

<sup>1</sup>Specification of regular constraints in SPARQL propert paths: <https://www.w3.org/TR/sparql11-property-paths/>. Access date: 07.07.2020.

problem nowadays, and new algorithms and solutions are being created [18, 23].

At the same time, utilization of more powerful languages, namely context-free languages, gain popularity in the last few years. *Context-Free Path Querying* problem (CFPQ) was introduced by Mihalis Yannakakis in 1990 in [24]. A number of different algorithms was proposed since that time, but recently, in [15] Jochem Kuijpers et al. show that state-of-the-art CFPQ algorithms are not performant enough to be used in practice. This fact motivates to find new algorithms for CFPQ.

One of the promising ways to achieve high-performance solutions for graph analysis problems is to reduce graph algorithms to linear algebra. This way, the description of basic linear algebra primitives, the GraphBLAS [13] API, was proposed. Solutions that use libraries that implement this API, such as SuiteSparse [9] and CombBLAS [7], show that reduction to linear algebra is a way to utilize high-performance parallel and distributed computations for graph analysis.

Rustam Azimov in [3] shows how to reduce CFPQ to matrix multiplication. Later, in [17] and [21], it was shown that utilization of appropriate libraries of linear algebra for Azimov's algorithm implementation allows one to get practical solution for CFPQ. But Azimov's algorithm requires transforming the input grammar to Chomsky Normal Form, which leads to the grammar size increase, thus worsen performance especially for regular queries and complex context-free queries.

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

Moreover, we show that this algorithm is a way to attack a long-standing problem of subcubic CFPQ. The best-known result for the general case is an  $O(n^3/\log n)$  algorithm of Swarat Chaudhuri [8]. Also, there are solutions for partial cases. For example, there is a truly subcubic algorithm for 1-Dyck language proposed by Phillip Bradford [6]. But this solution cannot be generalized to arbitrary CFPQs. So, in our knowledge, there is no truly subcubic general algorithm for CFPQs. In this work we show that incremental transitive closure is a bottleneck on the way to get subcubic time complexity for CFPQ.

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

- (1) We rethink and improve tensor-product-based algorithm for CFPQ. First of all, we reduce this algorithm to operations over Boolean matrices. All paths semantics. Previous matrix-based solution only single path. For both regular and context-free path queries. Correctness and time complexity.
- (2) We demonstrate interconnection between CFPQ and incremental transitive closure. Conjecture on sublinear incremental transitive closure and subcubic CFPQ. We show that incremental transitive closure is a bottleneck on the way to get subcubic CFPQ algorithm.
- (3) By using existing results we show how to get slightly subcubic algorithm for general case, and subcubic combinatorial algorithm for partial cases. This criterion is output-sensitive, so it is not practical, but open a theoretical way to find more subclass with subcubic complexity.
- (4) We implement the described algorithm and evaluate it on real-world data. RPQ, CFPQ. Results show that !!!

## 2 PRELIMINARIES

In this section we introduce basic notation and definitions from graph theory and formal language theory which are used in our work.

### 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 should give both language and grammar definitions.

First of all, we introduce edge-labeled digraph  $\mathcal{G} = \langle V, E, L \rangle$ , where  $V$  is a finite set of vertices,  $E \subseteq V \times L \times V$  is a finite set of edges,  $L$  is a finite set of edge labels. Note that one can always introduce bijection between  $V$  and  $Q = \{0, \dots, |V| - 1\}$ , thus in our work we guess that  $V = \{0, \dots, |V| - 1\}$ .

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

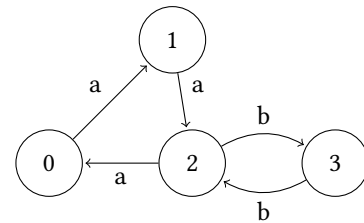


Figure 1: The example of input graph  $\mathcal{G}$

Each edge-labeled graph can be represented as an adjacency matrix  $M$ : square  $|V| \times |V|$  matrix, such that  $M[i, j] = \{l \mid e = (i, l, j) \in E\}$ . Adjacency matrix  $M_2$  of the graph  $\mathcal{G}$  is

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

In our work we use decomposition of the adjacency matrix to a set of Boolean matrices:

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

Matrix  $M_2$  can be represented as a set of two Boolean matrices  $M_2^a$  and  $M_2^b$  where

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

In this way we reduce operations which are necessary for our algorithm from operations over custom semiring (over edge labels) to operations over a Boolean semiring.

Also, we should define the path in the graph and the word formed by the path.

**Definition 2.1.** Path  $\pi$  in the graph  $(G) = \langle V, E, L \rangle$  is a sequence  $e_0, e_1, \dots, e_{n-1}$ , 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 path from  $v$  to  $u$  as  $v\pi u$ .

**Definition 2.2.** The word formed by a path

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

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

The next part is a definitions from the formal language theory.

**Definition 2.3.** A language  $\mathcal{L}$  over a finite alphabet  $\Sigma$  is a subset of all possible sequences formed by symbols from the alphabet:  $\mathcal{L}_\Sigma = \{\omega \mid \omega \in \Sigma^*\}$ .

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

**Definition 2.4.** To evaluate context-free path query with reachability semantics is to construct a set of pairs of vertices  $(v_i, v_j)$  such that there exists a path  $v_i \pi v_j$  in  $\mathcal{G}$  which forms the word from the given language:

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

**Definition 2.5.** To evaluate context-free path query with all paths semantics is to construct a set of path  $\pi$  in  $\mathcal{G}$  which forms the word from the given language:

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

Note that  $\Pi$  can be infinite, thus in practice, we should provide a way of enumerating such paths with reasonable complexity, instead of explicit construction of the  $\Pi$ .

## 2.2 Regular Path Queries and Finite State Machine

The first case of language-constrained path querying is *Regular Path Querying* (RPQ): the language  $L$  is a regular language. This case is widely spread in practice [?].

Usual way to specify regular languages is *regular expressions*. We use the following definition of regular expressions.

**Definition 2.6.** Regular expression (and regular language) over alphabet  $\Sigma$  can be inductively defined as follows.

- $\emptyset$  (empty language) is regular expression
- $\varepsilon$  (empty string) is regular expression
- $a_i \in \Sigma$  is regular expression
- if  $R_1$  and  $R_2$  are regular expressions, then  $R_1 \mid R_2$  (alternation),  $R_1 \cdot R_2$  (concatenation),  $R_1^*$  (Kleene star) are also regular expressions.

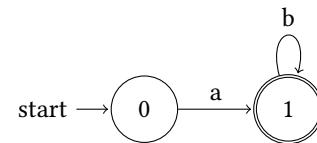
For example, one can specify regular expression  $R_1 = ab^*$  to find paths in the graph  $\mathcal{G}$  (fig. 1). Expected result is set of paths which start with  $a$ -labeled edge and contain zero or more  $b$ -labeled edges after that.

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

**Definition 2.7.** Deterministic Finite-State Machine  $T$  is a tuple  $\langle \Sigma, Q, Q_s, Q_f, \delta \rangle$  where

- $\Sigma$  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,
- $\delta : Q \times \Sigma \rightarrow Q$  is a transition function.

It is well known, that every regular expression can be converted to deterministic FSM without  $\varepsilon$ -transitions. To do it one can use [11]. In our work we use FSM as a representation of RPQ. FSM can be naturally represented by a directed edge-labeled graph:  $V = Q, L = \Sigma, E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$ , where some vertices have special markers to specify start and final states. Example of graph-style representation of FSM  $T_1$  for the regular expression  $R_1$  is presented in Figure 2.



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

As a result, FSM also can be represented as a set of Boolean adjacency matrices  $\mathcal{M}$  with additional information about

start and final vertices. Such representation of  $T_1$  is

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

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

### 2.3 Context-Free Path Querying and Recursive State Machines

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

**Definition 2.8.** Context-free grammar  $G = \langle \Sigma, N, S, P \rangle$  where  $\Sigma$  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, and  $P$  is a finite set of productions (grammar rules) of form  $N_i \rightarrow \alpha$  where  $N_i \in N$ ,  $\alpha \in (\Sigma \cup N)^*$ .

**Definition 2.9.** The sequence  $\omega_2 \in (\Sigma \cup N)^*$  is derivable from  $\omega_1 \in (\Sigma \cup N)^*$  in one derivation step, or  $\omega_1 \rightarrow \omega_2$ , in the grammar  $G = \langle \Sigma, N, S, P \rangle$  iff  $\omega_1 = \alpha N_i \beta$ ,  $\omega_2 = \alpha \gamma \beta$ , and  $N_i \rightarrow \gamma \in P$ .

**Definition 2.10.** 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  $(\rightarrow)$ .

Thus, one can use the grammar  $G_1 = \langle \{a, b\}, \{S\}, S, \{S \rightarrow a b; S \rightarrow a S b\} \rangle$  to find paths which form words in the language  $\mathcal{L}(G_1) = \{a^n b^n \mid n > 0\}$  in the graph  $\mathcal{G}$  (fig. 1).

Regular expressions can be transformed to a FSM, and a context free grammar can be transformed to *Recursive State Machine (RSM)* (also known as recursive networks [? ], recursive automata [? ], !!!) in the similar way. In our work we use the following definition of RSM.

**Definition 2.11.** A recursive state machine  $R$  over a finite alphabet  $\Sigma$  is defined as a tuple of elements  $(M, m, \{C_i\}_{i \in M})$ , where:

- $M$  is a finite set of labels of boxes.
- $m \in M$  is an initial box label.
- Set of *component state machines* or *boxes*, where  $C_i = (\Sigma \cup M, Q_i, q_i^0, F_i, \delta_i)$ :
  - $\Sigma \cup M$  is a set of symbols,  $\Sigma \cap M = \emptyset$
  - $Q_i$  is a finite set of states, where  $Q_i \cap Q_j = \emptyset, \forall i \neq j$
  - $q_i^0$  is an initial state for  $C_i$

- $F_i$  is a set of final states for  $C_i$ , where  $F_i \subseteq Q_i$
- $\delta_i : Q_i \times (\Sigma \cup M) \rightarrow Q_i$  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* [1]. A box works almost the same way as a 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 on input symbols reading. The pair  $(q_i, S)$ , where  $q_i$  is current state for box  $C_i$  and  $S$  is stack of *return states*, describes execution configurations.

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

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

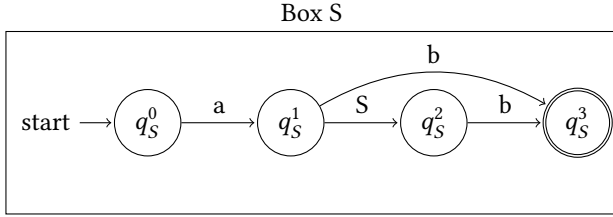
Some input sequence of the symbols  $a_1 \dots a_n$ , which forms some input word, is accepted, if machine reaches configuration  $(q, \langle \rangle)$ , where  $q \in F_m$ . It is also worth noting that the RSM makes nondeterministic transitions, without reading the input character when it *calls* some component or makes a *return*.

According to [1], recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages [11], it is clear that RSMs are equivalent to context-free languages. Thus RSMs suit to encode query grammars. Any CFG can be easily converted to an RSM with one box per nonterminal. The box which corresponds to a nonterminal  $A$  is constructed using the right-hand side of each rule for  $A$ .

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

Since  $R$  is a set of FSMs, it is useful to represent  $R$  as an adjacency matrix for the graph where vertices are states from  $\bigcup_{i \in M} Q_i$  and edges are transitions between  $q_i^a$  and  $q_i^b$  with label  $l \in \Sigma \cup M$ , if  $\delta_i(q_i^a, l) = q_i^b$ . An example of such adjacency matrix  $M_R$  for the machine  $R$  is provided in section ??.



**Figure 3: The recursive state machine  $R$  for grammar  $G$**

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

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

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

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

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

## 2.4 Graph Kronecker Product and Machines Intersection

First of all, we introduce classical Kronecker product definition, describe graph Kronecker product and its relation to Boolean matrices algebra, RSM and FSM intersection.

**Definition 2.12.** 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  $C = A \otimes B$ , where:

- $C$  has size  $m_1 * m_2 \times n_1 * n_2$
- $C[u * m_1 + v, n_1 * p + q] = A[u, p] \cdot B[v, q]$

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  $A[i, j] \cdot B$  (scalar to matrix).

**Definition 2.13.** 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 nice property, that if some path  $(u, v)\pi(p, q)$  exists in the result graph then paths  $u\pi_1p$  and  $v\pi_2q$  exist in the input graphs, and  $\omega(\pi) = \omega(\pi_1) = \omega(\pi_2)$ . These paths  $\pi_1$  and  $\pi_2$  could be easily found from  $\pi$  by its definition.

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

**Definition 2.14.** 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 a some graph  $\mathcal{G}$ , where:

- $M$  has size  $m_1 * m_2 \times n_1 * n_2$
- $M[u * m_1 + v, n_1 * p + q] = M_1[u, p] \cap M_2[v, q]$

By the definition, the Kronecker product for adjacency matrices gives an adjacency matrix with the same set of edges as in the resulting graph in the Def. 2.13. Thus,  $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$ , where  $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$ .

**Definition 2.15.** Given two FSMs  $T_1 = \langle \Sigma, Q^1, Q_S^1, Q_F^1, \delta^1 \rangle$  and  $T_2 = \langle \Sigma, Q^2, Q_S^2, Q_F^2, \delta^2 \rangle$ . The intersection of this two machines is a new FSM  $T = \langle \Sigma, Q, Q_S, Q_F, \delta \rangle$ , where:

- $Q = Q^1 \times Q^2$
- $Q_S = Q_S^1 \times Q_S^2$
- $Q_F = Q_F^1 \times Q_F^2$
- $\delta : Q \times \Sigma \rightarrow Q, \delta(\langle q_1, q_2 \rangle, s) = \langle q_1', q_2' \rangle,$   
if  $\delta(q_1, s) = q_1'$  and  $\delta(q_2, s) = q_2'$

According to [11], the above definition of the FSM intersection allows to construct the new machine with the following property:  $L(T) = L(T_1) \cap L(T_2)$ .

The most computationally expensive part of such procedure is the  $\delta$  function construction for the new machine  $T$ . Using adjacency matrices decomposition for FSMs we can reduce the intersection to the Kronecker product of such

matrices over Boolean semiring at some extent, since the transition function  $\delta$  of the machine  $T$  in matrix form is exactly the same as the product result. More precisely:

*Definition 2.16.* Given two adjacency matrices  $\mathcal{M}_1$  and  $\mathcal{M}_2$  over Boolean semiring. The Kronecker product of these matrices is a new matrix  $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$ , defined as follows:

- $\mathcal{M} = \{M_1^a \otimes M_2^a \mid a \in \Sigma\}$
- The element-wise operation is *and* over Boolean values

Applying the Kronecker product theory for both the FSM and the edge-labeled directed graph, we can intersect this objects as shown in Def. 2.16, since the graph could be interpreted as an FSM with transitions matrix represented as the adjacency matrix.

In this work we show how to express RSM and FSM intersection in terms of 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 provides the ability to solve all-pairs CFPQ in all-paths semantics (according to Hellings [? ]) and consists of two the following parts.

- (1) Index creation. In the first step, the algorithm computes an index which contains information which is necessary to restore paths for specified pairs of vertices. This index can be used to solve the reachability problem without paths extraction. 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 computed at the previous step. As far as the set of paths can be infinite, all paths cannot be enumerated explicitly, and advanced techniques such as lazy evaluation are required for implementation. Anyway, a single path can be always extracted by using standard techniques.

We describe both these steps, prove correctness, and provide time complexity estimations. For the first step we firstly introduce naïve algorithm. After that we show how to achieve cubic time complexity by using dynamic transitive closure algorithm and demonstrate that this technique allow us to get truly subcubic CFPQ algorithm for planar graphs.

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

#### 3.1 Index Creation Algorithm

The *index creation* algorithm outputs the final adjacency matrix  $\mathcal{M}_2$  for the input graph with all vertices pairs, which are reachable through some nonterminal in the input grammar

$G$ , and the index matrix  $C_3$ , which allows to extract later 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 set of FSMs, it could be easily presented as adjacency matrix for some graph over labels set  $\Sigma \cup S$ . As shown in the Def. 2.16 we can apply Kronecker product from Boolean matrices to *intersect* the RSM and the input graph at some extent. But the RSM contains the nonterminal symbols from  $N$  with additional *recursive calls* logic, what requires *transitive closure* step for such symbols extraction.

Applying the Kronecker product and transitive closure theory together, we get the idea of the algorithm: iterative Kronecker product evaluation for the RSM and the input graph, followed by transitive closure, nonterminal extraction and the update of the graph adjacency matrix.

*3.1.1 Over Boolean Matrices Version.* Listing 1 shows main steps of the algorithm. The algorithm accepts context-free grammar  $G = (\Sigma, N, P)$  and graph  $\mathcal{G} = (V, E, L)$  as an input. An RSM  $R$  is created from the grammar  $G$ . Note, that  $R$  must have no  $\varepsilon$ -transitions.  $\mathcal{M}_1$  and  $\mathcal{M}_2$  are the Boolean adjacency matrices for the machine  $R$  and the graph  $\mathcal{G}$  correspondingly.

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

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

For the given  $\mathcal{M}'_3$  a  $C_3$  transitive closure matrix is evaluated by the corresponding function call. Then the algorithm iterates over cells of the  $C_3$ . For the pair of indices  $(i, j)$ , it computes  $s$  and  $f$  — the initial and final states in the recursive automata  $R$  which relate to the concrete  $C_3[i, j]$  of the closure matrix. If the given  $s$  and  $f$  belong to the same box

$B$  of  $R$ ,  $s = q_B^0$ , and  $f \in F_B$ , then  $getNonterminals()$  returns the respective nonterminal. Then for each such nonterminal the respective matrix of the graph adjacency matrix  $M_2$  is updated and a new edge as a Boolean value in the appropriate cell is added.

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

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

---

#### Listing 1 Kronecker product based CFPQ

---

```

1: function CONTEXTFREEPATHQUERYING( $G, \mathcal{G}$ )
2:    $R \leftarrow$  Recursive automata for  $G$ 
3:    $M_1 \leftarrow$  Boolean adjacency matrix for  $R$ 
4:    $M_2 \leftarrow$  Boolean adjacency matrix for  $\mathcal{G}$ 
5:   for  $s \in 0..dim(M_1) - 1$  do
6:     for  $S \in getNonterminals(R, s, s)$  do
7:       for  $i \in 0..dim(M_2) - 1$  do
8:          $M_2^S[i, i] \leftarrow \{1\}$ 
9:   while Matrix  $M_2$  is changing do
10:     $M_3 \leftarrow M_1 \otimes M_2$  ▷ Evaluate Kronecker product
11:     $M'_3 \leftarrow \bigvee_{M_3^a \in M_3} M_3^a$  ▷ Collapse to Boolean matrix
12:     $C_3 \leftarrow transitiveClosure(M'_3)$ 
13:     $n \leftarrow dim(M_3)$  ▷ Matrix  $M_3$  size =  $n \times n$ 
14:    for  $(i, j) \in [0..n - 1] \times [0..n - 1]$  do
15:      if  $C_3[i, j]$  then
16:         $s, f \leftarrow getStates(C_3, i, j)$ 
17:         $x, y \leftarrow getCoordinates(C_3, i, j)$ 
18:        for  $S \in getNonterminals(R, s, f)$  do
19:           $M_2^S[x, y] \leftarrow \{1\}$ 
20:  return  $M_2$ 

```

---



---

#### Listing 2 Help functions for Kronecker product based CFPQ

---

```

1: function GETSTATES( $C, i, j$ )
2:    $r \leftarrow dim(M_1)$  ▷  $M_1$  is adjacency matrix for automata  $R$ 
3:   return  $\lfloor i/r \rfloor, \lfloor j/r \rfloor$ 
4: function GETCOORDINATES( $C, i, j$ )
5:    $n \leftarrow dim(M_2)$  ▷  $M_2$  is adjacency matrix for graph  $\mathcal{G}$ 
6:   return  $i \bmod n, j \bmod n$ 

```

---

LEMMA 3.1. Let  $\mathcal{G} = (V, E, L)$  be a graph and  $G = (\Sigma, N, P)$  be a grammar. Let  $\mathcal{G}_k = (V, E_k, L \cup N)$  be graph and  $M_k$  its adjacency matrix of the execution some iteration  $k \geq 0$  of the algorithm. Then for each edge  $e = (m, S, n) \in E_k$ , where  $S \in N$ , the following statement holds:  $\exists m\pi n : S \rightarrow_G l(\pi)$ .

PROOF. (Proof by induction)

**Basis:** For  $k = 0$  and the statement of the lemma holds, since  $M_0 = M$ ,  $M$  where is adjacency matrix of the graph  $G$ . Non-terminals, which allow to derive  $\varepsilon$ -word, are also added at algorithm preprocessing step, since each vertex of the graph is reachable by itself through an  $\varepsilon$ -transition.

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

For the algorithm iteration  $p$  the Kronecker product  $K_p$  and transitive closure  $C_p$  are evaluated as described in the algorithm. By the properties of this operations, some edge  $e = ((s, m), (f, n))$  exists in the directed graph, represented by adjacency matrix  $C_p$ , if and only if  $\exists s\pi'f$  in the RSM graph, represented by matrix  $M_r$ , and  $\exists m\pi n$  in graph, represented by  $M_{p-1}$ . Concatenated symbols along the path  $\pi'$  form some derivation string  $v$ , composed from terminals and non-terminals, where  $v \rightarrow_G l(\pi)$  by the inductive assumption.

The new edge  $e = (m, S, n)$  will be added to the  $E_p$  only if  $s$  and  $f$  are initial and final states of some box  $B$  of the RSM corresponding to the non-terminal  $S_B$ . In this case, the grammar  $G$  has the derivation rule  $S_B \rightarrow_G v$ , by the inductive assumption  $v \rightarrow_G l(\pi)$ . Therefore,  $S_B \rightarrow_G l(\pi)$  and this completes the proof of the lemma. □

LEMMA 3.2. Let  $\mathcal{G} = (V, E, L)$  be a graph and  $G = (\Sigma, N, P)$  be a grammar. Let  $\mathcal{G}_k = (V, E_k, L \cup N)$  be graph and  $M_k$  its adjacency matrix of the execution some iteration  $k \geq 1$  of the algorithm. For any path  $m\pi n$  in graph  $\mathcal{G}$  with word  $l = l(\pi)$  if exists the derivation tree of  $l$  for the grammar  $G$  and starting non-terminal  $S$  with the height  $h \leq k$ , then  $\exists e = (m, S, n) : e \in E_k$ .

PROOF. (Proof by induction)

**Basis:** Show that statement of the lemma holds for the  $k = 1$ . Matrix  $M$  and edges of the graph  $\mathcal{G}$  contains only labels from  $L$ . Since the derivation tree of height  $h = 1$  contains only one non-terminal  $S$  as a root and only symbols from  $\Sigma \cup \varepsilon$  as leaves, for all paths, which form a word with derivation tree of the height  $h = 1$ , the corresponding nonterminals will be added to the  $M_1$  via preprocessing step and first iteration of the algorithm. Thus, the lemma statement holds for the  $k = 1$ .

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

For the algorithm iteration  $p$  the Kronecker product  $K_p$  and transitive closure  $C_p$  are evaluated as described in the algorithm. By the properties of this operations, some edge  $e = ((s, m), (f, n))$  exists in the directed graph, represented by adjacency matrix  $C_p$ , if and only if  $\exists s\pi_1f$  in the RSM

graph, represented by matrix  $M_{RSM}$ , and  $\exists m\pi n$  in graph, represented by  $M_{p-1}$ .

For any path  $m\pi n$ , such that exist derivation tree of height  $h < k$  for the word  $l(\pi)$  with root non-terminal  $S$ , there exists edge  $e = (m, S, n) : e \in E_k$  by inductive assumption.

Suppose, that exists derivation tree  $T$  of height  $h = p$  with the root non-terminal  $S$  for the path  $m\pi n$ . The tree  $T$  is formed as  $S \rightarrow a_1..a_d, d \geq 1$  where  $\forall i \in [1..d]$   $a_i$  is sub-tree of height  $h_i \leq p - 1$  for the sub-path  $m_i\pi_i n_i$ . By inductive hypothesis, there exists path  $\pi_i$  for each derivation sub-tree, such that  $m = m_1\pi_1 m_2..m_d\pi_d m_{d+1} = n$  and concatenation of these paths forms  $m\pi n$ , and the root non-terminals of this sub-trees are included in the matrix  $M_{p-1}$ .

Therefore, vertices  $m_i \forall i \in [1..d]$  form path in the graph, represented by matrix  $M_{p-1}$ , with complete set of labels. Thus, new edge between vertices  $m$  and  $n$  with the respective non-terminal  $S$  will be added to the matrix  $M_p$  and this completes the proof of the lemma.  $\square$

**THEOREM 3.3.** *Let  $\mathcal{G} = (V, E, L)$  be a graph and  $G = (\Sigma, N, P)$  be a grammar. Let  $\mathcal{G}_R = (V, E_R, L)$  be a result graph for the execution of the algorithm  $??$ . The following statement holds:  $e = (m, S, n) \in E_R$ , where  $S \in N$ , if and only if  $\exists m\pi n : S \rightarrow_G l(\pi)$ .*

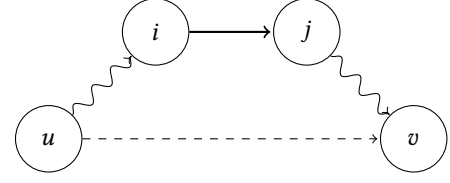
**PROOF.** This theorem is a consequence of the Lemma 3.1 and Lemma 3.2.  $\square$

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

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

**3.1.2 Application of Dynamic Transitive Closure.** In this subsection we show how to reduce the time complexity of the Algorithm 1 by avoiding redundant calculations.

It is easy to see that the most time-consuming steps in the Algorithm 1 are the Kronecker product and transitive closure computations. Recall that the matrix  $M_2$  is always changed in incremental manner i. e. elements (edges) are added to  $M_2$  (and are never deleted from it) on every iteration of the Algorithm 1. So one does not need to recompute the whole product or transitive closure if an appropriate data structure is maintained.



**Figure 4: The vertex  $j$  become reachable from the vertex  $u$  after the addition of edge  $(i, j)$ . Then the vertex  $v$  is reachable from  $u$  after inserting the edge  $(i, j)$  if  $v$  is reachable from  $j$ .**

To deal with the Kronecker product computation, we use the left-distributivity of the Kronecker product. Let  $A_2$  be a matrix with newly added elements and  $B_2$  be a matrix with the all previously found elements, such that  $M_2 = A_2 + B_2$ . Then by the left-distributivity of the Kronecker product we have  $M_1 \otimes M_2 = M_1 \otimes (A_2 + B_2) = M_1 \otimes A_2 + M_1 \otimes B_2$ . Notice that  $M_1 \otimes B_2$  is known and is already in the matrix  $M_3$  and its transitive closure also is already in the matrix  $C_3$ , because it was calculated on the previous iterations, so it is left to update some elements of  $M_3$  by computing  $M_1 \otimes A_2$ , which can be done in  $O(|A_2||M_1|)$  time, where  $|A|$  denotes the number of non-zero elements in a matrix  $A$ .

The fast computation of transitive closure can be obtained by using incremental dynamic transitive closure technique. We use an approach by Ibaraki and Katoh [12] to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices, which become reachable after insertion of the certain edge (see Figure 4 for details). The algorithm is presented in Listing 3 (we have slightly modified it to efficiently track new elements of the matrix  $C_3$ ).

---

**Listing 3** The dynamic transitive closure procedure

---

```

1: function ADD( $C_3, i, j$ )
2:    $n \leftarrow$  Number of rows in  $C_3$ 
3:    $C'_3 \leftarrow$  Empty matrix of size  $n \times n$ 
4:   for  $u \neq 0 \in \text{checkCondition}(C_3, i, j)$  do
5:      $\text{newReachablePairs}(C_3, C'_3, u, j)$ 
6:   return  $C'_3$ 
7: function CHECKCONDITION( $C_3, i, j$ )
8:    $A \leftarrow$  Empty array of size  $n$ 
9:   for  $u \in 0..n \mid u \neq j$  do  $\triangleright 1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0; 0 \wedge 1 = 1$ 
10:     $A[u] = C_3[u, j] \wedge C_3[u, i]$ 
11:   return  $A$ 
12: function NEWREACHABLEPAIRS( $C_3, C'_3, u, j$ )
13:    $C'_3[u, v] = C_3[u, v] \wedge C_3[j, v]$   $\triangleright 1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0;$ 
     $0 \wedge 1 = 1$ 

```

---

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



---

**Listing 4** Kronecker product based CFPQ using dynamic transitive closure
 

---

```

1: function CONTEXTFREEPATHQUERYING( $G, \mathcal{G}$ )
2:    $R \leftarrow$  Recursive automata for  $G$ 
3:    $M_1 \leftarrow$  Adjacency matrix for  $R$ 
4:    $M_2 \leftarrow$  Adjacency matrix for  $\mathcal{G}$ 
5:    $A_2 \leftarrow$  Adjacency matrix for  $\mathcal{G}$ 
6:    $C_3 \leftarrow$  The empty matrix
7:   for  $s \in 0..dim(M_1) - 1$  do
8:     for  $i \in 0..dim(M_2) - 1$  do
9:        $M_2[i, i] \leftarrow M_2[i, i] \cup getNonterminals(R, s, s)$ 
10:  while Matrix  $M_2$  is changing do
11:     $M'_3 \leftarrow M_1 \otimes A_2$ 
12:     $A_2 \leftarrow$  The empty matrix of size  $n \times n$ 
13:    for  $M'_3[i, j] \mid M'_3[i, j] = 1$  do
14:       $C_3[i, j] \leftarrow 1$ 
15:       $C'_3 \leftarrow \bigcup_{(i,j)} add(C_3, i, j) \triangleright$  Updating the transitive closure
16:       $C_3 \leftarrow C_3 + C'_3$ 
17:     $n \leftarrow dim(M_3)$ 
18:    for  $(i, j) \mid C'_3[i, j] \neq 0$  do
19:       $s, f \leftarrow getStates(C'_3, i, j)$ 
20:      if  $getNonterminals(R, s, f) \neq \emptyset$  then
21:         $x, y \leftarrow getCoordinates(C'_3, i, j)$ 
22:         $M_2[x, y] \leftarrow M_2[x, y] \cup getNonterminals(R, s, f)$ 
23:         $A_2[x, y] \leftarrow A_2[x, y] \cup getNonterminals(R, s, f)$ 
24:  return  $M_2$ 

```

---

**THEOREM 3.5.** Let  $\mathcal{G} = (V, E, L)$  be a graph and  $G = (\Sigma, N, P)$  be a grammar. The Algorithm 4 calculates a result graph  $\mathcal{G}_R = (V, E_R, L)$  in  $O(n^3)$  time.

**PROOF.** Let  $|A|$  be a number of non-zero elements in a matrix  $A$ . Consider the total time which is needed for computing the Kronecker products. The elements of the matrices  $A_2^{(i)}$  are pairwise distinct on every  $i$ -th iteration of the Algorithm therefore we have  $\sum_i T(M_1 \otimes A_2^{(i)}) = |M_1| \otimes \sum_i |A_2^{(i)}| = |M_1|O(n^2)$  operations in total.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that  $C_3$  has size of  $O(n^2)$  so no more than  $O(n^2)$  edges will be added during all iterations of the Algorithm. The function *checkCondition* from the Listing 3 takes  $O(n)$  time for every inserted edge  $(i, j)$ . Thus we have  $O(n^2n) = O(n^3)$  operations in total. The function *newReachablePairs* requires  $O(n)$  time for a given vertex  $u$ . This operation is performed for every pair  $(j, v)$  of vertices such that a vertex  $j$  became reachable from the vertex  $u$ . The vertex  $j$  become reachable from the vertex  $u$  (and accordingly the value of the matrix cell  $C_3[u, j]$  becomes 1 from 0) only once during the entire computation, so the function *newReachablePairs* will be executed at most  $O(n^2)$  times for every  $u$  and hence  $O(n^3)$  times in total for all vertices. Therefore  $O(n^3)$  operations are performed to maintain dynamic transitive closure during all iteration of the Algorithm 4.

Notice that the matrix  $C'_3$  contains only new elements, therefore  $C_3$  can be updated directly using only  $|C'_3|$  operations and hence  $O(n^2)$  operations in total. The same holds for cycle in line 18 of the Algorithm 4, because operations are performed only for non-zero elements of the matrix  $|C'_3|$ . Finally, we have that the time complexity of the Algorithm 4 is  $O(n^2) + O(n^3) + O(n^2) + O(n^2) = O(n^3)$ .  $\square$

**3.1.3 Speeding up by a factor of  $\log n$ .** In this subsection we use the Four Russians' trick to speed up the dynamic transitive closure algorithm from the Listing 3.

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

**PROOF.** Consider the function *checkCondition* from the Listing 3. Its operations are equivalent to the element-wise (Hadamard) product of two vectors of size  $n$ , where multiplication operation is denoted as  $\wedge$  and has the following properties:  $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$  and  $0 \wedge 1 = 1$ . The first vector represents reachability of a given vertex  $i$  from other vertices  $\{u_1, u_2, \dots, u_n\}$  of the graph and the second vector represents the same for a given vertex  $j$ . The function *newReachablePairs* also can be reduced to the computation of the Hadamard product of two vectors of size  $n$  for a given  $u_k$ . The first vector contains the information whether vertices  $\{v_1, v_2, \dots, v_n\}$  of the graph are reachable from a given vertex  $u_k$  and the second vector represents the same for a given vertex  $j$ . The element-wise product of two vectors can be calculated naively in time  $O(n)$  which gives the  $O(n^3)$  time for maintaining the transitive closure. Thus, the time complexity of the transitive closure can be reduced by speeding up element-wise product of two vectors of size  $n$ .

To achieve this goal, we use the Four Russians' trick. Split each vector into  $n/\log n$  parts of size  $\log n$ . Create a table  $S$  such that  $S(a, b) = a \wedge b$  where  $a, b \in \{0, 1\}^{\log n}$ . This takes a time  $O(n^2 \log n)$ , since there are  $2^{\log n} = n$  variants of boolean vectors of size  $\log n$  and hence  $n^2$  pairs of vectors  $(a, b)$  in total, and each component takes  $O(\log n)$  time. With table  $S$ , we can calculate product of two parts of size  $\log n$  in constant time. There are  $n/\log n$  such parts, so the element-wise product of two vectors of size  $n$  can be calculated in time  $O(n/\log n)$  with  $O(n^2 \log n)$  preprocessing. This gives us a dynamic transitive closure algorithm running in time  $O(n^3/\log n)$ : both of the functions *checkCondition* and *newReachablePairs* are evaluated no more than  $O(n^2)$  times during the whole computation, and each function calculates Hadamard product of two vectors in  $O(n/\log n)$  time.  $\square$

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

**COROLLARY 3.7.** *Let  $\mathcal{G} = (V, E, L)$  be a graph and  $G = (\Sigma, N, P)$  be a grammar. The result graph  $\mathcal{G}_R = (V, E_R, L)$  can be calculated in  $O(n^3/\log n)$  time.*

Subcubic for planar graphs using [20].

Cojecture on sublinear dynamic transitive closure and subcubic CFPQ.

### 3.2 Paths Extraction Algoritm

After index created one can enumerate all paths between specified vertices. Note, that 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 specified vertices derivable from specified nonterminal.

To do it we provide a function `GETPATHS( $v_s, v_f, N$ )`, where  $v_s$  is a start vertex of the graph,  $v_f$  – the final vertex, and  $N$  is a nonterminal. Implementation of this function is presented in Listing 5.

**Listing 5** Paths extraction algorithm

```

1:  $C_3 \leftarrow$  result of index creation algorithm: final transitive closure
2:  $M_1 \leftarrow$  the set of adjacency matrices of the input RSM
3:  $M_2 \leftarrow$  the set of adjacency matrices of the final graph
4: function GETPATHS( $v_s, v_f, N$ )
5:    $q_N^0 \leftarrow$  Start state of automata for  $N$ 
6:    $F_N \leftarrow$  Final states of automata for  $N$ 
7:    $res \leftarrow \bigcup_{f \in F_N} \text{GETPATHSINNER}((q_N, v_s), (f, v_f))$ 
8:   return  $res$ 
9: function GETSUBPATHS( $(v_i, s_i), (v_j, s_j), (v_k, s_k)$ )
10:   $l \leftarrow \{(v_i, t, v_k) \mid M_2^t[s_i, s_k] \wedge M_1^t[v_i, v_k]\}$ 
     $\cup \bigcup_{\{N \mid M_2^N[s_i, s_k]\}} \text{GETPATHS}(v_i, v_k, N)$ 
     $\cup \text{GETPATHSINNER}((v_i, s_i), (v_k, s_k))$ 
11:   $r \leftarrow \{(v_k, t, v_j) \mid M_2^t[s_k, s_j] \wedge M_1^t[v_k, v_j]\}$ 
     $\cup \bigcup_{\{N \mid M_2^N[s_k, s_j]\}} \text{GETPATHS}(v_k, v_j, N)$ 
     $\cup \text{GETPATHSINNER}((v_k, s_k), (v_j, s_j))$ 
12:  return  $l \cdot r$ 
13: function GETPATHSINNER( $(v_i, s_i), (v_j, s_j)$ )
14:   $parts \leftarrow \{(v_k, s_k) \mid C_3[(v_i, s_i), (v_k, s_k)] = 1 \wedge$ 
     $C_3[(v_k, s_k), (v_j, s_j)] = 1\}$ 
15:  return  $\bigcup_{(v_k, s_k) \in parts} \text{GETSUBPATHS}((v_i, s_i), (v_j, s_j), (v_k, s_k))$ 

```

Paths extraction is implemented as three mutually recursive functions. The entry point is  $\text{GETPATHS}(v_s, v_f, N)$ . This function returns a set of paths between  $v_s$  and  $v_f$  such that the word formed by the path is derivable from 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. To do it `GETPATHSINNER` $((s_i, v_i), (s_j, v_j))$  is used. This function finds

all possible vertices  $(s_k, v_k)$  which split path from  $(s_i, v_i)$  to  $(s_j, v_j)$  into two subpaths. After that, function `GETSUBPATHS` $((s_i, v_i), (s_j, v_j), (s_k, v_k))$  is used to compute corresponding subpaths. Each part of the path may be a single edge, or path with length more than one. In the second case `GETPATHSINNER` is used to restore corresponding paths. In the first case, the edge can be labeled by terminal or nonterminal. In the first case corresponding edge should be added to the result. In the second case, `GETPATHS` should be used to restore paths.

Note, that, first of all, we assume that sets are computed lazily. It is necessary to work correctly in the case of an infinite number of paths. Second, we use a set of path as a result, so we did not check duplicated paths manually.

### 3.3 An example

In this section we introduce detailed example to demonstrate steps of the proposed algorithm. Our example is based on the classical worst case scenario introduced by Jelle Hellings in [? ]. Namely, let we have a graph  $\mathcal{G}$  presented in figure 1 and the RSM  $R$  presented in figure [? ].

First step we represent graph as a set of boolean matrices as presented in 1, and RSM as a set of boolean matrices, as presented in ?? . Note, that we should add new empty matrix  $M_2^S$  to  $M_2$ . After that we should iteratively compute  $M_1$  and  $C$ .

**First iteration.** As far as  $M_2^{S,0}$  is empty (no edges with label  $S$  in the input graph), then correspondent block of the Kronecker product will be empty.

$$M_3^1 = M_1^a \otimes M_2^{a,0} + M_1^b \otimes M_2^{b,0} + M_1^S \otimes M_2^{S,0} =$$

$$\begin{pmatrix} \begin{matrix} (0,0) \\ (0,1) \\ (0,2) \\ (0,3) \end{matrix} & \begin{matrix} (1,0) \\ (1,1) \\ (1,2) \\ (1,3) \end{matrix} & \begin{matrix} (2,0) \\ (2,1) \\ (2,2) \\ (2,3) \end{matrix} & \begin{matrix} (3,0) \\ (3,1) \\ (3,2) \\ (3,3) \end{matrix} \end{pmatrix}$$

Transitive closure calculation introduces one new path of length 2 (respective cell is filled).

[illegible]



- (2) Compare other CFPQ algorithms and our algorithms
- (3) Investigate effect of grammar optimization

## 5.1 RPQ

In order to do smthng....

Dataset description, tools selection.

### 5.1.1 Dataset. Dataset for evaluation

We evaluate our solution on RPQs. We choose templates of the most popular RPQs which are presented in table ??  
We generate !!! queries for each template.

### 5.1.2 Results. Results of evaluation

Index creation.

Paths extraction

### 5.1.3 Conclusion.

## 5.2 CFPQ

Comparison with matrix-based algorithm.

### 5.2.1 Dataset. Dataset for evaluation. It should be CFPQ\_Data<sup>2</sup>.

Same-generation queries, memory aliases.

### 5.2.2 Results. Results of evaluation.

Index creation.

Paths extraction.

### 5.2.3 Conclusion.

## 5.3 Grammar transformation

On query optimization.

Memory aliases.

Synthetic???

## 6 RELATED WORK

Language constrained path querying is a wide area !!!!

RPQ algorithms: derivatives [? ], Glushkov [? ], etc.!!!! [? ] distributed, not linear algebra.

CFPQ algorithms: Hellings [? ], Bradford [6], Azimov [? ], Verbitskaya [? ], Ciro [? ], form static code analysis [? ],

Subcubic CFPQ: Bradford, Chatterjee, For trees — partial cases, RSM-s — 4 Russians method, Smth else?

Dynamic transitive closure [? ] [? ] [? ] [? ] algebraic, combinatorial, special graph types.

Implementation side. Linear algebra based approaches to evaluate queries (datalog, SPARQL, etc) [? ] Not focused on types of queries. Matrices, !!!! SPARQL !!!! Datalog !!!!

## 7 CONCLUSION AND FUTURE WORK

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

Also the algorithm opens a way to attack long-standing problem on subcubic CFPQ by reducing it to incremental transitive closure: *incremental transitive closure with  $O(n^{1-\epsilon})$  update time which returns all new reachable pairs on each insert, implies  $O(n^{3-\epsilon})$  CFPQ algorithm.* In this work we prove  $O(n^3/\log n)$  time complexity by providing  $O(n/\log n)$  transitive closure algorithm.

Thus, the first task for the future is to find truly sublinear algorithm for incremental transitive closure or, as a first step, to improve logarithmic factor. Also, it is interesting to get improved bounds in partial cases. For example, dynamic transitive closure for planar graphs can be supported !!!, and for undirected graph !!! Can we use these facts to provide better CFPQ algorithm for respective partial cases? In the case of planarity it is interesting to investigate properties of the input graph and grammar which allow to preserve planarity during query evaluation.

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

- (1) Can we provide more detailed analysis of CFPQ queries, than provided in [? ]?
- (2) Can we provide practical solution for CFPQ querying of streaming graphs?
- (3) Can we improve existing solutions [19] for RPQ?

From a practical perspective, it is necessary to analyze the usability of advanced algorithms for dynamic transitive closure. In the current work we evaluate naïve implementation in which transitive closure is recalculated on each iteration from scratch. In [10] it is shown that some of advanced algorithms for dynamic transitive closure can be efficiently implemented. Can one of these algorithms be efficiently parallelized and utilized in the proposed algorithm?

Also, it is necessary to evaluate GPGPU-based implementation. Experience in Azimov's algorithm shows that the utilization of GPGPUs allows one to improve performance because operations of linear algebra can be efficiently implemented on GPGPU [? ]. Moreover, for practical reason,

<sup>2</sup>!!!!

it is interesting to provide a multi-GPU version of the algorithm and to utilize unified memory, which, as shown in [?], is suitable for linear algebra based processing of out-of-GPGPU-memory data.

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 [?].

Finally, it is necessary to provide a multiple-source version of the algorithm and integrate it with a graph database. RedisGraph<sup>3</sup> [?] is a suitable candidate for this purpose. This database uses SuiteSparse—an implementation of GraphBLAS standard—as a base for graph processing. This fact allowed to Arseny Terkhov et.al. to integrate Azimov's algorithm to RedisGraph with minimal effort [?].

## REFERENCES

- [1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2001. Analysis of Recursive State Machines. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220.
- [2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. <https://doi.org/10.1145/3104031>
- [3] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, New York, NY, USA, Article 5, 10 pages. <https://doi.org/10.1145/3210259.3210264>
- [4] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '13)*. Association for Computing Machinery, New York, NY, USA, 175–188. <https://doi.org/10.1145/2463664.2465216>
- [5] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (May 2000), 809–837. <https://doi.org/10.1137/S0097539798337716>
- [6] P. G. Bradford. 2017. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*. 247–253. <https://doi.org/10.1109/UEMCON.2017.8249039>
- [7] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. <https://doi.org/10.1177/1094342011403516>
- [8] Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. <https://doi.org/10.1145/1328438.1328460>
- [9] Timothy A. Davis. 2018. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra.
- [10] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2020. Faster Fully Dynamic Transitive Closure in Practice. In *18th Symposium on Experimental Algorithms (SEA 2020)*. <http://eprints.cs.univie.ac.at/6345/>
- [11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [12] T. Ibaraki and N. Katoh. 1983. On-line computation of transitive closures of graphs. *Inform. Process. Lett.* 16, 2 (1983), 95 – 97. [https://doi.org/10.1016/0020-0190\(83\)90033-9](https://doi.org/10.1016/0020-0190(83)90033-9)
- [13] J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. <https://doi.org/10.1109/HPEC.2016.7761646>
- [14] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015*, Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.). Springer International Publishing, Cham, 3–18.
- [15] Jochem Kuijpers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaeker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management (SSDBM '19)*. ACM, New York, NY, USA, 121–132. <https://doi.org/10.1145/3335783.3335791>
- [16] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages. <https://doi.org/10.1145/2850413>
- [17] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. <https://doi.org/10.1145/3327964.3328503>
- [18] Maurizio Nolé and Carlo Sartiani. 2016. Regular Path Queries on Massive Graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. <https://doi.org/10.1145/2949689.2949711>
- [19] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. *ArXiv abs/2004.02012* (2020).
- [20] Sairam Subramanian. 1993. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms—ESA '93*, Thomas Lengauer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 372–383.
- [21] Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. <https://doi.org/10.1145/3398682.3399163>
- [22] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2960414.2960421>
- [23] Xin Wang, Simiao Wang, Yueqi Xin, Yajun Yang, Jianxin Li, and Xiaofei Wang. 2019. Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *World Wide Web* 23, 3 (Nov. 2019), 1465–1496. <https://doi.org/10.1007/s11280-019-00739-0>

<sup>3</sup>!!!

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