Noname manuscript No.
(will be inserted by the editor)

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

Ekaterina Shemetova · Rustam Azimov · Egor Orachev · Ilya Epelbaum · Semyon Grigorev

Received: date / Accepted: date

Abstract The Kronecker product-based algorithm for context-free path querying (CFPQ) was proposed by Orachev et al. (2020). We reduce this algorithm to operations over Boolean matrices and extend it with the mechanism to extract all paths of interest. We also 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 the alternative way to construct a slightly subcubic algorithm for CFPQ which is based on linear algebra and incremental transitive closure (a classic graph-theoretic problem), as opposed to the algorithm with the same

The research was supported by the Russian Science Foundation, grant number: 18-11-00100

E. Shemetova
Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia
Saint Petersburg Academic University, 8/3 Khlopin St., St. Petersburg, Russia
JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia
E-mail: katyacyfra@gmail.com

R. Azimov
Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia
JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia
E-mail: rustam.azimov19021995@gmail.com

E. Orachev
Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia
JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia
E-mail: egor.orachev@gmail.com

I. Epelbaum
Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia
JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia
E-mail: iliyepelbaun@gmail.com

S. Grigorev
Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia
JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia
E-mail: s.v.grigoriev@spbu.ru, semyon.grigorev@jetbrains.com

complexity proposed by Chaudhuri (2008). Our evaluation shows that our algorithm is a good candidate to be the universal algorithm for both regular and context-free path querying.

Keywords Graph databases · Regular path queries · Context-free path queries · CFL-reachability · Recursive state machines · Dynamic transitive closure

Acknowledgements We are grateful to Ekaterina Verbitskaia for careful reading and pointing out some mistakes.

1 Introduction

Language-constrained path querying (Barrett et al., 2000) is a technique for graph navigation querying. This technique allows one to use formal languages as constraints on paths in edge-labeled graphs: a path satisfies constraints if the labels along it form a word from the specified language.

The utilization of regular languages as constraints, or *Regular Path Querying* (RPQ), is the most well-studied and widespread. Different aspects of RPQs are actively studied in graph databases (Barceló Baeza, 2013; Angles et al., 2017; Libkin et al., 2016), while regular constraints are supported in such popular query languages as PGQL (van Rest et al., 2016) and SPARQL[1] (Kostylev et al., 2015) (known as property paths). Nevertheless, there is certainly room for improvement of RPQ efficiency, and new solutions are being created (Wang et al., 2019; Nolé and Sartiani, 2016).

At the same time, using more powerful languages as constraints, namely context-free languages, has gained popularity in the recent years. *Context-Free Path Querying* problem (CFPQ) was introduced by Yannakakis (1990), and nowadays is used in many areas. For example, CFPQ is used for interprocedural static code analysis (Chatterjee et al., 2017; Reps, 1997; Yan et al., 2011; Zheng and Rugina, 2008) In this area CFPQ is known as the context-free language reachability (*the CFL-reachability*) problem. Also, CFPQ can be used for biological data analysis (Sevon and Eronen, 2008), graph segmentation in data provenance analysis (Miao and Deshpande, 2019), and for data flow information preserving in machine learning based solutions for code analysis problems (Sui et al., 2020).

Many algorithms for CPFQ were proposed, but recently Kuijpers et al. (2019) showed that the state-of-the-art CFPQ algorithms are still not performant enough for practical use. This motivates further research of the new algorithms for CFPQ.

One promising way to achieve high-performance solutions for graph analysis problems is to reduce them to linear algebra operations. To facilitate this approach, the description of basic linear algebra primitives GraphBLAS API (Kepner et al., 2016) was proposed. Evaluation of the libraries that implement this

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

API, such as SuiteSparce (Davis, 2019) and CombBLAS (Buluç and Gilbert, 2011), show that reduction to linear algebra is a good way to utilize high-performance parallel and distributed computations for graph analysis.

Azimov and Grigorev (2018) showed how to reduce CFPQ to matrix multiplication. Later, it was shown by Mishin et al. (2019) and Terekhov et al. (2020) that by using the appropriate libraries for linear algebra for Azimov's algorithm implementation one can create a practical solution for CFPQ. However Azimov's algorithm requires transforming of the input grammar to Chomsky Normal Form. This leads to the grammar size increase and hence worsens performance, especially for regular queries and complex context-free queries.

To solve these problems, an algorithm based on automata intersection was proposed (Orachev et al., 2020). This algorithm is based on linear algebra and does not require the transformation of the input grammar. In this work we improve this algorithm by reducing it to operations over Boolean matrices, thus simplifying its description and implementation. Additionally, we added the support of all-paths query semantics. Under the *all-path query semantics*, a query is evaluated to all paths satisfying the conditions of the query. All-paths semantics is necessary, for example, in biological data analysis (Sevon and Eronen, 2008), where paths prove why the specified vertices are of interest (similar). In static code analysis (e.g. alias analysis) paths indicate the reason why two names are aliases which can be used to generate a good error message to the user of the static analysis tool. Reporting all such reasons (all paths) makes for a shorter feedback loop as well as provides a more detailed analysis.

We also show that this algorithm is performant enough for regular queries, so it is a good candidate for integration with the real-world query languages: one algorithm can be used to evaluate both regular and context-free queries. Having a unified environment simplifies the development of the querying tools by allowing for reuse of common optimizations for the querying algorithm. Note that a real-world context-free query is likely to have a regular subquery which can be significant in size. Our algorithm is capable to treat such regular subparts as a regular query thus imposing little overhead as compared to treating them as a generic context-free query. This makes a unified solution more promising in terms of performance.

Moreover, we show that this algorithm opens the way to tackle a long-standing problem about the existence of truly-subcubic $O(n^{3-\epsilon})$ CFPQ algorithm (Chaudhuri, 2008; Yannakakis, 1990). Currently, the best result is an $O(n^3/\log n)$ algorithm of Chaudhuri (2008). Also, there exist truly subcubic solutions which use fast matrix multiplication for some fixed subclasses of context-free languages (Bradford, 2017). Unfortunately, these solutions cannot be generalized to arbitrary CFPQs. In this work we identify incremental transitive closure as a bottleneck on the way to achieve subcubic time complexity for CFPQ.

To sum up, we make the following contributions.

1. We rethink and improve the CFPQ algorithm based on tensor-product proposed by Orachev et al. (2020). We reduce this algorithm to operations

over Boolean matrices. As a result, all-path query semantics is handled, as opposed to the previous matrix-based solution capable of handling only the single-path semantics. Best to our knowledge, our algorithm is the first CFPQ algorithm based on linear algebra which is capable to handle all-path query semantics. Also, both regular and context-free grammars can be used as queries.
2. We prove the correctness and time complexity for the proposed algorithm thus providing an upper bound on the complexity of the CFPQ problem in dependence on the size of the query (its context-free grammar) and the number of vertices in the input graph. The proposed algorithm has subcubic complexity in terms of the grammar and the input graph sizes, which is comparable with the state-of-the-art solutions. On the other hand, the algorithm does not require transforming the input grammar to Chomsky Normal Form. The transformation leads to at least quadratic blow-up in grammar size, thus by avoiding the transformation, our algorithm achieves better time complexity than other solutions in terms of the grammar size.
3. We demonstrate the interconnection between CFPQ and incremental transitive closure. We show that incremental transitive closure is a bottleneck on the way to achieve faster CFPQ algorithm for general case of arbitrary graphs as well as for special families of graphs, such as planar graphs.
4. We implement the described algorithm and evaluate it on real-world data for both RPQ and CFPQ. The evaluation shows that the proposed algorithm is comparable with the existing solutions for CFPQ and RPQ, thus the algorithm provides a promising way to handle both CFPQ and RPQ.

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* (Barrett et al., 2000) over directed edge-labeled graphs we first give both language and grammar definitions.

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

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

Definition 2 An *adjacency matrix* for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L \rangle$ is a matrix M, where:

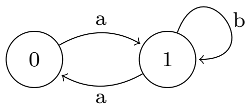


Fig. 1: The example of input graph $\mathcal{G}$

- 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

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

Definition 3 The *Boolean matrices decomposition*, 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.

$$\mathcal{M}_2 = \left\{ M_2^a = \begin{pmatrix} . & 1 \\ 1 & . \end{pmatrix}, M_2^b = \begin{pmatrix} . & . \\ . & 1 \end{pmatrix} \right\}.$$

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 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 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 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 language-constraint path querying 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 7 To evaluate language-constraint 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 8 To evaluate language-constraint 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 build a finite representation of 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 9 A *regular expression* over the alphabet Σ is a finite combination of patterns, which can be defined as follows: $\varnothing$ (empty language), ε (empty string), $a_i \in \Sigma$ are regular expressions, and if R_1 and R_2 are regular expressions, then $R_1 \mid R_2$ (alternation), $R_1 \cdot R_2$ (concatenation), R_1^* (Kleene star) are also regular expressions.

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

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

Definition 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 (Hopcroft et al., 2006). We use FSM as a representation of RPQ. FSM $T = \langle \Sigma, Q, Q_s, Q_f, \delta \rangle$ can be naturally represented by a directed edge-labeled graph $\mathcal{G} = \langle V, E, L \rangle$, where $V = Q$, $L = \Sigma$,

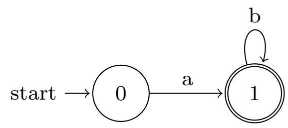


Fig. 2: The example of graph representation of FSM for regular expression ab^*

$E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$ and some vertices are marked as the start and final states. An example of the graph representation of FSM T_1 for the regular expression R_1 is presented in Figure 2.

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. For example, FSM T_1 can be represented as follows.

$$M_1^a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},\ M_1^b = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Note, that an edge-labeled graph can be viewed as an FSM where edges represent transitions, and all vertices are both start and final at the same time. Thus RPQ evaluation is an intersection of two FSMs. The query result can also be represented as FSM, because regular languages are closed under intersection (Hopcroft et al., 2006).

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 (Abiteboul et al., 1995).

Definition 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)*. EBNF is a code that expresses the grammar of a formal language. An EBNF consists of terminal symbols and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. Examples of terminal symbols include alphanumeric characters, punctuation marks, and whitespace characters. 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 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 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 $\mathcal{G}$ (Figure 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 Alur et al. (2001).

Definition 14 A *recursive state machine* R over a finite alphabet Σ is defined as a tuple of elements $\langle B, m, \{C_i\}_{i \in B} \rangle$, where:

- B is a finite set of labels of boxes,
- $m \in B$ is an initial box label,
- Set of *component state machines* or *boxes*, where $C_i = (\Sigma \cup B, Q_i, q_i^0, F_i, \delta_i)$:
  - $\Sigma \cup B$ is a set of symbols, $\Sigma \cap B = \varnothing$,
  - 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 $F_i \subseteq Q_i$,
  - $\delta_i : Q_i \times (\Sigma \cup B) \to Q_i$ 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* $|R|$ 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 $(q_i, \mathcal{S})$, where q_i is a current state for box C_i and $\mathcal{S}$ 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 $(q_i, \mathcal{S})$ to $(q', \mathcal{S}')$ on some input symbol a:

- $(q_i^k, \mathcal{S}) \rightsquigarrow (\delta_i(q_i^k, a), \mathcal{S})$
- $(q_i^k, \mathcal{S}) \rightsquigarrow (q_j^0, \delta_i(q_i^k, j) \circ \mathcal{S})$

– $(q_j^k, q_i^t \circ \mathcal{S}) \rightsquigarrow (q_i^t, \mathcal{S})$, 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 Alur et al. (2001), recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages (Hopcroft et al., 2006), 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 N_i is constructed using the right-hand side of each rule for N_i. It is easy to see that such a conversion implies that $|R_G| = O(|G|) = O(|P|)$, because the number of states in R_G does not exceed the sum of the lengths of every rule in G in the worst case.

An example of such RSM R constructed for the grammar G with rules $S \to aSb \mid ab$ is provided in Figure 3. 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 stack of return states is empty, the machine accepts the input sequence *aabb*.

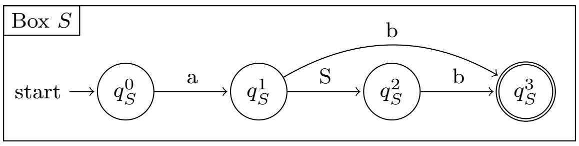


Fig. 3: The recursive state machine R for grammar G

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 for the RSM R from Figure 3 is:

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

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 (Hopcroft et al., 2006), 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 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 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 and only 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 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. 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 18 Given two FSMs $T_1 = \langle \Sigma, Q^1, Q^1_S, Q^1_F, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, Q^2_S, Q^2_F, \delta^2 \rangle$, the *intersection* of these two machines is a new FSM $T = \langle \Sigma, Q, Q_S, Q_F, \delta \rangle$, where:
- $Q = Q^1 \times Q^2$
- $Q_S = Q^1_S \times Q^2_S$
- $Q_F = Q^1_F \times Q^2_F$
- $\delta : Q \times \Sigma \to Q$, $\delta(\langle q_1, q_2 \rangle, s) = \langle q'_1, q'_2 \rangle$,
  if $\delta(q_1, s) = q'_1$ and $\delta(q_2, s) = q'_2$

According to Hopcroft et al. (2006) 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 representation 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 19 Given two sets of Boolean adjacency matrices $\mathcal{M}_1$ and $\mathcal{M}_2$, the *Kronecker product* of these matrices is a new matrix $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$, where $\mathcal{M} = \{M^a_1 \otimes M^a_2 \mid a \in \Sigma\}$ and the element-wise operation is *and* over Boolean values.

Applying the Kronecker product for both the FSM and the edge-labeled directed graph, we can intersect these objects as shown in Def. 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 (2015)) 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 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, 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 . As shown in the Def. 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 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 R adjacency matrix $\mathcal{M}_1$ and the input graph $\mathcal{G}$ adjacency matrix $\mathcal{M}_2$, followed by transitive closure, extraction of nonterminals and updating the graph adjacency matrix $\mathcal{M}_2$. Listing 1 shows main steps of the algorithm.

3.1.1 Application of Dynamic Transitive Closure

It is easy to see that the most time-consuming steps of the algorithm are the Kronecker product and transitive closure computations. Note that the adjacency matrix $\mathcal{M}_2$ is always changed 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 $\mathcal{M}_1 \otimes \mathcal{A}_2$.

Listing 1 Kronecker product based CFPQ using dynamic transitive closure

```
function contextFreePathQuerying(G, $\mathcal{G}$)
    $n \leftarrow$ The number of nodes in $\mathcal{G}$
    $R \leftarrow$ Recursive automata for $G$
    $\mathcal{M}_1 \leftarrow$ Boolean adjacency matrix for $R$
    $\mathcal{M}_2, \mathcal{A}_2 \leftarrow$ Boolean adjacency matrix for $\mathcal{G}$
    $C_3 \leftarrow$ The empty matrix of size $|R|n \times |R|n$
    for $s \in 0..dim(\mathcal{M}_1) - 1$ do
        for $S \in getNonterminals(R, s, s)$ do
            for $i \in 0..dim(\mathcal{M}_2) - 1$ do
                $M_2^S[i, i] \leftarrow 1$
            end for
        end for
    end for
    while $\mathcal{M}_2$ is changing do
        $M_3' \leftarrow \bigvee_{M^S \in \mathcal{M}_1 \otimes \mathcal{A}_2} M^S$
        $\mathcal{A}_2 \leftarrow$ The empty matrix
        $C_3' \leftarrow$ The empty matrix of size $|R|n \times |R|n$
        for $(i, j) \mid M_3'[i, j] \neq 0$ do
            $C_3' \leftarrow add(C_3, C_3', i, j)$          ▷ Updating the transitive closure
            $C_3 \leftarrow C_3 + C_3'$
        end for
        for $(i, j) \mid C_3'[i, j] \neq 0$ do
            $s, f \leftarrow getStates(C_3', i, j)$
            $x, y \leftarrow getCoordinates(C_3', i, j)$
            for $S \in getNonterminals(R, s, f)$ do
                $M_2^S[x, y] \leftarrow 1$
                $A_2^S[x, y] \leftarrow 1$
            end for
        end for
    end while
    return $\mathcal{M}_2, C_3$
end function
function getStates($C, i, j$)
    $r \leftarrow dim(\mathcal{M}_1)$          ▷ $\mathcal{M}_1$ is adjacency matrix for $R$
    return $\lfloor i/r \rfloor, \lfloor j/r \rfloor$
end function
function getCoordinates($C, i, j$)
    $n \leftarrow dim(\mathcal{M}_2)$          ▷ $\mathcal{M}_2$ is adjacency matrix for $\mathcal{G}$
    return $i \bmod n, j \bmod n$
end function
```

The fast computation of transitive closure can be obtained by using incremental dynamic transitive closure technique. Now we describe the function *add* from Listing 1. Let C_3 be a transitive closure matrix of the graph G with n vertices. We use an approach by Ibaraki and Katoh (1983) to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices, which become reachable after insertion of the certain edge. We have modified it to achieve a logarithmic speed-up.

For each newly inserted edge (i, j) and every node $u \neq j$ of G such that $C_3[u, i] = 1$ and $C_3[u, j] = 0$, one needs to perform operation $C_3[u, v] = C_3[u, v] \wedge C_3[j, v]$ for every node v, where $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$ and

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

To achieve logarithmic speed-up, we use the Four Russians' trick. First we split each vector into $n/\log n$ parts of size $\log n$. Then we create a table T such that $T(a,b) = a \wedge b$ where $a, b \in \{0,1\}^{\log n}$. This takes time $O(n^2 \log n)$, since there are $2^{\log n} = n$ variants of Boolean vectors of size $\log n$ and hence n^2 possible pairs of vectors (a,b) in total, and each component takes $O(\log n)$ time. With table T, we can calculate product of two parts of size $\log n$ in constant time. There are $n/\log n$ such parts, so the element-wise product of two vectors of size n can be calculated in time $O(n/\log n)$ with $O(n^2 \log n)$ preprocessing.

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

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

Theorem 2 *Let $\mathcal{G} = \langle V, E, L \rangle$ be a graph and $G = \langle \Sigma, N, S, P \rangle$ be a grammar. The algorithm from Listing 1 calculates resulting matrices $\mathcal{M}_2$ and C_3 in $O(|P|^3 n^3/\log(|P|n))$ time where $n = |V|$. Moreover, the maintaining of the dynamic transitive closure dominates the cost of the algorithm.*

Proof Let $|\mathcal{A}|$ be a number of non-zero elements in a matrix $\mathcal{A}$. Consider the total time which is needed for computing the Kronecker products. The elements of the matrices $\mathcal{A}_2^{(i)}$ are pairwise distinct on every i-th iteration of the algorithm therefore the total number of operations is $\sum_i T(\mathcal{M}_1 \otimes \mathcal{A}_2^{(i)}) = |\mathcal{M}_1| \sum_i |\mathcal{A}_2^{(i)}| = (|N| + |\Sigma|)|P|^2 \sum_i |\mathcal{A}_2^{(i)}| = O((|N| + |\Sigma|)^2 |P|^2 n^2)$.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that C_3 has size of the Kronecker product of $\mathcal{M}_1 \otimes \mathcal{M}_2$, 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. Checking condition whether $C_3[u,i] = 1$ and $C_3[u,j] = 0$ for every node $u \in V$ for each newly inserted edge (i,j) requires one multiplication of vectors per insertion, thus total time is $O(|P|^3 n^3 / \log(|P|n))$. Note that after checking the condition, at least one element $C[u',j]$ changes value from 0 to 1 and then never becomes 0 for some u' and j. Therefore the operation $C_3[u',v] = C_3[u',v] \wedge C_3[j,v]$ for all $v \in V$ is executed at most once for every pair of vertices (u',j) during the entire computation implying that the total time is equal to $O(|P|^2 n^2 |P| n / \log(|P|n)) = O(|P|^3 n^3 / \log(|P|n))$ (using the multiplication of vectors).

The matrix C_3' contains only new elements, therefore C_3 can be updated directly using only $|C_3'|$ operations and hence $|P|^2 n^2$ operations in total. The same holds for cycle in line 17 of the algorithm from Listing 1, because operations are performed only for non-zero elements of the matrix $|C_3'|$. Finally, the time complexity of the algorithm is $O((|N| + |\Sigma|)^2 |P|^2 n^2) + O(|P|^2 n^2) + O(|P|^2 n^2 \log(|P|n)) + O(|P|^3 n^3 / \log(|P|n)) + O(|P|^2 n^2) = O(|P|^3 n^3 / \log(|P|n))$. $\square$

The complexity analysis of the Algorithm 1 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. OMv Conjecture introduced by Henzinger et al. (2015) 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.

3.1.2 Index creation for RPQ

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

3.2 Paths Extraction Algorithm

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

Listing 2 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
end function
function getSubpaths$((s_i, v_i), (s_j, v_j), (s_k, v_k))$
  $$l \leftarrow \{(v_i, t, v_k) \mid M_2^t[s_i, s_k] \wedge M_1^t[v_i, v_k]\} \cup \bigcup_{\{N \mid M_2^N[s_i, s_k]\}} \text{getPaths}(v_i, v_k, N) \cup \text{getPathsInner}((s_i, v_i), (s_k, v_k))$$
  $$r \leftarrow \{(v_k, t, v_j) \mid M_2^t[s_k, s_j] \wedge M_1^t[v_k, v_j]\} \cup \bigcup_{\{N \mid M_2^N[s_k, s_j]\}} \text{getPaths}(v_k, v_j, N) \cup \text{getPathsInner}((s_k, v_k), (s_j, v_j))$$
  return $l \cdot r$
end function
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$((s_i, v_i), (s_j, v_j), (s_k, v_k))$
end function

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

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

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 as 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 3.

In the first step we represent graph and RSM as a set of Boolean matrices. 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.

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

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

As far as the input graph has no edges with label S, therefore, the correspon-

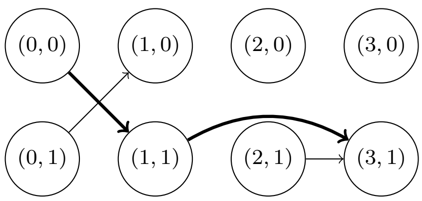


Fig. 4: The Kronecker product graph of RSM R and the input graph $\mathcal{G}$, edges forming new paths are marked bold

dent 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 4. 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 4).

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 5). 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.

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

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

The result graph is presented in Figure 6.

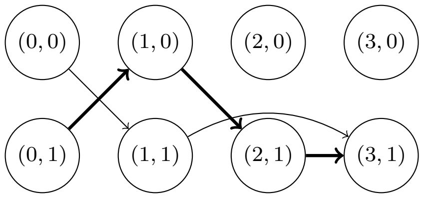


Fig. 5: The Kronecker product graph of RSM R and the updated graph $\mathcal{G}$, edges forming new paths are marked bold

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 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 7. First, we must query paths

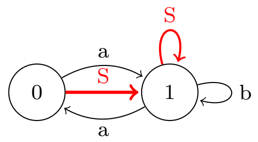


Fig. 6: The result graph $\mathcal{G}$

- `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 (leads to infinite set of paths)`
        - ..
          - `return` $r_{\infty}^{0\rightsquigarrow 1}$ `//`
            `An infinite set of path from 0 to 1`
          - ...
  - `return` $\{1 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{b} 1\} \cup (\{1 \xrightarrow{a} 0\} \cdot r_{\infty}^{0\rightsquigarrow 1} \cdot \{1 \xrightarrow{b} 1\})$

Fig. 7: Example of call stack trace

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 Evaluation

The goal of this evaluation was to investigate the applicability of the proposed algorithm to both regular and context-free path querying. We measured the execution time of the index creation, which solves the reachability problem, for both kinds of queries. The execution time for CFPQ was compared with the Azimov's algorithm for CFPQ reachability. We also investigated the practical applicability of paths extraction algorithm for both regular and context-free path queries.

For evaluation, we used a PC with Ubuntu 18.04 installed. It has Intel core i7-6700 CPU, 3.4GHz, and DDR4 64Gb RAM. We only measure the execution time of the algorithms themselves, thus we assume an input graph is loaded into RAM in the form of its adjacency matrix in the sparse format. Note, that the time needed to load an input graph into the RAM is excluded from the time measurements.

4.1 RPQ Evaluation

To investigate the applicability of the proposed algorithm for regular path querying we gathered a dataset which consists of both real-world and synthetically generated graphs. We generated the queries from the most popular RPQ templates.

4.1.1 Dataset

We gathered several graphs which represent real-world data from different areas and are frequently used for evaluation of the graph querying algorithms. Namely, the dataset consists of three parts. The first part is the set of LUBM graphs[2] (Guo et al., 2005) which have different numbers of vertices. The second one is the set of graphs from Uniprot database[3]: *proteomes*, *taxonomy* and *uniprotkb*. The last part consists of the RDF files *mappingbased_properties* from DBpedia[4] and *geospecies*[5]. A brief description of the graphs in the dataset is presented in Table 1.

Queries for evaluation were generated from the templates for the most popular RPQs, specifically the queries presented in Table 2 in Pacaci et al. (2020) and in Table 5 in Wang et al. (2019). These query templates are presented in Table 2. We generate 10 queries for each template and each graph. The most

[2] Lehigh University Benchmark (LUBM) web page: `http://swat.cse.lehigh.edu/projects/lubm/`. Access date: 07.07.2020.

[3] Universal Protein Resource (UniProt) web page: `https://www.uniprot.org/`. All files used can be downloaded via the link: `ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf/`. Access date: 07.07.2020.

[4] DBpedia project web site: `https://wiki.dbpedia.org/`. Access date: 07.07.2020.

[5] The Geospecies RDF: `https://old.datahub.io/dataset/geospecies`. Access date: 07.07.2020.

Table 1: Graphs for RPQ evaluation

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 2: Queries templates 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}	$a \cdot b \cdot (c \mid d \mid e)$

frequent relations from the given graph were used as symbols in the query template[6]. We used the same set of queries for all LUBM graphs to investigate scalability of the proposed algorithm.

4.1.2 Results

We averaged the execution time of index creation over 5 runs for each query. Index creation time for LUBM graphs set is presented in Figure 8. We can see that evaluation time depends on the query: there are queries which evaluate in less than 1 second even for the largest graphs (Q_2, Q_5, Q_{11}^2, Q_{11}^3), while the worst time is 6.26 seconds (Q_{14}). The execution time of our algorithm

[6] 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.

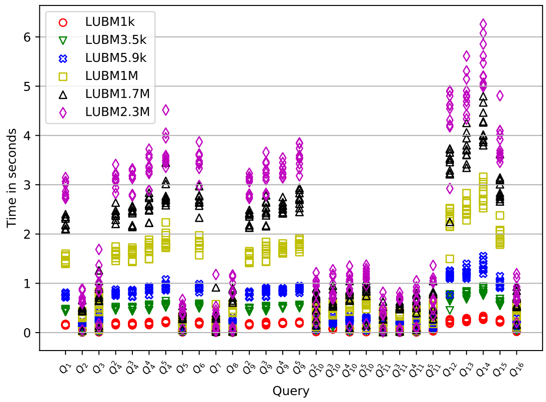


Fig. 8: Index creation time for LUBM graphs

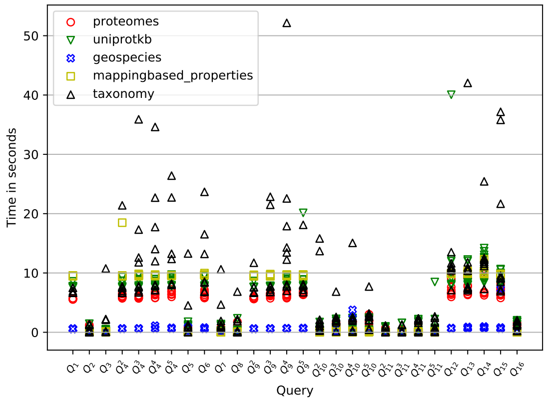
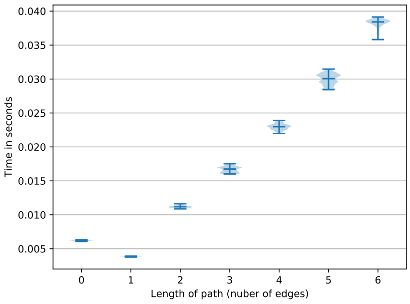


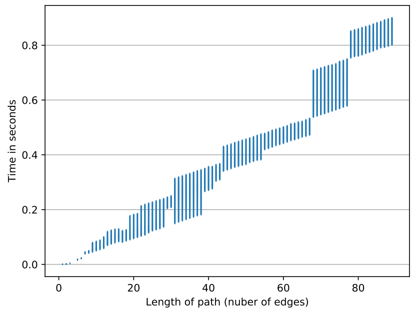
Fig. 9: Index creation time for real-world RDFs

is comparable with the recent results for the same graphs and queries implemented on a distributed system over 10 nodes (Wang et al., 2019), while we use only one node. We conclude that our algorithm demonstrates reasonable performance to be applied for real-world data analysis.

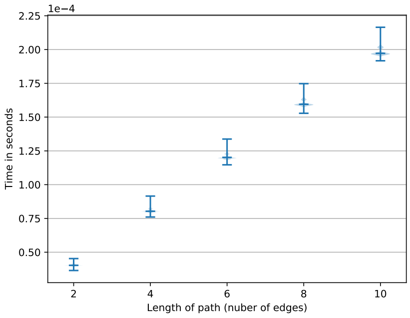
Index creation time for each query on the real-world graphs is presented in Figure 9. We can see that in some cases querying small graphs requires more time than querying bigger graphs. For example, conseder Q_{10}^4: querying the *geospecies* graph (450k vertices) in some cases requires more time than querying of *mappingbased_properties* (8.3M vertices) and *taxonomy* (5.7M vertices). We conclude that the evaluation time depends on the inner structure of a graph. On the other hand, *taxonomy* querying in many cases requires significantly more time, than for other graphs, while *taxonomy* is not the biggest graph. Finally, in most cases query execution lasts less than 10 seconds, even for bigger graphs, and no query requires more than 52.17 seconds.



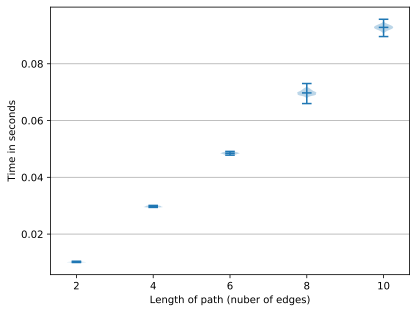
(a) *geospecies*, Q_{14}



(b) *mappingbased_properties*, Q_4^5



(c) *geospecies*, *Geo*



(d) *geospecies*, *Geo*

Fig. 10: Single path extraction for specific graph and query for our solution (a, b, d), and Azimov's (c)

We evaluate path extraction for queries which result in possibly long paths. Long paths usually require many iterations of transitive closure evaluation, thus we used the number of the iterations as a criterion to select the inputs for the evaluation. For each selected graph and query we measure paths extraction time for each reachable pair. Since the index can be reused from the previous step, we omit the time necessary to create the index. We limit by 10 the number of paths to extract.

In Figures 10a and 10b we show the time needed to extract a path of a specific length when only one path was extracted. The main observation is that time is linear on the path length, even if a generic path extraction procedure is used.

4.2 CFPQ Evaluation

We evaluate the applicability of the proposed algorithm for CFPQ processing over real-world graphs on a number of classical cases and compare them with the Azimov's algorithm. Currently only a single path version of Azimov's algorithm exists, and we use its implementation using PyGraphBLAS. Note that it is not so trivial to compare our results with the state-of-the-art results provided by Terekhov et al. (2020) (Azimov's algorithm) because our algorithm computes significantly more information. While the state-of-the-art solution computes only reachability facts or a single-path semantics, our algorithm computes data necessary to restore all possible paths.

4.2.1 Dataset

We use CFPQ_Data[7] dataset for evaluation. Namely, we use relatively big RDF files and respective same-generation queries G_1 (Eq. 1) and G_2 (Eq. 2) which are used in other works for CFPQ evaluation. We also use the *Geo* (Eq. 3) query provided by Kuijpers et al. (2019) for *geospecies* RDF. Note that we use $\overline{x}$ notation in queries to denote the inverse of x relation and the respective edge.

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

$$S \to \overline{subClassOf}\ \ S\ subClasOf \mid subClassOf \tag{2}$$

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

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

Additionally we evaluate our algorithm on memory aliases analysis problem: a well-known problem which can be reduced to CFPQ (Zheng and Rugina, 2008). To do it, we use some graphs built for different parts of Linux OS kernel (*arch*, *crypto*, *drivers*, *fs*) and the query MA (Eq. 4) (Wang et al., 2017). The detailed data about all the graphs used is presented in Table 3.

4.2.2 Results

We averaged the index creation time over 5 runs for both single-path Azimov's algorithm (**Mtx**) and the proposed algorithm (**Tns**) (see Table 4).

We can see that while in some cases our solution is comparable or just slightly better than Azimov's algorithm (*enzyme, eclass_514en, go*), there are cases when our solution is significantly faster (*go-hierarchy*, up to 9 times faster), and when Azimov's algorithm about 2 times faster (all memory aliases and *geospecies* with *Geo* query). Thus we can conclude that our solution is

[7] 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 3: Graphs for CFPQ evaluation: *bt* is broaderTransitive, *sco* is subCalssOf

Graph	#V	#E	#sco	#type	#bt	#a	#d
eclass_514en	239 111	523 727	90 512	72 517	—	—	—
enzyme	48 815	109 695	8 163	14 989	—	—	—
geospecies	450 609	2 201 532	0	89 062	20 867	—	—
go	272 770	534 311	90 512	58 483	—	—	—
go-hierarchy	45 007	980 218	490 109	0	—	—	—
taxonomy	5 728 398	14 922 125	2 112 637	2 508 635	—	—	—
arch	3 448 422	5 940 484	—	—	—	671 295	2 298 947
crypto	3 464 970	5 976 774	—	—	—	678 408	2 309 979
drivers	4 273 803	7 415 538	—	—	—	858 568	2 849 201
fs	4 177 416	7 218 746	—	—	—	824 430	2 784 943

Table 4: CFPQ evaluation results, time is measured in seconds

Name	G_1		G_2		*Geo*		*MA*	
	Tns	Mtx	Tns	Mtx	Tns	Mtx	Tns	Mtx
eclass_514en	0.25	0.27	0.23	0.26	—	—	—	—
enzyme	0.04	0.04	0.04	0.01	—	—	—	—
geospecies	0.09	0.06	0.01	0.01	34.12	16.58	—	—
go-hierarchy	0.19	1.43	0.29	0.86	—	—	—	—
go	1.68	1.74	1.37	1.14	—	—	—	—
pathways	0.02	0.01	0.01	0.01	—	—	—	—
taxonomy	5.37	2.71	3.28	1.56	—	—	—	—
arch	—	—	—	—	—	—	390.05	195.51
crypto	—	—	—	—	—	—	395.98	195.54
drivers	—	—	—	—	—	—	2114.16	1050.78
fs	—	—	—	—	—	—	745.97	370.73

competitive with Azimov's algorithm, and a detailed analysis of different cases is required.

Comparison of paths extraction is presented in Figures 10c and 10d. While both methods demonstrate linear time on the length of the extracted path, our generic solution is more than 1000 times slower than Azimov's single path extraction procedure. We conclude that current generic all-path extraction procedure is not optimal for single path extraction.

4.3 Conclusion

We conclude that the proposed algorithm is applicable for real-world data processing: the algorithm allows one to solve both the reachability problem and to extract paths of interest in a reasonable time even using naive implementation. While index creation time (reachability query evaluation) is comparable with other existing solutions, paths extraction procedure should be improved in the future. However, the state-of-the-art solution computes only reachability facts or a single-path semantics, whereas our algorithm computes data necessary to restore all possible paths (all-paths semantics). Finally, a detailed comparison of the proposed solution with other algorithms for CFPQ and RPQ is required.

To summarize the overall evaluation, the proposed algorithm is applicable for both RPQ and CFPQ over real-world graphs. Thus, the proposed solution is a promising unified algorithm for both RPQ and CFPQ evaluation.

5 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. (2000). 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 (2016), which is implemented on the top of Pregel-based system, or solution by Koschmieder and Leser (2012). But only a limited number of practical solutions provide the ability to restore paths of interest. A recent work by Wang et al. (2019) provides a Pregel-based provenance-aware RPQ algorithm which utilizes a Glushkov's construction (Glushkov, 1961). 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 by Hellings (2014) and Bradford (2017), (G)LR and (G)LL in works by Verbitskaia et al. (2016), Grigorev and Ragozina (2017), Santos et al. (2018), Medeiros et al. (2018). Unfortunately, none of them has better than cubic time complexity in terms of the input graph size. The algorithm by Azimov and Grigorev (2018) is, best to our knowledge, the first algorithm for CFPQ based on linear algebra. It was shown by Terekhov et al. (2020) that this algorithm can be applied for real-world graph analysis problems, while Kuijpers et al. (2019) show 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. (2020) provide the first CFPQ algorithm with single path semantics based on linear algebra, Hellings (2020) 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.

While CFPQ on n-node graph has a relatively straightforward $O(n^3)$ time algorithm, it is a long-standing open problem whether there is a truly subcubic $O(n^{3-\varepsilon})$ algorithm for this problem. The question on the existence of a subcubic CFPQ algorithm was stated by Yannakakis (1990). A bit later Reps (1997) proposed the CFL reachability as a framework for interprocedural static code analysis. Melski and Reps (1997) gave a dynamic programming formulation of the problem which runs in $O(n^3)$ time. The problem of the cubic bottleneck of context-free language reachability is also discussed by Heintze and McAllester (1997) and Melski and Reps (1997). The slightly subcubic algorithm with $O(n^3/\log n)$ time complexity was provided by Chaudhuri (2008). This result is inspired by recursive state machine reachability. The first truly subcubic algorithm with $O(n^\omega polylog(n))$ time complexity (ω is the best exponent for matrix multiplication) for an arbitrary graph and 1-Dyck language was provided by Bradford (2017), and Pavlogiannis and Mathiasen (2020). Other partial cases were investigated by Chatterjee et al. (2017), Zhang (2020).

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 (Kepner et al., 2016) 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 (Davis, 2019) or CombBLAS (Buluç and Gilbert, 2011). 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 (Jamour et al., 2019; Metzler and Miettinen, 2015) and for Datalog queries evaluation (SATO, 2017). Finally, RedisGraph (Cailliau et al., 2019), a linear-algebra powered graph database, was created and it was shown that in some scenarios it outperforms many other graph databases.

6 Conclusion and Future Work

In this work we present an improved version of the tensor-based algorithm for CFPQ: we reduce the algorithm to operations over Boolean matrices, and we provide the ability to extract all paths which satisfy the query. Moreover, the provided algorithm can handle grammars in EBNF, thus it does not require CNF transformation of the grammar and avoids grammar explosion. As a result, the algorithm demonstrates practical performance not only on CFPQ queries but also on RPQ ones, which is shown by our evaluation. Thus, we provide a universal linear algebra based algorithm for RPQ and CFPQ evaluation with all-paths semantics. Moreover our algorithm opens a way to tackle the long-standing problem of subcubic CFPQ by reducing it to incremental transitive closure: incremental transitive closure with $O(n^{3-\varepsilon})$ total update time for n^2 updates, such that each update returns all of the new reachable pairs, implies $O(n^{3-\varepsilon})$ CFPQ algorithm. We prove $O(|P|^3 n^3/\log(|P|n))$ 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 (van den Brand et al., 2019; Henzinger et al., 2015).

Thus, the first task for the future is to improve the logarithmic factor in the obtained bound. It is also interesting to improve bounds in partial cases for which dynamic transitive closure can be supported faster than in general case, for example, planar graphs (Subramanian, 1993), undirected graph and others. In the case of planarity, it is interesting to investigate properties of the input graph and grammar which allow us to preserve planarity during query evaluation.

Note that our solution also has subcubic time complexity in terms of the grammar size, while the state-of-the-art solutions have a cubic complexity in terms of grammar size, and, moreover, these solutions require transforming the input grammar into Chomsky Normal Form (which leads to a quadratic blow-up in the size of the grammar) (Azimov and Grigorev, 2018; Hellings, 2014, 2015; Melski and Reps, 1997; Terekhov et al., 2020).

An important task for future research is a detailed investigation of the paths extraction algorithm. Hellings (2020) provides a theoretical investigation of single-path extraction and shows that the problem is related to the formal language theory. Extraction of all paths is more complicated and should be investigated carefully in order to provide an optimal algorithm.

From the practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work, we evaluate naive implementation in which transitive closure is recalculated from scratch on each iteration. It is shown by Hanauer et al. (2020) that some advanced algorithms for dynamic transitive closure can be efficiently implemented. Can one of these algorithms be efficiently parallelized and utilized in the proposed algorithm?

Also, it is necessary to evaluate GPGPU-based implementation. Evaluation of Azimov's algorithm shows that it is possible to improve performance by using GPGPU because operations of linear algebra can be efficiently implemented on GPGPU (Mishin et al., 2019; Terekhov et al., 2020). 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 (Chien et al., 2019; Gera et al., 2020).

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 (Li et al., 2018). 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 (Shao et al., 2013).

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph[8] (Cailliau et al., 2019) 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 Terekhov et al. (2020) to integrate Azimov's algorithm to RedisGraph with minimal effort.

References

Abiteboul S, Hull R, Vianu V (1995) *Foundations of Databases*

Alur R, Etessami K, Yannakakis M (2001) Analysis of recursive state machines. In: Berry G, Comon H, Finkel A (eds) Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 207–220

Angles R, Arenas M, Barceló P, Hogan A, Reutter J, Vrgoč D (2017) Foundations of modern query languages for graph databases. *ACM Comput Surv* 50(5), DOI 10.1145/3104031, URL https://doi.org/10.1145/3104031

Azimov R, Grigorev S (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), ACM, New York, NY, USA, GRADES-NDA '18, pp 5:1–5:10, DOI 10.1145/3210259.3210264, URL http://doi.acm.org/10.1145/3210259.3210264

Barceló Baeza P (2013) Querying graph databases. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Association for Computing Machinery, New York, NY, USA, PODS '13, pp 175–188, DOI 10.1145/2463664.2465216, URL https://doi.org/10.1145/2463664.2465216

Barrett C, Jacob R, Marathe M (2000) Formal-language-constrained path problems. *SIAM J Comput* 30(3):809–837, DOI 10.1137/S0097539798337716, URL https://doi.org/10.1137/S0097539798337716

Bradford PG (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, pp 247–253, DOI 10.1109/UEMCON.2017.8249039

[8] 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.

Buluç A, Gilbert JR (2011) The combinatorial blas: Design, implementation, and applications. *Int J High Perform Comput Appl* 25(4):496–509, DOI 10.1177/1094342011403516, URL https://doi.org/10.1177/1094342011403516

Cailliau P, Davis T, Gadepally V, Kepner J, Lipman R, Lovitz J, Ouaknine K (2019) Redisgraph graphblas enabled graph database. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp 285–286, DOI 10.1109/IPDPSW.2019.00054

Chatterjee K, Choudhary B, Pavlogiannis A (2017) Optimal dyck reachability for data-dependence and alias analysis. *Proc ACM Program Lang* 2(POPL), DOI 10.1145/3158118, URL https://doi.org/10.1145/3158118

Chaudhuri S (2008) Subcubic algorithms for recursive state machines. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, NY, USA, POPL '08, pp 159–169, DOI 10.1145/1328438.1328460, URL https://doi.org/10.1145/1328438.1328460

Chien SWD, Peng IB, Markidis S (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, pp 50–57, DOI 10.1109/MCHPC49590.2019.00014, URL https://doi.org/10.1109/MCHPC49590.2019.00014

Davis TA (2019) Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans Math Softw* 45(4), DOI 10.1145/3322125, URL https://doi.org/10.1145/3322125

Gera P, Kim H, Sao P, Kim H, Bader D (2020) Traversing large graphs on gpus with unified memory. *Proc VLDB Endow* 13(7):1119–1133, DOI 10.14778/3384345.3384358, URL https://doi.org/10.14778/3384345.3384358

Glushkov VM (1961) THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys* 16(5):1–53, DOI 10.1070/rm1961v016n05abeh004112, URL https://doi.org/10.1070/rm1961v016n05abeh004112

Grigorev S, Ragozina A (2017) Context-free path querying with structural representation of result. In: Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia, ACM, New York, NY, USA, CEE-SECR '17, pp 10:1–10:7, DOI 10.1145/3166094.3166104, URL http://doi.acm.org/10.1145/3166094.3166104

Guo Y, Pan Z, Heflin J (2005) Lubm: A benchmark for owl knowledge base systems. *Web Semant* 3(2–3):158–182, DOI 10.1016/j.websem.2005.06.005, URL https://doi.org/10.1016/j.websem.2005.06.005

Hanauer K, Henzinger M, Schulz C (2020) Faster fully dynamic transitive closure in practice. In: Faro S, Cantone D (eds) 18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, LIPIcs, vol 160, pp 14:1–14:14, DOI 10.4230/LIPIcs.SEA.2020.14, URL http://eprints.cs.univie.ac.at/6345/

Heintze N, McAllester D (1997) On the cubic bottleneck in subtyping and flow analysis. In: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, USA, LICS '97, p 342

Hellings J (2014) Conjunctive context-free path queries. In: Proceedings of ICDT'14, pp 119–130

Hellings J (2015) Querying for paths in graphs using context-free path queries. *arXiv preprint arXiv:150202242*

Hellings J (2020) Explaining results of path queries on graphs: Single-path results for context-free path queries, to appear

Henzinger M, Krinninger S, Nanongkai D, Saranurak T (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, Association for Computing Machinery, New York, NY, USA, STOC '15, p 2130, DOI 10.1145/2746539.2746609, URL https://doi.org/10.1145/2746539.2746609

Hopcroft JE, Motwani R, Ullman JD (2006) *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA

Ibaraki T, Katoh N (1983) On-line computation of transitive closures of graphs. *Information Processing Letters* 16(2):95 – 97, DOI https://doi.org/10.1016/0020-0190(83)90033-9, URL http://www.sciencedirect.com/science/article/pii/0020019083900339

Jamour F, Abdelaziz I, Chen Y, Kalnis P (2019) Matrix algebra framework for portable, scalable and efficient query engines for rdf graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019, Association for Computing Machinery, New York, NY, USA, EuroSys '19, DOI 10.1145/3302424.3303962, URL https://doi.org/10.1145/3302424.3303962

Kepner J, Aaltonen P, Bader D, Buluc A, Franchetti F, Gilbert J, Hutchison D, Kumar M, Lumsdaine A, Meyerhenke H, McMillan S, Yang C, Owens JD, Zalewski M, Mattson T, Moreira J (2016) Mathematical foundations of the graphblas. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–9, DOI 10.1109/HPEC.2016.7761646

Koschmieder A, Leser U (2012) Regular path queries on large graphs. In: Ailamaki A, Bowers S (eds) Scientific and Statistical Database Management, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 177–194

Kostylev EV, Reutter JL, Romero M, Vrgoč D (2015) Sparql with property paths. In: Arenas M, Corcho O, Simperl E, Strohmaier M, d'Aquin M, Srinivas K, Groth P, Dumontier M, Heflin J, Thirunarayan K, Thirunarayan K, Staab S (eds) The Semantic Web - ISWC 2015, Springer International Publishing, Cham, pp 3–18

Kuijpers J, Fletcher G, Yakovets N, Lindaaker T (2019) An experimental study of context-free path query evaluation methods. In: Proceedings of the 31st International Conference on Scientific and Statistical Database Management, ACM, New York, NY, USA, SSDBM '19, pp 121–132, DOI 10.1145/3335783.3335791, URL http://doi.acm.org/10.1145/3335783.3335791

Li J, Sun J, Vuduc R (2018) Hicoo: Hierarchical storage of sparse tensors. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE Press, SC '18

Libkin L, Martens W, Vrgoč D (2016) Querying graphs with data. *J ACM* 63(2), DOI 10.1145/2850413, URL https://doi.org/10.1145/2850413

Medeiros CM, Musicante MA, Costa US (2018) Efficient evaluation of context-free path queries for graph databases. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '18, pp 1230–1237, DOI 10.1145/3167132.3167265, URL http://doi.acm.org/10.1145/3167132.3167265

Melski D, Reps T (1997) Interconvertbility of set constraints and context-free language reachability. *SIGPLAN Not* 32(12):7489, DOI 10.1145/258994.259006, URL https://doi.org/10.1145/258994.259006

Metzler S, Miettinen P (2015) On defining SPARQL with boolean tensor algebra. *CoRR* abs/1503.00301, URL http://arxiv.org/abs/1503.00301, 1503.00301

Miao H, Deshpande A (2019) Understanding data science lifecycle provenance via graph segmentation and summarization. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp 1710–1713

Mishin N, Sokolov I, Spirin E, Kutuev V, Nemchinov E, Gorbatyuk S, Grigorev S (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), ACM, New York, NY, USA, GRADES-NDA'19, pp 12:1–12:5, DOI 10.1145/3327964.3328503, URL http://doi.acm.org/10.1145/3327964.3328503

Nolé M, Sartiani C (2016) Regular path queries on massive graphs. In: Proceedings of the 28th International Conference on Scientific and Statistical Database Management, Association for Computing Machinery, New York, NY, USA, SSDBM '16, DOI 10.1145/2949689.2949711, URL https://doi.org/10.1145/2949689.2949711

Orachev E, Epelbaum I, Azimov R, Grigorev S (2020) Context-free path querying by kronecker product. In: Darmont J, Novikov B, Wrembel R (eds) Advances in Databases and Information Systems, Springer International Publishing, Cham, pp 49–59

Pacaci A, Bonifati A, Özsu MT (2020) Regular path query evaluation on streaming graphs. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '20, pp 1415–1430, DOI 10.1145/3318464.3389733, URL https://doi.org/10.1145/3318464.3389733

Pavlogiannis A, Mathiasen AA (2020) The fine-grained and parallel complexity of andersen's pointer analysis. 2006.01491

Reps T (1997) Program analysis via graph reachability. In: Proceedings of the 1997 International Symposium on Logic Programming, MIT Press, Cambridge, MA, USA, ILPS '97, p 519 van Rest O, Hong S, Kim J, Meng X, Chafi H (2016) Pgql: A property graph query language. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Association for Computing Machinery, New York, NY, USA, GRADES '16, DOI 10.1145/2960414.2960421, URL https://doi.org/10.1145/2960414.2960421

Santos FC, Costa US, Musicante MA (2018) A bottom-up algorithm for answering context-free path queries in graph databases. In: Mikkonen T, Klamma R, Hernández J (eds) Web Engineering, Springer International Publishing, Cham, pp 225–233

SATO T (2017) A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming* 17(3):244265, DOI 10.1017/S1471068417000023

Sevon P, Eronen L (2008) Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics* 5(2):100

Shao B, Wang H, Li Y (2013) Trinity: A distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '13, pp 505–516, DOI 10.1145/2463676.2467799, URL https://doi.org/10.1145/2463676.2467799

Subramanian S (1993) A fully dynamic data structure for reachability in planar digraphs. In: Lengauer T (ed) Algorithms—ESA '93, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 372–383

Sui Y, Cheng X, Zhang G, Wang H (2020) Flow2vec: Value-flow-based precise code embedding. *Proc ACM Program Lang* 4(OOPSLA), DOI 10.1145/3428301, URL https://doi.org/10.1145/3428301

Terekhov A, Khoroshev A, Azimov R, Grigorev S (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), Association for Computing Machinery, New York, NY, USA, GRADES-NDA'20, DOI 10.1145/3398682.3399163, URL https://doi.org/10.1145/3398682.3399163

van den Brand J, Nanongkai D, Saranurak T (2019) Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In: 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS), pp 456–480

Verbitskaia E, Grigorev S, Avdyukhin D (2016) Relaxed parsing of regular approximations of string-embedded languages. In: Mazzara M, Voronkov A (eds) Perspectives of System Informatics, Springer International Publishing, Cham, pp 291–302

Wang K, Hussain A, Zuo Z, Xu G, Amiri Sani A (2017) Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *SIGPLAN Not* 52(4):389–404, DOI 10.1145/3093336.3037744, URL https://doi.org/10.1145/3093336.3037744

Wang X, Wang S, Xin Y, Yang Y, Li J, Wang X (2019) Distributed pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *World Wide Web* 23(3):1465–1496, DOI 10.1007/s11280-019-00739-0, URL https://doi.org/10.1007/s11280-019-00739-0

Yan D, Xu H, Rountev A (2011) Demand-driven context-sensitive alias analysis for java. pp 155–165, DOI 10.1145/2001420.2001440

Yannakakis M (1990) Graph-theoretic methods in database theory. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, New York, NY, USA, PODS '90, pp 230–242, DOI 10.1145/298514.298576, URL http://doi.acm.org/10.1145/298514.298576

Zhang Q (2020) Conditional lower bound for inclusion-based points-to analysis. 2007.05569

Zheng X, Rugina R (2008) Demand-driven alias analysis for c. *SIGPLAN Not* 43(1):197–208, DOI 10.1145/1328897.1328464, URL http://doi.acm.org/10.1145/1328897.1328464