Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication

Nikita Mishin
Iaroslav Sokolov
Egor Spirin
mishinnikitam@gmail.com
sokolov.yas@gmail.com
egor@spirin.tech
Saint Petersburg State University
St. Petersburg, Russia

Vladimir Kutuev
Egor Nemchinov
Sergey Gorbatyuk
vladimir.kutuev@gmail.com
nemchegor@gmail.com
sergeygorbatyuk171@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
semen.grigorev@jetbrains.com
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Recently proposed matrix multiplication based algorithm for context-free path querying (CFPQ) offloads the most performance-critical parts onto boolean matrices multiplication. Thus, it is possible to utilize modern parallel hardware and software to achieve high performance of CFPQ easily. In this work, we provide results of empirical performance comparison of different implementations of this algorithm on both real data and synthetic data for the worst cases.

CCS CONCEPTS

• **Information systems → Query languages for non-relational engines**; • **Theory of computation → Grammars and context-free languages**; *Parallel computing models*; • **Computing methodologies → Massively parallel algorithms**; • **Computer systems organization →** *Single instruction, multiple data.*

KEYWORDS

Context-free path querying, transitive closure, graph databases, context-free grammar, GPGPU, CUDA, matrix multiplication, boolean matrix

ACM Reference Format:
Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2018. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of GRADES-NDA 2019: the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2019 (GRADES-NDA 2019).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Language-constrained path querying [**?**], and particularly Context-Free Path Querying (CFPQ) [**?**], allows one to use formal grammars as constraints for path: concatenation of labels along the path is treated as a word, and constraint on the path is a specification of language which should contain specific words. CFPQ is widely used for graph-structured data analysis in such domains as biological data analysis, RDF, network analysis, and huge amount of the real-world data makes performance of CFPQ critical for practical tasks. Big amount of work has done in this area and a number of algorithms for CFPQ proposed recently [7, 9**? ? ? ?**].

One of the most promising algorithms is a matrix-based algorithm, proposed by Rustam Azimov [4]. This algorithm offloads the most critical computations onto boolean matrices multiplication. As a result, it is easy to implementat and allows one to utilize modern massive-parallel hardware for CFPQ. The implementation provided by authors utilizes GPGPU by using cuSPARSE[1] library which is a floating point sparse matrices multiplication library. Even it does not use advanced algorithms for boolean matrices, it outperforms existing algorithms.

It is necessary to investigate the effect of specific algorithms and implementation techniques on the performance of CFPQ. One of the problems is that there is no publically available standard dataset for CFPQ algorithms evaluation which includes both graph-structured data and queries.

In this work, we do an empirical performance comparison of different implementations of matrices multiplication based algorithm for CFPQ on both real data and synthetic data for the worst cases. We make the following contributions in this paper.

(1) We provide a number of implementations of the matrix multiplication based CFPQ algoithm, which utilizes different modern software and hardware. Source code is available on GitHub:!!!
(2) We collect and publish a dataset which contains both real data and syntatic data for wirst cases. This dataset contains data and queries in the simplee textual format, so it can be used for other algorithms evaluation easily. We hope that this dataset can form a base of the unified benchmark for CFPQ algorithms.
(3) We provide evaluation which shows that GPGPU utilization for CFPQ can significantly improve performance, and that there are many open questions in this area.

GRADES-NDA 2019, June 30, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

[1] cuSparse is a library for GPGPU utilization for sparse matrices multiplication. Official documentation:https://docs.nvidia.com/cuda/cusparse/index.html. Access date: 12.03.2019

2 MATRIX-BASED ALGORITHM FOR CFPQ

Matrix-based algorithm for CFPQ was proposed by Rustam Azimov [4]. This algorithm can be expressed as showed in in listing 1 in terms of operations over matrices, and it is a sufficient advantage for implementation. It was shown that GPGPU utilization for queries evaluation can significantly improve performance in comparison to other implementations [4] even if float matrices used instead of boolean matrices.

Pseudocode of the algorithm is presented .

Listing 1 Context-free path quering algorithm

```
1:  function contextFreePathQuerying(D, G)
2:      n ← the number of nodes in D
3:      E ← the directed edge-relation from D
4:      P ← the set of production rules in G
5:      T ← the matrix n × n in which each element is ∅
6:      for all (i, x, j) ∈ E do                ▷ Matrix initialization
7:          T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:      while matrix T is changing do
9:          T ← T ∪ (T × T)                     ▷ Transitive closure calculation
10:     return T
```

Here $D = (V, E)$ be the input graph and $G = (N, \Sigma, P)$ be the input grammar. Each cell of the matrix T contains the set of nonterminals such that $N_k \in T[i,j] \iff \exists p = v_i \dots v_j$—path in D, such that $N_k \overset{*}{\underset{G}{\Rightarrow}} \omega(p)$, where $\omega(p)$ is a word formed by labels along the path p. Thus, this algorithm solves reachability problem, or, according to Hellings [6], process CFPQs by using relational query semantics.

As you can see, performance-critical part of this algorithm is matrix multiplication. Note, that the set of nonterminals is finite, and we can represent the matrix T as a set of boolean matrices: one for each nonterminal. In this case the matrix update operation be $T_{N_i} \leftarrow T_{N_i} + (T_{N_j} \times T_{N_k})$ for each production $N_i \rightarrow N_j\ N_k$ in P. Thus we can reduse CFPQ to boolean matrices multiplication. After such transfromation we can apply the next optimization: we can skip update if there are no changes in the matrices T_{N_j} and T_{N_k} at the previous iteration.

Thus, the most important part is efficient implementation of operations over boolean matrices, and in this work we compare effects of utilization of different approaches to matrices multiplication. All our implemetations are based on the optimized version of the algorithm.

3 IMPLEMENTATION

We implement the matrix-based algorithm for CFPQ by using a number of different programming languages and tools. Our goal is to investigate the effects of the following features of implementation.

- **GPGPU utilization.** It is well-known that GPGPUs are suitable for matrices operations, but the performance of the whole solution depends on numder of details. For example, overhead on data transferring may negate the effect of parallel computations. Can GPGPUs utilization for CFPQ improve performance in comparison with CPU version?
- **Existing libraries utilization** is a good practice in software engineering. Is it possible to achieve higher performance by using existing libraries for matrices operations or we need to create own solution to get more control?
- **Low-level programming**. GPGPU programming traditionally involves low-level programming in C-based languages (CUDA C, OpenCL C). But can we achieve high-performant solution with level languages such as a Python?
- **Sparce matrices.** Real graphs are often sparse. Can we gain something by using sparse matrix representation for CFPQ?

We provide the following implementations for investigation.

- CPU-based solutions
  [Scipy] Sparse matrices multiplication by using Scipy [8] in Python programming language.
  [M4RI] Dense matrices multiplication by using m4ri[2] [1] library which implements 4 Russian method [3] in C language. This library is choosen because it is one of performnat implementation of 4 russian method [2].
- GPGPU-based solutions
  [GPU4R] Our own implementation of 4 Russian method in CUDA C.
  [GPU_N] Our own implementation of the naïve boolean matrix multiplication in CUDA C with boolean values treated as bits and packed into `uint_32`.
  [GPU_Py] Manual implementation of naïve boolean matrix multiplication in Python by using numba compiler[3]. Boolean values packed into `uint_32`.

As far as a set of matrices and its size can be statically at the start of computations, all GPGPU based implementations allocate all required memory on the GPGPU once, at the start of computations. This way, it is possible to significantly reduce overhead on data transferring: all input data loads to GPGPU at the start, and the result loads from GPGPU to the host at the end. As a result, there is no active data transferring and memory allocating during query computation.

4 DATASET DESCRIPTION

We created and published a dataset for CFPQ algorithms evaluation. This dataset contains both the real-world data and synthetic data for different specific cases, such as the theoretical worst case, or matrices representation specific worst cases.

Our goal is to evaluate querying algorithms, not a graph storages or graph databases, so all data is presented in a text-based format to simplify usage in different environments. Grammars are in Chomsky Normal Form, and graphs are represented as a list of triples (edges). For details on data representation look at the appendix.

It is known that variants of the *same generation query* [**?**] are a important example of queries that are context-free but not regular, so we use this type of queries in our evaluation. The datatset includes data for next cases. Each case is a pair of set of graphs and a

[2] Actually we use pull request which is not merged yet: https://bitbucket.org/malb/m4ri/pull-requests/9/extended-m4ri-to-multiplication-over-the/diff. The original library implements operations over $GF(2)$, and this pull request contains operations over boolean semiring

[3] Numba is a JIT compiler which supports GPGPU for a subset of Python programming. Official page: http://numba.pydata.org/. Access date: 03.05.2019

set of grammars: each query (grammar) should be applied to each graph.

[RDF] The set of real-world RDF files (ontologies) from [9] and two variants of the same generation query (figure 4 in appendix) which describes hierarchy analysis.

[Worst] Theoretical worst case for CFPQ time complexity which is proposed by Hellings [7]: the graph is two cycles of coprime lengths with a single common vertex. The first cycle is labeled by an open bracket and the second cycle is labeled by a close bracket. Query is a grammar for $A^n B^n$ languge (grammar G_1, figure 1 in appendix).

[Full] The case when the input graph is sparse, but the result is a full graph. Such a case may be hard for sparse matrices representation. As an input graph, we use a cycle, all edges of which are labeled by the same token. As a query we use two grammars which describe sequence of tokens of arbitrary length: simple ambiguous grammar G_2 and highly ambiguous grammar G_3 (figure 3 in appendix).

[Sparse] Sparse graphs from [5] generated by the GTgraph graph generator, and emulates realistic sparse data. Names of these graphs have a form `Gn-p`, where `n` represents the total number of vertices, each pair of vertices is connected by probability `p`. The query is the same generation query represented by grammar G_1 (figure 1 in appendix).

5 EVALUATION

We evaluate all described implementations on all data sets and queries presented. We compare our impementations with [4]. We exclude time required to load data from files. The time required for data transfer is included.

For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7 8700k 3,7HGz CPU, Ddr4 32 Gb RAM, and Geforce 1080Ti GPGPU with 11Gb RAM.

Results of evaluation are summarized in the tables below. Time is measured in seconds. Result for each algorithm is an average time of 10 runs. Time is not presented if the time limit is exceeded, or if no memory enough to allocate the data necessary.

The results of the first dataset **[RDF]** are presented in a table 1. We can see, that in this case running time for all our implementations smaller than time for the reference implementation, and that **[GPU_N]** is faster than other implementations while other implementations demonstrate similar performance. Also, it is obvious that performance improvement in comparison with the first implementations is huge and it is necessary to extend dataset with new RDFs of the significantly biggest size.

Table 2: Worst case evaluation results

#V	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
16	0.032	< 0.001	0.008	0.002	0.027	0.309
32	0.118	0.001	0.034	0.008	0.136	0.441
64	0.476	0.041	0.133	0.032	0.524	0.988
128	2.194	0.226	0.562	0.129	2.751	3.470
256	15.299	1.994	3.088	0.544	11.883	15.317
512	121.287	23.204	13.685	2.499	43.563	102.269
1024	1593.284	528.521	88.064	19.357	217.326	1122.055
2048	-	-	-	325.174	-	-

Results of the theoretical worst case (**[Worst]** datatset) is presented in table 2. This case is really hard to process: even for a graph of 1024 vertices, query evaluation time is greater than 10 seconds even for most performant implementation. Also, we can see, that time grows fast with grows of vertices number.

Table 3: Sparse graphs querying results

Graph	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
G5k-0.001	10.352	0.647	0.113	0.041	0.216	5.729
G10k-0.001	37.286	2.395	0.435	0.215	1.331	35.937
G10k-0.01	97.607	1.455	0.273	0.138	0.763	47.525
G10k-0.1	601.182	1.050	0.223	0.114	0.859	395.393
G20k-0.001	150.774	11.025	1.842	1.274	6.180	-
G40k-0.001	-	97.841	11.663	8.393	37.821	-
G80k-0.001	-	1142.959	88.366	65.886	-	-

The next is a **[Sparse]** datatset presented in table 3. The evaluation shows that sparsity of graphs (value of parameter `p`) is important both for implementations which use sparse matrices and for implementations which use dense matrices. Note that the behavior of sparse matrices based implementation is as expected, but for dense matrices we can see, that more sparse graphs processed faster. Reasons of such behaviour demand further investigation. Note that we estimate only query execution time, so it is hard to compare our results with the results presented in [5]. But it would be interesting to do such a comparison in the future because the running time of our **[GPU_N]** implementation is significantly smaller than the provided in [5].

The last dataset is a **[Full]**, and results are shown in table 4

As we expect, this case is very hard for sparse matrices based implementations: running time grows too fast. Also, we can see, that the grammar size is important. Both queries specify the same restriction, but grammar G_3 in CNF contains 2 times more rules then grammar G_2, and as a result, the running time for big graphs differs more than 2 times.

Finally, we can conclude that GPGPU utilization for CFPQ can significantly improve performance, but more research on advanced optimization techniques should be done. On the other hand, high-level implementation (**[GPU_Py]**) is comparable with other GPGPU-based implementations. So, it may be balance between implementation complexity and performance. Highly optimized existing libraries can be useful: implementation based on m4ri is faster than reference implementation and other CPU-based implementation.

Table 1: RDFs querying results

RDF			Query G_4						Query G_5					
Name	#V	#E	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
atom-primitive	291	685	0.003	0.002	0.002	0.001	0.005	0.269	0.001	< 0.001	0.001	< 0.001	0.002	0.267
biomed.-mesure-primitive	341	711	0.003	0.005	0.002	0.001	0.005	0.283	0.004	< 0.001	0.001	< 0.001	0.005	0.280
foaf	256	815	0.002	0.009	0.002	< 0.001	0.005	0.270	0.001	< 0.001	0.001	< 0.001	0.002	0.263
funding	778	1480	0.004	0.007	0.004	0.001	0.005	0.279	0.002	< 0.001	0.003	< 0.001	0.004	0.274
generations	129	351	0.003	0.003	0.002	< 0.001	0.005	0.273	0.001	< 0.001	0.001	< 0.001	0.002	0.263
people_pets	337	834	0.003	0.003	0.003	0.001	0.007	0.284	0.001	< 0.001	0.001	< 0.001	0.003	0.277
pizza	671	2604	0.006	0.008	0.003	0.001	0.006	0.292	0.002	< 0.001	0.002	< 0.001	0.005	0.278
skos	144	323	0.002	0.004	0.002	< 0.001	0.005	0.273	< 0.001	< 0.001	0.001	< 0.001	0.002	0.265
travel	131	397	0.003	0.005	0.002	< 0.001	0.006	0.268	0.001	< 0.001	0.001	< 0.001	0.003	0.271
univ-bench	179	413	0.002	0.004	0.002	< 0.001	0.005	0.266	0.001	< 0.001	0.001	< 0.001	0.003	0.266
wine	733	2450	0.007	0.006	0.004	0.001	0.007	0.294	0.001	< 0.001	0.003	< 0.001	0.003	0.281

Table 4: Full querying results

#V	Query G_2						Query G_3					
	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
100	0.007	0.002	0.002	< 0.001	0.003	0.278	0.023	0.076	0.005	0.001	0.007	0.290
200	0.040	0.003	0.002	0.001	0.004	0.279	0.105	0.098	0.004	0.001	0.007	0.296
500	0.480	0.003	0.003	0.001	0.004	0.329	1.636	0.094	0.007	0.001	0.010	0.382
1000	3.741	0.007	0.005	0.001	0.006	0.571	13.071	0.106	0.009	0.001	0.009	0.839
2000	40.309	0.063	0.019	0.003	0.017	1.949	93.676	0.108	0.030	0.005	0.026	3.740
5000	651.343	0.366	0.125	0.038	0.150	99.651	1205.421	0.851	0.195	0.075	0.239	201.151
10000	-	1.932	0.552	0.315	0.840	1029.042	-	4.690	1.055	0.648	1.838	-
25000	-	33.236	7.252	5.314	15.521	-	-	70.823	15.240	10.961	36.495	-
50000	-	360.035	58.751	44.611	129.641	-	-	775.765	130.203	91.579	226.834	-
80000	-	1292.817	256.579	190.343	641.260	-	-	-	531.694	376.691	-	-

Moreover, it is comparable with some GPGPU-based implementations in some cases. Sparse matrices utilization should be investigated more. The main question is if we can create an efficient implementation for sparse boolean matrices multiplication.

6 CONCLUSION AND FUTURE WORK

We provide a number of implementations of the matrix-based algorithm for context-free path querying, collect a dataset for evaluation and provide results of evaluation of our implementation on the collected dataset. Our evaluation shows that GPGPU utilization for boolean matrices multiplication can significantly increase the performance of CFPQs evaluation, but requires more research on implementation details.

The first direction for future research is a more detailed CFPQ algorithms investigation. We should do more evaluation on sparse matrices on GPGPUs and investigate technics for high-performance GPGPU code creation. Also, it is necessary to implement and evaluate solutions for graphs which do not fit in RAM, and for big queries which disallow to allocate all required matrices on single GPGPU. We hope that it is possible to utilize existing technics for huge matrices multiplication for this problem.

Another direction is dataset improvement. First of all, it is necessary to collect more data, and more grammars/queries. Especially it would be important to add to the dataset more real-world graphs and more real-world queries. Secondly, it is necessary to discuss and fix the data format to be able to evaluate different algorithms. We beleave that it is necessary to create a public dataset for CFPQ algorithms evaluation, and collaboration with the community is required.

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100 and a grant from JetBrains Research.

REFERENCES

[1] Martin Albrecht and Gregory Bard. 2019. *The M4RI Library.* The M4RI Team. https://bitbucket.org/malb/m4ri
[2] MR Albrecht, GV Bard, and W Hart. 2008. Efficient multiplication of dense matrices over GF (2). *arXiv preprint arXiv:0811.1714* (2008).
[3] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. 1970. On economical construction of the transitive closure of an

$$s \to s\ s \qquad s \to A$$

(a) Simple ambiguous grammar G_2

$$s \to s\ s\ s \qquad s \to A$$

(b) Highly ambiguous grammar G_3

Figure 3: Queries for the `[Full]` dataset

$$s \to SCOR\ s\ SCO \qquad s \to TR\ S\ T \qquad s \to TR\ T \qquad s \to SCOR\ SCO$$

(a) Same generation query G_4

$$s \to SCOR\ s\ SCO \qquad s \to SCO$$

(b) Same generation query G_5

Figure 4: Queries for the `[RDF]` dataset

oriented graph. In *Doklady Akademii Nauk*, Vol. 194. Russian Academy of Sciences, 487–488.

[4] 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

[5] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh Patel. 2018. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *arXiv preprint arXiv:1812.03975* (2018).

[6] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.

[7] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[8] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–2019. SciPy: Open source scientific tools for Python. http://www.scipy.org/ [Online; accessed 5.3.2019].

[9] X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. 2016. Context-free path queries on RDF graphs. In *International Semantic Web Conference*. Springer, 632–648.

A DETAILS OF DATASET DESCRIPTION

Here we present details on collected dataset. Grammars (queryes) are stored in the files with `yrd` extension. Each line is a rule in the form of a triple or a pair. The example of grammar representation is presented in figure 1.

Graphs and are stored in the files with `txt` extension. Example of graph is presented in figure 2.

Cases for evaluation are placed in folders with the case-specific name. Grammars and graph are placed in subfolders with names `Grammars` and `Matrices` respectively.

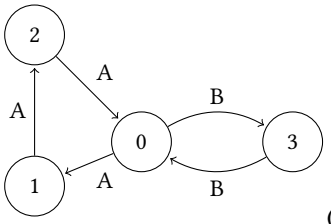
$$s \to A\ s\ B \qquad s \to A\ B$$

(a) Grammar G_1

```
s a b
s a s1
s1 s b
a A
b B
```

(b) Representation of grammar G_1 in `yrd` file

Figure 1: Example of grammar representation in the `yrd` file



(a) Example of input graph D_1

```
0 A 1
1 A 2
2 A 0
0 B 3
3 B 0
```

(b) Representation of the input graph D_1 in `txt` file

Figure 2: Example of graph representation in `txt` file

Queries which we use for evaluation presented in figures **??**.