Context-Free Path Querying by Using 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
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. Abstact is very abstract.

Keywords: 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 performanve [?]. One of the most promising algorithms is the algorithm, proposed by Rustam Azimov [2]. This algorithm allows one to offload computationsl intencive part to high-performance libraries for linear algebra, this way one can

* Supported by organization x.

utilize modern parallel hardware for CFPQ. But, as far as performance depends 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 requred 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 $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$

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 [5], 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 convertion is provided in [**?**]. An exaple of RSM R for the grammar G with rule $S \to aSb \mid ab$ 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.

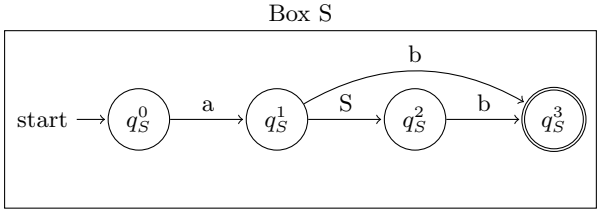


Fig. 1: The recursive state machine R for grammar G

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 [5].

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 nonterminals, 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, 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] ← M[i, i]2 ∪ getNonterminals(R, s, s)
8:     while Matrix M2 is changing do
9:         M3 ← M1 ⊗ M2                          ▷ Evaluate Kroncker 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
```

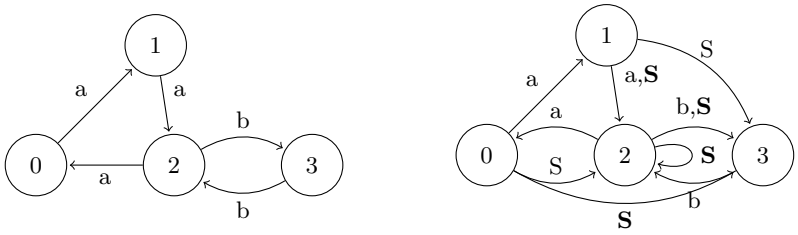
Listing 2 Help functions for Kronecker product based CFPQ

```
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
```

3.1 Example

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 [4]: graph $\mathcal{G}$ presented in Figure 2a and context-free grammar G for a language $\{a^n b^n \mid n \geq 1\}$: $S \to aSb \mid ab$.

Since the proposed algorithm processes grammar in form of 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 $F_S = \{(3)\}$.



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

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

Fig. 2: The input and result graphs for example

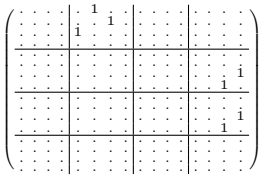
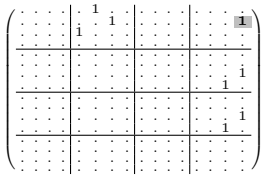
Adjacency matrices M_1 and M_2 for automata R and graph $\mathcal{G}$ respectively are initialised as follows:

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

After all the data is initialised in lines **2**–**4**, the algorithm handles ε-case. Because machine R does not have ε-transitions and ε-word is not included in grammar G language lines **5**–**7** of the algorithm do no affect the input data.

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.

For the first while loop iteration the tensor product $M_3^1 = M_1 \otimes M_2^0$ and transitive closure C_3^1 are evaluated as follows:

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

After the transitive closure evaluation matrix C_3^1 cell $(1, 15)$ 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 the value $C_3^1[1, 15]$:

- 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 call $getNonterminals()$ returns $\{S\}$ since this is the only non-terminal which could be derived in path from vertex 0 to 3 in the box S.

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

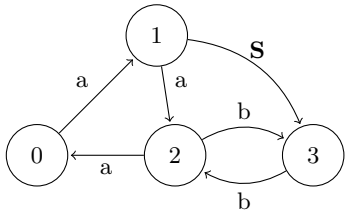


Fig. 3: The updated matrix M_2^1 and graph $\mathcal{G}$ after first loop iteration for example query

After the first loop iteration matrix symbol S is added to the cell $M_2^1[1, 3]$. It is relevant data, because initial graph has path $1 \to 2 \to 3$ which could be derived for S. The updated matrix and graph are depicted in Figure 3.

For the second loop iteration matrices M_3^2 and C_3^2 are evaluated as listed in Figure 4. 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 5.

The remaining matrices C_3 and M_2 for the algorithm main loop execution are listed in the Figure 6 and Figure 7 correspondingly. Evaluated matrices M_3 are 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.

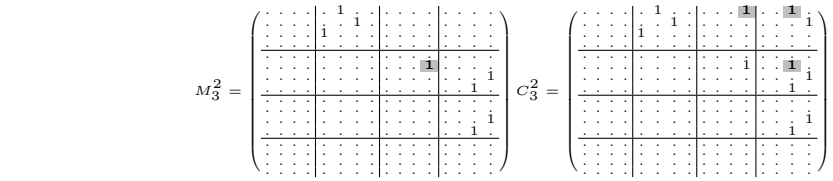


Fig. 4: The second iteration tensor product and transitive closure evaluation for example query

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

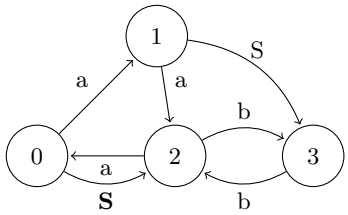


Fig. 5: The updated matrix M_2^2 and graph $\mathcal{G}$ after second loop iteration for example query

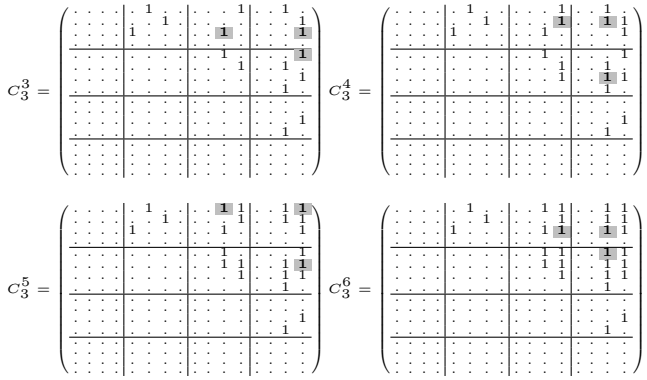


Fig. 6: Transitive closure for $3-6$ loop iterations for example query

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

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

Fig. 7: The updated matrix M_2 for $3-6$ loop iterations for example query

4 Evaluation

In this section, we introduce evaluation of implementation of described algorithm. We compare our implementation with [6] CPU-based results, accordingly we use dataset described in this article. First type of graphs is RDF. This set contains real-world graphs. The second type is Worst case. The theoretical worst case for CFPQ. And last type is Full graph. The set of sparse graph, where the result is a full graph. We exclude the time required to load data from file. The time required for data transfer and its conversion is included.

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.

We use SuitSparse:GraphBlas [3] to implement. GraphBlas is API for graph algorithms using linear algebra, respectively, using parallelism for matrix operations. This tool is so flexible that it allows to change built-in operations by creating custom types and actions on them. To load graphs dataset,use RedisGraph [4]. RedisGraph is graph database that uses GraphBlas, which allows us to combine them to obtain effective results.

The results of the evaluation are summarized in the tables below. Time is measured in seconds. The result for algorithm is averaged over 10 runs.

Table 1: RDF results

Name	#V	#E	Time
atm-prim	291	685	0.239425
biomed	341	711	0.240378
foaf	256	815	0.073081
funding	778	1480	0.431305
generations	129	351	0.041165
people_pets	337	834	0.178783
pizza	671	2604	1.137291
skos	144	323	0.019502
travel	131	397	0.046559
unv-bnch	179	413	0.048634
wine	733	2450	1.707681
core	1323	8684	0.281461
pathways	6238	37196	4.88529

Table 2: Worst case

#V	Time
32	0.010328
64	0.032295
128	0.159227
256	0.96474
512	7.13957
1024	121.987482

Table 3: Full graph

#V	Time
100	0.171031
200	1.043202
500	18.863558
1000	554.223892

The results of the first dataset **RDF** are presented in table 1. We can see, that in this case the running time of implementation of our algorithm is bigger than of the reference implementations of matrix-based algorithm except for one implementation. At the same time, the worst result for all types of graphs in the reference article is 8 milliseconds.

Results of the theoretical worst case **Worst** are presented in table 2. We can see, that the running time of implementation is much less than reference

[3] http://graphblas.org
[4] https://redislabs.com

implementations of another algorithm on CPU-based. For example, for a graph of 1024 vertices, our algorithm shows an improvement in time of more than 4 times compared to the best CPU-based result even in the first implementation.

The last dataset is **Full**, and results are shown in table 3. As can be seen from the results, the running time increases significantly with an increase in the number of vertices, this is especially noticeable between 500 and 1000, while the worst result for a graph of 1000 vertices in the reference article is 13.071.

To sum up, our first implementation of the described algorithm outperformed implementations matrix-based algorithm in only one type of graph. Other types of graphs require implementation rework.

5 Conclusion

We present !!!

Future research. Performance improvements. Detailed investigation of the algorithm formal properies such as time and space complexity. GraphBLAST. Paths, not just reachability.

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.: Conjunctive context-free path queries. In: ICDT (2014)
4. Hellings, J.: Querying for paths in graphs using context-free path queries. arXiv preprint arXiv:1502.02242 (2015)
5. 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)
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