

Context-Free Path Querying by Using Kronecker Product^{*}

Egor Orachev¹, Ilya Epelbaum¹, 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

`semyon.grigorev@jetbrains.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 regular path queries (RPQ) and allow one to use context-free grammars as constraints for paths. Algorithms for CFPQ are actively developed, but J. Kuijpers recently concludes, that existing algorithms are not performant enough to be used in real-world applications. Thus the development of new algorithms for CFPQ is required. 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, and avoids grammar growth, consequently, it provides the ability 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 [?], and particularly context-free path querying (CFPQ) [?], allows one to express path constraints for a graph in terms of context-free grammars: path in graph included to query result only if concatenated labels along this path form a word belongs to the language, generated by query grammar. CFPQ is widely used in bioinformatics [?], graph databases [?], and RDF analysis [?].

CFPQ algorithms are actively developed, but still there is a problem with its performance [5]. One of the most promising algorithms is the algorithm, proposed by Rustam Azimov [2]. This algorithm allows one to offload computational intensive part to high-performance libraries for linear algebra, this way one can utilize modern parallel hardware for CFPQ. But, as far as performance depends

^{*} Supported by organization x.

on grammar size, a number of productions for a grammar is still a problem, since it is processed in form of Chomsky Normal Form (CNF).

In this work, we propose new algorithm, expressed in terms of matrix operations, which can utilise expressive power of regular expressions as well as accept context-free queries, and also provide some space for future query optimisations.

Main contribution of this paper could be summarised as follows.

1. We introduce an new algorithm for CFPQ, which is based on recursive state machines intersection and can be expressed in terms of Kronecker product and transitive closure evaluation.
2. We provide a step-by-step example of the algorithm.
3. We provide an evaluation of the proposed algorithm and its comparison with matrix based algorithm. Evaluation results show that the idea is promising because we outperform matrix-based algorithm on the worst case data set, but optimizations are required to be applicable for real-world cases.

2 Recursive State Machines

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

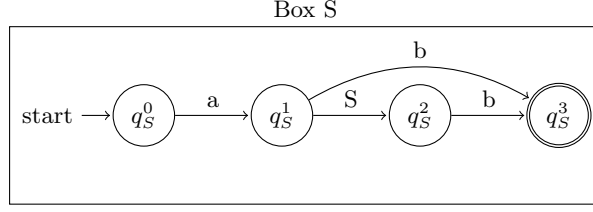
A recursive state machine R over a finite alphabet Σ is defined as tuple of elements $(M, m, \{C_i\}_{i \in M})$, where:

- M is a finite set of boxes' labels
- 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 set of symbols, $\Sigma \cap M = \emptyset$
 - Q_i is finite set of states, where $Q_i \cap Q_j = \emptyset, \forall i \neq j$
 - q_i^0 is an initial state for component state machine C_i
 - F_i is set of final states for C_i , where $F_i \subseteq Q_i$
 - δ_i is transition function for C_i , where $\delta_i : Q_i \times (\Sigma \cup M) \rightarrow Q_i$

RSM behaves as set of finite state machines (or FSM), so called *boxes* or *component state machines* [1], which are executed in classical definition of FSM with additional *recursive calls* and implicit *call stack*, what allows to *call* one component from another, and then return execution flow back.

Accordingly to [1], recursive state machines are equivalent to pushdown systems. Since pushdown systems are capable of accepting context-free languages [4], it is clear that RSMs are equals to context-free languages. Thus we can use a RSMs to encode query grammar. Algorithm for CFG to RSM conversion is provided in [?]. An exaple of RSM R for the grammar G with rule $S \rightarrow aSb \mid ab$ is provided in figure 1.

Since R is a set of FSMs, it is useful for computational tasks to represent R as a adjacency matrix, 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 our machine R is be provided in section 3.1.

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

3 Kronecker Product Based CFPQ Algorithm

In this section, we introduce an algorithm for computation of context-free reachability in graph \mathcal{G} and RSM R . The algorithm is based on generalisation of the FSM intersection for a RSM, created from input grammar, and an input graph. Since a graph can be interpreted as FSM, where edges with labels represent transitions between vertices of the graph, and a RSM is composed from set of FSMs, it is clear to evaluate intersection of such machines using classical algorithm for FSM, represented in [4].

The result of the intersection could be evaluated as a Kronecker product of the corresponding adjacency matrices for RSM and graph. To solve reachability problem it is enough to represent intersection result as a Boolean matrix, because we are interested only in reachability of vertices. It simplifies algorithm implementation and allows to express it in terms of basic matrix operations.

Listing 1 shows main steps of the solution. As an input algorithm accepts context-free grammar $G = (\Sigma, N, P)$ and graph $\mathcal{G} = (V, E, L)$. RSM R is created from G . Note, that R must have no ε -transitions. M_1 and M_2 are the adjacency matrices for machine R and graph \mathcal{G} correspondingly.

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

The algorithm is executed while matrix M_2 is changing. For each iteration Kronecker product of matrices M_1 and M_2 is evaluated. The result is saved in M_3 as a Boolean matrix. For given M_3 evaluated C_3 matrix via *transitiveClosure()* function call. The M_3 could be interpreted as an adjacency matrix for an oriented graph without 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 pair of indices (i, j) computes s and f — initial and final states in recursive automata R which relate to the concrete $C_3[i, j]$ of the tensor matrix. If given s and f belongs to a same box B of R and $s = q_B^0$ and $f \in F_B$, then *getNonterminals()* returns the respective non-terminal. If the conditional

statement is *true* then algorithm adds 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 indeces between Kronecker product arguments and result matrix. Implementation appeals to the blocked structure of the matrix C_3 , where each block corresponds to some automata and graph edge.

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

Listing 1 Kronecker product based CFPQ

```

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:   for  $s \in 0..dim(M_1) - 1$  do
6:     for  $i \in 0..dim(M_2) - 1$  do
7:        $M_2[i, i] \leftarrow M[i, i]_2 \cup getNonterminals(R, s, s)$ 
8:   while Matrix  $M_2$  is changing do
9:      $M_3 \leftarrow M_1 \otimes M_2$  ▷ Evaluate Kroncker product
10:     $C_3 \leftarrow transitiveClosure(M_3)$ 
11:     $n \leftarrow dim(M_3)$  ▷ Matrix  $M_3$  size =  $n \times n$ 
12:    for  $i \in 0..n - 1$  do
13:      for  $j \in 0..n - 1$  do
14:        if  $C_3[i, j]$  then
15:           $s, f \leftarrow getStates(C_3, i, j)$ 
16:          if  $getNonterminals(R, s, f) \neq \emptyset$  then
17:             $x, y \leftarrow getCoordinates(C_3, i, j)$ 
18:             $M_2[x, y] \leftarrow M_2[x, y] \cup getNonterminals(R, s, f)$ 
19:  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  $i \bmod n, j \bmod n$ 

```

This section is intended to provide step-by-step demonstration of the proposed algorithm. As an example consider the theoretical worst case for CFPQ time complexity, proposed by J.Hellings [3]: graph \mathcal{G} presented in Figure 2a and context-free grammar G for a language $\{a^n b^n \mid n \geq 1\}$: $S \rightarrow aSb \mid ab$.

Figure 1 consists of two directed graphs, (a) and (b), each with four nodes labeled 0, 1, 2, and 3.

(a) The input graph \mathcal{G} : Node 0 has a directed edge labeled 'a' to node 1 and a directed edge labeled 'a' to node 2. Node 1 has a directed edge labeled 'a' to node 2. Node 2 has a directed edge labeled 'b' to node 3 and a directed edge labeled 'b' to node 0.

(b) The result graph \mathcal{G} : Node 0 has a directed edge labeled 'a' to node 1 and a directed edge labeled 'S' to node 2. Node 1 has a directed edge labeled 'a' to node 2 and a directed edge labeled 'S' to node 3. Node 2 has a directed edge labeled 'b' to node 3 and a directed edge labeled 'S' to node 0. Node 3 has a directed edge labeled 'b' to node 2 and a directed edge labeled 'S' to node 1.

Fig. 2: The input and result graphs for example

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

Then the algorithm enters while loop and iterates as long as matrix M_2 is changing. We provide step-by-step evaluation of matrices M_3 , C_3 and updating of matrix M_2 . All the matrices are denoted with upper index of the current loop iteration. The first loop iteration is indexed as 1.

[illegible]

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

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

- Indices of the automata vertices $s = 0$ and $f = 3$, because value $C_3^1[1, 15]$ located in upper right matrix block $(0, 3)$.
- Indices of the graph vertices $x = 1$ and $y = 3$ are evaluated as value $C_3^1[1, 15]$ indices relatively to its block $(0, 3)$.
- 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 path derivable from S . As a result, S is added to the $M_2^1[1, 3]$. The updated matrix and graph after first loop iteration are presented in figure 3.

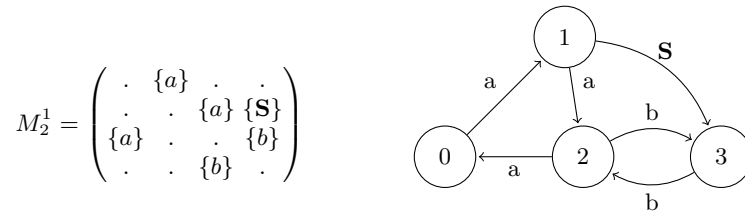


Fig. 3: The updated matrix M_2^1 and graph \mathcal{G} after first loop iteration for example query

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

[illegible]

For this iteration in the matrix C_3^2 appeared new non-zero values 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 adjacency matrix M_2 . The updated matrix and graph \mathcal{G} are depicted in Figure 4.

The remaining matrices C_3 and M_2 for the algorithm main loop execution are listed in the Figure 5 and Figure 6 correspondingly. Evaluated matrices M_3 are

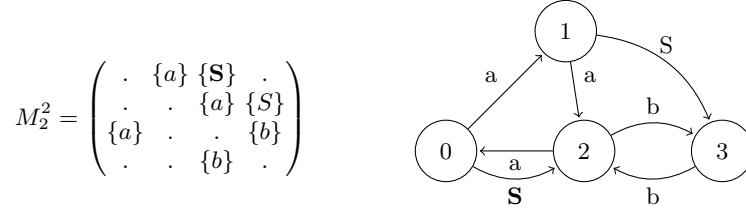


Fig. 4: The updated matrix M_2^2 and graph \mathcal{G} after second loop iteration for example query

not included because its computation is a straightforward process. The last loop iteration is 7. Although the matrix M_2^6 is updated with new non-terminal S for the cell $[2, 2]$ after transitive closure evaluation the new values to the matrix M_2 is not added. Therefore matrix M_2 has stopped changing and the algorithm is successfully finished. The graph \mathcal{G} with new edges is presented in the Figure 2b.

$$\begin{aligned}
 C_3^3 &= \begin{pmatrix} \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} & C_3^4 &= \begin{pmatrix} \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} \\
 C_3^5 &= \begin{pmatrix} \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} & C_3^6 &= \begin{pmatrix} \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}
 \end{aligned}$$

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

4 Evaluation

We implement the proposed algorithm by using SuitSparse: the implementation of GraphBlas API³. GraphBlas API specifies a set of linear algebra primitives and operation which allows one to formulate graph algorithms using linear algebra over custom semirings.

³ <http://graphblas.org>

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

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

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

We compare our implementation with results provided in [6], accordingly we use dataset described in this article which consists of **RDF**, **Worst case**, and **Full** subsets. For RDF querying we use same-generator query G_4 from [6].

For 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 an execution time for the proposed algorithm, and t_2 is a time for M4RI-based implementation — the best CPU version from [6]. The result for algorithm is averaged over 10 runs. We exclude the time required to load data from file. The time required for data transfer and its conversion is included.

Table 1: Evaluation results

	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	1.94
	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

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

To sum up, our prototype implementation of the described algorithm is not performant enough to be used for real-world applications but it outperforms a

matrix-based algorithm on **Worst case** dataset and comparable with it on some graphs from the **RDF** dataset. Thus we can conclude that we should improve our implementation to achieve better performance.

5 Conclusion

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

We implement the proposed algorithm by using SuitSparse and evaluate it on a number of 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 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 should analyze behavior dependency on 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 (EBNF).

Finally, it is necessary to compare our algorithm with the matrix-based one in cases when the size difference between Chomsky normal form and EBNF 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. Hellings, J.: Querying for paths in graphs using context-free path queries. arXiv preprint arXiv:1502.02242 (2015)
4. 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)
5. 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>
6. 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>