^2 |P|^2 n^2$ operations in total.

Now we derive the time complexity of maintainig the dynamic transitive closure. Notice that C_3 has size of the Kronecker product of $\mathcal{M}_1 \otimes \mathcal{M}_1$, which is equal to $|R|n \times |R|n = |P|n \times |P|n$ so no more than $|P|^2 n^2$ edges will be added during all iterations of the Algorithm. The function *checkCondition* from the Listing 3 takes $O(|P|n)$ time for every inserted edge (i, j). Thus we have $O(|P|^2 n^2 |P|n) = O(|P|^3 n^3)$ operations in total. The function *newReachablePairs* requires $O(|P|n)$ time for a given vertex u. This operation is performed for every pair (j, v) of vertices such that a vertex j became reachable from the vertex u. The vertex j become reachable from the vertex u (and accordingly the value of the matrix cell $C_3[u, j]$ becomes 1 from 0) only once during the entire computation, so the function *newReachablePairs* will be executed at most $O(|P|^2 n^2)$ times for every u and hence $O(|P|^3 n^3)$ times in total for all vertices. Therefore $O(|P|^3 n^3)$ operations are

Listing 4 Kronecker product based CFPQ using dynamic transitive closure

1: **function** CONTEXTFREEPATHQUERYING(G, $\mathcal{G}$)
2:   $n \leftarrow$ The number of nodes in $\mathcal{G}$
3:   $R \leftarrow$ Recursive automata for G
4:   $\mathcal{M}_1 \leftarrow$ Boolean adjacency matrix for R
5:   $\mathcal{M}_2$, $\mathcal{A}_2 \leftarrow$ Boolean adjacency matrix for $\mathcal{G}$
6:   $C_3 \leftarrow$ The empty matrix of size $|R|n \times |R|n$
7:   **for** $s \in 0..dim(\mathcal{M}_1) - 1$ **do**
8:     **for** $S \in$ *getNonterminals*(R, s, s) **do**
9:       **for** $i \in 0..dim(\mathcal{M}_2) - 1$ **do**
10:         $M_2^S[i, i] \leftarrow 1$
11:   **while** $\mathcal{M}_2$ is changing **do**
12:     $M_3' \leftarrow \bigvee_{M^S \in \mathcal{M}_1 \otimes \mathcal{A}_2} M^S$
13:     $\mathcal{A}_2 \leftarrow$ The empty matrix
14:     $C_3' \leftarrow$ The empty matrix of size $|R|n \times |R|n$
15:     **for** $(i, j) \mid M_3'[i, j] \neq 0$ **do**
16:       $C_3' \leftarrow$ *add*(C_3, C_3', i, j) ▷ Updating the transitive closure
17:       $C_3 \leftarrow C_3 + C_3'$
18:     **for** $(i, j) \mid C_3'[i, j] \neq 0$ **do**
19:       $s, f \leftarrow$ *getStates*(C_3', i, j)
20:       $x, y \leftarrow$ *getCoordinates*(C_3', i, j)
21:       **for** $S \in$ *getNonterminals*(R, s, f) **do**
22:         $M_2^S[x, y] \leftarrow 1$
23:         $A_2^S[x, y] \leftarrow 1$
24:   **return** $\mathcal{M}_2$, C_3

performed to maintain dynamic transitive closure during all iteration of the algorithm from Listing 4.

Notice that the matrix C_3' contains only new elements, therefore C_3 can be updated directly using only $|C_3'|$ operations and hence $O(|P|^2 n^2)$ operations in total. The same holds for cycle in line 18 of the algorithm from Listing 4, because operations are performed only for non-zero elements of the matrix $|C_3'|$. Finally, we have that the time complexity of the algorithm is $(|N| + |\Sigma|)^2 |P|^2 n^2 + O(|P|^3 n^3) + O(|P|^2 n^2) + O(|P|^2 n^2) = O(|P|^3 n^3)$. □

3.1.4 *Speeding up by a factor of* $\log n$.

In this subsection we use the Four Russians' trick to speed up the dynamic transitive closure algorithm from the Listing 3.

THEOREM 3.6. *The computation of transitive closure matrices can be done in $O(n^3 / \log n)$ time when n^2 edges are added to the graph.*

PROOF. Consider the function *checkCondition* from the Listing 3. Its operations are equivalent to the element-wise (Hadamard) product of two vectors of size n, where multiplication operation is denoted as $\wedge$ and has the following properties: $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$ and $0 \wedge 1 = 1$. 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 function *newReachablePairs* also can be reduced to the computation of the Hadamard product of two vectors of size n for a 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 $O(n)$ which gives the $O(n^3)$ time for maintaining the transitive closure. 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 this goal, we use the Four Russians' trick. Split each vector into $n/\log n$ parts of size $\log n$. Create a table S such that $S(a,b) = a \wedge b$ where $a, b \in \{0,1\}^{\log n}$. This takes a 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. This gives us a dynamic transitive closure algorithm running in time $O(n^3/\log n)$: both of the functions *checkCondition* and *newReachablePairs* are evaluated no more than $O(n^2)$ times during the whole computation, and each function calculates Hadamard product of two vectors in $O(n/\log n)$ time. □

Notice that the maintaining of the dynamic transitive closure dominates the cost of the algorithm from Listing 4, therefore we immediately deduce the following.

Corollary 3.7. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The resulting matrices $\mathcal{M}_2$ and C_3 can be calculated in $O(|P|^3 n^3/(\log n + \log |P|))$ time.*

Finally, we formulate the theorem which connects the time complexity of CFPQ and time complexity of specific incremental transitive closure of a directed graph.

Theorem 3.8. *Subcubic incremental transitive closure leads to subcubic CFPQ. Suppose the incremental transitive closure problem where only insertion queries are allowed and the result of each insertion is a set of newly connected pairs. If one can solve this problem in $O(n^{3-\varepsilon})$ total time for n^2 insertions, then one can solve CFPQ in $O(n^{3-\varepsilon})$, where n is a number of vertices in the graph in both cases.*

Theorem 3.8 shows that the maintaining of the incremental transitive closure dominates the cost of the algorithm. Thus, CFPQ can be solved in truly subcubic $O(n^{3-\varepsilon})$ time if there is an incremental dynamic algorithm for the transitive closure for a graph with n vertices with preprocessing time $O(n^{3-\varepsilon})$ and total update time $O(n^{3-\varepsilon})$. Unfortunately, such an algorithm is unlikely to exist: it was proven that there is no incremental dynamic transitive closure algorithm for a graph with n vertices and at most m edges with preprocessing time $poly(m)$, total update time $mn^{1-\varepsilon}$, and query time $m^{\delta-\varepsilon}$ for any $\delta \in (0, 1/2]$ per query that has an error probability of at most 1/3 assuming the widely believed Online Boolean Matrix-Vector Multiplication (OMv) Conjecture [24]. OMv Conjecture introduced by Henzinger et al. [24] states that for any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$-time algorithm that solves OMv with an error probability of at most 1/3.

3.2 Paths Extraction Algoritm

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 5.

Listing 5 Paths extraction algorithm

$C_3 \leftarrow$ result of index creation algorithm: final transitive closure
$\mathcal{M}_1 \leftarrow$ the set of adjacency matrices of the input RSM
$\mathcal{M}_2 \leftarrow$ the set of adjacency matrices of the final graph
function GETPATHS(v_s, v_f, N)
    $q_N^0 \leftarrow$ Start state of automata for N
    $F_N \leftarrow$ Final states of automata for N
    $res \leftarrow \bigcup_{f \in F_N}$ GETPATHSINNER($(q_N, v_s), (f, v_f)$)
    return res
function GETSUBPATHS($(s_i, v_i), (s_j, v_j), (s_k, v_k)$)
    $l \leftarrow \{(v_i, t, v_k) \mid M_2^t[s_i, s_k] \wedge M_1^t[v_i, v_k]\}$
        $\cup \bigcup_{\{N \mid M_2^N[s_i, s_k]\}}$ GETPATHS(v_i, v_k, N)
        $\cup$ GETPATHSINNER($(s_i, v_i), (s_k, v_k)$)
    $r \leftarrow \{(v_k, t, v_j) \mid M_2^t[s_k, s_j] \wedge M_1^t[v_k, v_j]\}$
        $\cup \bigcup_{\{N \mid M_2^N[s_k, s_j]\}}$ GETPATHS(v_k, v_j, N)
        $\cup$ GETPATHSINNER($(s_k, v_k), (s_j, v_j)$)
    return $l \cdot r$
function GETPATHSINNER($(s_i, v_i), (s_j, v_j)$)
    $parts \leftarrow \{(s_k, v_k) \mid C_3[(s_i, v_i), (s_k, v_k)] = 1 \wedge C_3[(s_k, v_k), (s_j, v_j)] = 1\}$
    return $\bigcup_{(s_k, v_k) \in parts}$ GETSUBPATHS($(s_i, v_i), (s_j, v_j), (s_k, v_k)$)

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.

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.

3.3 An example

In this section we introduce detailed example to demonstrate steps of the proposed algorithms. Namely, let we have a graph $\mathcal{G}$ presented in Figure 1 and the RSM R presented in Figure 5.

First step we represent graph as a set of Boolean matrices as shown in Figure 2, and RSM as a set of Boolean matrices as presented in Figure 4. Notice that we should formally add new empty matrix M_2^S to $\mathcal{M}_2$, where edges labeled by S will be added in time of the computation.

After the initialization, the algorithm handles ε-case. The input RSM does not have ε-transitions and does not have states that are both start and final, therefore, no edges added at this stage. After that we should iteratively compute $\mathcal{M}_2$ and C_3. The loop iteration number of matrices evaluation is provided as the subscript in parentheses.

First iteration. Firstly, we compute Kronecker product of the $\mathcal{M}_1$ and $\mathcal{M}_{2,(0)}$ matrices and store result in the $\mathcal{M}_{3,(1)}$, and collapse this matrix to the single Boolean matrix $M'_{3,(1)}$. For the sake of simplicity, we provide only $M'_{3,(1)}$, which is evaluated as follows in the equivalent way.

$$M'_{3,(1)} = M_1^a \otimes M_{2,(0)}^a + M_1^b \otimes M_{2,(0)}^b + M_1^S \otimes M_{2,(0)}^S =$$

$$\begin{array}{rl} & \begin{array}{cccccccc}(0,0)&(0,1)&(1,0)&(1,1)&(2,0)&(2,1)&(3,0)&(3,1)\end{array} \\ \begin{array}{r}(0,0)\\(0,1)\\(1,0)\\(1,1)\\(2,0)\\(2,1)\\(3,0)\\(3,1)\end{array} & \left(\begin{array}{cc|cc|cc|cc} . & . & . & 1 & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ \hline . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ \hline . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{array}\right)\end{array}$$

As far as the input graph has no edges with label S, therefore, the correspondent block of the Kronecker product will be empty. The Kronecker product graph of the input graph $\mathcal{G}$ and RSM R is shown in Figure 7. Then, the transitive closure evaluation result, stored in the matrix $C_{3,(1)}$, introduces one new path of length 2 (the thick path in Figure 7).

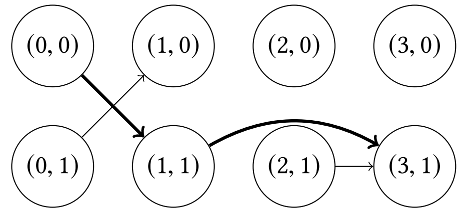


Figure 7: The Kronecker product graph of RSM R and the input graph $\mathcal{G}$

This path starts in the vertex $(0, 0)$ and finishes in the vertex $(3, 1)$. We can see, that 0 and 3 are a start and a final states of the some component state machine for label S in R respectively. Thus we can conclude that there exists a path between vertices 0 and 1 in the graph, such that respective word is derivable from S in the R execution flow.

As a result, we can add the edge $(0, S, 1)$ to the result graph, what is formally done by the update of the matrix M_2^S.

Second iteration. Modified graph Boolean adjacency matrices contain now edge with label S. Therefore, this label contributes to the non-empty corresponding matrix block in the evaluated matrix $M'_{3,2}$. The transitive closure evaluation introduces three new paths $(0, 1) \to (2, 1), (1, 0) \to (3, 1)$ and $(0, 1) \to (3, 1)$ (see Figure 8). Since only path between vertices $(0, 1)$ and $(3, 1)$ connects start and final states in the automata, and the edge $(1, S, 1)$ is added to the result graph.

$$M'_{3,(2)} = M_1^a \otimes M_{2,(2)}^a + M_1^b \otimes M_{2,(2)}^b + M_1^S \otimes M_{2,(2)}^S =$$

$$\begin{array}{rl} & \begin{array}{cccccccc}(0,0)&(0,1)&(1,0)&(1,1)&(2,0)&(2,1)&(3,0)&(3,1)\end{array} \\ \begin{array}{r}(0,0)\\(0,1)\\(1,0)\\(1,1)\\(2,0)\\(2,1)\\(3,0)\\(3,1)\end{array} & \left(\begin{array}{cc|cc|cc|cc} . & . & . & 1 & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ \hline . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & 1 \\ \hline . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ \hline . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{array}\right)\end{array}$$

The result graph is presented in Figure 9.

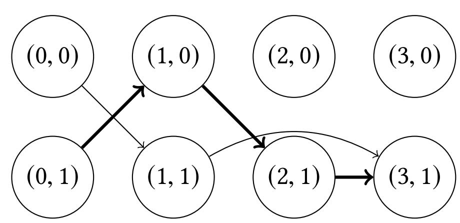


Figure 8: The Kronecker product graph of RSM R and the updated graph $\mathcal{G}$

At this point the index creation is finished. One can use it to answer reachability queries, but for some problems it can be used to restore paths for some reachable vertices. The result transitive closure matrix C_3 or so called *index* could be

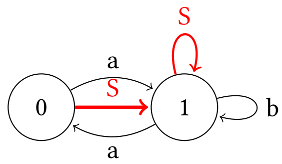


Figure 9: The result graph $\mathcal{G}$

`getPaths`$(1, 1, S)$
└ `getPathsInner`$((0,1),(3,1))$
  └ `parts=` $\{(1,0),(2,1)\}$
  └ `getSubpaths`$((0,1),(3,1),(1,0))$
    └ `l=`$\{1 \xrightarrow{a} 0\}$
      └ ...
        └ `getPathsInner`$((0,0),(3,1))$
          └ `parts =` $\{(1,1)\}$
            └ ..
              └ `getPaths`$(0, 1, S)$
                └ ..
                  └ `getSubpaths`$((0,0),(3,1),(1,1))$
                    └ $l = \{0 \xrightarrow{a} 1\}$
                    └ $r = \{1 \xrightarrow{b} 1\}$
                    └ `return` $\{0 \xrightarrow{a} 1 \xrightarrow{b} 1\}$
  └ `getSubpaths`$((0,1),(3,1),(2,1))$
    └ ...
      └ `getPaths(0, 1, S) // An alternative way to get paths from 0 to 1 which leads to infinite set of paths`
          └ `return` $r_{\infty}^{0 \rightsquigarrow 1}$ `// An infinite set of path from 0 to 1`
        └ ...
└ `return` $\{1 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{b} 1\} \cup (\{1 \xrightarrow{a} 0\} \cdot r_{\infty}^{0 \rightsquigarrow 1} \cdot \{1 \xrightarrow{b} 1\})$

Figure 10: Example of call stack trace

used for that. For example, let we try to restore paths from 2 to 2 derived from S in the result graph.

To get these paths we should call `getPaths(1, 1, s)` function. Partial trace of this call is presented below in Figure 10. First, we must query paths for all possible start and final states of the machine for the provided graph vertices. Since in the example RSM the component state machine with label S has single final state, the function `getPathsInner` is called with arguments $(0, 1)$ and $(3, 1)$. Note, that in the path extraction algorithm passed values to the functions is pairs of the machine state and graph vertex, which uniquely identify cell of the index matrix C_3. Possible paths concatenation vertices are stored as `parts={(1,0),(2,1)}`. Then we try to get parts of paths going through index vertex $(1, 0)$. All possible concatenations variants of the paths are queried in the corresponding `getSubpaths` function call. As the result, we get the set of possible paths in the graph from 1 to 1.

Lazy evaluation is required here, since the result graph may possibly have an infinite number of path between some vertices pair. Another approach here is to try to query some fixed number of paths, or just single path. Eventually, the paths enumeration problem is actual here: how can we enumerate paths with small delay.

4 IMPLEMENTATION DETAILS

In order to evaluate the proposed algorithm, we implement its naïve version: transitive closure computes on each iteration from scratch, without incremental techniques utilization. For implementation we use PyGraphBLAS[2] — a Python wrapper for SuiteSparse library [14][3]. SuiteSparse is a C implementation of GraphBLAS [28] standard which introduces linear algebra building blocks for graph analysis algorithms implementation. Thus we provide a highly-optimized parallel CPU implementation of the naïve version of the proposed algorithm[4].

In the current version we do not provide integration with graph database and graph query language, because our goal is the algorithm applicability evaluation. So, we suppose that graph is stored in file, and query is expressed in terms of context-free grammar and stored in file too. As it was shown in [47] it is possible to integrate SuiteSparse based implementation in the RedisGraph database. To provide integration with query language, it is necessary to extend the language first. It is possible, for example one can use existing proposal[5] to extend Cypher language, but it requires a lot of technical effort, so it is an interesting challenge for future research to provide full-stack support for CFPQ.

Paths extraction also is implemented in Python by using PyGraphBLAS. For evaluation we implement a version which has an additional parameter: a maximal number of paths to extract. This modification allows as to avoid lazy evaluation which is not natural for Python. Note that one can provide other modifications of paths extraction algorithm based on the proposed idea.

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.

[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.

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

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

(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.

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] [18] 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 [40] (Table 2) and [51] (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 reachability index creation average time of 5 runs is presented.

Reachability index creation time for each query for LUBM graphs set is presented in figure 11. We can observe linear !!!! dependency of evaluation time on graph size. Also we can see, that query evaluation time depends on query: there are queries which evaluate less 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 [51], while we use only one node. Note, that accurate comparison of different approaches is a huge interesting work for the future.

Reachability index creation time for each query for for real-world graphs is presented in figure 12. 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,

[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.

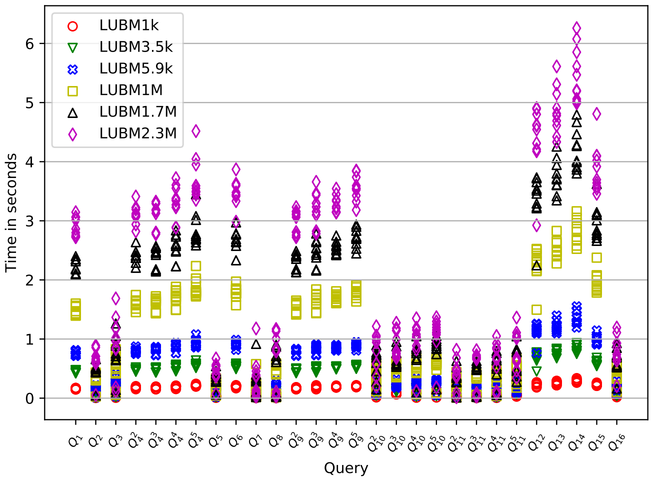


Figure 11: Reachability index creation time for LUBM graphs

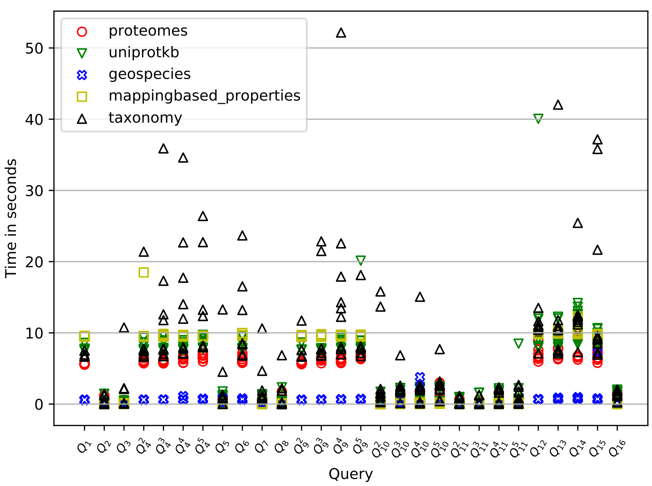
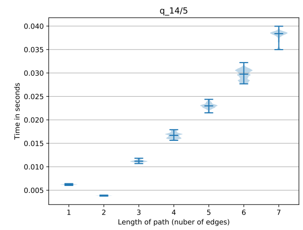


Figure 12: Reachability index creation time for real-world RDFs

taxonomy querying in relatively big number of cases requires significantly more time, than querying of other graphs, while *taxonomy* is not a biggest 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 reachability 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, reachability index creation time is not included because exactly the same index, as calculated at the previous step, is used for paths extraction.



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

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

Figure 13: Single path extraction

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
Aliases 1	6 442 630	24 465 430
Aliases 2	4 834 262	12 366 973
....	5 728 398	14 922 125

Table 3: Graphs for CFPQ evaluation

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.

5.2 CFPQ Evaluation

Comparison with matrix-based algorithm.

5.2.1 Dataset. Dataset for evaluation. It should be CFPQ_Data[11]

Same-generation queries, memory aliases.

5.2.2 Results. Results of evaluation.

Index creation.

[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.

Table 4: RDFs query G_1 and G_2 (time is measured in seconds and memory is measured in megabytes)

Name	G_1		G_2	
	Tensors	RG_CPU$_{path}$	Tensors	RG_CPU$_{path}$
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	...

Paths extraction.

5.2.3 Conclusion.

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. [5]. 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 Nolé and Sartiani which is implemented on the top of Pregel-based system [38], or solution of Koschmieder et al. [29]. But only a limited number of practical solutions provide the ability to restore paths of interest. A recent work of Xin Wang et al. [51] provides a Pregel-based provenance-aware RPQ algorithm which utilizes a Glushkov's construction [16]. 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 Hellings [21] and Bradford [7], (G)LR and (G)LL in works of Verbitskaia et al. [50], Grigorev et al. [17], Santos et al. [43], Medeiros et al. [34]. Unfortunately, none of them has better than cubic time complexity in terms of the input graph size. The algorithm of Azimov [3] is, best to our knowledge, the first algorithm for CFPQ based on linear algebra. It was shown by Terekhov et al. [47] that this algorithm can be applied for real-world graph analysis problems, while Kuijpers et al. shows in [31] that other state-of-the-art CFPQ algorithms are not performant enough to handle real-world graphs.

It is important 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 Terekhov et al. [47] provide the first CFPQ algorithm with single path semantics based on linear algebra, Hellings in [23] 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 Yannakakis in 1990 in [52]. A bit later Reps proposed the CFL reachability as a framework for interprocedural static code analysis [42]. Melski and Reps gave a dynamic programming formulation of the problem which runs in $O(n^3)$ time [35]. The problem of the cubic bottleneck of context-free language reachability is also discussed by Heintze and McAllester [20], and Melski and Reps [35]. The slightly subcubic algorithm with $O(n^3/\log n)$ time complexity was provided by Chaudhuri in [12]. This result is inspired by recursive state machine reachability. The first truly subcubic algorithm with $\widetilde{O}(n^\omega)$ time complexity (ω is the best exponent for matrix multiplication, $\widetilde{O}$ is the asymptotic upper-bound mod polylog factors) for general graph and 1-Dyck language was provided by Bradford in [**?**]. Unfortunately, this result cannot be generalized to general context-free queries A similar result was provided by Pavlogiannis and Mathiasen in [41]. Another partial case was investigated by Chatterjee et al. in [10]. The $O(m + n \cdot \alpha(n))$ algorithm for an arbitrary Dyck querying of a bidirected graph was described. Here m is a number of edges, n is a number of vertices in the input graph, and $\alpha(x)$ is an inverse Ackermann function. Specific types of static code analysis related to CFL-r, especially Andersen's Pointer Analysis also actively studying. For example, recently BMM-hardness of 1-Dyck reachability was proven by Zhang in [53]. Other partial cases such as tree querying also were studied.

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 [28] 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 [14] or CombBLAS [8]. 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 [27, 36] and for Datalog queries evaluation [44]. Finally, RedisGraph [9], 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 grammar to be in CNF transformation 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.

The first important task for future research is a detailed investigation of the paths extraction algorithm. Hellings in [23] provides a theoretical investigation of single path extraction and shows that the problem is related to formal language theory. All paths extraction is more complicated and should be investigated carefully in order to provide an optimal algorithm.

Also, the algorithm opens a way to attack the long-standing problem on 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.* In this work we prove $O(n^3/\log n)$ time complexity by providing $O(n^3/\log n)$ incremental transitive closure 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 dynamic graph problems. An algorithm for incremental dynamic transitive closure 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 and $m^{\delta-\varepsilon}$ query time per query for any $\delta \in (0, 1/2]$ will refute the online Boolean Matrix-Vector Multiplication (OMv) Conjecture, which is used to prove conditional lower bounds for many dynamic problems [24, 48].

Thus, the first task for the future is to improve the logarithmic factor in the obtained bound. Also, it is interesting to get improved bounds in partial cases. For example, fully dynamic transitive closure for planar graphs can be supported in $O(n^{2/3} \log n)$ time per update [46], and for undirected graph one can use *disjoint sets* which provide operations with time complexity bounded by inverse Ackermann function. Can we use these facts to provide a better CFPQ algorithm for respective partial cases? 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.

On the other hand, provided reduction open a way to investigate streaming graph querying. This way we can formulate the following questions.

(1) Can we provide a more detailed analysis of dynamic CFPQ queries than provided in [6]?
(2) Can we provide a practical solution for CFPQ querying of streaming graphs?
(3) Can we improve existing solutions for RPQ of streaming graphs?

From a practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work, we evaluate naïve implementation in which transitive closure recalculated on each iteration from scratch. In [19] it is shown that some of the 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. Experience in Azimov's algorithm shows that the utilization of GPGPUs allows one to improve performance because operations of linear algebra can be efficiently implemented on GPGPU [37, 47]. 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 [13, 15].

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 [32]. 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 [45].

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph[12] [9] 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 Terkhov et.al. to integrate Azimov's algorithm to RedisGraph with minimal effort [47].

REFERENCES

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2001. Analysis of Recursive State Machines. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031

[3] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264

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

[5] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (May 2000), 809–837. https://doi.org/10.1137/S0097539798337716

[6] Patricia Bouyer and Vincent Jugé. 2017. Dynamic Complexity of the Dyck Reachability. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*. Springer-Verlag, Berlin, Heidelberg, 265–280. https://doi.org/10.1007/978-3-662-54458-7_16

[7] P. G. Bradford. 2017. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*. IEEE, 247–253. https://doi.org/10.1109/UEMCON.2017.8249039

[8] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[9] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine. 2019. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 285–286. https://doi.org/10.1109/IPDPSW.2019.00054

[10] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158118

[11] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1328438.1328460

[12] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. *SIGPLAN Not.* 43, 1 (Jan. 2008), 159–169. https://doi.org/10.1145/1328897.1328460

[13] Steven Wei Der Chien, Ivy Bo Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing, MCHPC@SC 2019, Denver, CO, USA, November 18, 2019*. IEEE, 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014

[14] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[15] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

[16] V M Glushkov. 1961. THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys* 16, 5 (Oct. 1961), 1–53. https://doi.org/10.1070/rm1961v016n05abeh004112

[17] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[18] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semant.* 3, 2–3 (Oct. 2005), 158–182. https://doi.org/10.1016/j.websem.2005.06.005

[19] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2020. Faster Fully Dynamic Transitive Closure in Practice. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy (LIPIcs)*, Simone Faro and Domenico Cantone (Eds.), Vol. 160. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:14. https://doi.org/10.4230/LIPIcs.SEA.2020.14

[20] Nevin Heintze and David McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, USA, 342.

[21] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.

[22] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[23] Jelle Hellings. 2020. Explaining Results of Path Queries on Graphs: Single-Path Results for Context-Free Path Queries. (2020). To appear.

[24] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing (STOC '15)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2746539.2746609

[25] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[26] T. Ibaraki and N. Katoh. 1983. On-line computation of transitive closures of graphs. *Inform. Process. Lett.* 16, 2 (1983), 95 – 97. https://doi.org/10.1016/0020-0190(83)90033-9

[27] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proceedings of the Fourteenth EuroSys*

[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.

Conference 2019 (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 27, 15 pages. https://doi.org/10.1145/3302424.3303962

[28] J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[29] André Koschmieder and Ulf Leser. 2012. Regular Path Queries on Large Graphs. In *Scientific and Statistical Database Management*, Anastasia Ailamaki and Shawn Bowers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–194.

[30] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015*, Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.). Springer International Publishing, Cham, 3–18.

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

[32] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Article 19, 15 pages.

[33] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages. https://doi.org/10.1145/2850413

[34] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2018. Efficient Evaluation of Context-free Path Queries for Graph Databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1230–1237. https://doi.org/10.1145/3167132.3167265

[35] David Melski and Thomas Reps. 1997. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 74–89.

[36] Saskia Metzler and Pauli Miettinen. 2015. On Defining SPARQL with Boolean Tensor Algebra. *CoRR* abs/1503.00301 (2015). arXiv:1503.00301 http://arxiv.org/abs/1503.00301

[37] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. https://doi.org/10.1145/3327964.3328503

[38] Maurizio Nolé and Carlo Sartiani. 2016. Regular Path Queries on Massive Graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2949689.2949711

[39] Egor Orachev, Ilya Epelbaum, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying by Kronecker Product. In *Advances in Databases and Information Systems*, Jérôme Darmont, Boris Novikov, and Robert Wrembel (Eds.). Springer International Publishing, Cham, 49–59.

[40] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1415–1430. https://doi.org/10.1145/3318464.3389733

[41] Andreas Pavlogiannis and Anders Alnor Mathiasen. 2020. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. arXiv:cs.PL/2006.01491

[42] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS '97)*. MIT Press, Cambridge, MA, USA, 5–19.

[43] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.

[44] TAISUKE SATO. 2017. A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming* 17, 3 (2017), 244–265. https://doi.org/10.1017/S1471068417000023

[45] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/2463676.2467799

[46] Sairam Subramanian. 1993. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms—ESA '93*, Thomas Lengauer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 372–383.

[47] Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3398682.3399163

[48] J. van den Brand, D. Nanongkai, and T. Saranurak. 2019. Dynamic Matrix Inverse: Improved Algorithms and Matching Conditional Lower Bounds. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. 456–480.

[49] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[50] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2016. Relaxed Parsing of Regular Approximations of String-Embedded Languages. In *Perspectives of System Informatics*, Manuel Mazzara and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 291–302.

[51] Xin Wang, Simiao Wang, Yueqi Xin, Yajun Yang, Jianxin Li, and Xiaofei Wang. 2019. Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *World Wide Web* 23, 3 (Nov. 2019), 1465–1496. https://doi.org/10.1007/s11280-019-00739-0

[52] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576

[53] Qirun Zhang. 2020. Conditional Lower Bound for Inclusion-Based Points-to Analysis. arXiv:cs.PL/2007.05569