Context-Free Path Querying by Kronecker Product*

Egor Orachev[1], Ilya Epelbaum[1], Rustam Azimov[1,2], and Semyon Grigorev[1,2][0000−0002−7966−0698]

[1] St.Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, Russia, 199034
egor.orachev@gmail.com, iliyepelbaun@gmail.com, rustam.azimov19021995@gmail.com, s.v.grigoriev@spbu.ru

[2] JetBrains Research, Primorskiy prospekt 68-70, Building 1, St. Petersburg, Russia, 197374
semyon.grigorev@jetbrains.com

Abstract. Context-free path queries (CFPQ) extend the regular path queries (RPQ) by allowing context-free grammars to be used as constraints for paths. Algorithms for CFPQ are actively developed, but J. Kuijpers et al. have recently concluded, that existing algorithms are not performant enough to be used in real-world applications. Thus the development of new algorithms for CFPQ is justified. In this paper, we provide a new CFPQ algorithm which is based on such linear algebra operations as Kronecker product and transitive closure and handles grammars presented as recursive state machines. Thus, the proposed algorithm can be implemented by using high-performance libraries and modern parallel hardware. Moreover, it avoids grammar growth which provides the possibility for queries optimization.

Keywords: Context-free path querying · Graph database · Context-free grammars · CFPQ · Kronecker product · Recursive state machines.

1 Introduction

Language-constrained path querying [3], and particularly context-free path querying (CFPQ) [13], allows one to express constraints for paths in a graph in terms of context-free grammars. A path in a graph is included in a query result only if the labels along this path form a word that belongs to the language, generated by the query grammar. CFPQ is widely used in bioinformatics [12], graph databases querying [5, 10, 9], and RDF analysis [14].

CFPQ algorithms are actively developed but still suffer from poor performance [9]. The algorithm proposed by Rustam Azimov [2] is one of the most promising. This algorithm makes it possible to offload computational intensive parts to high-performance libraries for linear algebra, this way one can utilize modern parallel hardware for CFPQ. One disadvantage of this algorithm is that

* The research was supported by the Russian Science Foundation, grant №18-11-00100

a query grammar should be converted to a Chomsky Normal Form (CNF) which significantly increases its size. The performance of the algorithm depends on the grammar size, thus it is desirable to create the algorithm which does not modify the query grammar.

In this work, we propose a new algorithm for CFPQ which can be expressed in terms of matrix operations and does not require grammar transformation. This algorithm can be efficiently implemented on modern parallel hardware and it provides ways to optimize queries. The main contribution of this paper could be summarized as follows.

1. We introduce a new algorithm for CFPQ, which is based on the intersection of recursive state machines and can be expressed in terms of Kronecker product and transitive closure.
2. We provide a step-by-step example of the algorithm.
3. We provide an evaluation of the presented algorithm and its comparison with the matrix-based algorithm. The presented algorithm outperforms the previous matrix-based algorithm in the worst-case scenario, but further optimizations are required to make it applicable for real-world cases.

2 Recursive State Machines

In this section, we introduce recursive state machines (RSM) [1]. This kind of computational machine extends the definition of finite state machines and increases the computational capabilities of this formalism.

A recursive state machine R over a finite alphabet Σ 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 the component state machine C_i
  - F_i is a set of final states for C_i, where $F_i \subseteq Q_i$
  - δ_i is a transition function for C_i, where $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$

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 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 form 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 the input sequence, which is read as usual for FSA:

- $q' \leftarrow q_i^t, S' \leftarrow S$, where $q_i^t = \delta_i(q_i^k, a), q_i^k = q$
- $q' \leftarrow q_j^0, S' \leftarrow q_i^t \circ S$, where $q_i^t = \delta_i(q_i^k, j), q_i^k = q, j \in M$
- $q' \leftarrow q_i^t, S' \leftarrow S_{tail}$, where $S = q_i^t \circ S_{tail}, q_j^k = q, q_j^k \in F_j$

Some input sequence of the symbols $a_1 ... a_n$, which forms some input word, accepted, if machine reaches configuration $(q, \langle\rangle)$, where $q \in F_m$. It is also worth noting that the RSM makes not deterministic 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 [7], 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 $S \to aSb \mid ab$ is provided in figure 1.

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 4.1.

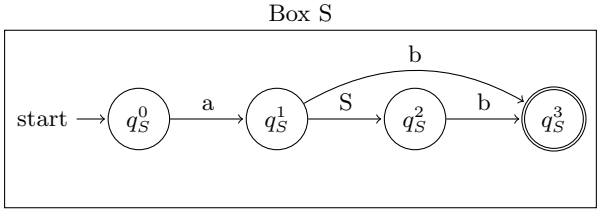


Fig. 1: The recursive state machine R for grammar G

3 Kronecker Product

In this section, we introduce the Kronecker product definition and its relation to the RSM and a directed graph intersection.

A directed labeled graph could be interpreted as an FSM, where transitions correspond to the labeled edges between vertices of the graph. As initial and final states could be chosen all vertices of the graph since this formal decision does not affect the algorithm idea. As shown in the section 2, an RSM is composed of a set of component state machines, which behave as a normal FSM. Notice, that these component machines are structurally independent, and actual communications are done only on *recursive calls* in time of the machine execution. It makes an intuition that one could apply automata theory to intersect an RSM with a directed graph.

The classical intersection algorithm of two deterministic FSMs is presented in [7]. The idea behind this algorithm is to create new FSM, which imitates the parallel work of both machines through an increase in the number of states and transitions. For any given FSMs $C_1 = (\Sigma \cup M, Q_1, q_1^0, F_1, \delta_1)$ and $C_2 = (\Sigma \cup M, Q_2, q_2^0, F_2, \delta_2)$ the intersection machine C defines as follows:

$C = (\Sigma \cup M, Q, q^0, F, \delta)$, where:

- States $Q = \{(q_1, q_2) \ : \ q_1 \in Q_1, q_2 \in Q_2\}$
- Final states $F = \{(q_1, q_2) \ : \ q_1 \in F_1, q_2 \in F_2\}$
- Initial state $q = (q_1^0, q_2^0)$
- Transition function $\delta((q_1, q_2), a) \rightarrow (\delta_1(q_1, a), \delta_2(q_2, a))$, where $a \in \Sigma \cup M$, only if $(\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are present.

A structurally similar procedure for constructing an intersection machine can be implemented using the Kronecker product for the corresponding transition matrices for FSMs if we provide a satisfying operation of elementwise multiplication, which allows further use of high-performance math libraries for intersection computation.

Since an RSM transitions matrix is composed as a blocked matrix of component state machines adjacency matrices, and a directed labeled graph could be presented as an adjacency matrix, it is convenient to implement the intersection of such objects via Kronecker product of the corresponding matrices. As an elementwise operation for one can employ the following function $\bullet : \Sigma \cup M \times \Sigma \cup M \rightarrow Boolean$, defined as follows:

- $A_1 \bullet A_2 = true$, if $A_1 \cap A_2 \neq \emptyset$
- $A_1 \bullet A_2 = false$, otherwise

An example of applying the Kronecker product is illustrated in the section 4.1.

The operation defined in this way allows us to construct an adjacency Boolean matrix for some directed graph, where a path between two vertices exists only if corresponding paths exist in some component state machine of the RSM and in source graph at the same time. This fact with transitive closure could be employed in order to extract all the path and components labels from M for context-free reachability problem-solving.

4 Kronecker Product Based CFPQ Algorithm

In this section, we introduce the algorithm for the computation of context-free reachability in a graph $\mathcal{G}$. The algorithm determines the existence of a path, which forms a sentence of the language defined by the input RSM R, between each pair of vertices in the graph $\mathcal{G}$. The algorithm is based on the generalization of the FSM intersection for an RSM, and an input graph. The idea of using the Kronecker product for the intersection of an RSM and a directed input graph is presented in the section 3.

Listing 1 shows the 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 ε-transitions. M_1 and M_2 are the adjacency matrices for the machine R and the graph $\mathcal{G}$ correspondingly.

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

The algorithm terminates when the matrix M_2 stops changing. Kronecker product of matrices M_1 and M_2 is evaluated for each iteration. The result is stored in M_3 as a Boolean matrix. For the given M_3 a C_3 matrix is evaluated by the $transitiveClosure()$ function call. The M_3 could be interpreted as an adjacency matrix for a directed graph with no labels, used to evaluate transitive closure in terms of classical graph definition of this operation. 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 non-terminal. If the condition holds then the algorithm adds the computed non-terminals to the respective cell of the adjacency matrix M_2 of the graph.

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 updated matrix M_2 which contains the initial graph $\mathcal{G}$ data as well as non-terminals from N. If a cell $M_2[i, j]$ for any valid indices i and j contains symbol $S \in N$, then vertex j is reachable from vertex i in grammar G for non-terminal S.

4.1 Example

This section provides a step-by-step demonstration of the presented algorithm. We consider the theoretical worst case for CFPQ time complexity, proposed by J.Hellings [5] as an example: the graph $\mathcal{G}$ is presented in Figure 2a and the context-free grammar G for a language $\{a^n b^n \mid n \geq 1\}$ is $S \to aSb \mid ab$.

Since the proposed algorithm processes grammar in the form of a recursive machine, we first provide RSM R in Figure 1. The initial box of the R is S, the initial state q_S^0 is (0), the set of final states $F_S = \{(3)\}$.

Listing 1 Kronecker product based CFPQ

```
1: function contextFreePathQuerying(G, 𝒢)
2:     R ← Recursive automata for G
3:     M1 ← Adjacency matrix for R
4:     M2 ← Adjacency matrix for 𝒢
5:     for s ∈ 0..dim(M1) − 1 do
6:         for i ∈ 0..dim(M2) − 1 do
7:             M2[i, i] ← M2[i, i] ∪ getNonterminals(R, s, s)
8:     while Matrix M2 is changing do
9:         M3 ← M1 ⊗ M2                          ▷ Evaluate Kronecker product
10:        C3 ← transitiveClosure(M3)
11:        n ← dim(M3)                            ▷ Matrix M3 size = n × n
12:        for i ∈ 0..n − 1 do
13:            for j ∈ 0..n − 1 do
14:                if C3[i, j] then
15:                    s, f ← getStates(C3, i, j)
16:                    if getNonterminals(R, s, f) ≠ ∅ then
17:                        x, y ← getCoordinates(C3, i, j)
18:                        M2[x, y] ← M2[x, y] ∪ getNonterminals(R, s, f)
19:    return M2
```

Listing 2 Help functions for Kronecker product based CFPQ

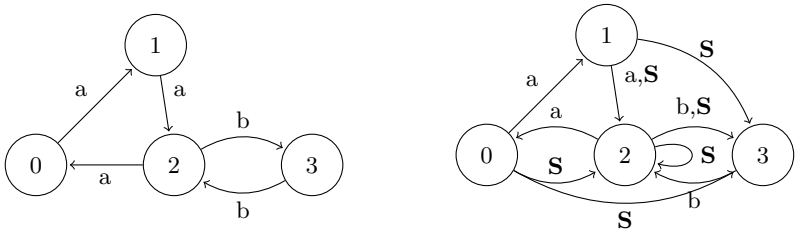
```
1: function getStates(C, i, j)
2:     r ← dim(M1)                    ▷ M1 is adjacency matrix for automata R
3:     return ⌊i/r⌋ , ⌊j/r⌋
4: function getCoordinates(C, i, j)
5:     n ← dim(M2)                    ▷ M2 is adjacency matrix for graph 𝒢
6:     return i mod n, j mod n
```

Adjacency matrices M_1 and M_2 for automata R and graph $\mathcal{G}$ respectively are initialized as follows:

$$M_1 = \begin{pmatrix} . & \{a\} & . & . \\ . & . & \{S\} & \{b\} \\ . & . & . & \{b\} \\ . & . & . & . \end{pmatrix}, \qquad M_2^0 = \begin{pmatrix} . & \{a\} & . & . \\ . & . & \{a\} & . \\ \{a\} & . & . & \{b\} \\ . & . & \{b\} & . \end{pmatrix}.$$

After the initialization in lines **2**–**4**, the algorithm handles ε-case. Because the machine R does not have ε-transitions and ε-word is not included in the language, lines **5**–**7** of the algorithm do not affect the input data.

Then the algorithm enters the while loop and iterates while matrix M_2 is changing. We provide both the values of the matrices M_3, C_3 at each algorithm step as well as how the matrix M_2 is updated. The current loop iteration number is provided in the superscript for each matrix. The first iteration is indexed as 1.

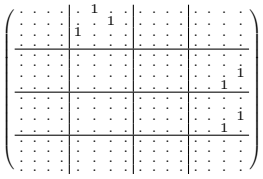
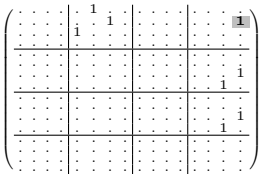


(a) The input graph $\mathcal{G}$

(b) The result graph $\mathcal{G}$

Fig. 2: The input and result graphs for example

During the first iteration the Kronecker product $M_3^1 = M_1 \otimes M_2^0$ and transitive closure C_3^1 are the following:

$M_3^1 =$  , $C_3^1 =$  .

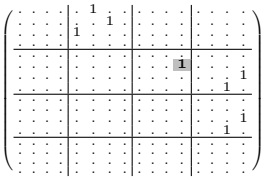
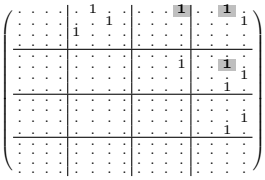
After the transitive closure evaluation $C_3^1[1, 15]$ contains a non-zero value. It means that the vertex with index 15 is accessible from the vertex with index 1 in the graph, represented by the adjacency matrix M_3^1.

Then the lines **14**–**18** are executed: the algorithm adds non-terminals to the graph matrix M_2^1. Because this step is additive we are only interested in the newly appeared values in the matrix C_3^1, such as value $C_3^1[1, 15]$, for which we get the following:

- Indices of the automata vertices $s = 0$ and $f = 3$, because value $C_3^1[1, 15]$ is located in the upper right matrix block $(0, 3)$.
- Indices of the graph vertices $x = 1$ and $y = 3$ are evaluated as the value $C_3^1[1, 15]$ indices relatively to its block $(0, 3)$.
- The function $getNonterminals()$ returns $\{S\}$ since this is the only non-terminal which could be derived in path from vertex 0 to 3 in the box S.

Thus we can conclude that the vertex with $id = 3$ is reachable from the vertex with $id = 1$ by the path derivable from S. As a result, S is added to the $M_2^1[1, 3]$. The updated matrix and graph after the first iteration are presented in figure 3.

For the second iteration matrices M_3^2 and C_3^2 are evaluated as follows:

$M_3^2 =$  , $C_3^2 =$  .

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

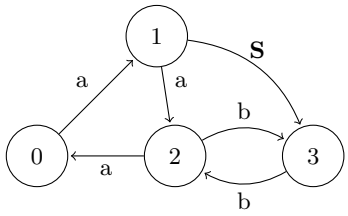


Fig. 3: Example: the updated matrix M_2^1 and graph $\mathcal{G}$ after first loop iteration

New non-zero values in the matrix C_3^2 have appeared during this iteration in cells with indices $[0, 11]$, $[0, 14]$ and $[5, 14]$. Because only the cell value with index $[0, 14]$ corresponds to the automata path with not empty non-terminal set $\{S\}$ its data affects the adjacency matrix M_2. The updated matrix and graph $\mathcal{G}$ are shown in Figure 4.

$$M_2^2 = \begin{pmatrix} . & \{a\} & \{\mathbf{S}\} & . \\ . & . & \{a\} & \{S\} \\ \{a\} & . & . & \{b\} \\ . & . & \{b\} & . \end{pmatrix}$$

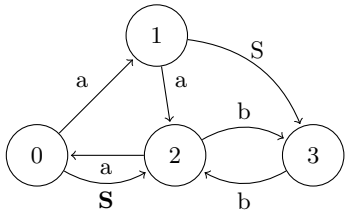


Fig. 4: Example: the updated matrix M_2^2 and graph $\mathcal{G}$ after second loop iteration

The remaining matrices C_3 and M_2 for the algorithm's main loop execution are listed in the Figure 5 and Figure 6 correspondingly. The evaluated matrices M_3 are not included because its computation is straightforward. The last loop iteration is 7. Although the matrix M_2^6 is updated with the new non-terminal S for the cell $[2, 2]$ after the transitive closure evaluation, the new values are not added to the matrix M_2. Therefore matrix M_2 has stopped changing and the algorithm has finished. The graph $\mathcal{G}$ with the new edges is presented in the Figure 2b.

5 Evaluation

We implement the proposed algorithm by using SuiteSparse[3] [4]: the implementation of GraphBlas API [8]. GraphBlas API specifies a set of linear algebra

[3] SuteSparse is a sparse matrix software which incudes GraphBLAS API implementation. Project web page: http://faculty.cse.tamu.edu/davis/suitesparse.html. Access date: 12.03.2020

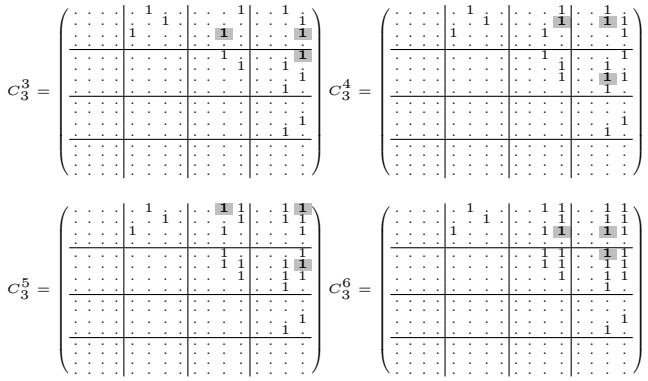


Fig. 5: Transitive closure for loop iterations $3-6$ for example query

$$M_2^3 = \begin{pmatrix} . & \{a\} & \{S\} & . \\ . & . & \{a\} & \{S\} \\ \{a\} & . & . & \{b, \mathbf{S}\} \\ . & . & \{b\} & . \end{pmatrix} M_2^4 = \begin{pmatrix} . & \{a\} & \{S\} & . \\ . & . & \{a, \mathbf{S}\} & \{S\} \\ \{a\} & . & . & \{b, S\} \\ . & . & \{b\} & . \end{pmatrix}$$

$$M_2^5 = \begin{pmatrix} . & \{a\} & \{S\} & \{\mathbf{S}\} \\ . & . & \{a, S\} & \{S\} \\ \{a\} & . & . & \{b, S\} \\ . & . & \{b\} & . \end{pmatrix} M_2^6 = \begin{pmatrix} . & \{a\} & \{S\} & \{S\} \\ . & . & \{a, S\} & \{S\} \\ \{a\} & . & \{\mathbf{S}\} & \{b, S\} \\ . & . & \{b\} & . \end{pmatrix}$$

Fig. 6: The updated matrix M_2 for loop iterations $3-6$ for example query

primitives and operations which allows one to formulate graph algorithms using linear algebra over custom semirings.

We compare our implementation with the results provided in [11]. We use the dataset described in this article which consists of **RDF**, **Worst case**, and **Full** subsets. For RDF querying we use same-generation query G_4 from [11].

For the evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz CPU, DDR4 32 Gb RAM.

The results of the evaluation are summarized in the table 1. Time is measured in seconds, t_1 is the execution time for the proposed algorithm, and t_2 is the time for M4RI-based implementation — the best CPU version form [11]. The time measurements are averaged over 10 runs. We exclude the time required to load data from the input file. The time required for the data transfer and its conversion is included.

We can see, that while RDF querying time is better for M4RI in general, in some cases execution times are comparable: for graphs *generations*, *travel*, *unv-bnch*, *skos*. Our algorithm demonstrates poor performance for the **Full** data set because SuiteSparse is based on sparse matrix representation, and in this case, the density of the matrices changes aggressively from very sparse to full. At the same time, we can see that in the **Worst case** our algorithms up to 4 times faster than M4RI (graph WC_5).

Table 1: Evaluation results: t_1 — proposed algortithm; t_2 — matrix-based algorithm

	Graph	#V	#E	t_1	t_2		Graph	#V	#E	t_1	t_2
RDF	atm-prim	291	685	0.24	0.02	RDF	core	1323	8684	0.28	0.12
	biomed	341	711	0.24	0.05		wine	733	2450	1.71	0.06
	foaf	256	815	0.07	0.02	Worst case	WC_1	64	65	0.03	0.04
	funding	778	1480	0.43	0.07		WC_2	128	129	0.16	0.23
	generations	129	351	0.04	0.03		WC_3	256	257	0.96	1.99
	people_pets	337	834	0.18	0.03		WC_4	512	513	7.14	23.21
	pizza	671	2604	1.14	0.08		WC_5	1024	1025	121.99	528.52
	skos	144	323	0.02	0.04	Full	F_1	100	100	0.17	0.02
	travel	131	397	0.05	0.05		F_2	200	200	1.04	0.03
	unv-bnch	179	413	0.05	0.04		F_3	500	500	18.86	0.03
	pathways	6238	37196	4.88	0.18		F_4	1000	1000	554.22	0.07

To sum up, our prototype implementation of the described algorithm is not performant enough to be used for real-world applications but it outperforms the matrix-based algorithm on the **Worst case** dataset and is comparable with it on some graphs from the **RDF** dataset. We conclude that we should improve our implementation to achieve better performance, while the algorithm idea is viable.

6 Conclusion

We presented a new algorithm for CFPQ which is based on Kronecker product and transitive closure. It can be implemented by using high-performance libraries for linear algebra. Also, our algorithm avoids grammar growth by handling queries represented as recursive state machines.

We implement the proposed algorithm by using SuiteSparse and evaluate it on several graphs and queries. We show that in some cases our algorithm outperforms the matrix-based algorithm, but in the future, we should improve our implementation for it to be applicable for real-world graphs analysis.

Also in the future, we should investigate such formal properties of the proposed algorithm as time and space complexity. Moreover, we plan to analyze how the behavior depends on the query type and its form. Namely, we should analyze regular path queries evaluation and context-free path queries in the form of extended context-free grammars (ECFG) [6]. The utilization of ECFGs may provide a way to optimize queries by minimization of both the right-hand sides of productions and the whole result RSM.

Finally, it is necessary to compare our algorithm with the matrix-based one in cases when the size difference between Chomsky Normal Form and ECFG representation of the query is significant.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. pp. 207–220. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
2. Azimov, R., Grigorev, S.: 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). pp. 5:1–5:10. GRADES-NDA '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3210259.3210264
3. Barrett, C., Jacob, R., Marathe, M.: Formal-language-constrained path problems. SIAM Journal on Computing **30**(3), 809–837 (2000), https://doi.org/10.1137/S0097539798337716
4. Davis, T.A.: Algorithm 9xx: Suitesparse:graphblas: graph algorithms in the language of sparse linear algebra (2018)
5. Hellings, J.: Querying for paths in graphs using context-free path queries. arXiv preprint arXiv:1502.02242 (2015)
6. Hemerik, K.: Towards a taxonomy for ecfg and rrpg parsing. In: Proceedings of the 3rd International Conference on Language and Automata Theory and Applications. pp. 410–421. LATA09, Springer-Verlag, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-00982-2_35
7. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
8. Kepner, J., Aaltonen, P., Bader, D., Bulu, A., Franchetti, F., Gilbert, J., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., McMillan, S., Yang, C., Owens, J.D., Zalewski, M., Mattson, T., Moreira, J.: Mathematical foundations of the graphblas. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–9 (Sep 2016). https://doi.org/10.1109/HPEC.2016.7761646
9. Kuijpers, J., Fletcher, G., Yakovets, N., Lindaaker, T.: An experimental study of context-free path query evaluation methods. In: Proceedings of the 31st International Conference on Scientific and Statistical Database Management. pp. 121–132. SSDBM19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3335783.3335791
10. Medeiros, C.M., Musicante, M.A., Costa, U.S.: Efficient evaluation of context-free path queries for graph databases. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1230–1237. SAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167132.3167265, http://doi.acm.org/10.1145/3167132.3167265
11. Mishin, N., Sokolov, I., Spirin, E., Kutuev, V., Nemchinov, E., Gorbatyuk, S., Grigorev, S.: 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-NDA19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3327964.3328503
12. Sevon, P., Eronen, L.: Subgraph queries by context-free grammars. Journal of Integrative Bioinformatics **5**(2), 100 (2008)
13. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 230–242. PODS '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/298514.298576

14. Zhang, X., Feng, Z., Wang, X., Rao, G., Wu, W.: Context-free path queries on rdf graphs. In: International Semantic Web Conference. pp. 632–648. Springer (2016)