PROGRAMMING, *2025*, *N*º *1*, *pp. 3–21*

UDC 004.51

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

© 2025 Shemetova E. N.[a1], Azimov R. Sh.[a2], Orachev E. S.[a3], Epelbaum I. V.[a4], Olemskaya A. V.[b5], Grigorev S. V.[a6]

[a] *St. Petersburg State University,*
7-9 Universitetskaya Embankment, St Petersburg, 199034, Russia
[b] *National Research University Higher School of Economics,*
3k1 Kantemirovskaya Street, St Petersburg, 194100, Russia
[1] *e-mail: katyacyfra@gmail.com,* ORCID: 0000
[2] *e-mail: rustam.azimov19021995@gmail.com,* ORCID: 0000-0003-0223-5172
[3] *e-mail: egor.orachev@gmail.com,* ORCID: 0000
[4] *e-mail: iliyepelbaun@gmail.com,* ORCID: 0000
[5] *e-mail: alexandra.olemskaya@gmail.com,* ORCID: 0000
[6] *e-mail: s.v.grigoriev@spbu.ru,* ORCID: 0000

Accepted for publication: 0.11.2024

The Kronecker product-based algorithm for context-free path querying (CFPQ) was proposed by **?**. 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 complexity proposed by **?**. Our evaluation shows that our algorithm is a good candidate to be the universal algorithm for both regular and context-free path querying.

1. Introduction

Language-constrained path querying **?** is a technique for graph navigation querying. This technique allows one to use formal languages as constraints on paths in edge-labeled graphs: 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 **???**, while regular constraints are supported in such popular query languages as PGQL **?** and SPARQL[1] **?** (known as property paths). Nevertheless, there is certainly room for improvement of RPQ efficiency, and new solutions are being created **??**.

At the same time, using more powerful languages as constraints, namely context-free languages, has gained popularity in recent years. *Context-Free Path Querying* problem (CFPQ) was introduced by **?**, and nowadays is used in many areas. For example, CFPQ is used for interprocedural static code analysis **????** 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 **?**, graph segmentation in data provenance analysis **?**, and for data flow information preserving in machine learning based solutions for code analysis problems **?**.

Many algorithms for CPFQ were proposed, but recently **?** showed that the state-of-the-art CFPQ

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

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 **?** was proposed. Evaluation of the libraries that implement this API, such as SuiteSparce **?** and CombBLAS **?**, show that reduction to linear algebra is a good way to utilize high-performance parallel and distributed computations for graph analysis.

? showed how to reduce CFPQ to matrix multiplication. Later, it was shown by **?** and **?** 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 **?**. 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 **?**, 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 **??**. Currently, the best result is an $O(n^3/\log n)$ algorithm of **?**. Also, there exist truly subcubic solutions that use fast matrix multiplication for some fixed subclasses of context-free languages **?**. 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 **?**. 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 a 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 a faster CFPQ algorithm for the 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 some basic notation and definitions from graph theory and formal language theory which will be used in the rest of the paper.

2.1 Language-Constrained Path Querying Problem

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

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 graph $\mathcal{G}$ which we use in the further examples is presented in Figure 1.

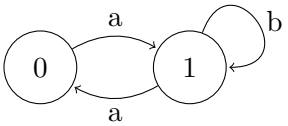

Figure 1.: The example of input graph $\mathcal{G}$

Definition 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

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

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.

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

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 a disjunction $\vee$ and a *multiplication* $\cdot$ as a conjunction $\wedge$ over Boolean values.

We also use the 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 that 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 **?**. 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 $V = Q$, $L = \Sigma$, $E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$ 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.

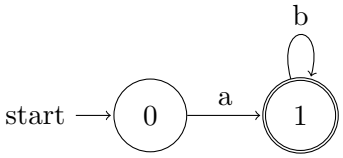

Figure 2.: The example of graph representation of FSM for regular expression ab^*

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

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

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

2.3 Context-Free Path Querying and Recursive State Machines

An even more general case than RPQ is a *Context-Free Path Querying Problem (CFPQ)*, where one

can use context-free languages as constraints. These constraints are more expressive than regular ones. For example, a classic same-generation query can be expressed by a context-free language, but not a regular language **?**.

Definition 11. *A* context-free grammar G *is a tuple* $\langle \Sigma, N, S, P \rangle$*, where:*

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

Definition 12. *The* size of the grammar $G = \langle \Sigma, N, S, P \rangle$ *is defined as the sum of the sizes of its productions:*

$$|G| = \sum_{p \in P} |p|,$$

where the size of a production $p = N \to \alpha$ *depends on the length of its right-hand side:*

$$|p| = 1 + |\alpha|.$$

Definition 13. *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 14. *Context-free grammar* $G = \langle \Sigma, N, S, P \rangle$ *specifies a* context-free language*:* $\mathcal{L}(G) = \{\omega \mid S \overset{*}{\to} \omega\}$*, where* $(\overset{*}{\to})$ *denotes zero or more derivation steps* $(\to)$.

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

Definition 15. *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.*

Definition 16. *The* size of RSM $|R|$ *is defined as the sum of the number of states in all boxes.*

RSM behaves as a set of finite state machines (or FSM). Each such FSM is called a *box* or a *component state machine*. A box works similarly to the classic 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 **?**, recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages **?**, RSMs are equivalent to context-free languages. Thus RSMs are suitable 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. Such 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_1 with rules $S \to aSb \mid ab$ is provided in Figure 3. For the given example of the grammar and the RSM, consider the following sequence of the machine configuration transitions, in the case where one wants to determine if the input word *aabb* belongs to the language $L(G_1)$. The RSM execution starts from configuration $(q_S^0, \langle\rangle)$, reads the symbol a and goes the to $(q_S^1, \langle\rangle)$. Then, in a nondeterministic manner it tries to read b but fails, and in the same time tries to derive S and goes to the configuration $(q_S^0, \langle q_S^2 \rangle)$, where q_S^2 is a *return* state. Then machine reads a and goes to $(q_S^1, \langle q_S^2 \rangle)$. In this case, it fails to derive S in the nondeterministic choice, but successfully reads b and goes to the configuration $(q_S^3, \langle q_S^2 \rangle)$. Since q_S^3 is a final state for the box S, the RSM tries to *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*.

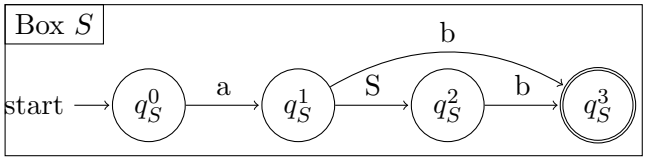

Figure 3.: The recursive state machine R for grammar G_1

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:

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

Similarly to an RPQ, a CFPQ is the intersection of the given context-free language and an 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 the classic Kronecker product definition, describe graph Kronecker product and its relation to Boolean matrices algebra, and RSM and FSM intersection.

Definition 17. *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 to 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 18. *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 can easily be 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 19. *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 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. 18. Thus, $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$, where $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$.

Definition 20. *Given two finite state machines $T_1 = \langle \Sigma, Q^1, Q^1_S, Q^1_F, \delta^1 \rangle$ 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 **?** an FSM intersection defines the machine for which $L(T) = L(T_1) \cap L(T_2)$.

The most substantial part of the 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 to some extent, since the transition function δ of the machine T in the matrix form is exactly the same as the product result. More precisely:

Definition 21. *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 a conjunction over Boolean values ($\wedge$).*

Applying the Kronecker product for both the FSM and the edge-labeled directed graph, we can intersect these objects as shown in Def. 21, 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 a Boolean semiring.

3. Context-free path querying by Kronecker product

In this section, we introduce the algorithm for context-free path querying which is based on the Kronecker product of Boolean matrices. The algorithm solves all-pairs CFPQ problem with all-path semantics (according to **?**). The algorithm works in the following two steps.

1. *Index creation.* During this step, the algorithm computes an index that contains the information necessary to restore paths for the given pairs of vertices. This index can be used to solve the reachability problem without extracting paths. Note that the index is finite even when 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, thus 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 the correctness of the algorithm, and provide time complexity estimations. For the first step, we start by introducing a naive algorithm. After that, we show how to achieve cubic time complexity by using a dynamic transitive closure algorithm and shave off a logarithmic factor to achieve the best known time complexity for the CFPQ problem. We finish by providing a step-by-step example of query evaluation with 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 easily be presented as an adjacency matrix for some graph over the set of labels. As shown in the Def. 21, we can apply Kronecker product for Boolean matrices to *intersect* the RSM and the input graph to some extent. But the RSM contains nonterminal symbols with the additional logic of *recursive calls*, which requires a *transitive closure* step to extract such symbols.

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

3.1.1 Application of Dynamic Transitive Closure

The most time-consuming steps of the algorithm are the computations of the Kronecker product and transitive closure. Note that the adjacency matrix $\mathcal{M}_2$ is changed incrementally i.e. elements (edges) are added to $\mathcal{M}_2$ at each iteration of the algorithm and are never deleted from it. So it is not necessary to recompute the whole product or transitive closure if some 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 all previously found elements, such that $\mathcal{M}_2 = \mathcal{A}_2 + \mathcal{B}_2$. Then by 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 is also 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 an incremental dynamic transitive closure technique. Let us 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 **?** to maintain dynamic transitive clo-

Listing 1 Kronecker product based CFPQ using dynamic transitive closure

```
1: function contextFreePathQuerying(G, 𝒢)
2:     n ← the number of nodes in 𝒢
3:     R ← recursive automata for G
4:     ℳ1 ← the set of adjacency matrices for R
5:     ℳ2, 𝒜2 ← the sets of adjacency matrices for 𝒢
6:     C3, M3 ← the empty matrices of size dim(ℳ1)n × dim(ℳ1)n
7:     for s ∈ 0..dim(ℳ1) − 1 do
8:         for S ∈ getNonterminals(R, s, s) do
9:             for i ∈ 0..dim(ℳ2) − 1 do
10:                M2^S[i, i] ← 1
11:            end for
12:        end for
13:    end for
14:    while ℳ2 is changing do
15:        M3' ← ⋁_{M^S ∈ ℳ1⊗𝒜2} M^S ▷ Using only new edges from 𝒜2
16:        M3 ← M3 + M3'  ▷ Updating the matrix for the Kronecker product result
17:        𝒜2 ← the empty matrix
18:        C3' ← the empty matrix of size dim(ℳ1)n × dim(ℳ1)n
19:        for (i, j) | M3'[i, j] ≠ 0 do
20:            C3' ← add(C3, C3', i, j)  ▷ Updating the transitive closure
21:            C3 ← C3 + C3'
22:        end for
23:        for (i, j) | C3'[i, j] ≠ 0 do
24:            s, f ← getStates(C3', i, j)
25:            x, y ← getCoordinates(C3', i, j)
26:            for S ∈ getNonterminals(R, s, f) do
27:                M2^S[x, y] ← 1
28:                A2^S[x, y] ← 1
29:            end for
30:        end for
31:    end while
32:    return ℳ2, M3
33: end function
34: function getStates(C, i, j)
35:    n ← dim(ℳ2)  ▷ ℳ2 the set of adjacency matrices for 𝒢
36:    return ⌊i/n⌋, ⌊j/n⌋
37: end function
38: function getCoordinates(C, i, j)
39:    n ← dim(ℳ2)
40:    return i mod n, j mod n
41: end function
```

sure. The key idea of their algorithm is to recalculate reachability information only for those vertices which become reachable after insertion of a certain edge.

We have modified the algorithm to achieve a logarithmic speed-up in the following way. 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 the element-wise product of two vectors of size n.

To achieve logarithmic speed-up, we use the Four Russians' trick by **?**. Let us assume an architecture with word size $w = \theta(\log n)$. 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. Assuming constant-time logical operations on words, we can store a polynomial number of lookup tables (arrays) T_i (one array for each vector of size $\log n$), such that given an index of a table T_i, and any $O(\log n)$ bit vector b, we can look up $T_i(b)$ in constant time. The index of each array T_a is stored in array T, which can be accessed in constant time for a given log-size vector a. Thus, we can calculate the product of two parts a and b of size $\log n$ in constant time using the table T. 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. By 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 M_3 in $O(|P|^3 n^3/\log(|P|n))$ time on a word RAM with word size $w = \theta(\log|P|n)$, where $n = |V|$. Moreover, maintaining of the dynamic transitive closure dominates the cost of the algorithm.*

Proof. Let $|\mathcal{A}|$ be the 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 $\sum\limits_i T(\mathcal{M}_1 \otimes \mathcal{A}_2^{(i)}) = |\mathcal{M}_1| \sum\limits_i |\mathcal{A}_2^{(i)}| = (|N| + |\Sigma|)|P|^2 \sum\limits_i |\mathcal{A}_2^{(i)}| = O((|N| + |\Sigma|)^2 |P|^2 n^2)$.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that C_3 has a size of the Kronecker product of $\mathcal{M}_1 \otimes \mathcal{M}_2$, which is equal to $dim(\mathcal{M}_1)n \times dim(\mathcal{M}_1)n = |P|n \times |P|n$ so no more than $|P|^2 n^2$ edges will be added during all iterations of the Algorithm. Checking 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 the loop in line 19 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 $O((|N| + |\Sigma|)^2 |P|^2 n^2) + O(|P|^2 n^2) + O(|P|^2 n^2 \log(|P|n)) + O(|P|^3 n^3/\log(|P|n)) + O(|P|^2 n^2) = O(|P|^3 n^3/\log(|P|n))$. □ □

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 exists 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 by **?** 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 **?** 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 the 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, and has no *recursive calls*. The adjacency matrix of the machine is built over Σ only. Therefore, during the calculation of 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 M_3 already stores data about all paths derivable from nonterminals. This data can be used to construct these paths. However, the set of such paths can be infinite. From a practical perspective, it is

necessary to use lazy evaluation or limit the resulting set of paths in some other way. For example, one can try to query some fixed number of paths, query paths of fixed maximum length, or just query a single path. The problem of paths enumeration, namely how to enumerate paths with the smallest delay, may be relevant here.

The most natural way to use the created index is to query paths between the specified vertices derivable from the specified nonterminal. To do so, we provide a function getPaths(v_s, v_f, N), where v_s is a start vertex of the graph, v_f — the final vertex, and N is a nonterminal. Implementation of this function is presented in Listing 2.

Paths extraction is implemented as two functions. The entry point is getPaths(v_s, v_f, N). This function returns a set of the paths in the input graph 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^0, v_s) to vertices of the form (q_N^f, v_f) in the resulting Kronecker product M_3, where q_N^0 is an initial state of RSM for N and q_N^f is one of the final states. For this reason, we provide the function genPaths$((s_i, v_i), (s_j, v_j))$. This function explores the graph corresponding to the resulting Kronecker product M_3 in a breadth-first manner and return all paths from (s_i, v_i) to (s_j, v_j). For each path, we also store the vector q that indicates which vertex is current now. We find the next vertices using the Boolean vector-matrix multiplication in line 26 of the algorithm from Listing 2. After that, we add new edges to our paths in lines 31 and 34. If we reach the vertex (s_j, v_j) then we can add the collected path to the resulting set (see lines 24-25). The paths constructed by genPaths is used to construct the corresponding paths in the input graph. Each edge $((s_i, v_i), (s_j, v_j))$ corresponds to set of paths in the input graph. This set is computed in line 13 and is used as subpaths for constructing the resulting paths in line 14. Note that in lines 14, 31, and 34 we use the operation $\cdot$ which naturally generalizes the path concatenation operation by constructing all possible concatenations of path pairs from the given two sets. Finally, if a single-edge subpath is labeled by a terminal, then the corresponding edge should be added to the result and if the label is nonterminal, getPaths should be used to restore paths.

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

3.3 An example

In this section, we introduce a detailed example to demonstrate the steps taken by the proposed algorithms. Namely, consider the graph $\mathcal{G}$ presented in Figure 1 and the RSM R presented in Figure 3.

In the first step, we represent both the graph and RSM as a set of Boolean matrices. Notice that we should add a new empty matrix M_2^S to $\mathcal{M}_2$, where edges labeled by S will be added at the time of the computation.

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

The first iteration. First of all, we compute the Kronecker product of the $\mathcal{M}_1$ and $\mathcal{M}_{2,(0)}$ matrices and collapse the result to the single Boolean matrix $M_{3,(1)}$. For the sake of simplicity, we provide only $M_{3,(1)}$, which is evaluated as follows.

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

	(0,0)	(0,1)	(1,0)	(1,1)	(2,0)	(2,1)	(3,0)	(3,1)
(0,0)	.	.	.	1	.	.	.	.
(0,1)	.	.	1	.	.	.	.	.
(1,0)	.	.	.	.	.	.	.	.
(1,1)	.	.	.	.	.	.	.	1
(2,0)	.	.	.	.	.	.	.	.
(2,1)	.	.	.	.	.	.	.	1
(3,0)	.	.	.	.	.	.	.	.
(3,1)	.	.	.	.	.	.	.	.

As far as the input graph has no edges with label S, 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 4. Then, the transitive closure evaluation result, stored in the matrix $C_{3,(1)}$, introduces one new path of length 2 (the thick edges 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

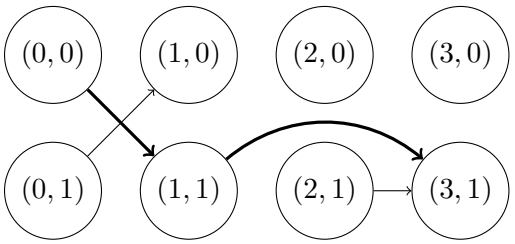

Figure 4.: The Kronecker product graph of RSM R and the input graph $\mathcal{G}$ (edges which form new paths are thick)

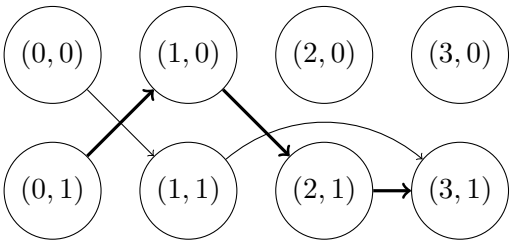

Figure 5.: The Kronecker product graph of RSM R and the updated graph $\mathcal{G}$ (edges which form new paths are thick)

the start and final states of 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 the respective word is derivable from S in the R execution flow.

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

The second iteration. The modified graph Boolean adjacency matrices contain an 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 the path between vertices $(0,1)$ and $(3,1)$ connects the start and final states in the automaton, the edge $(1, S, 1)$ is added to the resulting graph.

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

	(0,0)	(0,1)	(1,0)	(1,1)	(2,0)	(2,1)	(3,0)	(3,1)
(0,0)	.	.	.	1	.	.	.	.
(0,1)	.	.	1	.	.	.	.	.
(1,0)	.	.	.	.	.	**1**	.	.
(1,1)	.	.	.	.	.	.	.	1
(2,0)	.	.	.	.	.	.	.	.
(2,1)	.	.	.	.	.	.	.	1
(3,0)	.	.	.	.	.	.	.	.
(3,1)	.	.	.	.	.	.	.	.

The result graph is presented in Figure 6.

No more edges will be added to the graph $\mathcal{G}$ at the last iteration. However, the new edge $(1,1) \to (2,1)$ will be added to the resulting Kronecker product, and the transitive closure evaluation introduces one new path $(0,0) \to (3,1)$ that connects the start and final states in the automaton (see Figure 7). At this point, the index creation is finished. One can use it to answer reachability queries, but it also can be used to restore paths for some reachable

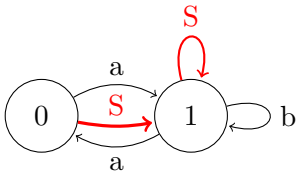

Figure 6.: The result graph $\mathcal{G}$

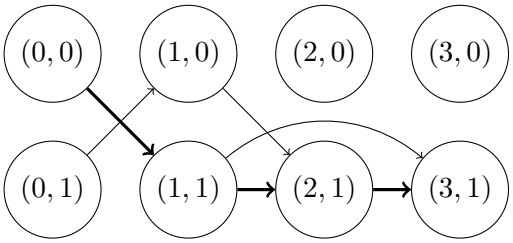

Figure 7.: The Kronecker product graph of RSM R and the final graph $\mathcal{G}$ (edges which form new paths are thick)

vertices. The resulting Kronecker product matrix M_3, or so-called *index*, can be used for it. For example, we can restore paths from vertex 1 to vertex 1 derived from S in the resulting graph.

To get these paths we should call `getPaths(1, 1, S)` function. A partial trace of this call is presented in Figure 8. First, we query paths for all possible start and final states of the machine for the provided graph vertices. Since the component state machine with label S in the example RSM has the single final state, the function `genPaths` is called with the arguments $(0,1)$ and $(3,1)$. Note, that the values passed to the functions in the path extraction algorithm are the pairs of the machine state and graph vertex, which uniquely identify a cell of the index matrix M_3. As a result, we get the set of all possible paths in the graph from 1 to 1 derived from S.

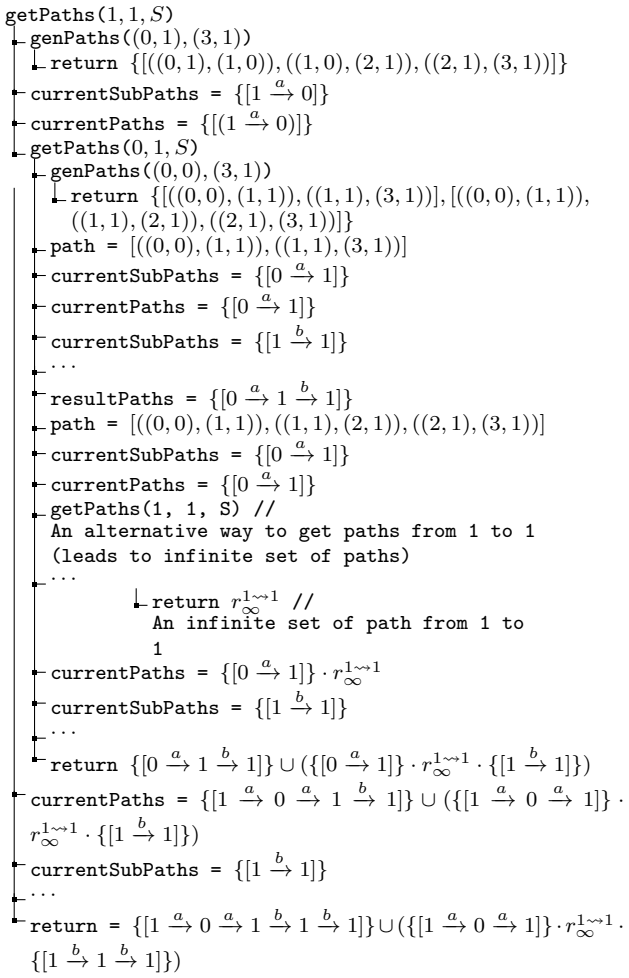

Figure 8.: Example of call stack trace

4. Evaluation

The goal of this evaluation is 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 Azimov's algorithm for CFPQ reachability. We also investigated the practical applicability of the paths extraction algorithm to 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 that 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 that represent real-world data from different areas and are frequently used for the evaluation of the graph querying algorithms. Namely, the dataset consists of three parts. The first part is the set of LUBM graphs[2] **?** 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*

[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.

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

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 **?** and in Table 5 in **?**. These query templates are presented in Table 2. We generate 10 queries for each template and each graph. The most 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 the scalability of the proposed algorithm.

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}	$a \cdot b \cdot (c \mid d \mid e)$

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 9. We can see that evaluation time depends on the query: there are queries evaluated 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 is comparable with the recent results for the same graphs and queries implemented on a distributed system over 10 nodes **?**,

[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.

[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.

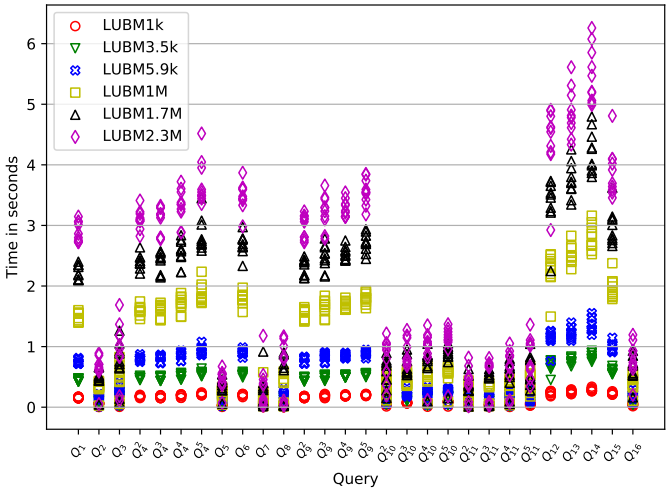

Figure 9.: Index creation time for LUBM graphs

while we use only one node. We conclude that our algorithm demonstrates reasonable performance to be applied to the real-world data analysis.

Index creation time for each query on the real-world graphs is presented in Figure 10. We can see that querying small graphs requires more time than querying bigger graphs in some cases. For example, consider Q^4_{10}: 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.

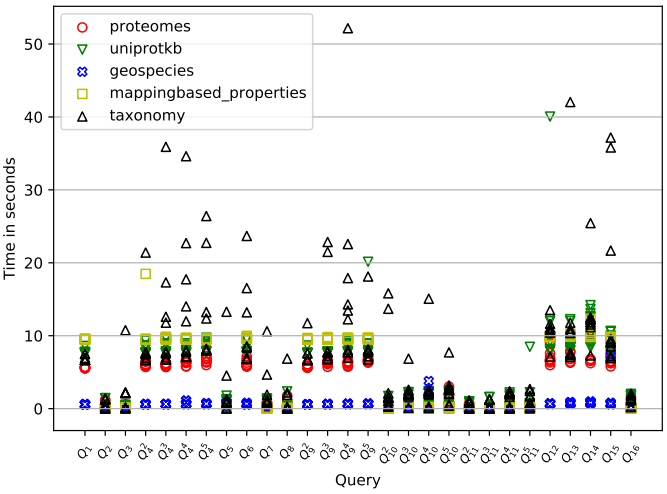

Figure 10.: Index creation time for real-world RDFs

4.2 CFPQ Evaluation

We evaluate the applicability of the proposed algorithm to CFPQ processing over real-world graphs on a number of classic cases and compare them with 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 trivial to compare our results with the state-of-the-art results provided by **?** (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 **?** 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 **?**. 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) **?**. 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 1.3 times faster (all memory aliases and *geospecies* with *Geo* query). Thus

[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	
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 2
crypto	3 464 970	5 976 774	—	—	—	678 408	2 3
drivers	4 273 803	7 415 538	—	—	—	858 568	2 8
fs	4 177 416	7 218 746	—	—	—	824 430	2 7

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	0.24	0.27	0.25	0.26	—	—	—	—
enzyme	0.03	0.04	0.02	0.01	—	—	—	—
geospecies	0.08	0.06	< 0.01	0.01	26.12	16.58	—	—
go-hierarchy	0.16	1.43	0.23	0.86	—	—	—	—
go	1.56	1.74	1.21	1.14	—	—	—	—
pathways	0.01	0.01	0.01	0.01	—	—	—	—
taxonomy	4.81	2.71	3.75	1.56	—	—	—	—
arch	—	—	—	—	—	—	262.45	195.51
crypto	—	—	—	—	—	—	267.52	195.54
drivers	—	—	—	—	—	—	1309.57	1050.78
fs	—	—	—	—	—	—	470.49	370.73

we can conclude that our solution is performant enough because Azimov's algorithm is the fastest known practical CFPQ algorithm **?** and the current version, which we use for evaluation, computes index only for single-path semantics, but our algorithm computes much more information (index for all paths) in a comparable time.

Best to our knowledge, the proposed algorithm is the first algorithm that provides information about all paths of interest (Azimov's algorithm computes information about only one path). The direct comparison with other solutions is impossible, and we just estimate the running time of our algorithm for a small number of cases. Namely, we extract all paths with length not greater than 20 edges between all pairs of vertices from indices created for graphs *go* and *eclass_514en* and query G_1. Paths extraction for one pair of vertices requires 2.64 seconds averaged over all pairs for *go* graph. The maximal time is 4699 seconds and 217 737 paths were extracted during this time. The average number of paths between two vertices is 184. For *eclass_514en* paths extraction for one pair of vertices requires 1.27 seconds averaged over all pairs. The maximal time is 8.04 seconds and only one path is extracted during this time. The average number of paths between two vertices is 3. We can see that paths can be extracted in a reasonable time, but a detailed analysis of paths extraction algorithm performance depends on graphs structure.

4.3 Conclusion

We conclude that the proposed algorithm is applicable to real-world data processing: the algorithm allows one to solve both the reachability problem and to extract paths of interest in a reasonable time. While index creation time (reachability query evaluation) is comparable with other existing solutions, the 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 to 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 **?**. In order to research practical limits and restrictions of RPQ, a number of high-performance RPQ algorithms were provided. For example, the derivative-based solution provided by **?**, which is implemented on top of the Pregel-based system, or the solution by **?**. But only a limited number of practical solutions provide the ability to restore paths of interest. A recent work by **?** provides a Pregel-based provenance-aware RPQ algorithm which utilizes a Glushkov's construction **?**. There is a lack of research of the applicability of linear algebra-based RPQ algorithms with paths-providing semantics.

On the other hand, many CFPQ algorithms with various properties were proposed recently. They employ the ideas of different parsing algorithms, such as CYK in works by **?** and **?**, (G)LR and (G)LL in works by **?**, **?**, **?**, **?**. Unfortunately, none of them has better than cubic time complexity in terms of the input graph size. The algorithm by **?** is, best to our knowledge, the first algorithm for CFPQ which is based on linear algebra. It was shown by **?** that this algorithm can be applied to real-world graph analysis problems, while **?** 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 **?** provide the first CFPQ algorithm with single path semantics based on linear algebra, **?** 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 the early stage. We agree with this point of view, and we only demonstrate the applicability of our solution to 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 exists a truly subcubic $O(n^{3-\varepsilon})$ algorithm for this problem. The question on the existence of a subcubic CFPQ algorithm was stated by **?**. A bit later **?** proposed the CFL reachability as a framework for interprocedural static code analysis. **?** 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 **?** and **?**. The slightly subcubic algorithm with $O(n^3/\log n)$ time complexity was provided by **?**. 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 **?**, and **?**. Other partial cases were investigated by **?**, **?**.

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

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 size blow-up. 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 determining whether a subcubic CFPQ exists. This is done by reducing the algorithm 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 demonstrate 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 preprocessing 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 **??**.

? obtained a conditional cubic lower bound for the CFL-reachability problem via the combinatorial BMM Conjecture. The combinatorial BMM Conjecture states that there is no truly subcubic $O(n^{3-\varepsilon})$ combinatorial algorithm for multiplying two $n \times n$ Boolean matrices. Such a lower bound holds only for *combinatorial* algorithms, leaving the possibility of an improvement via an algebraic algorithm. Can the above mentioned OMv Conjecture be used to establish conditional *algebraic*

lower bounds for the CFL-reachability (CFPQ evaluation) problem? Another powerful assumption relative to which many conditional lower bounds are proved is the strong exponential time hypothesis (SETH) introduced by **?**. Recently **?** showed that there cannot be a fine-grained reduction from SAT to CFL-reachability for a conditional lower bound stronger than n^{ω}, unless the nondeterministic strong exponential time hypothesis (NSETH) fails. Strong assumptions like 3SUM Conjecture by **?**, multiphase problem by **?** are variants of OMv conjecture or can be strengthened through it **?**. So the OMv Conjecture seems to be a good candidate for a proving conditional lower bound for the CFL-reachability (CFPQ evaluation) problem.

Also, an interesting task for the future is to improve the logarithmic factor in the obtained bound.

We also plan to improve bounds in partial cases for which dynamic transitive closure can be computed faster than in the general case. Examples of such cases are querying over planar graphs **?**, undirected graph, and others. In the case of planarity, it is interesting to investigate properties of the input graph and grammar which allow us to preserve planarity during query evaluation.

Note that both our algorithm and the state-of-the-art solutions have subcubic time complexity in terms of the grammar size **?????**. However, our approach does not require the input grammar to be translated to the Chomsky Normal Form and thus avoids the quadratic blow-up in its size which leads to a better performance in practice.

An important task for future research is a detailed investigation of the paths extraction algorithm. **?** provides a theoretical investigation of the 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 a practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work, we evaluate the naive implementation in which transitive closure is recalculated from scratch on each iteration. It is shown by **?** that some advanced algorithms for dynamic transitive closure can be efficiently implemented. Can one of these algorithms be efficiently parallelized and utilized in the proposed algorithm?

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

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

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph[8] **?** 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 **?** to integrate Azimov's algorithm to RedisGraph with minimal effort.

[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.

Listing 2 Paths extraction algorithm

```
1: M_3 ← the result of index creation algorithm: final Kronecker product
2: R ← recursive automata for the input RSM
3: M_1 ← the set of adjacency matrices for R
4: M_2 ← the set of adjacency matrices of the final graph
5: function getPaths(v_s, v_f, N)
6:     q_N^0 ← Start state of automata for N
7:     F_N ← Final states of automata for N
8:     paths ← ⋃_{q_N^f ∈ F_N} genPaths((q_N^0, v_s), (q_N^f, v_f))
9:     resultPaths ← ∅
10:    for path ∈ paths do
11:        currentPaths ← ∅
12:        for ((s_i, v_i), (s_j, v_j)) ∈ path do
13:            currentSubPaths ←{(v_i, t, v_j) | M_2^t[v_i, v_j] ∧ M_1^t[s_i, s_j]}
                                 ∪ ⋃_{{N | M_2^N[v_i,v_j] ∧ M_1^N[s_i,s_j]}}
               getPaths(v_i, v_j, N)
14:            currentPaths ← currentPaths · currentSubPaths  ▷ Concatenation of paths
15:        end for
16:        resultPaths ← resultPaths ∪ currentPaths
17:    end for
18:    return resultPaths
19: end function
20: function genPaths((s_i, v_i), (s_j, v_j))
21:    q ← vector of zeros with size dim(M_3)  ▷ Vector for indicating the current vertex
22:    q[s_i dim(M_2) + v_i] ← 1
23:    resultPaths ← ∅
24:    supposedPaths ← {([ ], q)} ▷ Set of pairs: path and the vector for current vertex
25:    while supposedPaths is changing do
26:        for (path, q) ∈ supposedPaths do
27:            if q[s_j dim(M_2) + v_j] = 1 then
28:                resultPaths ← resultPaths ∪ path
29:            end if
30:            q ← q · (M_3)^T  ▷ Boolean vector-matrix multiplication
31:            Remove (path, q) from supposedPaths
32:            for j such that q[j] = 1 do
33:                pathNew ← path
34:                if path is empty path then
35:                    pathNew ← pathNew · [((s_i, v_i), (⌊j/dim(M_2)⌋, j mod dim(M_2)))]
36:                else
37:                    (s_k, v_k) ← the last vertex of path
38:                    pathNew ← pathNew · [((s_k, v_k), (⌊j/dim(M_2)⌋, j mod dim(M_2)))]
39:                end if
40:                qNew ← vector of zeros with size dim(M_3)
41:                qNew[j] ← 1
42:                Add (pathNew, qNew) to supposedPaths
43:            end for
44:        end for
45:    end while
46:    return resultPaths
47: end function
```