SAINT-PETERSBURG STATE UNIVERSITY

On the rights of the manuscript

Rustam Azimov

Context-Free Path Querying Using Linear Algebra

Scientific specialty
2.3.5. Mathematical and software support
for computers, complexes and computer networks

DISSERTATION
for the degree of Candidate of Physico-Mathematical Sciences

Translation from Russian

Scientific supervisor:
PhD
Semyon Grigorev

Saint Petersburg
2022

Contents

Page

Introduction

In the modern world, there is more and more data that requires processing and analysis. At the same time, graphs are one of the most common and convenient structures, allowing us to compactly represent large amounts of information and implement efficient algorithms for its analysis. Graphs are used in static program analysis [1; 2], bioinformatics [3], social networks [4], network analysis [5], etc. Also, graph databases, used to store data in the form of graphs and implement queries to them, are currently being actively developed. For example, there are such popular graph databases as RedisGraph[1] and Neo4j[2].

One of the most important graph analysis problems is path querying. There are different variations of this problem: the direct search for certain paths, the reachability problem (the paths themselves are not requested, but it is necessary to prove their existence), etc.

In the path querying problems special properties of the desired paths can be specified. Such properties can be described, for example, using a formal language over some alphabet [6]. In this way, one can constrain the set of words obtained by concatenation of labels on the edges of some path. Thus, in path querying problems for a given labeled graph and a formal language, only paths that form words from this language are requested. Such problems are also called formal language-constrained path querying problems. Constraints represented as context-free languages (CFLs) are used in context-free path querying (CFPQ) problem and are being actively studied [1; 7—10]. CFLs allow one to describe a wider set of constraints than regular expressions that are actively used in practice [11; 12].

From a practical point of view, one of the common ways to obtain high-performance implementations of graph analysis algorithms is to use linear algebra methods [13]. Hence, the existing algorithms are translated into the language of sparse linear algebra. As a result, sparse matrices (matrices with a small number of nonzero elements) are used to represent a graph, and matrix operations (matrix multiplication, addition, matrix transposition, etc.) are used to perform graph analysis. For example, a graph can be represented using the adjacency matrix, and such a transformation of a directed graph as inverting the edge directions can be

[1] Graph database RedisGraph: https://oss.redislabs.com/redisgraph/ (date of access: 14.01.2022).

[2] Graph database Neo4j: https://neo4j.com/ (date of access: 14.01.2022).

done by the adjacency matrix transposition. For those graph analysis algorithms that allow such translation, it becomes possible to use parallel computing, in particular, based on GPU technologies, which can significantly improve their performance. In addition, such algorithms are often easy to implement, as they allow one to use the existing linear algebra libraries (SuiteSparse:GraphBLAS[3], cuSPARSE[4], cuBLAS[5], cuBool[6], m4ri[7], Scipy[8], etc.).

However, there have been no studies so far on the possibility of applying the linear algebra methods to CFPQ problem. The existing solutions to this problem suffer from insufficient performance and cannot cope with the constantly growing sizes of real graphs [14]. At the same time, the creation of new linear algebra solutions allows one to solve this problem using the theoretical and practical linear algebra results.

Recently, a significant number of works have appeared devoted to classical graph analysis algorithms translated into the linear algebra language. For example, Aydin Buluç, Upasana Sridhar, Peter Zhang, Ariful Azad, and Leyuan Wang in their works [15—19] have shown the practical applicability of algebraic versions of such algorithms as breadth-first search, Dijkstra's algorithm, Bellman-Ford's algorithm, algorithm for finding a matching in a bipartite graph, and algorithm for triangle counting.

Recently, the idea of using linear algebra methods to solve CFPQ problem has become very popular. Hence, the GraphBLAS [20] standard that defines the basic building blocks of graph analysis algorithms in terms of linear algebra was recently created. This standard uses the adjacency matrices for graph representation and matrix operations to perform graph analysis. Also, since the real data are often sparse, it makes sense to use a sparse format for these matrices. Note that not every graph analysis algorithm can be formulated in terms of linear algebra. For example,

[3] SuiteSparse:GraphBLAS is a `C` implementation of the GraphBLAS standard: https://github.com/DrTimothyAldenDavis/GraphBLAS (date of access: 14.01.2022).

[4] GPU-based linear algebra library cuSPARSE for computing sparse matrix operations: https://docs.nvidia.com/cuda/cusparse/index.html (date of access: 14.01.2022).

[5] GPU-based linear algebra library cuBLAS: https://docs.nvidia.com/cuda/cublas/index.html (date of access: 14.01.2022).

[6] Linear algebra library cuBool for computing sparse Boolean matrix operations on GPU: https://github.com/JetBrains-Research/cuBool (date of access: 14.01.2022).

[7] m4ri is a library for fast arithmetic with dense Boolean matrices: https://github.com/malb/m4ri (date of access: 14.01.2022).

[8] Scipy is a free and open-source `Python` library used for scientific and technical computing: https://scipy.org/ (date of access: 14.01.2022).

it is still not done for the depth-first search algorithm [21]. Also, at present, it is still not done for CFPQ algorithms.

The CFPQ problem is one of the important graph analysis problems. Its partial case is the problem of context-free recognition, where strings are analyzed. Leslie Valiant has done research [22] on context-free recognition using matrix operations. He proposed a subcubic algorithm that for a given string and a context-free grammar determines whether this string is generated by a given grammar using Boolean matrix multiplication. For the first time, the question of the possibility of finding a matrix-based CFPQ algorithm was raised by Mihalis Yannakakis [23]. He pointed out that Valiant's algorithm could be extended to analyze graphs without cycles (DAGs), but doubted the possibility of creating a subcubic algorithm to analyze arbitrary graphs.

However, for a partial case of CFPQ problem, there is an algorithm formulated in the language of linear algebra. Such an algorithm was proposed by Philip Bradford [8], who studied the reachability problem in a graph with given particular context-free path constraints.

In addition, there are a number of papers [9; 24—26] devoted to CFPQ problem with an arbitrary graph and arbitrary context-free path constraints that are based on various parsing techniques (LR, LL, GLL, CYK). In these works, Semyon Grigorev, Jelle Hellings, Ciro Medeiros, and Martin Musicante proposed algorithms that require the context-free path constraints to be represented as a context-free grammar (CFG) in some normal form. Algorithms that do not require additional transformations of structures that describe input path constraints are of particular interest, since almost any such transformation usually leads to an increase in the size of these structures, and can adversely affect performance. In addition, after such transformations, it may be difficult to interpret graph analysis results in terms of the original structure specified by the user. Examples of CFPQ algorithms that do not require transformations of the input CFG are the algorithms [24; 26] based on the LL and GLL parsing algorithms.

Thus, at the moment, there is no linear algebra-based CFPQ algorithm for an arbitrary graph and arbitrary context-free path constraints. Therefore, it is necessary to explore the possibility of developing such algorithms.

The goal of this work is to study the applicability of linear algebra methods to the context-free path querying problem in order to obtain high-performance implementations based on parallel computing.

Achieving this goal is ensured by solving the following **tasks**.

1. To develop an approach to the context-free path querying based on linear algebra methods.
2. To devise a CFPQ algorithm that uses the proposed approach.
3. To devise a CFPQ algorithm that uses the proposed approach and does not require a transformation of the input context-free grammar.
4. To implement the devised algorithms using parallel computing, conduct their experimental study on real data, and compare them with existing implementations.

Theoretical and practical influence. The theoretical influence of this thesis research lies in the development of a linear algebra based approach to the CFPQ, in the formal algorithms development using the obtained approach, as well as in the formal proof of the termination, correctness, and the time complexity of the developed algorithms.

In this thesis, the proposed algorithms were implemented using parallel computing that allowed us to obtain up to 3 orders of magnitude faster graph analysis time and consume up to 2 orders of magnitude less memory compared to existing solutions. In addition, the implementations made can be integrated with graph databases such as RedisGraph. This will extend the query languages for these databases.

Methodology and research methods. The research methodology is based on the linear algebra and the graph theory. The work uses the GraphBLAS standard that combines these areas. In addition, the formal language theory was used in this work, as well as the complexity theory. Finally, CPU and GPU technologies were used to implement the algorithms.

The main results submitted for defense.

1. An approach to the context-free path querying based on linear algebra methods that allows one to use theoretical and practical linear algebra results was developed.
2. A CFPQ algorithm that uses the proposed approach was devised. Termination and correctness of the devised algorithm were proved, as well as its time complexity. The proposed algorithm uses matrix operations, which make it possible to apply a wide class of optimizations and allows one to automatically parallelize computations using existing linear algebra libraries.

3. A CFPQ algorithm that uses the proposed approach and does not require a transformation of the input context-free grammar was devised. Termination and correctness of the devised algorithm were proved, as well as its time complexity. The proposed algorithm makes it possible to work with arbitrary input context-free grammars without any transformations. This allows one to avoid a significant increase in the grammar size that affects analysis performance.
4. The devised algorithms are implemented using parallel computing techniques. An experimental study of the devised algorithms was provided using real RDF data and graphs built for static program analysis. The obtained implementations were compared with each other, with existing solutions from the field of static program analysis, and with solutions based on various parsing techniques. The comparison results show that the proposed implementations for the reachability problem allow one to obtain up to 2 orders of magnitude faster graph analysis time and consume up to 2 times less memory in comparison with existing solutions, and for the problems of finding one and all paths in a graph allow one to speed up the analysis time up to 3 orders of magnitude and consume up to 2 orders of magnitude less memory.

Scientific novelty.

1. A new approach to the CFPQ that allows one to use theoretical and practical linear algebra results is proposed.
2. For the first time, a linear algebra based CFPQ algorithm for an arbitrary graph and arbitrary context-free path constraints was obtained. This algorithm makes it possible to apply a wide class of matrix optimizations, to use parallel computing techniques, and to significantly improve the performance.
3. In this thesis, a linear algebra based CFPQ algorithm that does not require a transformation of the input context-free grammar was also proposed, in contrast to the algorithms proposed in the works of Semyon Grigorev, Jelle Hellings, and Philip Bradford. Thus, the algorithm proposed in this thesis allow one to avoid a significant increase in the grammar size that affects analysis performance.

The reliability and approbation of the results. The reliability and approbation of the research results are based on the use of formal proof methods and engineering experiments.

The main results of the work were presented at a number of international scientific conferences: Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (co-located with the SIGMOD conference) 2018 (Houston, Texas, USA), 2020 (Portland, Oregon, USA), and 2021 (Xi'an, Shaanxi, China); 24th European Conference on Advances in Databases and Information Systems (ADBIS) 2020 (Lyon, France); VLDB PhD Workshop in cooperation with the 47th International Conference on Very Large Data Bases 2021 (Copenhagen, Denmark). The study was also supported by the RSF grant №18-11-00100 and the RFBR grant №19-37-90101.

Publications. All results of this dissertation are presented in 8 scientific papers [27—34]. 7 works [27—32; 34] are indexed in the Scopus database. Works [27—30; 32—34] are made with co-authors. In the paper [27], the author owns the development and implementation of an algorithm that solves the CFPQ problem with the reachability query semantics using linear algebra methods, proof of the algorithm correctness, setting up experiments; co-authors participated in the discussion of the main ideas of the paper, carried out a review of the research area. In the paper [28], the author owns the development and implementation of an algorithm that solves the single-path CFPQ problem using linear algebra methods, proof of the algorithm correctness; co-authors conducted an experimental study, participated in the formalization and improvement of the paper. In the work [29], the author's contribution consists in proving the correctness of the CFPQ algorithm that does not requires a transformation of the input grammar, as well as in working on the text; the co-authors own the idea of the algorithm and setting up the experiments. In the work [30], the author owns the development and implementation of the all-path CFPQ algorithm using linear algebra methods, proof of the algorithm correctness, work on the text; co-authors conducted an experimental study. In the papers [32; 33], the author's contribution is developing and implementing a path querying algorithm that solves the reachability problem in a graph with given path constraints in the form of conjunctive languages, proving the algorithm correctness, and setting up experiments; co-authors participated in the discussion of the main ideas of the paper and performed a review of the research area. In the work [34], the author is responsible for the development and implementation of the all-path

CFPQ algorithm using matrices with sets of intermediate vertices, as well as work on the text; co-authors conducted an experimental study.

Acknowledgments. First of all, I would like to thank my supervisor, Semyon Grigorev, for his guidance at all stages of this research, for his willingness to support and share his experience, and for his invaluable contribution to my work. I would also like to thank Dmitry Koznov for his wisdom, active participation, and numerous conversations about my dissertation, which had a great impact on my work and on me in general.

I express my gratitude to Andrey Terekhov and the Department of System Programming of St. Petersburg State University, to Dmitry Bulychev, Andrey Ivanov, as well as to JetBrains and Huawei for the unique opportunity to engage in science as the main activity.

I am grateful to Vladimir Kutuev and Vlada Pogozhelskaya for their assistance with the experiments.

I would like to express special gratitude to my wife, Svetlana Azimova, and my son, Artyom Azimov, for inspiration, love, and support, because they occupy all my thoughts and all my heart, and I do all things in my life exactly for the sake of them. I also want to thank my brother, Timur Azimov, for sharing my interests and supporting. Finally, I am grateful to my dear parents, Shukhratullo Azimov and Elena Azimova, who have been my support throughout my life and made me who I am today.

Scope and structure of work. The thesis consists of an introduction, 6 chapters, and a conclusion. The full scope of the dissertation is 121 pages, including 20 figures and 17 tables. The list of references contains 79 titles.

Chapter 1. Background

In this chapter, we introduce the main terms and definitions used in the work, provide a formal statement of CFPQ problem, and consider existing CFPQ algorithms. In addition, the main ideas of using linear algebra methods for solving graph analysis problems are considered, as well as existing linear algebra libraries that can be used to obtain the corresponding high-performance implementations.

1.1 Basic Concepts of the Linear Algebra

In this section, we introduce a number of terms and definitions of the linear algebra used in the work.

Definition 1.1.1 (An algebraic structure)**.** An *algebraic structure* consists of a nonempty set $\mathcal{S}$ (a domain), a collection of operations on $\mathcal{S}$ of finite arity (typically binary operations), and a finite set of axioms.

Next, we provide definitions for such algebraic structures as semigroups, monoids, groups, and semirings.

Definition 1.1.2 (A semigroup)**.** The set $\mathcal{S}$ with the binary operation $\circ : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ defined on it is called a *semigroup* and denoted by $\langle \mathcal{S}, \circ \rangle$ if the following axiom holds.

- Associativity: $\forall (a, b, c \in \mathcal{S})\colon\ (a \circ b) \circ c = a \circ (b \circ c)$.

Definition 1.1.3 (A monoid)**.** A *monoid* $\langle \mathcal{S}, \circ, \mathbb{0} \rangle$ is a semigroup with element $\mathbb{0} \in \mathcal{S}$ such that the following axiom holds.

- Identity (neutral) element $\mathbb{0}$: $\forall a \in \mathcal{S}\colon (\mathbb{0} \circ a = a \circ \mathbb{0} = a)$.

Definition 1.1.4 (A group)**.** A nonempty set $\mathcal{S}$ with a binary operation $\circ : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ is called a *group* $\langle \mathcal{S}, \circ, \mathbb{0} \rangle$ if it is a monoid with identity element $\mathbb{0}$ and an additional requirement for the existence of inverse elements.

- Inverse elements: $\forall a \in \mathcal{S} \quad \exists a^{-1} \in \mathcal{S}\colon (a \circ a^{-1} = a^{-1} \circ a = \mathbb{0})$.

Definition 1.1.5 (A semiring)**.** Nonempty set $\mathcal{S}$ with two binary operations $\oplus$ (often called addition) and $\otimes$ (often called multiplication) is called a *semiring* $\langle \mathcal{S}, \oplus, \otimes, \mathbb{0} \rangle$ if the following axioms hold.

1. $\langle \mathcal{S}, \oplus, \mathbb{0} \rangle$ is a commutative monoid, i.e. for any $a, b, c \in \mathcal{S}$ we have:
   - $(a \oplus b) \oplus c = a \oplus (b \oplus c)$,
   - $\mathbb{0} \oplus a = a \oplus \mathbb{0} = a$,
   - $a \oplus b = b \oplus a$.
2. $\langle \mathcal{S}, \otimes, \mathbb{1} \rangle$ is a monoid with some neutral element $\mathbb{1} \in \mathcal{S}$, and $\mathbb{0}$ is *absorbing* for $\otimes$, i.e. for any $a, b, c \in \mathcal{S}$ the following axioms hold:
   - $(a \otimes b) \otimes c = a \otimes (b \otimes c)$,
   - $\mathbb{1} \otimes a = a \otimes \mathbb{1} = a$,
   - $\mathbb{0} \otimes a = a \otimes \mathbb{0} = \mathbb{0}$.
3. $\otimes$ distributes over $\oplus$:
   - $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$,
   - $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.

If the operation $\otimes$ is commutative, then this algebraic structure is called a commutative semiring.

In addition, on the domains of the introduced algebraic structures, a partial order relation can be specified, which is defined as follows.

Definition 1.1.6 (A partial order relation)**.** A binary relation $\preceq$ on a set $\mathcal{S}$ is called a *partial order relation* if the following conditions are satisfied for any $a, b, c \in \mathcal{S}$.

- Reflexivity: $a \preceq a$.
- Antisymmetry: if $a \preceq b$ and $b \preceq a$ then $a = b$.
- Transitivity: if $a \preceq b$ and $b \preceq c$ then $a \preceq c$.

Next, we introduce the concepts of a matrix and a vector, and also define some operations on them.

Definition 1.1.7 (A matrix)**.** Assume that there is some algebraic structure with set $\mathcal{S}$ as domain. Then a *matrix over this algebraic structure* is a rectangular array of size $n \times m, n > 0, m > 0$ filled with elements from the set $\mathcal{S}$. Such matrix has n rows and m columns, while n and m are called its dimensions.

Indexes are used to access matrix elements. Indexing in matrices starts from the upper left corner, the element's row is indicated first and its column is indicated second. In this thesis, we will use indexing of rows and columns starting from zero. For example, $M[0,0]$ is a matrix element located at the intersection of the top row and left column of the matrix M.

Definition 1.1.8 (A vector)**.** If a matrix has only one row or only one column it is called a *vector*. A matrix having only one row is called a *row vector*, and a matrix having only one column is called a *column vector*.

Definition 1.1.9 (A scalar multiplication)**.** For a semigroup $\langle \mathcal{S}, \circ \rangle$, the matrix $M_{n\times m}$ over this semigroup, and for $x \in \mathcal{S}$, the result of applying the operation $scalar(M, x, \circ)$ is a matrix $P_{n\times m}$ such that $P[i,j] = M[i,j] \circ x$, and $scalar(x, M, \circ) = P_{n\times m}$ where $P[i,j] = x \circ M[i,j]$.

Definition 1.1.10 (Element-wise operations)**.** For a semiring $\langle \mathcal{S}, \oplus, \otimes, \mathbb{0} \rangle$, and matrices $M_{n\times m}$, $N_{n\times m}$ of the same size over this semiring, the result of applying the *element-wise addition* operation $\bigoplus$ is a matrix $P_{n\times m} = M \bigoplus N$ such that $P[i,j] = M[i,j] \oplus N[i,j]$. And the result of applying the *element-wise multiplication* operation $\bigotimes$ is a matrix $P_{n\times m} = M \bigotimes N$ such that $P[i,j] = M[i,j] \otimes N[i,j]$.

Definition 1.1.11 (A matrix multiplication)**.** For a semiring $\langle \mathcal{S}, \oplus, \otimes, \mathbb{0} \rangle$, and matrices $M_{n\times m}$, $N_{m\times k}$ over this semiring, the matrix $P_{n\times k} = M \cdot N$ defined as $P[i,j] = \bigoplus_{0 \leqslant l < m} M[i,l] \otimes N[l,j]$.

Definition 1.1.12 (The Kronecker product)**.** For a semigroup $\langle \mathcal{S}, \circ \rangle$, and matrices $M_{m\times n}$, $N_{p\times q}$ over this semigroup, the *Kronecker product* of the matrices M and N is the following block matrix C of size $mp \times nq$.

$$C = M \times N = \begin{pmatrix} scalar(M[0,0], N, \circ) & \cdots & scalar(M[0, n-1], N, \circ) \\ \vdots & \ddots & \vdots \\ scalar(M[m-1, 0], N, \circ) & \cdots & scalar(M[m-1, n-1], N, \circ) \end{pmatrix}$$

1.2 Basic Concepts of the Graph Theory

In this section, we introduce a number of terms and definitions of the graph theory used in the work.

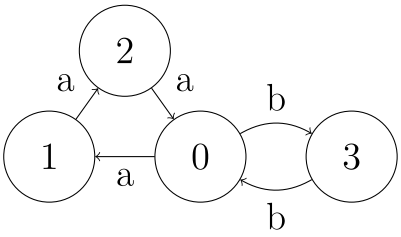


Figure 1.1 — An example of a graph

Definition 1.2.1 (A graph)**.** A *graph* $\mathcal{G}$ is a tuple $\langle V, E, L\rangle$ where V is a finite set of vertices, $E \subseteq V \times L \times V$ is a finite set of labeled edges, and L is a finite set of edge labels.

Further in the work, we will use the term *graph* meaning just a finite labeled directed graph, unless otherwise stated.

For simplicity, we assume that the vertices are natural numbers ranging from 0 to $|V| - 1$ where $|V|$ is a number of graph vertices.

Example 1.2.1 (An example of a graph)**.** The graph

$$\mathcal{G}_1 = \langle\{0, 1, 2, 3\}, \{(0, a, 1), (1, a, 2), (2, a, 0), (0, b, 3), (3, b, 0)\}, \{a, b\}\rangle$$

is presented in Figure 1.1.

Definition 1.2.2 (An edge)**.** *Edge* of a labeled directed graph $\mathcal{G} = \langle V, E, L\rangle$ is an ordered triple $e = (v_i, l, v_j) \in V \times L \times V$.

Example 1.2.2 (An example of graph edges)**.** $(0, a, 1)$ and $(3, b, 0)$ are the edges of the graph $\mathcal{G}_1$ presented in Figure 1.1. At the same time, $(3, b, 0)$ and $(0, b, 3)$ are distinct edges.

Definition 1.2.3 (Multiple edges)**.** Two distinct edges $e_1 = (u_1, l_1, v_1)$ and $e_2 = (u_2, l_2, v_2)$ of a labeled directed graph $\mathcal{G} = \langle V, E, L\rangle$ are called *multiple* if $u_1 = u_2$ and $v_1 = v_2$.

Definition 1.2.4 (A path)**.** A *path* π in a graph $\mathcal{G}$ is a sequence of labeled edges such that for any two successive edges $e_1 = (u_1, l_1, v_1)$ and $e_2 = (u_2, l_2, v_2)$ in this sequence, the end vertex of the first edge is the start vertex of the second one, i.e. $v_1 = u_2$. We will denote the path from v_0 to v_n as follows:

$$v_0 \pi v_n = e_0, e_1, \ldots, e_{n-1} = (v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_n, v_n).$$

Definition 1.2.5 (A cycle)**.** A *cycle* is a path π in a graph $\mathcal{G}$ such that its initial and final vertices are equal, and all edges of this path are distinct.

In this work, we assume that for the set L of edge labels the concatenation operation $(\cdot) : L^* \times L^* \to L^*$ is always defined. The concatenation operation will often be omitted: $l_1 \cdot l_2 = l_1 l_2$.

Definition 1.2.6 (The word formed by a path)**.** We will denote the *word formed by a path* $\pi = (v_0, l_0, v_1), (v_1, l_1, v_2), \dots, (v_{n-1}, l_n, v_n)$ as the word $\lambda(\pi) = l_0 l_1 \dots l_n$ obtained by concatenating edge labels along the path.

Example 1.2.3 (An example of paths)**.** $(0, a, 1), (1, a, 2) = 0\pi_1 2$ is a path from the vertex 0 to the vertex 2 in the graph $\mathcal{G}_1$. At the same time, $(0, a, 1), (1, a, 2), (2, b, 3), (3, b, 2) = 0\pi_2 2$ is also a path from the vertex 0 to the vertex 2 in this graph, although it is different from the path $0\pi_1 2$. Moreover, $\lambda(\pi_1) = aa$ and $\lambda(\pi_2) = aabb$.

We also define the notion of a negative cycle for weighted graphs that associate each edge with a real number. Such graphs form a partial case of labeled graphs.

Definition 1.2.7 (A negative cycle)**.** A *negative cycle* in a finite weighted graph $\mathcal{G} = \langle V, E, \mathbb{R} \rangle$ is a cycle with a negative sum of edge weights.

In addition, we will use a relation that reflects the fact that there is a path between two vertices.

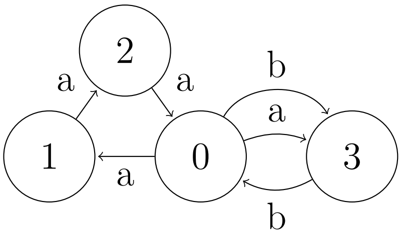
Definition 1.2.8 (The reachability)**.** The *reachability* is a binary relation P over the graph vertices, where: $(v_i, v_j) \in P \iff \exists v_i \pi v_j$.

We also define the concept of an adjacency matrix that is one of the common matrix representation of graphs.

Definition 1.2.9 (An adjacency matrix)**.** An *adjacency matrix* M of the graph $\mathcal{G} = \langle V, E, L \rangle$ is a square matrix of size $n \times n$, where $|V| = n$, and its cells contain sets of the edge labels. Specifically, $l \in M[i, j] \iff \exists e = (i, l, j) \in E$.

Note that our definition of the adjacency matrix differs from the classical one, in which the matrix is Boolean and reflects only the fact of the presence of at least one edge.

Example 1.2.4 (An example of a graph and its adjacency matrix)**.** An example of a labeled graph is shown below.



The adjacency matrix for this graph is the following.

$$M = \begin{pmatrix} \varnothing & \{a\} & \varnothing & \{a,b\} \\ \varnothing & \varnothing & \{a\} & \varnothing \\ \{a\} & \varnothing & \varnothing & \varnothing \\ \{b\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Such adjacency matrices can be represented as a set of Boolean matrices. For brevity, the value *true* of the Boolean variable will be denoted by 1, and the value $false$ — by 0.

Definition 1.2.10 (The Boolean decomposition of an adjacency matrix)**.** The *Boolean decomposition of an adjacency matrix* M of the graph $\mathcal{G} = \langle V, E, L \rangle$ is the set of Boolean matrices $\mathcal{M} = \{M^l \mid l \in L, M^l[i,j] = 1 \iff l \in M[i,j]\}$.

Example 1.2.5 (The Boolean decomposition of the matrix M from example 1.2.4)**.** The Boolean decomposition $\mathcal{M}$ consists of matrices M^a and M^b.

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

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

Definition 1.2.11 (The Kronecker product of two graphs)**.** For two labeled graphs $\mathcal{G}_1 = \langle V_1, E_1, L_1 \rangle$ and $\mathcal{G}_2 = \langle V_2, E_2, L_2 \rangle$, the *Kronecker product of graphs* $\mathcal{G}_1$ and $\mathcal{G}_1$ is a graph $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2 = \langle V, E, L \rangle$ where

- $V = V_1 \times V_2$,
- $E = \{((u,v), l, (p,q)) \mid (u,l,p) \in E_1 \wedge (v,l,q) \in E_2\}$,
- $L = L_1 \cap L_2$.

From this definition, as well as from the definition 1.1.12, it follows that the Kronecker product of the adjacency matrices of the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ defined for the semigroup $\langle 2^{L_1} \cup 2^{L_2}, \cap\rangle$, is the adjacency matrix of the Kronecker product $\mathcal{G}_1 \times \mathcal{G}_2$ of these graphs. Here 2^L is the set of all subsets of a set L.

1.3 Basic Concepts of the Formal Language Theory

In this section, we introduce a number of terms and definitions of the formal language theory used in the work.

Definition 1.3.1 (An alphabet)**.** An *alphabet* Σ is a nonempty set of *symbols* (also called *terminal symbols* or *terminals*).

We also assume that the concatenation operation $(\cdot) : \Sigma^* \times \Sigma^* \to \Sigma^*$ is always defined for the alphabet Σ.

Definition 1.3.2 (A word)**.** A *word* (also called *string*) over the alphabet Σ is a finite sequence of concatenated symbols: $w = a_0 \cdot a_1 \cdot \ldots \cdot a_m$ where w is a word, for all $a_i \in \Sigma$.

Definition 1.3.3 (Word length)**.** For a word $w = a_0 \cdot a_1 \cdot \ldots \cdot a_m$ over the alphabet Σ, the *length* $|w|$ of this word is a number of symbols in the sequence, i.e. $|w| = m + 1$. In addition, the empty word will be denoted by ε where $|\varepsilon| = 0$.

Definition 1.3.4 (A language)**.** A *language* over an alphabet Σ is a set of words over this alphabet.

As an example of languages, one can cite the language of all binary numbers $\{0, 1, 10, 11, 100 \ldots\}$, as well as the Dyck language of all balanced words of square brackets $\{[\,], [\,[\,]\,], [\,][\,], [\,[\,]\,]\,[\,], \ldots\}$.

Any language over the alphabet Σ is a subset of Σ^* — the set of all words over this alphabet. Note that a language may be an infinite set.

Definition 1.3.5 (A formal grammar)**.** A *formal grammar* G is a tuple $\langle \Sigma, N, P, S \rangle$ where:

- Σ is a finite set of terminal symbols (or terminals),
- N is a finite set of *nonterminal symbols* (or *nonterminals*), and $\Sigma \cap N = \varnothing$,
- P is a finite subset of the set $(\Sigma^* \cdot N \cdot (\Sigma \cup N)^*) \times ((\Sigma \cup N)^+ \cup \{\varepsilon\})$,
- $S \in N$ is the start nonterminal.

The element $(a, b) \in P$ is called the *derivation rule* and is written as follows: $a \to b$. In this case, a is called the *left-hand side* of the rule, and b — the *right-hand side*. The left-hand side of any rule in P must contain at least one nonterminal.

Definition 1.3.6 (A derivation of a word)**.** We use the notation $w_1 \to_G w_2$ to denote that a sequence of symbols $w_2 \in (\Sigma \cup N)^*$ can be derived from a sequence $w_1 \in (\Sigma \cup N)^+$ where $w_1 = x_1 \cdot y \cdot x_2, w_2 = x_1 \cdot z \cdot x_2$ for some $x_1, x_2, z \in (\Sigma \cup N)^*$ and $y \in (\Sigma \cup N)^+$, and there is a derivation rule $(y \to z)$ in this grammar. The index G in the notation $\to_G$ can be omitted if the grammar is clear from the context.

Also, we use the notation $w_1 \Rightarrow_G w_2$ to denote, that there are sequences $z_0, z_1, \ldots, z_n (n \geqslant 0)$ such that $w_1 = z_0 \to z_1 \to \ldots \to z_n = w_2$.

Definition 1.3.7 (A language generated by a grammar)**.** A language *generated* by a grammar $G = \langle \Sigma, N, P, S \rangle$ is a set of words $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow w\}$.

The regular and the context-free languages are important classes of languages generated by formal grammars [35]. One common way to describe the regular languages (generated by the regular grammars) is to specify them using the regular expressions.

Definition 1.3.8 (A regular expression)**.** A sequence of symbols R is called a *regular expression* over an alphabet Σ if its structure matches one of the following:

- a where $a \in \Sigma$;
- ε where ε is an empty word;
- $\varnothing$ (corresponds to the empty language);
- $R_1 \mid R_2$ (the disjunction) where R_1 and R_2 are regular expressions;
- $R_1 \cdot R_2$ (the concatenation) where R_1 and R_2 are regular expressions;
- $(R_1)^*$ (the Kleene star) where R_1 is a regular expression.

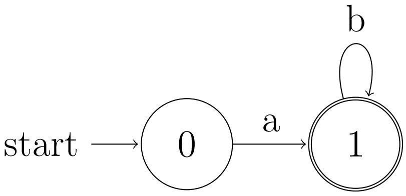
For example, the regular expression $R = a \cdot (b)^*$ over the alphabet $\Sigma = \{a, b\}$ describes a language consisting of words that begin with the symbol a followed by some (zero or more) number of symbols b.

Definition 1.3.9 (A deterministic finite automaton)**.** A *deterministic finite automaton* (DFA) T without ε-transitions is a tuple $\langle \Sigma, Q, Q_s, Q_f, \delta \rangle$ where:

- Σ is a finite alphabet of input symbols,
- Q is a finite set of states,
- $q_s \in Q$ is the start (or initial) state,
- $Q_f \subseteq Q$ is a set of final states,
- $\delta : Q \times \Sigma \rightharpoonup Q$ is a partial transition function.

It is known that any regular expression can be expressed using a deterministic finite automaton without ε-transitions [36]. Moreover, the finite deterministic automaton $T = \langle \Sigma, Q, q_s, Q_f, \delta \rangle$ can be naturally represented as a labeled graph $\mathcal{G} = \langle V, E, L \rangle$ where $V = Q$, $L = \Sigma$, $E = \{(q_i, l, q_j) \mid \delta(q_i, l) = q_j\}$, and some vertices are labeled as initial or final states. Thus, the adjacency matrix of such a graph representation contain information about the transition function δ.

Example 1.3.1 (A graph representation example for the regular expression $a \cdot (b)^*$)**.** The graph vertex corresponding to the initial state of the DFA will be marked with the word *start*, and the vertices corresponding to the final states will be marked with a double circle. In this example, the vertex 0 corresponds to the initial state, and the vertex 1 corresponds to the final state.



Definition 1.3.10 (DFA intersection)**.** For two DFAs $T_1 = \langle \Sigma, Q^1, q^1_s, Q^1_f, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, q^2_s, Q^2_f, \delta^2 \rangle$ be given, the *intersection* of these two automata is the new DFA $T = \langle \Sigma, Q, q_s, Q_f, \delta \rangle$ such that:

- $Q = Q^1 \times Q^2$;
- $q_s = \langle q^1_s, q^2_s \rangle$;
- $Q_f = Q^1_f \times Q^2_f$;
- $\delta : Q \times \Sigma \rightharpoonup Q$;
- $\delta(\langle q_1, q_2 \rangle, s) = \langle q'_1, q'_2 \rangle$ if $\delta(q_1, s) = q'_1$ and $\delta(q_2, s) = q'_2$.

According to [36], the DFA T, which is the intersection of the DFAs T_1 and T_2, recognizes the language $L_1 \cap L_2$ where L_1 is the language recognized by the automaton T_1, and L_2 is the language recognized by the automaton T_2. In addition, the intersection of two DFAs can be described using the adjacency matrices of these automata graph representations.

Definition 1.3.11 (The Kronecker product of two Boolean matrix decompositions)**.** For a semigroup $\langle \{0, 1\}, \wedge \rangle$, and Boolean decompositions $\mathcal{M}_1$ and $\mathcal{M}_2$ of adjacency matrices corresponding to graph representations of DFAs $T_1 = \langle \Sigma, Q^1, q_s^1, Q_f^1, \delta^1 \rangle$ and $T_2 = \langle \Sigma, Q^2, q_s^2, Q_f^2, \delta^2 \rangle$, the *Kronecker product of Boolean decompositions* of this matrices is the $\mathcal{M}_1 \times \mathcal{M}_2 = \{M_1^a \times M_2^a \mid a \in \Sigma\}$.

According to [37], the following theorem holds.

Theorem 1.3.1 (Intersection of two DFAs and the Kronecker product)**.** For the intersection T of DFAs T_1 and T_2, and Boolean decompositions $\mathcal{M}$, $\mathcal{M}_1$, and $\mathcal{M}_2$ of adjacency matrices corresponding to graph representations of DFAs T, T_1, and T_2, the following holds: $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$.

Thus, the transition function of DFAs intersection can be computed using a series of Kronecker products over Boolean matrices.

Another important class of formal languages that generalizes the class of the regular languages is the context-free languages (CFLs) generated by grammars of the following form.

Definition 1.3.12 (A context-free grammar)**.** A *context-free grammar* (CFG) G is a tuple $\langle \Sigma, N, P, S \rangle$ where

- Σ is a finite set of terminals,
- N is a finite set of nonterminals, and $\Sigma \cap N = \varnothing$,
- P is a finite subset of the set $N \times ((\Sigma \cup N)^+ \cup \{\varepsilon\})$,
- $S \in N$ is the start nonterminal.

To represent the derivation of a word in a CFG, the derivation trees are usually used.

Definition 1.3.13 (A derivation tree)**.** A *derivation tree* for a word $w \in \Sigma^*$ and a CFG $G = \langle \Sigma, N, P, S \rangle$ is an ordered rooted tree with the following properties.

- The root is labeled by S.

- If its internal node is labeled by $A \in N$, and $X_1, \ldots, X_k \in \Sigma \cup N$ are labels of this node children from left to right then the rule $A \to X_1 \ldots X_k \in P$.
- If its internal node is labeled by $A \in N$, and ε is the label of the only child of this node then the rule $A \to \varepsilon \in P$.
- The word $w = a_1 \ldots a_m$ where $a_1, \ldots, a_m \in \Sigma \cup \{\varepsilon\}$ are the labels of all leaves of this tree from left to right.

Often the algorithms that use a CFG require it to be in some normal form. Further in this work, we will use the following normal form.

Definition 1.3.14 (A context-free grammar in the weak Chomsky normal form)**.** We say that a CFG $G = \langle \Sigma, N, P, S \rangle$ is in the *weak Chomsky normal form (WCNF)* if the set P contains only rules of the following two types:

- $A \to a$ where $A \in N$ and $a \in (\Sigma \cup \{\varepsilon\})$;
- $A \to BC$ where $A, B, C \in N$.

Note that the described normal form of a CFG differs from the classical Chomsky normal form [38], in which rules of the form $A \to \varepsilon$ are not allowed for all nonterminals A, except for the start nonterminal S. However, this limitation is not essential for use in this work, so the more general normal form will be used.

While a regular expression can be written as a DFA, any CFG can be expressed as a recursive automaton [39].

Definition 1.3.15 (A recursive automaton)**.** A *recursive automaton* R is a tuple $\langle \Sigma, B, m, \{C_i\}_{i \in B} \rangle$ where:

- Σ is a finite alphabet of input symbols,
- B is a finite set of automata labels,
- $m \in B$ is the label of the initial automata,
- $C_i = \langle \Sigma \cup B, Q_i, q_s^i, Q_f^i, \delta_i \rangle$ is a set of automata such that:
  - $\Sigma \cup B$ is a set of transition symbols where $\Sigma \cap B = \varnothing$;
  - Q_i is a finite set of automata states where $Q_i \cap Q_j = \varnothing, \forall i \neq j$;
  - q_s^i is the initial state of automata C_i;
  - $Q_f^i \subseteq Q_i$ is a set of final states,
  - $\delta_i : Q_i \times (\Sigma \cup B) \rightharpoonup Q_i$ is a partial transition function.

The recursive automata, in turn, can be represented as graphs or adjacency matrices for these graphs [39].

Example 1.3.2 (An example of a graph representation of a recursive automaton for a CFG generating the CFL $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$). For a CFG G we have the following derivation rules.

$$\begin{aligned} 0: \quad S &\rightarrow a\ S\ b \\ 1: \quad S &\rightarrow a\ b \end{aligned}$$

Then the recursive automaton $R = \langle \{a, b\}, \{S\}, S, \{C_S\} \rangle$ for the CFG G is presented in Figure 1.2. We will indicate the automata labels in the upper left corner. In this example, the recursive automaton contains only one finite automaton C_S with label S corresponding to the start nonterminal.

An automaton C_S for the nonterminal S

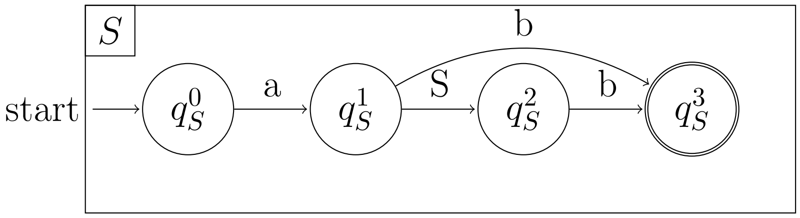


Figure 1.2 — Recursive automaton R for the CFG G

Thus, any CFL can be described using such linear algebra objects as matrices.

1.4 A Formal Statement of the CFPQ Problem

To study the applicability of linear algebra methods to CFPQ problem, we must first introduce a formal statement of this problem. In this section, a formal statement is introduced for three main types of such a problem.

In this problem, we have a labeled graph $\mathcal{G} = \langle V, E, L \rangle$ and a CFL $\mathcal{L}$ as input. The result is information about paths π of the graph $\mathcal{G}$ such that $\lambda(\pi) \in \mathcal{L}$. The type of required information (also called the *query semantics*) depends on the type of a CFPQ problem being solved. There are three typical query semantics:

- the reachability,
- the single-path,
- the all-path.

A formal statement of CFPQ problem with these query semantics is the following.

Reachability. For a labeled graph $\mathcal{G} = \langle V, E, L \rangle$ and a CFL $\mathcal{L}$, it is necessary to find all pairs of vertices $i, j \in V$, between which there is at least one path π such that $\lambda(\pi) \in \mathcal{L}$. In other words, it is necessary to construct the following set:

$$\{(i, j) \mid i, j \in V, \exists i\pi j, \lambda(\pi) \in \mathcal{L}\}.$$

Single-path. For a labeled graph $\mathcal{G} = \langle V, E, L \rangle$, a CFL $\mathcal{L}$, and for all pairs of vertices $i, j \in V$, it is necessary to provide one path π such that $\lambda(\pi) \in \mathcal{L}$ if such a path exists.

All-path. For a labeled graph $\mathcal{G} = \langle V, E, L \rangle$, a CFL $\mathcal{L}$ be given, and for all pairs of vertices $i, j \in V$, it is necessary to provide any finite given number of paths π such that $\lambda(\pi) \in \mathcal{L}$ if there are such a number of paths.

We illustrate the formulations of CFPQ problem on a small graph using the classical CFL $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$ that cannot be described using a regular expression and is used for finding vertices in a graph that are at the same level of the hierarchy [40].

Suppose that there be a graph database or any other object that can be represented as a graph. Then the CFL $\mathcal{L}$ can be used to identify similar vertices in this graph. For graph databases, this problem is aimed at finding all vertices that are at the same level of the hierarchy. For example, consider the small two-cycle graph shown in Figure 1.1. One of the cycles has three edges labeled by a, the other cycle has two edges labeled by b. Both cycles have a shared vertex 0.

Suppose that the result of solving the CFPQ problem with the reachability query semantics and with given CFL is the set R of pairs of vertices. Then, for example, the pair of vertices $(0, 0) \in R$ since there is a path from the vertex 0 to the vertex 0 forming the string $w = aaaaaabbbbbb = a^6 b^6 \in \mathcal{L}$.

The result of solving the CFPQ problem with the single-path query semantics also contains one example of a path of the desired form for each pair of vertices $(i, j) \in R$. For example, if we want to provide a proof of the existence of a path of the desired form between the vertices 0 and 0, then to the result can be added the path $\pi = (0, a, 1), (1, a, 2), (2, a, 0), (0, a, 1), (1, a, 2), (2, a, 0), (0, b, 3), (3, b, 0), (0, b, 3), (3, b, 0), (0, b, 3), (3, b, 0)$ for the pair $(0, 0) \in R$ where $\lambda(\pi) = a^6 b^6 \in \mathcal{L}$.

The result of solving the CFPQ problem with the all-path query semantics contains all paths of the desired form between vertices $(i, j) \in R$. For example, for a pair of vertices $(0, 0) \in R$, paths $0\pi_k 0$ should be returned where $\lambda(\pi_k) = a^{6k} b^{6k}$

for all $k \geqslant 1$. The set of all paths can be infinite, so it is necessary either to describe this set using some finite structure, or to have an algorithm for generating k such paths for each pair of vertices and for any $k \geqslant 1$.

1.5 Existing CFPQ Algorithms

In this section, the main existing CFPQ algorithms are considered. All the algorithms require that the given path constraints are expressed in the form of a context-free grammar. Thus, we assume that an input labeled graph $\mathcal{G} = \langle V, E, L \rangle$ and a CFG $G = \langle \Sigma, N, P, S \rangle$ be given.

Algorithms for arbitrary graphs and arbitrary context-free grammars. There is a number of the CFPQ algorithms for the reachability query semantics based on various parsing techniques. For example, Jelle Hellings in his work [9] proposed a CFPQ algorithm based on the CYK parsing algorithm. This CFPQ algorithm has the $O(|N||E| + (|N||V|)^3)$ time complexity. Also, in the work [5], an implementation of this algorithm was obtained for the RDF analysis. In addition, there are works [24] and [25], in which LL and LR parsing algorithms [35] were used to solve this problem. For the algorithm in the work [24], the worst-case time complexity is also known: $O(|V|^3|P|)$. From a practical point of view, in the work [14] an experimental study of the main existing CFPQ algorithms was provided. This study showed that the existing solutions are not performant enough to be used in practice.

In addition, Jelle Hellings proposed a CFPQ algorithm for the single-path query semantics [10]. In this algorithm, the so-called annotated grammar is built, which is a finite structure containing information about all required paths. Further, the formal language theory results are used to find the shortest string generated by this grammar. For each pair of vertices, such strings are used to find the shortest path between them that satisfies the given path constraints. Finding such paths solves the CFPQ problem with the single-path query semantics. Time complexity of the proposed algorithm is

$$O(|N||V|^2(|N||V|^2 log(|N||V|^2) + |P|(|V|^3 + |E|)) + L)$$

where L is the sum of the lengths of the found shortest paths.

Also, there is a number of the CFPQ algorithms [26; 41; 42] for the all-path query semantics based on the parsing algorithms GLL [43] and GLR [44]. In the proposed algorithms, the SPPF [45] used to store information about all paths found. This structure is used to generate any number of paths that satisfy the given constraints. In addition, for the algorithm proposed in [26], the worst-case time complexity of building such an SPPF is $O(|V|^3 \max_{v \in V} deg^+(v))$ where $deg^+(v)$ is the number of outgoing edges from a vertex v. The annotated grammar constructed by Hellings [46] can be also used to generate the required paths. The time complexity of constructing such a grammar is $O(|N||E| + (|N||V|)^3)$, but the complexity of generating the required paths has not yet been studied enough.

Algorithms for partial cases of CFPQ problem. Reps et al. [7; 47; 48] proposed the CFPQ framework with the reachability query semantics for interprocedural program analysis. Later, CFPQ problem has been used to formulate a variety of static analysis problems, such as points-to and alias analyses [2; 49—53], data-dependence analysis [49], type inference analysis [54], type-based flow analysis [1], etc. In these works, graphs built from programs are analyzed using a certain CFL as path constraints. An example of such a language is the Dyck language [7] (the language of the balanced bracket sequences). Therefore, the proposed algorithms are designed to solve a particular case of CFPQ problem.

Another example of an algorithm for solving a particular case of CFPQ problem is the matrix-based algorithm proposed by Philip Bradford in [8]. This algorithm has a worst-case time complexity $O(|V|^{\omega} log^3|V|)$ where ω is the best exponent for estimating the time complexity of multiplying two matrices of size $n \times n$. Note that this algorithm was formulated using linear algebra methods, however, it can be used only for a partial case of the context-free path constraints expressed by the Dyck language with one type of brackets.

In addition, the CFPQ problem was used in [55] for a data provenance via graph segmentation. In this work, a graph is queried using a certain CFL that cannot be described by regular expressions, and can be viewed as a palindrome language [36]. For such a CFL and for CFPQ problem with the reachability query semantics and with fixed set of final vertices V_{dst}, the linear algorithm was proposed assuming that $|V_{dst}|$ is a constant.

1.6 Graph Analysis Using Linear Algebra

This section is devoted to the main ideas of using linear algebra methods for solving graph analysis problems, as well as to the approach to solve path querying problems that is based on these ideas.

1.6.1 Main Ideas

Recently, many graph analysis problems have been solved using linear algebra methods, which, in turn, allows one to use the available parallelization tools, for example, for matrix multiplication, achieving efficient implementation of these algorithms on GPU. In this case, the existing algorithm that solves the desired problem is translated into the language of linear algebra. To accomplish this translation, the following steps can be distinguished.

1. A representation of the main entities of the algorithm (the graph, the set of vertices visited at the current iteration, etc.) using linear algebra objects, i.e. using matrices and vectors.
2. A choice of the graph traversal procedure, and its formulation using a set of operations on linear algebra objects.
3. An implementation of the semantic part of the algorithm by a modification of the selected linear algebra operations. In this case, various algebraic structures (semigroups, monoids, semirings, etc.) are used to define such linear algebra operations.

The choice of objects to represent information about graphs is very important. For example, the graph itself can be described by an adjacency matrix, and the row/column of this matrix describes the outgoing/incoming edges of the vertex specified by the row/column number.

After selecting suitable objects, it is important to define a graph traversal procedure. In essence, this process consists of traversing all the necessary paths. The different graph traversal methods offer a different order of visiting vertices, edges, and paths in the graph. In other words, these traversals have different data access patterns. For the selected objects, there can be linear algebra operations that have a similar or matching data access pattern. Then the process of graph traversing can be formulated using these operations.

However, if one is solving not just the reachability problem then it is also necessary to perform certain actions in the graph traversing process. For example, calculating some information about the visited paths or checking paths for compliance with certain constraints depending on the problem (search for the shortest paths, simple paths, search for paths with given context-free constraints, etc.). These actions can be implemented additionally, however, some operations of linear algebra can be modified in order to perform all necessary actions.

For example, the Bellman-Ford [56; 57] algorithm is presented in Listing 1. This algorithm finds the shortest distances in a weighted graph without multiple edges with the set of vertices V, a set of edges E, a weight function W, and a distinguished starting vertex $s \in V$. If there are no negative cycles in the input graph then this algorithm returns the shortest distances from the starting vertex to all the others.

Listing 1 The Bellman-Ford algorithm

```
function BELLMAN-FORD($V$ is a set of vertices, $E$ is a set of edges, $W$ is a weight function, $s$ is the start vertex)
    $d \leftarrow$ the shortest distances from the vertex $s$ to all the others
    for all $v \in V$ do
        $d(v) \leftarrow \infty$
    $d(s) \leftarrow 0$                                    ▷ The distance to the start vertex
    for $k = 1$ to $N - 1$ do
        if $d(v) > d(u) + W(u, v)$ then
            $d(v) \leftarrow d(u) + W(u, v)$
    for all $(u, v) \in E$ do
        if $d(v) > d(u) + W(u, v)$ then
            return "A negative-weight cycle exists."
    return $d$
```

According to [13], this algorithm can be translated into the language of linear algebra as follows. The graph is represented as an adjacency matrix, and the vector d is used to store the distances from the starting vertex s to all other vertices. Note that due to the absence of multiple edges in the graph, the elements of the adjacency matrix are sets consisting of at most one real number. Therefore, for convenience, we transform the adjacency matrix and use the matrix A with elements — real numbers or special element ∞ denoting the absence of an edge between two vertices. The graph traversal procedure is specified by multiplying a vector by a matrix, as shown in Listing 2. In the presented graph traversal, the entire process of finding paths is reduced to calculating a series of such multiplications $d \cdot A$ in the line 5. However, to solve the problem of finding the shortest paths, additional actions must be specified in the process of traversing the graph. To do this, the operation of multiplying a vector by a matrix is redefined using such an algebraic structure as the semiring $\langle \mathbb{R} \cup \{\infty\}, min, +, \infty \rangle$, which allows one to use the minimum operation as the addition, and use the addition of real numbers as the multiplication operation. The operation *min* is used to calculate the length of a shorter path between two vertices, and the operation $+$ allows one to calculate the lengths of visited paths as the sum of the lengths of their subpaths.

Listing 2 The graph traversal for the algebraic Bellman-Ford algorithm

```
1: function ALGEBRAIC-BELLMAN-FORD($A$ is an adjecency matrix, $s$ is the start vertex)
2:     $d \leftarrow$ vector of the shortest distances from the vertex $s$ to all the others with all elements equal to $\infty$
3:     $d(s) \leftarrow 0$                              ▷ The distance to the start vertex
4:     for $k = 1$ to $N - 1$ do
5:         $d \leftarrow d \cdot A$                      ▷ Vector-matrix multiplication
6:     if $d \neq d \cdot A$ then
7:         return “A negative-weight cycle exists.”
8:     return $d$
```

Thus, suitable linear algebra objects, the graph traversal procedures, and the additional actions in the process of this traversal make it possible to construct algorithms for solving graph analysis problems using the linear algebra operations.

1.6.2 Algebraic Path Problems

The ideas presented in the previous section formed the basis of the approach called *algebraic path problems* [58]. The essence of this approach is to construct a semiring for solving the path querying problem, and this semiring allows one to use algebraic operations to calculate the necessary information about required paths. More formally, for a graph $\mathcal{G} = \langle V, E, L \rangle$, a semiring $\langle \mathcal{S}, \oplus, \otimes, \mathbb{0} \rangle$ is constructed, in which $L \subseteq \mathcal{S}$ and which allows one to reduce the original graph analysis problem to the calculation of values

$$d_{i,j} = \bigoplus \{\lambda_{algebraic}(\pi) \mid \pi \text{ — a path from the vertex } i \text{ to the vertex } j\}$$

for all $i, j \in V$. Specifically, for the path $\pi = (v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_n, v_n)$ we have $\lambda_{algebraic}(\pi) = (\ldots(l_0 \otimes l_1) \otimes l_2) \ldots) \otimes l_n$.

Such values can be calculated using the Gauss-Jordan elimination [59—61], using the Floyd-Warshall algorithm [62—64], and using matrix multiplication [65; 66]. The time complexity of algorithms that use matrix multiplication is $\Theta(|V|^3 log(|V|)(T_{\otimes} + T_{\oplus}))$, and the time complexity of all the other mentioned algorithms is $O(|V|^3(T_{\otimes} + T_{\oplus}) + |V|^2 T_*)$. Here $T_{\otimes}$, $T_{\oplus}$ and T_* is the time complexity of computing the operations $s_1 \otimes s_2$, $s_1 \oplus s_2$ and s^* for all $s, s_1, s_2 \in \mathcal{S}$. In addition, there are some algorithms specialized for solving partial cases of this problem. For example, the algorithm presented in Listing 2 is the result of applying the *algebraic path problems* approach where $d_{i,j}$ values are calculated only for a fixed initial vertex i.

Leslie Valiant [22] showed how the parsing of CFLs can be done using matrix operations modified with algebraic structures similar to semirings but without the associativity of the multiplication operation. Thus, it may be possible to use ideas similar to the ideas of the *algebraic path problems* approach to solve CFPQ problem.

1.7 Existing Linear Algebra Libraries

One of the reasons for using linear algebra methods in graph analysis problems is the existence of a large number of high-performance linear algebra libraries that allow one to obtain good practical results for these problems. For example, there are such linear algebra libraries as cuBLAS, cuSPARSE, cuBool, m4ri, Scipy, and CUSP[1].

[1] Linear algebra library CUSP provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems on GPU: http://cusplibrary.github.io/ (date of access: 14.01.2022).

In order to implement a linear algebra based graph analysis algorithm, it is necessary to create the linear algebra objects (adjacency matrices, vectors) and to use library functions to perform efficient calculation of algebraic operations on the created objects. The data type used in matrices and vectors depends on the graph analysis problem. Often, it is sufficient to use Boolean matrices and vectors to solve a problem (for example, to solve a reachability problem). Also, some matrices with a more complex data type can be expressed using a set of Boolean matrices. For example, a matrix with finite sets as elements can be expressed using its Boolean decomposition. Therefore, the presence in the chosen linear algebra library of an efficient implementation of operations on Boolean matrices and vectors is extremely important for many graph analysis problems. However, if the problem cannot be reduced to the calculation of operations on a set of Boolean matrices and vectors, then it is necessary to use linear algebra objects with a user-defined data type, and the libraries used should be able to calculate operations on them. This situation arises when solving more complex graph analysis problems, for example, finding all paths in a graph with some constraints.

To obtain high-performance implementations it is necessary to take into account the properties of the data and the selected operations used. For example, basic operations on matrices and vectors can be efficiently computed using parallel computing systems (on the CPU, GPU, using distributed computations, etc.). In addition, real data are sparse in practice, so it is important to be able to compute operations on sparse matrices and vectors in the chosen library.

Thus, we can distinguish the following criteria that can be used when choosing a linear algebra library to obtain a high-performance implementation of the graph analysis algorithm:

- implemented operations on Boolean matrices/vectors,
- the ability to use user-defined data types and operations on them,
- parallel computing,
- working with sparse matrices/vectors.

All of these criteria were taken into account when Aydin Buluç, Benjamin Brock, Timothy Mattson, and others created the GraphBLAS [20] standard that defines the basic building blocks for graph analysis algorithms in terms of linear algebra. This standard has been in development since 2013 and was first published in May 2017. To implement the selected graph analysis algorithm, one can use any of the existing implementations of the GraphBLAS standard

(SuiteSparse:GraphBLAS [67—69], IBM GraphBLAS[2], GBTL[3] (GraphBLAS Template Library), GraphBLAST[4] [70]). However, the most stable and optimized are the CPU implementation SuiteSparse:GraphBLAS and the GPU implementation GraphBLAST.

The GraphBLAS standard uses sparse formats for storing matrices (CSR, CSC, COO) [20], as well as algebraic structures that can be used to modify operations on matrices and vectors. When applying the *algebraic path problems* approach, it is necessary to use semirings and the GraphBLAS standard provides such an opportunity. However, it should be noted that the GraphBLAS standard imposes weaker requirements on the used algebraic structures, which allows one to use structures of a more general form than semirings. For example, within the framework of the standard, it is possible to create an algebraic structure and use it to define the operation of multiplying matrices with different data types that cannot be done using semirings. Also in the GraphBLAS standard there are many built-in binary operations from which various algebraic structures can be composed. For example, the standard has the built-in semiring $\langle \mathbb{R} \cup \{\infty\}, min, +, \infty \rangle$ that was used for the Bellman-Ford algorithm in Listing 2. However, if none of the built-in algebraic structures is suitable for the selected graph analysis algorithm then it is also possible to create your own (user-defined) structures, describing the data type, binary operations, and also the corresponding neutral elements.

The characteristics of all the listed linear algebra libraries according to the selected criteria are presented in Table 1.

Thus, to obtain high-performance parallel CPU implementations of the algorithms proposed in this thesis with the ability to use user-defined data types it is advisable to use the SuiteSparse:GraphBLAS implementation or the IBM GraphBLAS implementation of the GraphBLAS standard. However, in this work, we chose SuiteSparse:GraphBLAS since this implementation is the most stable and optimized. The operations on sparse Boolean matrices and vectors can be computed on GPU using the libraries cuBool, GraphBLAST, and CUSP.

[2] IBM GraphBLAS is a C++ implementation of the GraphBLAS standard: https://github.com/IBM/ibmgraphblas (date of access: 14.01.2022).

[3] GBTL is another C++ implementation of the GraphBLAS standard: https://github.com/cmu-sei/gbtl (date of access: 14.01.2022).

[4] GraphBLAST is an implementation of the GraphBLAS standard on the GPU using the CUDA platform: https://github.com/gunrock/graphblast (date of access: 14.01.2022).

Table 1 — The characteristics of existing linear algebra libraries

A library	Boolean data type	User-defined data type	Parallel computing	Sparse matrices/ vectors
cuBLAS	-	-	GPU	-
cuSPARSE	-	-	GPU	+
cuBool	+	-	GPU	+
CUSP	+	+	GPU	+
m4ri	+	-	CPU	-
Scipy	+	-	-	+
SuiteSparse:GraphBLAS	+	+	CPU	+
IBM GraphBLAS	+	+	CPU	+
GBTL	+	-	CPU	+
GraphBLAST	+	-	GPU	+

1.8 Summary

Based on the provided background the following conclusions can be drawn.

- The CFPQ problem is relevant in various areas — in graph databases, bioinformatics, in static program analysis, etc.
- The linear algebra methods can be used for obtaining high-performance implementations of graph analysis algorithms.
- Currently, no research has been conducted on the application of linear algebra methods to solving CFPQ problem.

The provided background also allows us to identify the following approaches, technologies, and tools that are appropriate to use to solve CFPQ problem.

- The ideas of the *algebraic path problems* approach can be used to solve CFPQ problem, but it is necessary to use more general algebraic structures than the semirings.
- For CFPQ problem it is possible to use linear algebra objects not only for representing graphs, but also for the context-free constraints.
- For the efficient analysis of real graphs it is necessary to use parallel computations and sparse matrix formats.

- It is advisable to use the GraphBLAS standard that allows one to use user-defined data types for matrices and vectors, as well as to obtain high-performance implementations of graph analysis algorithms using operations on such matrices and vectors.
- For obtaining parallel CPU implementations of graph analysis algorithms, one can use the SuiteSparse:GraphBLAS implementation of the GraphBLAS standard. And for obtaining such implementations on GPU, the cuBool, GraphBLAST, and CUSP linear algebra libraries can be used.

Chapter 2. Linear Algebra Based Approach to Context-Free Path Querying

This chapter is devoted to the linear algebra based approach to CFPQ. The proposed approach gets a CFPQ problem as input, and returns a CFPQ algorithm and some other artifacts. Despite the fact that it is not possible to create a completely deterministic procedure for creating such algorithms, it seems appropriate to accumulate successful experience in creating a number of CFPQ algorithms in the form of the approach proposed below. This approach can be used as a guide, it also contains successful examples and links to required theoretical results and software libraries.

The chapter begins with a detailed description of the proposed approach. After that, this approach is applied for a partial case of the CFPQ problem, in which the constraints are given by regular languages. At the end, the applicability and limitations of the proposed approach are analyzed.

2.1 Description of the Approach

Summarizing the experience of Leslie Valiant in creating a matrix-based algorithm for parsing strings, as well as the *algebraic path problems* approach, an approach to solve CFPQ problem using linear algebra methods is provided.

A schema of the proposed approach is shown in Figure 2.1. The background of the approach is knowledge from the following areas: the linear algebra, the formal language theory, and the graph theory. The apparatus of the linear algebra provides: algorithms for calculating operations on matrices and vectors that can be used for the graph analysis; methods for solving systems of linear equations (such systems may arise in the matrix form during the process of constructing a graph analysis algorithm); theoretical properties of such algorithms and methods, which will mainly determine the theoretical properties of the resulting algorithm; practical results in the form of existing high-performance linear algebra libraries that will be used in the constructed algorithm implementation. The graph theory provides the following: various graph representations and their connection with linear algebra objects; existing traversal and path querying algorithms, which will serve as the basis for the CFPQ algorithms; successful experience in solving graph analysis problems using linear algebra methods, which is well described in the book [13], and for the path querying problems is collected as an approach *algebraic path problems* [58; 71—73]; practical experience in applying linear algebra methods to graph analysis problems, which is presented in the form of the GraphBLAS [20] standard and its implementations. In turn, the formal language theory offers: methods for describing CFLs (formal grammars, recursive automata), which will be used in the constructed algorithm to describe the context-free path constraints; normal forms of CFGs and algorithms for transforming grammars into these normal forms, which will allow one to use simpler derivation rules to describe path constraints.

As input, the proposed approach receives a CFPQ problem. Such problem has many formulations and may differ in the following dimensions:

- according to the type of the required information about the paths (reachability problem, single-path query semantics, all-path query semantics);
- by the sets of initial and final vertices of the desired paths (searching for paths: between all vertices, with a fixed set of initial vertices, with one initial and one final vertex, etc.);
- by the presence of additional constraints on the desired paths (searching for shortest paths, for simple paths, etc.).

Note that the problem being solved may have a practical or theoretical purpose. For example, with a practical purpose, the goal of applying the proposed approach is to obtain a high-performance implementation of the graph analysis algorithm. And with a theoretical purpose, the main result is to obtain an algorithm that has some unique theoretical properties.

Also, some additional information about the analyzed graphs and the used context-free path constraints can be indicated. For example, the input graph sparsity or special form of the input graph may be known (for example, that only acyclic graphs will be given to the algorithm as input). This, in turn, allows one to use specific structures to store information about graphs, as well as operations on these structures. To store adjacency matrices of sparse graphs, one should choose the appropriate sparse formats CSR, CSC, COO. And in the case of acyclicity of graphs their adjacency matrices will have a triangular form that will reduce the complexity of calculating some operations on them. For example, the first step of

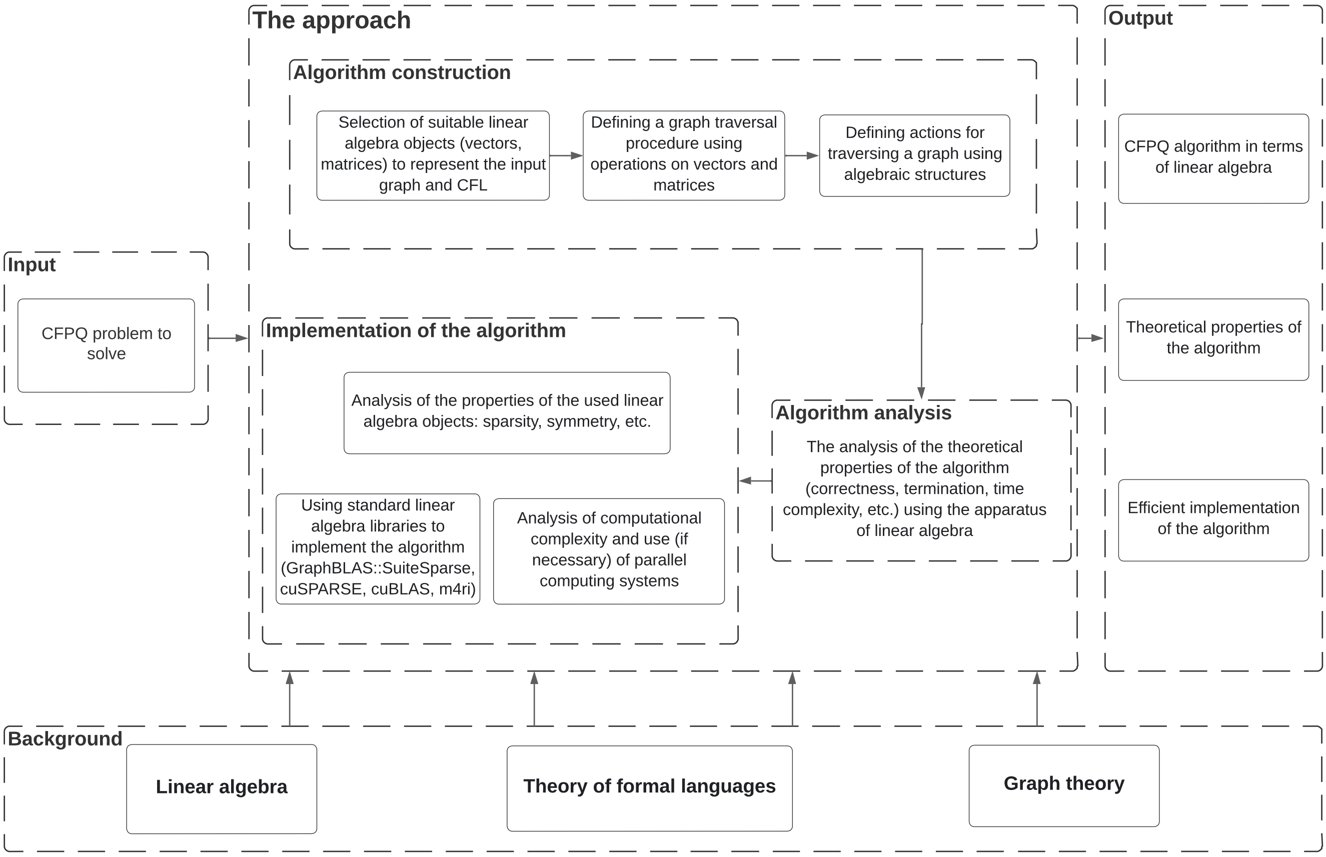

The approach
Algorithm construction
Selection of suitable linear algebra objects (vectors, matrices) to represent the input graph and CFL
Defining a graph traversal procedure using operations on vectors and matrices
Defining actions for traversing a graph using algebraic structures
Input
CFPQ problem to solve
Implementation of the algorithm
Analysis of the properties of the used linear algebra objects: sparsity, symmetry, etc.
Using standard linear algebra libraries to implement the algorithm (GraphBLAS::SuiteSparse, cuSPARSE, cuBLAS, m4ri)
Analysis of computational complexity and use (if necessary) of parallel computing systems
Algorithm analysis
The analysis of the theoretical properties of the algorithm (correctness, termination, time complexity, etc.) using the apparatus of linear algebra
Output
CFPQ algorithm in terms of linear algebra
Theoretical properties of the algorithm
Efficient implementation of the algorithm
Background
Linear algebra
Theory of formal languages
Graph theory

Figure 2.1 — A schema of the linear algebra based approach to CFPQ

the Gauss-Jordan elimination for solving a system of linear equations can be skipped, since it transforms the matrix to a triangular form. And such an operation as the transitive closure calculation for a triangular matrix has a complexity equal to the complexity of calculating a single matrix multiplication. Similarly, the properties of the CFLs used as path constraints can be taken into account in order to find the most appropriate structure for storing these constraints. So, if the normalization of a formal grammar for a given CFL leads to a significant increase of the grammar size then it is advisable to use the representation of the CFL in the form of a recursive automaton. In addition, if the given constraints correspond to some subclass of the class of CFLs (for example, regular languages) then it is advisable to use the corresponding simpler structures (for example, finite automata) to describe such constraints.

The proposed approach is divided into the following parts:

- a construction of a CFPQ algorithm in terms of linear algebra;
- an analysis of the theoretical properties of the constructed algorithm and the given problem;
- an implementation of the constructed algorithm.

An important part of the proposed approach is the algorithm construction. The most important here is the choice of suitable linear algebra objects (vectors, matrices) to represent information about graphs. Also, such objects can be used to represent the context-free path constraints by reducing CFLs to recursive automata [39] with subsequent representation of these automata as graphs. The majority of algebraic graph analysis algorithms (breadth-first search algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm) [13] use adjacency matrices to represent information about graphs. However, some of them also use vectors (in the Bellman-Ford algorithm a vector is used to store the distances from the starting vertex to all the others; in the breadth-first search algorithm, one vector is used to store the set of vertices processed at the current iteration, another vector is used to store information about all graph vertices reachable from the starting one). Thus, adjacency matrices should be used to represent graphs, and vectors should be used to store information about graph vertices (reachability from the starting vertex; distance from the starting vertex; set of vertices incident to a given vertex).

Next, we need to specify a graph traversal procedure. As mentioned above, if suitable linear algebra objects was selected then this procedure can be formulated using operations on these objects (vector and matrix operations). For example, in the algorithm for computing the transitive closure [71] a repeated squaring of a matrix is used to traverse a graph. And in the Bellman-Ford [13] algorithm, paths of a certain length are traversed at each iteration by multiplying a vector by an adjacency matrix. Note that algorithms for finding paths between all graph vertices use matrix-matrix multiplication, and in the case of a fixed starting vertex, the graph can be traversed using vector-matrix multiplication.

Further, it is necessary to set additional actions in the process of graph traversing, which allow one to analyze the visited paths and check them for compliance with the given context-free path constraints. We propose to consider algebraic structures over the linear algebra objects discussed above.

These actions can be performed by modifying selected operations on matrices and vectors using various algebraic structures (semirings, monoids, etc.). Such semiring modifications are widely used for solving path querying problems and form the main idea of the *algebraic path problems* [58] approach. For example, in the Bellman-Ford algorithm, the semiring $\langle \mathbb{R} \cup \{\infty\}, min, +, \infty \rangle$ is used to define the vector-matrix multiplication. Also, many semirings are known for this approach, which are used to solve various path querying problems that arise in the network analysis [71]. Note that, unlike the *algebraic path problems* approach, the proposed approach is not limited to using semirings for modifying the linear algebra operations. For example, Leslie Valiant, in his study of string parsing for the context-free grammars [22], showed how an algebraic structure similar to a semiring can be used. This structure uses non-associative multiplication operation $\otimes$.

The constructed algorithms may have unique theoretical properties, which may be the main reason for applying the proposed approach to a given problem. Therefore, another important part of the approach is the analysis of these properties. For example, Leslie Valiant in his work [22] created the first subcubic parsing algorithm for the context-free grammars. This became possible due to the formulation of the parsing algorithm using linear algebra methods and the efficient calculation of the used algebraic operations. Thus, the formulation of an algorithm in terms of linear algebra also makes it possible to obtain a theoretical result for the problem itself, by answering some open theoretical question in this area. One of such open problems is obtaining a subcubic algorithm for the CFPQ problem with the reachability query semantics and with complexity $O(n^{3-\varepsilon})$ where n is the number of vertices of the analyzed graph and $\varepsilon > 0$.

From a practical point of view, an important part of the proposed approach is the implementation of the constructed algorithm. Such algorithms are easy to implement since the most difficult part is to effectively implement the calculation of the necessary linear algebra operations that are already implemented in existing libraries. The existing linear algebra libraries were discussed in section 1.7, and their characteristics are presented in Table 1.

Since graphs are sparse in practice, it is important to store these objects using sparse formats. For example, when choosing adjacency matrices of input graphs as linear algebra objects, the CSR and CSC formats are used for sparse graphs, and the COO format is used for hypersparse graphs. Also, some linear algebra operations can be efficiently computed in parallel. Therefore, the constructed algorithms can be effectively implemented in parallel on the CPU, GPU, using distributed computing, etc. Depending on the selected linear algebra operations, the object storage format, and the selected computing system, the choice of the appropriate linear algebra library is made. Also, the selected operations can be implemented independently.

Some graph analysis problems can be solved using operations on a set of Boolean matrices and vectors. In such cases, for efficient parallel computation of operations on sparse Boolean matrices on CPU it is recommended to use the

SuiteSparse:GraphBLAS library (implementation of the GraphBLAS standard). In addition, this library allows one to use user-defined data types to solve more complex problems, for example, the CFPQ problem with the all-path query semantics. Moreover, cuSPARSE, cuBool, GraphBLAST, and CUSP libraries with implemented operations on sparse matrices and vectors can be used to obtain GPU implementations. Note that the cuSPARSE library does not provide special operations on Boolean matrices and vectors. Therefore, to calculate operations on Boolean matrices and vectors using this library, the floating-point numbers should be used as a data type, which will significantly affect the memory cost and running time of the implementation.

As a result, the proposed approach makes it possible to obtain one or more of the following results:

- a CFPQ algorithm in terms of linear algebra;
- theoretical properties of the constructed algorithm and the solved CFPQ problem;
- an efficient implementation of the constructed algorithm.

2.2 An Example of Constructing an Algorithm

To demonstrate the application of the proposed approach, consider the construction of an algorithm for a partial case of the CFPQ problem with the reachability query semantics, in which the constraints are given by a strict subclass of CFLs — regular languages. In this problem, the input is a labeled graph and a regular language that describes the path constraints.

First, it is necessary to select linear algebra objects to represent information about the input graph and the regular language. One of the ways to define a regular language is to construct an appropriate finite automaton [36]. In turn, the input labeled graph can also be represented as a finite automaton, in which all states are both initial and final. Then the stated reachability problem with given constraints can be solved by finding the intersection of these two finite automata, as well as by solving the reachability problem for the graph representation of this intersection via the transitive closure calculation. According to the theorem 1.3.1, the intersection of finite automata can be calculated using the Kronecker product applied to matrices from Boolean decompositions of adjacency matrices for graph representations of these automata. Thus, the input graph and regular language will be represented as a set of Boolean matrices, and the graph traversal procedure will be a series of Kronecker products of Boolean matrices followed by a transitive closure calculation.

Suppose that the finite automaton F_1 describes the input graph, and F_2 — the input regular language. The Listing 3 provides an algorithm that uses linear algebra methods to solve the reachability problem with given path constraints in the form of a regular language. In the line 7, the algorithm traverses the graph using the Kronecker product, as a result of which the transition matrix of the finite automaton F will be built, which is the intersection of the finite automata F_1 and F_2. Here Boolean matrices M_i^a store information about transitions with symbol a in automaton F_i. Then, in the line 8, the matrix M^* is calculated, which contains information about all paths in the input graph corresponding to the given path constraints. For this, the *transitiveClosure* function and the element-wise addition operation $\bigvee$ defined over the Boolean semiring $\langle\{0,1\},\vee,\wedge,0\rangle$ are used. Note that the *transitiveClosure* function can also be implemented using linear algebra methods, namely through a repeated squaring of a Boolean matrix. Finally, lines 9-12 calculate the set *result*, which is the answer to the reachability problem in a graph with regular language path constraints. Here the value 1 in the cell $M^*[(i,s_1),(j,s_2)]$ indicates the existence of the path π_1 in the input graph from the vertex i to the vertex j, and the existence of the path π_2 in the graph representation of the finite automaton F_2 from vertex s_1 to vertex s_2 where $\lambda(\pi_1) = \lambda(\pi_2)$. This means that for any pair of vertices (i,j) in the input graph, the vertex j is reachable from the vertex i by at least one path that satisfies the input constraints if and only if $M^*[(i,s_1),(j,s_2)] = 1$ for the vertex s_1 corresponding to the initial state of the finite automaton F_2 and the vertex s_2 corresponding to one of its final states. Thus, the constructed algorithm solves the given graph analysis problem.

To solve the reachability problem, the Kronecker product of Boolean matrices was used, based on standard logical operations from the Boolean semiring $\langle\{0,1\},\vee,\wedge,0\rangle$. In this case, it is not required to specify additional actions when traversing the graph by changing this algebraic structure. However, such actions may be needed when solving other path querying problems. For example, when solving path querying problems with the all-path query semantics.

Listing 3 A reachability regular path querying algorithm

```
function AlgebraicRegularPathQuerying($F_1, F_2$)
    $\mathcal{M}_1 \leftarrow$ the Boolean decomposition of the adjacency matrix for $F_1$
    $\mathcal{M}_2 \leftarrow$ the Boolean decomposition of the adjacency matrix for $F_2$
    $q_s \leftarrow$ the initial state of the finite automaton $F_2$ describing the regular language
    $Q_f \leftarrow$ the set of final states of the automaton $F_2$
    $\Sigma \leftarrow$ the set of common symbols on transitions in automata $F_1$ и $F_2$
    $\mathcal{M} \leftarrow \{M_1^a \times M_2^a \mid a \in \Sigma\}$          ▷ The Kronecker product
    $M^* \leftarrow transitiveClosure(\bigvee_{M \in \mathcal{M}} M)$
    $result \leftarrow \varnothing$
    for $i, j, s \mid M^*[(i, q_s), (j, s)] = 1$ do
        if $s \in Q_f$ then
            $result \leftarrow result \cup \{(i, j)\}$
    return $result$
```

Thus, it was shown how a CFPQ algorithm can be constructed using linear algebra methods when path constraints are formulated in the form of a regular language.

2.3 On Applicability and Limitations of the Approach

Not every graph analysis algorithm, in particular, a path querying algorithm, can be formulated in terms of linear algebra. For example, many attempts have been made to find such a formulation for the depth-first search algorithm — this algorithm is one of the fundamental graph analysis algorithms. However, recently such a formulation was found only for particular cases of graphs (for binary trees, acyclic graphs) [21]. In turn, the breadth-first search algorithm can be easily formulated in terms of linear algebra, and such a formulation was found a long time ago [13].

Thus, when applying the proposed approach at the stage of constructing the algorithm, the following issues may arise:

- the challenge of formulating a graph traversal procedure using operations on selected linear algebra objects;

- the challenge of finding algebraic structures for additional actions that are necessary for the stated graph analysis problem during the graph traversal.

In case of failure in solving these issues, it is necessary to return to the choice of linear algebra objects. Such a return is also necessary when issues are identified at later stages of the approach. For example, the unsatisfactory time complexity of the constructed algorithm can be identified at the stage of analyzing the theoretical properties. At the same time, at the implementation stage, it may be difficult to use standard linear algebra libraries:

- the properties of the linear algebra objects used to represent them do not allow one to use existing linear algebra libraries;
- unsatisfactory running time or high memory consumption of the implementation were detected.

Note that not all stages of the approach are mandatory. For example, in the case of the practical purpose of the problem being solved, the stage of analyzing the theoretical properties of the algorithm can be skipped, and when choosing the linear algebra objects and operations, it is necessary to be guided by the properties of existing linear algebra libraries. On the other hand, when the problem being solved has theoretical purpose, there may be no stage of implementing the constructed algorithm, and at the stage of constructing the algorithm, it is necessary to select suitable linear algebra objects and operations to obtain the best theoretical properties of the algorithm.

2.4 Summary

Thus, in this chapter, we proposed a linear algebra based approach to CFPQ. The novelty of this approach is as follows.

1. The existing CFPQ approaches either do not use linear algebra methods or are intended only for a partial case of the context-free path constraints and/or for specialized graphs.
2. The proposed approach allows one to use a wide class of optimizations of linear algebra operations for efficient analysis of large graphs.

3. The proposed approach makes it possible to build algorithms that are easy to implement, portable, and allow one to use parallel computations and existing linear algebra libraries.

Chapter 3. A Matrix-Based CFPQ Algorithm

In this chapter, we present a matrix-based algorithm constructed by applying the approach from the previous chapter to solve CFPQ problem with the reachability, the single-path, and the all-path query semantics. Also, the correctness and time complexity of the obtained algorithm are formulated and proved. In addition, the details of the implementation of the algorithm are given, as well as a step-by-step demonstration of the proposed algorithm.

3.1 Algorithm Construction

This section describes the process of constructing a CFPQ algorithm using matrix multiplication operations.

For an input labeled graph $\mathcal{G} = \langle V, E, L \rangle$ and an input CFL, first, it is necessary to select linear algebra objects to represent information about the graph. The applicability of such objects of linear algebra as matrices in graph analysis problems is well known. Therefore, to represent the input graph, we will use its adjacency matrix, the cells (i, j) of which will contain information about the edges between the vertex i and the vertex j. The input CFL will be described by the CFG $G = \langle \Sigma, N, P, S \rangle$ in the WCNF, which, as we will see, will be convenient for further construction of algebraic structures with operations that take into account the given context-free constraints while traversing the graph. Then the adjacency matrix must be initialized to include information about the given constraints. To describe the connection of the found paths in a graph with given constraints, nonterminal symbols of the grammar G will be used. Thus, the proposed algorithm finds paths π that form a string $\lambda(\pi)$ that can be derived from some nonterminal A of the CFG G. Therefore, to initialize the adjacency matrix of the input graph, we will visit all edges with labels $a \in \Sigma \cap L$ and all corresponding derivation rules of the form $A \to a$. Thus, all paths of length 1 will be visited, but it is also necessary to store information about empty paths using rules of the form $A \to \varepsilon$ in the grammar G. Therefore, for each such rule and for each vertex i of the graph $\mathcal{G}$, the matrix will contain information about existence of an empty path π from the vertex i to the vertex i such that the string $\lambda(\pi)$ can be derived from the nonterminal A.

Next, we describe a graph traversal procedure that computes the transitive closure of an initialized matrix. Such a transitive closure can be computed using the known technique [71], which performs a series of multiplications of the adjacency matrix by itself. This allows one to traverse the graph and to visit all the paths necessary for graph analysis. It remains only to redefine the matrix multiplication in order to check paths for compliance with the input context-free path constraints. Also, all the necessary to solve the given CFPQ problem information must be calculated. The rules of the form $A \to BC$ where $A, B, C \in N$ will be used to check the paths for compliance with the input constraints. On the right side of such rules there is only a concatenation of two nonterminals that corresponds to the concatenation of two substrings. Similarly, in the process of graph analysis, rules of this type correspond to the concatenation of two paths $i\pi_1 k$ and $k\pi_2 j$ where $B \Rightarrow_G \lambda(\pi_1)$ and $C \Rightarrow_G \lambda(\pi_2)$. In this case, for the path $i\pi j$ resulting from such a concatenation, we can say that $A \Rightarrow_G \lambda(\pi)$. Thus, the process of traversing the graph $\mathcal{G}$ will be closely related to the process of strings derivation formed by the visited paths. And, finally, in order to calculate all the necessary information for solving a CFPQ problem we construct the algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ where:

- *MatrixElements* is a set containing all possible values of the elements of the discussed matrices;
- $\oplus$ is the operation of adding matrix elements, which will be used when aggregating information about several paths in a graph between the same vertices;
- $\otimes$ is the operation of multiplying matrix elements, which will be used when aggregating information about two paths that can be concatenated into one longer path;
- $\bot$ is an neutral element for the addition operation that will denote the absence of the desired paths for a particular pair of vertices.

Then, using the constructed algebraic structure, we define the matrix multiplication operation $a \cdot b = c$ where a and b are matrices of suitable sizes with elements from the set *MatrixElements* as

$$c_{i,j} = \bigoplus_{k=1}^{n} a[i,k] \otimes b[k,j].$$

Also, for matrices of the same size, we use the element-wise addition operation $\bigoplus$ defined over constructed algebraic structure. At the initialization stage, it is necessary to determine the elements used to describe information about paths of length 0 and 1. Since the values of these elements depend on the particular CFPQ problem, we denote them as $\alpha^0_{i,j}, \alpha^1_{i,j} \in \mathit{MatrixElements}$ for each pair of vertices (i, j).

Thus, in Listing 4 we present a matrix-based CFPQ algorithm. The presented algorithm takes as input a graph and the context-free path constraints already expressed in the form of a CFG in the WCNF. Note that the information about the paths in the graph is stored in the set T of matrices, which consists of $|N|$ matrices T^A, one for each nonterminal symbol $A \in N$. In this case, the cell $T^A[i, j]$ contains information about the found paths in the graph from the vertex i to the vertex j forming strings derived from the nonterminal A in the grammar G. This idea is similar to the idea of using the Boolean decomposition of the adjacency matrix, however, the resulting matrices T^A will be Boolean only for the reachability problem. As a result, the proposed algorithm solves the stated CFPQ problem since all the necessary information about the paths from the vertex i to the vertex j that satisfy the given constraints will be written in cell $T^S[i, j]$.

Listing 4 A matrix-based CFPQ algorithm

```
function MatrixBasedCFPQ($\mathcal{G} = \langle V, E, L\rangle$, $G = \langle N, \Sigma, P, S\rangle$)
    $n \leftarrow |V|$
    $T \leftarrow \{T^A \mid A \in N$ where $T^A$ is a matrix of size $n \times n$ with all elements equal to $\perp$ $\}$
    for all $(i, x, j) \in E$, $A \mid A \to x \in P$ do
        $T^A[i, j] \leftarrow \alpha^1_{i,j}$                ▷ Matrix initialization using rules $A \to x$
    for $A \mid A \to \varepsilon \in P$ do
        for all $i \in \{0, \dots, n-1\}$ do
            $T^A[i, i] \leftarrow \alpha^0_{i,i}$            ▷ Matrix initialization using rules $A \to \varepsilon$
    while any matrix in $T$ is changing do
        for all $A \to BC \in P$ where $T^B$ or $T^C$ are changed do
            $T^A \leftarrow T^A \bigoplus (T^B \cdot T^C)$
    return $T$
```

Next, we present various algebraic structures that allow one to modify the matrix operations in the algorithm shown in Listing 4 to solve the CFPQ problem with the reachability, the single-path, or with the all-path query semantics.

Reachability. Since the solution of the reachability problem requires only information about existence of paths that form certain words from the labels of their edges, the cells of the adjacency matrix can only contain Boolean values 0 or 1. In this case, the cell $T^A[i,j]$ has the value 1 if there is a path from the node i to the node j that forms a string derived from the nonterminal A in the grammar G, and value 0 otherwise. During the matrix initialization, $\alpha^0_{i,j}$ and $\alpha^1_{i,j}$ values will be written to $T^A[i,j]$ when a corresponding path of length 0 or 1 is found. And since for the given problem the existence of a path is written in the corresponding cell using the value 1 then $\alpha^0_{i,j} = \alpha^1_{i,j} = 1$. Thus, the algorithm shown in Listing 4 solves the reachability problem in a graph with given context-free constraints using the algebraic structure $\langle \{0,1\}, \vee, \wedge, 0 \rangle$ with logical operations of disjunction and conjunction. In the graph $\mathcal{G}$ there is a path from i to j that satisfies the given constraints only if $T^S[i,j] = 1$.

Single-path. To solve the CFPQ problem with the single-path query semantics it is necessary for each pair of vertices (i,j) to construct at least one path from the vertex i to the vertex j that satisfies the given constraints, if such paths exist. To do this, we store additional information about the found paths in the matrix elements and construct a new algebraic structure for modifying operations on these matrices. We will use matrices whose elements contain information about the found paths in the form of tuples (*left*, *right*, *middle*, *height*) where *left* and *right* are initial and final vertices of the found path, *middle* is the one of the intermediate vertices of the found path π with the string $\lambda(\pi)$ derived from the nonterminal A in the grammar G, and *height* is the minimum height of the derivation tree of the string $\lambda(\pi)$ from the nonterminal A. In the case when no such paths are found for a certain pair of vertices (i,j) then we will use the special tuple $\bot = (0,0,0,0)$.

For a specific pair of vertices (i,j) and a nonterminal A, in the proposed algorithm we visit a path that forms a string derived from the nonterminal A and has the minimum height of the derivation tree. In addition, concatenations of two such paths for different triples (A,i,j) will be visited. Therefore, the proposed algorithm will use only tuples (*left*, *right*, *middle*, *height*) with *height* $\leqslant h+1$ where

h is the maximum of such minimum heights of derivation trees for different triples (A, i, j). We denote the set of all possible such tuples including $\bot$ as *PathIndex*, and construct an algebraic structure using this set as a domain. The neutral element in such a structure will be the tuple $\bot$. In this case, initially all matrices will be initialized with this neutral element. Next, we define the multiplication and addition operations for this structure.

In the process of graph traversal using the matrix multiplication operation and derivation rules of the form $A \to BC$, we visit the paths $i\pi j$ that are the concatenation of two shorter paths $i\pi_1 k$ and $k\pi_2 j$. Thus, it is convenient to choose the vertex k as an intermediate vertex *middle* of the path $i\pi j$. Then the multiplication operation $\otimes$ for $PI_1, PI_2 \in PathIndex$ can be defined as follows.

$$PI_1 \otimes PI_2 = \begin{cases} (PI_1.left, PI_2.right, PI_1.right, max(PI_1.height, PI_2.height) + 1), \\ \qquad \text{if } PI_1 \neq \bot \neq PI_2 \\ \bot, \qquad \text{otherwise} \end{cases}$$

In addition, in the process of graph analysis for the same pair of vertices several paths can be found that satisfy the given context-free constraints. Such paths are aggregated using the addition operation $\oplus$ for $PI_1, PI_2 \in PathIndex$ that can be defined as follows.

$$PI_1 \oplus PI_2 = \begin{cases} PI_1, & \text{if } PI_1 \neq \bot \neq PI_2 \text{ and} \\ & (PI_1.height, PI_1.middle) \leqslant (PI_2.height, PI_2.middle) \\ PI_1, & \text{if } PI_2 = \bot \\ PI_2, & \text{otherwise} \end{cases}$$

In the presented definition, we use the lexicographical order over pairs of integers $(PI.height, PI.middle)$ for $PI \in PathIndex$. Thus, when the operation $\oplus$ is used, the elements of the matrix T^A will contain information about the found paths that form words with the minimum heights of the derivation trees from the nonterminal A in the grammar G. Next, we determine the values of the elements $\alpha^0_{i,j}, \alpha^1_{i,j} \in PathIndex$. For a path of length 1 that form a string derived from the nonterminal A_k, the information about this path will be added to the corresponding matrix element T^{A_k} in the form of a tuple $\alpha^1_{i,j} = (i, j, i, 1)$. In turn, for paths of length 0 , the tuples $\alpha^0_{i,i} = (i, i, i, 1)$ are used. In these cases, the initial vertices of paths of length 0 and 1 are chosen as intermediate vertices. Thus, the algorithm presented in Listing 4 allows one to solve the CFPQ problem with the single-path query semantics using the semiring $\langle \mathit{PathIndex}, \oplus, \otimes, \bot\rangle$ with operations defined for this problem. The result of the algorithm is a set of matrices T^A for all nonterminal symbols $A \in N$, whose elements (i, j) contain information about found paths. For all pairs of vertices (i, j), these elements describe a path that forms a word derived from the nonterminal A with the minimum height of the derivation tree. If there are no such paths for a pair of vertices (i, j) then $T^A[i, j] = \bot$.

However, in addition to constructing the set of matrices T containing information about the found paths, it is necessary to construct the path $i\pi j$ for any i, j and nonterminal A such that $A \Rightarrow_G \lambda(\pi)$ if such a path exists. Therefore, in Listing 5 we present an algorithm for the extraction of a single path corresponding to given context-free constraints for any pair of vertices (i, j) and nonterminal A. The presented algorithm extracts the path corresponding to the string with the minimum height of the derivation tree. The algorithm returns the empty path π_ε only when $i = j$ and $A \to \varepsilon \in P$. It should be noted that if $T^A[i, j] = \bot$ then the proposed algorithm returns a special path $\pi_\varnothing$ denoting the absence of paths corresponding to the given constraints.

Thus, the algorithms presented in Listings 4 and 5 allow one to solve the CFPQ problem with the single-path query semantics.

All-path. In order to solve the CFPQ problem with the all-path query semantics it is necessary for each pair of vertices (i, j) to construct any given number of paths from the vertex i to the vertex j that satisfy the given constraints. To do this, we add additional information about all found paths in the graph to the the matrix elements and construct a new algebraic structure for modifying operations on these matrices. We use the set of triples (*left*, *right*, *middles*) where *left* and *right* are the initial and the final vertex of the found paths, and *middles* is the set of some intermediate vertices of these paths. In the process of traversing the graph, $T^A[i, j].\mathit{middles}$ will contain all intermediate vertices k of visited paths $i\pi j$ formed by the concatenation of two shorter paths $i\pi_1 k$ and $k\pi_2 j$. If the path has length 0 or 1 and has no intermediate vertices, then we will use the elements $\alpha^0_{i,j} = \alpha^1_{i,j} = (i, j, \{n\})$ with the special value $n = |V|$. This value is different from the number of any graph vertex, and it describes the case when the path found has no intermediate vertices. If there

Listing 5 Single path extraction algorithm

```
function ExtractSinglePath($i, j, A, T = \{T^{A_i}\}, G = \langle N, \Sigma, P, S\rangle$)
    $index \leftarrow T^A_{i,j}$
    if $index = \bot$ then
        return $\pi_\varnothing$                                  ▷ Such path does not exist
    if $index.height = 1$ then
        if $(i = j) \wedge (A \to \varepsilon \in P)$ then
            return $\pi_\varepsilon$                              ▷ Return the empty path
        for all $x \mid (i,x,j) \in E$ do
            if $A \to x \in P$ then
                return $[(i,x,j)]$                                ▷ Return a path of length 1
    for all $A \to BC \in P$ do
        $index_B \leftarrow T^B[i, index.middle]$
        $index_C \leftarrow T^C[index.middle, j]$
        if $(index_B \neq \bot) \wedge (index_C \neq \bot)$ then
            $maxH \leftarrow max(index_B.height, index_C.height)$
            if $index.height = maxH + 1$ then
                $\pi_1 \leftarrow$ ExtractSinglePath($i, index.middle, B, T, G$)
                $\pi_2 \leftarrow$ ExtractSinglePath($index.middle, j, C, T, G$)
                return $\pi_1 + \pi_2$                            ▷ Return the concatenation of two paths
```

are no paths corresponding to the given constraints for some pair of vertices, we will use the triple $\bot = (0, 0, \varnothing)$. We denote the set of all possible values of such elements, including $\bot$, as *AllPathIndex*. Next, we build an algebraic structure over the elements of the set *AllPathIndex*, which allows one to modify the matrix multiplication and matrix addition operations to solve the CFPQ problem with the all-path query semantics. The neutral element in this structure will be the $\bot$ element.

To construct an algebraic structure, we first define the multiplication operation $\otimes$ for $API_1, API_2 \in AllPathIndex$ as follows.

$$API_1 \otimes API_2 = \begin{cases} (API_1.left, API_2.right, \{API_1.right\}), \text{if } API_1 \neq \bot \neq API_2 \\ \bot, \qquad \text{otherwise} \end{cases}$$

For a pair of vertices several paths that satisfy the given constraints can be found. Thus, it is necessary to store information about all the paths found. Therefore, we define the addition operation $\oplus$ for $API_1, API_2 \in AllPathIndex$ as follows.

$$API_1 \oplus API_2 = \begin{cases} (API_1.left, API_1.right, & \\ API_1.middles \cup API_2.middles), & \text{if } API_1 \neq \bot \\ API_2, \quad \text{otherwise} & \end{cases}$$

Thus, the algorithm presented in Listing 4 allows one to solve the CFPQ problem with the all-path query semantics using the algebraic structure $\langle AllPathIndex, \oplus, \otimes, \bot \rangle$ with operations defined for this problem. The result of the algorithm is a set of matrices T^A for all nonterminal symbols $A \in N$, whose cells (i,j) contain information about all paths from i to j that form strings derived from the nonterminal A. If there are no such paths for a pair of vertices (i,j) then $T^A[i,j] = \bot$.

After constructing the set of matrices T that contains information about all paths found, it is necessary to extract any given number of paths $i\pi j$ for any i,j and nonterminal A such that $A \Rightarrow_G \lambda(\pi)$. Therefore, in Listing 6 we present a path extraction algorithm. For any pair of vertices (i,j) and nonterminal A, this algorithm constructs paths corresponding to the given context-free path constraints. The algorithm returns the empty path π_ε only when $i = j$ and $A \to \varepsilon \in P$.

Thus, the algorithms presented in Listings 4 and 6 allow one to solve the CFPQ problem with the all-path query semantics.

3.2 Correctness of the Algorithm

In this section, we formulate and prove statements about the correctness and termination of the proposed CFPQ algorithm.

Suppose that the particular CFPQ problem is given. Also, suppose that the algebraic structure $\langle MatrixElements, \oplus, \otimes, \bot \rangle$ is constructed for this problem, and the matrix operations are modified in the algorithm shown in Listing 4. Moreover, the defined elements $\alpha^0_{i,j}$ and $\alpha^1_{i,j}$ are used to initialize the matrices of this algorithm. Then we first prove that if the constructed structure has some properties then

Listing 6 All paths extraction algorithm

```
function ExtractAllPaths($i, j, A, T = \{T^{A_k} \mid A_k \in N\}, G = \langle N, \Sigma, P, S \rangle$)
    $index \leftarrow T^A_{i,j}$
    if $index = \bot$ then
        return $\varnothing$                                    ▷ Such paths do not exist
    $n \leftarrow$ the number of rows in the matrix $T^A$
    $resultPaths \leftarrow \varnothing$
    for all $m \in index.middles$ do
        if $m = n$ then                                          ▷ Add single-edge or empty paths
            for all $x \mid A \to x \in P$ do
                if $(i, x, j) \in E$ then
                    $resultPaths \leftarrow resultPaths \cup \{((i, x, j))\}$
            if $(i = j) \wedge (A \to \varepsilon \in P)$ then
                $resultPaths \leftarrow resultPaths \cup \{\pi_\varepsilon\}$
        else ▷ Add to the result the concatented paths from $i$ to $m$ and from $m$ to $j$
            for all $A \to BC \in P$ do
                $index_B \leftarrow T^B[i, m]$
                $index_C \leftarrow T^C[m, j]$
                if $(index_B \neq \bot) \wedge (index_C \neq \bot)$ then
                    $lPaths \leftarrow$ ExtractAllPaths($i, m, B, T, G$)
                    $rPaths \leftarrow$ ExtractAllPaths($m, j, C, T, G$)
                    $resultPaths \leftarrow resultPaths \cup lPaths \cdot rPaths$
    return $resultPaths$
```

the resulting CFPQ algorithm terminates in a finite number of steps and correctly solves the problem. And then we will show that the algebraic structures proposed in the previous section for the reachability, the single-path, and the all-path query semantics have such properties.

Termination. To prove the termination of the algorithm shown in Listing 4, we first introduce the notation for the intermediate matrices from the set T that are obtained in this algorithm. For any pair of vertices (i, j) and for any nonterminal $A \in N$, we will use the notation $T^{A,0}[i, j]$ for the value in cell $T^A[i, j]$ after matrix initialization in lines 2–8 of the algorithm, and $T^{A,k}[i, j]$ — for the value in cell $T^A[i, j]$ after k iterations of the loop in lines 9–11, for $k \geqslant 1$.

Then we prove that if some partial order relation $\preceq$ can be defined on the elements of the used finite algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ then the proposed algorithm terminates.

Theorem 3.2.1 (Algorithm termination)**.** Assume that a partial order relation $\preceq$ can be defined on the domain of the used algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ such that for any pair of vertices (i, j), any nonterminal $A \in N$, and any $k \geqslant 1$, $T^{A,k-1}[i, j] \preceq T^{A,k}[i, j]$. Also, assume that the domain *MatrixElements* is finite. Then the algorithm presented in Listing 4 terminates in a finite number of steps.

Proof. At the next iteration of the loop in lines 9–11 of the algorithm, the values in the cells $T^A[i, j]$ can either remain unchanged or change as a result of the operations $T^A \leftarrow T^A \bigoplus (T^B \cdot T^C)$ in the line 11. The algorithm continues its work until the value in at least one cell changes. By the hypothesis of the theorem, the values in the matrix cells increase monotonically with respect to the given partial order $\preceq$. Thus, due to the finiteness of the domain *MatrixElements*, at some iteration, the matrices will stop changing and the algorithm will terminate. □

Correctness. To prove the correctness of the algorithm shown in Listing 4, we introduce the notation that describes the relationship between the visited paths in the graph and the values in the cells $T^{A,k}[i, j]$ of the intermediate matrices obtained in this algorithm. For any pair of vertices (i, j) and for any nonterminal $A \in N$, we will use the notation $\mathcal{P}_{\mathcal{G},k}(i, j, A)$ for the set of all paths $i\pi j$ of the graph $\mathcal{G}$ such that there exists a derivation tree of minimum height $h \leqslant k$ for the string $\lambda(\pi)$ from the nonterminal A of the grammar G. We will also say that some information allows us to correctly solve a CFPQ problem for the set of paths $\mathcal{P}$ from the vertex i to the vertex j, if this information:

- allows one to answer the question of the existence of at least one path $\pi \in \mathcal{P}$ satisfying the given constraints for the reachability problem;
- allows one to construct at least one path $\pi \in \mathcal{P}$ satisfying the given constraints, if such exists, for the single-path query semantics;
- allows one to construct any finite number of paths $\pi \in \mathcal{P}$ satisfying given constraints for the all-path query semantics.

Then we prove that if the operations of the used algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ and its elements have some properties then the proposed algorithm correctly solves the given CFPQ problem.

Lemma 3.2.2 (Algorithm correctness)**.** Suppose that $\mathcal{G} = \langle V, E, L \rangle$ is the input graph and $G = \langle N, \Sigma, P, S \rangle$ is the input CFG for the algorithm presented in Listing 4. Also, suppose that the used algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ and any elements $\alpha_1, \alpha_2 \in \mathit{MatrixElements}$ have the following properties:

- the algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ is a semiring without the associativity of the operation $\otimes$;
- the selected elements $\alpha^l_{i,j} \in \mathit{MatrixElements}$ used in lines 4–8 of the algorithm, allow one to correctly solve the CFPQ problem for the set of paths of the graph $\mathcal{G}$ from the vertex i to the vertex j of length l where $l \in \{0, 1\}$;
- if the elements α_1 and α_2 allow one to correctly solve the stated CFPQ problem for the sets of paths $\mathcal{P}_1$ and $\mathcal{P}_2$ from the vertex i to the vertex j then the element $\alpha_1 \oplus \alpha_2$ allows one to correctly solve the stated problem for the set of paths $\mathcal{P}_1 \cup \mathcal{P}_2$;
- if the elements α_1 and α_2 allow one to correctly solve the stated CFPQ problem for the set of paths $\mathcal{P}_1$ from the vertex i to the vertex k and for the set $\mathcal{P}_2$ of paths from the vertex k to the vertex j then the element $\alpha_1 \otimes \alpha_2$ allows one to correctly solve this problem for the set of paths $\mathcal{P}_1 \cdot \mathcal{P}_2$ of all possible concatenations of paths from these sets.

Then for any pair of vertices (i, j) of the graph $\mathcal{G}$, for any nonterminal $A \in N$, and for any $k \geqslant 0$ the value in cell $T^{A,k}[i, j]$ allows one to correctly solve the stated CFPQ problem for the set of paths $\mathcal{P}_{\mathcal{G},k+1}(i, j, A)$.

Proof. (Proof by induction)

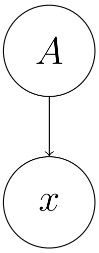


Figure 3.1 — A derivation tree of the minimal height $h = 1$ for the string $x = \lambda(\pi)$ where $x \in \Sigma \cup \{\varepsilon\}$

Base case: We will show that the statement of the lemma is correct for $k = 0$. For grammars in the WCNF, derivation trees with height 1 only correspond to the strings of length 1 or the empty string ε. Thus, the set $\mathcal{P}_{\mathcal{G},1}(i, j, A)$ contains only paths of length 0 or 1. For all such paths and only for them, there exists a derivation tree of minimum height $h = k + 1 = 1$ shown in the Figure 3.1. Traversing of such paths occurs in lines 4–8 of the algorithm. Therefore, for any pair of vertices (i, j) and any nonterminal $A \in N$, $T^{A,0}[i, j] \neq \bot$ if and only if either there exists a path $i\pi j$ of length 1 that contains a single edge $(i, x, j) \in E$ and $(A \to x) \in P$, or $i = j$ and $(A \to \varepsilon) \in P$. For a path $\pi \in \mathcal{P}_{\mathcal{G},1}(i, j, A)$ to the cell $T^{A,0}[i, j]$ will be written the corresponding elements $\alpha^0_{i,j}$ or $\alpha^1_{i,j}$. And by the lemma conditions, the distinguished elements $\alpha^l_{i,j} \in \mathit{MatrixElements}$ allow one to correctly solve the stated CFPQ problem for sets of paths of length l where $l \in \{0, 1\}$. Thus, the statement of the lemma is correct for $k = 0$.

Inductive step: Assume that the statement of the lemma is correct for any $k \leqslant (p - 1)$ where $p \geqslant 1$, and show that it is also correct for $k = p$.

The loop operations in lines 9-11 of the algorithm mean that the value $T^{A,p}[i, j] = T^{A,p-1}[i, j] \bigoplus_{A \to BC \in P}(T^{B,p-1} \cdot T^{C,p-1})[i, j]$. Next, we prove that the value in the cell $T^{A,p}[i, j]$ allows one to correctly solve the stated CFPQ problem for the set of paths $\mathcal{P}_{\mathcal{G},p+1}(i, j, A)$.

Suppose that the value $\bigoplus_{A \to BC \in P}(T^{B,p-1} \cdot T^{C,p-1})[i, j]$ allows one to correctly solve the problem for the set of paths $\mathcal{P}'$. By the induction hypothesis, the value in the cell $T^{A,p-1}[i, j]$ allows one to do this for the set of paths $\mathcal{P}_{\mathcal{G},p}(i, j, A)$. Then, by the condition of the lemma, we obtain that the value $T^{A,p}[i, j] = T^{A,p-1}[i, j] \bigoplus_{A \to BC \in P}(T^{B,p-1} \cdot T^{C,p-1})[i, j]$ allows one to correctly solve the problem for the set of paths $\mathcal{P}_{\mathcal{G},p}(i, j, A) \cup \mathcal{P}'$. Thus, to complete the proof of the lemma, it remains to show that $\mathcal{P}_{\mathcal{G},p}(i, j, A) \cup \mathcal{P}' = \mathcal{P}_{\mathcal{G},p+1}(i, j, A)$.

The existence of paths $\pi \in \mathcal{P}_{\mathcal{G},p+1}(i, j, A)$ is equivalent to the existence of the rule $A \to BC \in P$ and paths $\pi_1 \in \mathcal{P}_{\mathcal{G},p}(i, r, B)$, $\pi_2 \in \mathcal{P}_{\mathcal{G},p}(r, j, C)$ where the path π is obtained as a result of the concatenation of the paths π_1 and π_2, and the derivation tree of the string $\lambda(\pi)$ from the nonterminal A with the minimum height $h \leqslant p+1$ is shown in the Figure 3.2. By the inductive hypothesis, the values $T^{B,p-1}[i, r]$ and $T^{C,p-1}[r, j]$ allow one to correctly solve the stated problem for the set of all such paths π_1 and for the set of all such paths π_2, respectively. Using the properties of the operations $\oplus$ and $\otimes$ from the conditions of the lemma, as well as the construction of matrix operations, we obtain that the value of $(T^{B,p-1} \cdot T^{C,p-1})[i, j]$ allows one to correctly solve the stated problem for the set of paths $\mathcal{P}_{BC}$ obtained by concatenation of such paths π_1 and π_2. In addition, using the property of the operation $\oplus$ from the conditions of the lemma, we obtain that $\bigoplus_{A\to BC\in P}(T^{B,p-1} \cdot T^{C,p-1})[i, j]$ allows one to correctly solve the stated problem for the set of paths $\mathcal{P}' = \bigcup_{A\to BC\in P} \mathcal{P}_{BC}$. Thus, $\mathcal{P}_{\mathcal{G},p}(i, j, A) \cup \mathcal{P}' = \mathcal{P}_{\mathcal{G},p}(i, j, A) \cup \bigcup_{A\to BC\in P} \mathcal{P}_{BC} = \mathcal{P}_{\mathcal{G},p+1}(i, j, A)$, and it proves the statement of the lemma.

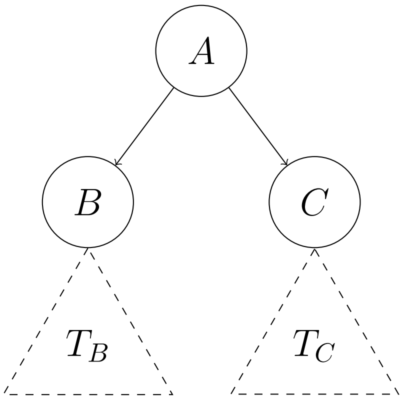


Figure 3.2 — A derivation tree of the minimal height $h = 1 + max(h_1, h_2)$ for the string $\lambda(\pi)$ where T_B и T_C are derivation trees for strings $\lambda(p_1)$ and $\lambda(\pi_2)$ with heights h_1 and h_2 respectively

□

A consequence of the lemma 3.2.2 is the following theorem on the correctness of the algorithm shown in Listing 4.

Theorem 3.2.3 (Algorithm correctness)**.** Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph and $G = \langle N, \Sigma, P, S\rangle$ is the input CFG for the algorithm presented in Listing 4. Also, suppose that the chosen algebraic structure satisfy the properties presented in the condition of the lemma 3.2.2. Then for any pair of vertices (i, j) of the graph $\mathcal{G}$, for any nonterminal $A \in N$, the value in the cell $T^A[i, j]$ allows one to correctly solve the stated CFPQ problem for the set of all paths of the graph $\mathcal{G}$ from the vertex i to the vertex j.

Proof. The algorithm returns the set of matrices T only if for some $k \geqslant 1$ $T^{A,k} = T^{A,k-1}$ for all nonterminals $A \in N$. That is, for any nonterminal A $T^{A,p} = T^A$ for any $p \geqslant k$. Thus, by the lemma 3.2.2, for any vetex pair (i, j) of the graph $\mathcal{G}$, for any nonterminal $A \in N$, the cell $T^A[i, j]$ allows one to correctly solve the stated CFPQ problem for the set of all paths $i\pi j$ such that there exists a derivation tree for the string $\lambda(\pi)$ and the nonterminal A of the grammar G. This completes the proof of the theorem. □

Using the theorem 3.2.3 and induction on the heights of the derivation trees corresponding to the paths being extracted using the proposed algorithm for CFPQ problem with the single-path and the all-path query semantics, the following theorem can be proved.

Theorem 3.2.4 (Correctness of the path extraction algorithms)**.** Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph, $G = \langle N, \Sigma, P, S\rangle$ is the input CFG, and T is the set of matrices returned by the algorithm shown in Listing 4. Also, suppose that the chosen algebraic structure satisfy the properties presented in the conditions of the lemma 3.2.2. Then for any pair of vertices (i, j) and any nonterminal $A \in N$:

- for the single-path query semantics the algorithm shown in Listing 5 will construct a path $i\pi j$ such that there is a derivation tree for the string $\lambda(\pi)$ and the nonterminal A of the grammar G if such a path exists;
- for the all-path query semantics, the algorithm shown in Listing 6 will construct the set of all paths $i\pi j$ such that there is a derivation tree for the string $\lambda(\pi)$ and the nonterminal A of the grammar G.

Properties of the proposed algebraic structures. It remains to show that the algebraic structures proposed in section 3.1 for CFPQ problem have properties that ensure the termination and the correctness of the proposed algorithm.

First, we will show the termination of the algorithm for the proposed structures. The algebraic structures for the reachability and the all-path query semantics contain finite domains. We will show that the domain of the proposed structure for the single-path query semantics is also finite. It remains to show that for given graph and a CFG, $h \leqslant |N||V|^2$. Each internal node of such derivation trees is associated with some nonterminal A and a subtree with this node as a root. The leaves of such a subtree form a string corresponding to some path in the graph from the vertex i to the vertex j. Therefore, with each internal node of such trees we will associate the triple (A, i, j). If for some triple (A, i, j) there is a derivation tree of minimum height $h > |N||V|^2$ then in such a tree there is a path of length $h > |N||V|^2$ from the root to some leaf. Also, there are two different nodes u and v on this path that are associated with the same triple (B, k, l) since there are only $|N||V|^2$ different triples. Suppose that the node u be closer to the root of the tree than the node v. Then replacing in this tree the subtree corresponding to the node u with the subtree corresponding to the node v, we get a new derivation tree for the triple (A, i, j) with decreased distances from the root of the tree to the leaves. Such transformations can be repeated until the height of the tree is less than or equal to $|N||V|^2$. Thus, the original derivation tree did not have a minimum height for the triple (A, i, j). This means that all such heights $h \leqslant |N||V|^2$, and the tuples that will be used in the proposed algorithm contain *height* $\leqslant |N||V|^2 + 1$. Therefore, the proposed structure for the single-path query semantics also has a finite domain.

Since all three proposed algebraic structures contain a finite domain, it remains to show that for the elements of these structures can be defined a partial order relation $\preceq$ such that for any pair of vertices (i, j) and for any $k \geqslant 1$, $T^{A,k-1}[i, j] \preceq T^{A,k}[i, j]$.

Denote $\preceq_{\mathit{rel}}$ as the partial order relation corresponding to the reachability problem, $\preceq_{\mathit{single}}$ — the single-path query semantics, and $\preceq_{\ \mathit{all}}$ — the all-path query semantics. For brevity, we will use the notation z_l, z_r, z_h, z_m, and z_{ms} for *z.left*, *z.right*, *textitz.height*, *z.middle*, and *z.middles* respectively, where z is an element of the set *MatrixElements* of one of the three provided algebraic structures. Then for any elements x and y of these domains we define partial order relations as follows:

$$\begin{aligned}
x \preceq_{\mathit{rel}} y \quad &\iff \quad (x = 0) \wedge (y = 1), \\
x \preceq_{\mathit{single}} y \quad &\iff \quad (x = \bot = (0, 0, 0, 0)) \vee \\
&\qquad ((x_l = y_l) \wedge (x_r = y_r) \wedge ((y_h, y_m) \leqslant (x_h, x_m))), \\
x \preceq_{\mathit{all}} y \quad &\iff \quad (x = \bot = (0, 0, \varnothing)) \vee ((x_l = y_l) \wedge (x_r = y_r) \wedge (x_{ms} \subseteq y_{ms})).
\end{aligned}$$

It is necessary to show that for any $k \geqslant 1$, $T^{A,k-1}[i, j] \preceq_{\mathit{rel}} T^{A,k}[i, j]$ for the CFPQ problem with the reachability query semantics, $T^{A,k-1}[i, j] \preceq_{\mathit{single}} T^{A,k}[i, j]$ for the single-path query semantics, and $T^{A,k-1}[i, j] \preceq_{\mathit{all}} T^{A,k}[i, j]$ for the all-path query semantics. At the next iteration of the loop in lines 9–11 of the algorithm, the value in the cell $T^A[i,j]$ may either remain unchanged or change as a result of the operations $T^A \leftarrow T^A \bigoplus (T^B \cdot T^C)$ in the line 11. If the value has not changed then the statement of the theorem is obvious due to the reflexivity of the discussed partial order relations. Otherwise $T^{A,k}[i,j] = T^{A,k-1}[i,j] \oplus (T^{B,k-1} \cdot T^{C,k-1})[i,j]$. Next, for all three path query semantics, we prove that $T^{A,k-1}[i,j] \preceq T^{A,k-1}[i,j] \oplus (T^{B,k-1} \cdot T^{C,k-1})[i,j]$ for the corresponding partial order relation $\preceq$.

Reachability. Using the Boolean algebraic structure with the disjunction operation as the $\oplus$ operation, the value in the cell $T^A[i,j]$ can only change from 0 to 1. Thus, $T^{A,k-1}[i,j] \preceq_{rel} T^{A,k}[i,j]$. This fact proves the termination of the algorithm for the reachability problem.

Single-path. By the definition of the operation $\oplus$ of a semiring over the set of tuples *PathIndex*, the value in the cell $T^A[i,j]$ can only change if $T^{A,k-1}[i,j] = \bot$ or

$$((T^{A,k}[i,j].height, T^{A,k}[i,j].middle) \leqslant (T^{A,k-1}[i,j].height, T^{A,k-1}[i,j].middle)).$$

If $T^{A,k-1}[i,j] = \bot$ then by the definition of the partial order relation $\preceq_{single}$, $T^{A,k-1}[i,j] \preceq_{single} T^{A,k}[i,j]$. Otherwise, by the definitions of the matrix multiplication $(\cdot)$ and the operation of multiplication of matrix elements $\otimes$, we obtain that $T^{A,k-1}[i,j].left = (T^{B,k-1} \cdot T^{C,k-1})[i,j].left$ and $T^{A,k-1}[i,j].right = (T^{B,k-1} \cdot T^{C,k-1})[i,j].right$. Thus, by the definition of the partial order relation $\preceq_{single}$, $T^{A,k-1}[i,j] \preceq_{single} T^{A,k}[i,j]$. This fact proves the termination of the algorithm for the single-path query semantics.

All-path. By the definition of the operation $\oplus$ of a semiring over the set of triples *AllPathIndex*, the value in the cell $T^A[i,j]$ can only change if $T^{A,k-1}[i,j] = \bot$ or $(T^{A,k-1}[i,j].middles \subseteq T^{A,k}[i,j].middles)$. If $T^{A,k-1}[i,j] = \bot$ then the statement of the theorem is proved by the definition of the partial order relation $\preceq_{all}$. Otherwise, by the definitions of the matrix multiplication $(\cdot)$ and the operation of multiplication of matrix elements $\otimes$, we obtain that $T^{A,k-1}[i,j].left = (T^{B,k-1} \cdot T^{C,k-1})[i,j].left$ and $T^{A,k-1}[i,j].right = (T^{B,k-1} \cdot T^{C,k-1})[i,j].right$. Thus, by the definition of the partial order relation $\preceq_{all}$, $T^{A,k-1}[i,j] \preceq_{all} T^{A,k}[i,j]$. This fact proves the termination of the algorithm for the all-path query semantics.

Second, we prove the correctness of the algorithm for the proposed algebraic structures by showing that they satisfy the following properties specified in the conditions of the lemma 3.2.2.

All three proposed algebraic structures are semirings without the associativity of the multiplication operation. It remains to show the validity of the properties for the elements $\alpha^l_{i,j}$ and for the operations of addition and multiplication.

Reachability. In the proposed algebraic structure for the CFPQ problem with the reachability query semantics, the elements $\alpha^0_{i,j} = \alpha^1_{i,j} = 1$ were selected, which are used in the algorithm shown in Listing 4 when there exists a path from $\mathcal{P}_{\mathcal{G},1}(i, j, A)$ for some nonterminal $A \in N$. If such a path does not exist then the corresponding cell will contain the value $\perp = 0$. Thus, the elements $\alpha^0_{i,j}$ and $\alpha^1_{i,j}$ allow one to determine the existence of paths of length 0 or 1 that satisfy the given constraints, which proves the corresponding property of the algebraic structure $\langle\{0, 1\}, \vee, \wedge, 0\rangle$. The validity of the property for the operation $\oplus$ is obvious, since $\alpha_1 \vee \alpha_2 = 1$ only if there is at least one required path in the set $\mathcal{P}_1$ or in the set $\mathcal{P}_2$, which is equivalent to solving the problem for the set $\mathcal{P}_1 \cup \mathcal{P}_2$. And, finally, if the elements α_1 and α_2 allow one to correctly solve the reachability problem for the set of paths $\mathcal{P}_1$ from the vertex i to the vertex k and for the set $\mathcal{P}_2$ paths from the vertex k to the vertex j then the element $\alpha_1 \wedge \alpha_2 = 1$ only if there was at least one desired path in the set $\mathcal{P}_1$ and at least one such a path in the set $\mathcal{P}_2$. The concatenation of such paths is the desired path from i to j, so the value $\alpha_1 \wedge \alpha_2$ will solve the problem for the set of paths $\mathcal{P}_1 \cdot \mathcal{P}_2$.

Single-path. Consider the proposed structure $\langle \mathit{PathIndex}, \oplus, \otimes, \perp\rangle$ for the single-path query semantics. To correctly solve the CFPQ problem, we will use the element $\perp = (0, 0, 0, 0) \in \mathit{PathIndex}$ to indicate the absence of the desired paths and the $(\mathit{left}, \mathit{right}, \mathit{middle}, \mathit{height}) \in \mathit{PathIndex}$ if there is a desired path π from *left* to *right*, with some distinguished intermediate vertex *middle*, and the minimum height *height* of the derivation tree of the string $\lambda(\pi)$ for the corresponding grammar and nonterminal. The remaining properties follow from the construction of this algebraic structure in section 3.1:

- the distinguished elements $\alpha^0_{i,i} = (i, i, i, 1)$ and $\alpha^1_{i,j} = (i, j, i, 1)$ describe the desired paths of length 0 or 1 with the derivation tree of height 1 if such paths exist;
- if the elements α_1 and α_2 describe the desired paths with the minimum height of the derivation tree for the sets of paths $\mathcal{P}_1$ and $\mathcal{P}_2$ from the vertex i to the vertex j then the element $\alpha_1 \oplus \alpha_2$ describes the desired path with the minimum height of the derivation tree for the set of paths $\mathcal{P}_1 \cup \mathcal{P}_2$;

- if the elements α_1 and α_2 describe the required paths with the minimum height of the derivation tree for the set of paths $\mathcal{P}_1$ from the vertex i to the vertex k and for the set $\mathcal{P}_2$ of paths from the vertex k to the vertex j then the element $\alpha_1 \otimes \alpha_2$ describes the desired path with the minimum height of the derivation tree for the set of paths $\mathcal{P}_1 \cdot \mathcal{P}_2$.

All-path. For the all-path query semantics in the proposed structure $\langle \mathit{AllPathIndex}, \oplus, \otimes, \bot \rangle$, the element $\bot = (0, 0, \varnothing) \in\ textitAllPathIndex$ is used to indicate that there are no desired paths, and the element $(\mathit{left}, \mathit{right}, \mathit{middles}) \in \mathit{AllPathIndex}$ is used if there are desired paths π from the vertex *left* to the vertex *right* with some distinguished *middle* $\in$ *middles*. The remaining properties follow from the construction of this algebraic structure in section 3.1:

- the selected elements $\alpha^0_{i,j} = \alpha^1_{i,j} = (i, j, \{n\})$ describe all required paths of length 0 or 1 using the special value $n = |V|$ in the set *middles*;
- if the elements α_1 and α_2 describe all the required paths for the sets of paths $\mathcal{P}_1$ and $\mathcal{P}_2$ from the vertex i to the vertex j then the element $\alpha_1 \oplus \alpha_2$ describes all required paths for the set of paths $\mathcal{P}_1 \cup \mathcal{P}_2$;
- if the elements α_1 and α_2 describe all required paths for the set of paths $\mathcal{P}_1$ from the vertex i to the vertex k and for the set $\mathcal{P}_2$ of paths from the vertex k to the vertex j then the element $\alpha_1 \otimes \alpha_2$ describes all the required paths for the set of paths $\mathcal{P}_1 \cdot \mathcal{P}_2$.

Thus, for all three algebraic structures proposed in section 3.1 for the CFPQ problem with the reachability, the single-path, and the all-path query semantics, the theorems 3.2.1 and 3.2.3 are correct.

3.3 Time Complexity of the Algorithm

In this section, we present the worst-case time complexity of the proposed CFPQ algorithm.

Suppose that the algebraic structure $\langle \mathit{MatrixElements}, \oplus, \otimes, \bot \rangle$ with finite domain *MatrixElements* is given, as well as a partial order relation $\preceq$ defined on this structure such that for any pair of vertices (i, j) and for any $k \geqslant 1$, $T^{A,k-1}[i, j] \preceq T^{A,k}[i, j]$. Then, according to the theorem 3.2.1, the algorithm will terminate in a finite number of steps. Then suppose that N_{iter} is the maximum number of loop iterations in lines 9–11 of the algorithm shown in Listing 4. Note that the value of N_{iter} depends on the chosen algebraic structure.

Also, we assume that all operations on elements from the set *MatrixElements* are computed in $O(1)$. In addition, for the algebraic structure used, we estimate the worst-case number of elementary operations needed to compute operations on matrices of size $n \times n$ as:

- $O(n^2)$ — for $\bigoplus$ element-wise addition of two matrices,
- $O(n^3)$ — for $(\cdot)$ multiplication of two matrices.

Note that these estimates can be improved for algebraic structures with some properties. For example, there exists an algorithm with subcubic complexity $O(n^{3-\varepsilon})$ where $\varepsilon > 0$ for such algebraic structures as rings [74].

Taking into account the assumptions made, the following estimation of the time complexity of the pathfinding algorithm in a graph with given KC constraints is valid. The following theorem on the time complexity of the proposed CFPQ algorithm is correct with respect to the discussed assumptions.

Theorem 3.3.1 (The worst-case time complexity of the matrix-based CFPQ algorithm)**.** For the input graph $\mathcal{G} = \langle V, E, L\rangle$, and the input CFG $G = \langle N, \Sigma, P, S\rangle$, the algorithm presented in Listing 4 has the following worst-case time complexity: $O(|N||P||V|^5)$.

Proof. At the initialization stage, the algorithm writes values to the cells of these matrices. In the line 5 the algorithm performs the write operation $O(|E|)$ times, and in the line 8 — $O(|V||N|)$ times. Therefore, $O(|E| + |V||N|)$ elementary operations are performed at the initialization stage for all path query semantics.

Next, consider the loop in lines 9–11 that computes the transitive closure. The algorithm continues calculations while any matrix from the set T changes. There are $|N|$ matrices in this set, and each has $|V|^2$ elements. Due to the properties of the three proposed algebraic structures, the values of the cells of the matrices increase monotonically with respect to the corresponding partial order relation. Next, consider the maximum number of iterations N_{iter} that can occur when solving CFPQ problem with various path query semantics.

Reachability and single-path. At each iteration of the loop in lines 9–11, the heights of the derivation trees of the strings corresponding to visited paths in the graph are increased by one. For the reachability and the single-path query semantics, the value of a particular cell can change only during one iteration of the loop in lines 9–11. Therefore, the maximum number of iterations of the discussed loop can be achieved in the case when the value of only one cell in one matrix changes for each iteration. For the reachability problem, the value can change from $\bot = 0$ to 1, and for the single-path query semantics the value either changes from $\bot = (0,0,0,0)$ to a tuple describing some path found at this iteration, or the value for the found path is changed to the value for another path with a smaller number of intermediate vertex *middle*. During one iteration of the loop in lines 9–11, the operations in line 11 are executed $O(|P|)$ times. Therefore, for the reachability and the single-path query semantics, the algorithm has the following worst-case time complexity: $O(N_{iter}|P||V|^3)$ where the maximum number of loop iterations in lines 9–11 is $N_{iter} = |N||V|^2$.

All-path. For this problem, we prove that the number of loop iterations in lines 9–11 exceeds the maximum number of such iterations for the single-path query semantics by no more than 1, that is, it is always less than or equal to $|N||V|^2 + 1$. To do this, we will show that changing the value in one of the matrices at iteration with number k for the all-path query semantics leads to the fact that a value in one of the matrices will change at iterations with number k or $k-1$ when solving the CFPQ problem with the single-path query semantics and the same input.

Similarly to the query semantics already discussed, for the all-path query semantics at each iteration of the loop in lines 9–11, the heights of the derivation trees of strings corresponding to the visited paths are increased by one. Suppose that the value in the cell $T^A[i,j]$ is changed at the iteration with number k after finding paths from i to j that form strings derived from the nonterminal A and that have derivation trees with height $k+1$. This could happen in two cases: either $T^{A,k-1}[i,j] = \bot$ and $k{+}1$ is the minimum height of derivation trees for such paths, or $T^{A,k-1}[i,j] = (i, j, \mathit{middles}) \neq \bot$ and there is at least one path π with distinguished intermediate vertex $m \notin \mathit{middles}$. In the first case, at the same iteration, the value in the cell $T^A[i,j]$ will be changed when solving the problem for the single-path query semantics, so $k \leqslant |N||V|^2$. Of greater interest is the second case, which we consider in more detail. Since $m \notin \mathit{middles}$ then $k+1$ is the minimum height of derivation trees for paths from the node i to the node j with distinguished intermediate node m forming strings derived from the nonterminal A. Suppose that such a derivation tree of minimum height be formed using the derivation rule $(A \to B\ C) \in P$ at the root level where the path $i\pi j$ is the concatenation of the paths $i\pi_1 m$ and $m\pi_2 j$, while $B \Rightarrow_G \lambda(\pi_1)$ and $C \Rightarrow_G \lambda(\pi_2)$. Then at least one of the strings $\lambda(\pi_1)$ or $\lambda(\pi_2)$ has the minimum height of the derivation tree for the corresponding nonterminal equal to k. Otherwise, it would be possible to build a derivation tree for the string $\lambda(\pi)$ with a height less than $k+1$. Thus, at the previous iteration of the loop numbered $k-1$, at least one path with the minimum height of the derivation tree for the corresponding nonterminal was found and the value in cell $T^B[i,m]$ or cell $T^C[m,j]$ was changed. The values in these cells will also change on $k-1$ iterations of the loop when solving the problem for the single-path query semantics. Thus, to solve the problem with the single-path query semantics, at least $k-1$ iterations are required, and from the already proven upper bound on the number of such iterations, we obtain that $k-1 \leqslant |N||V|^2$ and, therefore, $k \leqslant |N||V|^2+1$.

We have shown that when solving the CFPQ problem with the all-path query semantics, the number of iterations in the loop does not exceed $|N||V|^2+1$, and therefore the worst-case time complexity obtained for all path query semantics is $O(|N||P||V|^5)$. □

Note that the given time complexity of the algorithm presented in Listing 4 can be significantly improved for some algebraic structures and for sparse graphs. However, in the general case the obtained upper bounds of the algorithm time complexity are exact since there is an example of a graph and a CFG using which these upper bounds are reached. This example was first published by Jelle Hellings in the work [46]. Such example uses a graph similar to $\mathcal{G}_1$ shown in Figure 1.1, but with 2^n+1 edges labeled by a and 2^n edges labeled by b. The CFG used in this example describes the language $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$. The fact that the lengths of two cycles in this graph are coprime leads to the fact that the minimum height of the derivation trees of paths from the vertex 0 to the vertex 0 that satisfy the given constraints is $O(|V|^2)$. A step-by-step demonstration of the proposed algorithm on a particular case of such example will be provided in the next section.

Further, using induction on the minimum heights of the derivation trees corresponding to the extracted path using the algorithm presented in Listing 5, the following theorem can be proved.

Theorem 3.3.2 (The time complexity of the single path extraction algorithm). Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph, $G = \langle N, \Sigma, P, S\rangle$ is the input CFG, and T is the set of matrices returned by the algorithm shown in Listing 4 using an algebraic structure corresponding to the single-path query semantics. Then the algorithm shown in Listing 5 has the following worst-case time complexity: $O(d|N|^2)$

where d is the number of nodes in the minimum height derivation tree for the path extracted by the algorithm.

In this work, the time complexity of the algorithm presented in Listing 6 is not given since it depends on the specific method of limiting the number of paths to be extracted and methods for constructing the corresponding sets of paths.

3.4 An Example

In this section, a step-by-step demonstration of the proposed algorithm is provided using the example based on the CFL of the balanced nested bracket sequences.

The graph $\mathcal{G}_1$ is showed in Figure 1.1 and the CFL $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$ is the context-free path constraints. We represent this CFL as a CFG G in the WCNF with the following derivation rules.

$$\begin{array}{rrcl@{\qquad}rrcl}
0: & S & \to & A\ B & 3: & A & \to & a \\
1: & S & \to & A\ S_1 & 4: & B & \to & b \\
2: & S_1 & \to & S\ B & & & &
\end{array}$$

Next, we provide a demonstration of the algorithm presented in Listing 4 for CFPQ problem with various path query semantics.

Reachability. For the discussed graph and CFG, the algorithm shown in Listing 4 initializes the set of matrices T shown in Figure 3.3.

Since there are no derivation rules of the form $A \to \varepsilon$ in the given grammar, the algorithm proceeds to execute the loop in lines 9–11. Further, only the matrices T^S and T^{S_1} will change. After the first iteration of the loop, all elements of the matrix $T^{S_1,1}$ will remain equal to 0, and the elements of the matrix $T^{S,1}$ are shown in Figure 3.4. When finding the next path at some iteration, the algorithm adds reachability information to the corresponding matrix. For example, at the first iteration of the loop, the path $2\pi 3 = (2, a, 0), (0, b, 3)$ was found, forming the string $\lambda(\pi) = ab$ derived from the nonterminal S. Information about the reachability of the vertex 3 from the vertex 2 is added to the cell $T^{S,1}[2, 3]$.

$$T^{A,0} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$T^{B,0} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.3 — The set of matrices T after initialization for the reachability problem (elements $T^{S_1,0}[i,j]$ and $T^{S,0}[i,j]$ are equal to $\perp = 0$ for all i, j)

$$T^{S,1} = T^{S,0} \bigoplus (T^{A,0} \cdot T^{B,0}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.4 — Elements of the matrix $T^{S,1}$ after first loop iteration for the reachability problem

To completely traverse the graph in accordance with the input CFG, this process is repeated while new information is added to at least one of the matrices. In this example, the algorithm will perform 13 iterations of the loop, after which it will complete its work. The resulting matrix $T^{S,13}$ is represented in Figure 3.5.

$$T^{S,13} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.5 — Elements of the resulting matrix $T^{S,13}$ for the reachability problem

Finally, the resulting matrix can be used to construct a set of pairs of vertices, which is the answer to the CFPQ problem with the reachability query semantics. In this example, this set is equal to $\{(0,0), (0,3), (1,0), (1,3), (2,0), (2,3)\}$.

Single-path. Similarly to the reachability problem, for the single-path query semantics the algorithm initializes the set of matrices T shown in Figure 3.6.

$$T^{A,0} = \begin{pmatrix} \bot & (0,1,0,1) & \bot & \bot \\ \bot & \bot & (1,2,1,1) & \bot \\ (2,0,2,1) & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

$$T^{B,0} = \begin{pmatrix} \bot & \bot & \bot & (0,3,0,1) \\ \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \\ (3,0,3,1) & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.6 — The set of matrices T after initialization for the single-path query semantics (elements $T^{S_1,0}[i,j]$ and $T^{S,0}[i,j]$ are equal to $\bot = (0,0,0,0)$ for all i,j)

Only matrices T^S and T^{S_1} will change in the process of graph analysis. After the first iteration of the loop in lines 9–11, all elements of the matrix $T^{S_1,1}$ will remain equal to $\bot$, and the elements of the matrix $T^{S,1}$ are shown in Figure 3.7. Unlike the previous case, when solving the CFPQ problem with the single-path query semantics, the added information not only about reachability, but also about the initial and final vertices of the path, one of its intermediate vertices, and also about the height of the corresponding derivation tree. For example, at the first iteration of the loop, the path $2\pi 3 = (2,a,0),(0,b,3)$ was found, forming the string $\lambda(\pi) = ab$ derived from the nonterminal S. The value $(2,3,0,2)$ is written to the cell $T^{S,1}[2,3]$ since the path from the node 2 to the node 3 was found, with an intermediate node 0 and with a derivation tree for the string $\lambda(\pi) = ab$ of height 2, which is shown in Figure 3.8.

$$T^{S,1} = T^{S,0} \bigoplus (T^{A,0} \cdot T^{B,0}) = \begin{pmatrix} \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & (2,3,0,2) \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.7 — Elements of the matrix $T^{S,1}$ after first loop iteration for the single-path query semantics

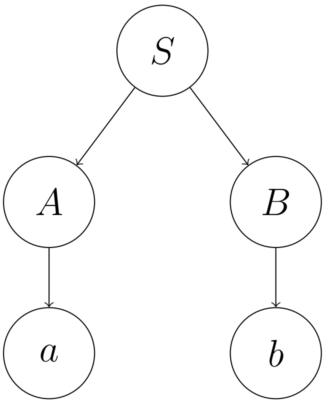


Figure 3.8 — A derivation tree with minimal height $h = 2$ for the nonterminal S and the string $\lambda(\pi) = ab$

Similarly to reachability problem, the algorithm will execute 13 iterations of the loop in lines 9–11, after which it will complete its work. The resulting matrix $T^{S,13}$ for the single-path query semantics is presented in Figure 3.9.

$$T^{S,13} = \begin{pmatrix} (0,0,1,12) & \bot & \bot & (0,3,1,6) \\ (1,0,2,4) & \bot & \bot & (1,3,2,10) \\ (2,0,0,8) & \bot & \bot & (2,3,0,2) \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.9 — Elements of the resulting matrix $T^{S,13}$ for the single-path query semantics

The constructed matrices also contain reachability information, so the set of pairs of vertices $\{(0,0),(0,3),(1,0),(1,3),(2,0),(2,3)\}$ can be constructed, which is the answer to the CFPQ problem with the reachability query semantics. However, the constructed matrices also contain information sufficient to construct one path corresponding to the given constraints for each pair of vertices from this set. For this, the algorithm presented in Listing 5 can be used. For example, to extract such a path from the vertex 0 to the vertex 0 it is necessary to pass to this algorithm as input: the initial and final vertices $i = j = 0$; the starting nonterminal S of the grammar; the resulting matrices for all nonterminals, calculated by the algorithm presented in Listing 4; as well as the context-free constraints in the form of the grammar G. Then the algorithm shown in Listing 5 will return the path $0\pi0$ for which $\lambda(\pi) = a^6b^6$. The value $T^{S,13}[0,0] = (0,0,1,12)$ corresponds to the derivation tree with minimum height for the string a^6b^6 and the starting nonterminal S.

All-path. Similarly to the previous two path query semantics, we provide the matrices from the set T for the all-path query semantics and the structure $\langle \mathit{AllPathIndex}, \oplus, \otimes, \bot \rangle$ with the neutral element $\bot = (0, 0, \varnothing)$. The initialized set of matrices T is shown in Figure 3.10 (in the process of matrix initialization, instead of intermediate vertices, a special value $4 = |V|$ is used indicating that the length of the path found is 0 or 1). The elements of the matrix $T^{S,1}$ obtained after the first iteration of the loop in lines 9–11 are shown in Figure 3.11. And finally, the resulting matrix $T^{S,13}$ for the all-path query semantics is presented in Figure 3.12.

$$T^{A,0} = \begin{pmatrix} \bot & (0,1,\{4\}) & \bot & \bot \\ \bot & \bot & (1,2,\{4\}) & \bot \\ (2,0,\{4\}) & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

$$T^{B,0} = \begin{pmatrix} \bot & \bot & \bot & (0,3,\{4\}) \\ \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \\ (3,0,\{4\}) & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.10 — The set of matrices T after initialization for the all-path query semantics (elements $T^{S_1,0}[i,j]$ and $T^{S,0}[i,j]$ are equal to $\bot = (0,0,\varnothing)$ for all i, j)

$$T^{S,1} = \begin{pmatrix} \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & (2,3,\{0\}) \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.11 — Elements of the matrix $T^{S,1}$ after first loop iteration for the all-path query semantics

$$T^{S,13} = \begin{pmatrix} (0,0,\{1\}) & \bot & \bot & (0,3,\{1\}) \\ (1,0,\{2\}) & \bot & \bot & (1,3,\{2\}) \\ (2,0,\{0\}) & \bot & \bot & (2,3,\{0\}) \\ \bot & \bot & \bot & \bot \end{pmatrix}$$

Figure 3.12 — Elements of the resulting matrix $T^{S,13}$ for the all-path query semantics

We can notice that for this example, in the resulting matrix T^S there are no elements with more than one intermediate vertex. Therefore, the resulting matrix is very similar to the corresponding matrix for the single-path query semantics but without the information about the heights of the derivation trees. However, in the general case, the number of found intermediate vertices for paths between two vertices can be more than one. After the algorithm constructs such matrices, it is known that they contain all such intermediate vertices, therefore, using the algorithm presented in Listing 6 all paths that satisfy the given constraints can be extracted. For example, all paths from the vertex 0 to the vertex 0 can be restored forming strings from the language $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$. The set of such paths is infinite and they form strings from the set $\{a^6b^6, a^{12}b^{12}, a^{18}b^{18}, \ldots\}$. For any $m \geqslant 1$, the path forming the string $a^{6m}b^{6m}$ traverses the cycle labeled by a — $2m$ times and the cycle labeled by b — $3m$ times. When implementing the algorithm for extracting all paths, it is necessary to limit the number of extracted paths and add the corresponding input parameter to the algorithm.

Thus, the demonstration of the described CFPQ algorithm was provided.

3.5 Implementation

In this section, we present the implementation details for the obtained matrix-based CFPQ algorithm. All the implementations listed in this section are available as part of the *CFPQ_PyAlgo*[1] platform.

The properties of real graphs in such areas as RDF data analysis and static program analysis were taken into account. Such graphs are quite large (up to tens of millions of vertices) but they are sparse (up to tens of millions of edges). For graphs of such sizes it is advisable to use parallel computing systems since the time spent on organizing parallel computing (for example, data exchange between the CPU and GPU) will be insignificant compared to the time of the analysis itself. Therefore, the implementations must use parallel computations and must store matrices using sparse formats. Since the discussed graphs are not hypersparse, the CSR format

[1] CFPQ_PyAlgo — platform for developing and testing of CFPQ algorithms: https://github.com/JetBrains-Research/CFPQ_PyAlgo (date of access: 14.01.2022).

was chosen to store sparse matrices. Next, the obtained implementations of the proposed algorithm will be described.

Reachability. To solve the CFPQ problem with the reachability query semantics, the operations of multiplication and addition over a set of Boolean matrices were used. And to implement this algorithm, libraries were chosen for the parallel calculation of these operations on sparse Boolean matrices on the CPU and GPU. The CPU implementation was created using the SuiteSparse:GraphBLAS library, and for the GPU implementation we used the cuBool library. It is worth noting that the SuiteSparse:GraphBLAS library contains built-in algebraic structures that can be used to compute operations on Boolean matrices. As a result, the following implementations were obtained:

- $\mathit{MtxReach}_{CPU}$ — a `Python` implementation using the pygraphblas package, which is a wrapper for the SuiteSparse:GraphBLAS library;
- $\mathit{MtxReach}_{GPU}$ — a `Python` implementation using the pycubool package, which is a wrapper for the cuBool library.

To solve CFPQ problem with other path query semantics, more complex types for matrix elements were used. This leads to a significant increase in memory consumption, which is especially critical for GPU implementations. The proposed matrix-based algorithm for the single-path and all-path query semantics can be implemented on the GPU, for example, using CUSP library that allows one to calculate matrix operations with user-defined data type. However, to evaluate such implementations on real data, further research is needed to optimize the proposed algorithm and the used algebraic structures. Such a study should focus on the algorithm memory consumption and implementation details of the selected GPU linear algebra library. In this thesis, the proposed algorithm for the single-path and all-path query semantics was implemented only on the CPU.

Single-path. The algorithm for the single-path query semantics was also implemented using the SuiteSparse:GraphBLAS library. However, instead of using one of the built-in algebraic structures, to calculate operations on matrices in this algorithm, it is necessary to define a user-defined data type and operations on them corresponding to the structure $\langle \mathit{PathIndex}, \oplus, \otimes, \bot \rangle$. In the result we have the following implementation:

- *MtxSingle*$_{CPU}$ — a `Python` implementation using the pygraphblas package, which is a wrapper for the SuiteSparse:GraphBLAS library.

All-path. The implementation for the all-path query semantics was also obtained using the SuiteSparse:GraphBLAS library. However, at the moment, the pygraphblas package does not allow one to use a user-defined data type and operations on them corresponding to the algebraic structure $\langle \mathit{AllPathIndex}, \oplus, \otimes, \perp \rangle$. Therefore, the implementation of the algorithm for the all-path query semantics was written in `C++` using the SuiteSparse:GraphBLAS library, and a `Python` wrapper was written for this algorithm. As a result, the following implementation was obtained:

- *MtxAll*$_{CPU}$ — a `Python` implementation using our own wrapper for the `C++` algorithm using the SuiteSparse:GraphBLAS library.

In addition, for the single-path and the all-path query semantics the algorithms for extracting the desired paths from the information contained in the constructed matrices were implemented in `Python`. It should be noted that in the implementation of the algorithm for extracting all paths, a parameter was additionally used that limits the length of the extracted paths for the process termination.

Chapter 4. A Kronecker Product-Based CFPQ Algorithm

In this chapter, we present a Kronecker product-based algorithm that does not require a transformation of the input context-free grammar and solves CFPQ problem with the reachability, the single-path, and the all-path query semantics. Also, the correctness and time complexity of the obtained algorithm are formulated and proved. In addition, the implementation details are given, as well as a step-by-step demonstration of the proposed algorithm.

4.1 Algorithm Construction

In this section, we describe the process of constructing a Kronecker product-based CFPQ algorithm.

Suppose that an input labeled graph $\mathcal{G} = \langle V, E, L \rangle$ is given, as well as an input CFL that is used as path constraints. The applicability of matrices to describe graphs has already been shown in the previous chapter. However, for the input context-free path constraints in the proposed algorithm it is necessary to build a CFG and transform it into the WCNF. Such a transformation of the grammar can lead to at least a quadratic increase in its size [36]. However, the algorithm proposed in this chapter uses recursive automata to describe the input context-free path constraints and does not require a transformation of the CFG.

It is known that any CFG can be represented by a recursive automaton [39]. In turn, the recursive automaton can be represented as a graph. Therefore, context-free path constraints can be expressed in terms of linear algebra objects such as matrices. In this case, a CFPQ problem can be solved by finding the intersection of the recursive automaton corresponding to the context-free path constraints and the finite automaton corresponding to the input graph. The proposed algorithm is based on a generalization of the FSA intersection algorithm. As shown in section 2.2, the intersection of automata can be calculated using the Kronecker product applied to matrices from Boolean decompositions of adjacency matrices for graph representations of these automata. Thus, using the Kronecker product operation $\times$ and the element-wise addition operation $\bigvee$ defined over the semiring

$\langle\{0,1\},\vee,\wedge,0\rangle$, a matrix will be constructed that describes the intersection of the recursive automaton for the context-free constraints and the finite automaton for the input graph. However, such a recursive automaton may contain transitions with nonterminal symbols that have a special logic of recursive calls, which requires the addition of a separate step — calculation of the transitive closure of the constructed matrix for processing these symbols. After calculating the transitive closure, new edges (i, A, j) can be added to the graph when a path $i\pi j$ is found for which $A \Rightarrow_G \lambda(\pi)$. After updating the graph, the process is repeated until convergence.

The described Kronecker product-based CFPQ algorithm is shown in the Listing 7. The key idea of the algorithm is to iteratively calculate the Kronecker product for the Boolean decomposition of the adjacency matrices $\mathcal{M}_1$ of the recursive automaton and the Boolean decomposition of the adjacency matrices $\mathcal{M}_2$ of the input graph, followed by computing the transitive closure of the resulting matrix. The most time-consuming steps in the proposed algorithm are the calculation of the Kronecker product and the transitive closure. In addition, the proposed algorithm uses the *getNonterminals* function, which returns the set of nonterminal symbols of recursive automaton transitions between given two states.

The result of the algorithm shown in the Listing 7 is a Boolean decomposition $\mathcal{M}_2$ of adjacency matrices of the input graph with added edges with nonterminal symbols as labels that describe the reachability of vertices in accordance with the given context-free constraints, as well as the matrix M_3, which contains the necessary information to extract all the desired paths. Next, we will show how the proposed algorithm allows one to solve CFPQ problem.

Reachability. This problem can be solved using the set of matrices $\mathcal{M}_2$. Namely, in the graph $\mathcal{G}$ there exists a path from the vertex i to the vertex j that satisfies the given context-free constraints only if $\mathcal{M}_2^S[i,j] = 1$.

Single-path and all-path. CFPQ problem with these path query semantics can be solved using the matrix M_3 that for any nonterminal symbol $A \in N$ and for any pair of vertices (i, j) contains enough information to construct all paths $i\pi j$ such that $A \Rightarrow_G \lambda(\pi)$. Therefore, the Listing 8 presents an algorithm for recovering all paths in the graph that correspond to the given context-free constraints. It is also worth noting that the presented path extraction algorithm is based on the breadth-first search method applied on the graph corresponding to the matrix M_3. And since the

Listing 7 A Kronecker product-based CFPQ algorithm

```
function KroneckerBasedCFPQ($\mathcal{G} = \langle V, E, L \rangle$, $G = \langle N, \Sigma, P, S \rangle$)
    $n \leftarrow$ |V|
    $R \leftarrow$ a recursive automaton for grammar $G$ with $m$ states
    $\mathcal{M}_1 \leftarrow$ the Boolean decomposition of the adjacency matrix for $R$ with matrix sizes $m \times m$
    $\mathcal{M}_2 \leftarrow$ the Boolean decomposition of the adjacency matrix for $\mathcal{G}$ with matrix sizes $n \times n$
    $C_3, M_3 \leftarrow$ empty matrices of size $mn \times mn$
    for $q \in \{0, \dots, m-1\}$ do
        for $A \in getNonterminals(R, q, q)$ do
            for $i \in \{0, \dots, n-1\}$ do
                $\mathcal{M}_2^A[i, i] \leftarrow 1$
    while any matrix in $\mathcal{M}_2$ is changing do
        $M_3 \leftarrow \bigvee_{M^A \in \mathcal{M}_1 \times \mathcal{M}_2} M^A$                ▷ Kronecker product
        $C_3 \leftarrow transitiveClosure(M_3)$                ▷ Transitive closure calculation
        for $(i, j) \mid C_3[i, j] = 1$ do
            $q_1, q_2 \leftarrow getStates(i, j, n)$
            $x, y \leftarrow getCoordinates(i, j, n)$
            for $A \in getNonterminals(R, q_1, q_2)$ do
                $\mathcal{M}_2^A[x,y] \leftarrow 1$
    return $\mathcal{M}_2, M_3$
function getStates($i, j, n$)    ▷ Obtaining state numbers of the automaton $R$ by cell indices in the matrix $C_3$
    return $\lfloor i/n \rfloor, \lfloor j/n \rfloor$
function getCoordinates($i, j, n$)    ▷ Obtaining vertex numbers of graph $\mathcal{G}$ by cell indices in the matrix $C_3$
    return $i \bmod n, j \bmod n$
```

algorithm [13] is known for this graph traversal method, which uses linear algebra operations, then these operations can be used to implement the BFS traversal function, which returns paths from the selected initial vertex to the final vertex. However, the number of such paths can be infinite, so when implementing the BFS function it is necessary to use lazy computations. In addition, the parameter k is used to limit the number paths in the graph returned by the algorithm. This algorithm returns k of found paths with numbers of vertices that form the lexicographically smallest sequences. Thus, the proposed algorithm allows one to extract any number of paths that satisfy the given context-free constraints.

Thus, the algorithms presented in the Listings 7 and 8 allow one to solve the CFPQ problem with the all-path query semantics using the Kronecker product.

Listing 8 All paths extraction algorithm for the Kronecker product-based CFPQ algorithm

```
$\mathcal{M}_2 \leftarrow$ the Boolean decomposition of the resulting adjacency matrix of the algorithm in the Listing 7
$M_3 \leftarrow$ the resulting matrix $M_3$ of the algorithm in the Listing 7
$R \leftarrow$ a recursive automaton for the input context-free constraints
$\mathcal{M}_1 \leftarrow$ the Boolean decomposition of the adjacency matrix for $R$
function ExtractAllPaths($v_s, v_f, A, k$)
    $q_A^0 \leftarrow$ the initial state of the automaton for the nonterminal $A$
    $F_A \leftarrow$ the final states of the automaton for the nonterminal $A$
    $paths \leftarrow \bigcup_{q_A^f \in F_A} \text{BFS}(M_3, (q_A^0, v_s), (q_A^f, v_f))$
    $resultPaths \leftarrow \varnothing$
    for $path \in paths$ do
        $currentPaths \leftarrow \varnothing$
        for $((s_i, v_i), (s_j, v_j)) \in path$ do
            $newEdges \leftarrow \{(v_i, t, v_j) \mid M_2^t[v_i, v_j] \wedge M_1^t[s_i, s_j]\}$
            $currentPaths \leftarrow currentPaths \cdot newEdges$                    ▷ Path concatenation
            $k' \leftarrow k - |resultPaths| - |currentPaths|$
            $newSubPaths \leftarrow \bigcup_{\{N \mid M_2^N[v_i,v_j] \wedge M_1^N[s_i,s_j]\}}$ ExtractAllPaths($v_i, v_j, N, k'$)
            $currentPaths \leftarrow currentPaths \cdot newSubPaths$                 ▷ Path concatenation
        $resultPaths \leftarrow resultPaths \cup currentPaths$
        if $|resultPaths| \geqslant k$ then
            return TopK($resultPaths$)
    return $resultPaths$
function TopK($paths$)                                                    ▷ Choice of $k$ paths
    $kPaths \leftarrow k$ paths from $paths$ with the smallest vertex numbers in lexicographic order
    return $kPaths$
```

4.2 Correctness of the Algorithm

In this section, we formulate and prove statements about the correctness and termination of the described algorithm.

To prove the termination of the presented CFPQ algorithm, we will use the partial order relation $\preceq_{rel}$ defined in section 3.2 on the elements of the used Boolean semiring $\langle\{0,1\}, \vee, \wedge, 0\rangle$. First, we prove the theorem on the monotonicity of the algorithm presented in the Listing 7. Further, for any indices (i,j) and for any nonterminal $A \in N$, we will use the following notation:

- $\mathcal{M}_2^{A,0}[i,j]$, $M_3^0[i,j]$, and $C_3^0[i,j]$ for values in cells $\mathcal{M}_2^A[i,j]$, $M_3[i,j]$, and $C_3[i,j]$ after matrix initialization in lines 2–10 of the algorithm shown in the Listing 7;

- $\mathcal{M}_2^{A,k}[i,j]$, $M_3^k[i,j]$, and $C_3^k[i,j]$ for values in cells $\mathcal{M}_2^A[i,j]$, $M_3[i,j]$, and $C_3[i,j]$ after k iterations of the loop in lines 11–18, for $k \geqslant 1$.

Theorem 4.2.1 (Algorithm monotonicity)**.** Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph and $G = \langle N, \Sigma, P, S\rangle$ is the input CFG for the algorithm presented in the Listing 7. Then for any indices (i,j) and for any nonterminal $A \in N$, the values in the cells $\mathcal{M}_2^A[i,j]$, $M_3[i,j]$, and $C_3[i,j]$ increase monotonically according to the partial order relation $\preceq_{rel}$.

Proof. It is necessary to prove that for any $k \geqslant 1$, $\mathcal{M}_2^{A,k-1}[i,j] \preceq_{rel} \mathcal{M}_2^{A,k}[i,j]$, $M_3^{k-1}[i,j] \preceq_{rel} M_3^k[i,j]$, and $C_3^{k-1}[i,j] \preceq_{rel} C_3^k[i,j]$. For $k = 1$ this statement is correct since all elements of the matrices M_3 and C_3 are equal to $\perp = 0$, and the values of the elements of the matrices $\mathcal{M}_2^A$ can change only in the line 18 from 0 to 1. For $k > 1$ this statement is correct since, by the properties of the defined Kronecker product and transitive closure operations, changes in some values of the elements of the matrices $\mathcal{M}_2^{A,k-1}$ from 0 to 1 can only lead to the same changes for some elements of the matrices M_3^{k-1} and C_3^{k-1}. This proves the statement of the theorem. □

A consequence of the proved theorem is the following theorem on the termination of the algorithm presented in the Listing 7.

Theorem 4.2.2 (Algorithm termination)**.** The algorithm shown in the Listing 7 terminates in a finite number of steps.

Proof. According to the theorem 4.2.1, the values in the cells of the matrices from the set $\mathcal{M}_2^A$ increase monotonically. And since the matrices and the sets of possible values of the elements of these matrices are finite, the algorithm presented in the Listing 7 terminates in a finite number of steps. □

Moreover, using arguments similar to those in the proofs of the lemma 3.2.2 and the theorem 3.2.3, the following theorem can be proved.

Theorem 4.2.3 (Algorithm correctness)**.** Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph and $G = \langle N, \Sigma, P, S\rangle$ is the input CFG for the algorithm presented in the Listing 7. Then for any pair of vertices (i,j) of the graph $\mathcal{G}$, for any nonterminal $A \in N$, the value in the cell $\mathcal{M}_2^{A,k}[i,j] = 1$ only if there exists a path $i\pi j$ such that $A \Rightarrow_G \lambda(\pi)$.

Proof. By induction on the minimum heights of the derivation trees corresponding to visited paths in the graph. □

A consequence of the theorem 4.2.3 is the correctness of the M_3 matrix returned by the algorithm from the Listing 7 that describing the intersection of a finite automaton for the input graph and a recursive automaton for the input context-free constraints. Thus, the proposed algorithm for extracting all paths in a graph, based on a breadth-first search of the matrix M_3, is also correct, which is shown in the following theorem.

Theorem 4.2.4 (Correctness of the path extraction algorithm)**.** Suppose that $\mathcal{G} = \langle V, E, L\rangle$ is the input graph, $G = \langle N, \Sigma, P, S\rangle$ is the input CFG, and $(\mathcal{M}_2, M_3)$ is the result returned by the algorithm shown in the Listing 7. Then, for any pair of vertices (i, j) and any nonterminal $A \in N$, the algorithm presented in the Listing 8 will construct a set of the given number of paths $i\pi j$ such that there exists a derivation tree for the string $\lambda(\pi)$ and the nonterminal A of the grammar G, if such paths exist.

4.3 Time Complexity of the Algorithm

In this section, we present the worst-case time complexity of the proposed algorithm.

We assume that all operations on matrix elements are computed in $O(1)$ elementary operations. In addition, we estimate the worst-case number of elementary operations needed to compute operations on Boolean matrices as:

- $O(n^2)$ — for the element-wise addition operation $\bigvee$ of two Boolean matrices of size $n \times n$,
- $O(m^2n^2)$ — for the Kronecker product operation $\times$ of two Boolean decompositions of the adjacency matrix of size $m \times m$ and the adjacency matrix of size $n \times n$,
- $O(n^3 \log n)$ — for the transitive closure of a Boolean matrix of size $n \times n$.

Note that calculating the transitive closure of a Boolean matrix in such number of elementary operations can be done using the repeated squaring technique [71].

Then the following theorem on the time complexity of the CFPQ algorithm proposed in this chapter is correct.

Theorem 4.3.1 (The time complexity of the Kronecker product-based CFPQ algorithm)**.** Suppose that $\mathcal{G} = \langle V, E, L \rangle$ is the input graph, $G = \langle N, \Sigma, P, S \rangle$ is the input CFG, and R is the recursive automaton for the grammar G with a set of states Q. Then for the algorithm presented in the Listing 7 has the following worst-case time complexity: $O(|N||Q|^3|V|^5 \log(|Q||V|))$.

Proof. In lines 7–10 of the algorithm, for each state q the values in cells $[q, q]$ are read in $|N|$ Boolean matrices of size $|Q| \times |Q|$ using the function *getNonterminals*, and then for each resulting nonterminal A and each vertex i of the graph, a write operation is made in the cell $[i, i]$ of the corresponding Boolean matrix. Therefore $O(|N||Q||V|)$ elementary operations are performed in these lines.

Next, consider the loop in lines 11–18. The algorithm continues calculations until any matrix from the set $\mathcal{M}_2$ changes. The total number of matrices in this set that can be changed is $|N|$ (they correspond to nonterminal symbols of the grammar G) and each of them has $|V|^2$ elements. At each iteration, only some zero values of the cells of these matrices can change to the value 1. Therefore, the maximum number of iterations of the discussed loop can be achieved in the case when the value of only one cell in one matrix changes for each iteration. Thus, the maximum number of iterations of the discussed loop is $|N||V|^2$. And at each iteration, the following is performed: calculation of the Kronecker product in $O(|Q|^2|V|^2)$ operations; calculation of the element-wise addition of the resulting Boolean matrices in $O(|Q|^2|V|^2)$ operations; computing the transitive closure of a Boolean matrix of size $|Q||V| \times |Q||V|$ in $O(|Q|^3|V|^3 \log(|Q||V|))$ operations; and calculations in lines 14–18 in $O(|N||Q||V|)$ operations. The most time-consuming is the computation of the transitive closure of a Boolean matrix of size $|Q||V| \times |Q||V|$, so the algorithm will terminate in $O(|N|||V|^2(|Q|^3|V|^3 \log(|Q||V|)))$. From this we obtain the stated worst-case time complexity of the algorithm presented in the Listing 7. □

Note that the given time complexity of the algorithm presented in the Listing 7 can be significantly improved for sparse graphs. It is also possible to apply a fast transitive closure computation by using the incremental dynamic transitive closure technique [75]. The key idea of this approach is to recalculate reachability information only for those vertices that become reachable after insertion of a certain edge in a graph.

In this work, the time complexity of the algorithm presented in the Listing 8 is not given since it depends on the specific implementation of the *BFS* function.

4.4 An Example

In this section, a step-by-step demonstration of the proposed algorithm is provided using the example based on the CFL $\mathcal{L} = \{a^n b^n \mid n \geqslant 1\}$. Consider the graph $\mathcal{G}_1$ shown in Figure 1.1. For the algorithm presented in the previous chapter it was necessary to represent the input context-free constraints using a CFG in the WCNF. An example of such a grammar for the language $\mathcal{L}$ was discussed in section 3.4. However, for the algorithm shown in the Listing 7 it is not necessary to transform the grammar to a normal form. Therefore, consider a grammar with fewer derivation rules and nonterminal symbols. The derivation rules for such a CFG G are as follows.

$$\begin{array}{rcl} 0: S & \to & a\ S\ b \\ 1: S & \to & a\ b \end{array}$$

The algorithm proposed in the Listing 7 uses the representation of the CFG as a recursive automaton. Such an automaton $R = \langle \{a,b\}, \{S\}, S, \{C_S\} \rangle$ for the CFG G is presented in Figure 1.2.

The label of the starting automaton C_S is the starting nonterminal S of the grammar, the initial state of this automaton is the state q_S^0, and the set of final states is $\{q_S^3\}$. The algorithm will use Boolean decompositions of $\mathcal{M}_1$ and $\mathcal{M}_2$ adjacency matrices of the recursive automaton and the input graph. For a more compact representation in this example, we will use the adjacency matrices M_1 and M_2 for the automaton R and the graph $\mathcal{G}_1$, which look like this (zero values are omitted):

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

In lines 7–10 the algorithm uses all ε-transitions in the automaton R and adds information about empty paths in the graph to the matrix M_2. However, in the used example the recursive automaton does not contain ε-transitions, so the matrix M_2 does not change in these lines.

Next, the algorithm loops through lines 11–18 while the M_2 matrix changes. We present the values of the matrices M_2, M_3, and C_3 at each iteration of the algorithm. At the first iteration of the loop, the Kronecker product $M_3^1 = M_1 \times M_2^0$ and the transitive closure C_3^1 are calculated as follows (new nonzero values are highlighted).

$$M_3^1 = \left(\begin{array}{cccc|cccc|cccc|cccc}
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}\right)$$

$$C_3^1 = \left(\begin{array}{cccc|cccc|cccc|cccc}
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \colorbox{lightgray}{$\mathbf{1}$} \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}\right)$$

After computing the transitive closure, the cell $C_3^1[1, 15]$ contains a nonzero value. In order to understand what this means for the original input graph $\mathcal{G}_1$ and the recursive automaton R, we analyze the *getStates* and *getCoordinates* functions used in the algorithm. Using the *getStates* function, the values of the states q_1 and q_2 of the recursive automaton are calculated, which correspond to some found path in the graph. To calculate the initial and final vertices of this path x and y the *getCoordinates* function is used. Then, information is added to the matrix M_2 that there is a path in the graph from the vertex x to the vertex y, the labels on the edges of which form a word derived from some nonterminal on transitions from the state q_1 to the state q_2 of the recursive automaton. For example, after the first iteration a new nonzero value in cell $C_3^1[1, 15]$ of matrix C_3^1 means the following.

- The states of the automaton are $q_1 = 0$ and $q_2 = 3$ since the cell $C_3^1[1, 15]$ is in the upper right block of the matrix C_3 with the block coordinates $(0, 3)$.
- The vertices $x = 1$ and $y = 3$ since the cell $C_3^1[1, 15]$ has exactly these coordinates inside its block.
- The *getNonterminals* function returns the set $\{S\}$ since it is the only nonterminal on transitions from the state 0 to the state 3 of the recursive automaton R.
- Thus, there is a path in the graph from the vertex 1 to the vertex 3, the edge labels of which form a word derivable from the nonterminal S.

As a result, at the first iteration of the loop the nonterminal S is added to the cell $M_2^1[1, 3]$. The updated matrix M_2 and the corresponding updated graph are presented in Figure 4.1.

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

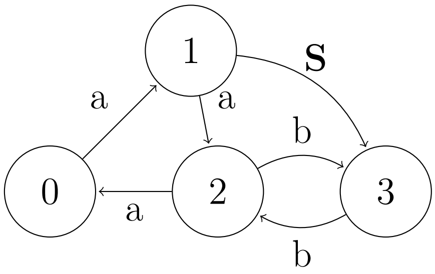


Figure 4.1 — The updated matrix M_2 and the corresponding updated graph after the first iteration of the algorithm

At the second iteration of the loop, we obtain the following matrices M_3^2 and C_3^2:

$$M_3^2 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & \mathbf{1} & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

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

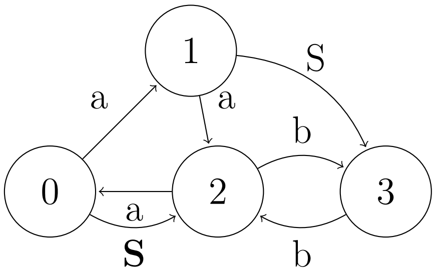


Figure 4.2 — The updated matrix M_2 and the corresponding updated graph after the second iteration

$$C_3^2 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & . & \mathbf{1} & . & . & \mathbf{1} & . \\
. & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & \mathbf{1} & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right).$$

New nonzero values of the C_3^2 matrix appeared in the cells $[0, 11]$, $[0, 14]$, and $[5, 14]$. However, only the cell with the index $[0, 14]$ corresponds to the states q_1 and q_2 of the automaton, between which there is at least one transition with a nonterminal symbol. Therefore, only the value in the cell $[5, 14]$ will affect the update of the M_2 matrix. The updated matrix M_2 and the corresponding updated graph after the second iteration are presented in Figure 4.2.

The matrices M_2 and C_3 calculated on the remaining iterations of the loop are presented in Figures 4.3 and 4.5. In this example, the number of the last loop iteration is 7, at which no new nonzero values are added to the matrix M_2 and the algorithm terminates.

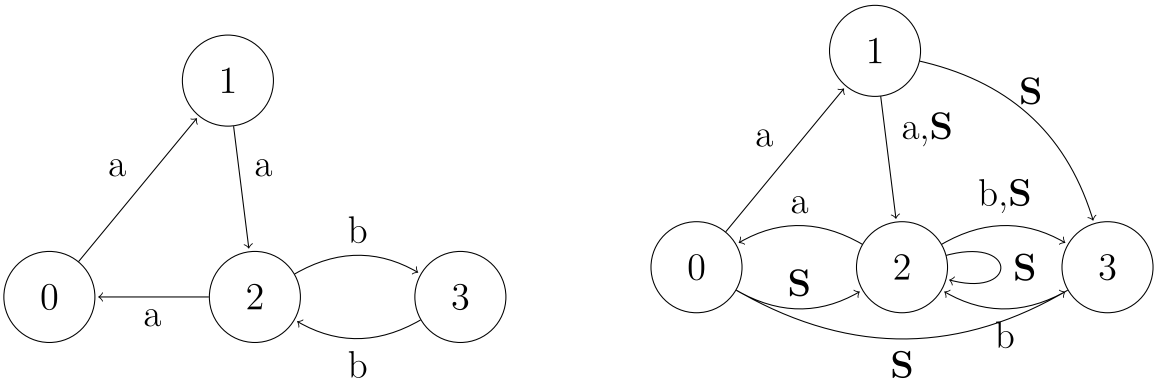
The original graph and the resulted graph with all the added edges are shown in Figure 4.4.

The result of the algorithm shown in the Listing 7 is the matrices M_2 and M_3. The matrix M_2 contains all information about the reachability in the graph in accordance with the given context-free constraints, and is shown in Figure 4.4б. And the matrix M_3 contains information sufficient to extract all the desired paths.

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

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

Figure 4.3 — Updated matrix M_2 for algorithm iterations from 3 to 6



a) The initial input graph $\mathcal{G}_1$ б) The resulting graph

Figure 4.4 — The initial input graph $\mathcal{G}_1$ and the graph corresponding to the resulting Boolean decomposition of the adjacency matrix $\mathcal{M}_2$

$$C_3^3 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & . & 1 & . & . & 1 & . \\
. & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & 1 & . & . & . & . & . & \mathbf{1} & . & . & . & . & \mathbf{1} \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & \mathbf{1} \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & 1 & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

$$C_3^4 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & . & 1 & . & . & 1 & . \\
. & . & . & . & . & . & 1 & . & . & . & . & \mathbf{1} & . & . & \mathbf{1} & 1 \\
. & . & . & . & 1 & . & . & . & . & . & 1 & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & 1 & . \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & \mathbf{1} & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

$$C_3^5 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & \mathbf{1} & 1 & . & . & 1 & \mathbf{1} \\
. & . & . & . & . & . & 1 & . & . & . & . & 1 & . & . & 1 & 1 \\
. & . & . & . & 1 & . & . & . & . & . & 1 & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & 1 & 1 & . & . & 1 & \mathbf{1} \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & 1 & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

$$C_3^6 = \left(\begin{array}{cccc|cccc|cccc|cccc}
. & . & . & . & . & 1 & . & . & . & . & 1 & 1 & . & . & 1 & 1 \\
. & . & . & . & . & . & 1 & . & . & . & . & 1 & . & . & 1 & 1 \\
. & . & . & . & 1 & . & . & . & . & . & 1 & \mathbf{1} & . & . & \mathbf{1} & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & 1 & 1 & . & . & \mathbf{1} & 1 \\
. & . & . & . & . & . & . & . & . & . & 1 & 1 & . & . & 1 & 1 \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & 1 & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

Figure 4.5 — The transitive closure matrix C_3 on iterations of the algorithm from 3 to 6

4.5 Implementation

In this section, we present the implementation details for the obtained Kronecker product-based CFPQ algorithm.

The properties of real graphs were taken into account. In this case, the arguments from the section 3.5 remain applicable, as a result of which the choice was made to use parallel computing and a sparse format for storing matrices.

The proposed algorithm uses the Kronecker product over Boolean matrices, as well as the operation of calculating the transitive closure of a matrix. The algorithm was implemented on the CPU using the SuiteSparse:GraphBLAS library and on the GPU using the cuBool library, in which the Kronecker product operation is implemented and the transitive closure can be calculated using a series of Boolean matrix multiplications (squaring the matrix M_3). As a result, the following implementations were obtained:

- $KronAll_{CPU}$ — a `Python` implementation using the pygraphblas package, which is a wrapper for the SuiteSparse:GraphBLAS library;
- $KronAll_{GPU}$ — a `Python` implementation using the pycubool package, which is a wrapper for the cuBool library.

Note that the proposed algorithm allows one to construct matrices that contain information sufficient to construct any given number of paths that correspond to the given context-free path constraints. The answer to the reachability problem can be obtained using the $\mathcal{M}_2$ matrices computed by developed implementations. For the single-path and the all-path query semantics, the algorithm for the desired paths extraction presented in Listing 8 was implemented using the information contained in the matrices $\mathcal{M}_2$ and M_3.

Implementations $KronAll_{CPU}$ and $KronAll_{GPU}$ are available as part of the *CFPQ_PyAlgo* platform.

Chapter 5. Experimental study

The correctness of the proposed algorithms is formally proved in previous chapters. However, their performance requires experimental evaluation and comparison with existing solutions on real data.

5.1 Experimental Setup

In this work, it has been shown how the proposed approach can be used to obtain CFPQ algorithms and their implementations. The *goal* of this experimental study is to answer the following *questions*.

[Q1] What is the performance of the obtained CFPQ implementations for the reachability query semantics compared to existing solutions on real data?

[Q2] What is the performance of the obtained CFPQ implementations for the single-path and all-path query semantics compared to existing solutions on real data?

[Q3] What are the overhead costs of storing information about the found paths in the obtained implementations compared to the obtained reachability implementations?

[Q4] What is the performance of the obtained CFPQ implementations that do not require a transformation of the input CFG compared to other proposed implementations?

To answer these questions, it is necessary to calculate the following two *metrics*:

[M1] the implementation running time,

[M2] the implementation memory consumption.

In this experimental study, we select graphs obtained from real RDF data and from `C/C++` programs. Graph characteristics are presented in Tables 2 and 3. These tables contain the graph number; the number of graph vertices and edges; and the number of some edges with labels that later appear in the context-free path constraints.

Table 2 — Graph characteristics for the RDF analysis [5]*

№	Graph	$\|V\|$	$\|E\|$	$\#subClassOf$	$\#type$
1	eclass	239,111	360,248	90,962	72,517
2	go	582,929	1,437,437	94,514	226,481
3	go_h	45,007	490,109	490,109	0
4	taxonomy	5,728,398	14,922,125	2,112,637	2,508,635
5	taxonomy_h	2,112,625	32,876,289	32,876,289	0

* $|V|$ is a number of graph vertices, $|E|$ is a number of graph edges, $\#subClassOf$ and $\#type$ are number of edges with labels *subClassOf* and *type* respectively.

Table 3 — Graph characteristics for the static program analysis [76]*

№	Graph	$\|V\|$	$\|E\|$	$\#a$	$\#d$
6	apache_httpd_ptg	1,721,418	1,510,411	362,799	1,147,612
7	arch_after_inline	3,448,422	2,970,242	671,295	2,298,947
8	block_after_inline	3,423,234	2,951,393	669,238	2,282,155
9	crypto_after_inline	3,464,970	2,988,387	678,408	2,309,979
10	drivers_after_inline	4,273,803	3,707,769	858,568	2,849,201
11	fs_after_inline	4,177,416	3,609,373	824,430	2,784,943
12	init_after_inline	2,446,224	2,112,809	481,994	1,630,815
13	ipc_after_inline	3,401,022	2,931,498	664,151	2,267,347
14	kernel_after_inline	11,254,434	9,484,213	1,981,258	7,502,955
15	lib_after_inline	3,401,355	2,931,880	664,311	2,267,569
16	mm_after_inline	2,538,243	2,191,079	498,918	1,692,161
17	net_after_inline	4,039,470	3,500,141	807,162	2,692,979
18	postgre_sql_ptg	5,203,419	4,678,543	1,209,597	3,468,946
19	security_after_inline	3,479,982	3,003,326	683,339	2,319,987
20	sound_after_inline	3,528,861	3,049,732	697,159	2,352,573

* $|V|$ is a number of graph vertices, $|E|$ is a number of graph edges, $\#a$ and $\#d$ are number of edges with labels a and d respectively.

This experimental study will perform two analyses: RDF data analysis [5] and pointer analysis of `C/C++` programs [76]. Both analyses use languages describing different balanced bracket sequences with two types of brackets as path constraints. The derivation rules of the CFG G_{RDF} for the RDF analysis are as follows.

$$\begin{array}{lllllll} 0: & S & \rightarrow & \overline{subClassOf}\ S\ subClassOf & 2: & S & \rightarrow & \overline{type}\ S\ type \\ 1: & S & \rightarrow & \overline{subClassOf}\ subClassOf & 3: & S & \rightarrow & \overline{type}\ type \end{array}$$

Also, the derivation rules of the CFG G_C for the pointer analysis of `C/C++` programs are as follows.

$$\begin{array}{llllllll} 0: & S & \rightarrow & \overline{d}\ V\ d & 4: & V_3 & \rightarrow & a\ V_2\ V_3 \\ 1: & V & \rightarrow & V_1\ V_2\ V_3 & 5: & V_1 & \rightarrow & \varepsilon \\ 2: & V_1 & \rightarrow & V_2\ \overline{a}\ V_1 & 6: & V_2 & \rightarrow & \varepsilon \\ 3: & V_2 & \rightarrow & S & 7: & V_3 & \rightarrow & \varepsilon \end{array}$$

The CFL generated by the grammar G_{RDF} describes constraints for finding graph vertices that are on the same level of some hierarchy [5], and the language generated by the grammar G_C allows one to find a set of heap objects that can flow to pointer variables [2].

Note that these grammars also use labels $\overline{subClassOf}$, $\overline{type}$, $\overline{texta}$, and $\overline{d}$. Because for described analyses it is necessary to visit paths in a graph with inverse edges. Therefore, for all graph edges (i, x, j) we explicitly add the inverse edges $(j, \overline{x}, i)$ where $x \in \{subClassOf, type, a, d\}$. All graphs and corresponding grammars are available as part of the dataset *CFPQ_Data*[1].

The proposed implementations are described in sections 3.5 and 4.5. Also, for described analyses we use existing open-source solutions that we were able to run. The complete list of notations for the implementations used in this experimental study is given below.

- **MRC** is the $MtxReach_{CPU}$ implementation of the proposed matrix-based CFPQ algorithm for the reachability query semantics.
- **MRG** is the $MtxReach_{GPU}$ GPU implementation of the proposed matrix-based CFPQ algorithm for the reachability query semantics.
- **MSC** is the $MtxSingle_{CPU}$ implementation of the proposed matrix-based CFPQ algorithm for the single-path query semantics.

[1] CFPQ_Data is the dataset for CFPQ problem: https://jetbrains-research.github.io/CFPQ_Data/ (date of access: 14.01.2022).

- **MAC** is the $MtxAll_{CPU}$ implementation of the proposed matrix-based CFPQ algorithm for the all-path query semantics.
- **KAC** is the $KronAll_{CPU}$ implementation of the proposed Kronecker product-based CFPQ algorithm for the all-path query semantics.
- **KAG** is the $KronAll_{GPU}$ GPU implementation of the proposed Kronecker product-based CFPQ algorithm for the all-path query semantics.
- **Graspan**[2] is a disk-based highly parallel interprocedural static analysis engine [76]. This implemenation is CPU-based and written in `C++`. Note that there is also a GPU *Graspan* implementation, but we were unsuccessful in running it.
- **LL**[3] is the implementation of the CFPQ algorithm [24] for the reachability query semantics based on the LL parsing algorithm. This implemenation is CPU-based and written in `Go` language.
- $\mathbf{GLL_R}$ and $\mathbf{GLL_A}$[4] the implementations of the CFPQ algorithm [26] for the single-path and all-path query semantics based on the GLL parsing algorithm. These implemenations are CPU-based and written in `Java`. While the GLL_A implementation constructs the SPPF to solve the CFPQ problem with all-path query semantics, the GLL_R implementation solves the CFPQ problem with the reachability query semantics and omits this construction.

In addition, these implementations use different context-free path constraints representations. For some implementations the CFGs G_{RDF} and G_C were converted to the WCNF, and for some implementations the corresponding recursive automata were constructed. Note that a transformation of the grammar G_{RDF} into the WCNF increased the number of nonterminals from 1 to 7 and the number of derivation rules from 4 to 10, while such a transformation of the the grammar G_C increased the number of nonterminals from 5 to 13 and the number of derivation rules from 8 to 34.

To compare the performance of the discussed implementations, the average running time for 5 runs according to **[M1]** were calculated, as well as the consumed memory according to **[M2]**. For the single-path and all-path query semantics the

[2] *Graspan* is a disk-based highly parallel interprocedural static analysis engine: https://github.com/Graspan/Graspan-C (date of access: 14.01.2022).

[3] *LL* is the implementation of the CFPQ algorithm for the reachability query semantics based on the LL parsing algorithm: https://gitlab.com/ciromoraismedeiros/rdf-ccfpq (date of access: 14.01.2022).

[4] GLL_R and GLL_A are the implementations of the CFPQ algorithm for the single-path and all-path query semantics based on the GLL parsing algorithm: https://github.com/JetBrains-Research/GLL4Graph (date of access: 14.01.2022).

path extraction time was not measured. The type of information about the paths to be extracted, as well as the choice from the variety of ways to obtain this information strongly depends on the application areas. For evaluation, we use a PC with Ubuntu 18.04 installed. In has Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, DDR4 64GB RAM, a 64GB swap file, and GeForce GTX 1070 GPU with 8GB DDR5 memory.

5.2 Results

[Q1]. To answer the first question, we compared all the implementations that can solve the CFPQ problem with the reachability query semantics for the RDF analysis and the static program analysis. For the Kronecker product-based algorithm there is no specialized implementation for solving the reachability problem, and for this comparison the implementations $KronAll_{CPU}$ and $KronAll_{GPU}$ were used. For other algorithms only implementations specialized for solving the CFPQ problem with the reachability query semantics were chosen. According to **[M1]**, the average running time in seconds of the discussed implementations for the RDF analysis is presented in Table 4, and for pointer analysis in `C/C++` programs — in Table 5. These tables contain the graph numbers and the size of the resulting set for the CFPQ problem with the reachability query semantics. That is, *#result* is the number of pairs of vertices (i, j) such that there exists at least one path from the vertex i to the vertex j that forms a word from the language generated by CFGs G_{RDF} and G_C. In addition, the smallest analysis time for each graph is highlighted.

The RDF analysis results presented in Table 4. The smallest analysis time was demonstrated by the implementations $MtxReach_{CPU}$ (*MRC*) and $MtxReach_{GPU}$ (*MRG*) of the proposed matrix-based algorithm. Note that all implementations $MtxReach_{CPU}$, $MtxReach_{GPU}$, $KronAll_{CPU}$ (*KAC*), and $KronAll_{GPU}$ (*KAG*) demonstrate smaller running time than existing solutions. The implementations *LL* and GLL_R demonstrate comparable running time to the *Graspan* tool. While the proposed implementations analyze these graphs up to 277 times faster than the best considered existing solution. Note that for this analysis, the most time-consuming is the graph *taxonomy_h* with number 5 and with more than 5 million pairs of vertices in the resulting set. The implementation GLL_R did not complete the analysis in 5 hours and this is marked by OOT (Out Of Time) in the table,

Table 4 — The running time in seconds of the CFPQ algorithms with the reachability query semantics for the RDF analysis [5]*

№	*#result*	*Graspan*	*LL*	GLL_R	*MRC*	*MRG*	*KAC*	*KAG*
1	90,994	2.5	9.3	1.5	**0.1**	**0.1**	0.3	0.2
2	640,316	12.8	46.0	5.6	1.2	**0.8**	3.2	3.1
3	588,976	0.9	51.4	3.7	**0.1**	0.2	0.2	0.2
4	151,706	3,938.0	253.1	45.5	**1.0**	**1.0**	6.0	3.9
5	5,351,657	3,817.1	3,023.8	OOT	**10.9**	OOM	11.7	OOM

* *#result* is the size of resulting set; *Graspan* is a static program analysis tool [76]; *LL* is the implementation of the algorithm [24] based on the LL parsing algorithm; GLL_R is the reachability implementation of the algorithm [26] based on the GLL parsing algorithm; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

while the implementations $MtxReach_{GPU}$ and $KronAll_{GPU}$ terminate because of out of memory error and this is marked by OOM.

To compare implementations using larger graphs with millions of vertices and edges, as well as using more time-consuming analysis with tens of millions pairs of vertices in the resulting sets, the results presented in Table 5 are more revealing. For pointer analysis in `C/C++` programs, the GPU implementation $MtxReach_{GPU}$ of the proposed matrix-based algorithm showed the best results on all graphs. Such results can be explained by the fact that for time-consuming analysis of large graphs the data exchange between CPU and GPU for implementing high-performance parallel analysis on GPU is reasonable. Therefore, the implementation $MtxReach_{GPU}$ allows one to perform this analysis on selected graphs up to 19 times faster than the best considered existing solution, i.e. the *Graspan* tool. Another proposed GPU implementation $KronAll_{GPU}$ demonstrates longer running time since this implementation is not specialized for the CFPQ problem with the reachability query semantics and computes redundant information, which is sufficient to construct any number of paths found. If we consider only CPU implementations, the proposed implementation $MtxReach_{CPU}$ shows comparable results to the *Graspan* tool, and for the most time-consuming graphs (with numbers 6, 10, and 18) it achieves almost a 5 times faster analysis. This is an impressive result, taking into account that the *Graspan* tool specializes for this type of graph analysis. The running time of the GLL_R implementation is comparable to the running time of the $MtxReach_{CPU}$

Table 5 — The running time in seconds of the CFPQ algorithms with the reachability query semantics for the static program analysis [76]*

№	*#result*	*Graspan*	*LL*	GLL_R	*MRC*	*MRG*	*KAC*	*KAG*
6	92,806,768	2,619.1	8,390.4	OOT	536.7	**135.0**	6,165.0	OOM
7	5,339,563	49.8	928.2	130.8	119.9	**34.5**	307.1	96.7
8	5,351,409	51.3	924.9	113.0	123.9	**34.4**	311.7	96.8
9	5,428,237	52.4	935.4	128.8	122.1	**34.7**	314.2	98.0
10	18,825,025	330.2	3,660.7	371.2	279.4	**69.8**	1,381.5	OOM
11	9,646,475	95.4	2,000.8	167.7	105.7	**49.6**	533.1	148.4
12	3,783,769	39.2	644.4	87.2	45.8	**24.6**	215.9	68.7
13	5,249,389	55.3	898.5	109.4	79.5	**34.0**	301.3	95.6
14	16,747,731	161.7	OOM	614.0	378.1	**104.8**	978.8	292.9
15	5,276,303	52.9	900.2	111.1	121.8	**34.1**	300.7	96.0
16	3,990,3	39.1	671.3	77.9	84.1	**25.5**	226.6	71.8
17	8,833,403	95.2	1,851.0	160.6	206.3	**55.2**	684.7	176.1
18	90,661,446	1,711.9	OOM	OOT	969.9	**170.4**	5,072.0	OOM
19	5,593,387	56.4	942.7	115.8	181.7	**35.1**	320.7	99.2
20	6,085,269	58.9	968.8	120.1	133.6	**36.1**	339.5	103.9

* *#result* is the size of resulting set; *Graspan* is a static program analysis tool [76]; *LL* is the implementation of the algorithm [24] based on the LL parsing algorithm; GLL_R is the reachability implementation of the algorithm [26] based on the GLL parsing algorithm; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

implementation, but the former lacked 5 hours to analyze the two most time-consuming graphs with numbers 6 and 18. The longest running time was shown by the implementations *LL* and $KronAll_{CPU}$. However, the $KronAll_{CPU}$ implementation is not specialized in solving the CFPQ problem with the reachability query semantics. In this analysis, the computation of redundant information significantly affected the running time of the $KronAll_{CPU}$ implementation.

According to **[M2]**, we also compared the memory consumption of the discussed implementations. The results of this comparison are presented in Tables 6 and 7, where for each graph the lowest memory consumption is highlighted. For the RDF analysis, the *Graspan* tool shows the lowest memory consumption that can be seen in Table 6. For this analysis, the proposed implementations consume up to 6 times more memory than the *Graspan* tool, but significantly less memory than other existing solutions *LL* and GLL_R. For the pointer analysis in `C/C++` programs, the $MtxReach_{GPU}$ implementation of the proposed matrix-based algorithm has the lowest memory consumption as shown in Table 7. Also, the $MtxReach_{CPU}$ implementation consumes a comparable amount of memory to the *Graspan* tool. Thus, for this analysis, the proposed implementations reduce memory consumption by up to 2 times compared to the best considered existing solution. As expected, the proposed implementations $KronAll_{CPU}$ and $KronAll_{GPU}$ consume more memory than the *Graspan* tool due to computing redundant information to the CFPQ problem with the reachability query semantics. However, these implementations consume significantly less memory than other existing solutions *LL* and GLL_R.

Table 6 — The memory consumption in megabytes of the CFPQ algorithms with the reachability query semantics for the RDF analysis [5]*

№	*#result*	*Graspan*	*LL*	GLL_R	*MRC*	*MRG*	*KAC*	*KAG*
1	90,994	**101**	470	1,263	240	307	279	357
2	640,316	**193**	1,406	1,192	468	727	468	829
3	588,976	**62**	431	3,177	263	387	266	573
4	151,706	**1,498**	15,200	9,877	3,229	1,651	3,229	2,463
5	5,351,657	**1,098**	20,395	OOT	6,804	OOM	6,804	OOM

* *#result* is the size of resulting set; *Graspan* is a static program analysis tool [76]; *LL* is the implementation of the algorithm [24] based on the LL parsing algorithm; GLL_R is the reachability implementation of the algorithm [26] based on the GLL parsing algorithm; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

Thus, the answer to **[Q1]** is the following. The proposed implementations for the CFPQ problem with the reachability query semantics compared to the best considered existing solutions allow one to:

- speed up the RDF analysis time by up to 277 times, while increase memory consumption by up to 6 times;
- speed up the pointer analysis of `C/C++` programs by up to 19 times, while reduce memory consumption by up to 2 times.

[Q2]. To answer the second question, the CFPQ implementations for the single-path and all-path query semantics was compared using the RDF analysis and the

Table 7 — The memory consumption in megabytes of the CFPQ algorithms with the reachability query semantics for the static program analysis [76][*]

№	*#result*	*Graspan*	*LL*	GLL_R	*MRC*	*MRG*	*KAC*	*KAG*
6	92,806,768	11,094	53,652	OOT	11,619	**5,585**	40,110	OOM
7	5,339,563	1,643	33,275	30,573	1,342	**863**	2,954	2,209
8	5,351,409	1,638	33,082	29,866	1,331	**849**	2,988	2,219
9	5,428,237	1,660	33,472	29,317	1,305	**849**	3,052	2,235
10	18,825,025	3,474	46,060	49,262	3,181	**2,861**	9,012	OOM
11	9,646,475	2,330	41,429	31,128	1,958	**1,099**	4,779	3,723
12	3,783,769	1,170	23,478	21,537	1,015	**687**	2,205	1,649
13	5,249,389	1,620	32,405	29,205	1,268	**845**	2,922	2,173
14	16,747,731	5,083	OOM	41,449	3,466	**1,959**	8,261	5,763
15	5,276,303	1,624	32,414	29,432	1,299	**845**	2,980	2,271
16	3,990,305	1,219	24,338	86,190	1,036	**691**	2,328	1,697
17	8,833,403	2,271	40,185	31,358	1,888	**1,111**	4,680	4,069
18	90,661,446	11,871	OOM	OOT	11,018	**5,297**	36,812	OOM
19	5,593,387	1,688	33,689	30,228	1,336	**857**	3,067	2,309
20	6,085,269	1,763	34,324	31,699	1,454	**887**	3,308	2,415

[*] *#result* is the size of resulting set; *Graspan* is a static program analysis tool [76]; *LL* is the implementation of the algorithm [24] based on the LL parsing algorithm; GLL_R is the reachability implementation of the algorithm [26] based on the GLL parsing algorithm; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

pointer analysis of `C/C++` programs. The only existing considered implementation that solves CFPQ problem with these path query semantics is the GLL_A implementation of the algorithm based on the GLL parsing algorithm. According to **[M1]**, the average running time in seconds of the discussed implementations for the RDF analysis is presented in Table 8, and for the pointer analysis in `C/C++` programs — in Table 9. In addition, the smallest analysis time for each graph is highlighted.

The RDF analysis results presented in Table 8. The smallest analysis time was demonstrated by the implementations $MtxSingle_{CPU}$ (*MSC*) and $MtxAll_{CPU}$ (*MAC*) of the proposed matrix-based algorithm. For the RDF analysis, the proposed implementations allow one to solve the CFPQ problem with single-path query

Table 8 — The running time in seconds of the CFPQ algorithms with the single-path and all-path query semantics for the RDF analysis [5]*

№	*#result*	GLL_A	*MSC*	*MAC*	*KAC*	*KAG*
1	90,994	3.0	0.2	**0.1**	0.3	0.2
2	640,316	20.1	2.1	**0.5**	3.2	3.1
3	588,976	140.1	0.4	**0.2**	**0.2**	**0.2**
4	151,706	1,878.4	**3.0**	5.0	6.0	3.9
5	5,351,657	OOT	25.7	**11.0**	11.7	OOM

* *#result* is the size of resulting set; GLL_A is the implementation of the algorithm [26] for the all-path query semantics based on the GLL parsing algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

semantics up to 1636 times faster compared to the GLL_A implementation since the latter lacked to complete this analysis in 5 hours for the most time-consuming graph with number 5. The same conclusion is correct for the all-path query semantics, because the best analysis time for the graph with number 5 was shown by the implementation $MtxAll_{CPU}$ that solves the CFPQ problem with the all-path query semantics.

For the single-path query semantics, the smallest time of the pointer analysis of `C/C++` programs was demonstrated by the proposed implementations $MtxSingle_{CPU}$ and $KronAll_{GPU}$ as shown in Table 9. Such results can be explained by the facts that the $MtxSingle_{CPU}$ implementation is the only considered implementation specialized on the single-path query semantics, and the $KronAll_{GPU}$ implementation is the only GPU implementation in this comparison. For the all-path query semantics, the proposed implementations allow one to speed up the analysis time by up to 27 times compared to the existing solution GLL_A. This result is achieved using GPU implementation $KronAll_{textitGPU}$ to analyze the graph with number 14.

According to **[M2]**, we also compared the memory consumption of the discussed implementations for the single-path and all-path query semantics. The results of this comparison are presented in Tables 10 and 11, where the lowest memory consumption is highlighted. For the RDF analysis, the lowest memory consumption is demonstrated by the proposed $MtxAll_{CPU}$ implementation for both query semantics as can be seen in Table 10. The exception is the least time-

Table 9 — The running time in seconds of the CFPQ algorithms with the single-path and all-path query semantics for the static program analysis [76]*

№	*#result*	GLL_A	*MSC*	*MAC*	*KAC*	*KAG*
6	92,806,768	OOT	**1,611.5**	OOM	6,165.0	OOM
7	5,339,563	728.5	132.8	432.5	307.1	**96.7**
8	5,351,409	771.3	111.6	OOM	311.7	**96.8**
9	5,428,237	750.2	139.1	OOM	314.2	**98.0**
10	18,825,025	1,222.3	**699.1**	OOM	1,381.5	OOM
11	9,646,475	1,150.9	**135.6**	OOM	533.1	148.4
12	3,783,769	368.9	**53.4**	261.8	215.9	68.7
13	5,249,389	692.7	166.7	405.5	301.3	**95.6**
14	16,747,731	7,923.0	474.9	OOM	978.8	**292.9**
15	5,276,303	712.6	166.0	437.8	300.7	**96.0**
16	3,990,305	396.8	95.6	301.8	226.6	**71.8**
17	8,833,403	1,010.1	**145.8**	OOM	684.7	176.1
18	90,661,446	OOT	**2,024.0**	OOM	5,072.0	OOM
19	5,593,387	741.1	142.3	OOM	320.7	**99.2**
20	6,085,269	759.0	153.2	OOM	339.5	**103.9**

* *#result* is the size of resulting set; GLL_A is the implementation of the algorithm [26] for the all-path query semantics based on the GLL parsing algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

consuming graph with number 1. Thus, for the single-path and all-path query semantics, the proposed implementations consume for the RDF analysis up to 152 times less memory than the existing solution. Note that the proposed $MtxSingle_{CPU}$ implementation is specialized for solving the CFPQ problem with the single-path query semantics and consumes more memory than the $MtxAll_{CPU}$ implementation for the all-path query semantics. This can be explained by the fact that the $MtxAll_{CPU}$ implementation was written in `C++` with using its own `Python` wrapper, and the small number of distinct intermediate vertices in matrix elements from the *AllPathIndex* set can consume less memory than elements from the *PathIndex* set. Also, the proposed implementations consume comparable amount of memory to the existing GLL_A implementation for the pointer analysis of `C/C++` programs with both query semantics as shown in Table 11. However, for most of the discussed graphs the lowest memory consumption was shown by the proposed GPU implementation $KronAll_{GPU}$, while the implementation $MtxAll_{CPU}$ was terminated because of out of memory error.

Table 10 — The memory consumption in megabytes of the CFPQ algorithms with the single-path and all-path query semantics for the RDF analysis [5]*

№	*#result*	GLL_A	*MSC*	*MAC*	*KAC*	*KAG*
1	90,994	**49**	257	200	279	357
2	640,316	649	545	**337**	468	829
3	588,976	30,444	290	**200**	266	573
4	151,706	9,108	3,805	**1,595**	3,229	2,463
5	5,351,657	OOT	8,058	**2,720**	6,804	OOM

* *#result* is the size of resulting set; GLL_A is the implementation of the algorithm [26] for the all-path query semantics based on the GLL parsing algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

According to the obtained results, **[Q2]** can be answered as follows. The proposed CFPQ implementations for the single-path and all-path query semantics compared to the considered existing solution allow one to:

- speed up RDF analysis time by up to 1636 times, while reduce memory consumption by up to 152 times;
- speed up the pointer analysis of `C/C++` programs by up to 27 times, while consume comparable amount of memory.

[Q3]. To answer the third question, all proposed CFPQ implementations was compared. According to **[M1]**, the average running time in seconds of the proposed implementations for the RDF analysis is presented in Table 12, and for the pointer analysis of `C/C++` programs — in Table 13.

The proposed implementations demonstrate comparable RDF analysis time as shown in Table 12. Note that storing information about the found paths in the proposed implementations slows down the analysis time by up to 3 times for the single-path query semantics, and by up to 4 times for the all-path query semantics. In turn, the results of the pointers analysis of `C/C++` programs presented in Table 13 show that the $MtxReach_{GPU}$ implementation has the lowest running time compared

Table 11 — The memory consumption in megabytes of the CFPQ algorithms with the single-path and all-path query semantics for the static program analysis [76]*

№	*#result*	GLL_A	*MSC*	*MAC*	*KAC*	*KAG*
6	92,806,768	OOT	**35,666**	OOM	40,110	OOM
7	5,339,563	6,043	2,651	62,423	2,954	**2,209**
8	5,351,409	5,874	2,651	OOM	2,988	**2,219**
9	5,428,237	6,250	2,676	OOM	3,052	**2,235**
10	18,825,025	**4,608**	8,332	OOM	9,012	OOM
11	9,646,475	8,629	4,214	OOM	4,779	**3,723**
12	3,783,769	4,242	1,964	62,404	2,205	**1,649**
13	5,249,389	5,992	2,589	62,384	2,922	**2,173**
14	16,747,731	**5,136**	7,156	OOM	8,261	5,763
15	5,276,303	6,094	2,632	62,421	2,980	**2,271**
16	3,990,305	4,545	2,073	62,453	2,328	**1,697**
17	8,833,403	9,346	4,201	OOM	4,680	**4,069**
18	90,661,446	OOT	**32,635**	OOM	36,812	OOM
19	5,593,387	6,157	2,759	OOM	3,067	**2,309**
20	6,085,269	6,842	2,991	OOM	3,308	**2,415**

* *#result* is the size of resulting set; GLL_A is the implementation of the algorithm [26] for the all-path query semantics based on the GLL parsing algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

to other proposed implementations. For both query semantics, the implementation $KronAll_{GPU}$ shows the lowest running time in most cases and slows down the analysis time by up to 3 times compared to the best proposed reachability implementation. The exceptions are the most time-consuming graphs with numbers 6, 10, and 18. For these graphs the $KronAll_{GPU}$ implementation terminates due to out of memory error. For these graphs, the proposed implementations solved the CFPQ problem with the single-path query semantics up to 12 times longer than the CFPQ problem with the reachability query semantics, and the CFPQ problem with the all-path query semantics — up to 46 times longer.

Table 12 — The running time in seconds of the proposed CFPQ algorithms for the RDF analysis [5]*

№	*#result*	*MRC*	*MRG*	*MSC*	*MAC*	*KAC*	*KAG*
1	90,994	0.1	0.1	0.2	0.1	0.3	0.2
2	640,316	1.2	0.8	2.1	0.5	3.2	3.1
3	588,976	0.1	0.2	0.4	0.2	0.2	0.2
4	151,706	1.0	1.0	3.0	5.0	6.0	3.9
5	5,351,657	10.9	OOM	25.7	11.0	11.7	OOM

* *#result* is the size of resulting set; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

According to **[M2]**, we also compared the memory consumption of the proposed implementations. The results of this comparison are presented in Tables 14 and 15. For the RDF analysis, the $MtxAll_{CPU}$ implementation for the all-path query semantics showed the lowest memory consumption. Therefore, for this analysis on the selected graphs using the proposed implementations there is no increase in memory consumption when storing information about the found paths. At the same time, for the pointer analysis of `C/C++` programs, the $MtxReach_{GPU}$ implementation that solves the reachability problem showed the lowest memory consumption. For the single-path and all-path query semantics, the lowest memory consumption is shown by the $KronAll_{GPU}$ implementation in most cases. This implementation consume up to 4 times more memory compared to the best reachability implementation. The exceptions are the three most time-consuming graphs. For these graphs, the proposed implementations consumed up to 6 times more memory for the single-path query semantics than for the reachability query semantics, and up to 7 times more memory for the all-path query semantics.

Thus, **[Q3]** can be answered as follows. The overhead costs of storing information about the found paths in the obtained implementations compared to the obtained reachability implementations are:

- up to 4 times slower RDF analysis;

Table 13 — The running time in seconds of the proposed CFPQ algorithms for the static program analysis [76][*]

№	*#result*	*MRC*	*MRG*	*MSC*	*MAC*	*KAC*	*KAG*
6	92,806,768	536.7	135.0	1,611.5	OOM	6,165.0	OOM
7	5,339,563	119.9	34.5	132.8	432.5	307.1	96.7
8	5,351,409	123.9	34.4	111.6	OOM	311.7	96.8
9	5,428,237	122.1	34.7	139.1	OOM	314.2	98.0
10	18,825,025	279.4	69.8	699.1	OOM	1,381.5	OOM
11	9,646,475	105.7	49.6	135.6	OOM	533.1	148.4
12	3,783,769	45.8	24.6	53.4	261.8	215.9	68.7
13	5,249,389	79.5	34.0	166.7	405.5	301.3	95.6
14	16,747,731	378.1	104.8	474.9	OOM	978.8	292.9
15	5,276,303	121.8	34.1	166.0	437.8	300.7	96.0
16	3,990,305	84.1	25.5	95.6	301.8	226.6	71.8
17	8,833,403	206.3	55.1	145.8	OOM	684.7	176.1
18	90,661,446	969.9	170.4	2,024.0	OOM	5,072.0	OOM
19	5,593,387	181.7	35.1	142.3	OOM	320.7	99.2
20	6,085,269	133.6	36.1	153.2	OOM	339.5	103.9

[*] *#result* is the size of resulting set; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

- up to 46 times slower pointer analysis of `C/C++` programs with up to 7 times higher memory consumption.

[Q4]. To answer the last question, the proposed implementations for the Kronecker product-based algorithm that do not require a transformation of the input CFG was compared to the proposed implementations of the matrix-based algorithm. The results presented in Tables 12, 13, 14, and 15 can also be used for this comparison. There are no specialized implementations of the Kronecker product-based algorithm for CFPQ problem with the reachability and single-path query semantics. Thus, we compare only implementations that solve the CFPQ problem with the all-path query semantics. In addition, the matrix-based CFPQ algorithm for the

Table 14 — The memory consumption in megabytes of the proposed CFPQ algorithms for the RDF analysis [5]*

№	*#result*	*MRC*	*MRG*	*MSC*	*MAC*	*KAC*	*KAG*
1	90,994	240	307	257	200	279	357
2	640,316	468	727	545	337	468	829
3	588,976	263	387	290	200	266	573
4	151,706	3,229	1,651	3,805	1,595	3,229	2,463
5	5,351,657	6,804	OOM	8,058	2,720	6,804	OOM

* *#result* is the size of resulting set; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

all-path query semantics is implemented only on CPU. Therefore, we compare only CPU implementations. Thus, for this comparison the $MtxAll_{CPU}$ and $KronAll_{CPU}$ implementations were chosen.

In Tables 12 and 14 one can see that despite the increase in the grammar size after its transformation into the WCNF, the RDF analysis time of the $MtxAll_{CPU}$ implementation is not slower than the one of the $KronAll_{CPU}$ implementation that does not require a transformation of the input CFG. Also, the $MtxAll_{CPU}$ consumes up to 2 times less memory than the $KronAll_{CPU}$ implementation. The reason for this is the relatively small grammar size for this analysis, as well as the relatively small analysis complexity on the discussed graphs. However, other conclusions we can made for more complex pointer analysis of `C/C++` programs. In Tables 13 and 15 one can see that the $MtxAll_{CPU}$ implementation is out of memory for most graphs, while for other graphs the $KronAll_{CPU}$ implementation completes analysis up to one and a half times faster and consumes up to 28 times less memory. The grammar transformation has led to the significant increase in the number of nonterminals and derivation rules of the grammar. Thus, the matrix-based algorithm must store a bigger number of Boolean matrices with large number of nonzero elements, and it must compute more matrix operations. Therefore, for such time-consuming analysis, this had a significant impact on the performance of the $MtxAll_{CPU}$ implementation.

Table 15 — The memory consumption in megabytes of the proposed CFPQ algorithms for the static program analysis [76]*

№	*#result*	*MRC*	*MRG*	*MSC*	*MAC*	*KAC*	*KAG*
6	92,806,768	11,619	5,585	35,666	OOM	40,110	OOM
7	5,339,563	1,342	863	2,651	62,423	2,954	2,209
8	5,351,409	1,331	849	2,651	OOM	2,988	2,219
9	5,428,237	1,305	849	2,676	OOM	3,052	2,235
10	18,825,025	3,181	2,861	8,332	OOM	9,012	OOM
11	9,646,475	1,958	1,099	4,214	OOM	4,779	3,723
12	3,783,769	1,015	687	1,964	62,404	2,205	1,649
13	5,249,389	1,268	845	2,589	62,384	2,922	2,173
14	16,747,731	3,466	1,959	7,156	OOM	8,261	5,763
15	5,276,303	1,299	845	2,632	62,421	2,980	2,271
16	3,990,305	1,036	691	2,073	62,453	2,328	1,697
17	8,833,403	1,888	1,111	4,201	OOM	4,680	4,069
18	90,661,446	11,018	5,297	32,635	OOM	36,812	OOM
19	5,593,387	1,336	857	2,759	OOM	3,067	2,309
20	6,085,269	1,454	887	2,991	OOM	3,308	2,415

* *#result* is the size of resulting set; *MRC* and *MRG* are CPU and GPU reachability implementations of the proposed matrix-based algorithm; *MSC* and *MAC* are implementations of the proposed matrix-based algorithm for the single-path and all-path query semantics; *KAC* and *KAG* are CPU and GPU implementations of the proposed Kronecker product-based algorithm for the all-path query semantics.

Thus, **[Q4]** can be answered as follows. The proposed CPU implementation that does not require a transformation of the input CFG compared to the proposed CPU implementation of the matrix-based algorithm for the all-path query semantics:

- does not speed up the RDF analysis consuming up to 2 times more memory due to the relatively small grammar size, as well as the relatively small analysis complexity;
- speeds up the pointer analysis of `C/C++` programs by up to one and a half times consuming up to 28 times less memory.

Summary. Thus, according to the obtained results, we can conclude the following. The proposed approach to the CFPQ allows one to obtain high-performance parallel implementations that speed up the graph analysis time and consume less memory compared to the existing solutions on real data.

5.3 Limitations

The proposed approach and obtained CFPQ algorithms impose some limitations on their implementations. This section is devoted to a discussion of these limitations.

Firstly, to apply the obtained implementations it is required to formulate the needed graph analysis as a CFPQ problem. For this purpose, we must describe the path constraints using the appropriate CFL for this analysis. Note that some constraints cannot be described using the CFLs. For those constraints that can be expressed in the form of a CFL it is necessary to describe this language in the form required by the obtained algorithm. The algorithms proposed in this thesis use CFGs and recursive automata to represent CFLs. And for the matrix-based algorithm such grammars must be transformed into the WCNF. Such transformations can lead to the significant grammar size increase that will significantly affect the resulting implementation performance.

Secondly, the performance of the proposed implementations strongly depends on the input graph sparsity. When analyzing dense large graphs, the proposed implementations will either show unsatisfactory running time, or they will consume too much memory. Also, the proposed implementations are inefficient for small graphs and for simple analysis. In these cases the time spent on the organization of parallel computations exceeds, or is comparable to the analysis time itself.

Besides, after constructing a CFPQ algorithm using the proposed approach it is necessary to find a linear algebra library with implemented necessary operations, or to implement them independently. In the case of using standard matrix operations (matrix addition, multiplication, transposition, etc.) with standard data types (Boolean, integer, floating-point, etc.) such a library is easy to find. However, if the algorithm uses linear algebra objects with some custom data type for more complex graph analysis one may not find a suitable library. For using the GraphBLAS implementations, one need to make sure that the custom data type used as matrix and vector elements, and the operations on them, can be implemented using an algebraic structure similar to the semiring that the GraphBLAS standard allows.

Chapter 6. Comparison and Relation

In this chapter, we present a comparison and relation of the obtained results to the main existing CFPQ solutions. A description of the existing solutions is presented in section 1.5.

We compare the following existing CFPQ tools: the *LL* implementation of the CFPQ algorithm [24] based on the LL(1) parsing algorithm; the implementations GLL_R and GLL_A [26] of the CFPQ algorithm based on the GLL parsing algorithm; the *Graspan* [76] static program analysis tool. The comparison was made with the proposed matrix-based implementations $MtxReach_{CPU}$, $MtxReach_{GPU}$, $MtxSingle_{CPU}$, and $MtxAll_{CPU}$, as well as with the proposed Kronecker product-based implementations $KronAll_{CPU}$ and $KronAll_{GPU}$. The comparison criteria presented in Table 16.

Table 16 — Criteria for comparing CFPQ tools

Criteria	The column name in Table 17	Description
Does not require a transformation of the input CFG	Without transformations	Is there no need to transform the input CFG, for example into WCNF, in order to apply the algorithm and the corresponding tool?
Path extraction	Path extraction	Does the algorithm and the corresponding tool compute information sufficient to construct found paths corresponding to the input context-free constraints?
GPU usage	GPU	Does the tool make computations on GPU?
The use of linear algebra	Linear algebra	Are the algorithm and the corresponding tool linear algebra based?

The main results of the comparison are presented in Table 17. Thus, we can conclude the following.

- In this work, we proposed the first linear algebra based CFPQ algorithms for the arbitrary context-free path constraints.
- Existing tools are mainly implemented on CPU.
- In this work, we proposed the first GPU implementation of a CFPQ algorithm for the single-path and all-path query semantics.

Table 17 — The CFPQ tool comparison

A tool	Without transformations	Path extraction	GPU	Linear algebra
LL	+	−	−	−
GLL_R	+	−	−	−
GLL_A	+	+*	−	−
Graspan	−**	−	+***	−
$MtxReach_{CPU}$	−	−	−	+
$MtxReach_{GPU}$	−	−	+	+
$MtxSingle_{CPU}$	−	+	−****	+
$MtxAll_{CPU}$	−	+	−****	+
$KronAll_{CPU}$	+	+	−	+
$KronAll_{GPU}$	+	+	+	+

* The implementation of the algorithm [26] based on the GLL parsing algorithm, builds an SPPF that contains information sufficient to construct all found paths.

** The *Graspan* tool [76] requires to transform the input CFG into the WCNF.

*** There is also a GPU implementation of the *Graspan* tool [76] that the author failed to run. This implementation allows to speed up the pointer analysis of `C/C++` programs from 3.5 to 650 times.

**** The proposed matrix-based CFPQ algorithm was implemented on GPU only for the reachability query semantics. However, for the matrix-based CFPQ algoritm with the single-path and all-path query semantics it may be possible to obtain a high-performance GPU implementation using the CUSP library and the GraphBLAST library.

Conclusion

The main results submitted for defense.

1. An approach to the context-free path querying based on linear algebra methods that allows one to use theoretical and practical linear algebra results was developed.
2. A CFPQ algorithm that uses the proposed approach was devised. Termination and correctness of the devised algorithm were proved, as well as its time complexity. The proposed algorithm uses matrix operations, which make it possible to apply a wide class of optimizations and allows one to automatically parallelize computations using existing linear algebra libraries.
3. A CFPQ algorithm that uses the proposed approach and does not require a transformation of the input context-free grammar was devised. Termination and correctness of the devised algorithm were proved, as well as its time complexity. The proposed algorithm makes it possible to work with arbitrary input context-free grammars without any transformations. This allows one to avoid a significant increase in the grammar size that affects analysis performance.
4. The devised algorithms are implemented using parallel computing techniques. An experimental study of the devised algorithms was provided using real RDF data and graphs built for static program analysis. The obtained implementations were compared with each other, with existing solutions from the field of static program analysis, and with solutions based on various parsing techniques. The comparison results show that the proposed implementations for the reachability problem allow one to obtain up to 2 orders of magnitude faster graph analysis time and consume up to 2 times less memory in comparison with existing solutions, and for the problems of finding one and all paths in a graph allow one to speed up the analysis time up to 3 orders of magnitude and consume up to 2 orders of magnitude less memory.

CFPQ_PyAlgo platform for developing, testing, and benchmarking CFPQ algorithm was created using the obtained implementations.

We give the following **application recommendations for the work results**. The developed approach and the obtained algorithms are applicable for the CFPQ using linear algebra. Also, provided approach allows one to obtain high-performance parallel CFPQ implementations that are compact and portable. These implementations can be created using existing linear algebra libraries. The *CFPQ_PyAlgo* platform can be used in static program analysis [1; 2], RDF analysis [5], bioinformatics [3], etc. In addition, CFPQ implementations can be integrated with graph databases such as RedisGraph. In work [28], we provided such prototype implementations. However, to provide full integration it is necessary to extend Cypher graph query language used in RedisGraph and to support syntax for specification of context-free path constraints. Moreover, there is a proposal[1] that describes such syntax extension.

Also, we identify **prospects for further development of the topic**. First of all, the proposed approach and developed algorithms can be applied to creation of a specialized tool for a specific graph analysis problem. The CFPQ algorithms for graphs of a certain type and for specific path constraints can be created using the proposed approach. For example, for a static program analysis the structure of graphs derived from programs in a particular programming language can be taken into account, as well as the properties of a particular CFL for the chosen analysis (alias analysis [2], taint analysis [77], etc.).

In practice, CFPQ problem is rarely solved without fixing a relatively small set of possible source and destination vertices. Thus, often the information about paths between any vertices is redundant. Therefore, another direction of further research is modification of all proposed CFPQ algorithms with additional restrictions on sets of source and destination vertices.

In this work, the proposed matrix-based CFPQ algorithm for the single-path and all-path query semantics was implemented only on CPU. For these query semantics, we use more complex data types for matrix elements. This leads to a significant increase in memory consumption and does not allow us to obtain a high-performance GPU implementation. Therefore, further research is needed to optimize this algorithm and the algebraic structures used. In the future, it may be possible to

[1] A proposal with path pattern syntax for openCypher:
https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc
(date of access: 14.01.2022).

obtain high-performance GPU implementations of this algorithm using the CUSP library and the GraphBLAST library.

In addition, there are some graph analysis problems that cannot be expressed using the context-free path constraints. For example, the context-sensitive data-dependence program analysis [78] uses an interleaved matched-parenthesis language that is not context-free. This problem is well-known to be undecidable [78]. However, path constraints in the form of linear conjunctive languages [79] that belong to a wider class of languages than context-free ones, can be used to approximate the result of such analysis. Thus, the relevant research direction is to extend the proposed approach to solve path querying problems with path constraints expressed by languages from a broader class of languages than the CFLs.

References

1. *Rehof, J.* Type-base flow analysis: from polymorphic subtyping to CFL-reachability [Text] / J. Rehof, M. Fähndrich // ACM SIGPLAN Notices. — 2001. — Vol. 36, no. 3. — P. 54—66.
2. *Zheng, X.* Demand-driven alias analysis for C [Text] / X. Zheng, R. Rugina // Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 2008. — P. 197—208.
3. *Sevon, P.* Subgraph queries by context-free grammars [Text] / P. Sevon, L. Eronen // Journal of Integrative Bioinformatics. — 2008. — Vol. 5, no. 2. — P. 157—172.
4. *Truong, Q.-D.* Graph Methods for Social Network Analysis [Text] / Q.-D. Truong, T. Dkaki, Q.-B. Truong //. Vol. 168. — 03/2016. — P. 276—286.
5. Context-free path queries on RDF graphs [Text] / X. Zhang [et al.] // International Semantic Web Conference. — Springer. 2016. — P. 632—648.
6. *Barrett, C.* Formal-language-constrained path problems [Text] / C. Barrett, R. Jacob, M. Marathe // SIAM Journal on Computing. — 2000. — Vol. 30, no. 3. — P. 809—837.
7. *Reps, T.* Program analysis via graph reachability [Text] / T. Reps // Information and software technology. — 1998. — Vol. 40, no. 11/12. — P. 701—726.
8. *Bradford, P. G.* Efficient exact paths for dyck and semi-dyck labeled path reachability [Text] / P. G. Bradford // 2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON). — IEEE. 2017. — P. 247—253.
9. *Hellings, J.* Conjunctive Context-Free Path Queries. [Text] / J. Hellings // ICDT. — 2014. — P. 119—130.
10. *Hellings, J.* Explaining Results of Path Queries on Graphs [Text] / J. Hellings // Software Foundations for Data Interoperability and Large Scale Graph Data Analytics. — Springer, 2020. — P. 84—98.

11. *Koschmieder, A.* Regular path queries on large graphs [Text] / A. Koschmieder, U. Leser // International Conference on Scientific and Statistical Database Management. — Springer. 2012. — P. 177—194.

12. Answering regular path queries using views [Text] / D. Calvanese [et al.] // Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073). — IEEE. 2000. — P. 389—398.

13. *Kepner, J.* Graph algorithms in the language of linear algebra [Text] / J. Kepner, J. Gilbert. — SIAM, 2011.

14. An experimental study of context-free path query evaluation methods [Text] / J. Kuijpers [et al.] // Proceedings of the 31st International Conference on Scientific and Statistical Database Management. — 2019. — P. 121—132.

15. *Buluç, A.* Parallel breadth-first search on distributed memory systems [Text] / A. Buluç, K. Madduri // Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. — 2011. — P. 1—12.

16. Delta-stepping SSSP: From vertices and edges to GraphBLAS implementations [Text] / U. Sridhar [et al.] // 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — IEEE. 2019. — P. 241—250.

17. GBTL-CUDA: Graph algorithms and primitives for GPUs [Text] / P. Zhang [et al.] // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — IEEE. 2016. — P. 912—920.

18. *Azad, A.* Parallel triangle counting and enumeration using matrix algebra [Text] / A. Azad, A. Buluç, J. Gilbert // 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. — IEEE. 2015. — P. 804—811.

19. A comparative study on exact triangle counting algorithms on the gpu [Text] / L. Wang [et al.] // Proceedings of the ACM Workshop on High Performance Graph Processing. — 2016. — P. 1—8.

20. Mathematical foundations of the GraphBLAS [Text] / J. Kepner [et al.] // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — IEEE. 2016. — P. 1—9.

21. *Spampinato, D. G.* Linear algebraic depth-first search [Text] / D. G. Spampinato, U. Sridhar, T. M. Low // Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. — 2019. — P. 93—104.

22. *Valiant, L. G.* General context-free recognition in less than cubic time [Text] / L. G. Valiant // Journal of computer and system sciences. — 1975. — Vol. 10, no. 2. — P. 308—315.

23. *Yannakakis, M.* Graph-theoretic methods in database theory [Text] / M. Yannakakis // Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. — 1990. — P. 230—242.

24. *Medeiros, C. M.* Efficient evaluation of context-free path queries for graph databases [Text] / C. M. Medeiros, M. A. Musicante, U. S. Costa // Proceedings of the 33rd Annual ACM Symposium on Applied Computing. — 2018. — P. 1230—1237.

25. *Santos, F. C.* A bottom-up algorithm for answering context-free path queries in graph databases [Text] / F. C. Santos, U. S. Costa, M. A. Musicante // International Conference on Web Engineering. — Springer. 2018. — P. 225—233.

26. *Grigorev, S.* Context-free path querying with structural representation of result [Text] / S. Grigorev, A. Ragozina // Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia. — 2017. — P. 1—7.

27. *Azimov, R.* Context-free path querying by matrix multiplication [Text] / R. Azimov, S. Grigorev // Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — 2018. — P. 1—10.

28. Context-free path querying with single-path semantics by matrix multiplication [Text] / A. Terekhov [et al.] // Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — 2020. — P. 1—12.

29. Context-free path querying by kronecker product [Text] / E. Orachev [et al.] // European Conference on Advances in Databases and Information Systems. — Springer. 2020. — P. 49—59.

30. *Azimov, R.* Context-free path querying with all-path semantics by matrix multiplication [Text] / R. Azimov, I. Epelbaum, S. Grigorev // Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — 2021. — P. 1—7.

31. *Azimov, R.* Context-Free Path Querying In Terms of Linear Algebra [Text] / R. Azimov // Proceedings of the VLDB 2021 PhD Workshop (VLDB-PhD'21). — 2021.

32. *Azimov, R.* Path Querying with Conjunctive Grammars by Matrix Multiplication [Text] / R. Azimov, S. Grigorev // Programming and Computer Software. — 2019. — Vol. 45, no. 7. — P. 357—364.

33. *Azimov, R.* Path Querying Using Conjunctive Grammars [Text] / R. Azimov, S. Grigorev // Proceedings of the Institute for System Programming of the RAS. — 2018. — Vol. 30, no. 2. — 149—166 (In Russ.)

34. *Azimov, R.* Context-free path querying with all-path semantics using matrices with sets of intermediate vertices [Text] / R. Azimov, S. Grigorev // Scientific and Technical Journal of Information Technologies, Mechanics and Optics. — 2021. — Vol. 21, no. 4. — 499—505 (In Russ.)

35. *Aho, A. V.* The theory of parsing, translation, and compiling [Text]. Vol. 1 / A. V. Aho, J. D. Ullman. — Prentice-Hall Englewood Cliffs, NJ, 1973.

36. *Hopcroft, J. E.* Introduction to automata theory, languages, and computation [Text] / J. E. Hopcroft, R. Motwani, J. D. Ullman // Acm Sigact News. — 2001. — Vol. 32, no. 1. — P. 60—65.

37. *Rivera, F. F.* Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28–September 1, 2017, Proceedings [Text]. Vol. 10417 / F. F. Rivera, T. F. Pena, J. C. Cabaleiro. — Springer, 2017.

38. *Chomsky, N.* On certain formal properties of grammars [Text] / N. Chomsky // Information and control. — 1959. — Vol. 2, no. 2. — P. 137—167.

39. Analysis of recursive state machines [Text] / R. Alur [et al.] // ACM Transactions on Programming Languages and Systems (TOPLAS). — 2005. — Vol. 27, no. 4. — P. 786—818.

40. *Abiteboul, S.* Foundations of databases [Text]. Vol. 8 / S. Abiteboul, R. Hull, V. Vianu. — Addison-Wesley Reading, 1995.

41. *Verbitskaia, E.* Relaxed parsing of regular approximations of string-embedded languages [Text] / E. Verbitskaia, S. Grigorev, D. Avdyukhin // International Andrei Ershov Memorial Conference on Perspectives of System Informatics. — Springer. 2015. — P. 291—302.

42. Parser combinators for context-free path querying [Text] / E. Verbitskaia [et al.] // Proceedings of the 9th ACM SIGPLAN International Symposium on Scala. — 2018. — P. 13—23.

43. *Scott, E.* GLL parsing [Text] / E. Scott, A. Johnstone // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177—189.

44. *Scott, E.* BRNGLR: a cubic Tomita-style GLR parsing algorithm [Text] / E. Scott, A. Johnstone, R. Economopoulos // Acta informatica. — 2007. — Vol. 44, no. 6. — P. 427—461.

45. *Tomita, M.* LR parsers for natural languages [Text] / M. Tomita // 10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics. — 1984. — P. 354—357.

46. *Hellings, J.* Querying for paths in graphs using context-free path queries [Text] / J. Hellings // arXiv preprint arXiv:1502.02242. — 2015.

47. *Horwitz, S.* Demand interprocedural dataflow analysis [Text] / S. Horwitz, T. Reps, M. Sagiv // ACM SIGSOFT Software Engineering Notes. — 1995. — Vol. 20, no. 4. — P. 104—115.

48. *Reps, T.* Precise interprocedural dataflow analysis via graph reachability [Text] / T. Reps, S. Horwitz, M. Sagiv // Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1995. — P. 49—61.

49. *Chatterjee, K.* Optimal Dyck reachability for data-dependence and alias analysis [Text] / K. Chatterjee, B. Choudhary, A. Pavlogiannis // Proceedings of the ACM on Programming Languages. — 2017. — Vol. 2, POPL. — P. 1—30.

50. *Dietrich, J.* Giga-scale exhaustive points-to analysis for java in under a minute [Text] / J. Dietrich, N. Hollingum, B. Scholz // Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. — 2015. — P. 535—551.

51. An incremental points-to analysis with CFL-reachability [Text] / Y. Lu [et al.] // International Conference on Compiler Construction. — Springer. 2013. — P. 61—81.

52. Demand-driven points-to analysis for Java [Text] / M. Sridharan [et al.] // ACM SIGPLAN Notices. — 2005. — Vol. 40, no. 10. — P. 59—76.

53. *Yan*, *D.* Demand-driven context-sensitive alias analysis for Java [Text] / D. Yan, G. Xu, A. Rountev // Proceedings of the 2011 International Symposium on Software Testing and Analysis. — 2011. — P. 155—165.

54. *Milanova*, *A.* CFL-reachability and context-sensitive integrity types [Text] / A. Milanova, W. Huang, Y. Dong // Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools. — 2014. — P. 99—109.

55. *Miao*, *H.* Understanding data science lifecycle provenance via graph segmentation and summarization [Text] / H. Miao, A. Deshpande // 2019 IEEE 35th International Conference on Data Engineering (ICDE). — IEEE. 2019. — P. 1710—1713.

56. *Bellman*, *R.* On a routing problem [Text] / R. Bellman // Quarterly of applied mathematics. — 1958. — Vol. 16, no. 1. — P. 87—90.

57. *Ford*, *L. R.* Flows in networks [Text] / L. R. Ford, D. R. Fulkerson // Flows in Networks. — Princeton university press, 2015.

58. *Rote*, *G.* Path problems in graphs [Text] / G. Rote // Computational graph theory. — Springer, 1990. — P. 155—189.

59. *Rote*, *G.* A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion) [Text] / G. Rote // Computing. — 1985. — Vol. 34, no. 3. — P. 191—219.

60. *Tarjan*, *R. E.* Fast algorithms for solving path problems [Text] / R. E. Tarjan // Journal of the ACM (JACM). — 1981. — Vol. 28, no. 3. — P. 594—614.

61. *Tarjan*, *R. E.* A unified approach to path problems [Text] / R. E. Tarjan // Journal of the ACM (JACM). — 1981. — Vol. 28, no. 3. — P. 577—593.

62. *Fletcher*, *J. G.* A more general algorithm for computing closed semiring costs between vertices of a directed graph [Text] / J. G. Fletcher // Communications of the ACM. — 1980. — Vol. 23, no. 6. — P. 350—351.

63. *Floyd, R. W.* Algorithm 97: shortest path [Text] / R. W. Floyd // Communications of the ACM. — 1962. — Vol. 5, no. 6. — P. 345.

64. *Warshall, S.* A theorem on boolean matrices [Text] / S. Warshall // Journal of the ACM (JACM). — 1962. — Vol. 9, no. 1. — P. 11—12.

65. *Aho, A. V.* The design and analysis of computer algorithms [Text] / A. V. Aho, J. E. Hopcroft. — Pearson Education India, 1974.

66. Introduction to algorithms [Text] / T. H. Cormen [et al.]. — MIT press, 2009.

67. *Davis, T. A.* Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra [Text] / T. A. Davis // ACM Transactions on Mathematical Software (TOMS). — 2019. — Vol. 45, no. 4. — P. 1—25.

68. *Davis, T. A.* Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss [Text] / T. A. Davis // 2018 IEEE High Performance extreme Computing Conference (HPEC). — IEEE. 2018. — P. 1—6.

69. *Davis, T.* Algorithm 9xx: SuiteSparse: GraphBLAS: graph algorithms in the language of sparse linear algebra [Text] / T. Davis // Submitted to ACM TOMS. — 2018.

70. *Yang, C.* GraphBLAST: A high-performance linear algebra-based graph framework on the GPU [Text] / C. Yang, A. Buluç, J. D. Owens // ACM Transactions on Mathematical Software (TOMS). — 2022. — Vol. 48, no. 1. — P. 1—51.

71. *Baras, J. S.* Path problems in networks [Text] / J. S. Baras, G. Theodorakopoulos // Synthesis Lectures on Communication Networks. — 2010. — Vol. 3, no. 1. — P. 1—77.

72. *Chen, G.-H.* On the parallel computation of the algebraic path problem [Text] / G.-H. Chen, B.-F. Wang, C.-J. Lu // IEEE Transactions on Parallel & Distributed Systems. — 1992. — Vol. 3, no. 02. — P. 251—256.

73. *Lengauer, T.* Unstructured path problems and the making of semirings [Text] / T. Lengauer, D. Theune // Workshop on Algorithms and Data Structures. — Springer. 1991. — P. 189—200.

74. Gaussian elimination is not optimal [Text] / V. Strassen [et al.] // Numerische mathematik. — 1969. — Vol. 13, no. 4. — P. 354—356.

75. *Ibaraki*, *T.* On-line computation of transitive closures of graphs [Text] / T. Ibaraki, N. Katoh // Information Processing Letters. — 1983. — Vol. 16, no. 2. — P. 95—97.

76. Systemizing Interprocedural Static Analysis of Large-Scale Systems Code with Graspan [Text] / Z. Zuo [и др.] // ACM Trans. Comput. Syst. – New York, NY, USA, 2021. – Июль. – Т. 38, № 1/2. – URL: https://doi.org/10.1145/3466820.

77. Scalable and precise taint analysis for android [Text] / W. Huang [et al.] // Proceedings of the 2015 International Symposium on Software Testing and Analysis. — 2015. — P. 106—117.

78. *Zhang*, *Q.* Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability [Text] / Q. Zhang, Z. Su // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. — Paris, France : Association for Computing Machinery, 2017. — P. 344—358. — (POPL 2017). — URL: https://doi.org/10.1145/3009837.3009848.

79. *Okhotin*, *A.* Conjunctive grammars [Text] / A. Okhotin // Journal of Automata, Languages and Combinatorics. — 2001. — Vol. 6, no. 4. — P. 519—535.

List of Figures

List of Tables