Improvements on graph path queries : expression, evaluation, and minimum-weight satisfiability

Ciro Morais Medeiros

▶ To cite this version:

Ciro Morais Medeiros. Improvements on graph path queries : expression, evaluation, and minimum-weight satisfiability. Computer science. Université d'Orléans; Universidade federal do Rio Grande do Norte (Natal, Brésil), 2022. English. NNT : 2022ORLE1038 . tel-04186027

HAL Id: tel-04186027

https://theses.hal.science/tel-04186027

Submitted on 23 Aug 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LIFO - Laboratoire d'Informatique Fondamentale d'Orléans / DIMAp - Departamento de Informática e Matemática Aplicada

THÈSE EN COTUTELLE INTERNATIONALE présentée par :

Ciro MORAIS MEDEIROS

soutenue le : **30 août 2022**

pour obtenir le grade de : **Docteur de l'Université d'Orléans et de l'Universidade Federal do Rio Grande do Norte**

Discipline/Spécialité : **Informatique**

Improvements on Graph Path Queries: Expression, Evaluation, and Minimum-Weight Satisfiability

THÈSE dirigée par :

M. MUSICANTE Martin	Professeur, UFRN
Mme HALFELD-FERRARI Mirian	Professeur, UO

RAPPORTEURS :

Mme HARA Carmem	Professeur, UFPR
Mme REYES Nora	Professeur, UNSL

JURY :

M. TRAVERS Nicolas	Professeur, ESILV, Président du jury
M. LIEDLOFF Mathieu	Maître de conférences-HDR, UO
M. TOUMANI Farouk	Professeur, UBP - Clermont-Ferrand II
Mme GOLDBARG Elizabeth	Professeur, UFRN
M. EICHLER Cédric	Maître de conférences, INSA, Invité

To my parents

Acknowledgements

This work was carried out with the support of many people, some of whom I mention below.

First of all, I express my sincere gratitude to my supervisors Martin Musicante and Mirian Halfeld Ferrari for their support and trust placed in me in the most difficult moments.

I would like to thank Carmem Hara, Nora Reyes, Nicolas Travers, Mathieu Liedloff, Farouk Toumani, Elizabeth Goldbarg, and Cédric Eichler, members of the jury, for their insightful comments. I include Umberto Costa in this thanks.

I would like to thank the staff of the institutions involved, including the Programa de Pós-Graduação em Sistemas e Computação, the École Doctorale "Mathématiques, Informatique, Physique Théorique et Ingénierie des Systèmes", and the Laboratoire d'Informatique Fondamentale d'Orléans for the warm welcoming and the infrastructure offered. In addition to these, I would also like to thank the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior for funding this research.

I am thankful for the support from my family, especially my aunt Grinaura this past year. And I extend this thanks to my close friends, whose names it is unnecessary to list here, and to Beatriz, for her support.

Finally, I thank my parents Maria do Carmo and João Zacarias for their unconditional love and dedication. Before I can claim any merit, it is firstly yours.

List of Figures

2.1 Example Graph . . . 4
2.2 Annotated graph D' corresponding to the transitive closure of G and D . . . 5
2.3 Automaton that recognizes the language given by `abb*` . . . 6
2.4 Product-automaton that recognizes regular paths in D that belong to the language of `abb*` 6

3.1 Summary of non-regular expressions . . . 8

4.1 Example of hierarchy database D [39]. . . . 18

5.1 Example Graph (reproduction of Figure 2.1). . . . 24
5.2 Result graph for the query of Example 1. . . . 25
5.3 Experiments with ab-complete graphs and grammars G_1, G_2, G_3 and G_4. . . . 30
5.4 Experiments with ab-linear graphs using grammars G_1, G_2, G_3, G_4. . . . 31
5.5 Experiments with a-complete graphs and grammars G_5 and G_6. . . . 32
5.6 Experiments with a-cycle graphs grammars G_5 and G_6. . . . 33
5.7 Experiments with a-path graphs grammars G_5 and G_6. . . . 33
5.8 Experiment with synthetic social networks and grammar G_7 [29]. . . . 35
5.9 Experiment with synthetic social networks and grammar G_8 [29]. . . . 36
5.10 RDF representation of the mutation of species s_i into species s_j. . . . 40
5.11 Experiments with phylogenetic trees . . . 41
5.12 Experiments with a single start vertex over linear graphs . . . 42

6.1 Example execution of the RGM algorithm . . . 46
6.2 Example execution of the CFGM algorithm . . . 49
6.3 Experiments with complete graphs and alphabetical-sort (-a), implementation-dependant (-i), and random (-r) behaviors . . . 52
6.4 Example fractal triangles . . . 53
6.5 Experiments with fractal triangles and alphabetical-sort (-a), implementation-dependant (-i), and random (-r) behaviors . . . 54

List of Tables

3.1 Summary of path query evaluation algorithms 10

5.1 Step-by-step behavior of Algorithm 1 (underlined vertices mean they were either marked or added on that step). 26
5.2 Implementations used in the experiments. 28
5.3 Summary of grammars used for the experiments with complete and linear graphs. 29
5.4 Dense and Sparse grammars 30
5.5 Grammars for querying a simulated social network. 34
5.6 Summary of grammars used for the experiments with RDF Ontologies. 34
5.7 Performance evaluation on RDF databases with grammars G_9 and G_{10} *(Time in ms, Memory in Mb).* 37
5.8 Performance evaluation on RDF databases with grammars G_{11} and G_{12} *(Time in ms, Memory in Mb).* 38
5.9 Summary of grammars used for the experiments with synthetic phylogenetic trees. 40

List of Abbreviations and Acronyms

APMWS All-Positive Minimum Weighted Boolean Satisfiability

CFG Context-Free Grammar

CFPQ Context-Free Path Query

FLGM Formal-Language-Constrained Graph Minimization

MINW-SAT Minimum-Weight Satisfiability

SM Standard Matching-Choice

List of Symbols

D, D', D^- Respectively a graph, also called data graph or graph database, an annotated graph, and a minimized graph.

V The set of vertices of a graph.

E The set of edge labels of a graph.

s, p, o Respectively the subject, predicate and object of a triple.

G A grammar.

N Set of non-terminal symbols of a grammar.

Σ Set of terminal symbols (alphabet) of a grammar.

P The set of production rules of a grammar, also used to represent parts of an SM Expresion.

S Usually the start symbol of a grammar, also used as non-terminal symbol or as a set of pairs of open-close strings.

ε The empty string.

$A, B, C, etc.$ Usually non-terminal symbols, but may have other uses.

α, β, γ Strings of terminal and non-terminal symbols.

A, PA Respectively an automaton, also used as non-terminal symbol, and a product automaton.

I A set of trace items.

C Usually a position set, also used as a set of strings or cost of a solution to a problem instance.

$\varphi_{SM}, \varphi_{RE}$ Functions for translating SM expressions into grammars.

$\mathcal{L}$ The language of a given expression, grammar, or automaton.

Liste des Algorithmes

1 The Trace-Item-based Algorithm [40] 23

2 The RGM Algorithm 45

3 The CFGM Algorithm 47

4 Auxiliary subprograms for Algorithm 3 48

Contents

1 Introduction **1**

2 Background **3**

2.1 Formal Languages . . . 3

2.2 Graphs . . . 4

3 State-of-Art **7**

3.1 Non-Regular Expressions . . . 7

3.2 Context-Free Path Queries . . . 8

3.3 Minimum-Weight Satisfiability . . . 11

4 Standard Matching-Choice Expressions for Defining Path Queries **13**

4.1 Matching-Choice Languages . . . 13

4.2 Standard Matching-Choice Expressions . . . 14

4.3 Defining Query Patterns with SM Expressions . . . 17

4.4 Conclusions . . . 19

5 Context-Free Path Query Evaluation **20**

5.1 The Trace-Item-Based Algorithm . . . 23

5.2 Complexity . . . 25

5.3 Experiments . . . 28

5.3.1 Ambiguity and Normal Form . . . 28

5.3.2 Length of Derivations . . . 29

5.3.3 Synthetic Social Networks . . . 32

5.3.4 Ontologies . . . 34

5.3.5 Phylogenetic Trees . . . 39

5.3.6 Single Source Queries . . . 40

5.4 Conclusions . . . 41

6 Formal-Language-Constrained Graph Minimization **43**

6.1 Problem Definition . . . 43

6.2 Regular-Language-Constrained Graph Minimization . . . 45

6.3 Context-Free-Language-Constrained Graph Minimization . . . 47

6.4 Experiments . . . 50

6.4.1 Complete Graphs . . . 51

6.4.2 Fractal Triangles . . . 51

6.5 Conclusions . . . 55

7 Final Remarks **56**

7.1 Publications Resulting from This Research . . . 56

7.2 Future Work . . . 56

Chapter 1

Introduction

We deal with three problems related to graph databases and context-free languages: expressing context-free languages using alternative notations, context-free path querying, and minimizing graphs constrained to context-free path queries.

Path queries define patterns to match paths in a labeled, directed data graph, such that the string formed by the concatenation of the labels belongs to a given language. Graph query languages usually support regular expressions for the definition of path queries. However, regular expressions are not enough to specify some important properties, such as same generation queries [1], where one wants to find pairs of vertices on the same level of a given hierarchy. To query non-regular paths, context-free languages can be used. Context-free languages are usually specified by context-free grammars, but these are somehow complex and are not as popular as regular expressions. Also, evaluating context-free patterns require more computational power than regular ones.

Context-free path queries [26] define path patterns in terms of a context-free grammar. This kind of query is interesting in domains such as genetics [40], data science [11], and source code analysis [62]. Context-free path queries can answer, for instance, same generation queries. The evaluation of a context-free path query, however, is more complex than the evaluation of a regular path query.

An answer to a path query may come from several subgraphs in the database. Such redundancy is sometimes undesired, either for storage, privacy, or cost issues. Minimum weight/cost satisfiability in graph databases can reduce the amount of data in a way that the answers to some given queries are preserved.

Three main publications resulted from the work in this thesis:

1. In Medeiros *et al.* [35], we introduce SM (Standard Matching-Choice) Expressions and use them as the basis for the specification of non-regular languages for querying graph databases. SM Expressions allow us to specify path queries in terms of a meaningful subset of context-free languages. We define the syntax and semantics of SM Expressions and formalize the translation of SM Expressions into a set of rules of context-free grammars. The translation from SM Expressions into context-free grammars make it possible to evaluate queries using solutions found in the literature. Finally, we illustrate the usage of SM Expressions with example queries. A previous attempt on defining non-regular expressions can be found in Medeiros *et al.* [37].

2. In Medeiros *et al.* [41], we present an algorithm for context-free path query processing. We prove the correctness of our approach and show its runtime and memory complexity. Also, we show its viability by means of experiments with prototypes. The experiments include both synthetic and real databases. Our algorithm shows performance gains when compared to other algorithms implemented using single-thread programs. This paper is an evolution of previous proposals [38, 39, 40].

3. In Medeiros *et al.* [36], we present a problem of graph minimization. Minimizing the amount of data in a graph is desirable in situations such as data privacy preservation, network optimization and source code analysis. Since naïvely deleting data from a graph database can make it useless, we concentrate on the problem of reducing the amount of data in a graph database respecting a set of

user-defined utility queries. We formalize that problem, study its complexity and develop solutions for its regular and context-free versions.

This document is organized as follows. In Chapter 2, we provide the theoretical background. In Chapter 3, we provide an overview of current proposals and techniques related to our work. In Chapter 4, we define a notation for expressing a subset of context-free languages, which can be incorporated in graph query languages. In Chapter 5, we present an algorithm for the evaluation of context-free path queries. In Chapter 6, we formalize the graph minimization problem and present our solutions to its regular and context-free versions. In Chapter 7, we summarize the discussion.

Chapter 2

Background

Let us present a background on formal languages, graph databases, context-free path queries and related.

2.1 Formal Languages

In this section, we introduce basic concepts from the field of Formal Languages. These concepts are the base for recognizing string patterns.

A *language* is a set of *strings* built from a finite alphabet. A language can be defined by a *grammar*.

Definition 1 (Grammar)**.** *A* grammar *is a quadruple $G = (N, \Sigma, P, S)$ where N is the set of non-terminal symbols, Σ is the set of terminal symbols (alphabet), P is the set of production rules in the form $\alpha \to \beta$, for $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^*$, and $S \in N$ is the start symbol.*

*Given a string $\alpha\delta\gamma$, where $\delta \in (N \cup \Sigma)^+$, and a production rule $\delta \to \beta$, one can apply this rule to produce the string $\alpha\beta\gamma$, denoted $\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$. If, for a given string s, one can successively apply production rules and generate a string s', denoted $s \Rightarrow^* s'$, we say that s' is s-*derivable.

Given a grammar and a string, *string recognition* consists of verifying if the string belongs to the language generated by the grammar. There exists a wide range of algorithms for string recognition, varying in computational complexity and power. Each recognizer is adequate to a class of languages.

Chomsky [17] hierarchically classified grammars and their corresponding languages according to the form of their production rules. The more one descends on this hierarchy, more constraints are added to the form of the grammars' production rules, thus restricting the class of languages generated. Let α, β and γ be arbitrary strings, X and Y non-terminal symbols and a a terminal symbol. Grammars in the widest class in Chomsky's hierarchy contain rules of the form $\alpha \to \beta$. The only constraint imposed for this class is that the left-hand side is not empty. That means any string can be rewritten to any other. The class of *context-sensitive* grammars contains those whose production rules are of the form $\alpha X \beta \to \alpha\gamma\beta$. That means X can derive γ only if it is in the context α and β. The grammars in the *context-free* class do not use contexts α and β, so the production rules are of the form $X \to \gamma$. The last and most restrictive class are *regular* grammars. Production rules in a regular grammar are of the form $X \to a$ or $X \to a\ Y$. The empty rule $X \to \varepsilon$ is allowed only if X is the start symbol.

In this work, context-free and regular grammars receive more attention. Regular grammars are less expressive, but are simpler and can be evaluated faster. Regular expressions are a widespread alternative notation for regular grammars. Context-free grammars, on their turn, are more expressive than regular grammars, but they increase the computational power required for recognizing their associated languages. Although some initiatives propose more pragmatic notations for expressing context-free languages [37, 35, 59, 61, 57], the most popular notation is context-free grammars (CFG). We denote grammars by their set of production rules only. Also, we say that a grammar is in Normal Form [34] if its production rules are of the form $A \to BC, A \to a, A \to \varepsilon$. This is a variation of Chomsky Normal Form [34] that allows empty production rules for any non-terminal symbol. Normal forms are useful because they allow us to make assumptions on the right-hand side of rules.

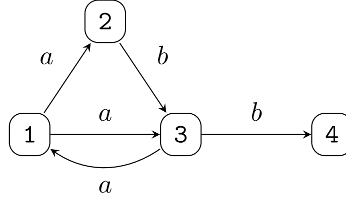


Figure 2.1: Example Graph

2.2 Graphs

In this section, we formalize concepts related to graphs and path queries over them. To illustrate those definitions, we refer to the data graph D from Figure 2.1 and grammar G given by $\{S \to a\ S\ b, S \to \varepsilon\}$, which defines the language $a^n b^n$.

Definition 2 (Graph)**.** *A* graph *(also called* data graph *and* graph database*) is a set of triples in $V \times L \times V$, where V is a set of vertices, representing subjects and objects, and L is a set of edge labels, representing predicates. All graphs in this thesis are directed. We call* subgraph *any subset of a graph. The graph from Figure 2.1 is $D = \{(1, a, 2), (1, a, 3), (2, b, 3), ...\}$.*

Triples in a graph database can be arranged to form paths.

Definition 3 (Paths and Traces)**.** *Given graph D, a* path *is a sequence of triples $(t_1, t_2, ..t_n)$ in D, where $t_i = (v_i, l_i, v_{i+1})$, for $0 < i < n$. The set of paths between two vertices v_1 and v_n is denoted by $paths(v_1, v_n)$:*

$$paths(v_1, v_n) = \{\pi \mid \pi = ((v_1, l_1, v_2), ..., (v_{n-1}, l_{n-1}, v_n)) \ \wedge\ (v_i, l_i, v_{i+1}) \in D\}.$$

The concatenation of edge labels l_i of given path π form a string called the trace *of π. For the path $\pi = ((1, a, 2), (2, b, 3), (3, b, 4)) \in paths(1, 4)$ in Figure 2.1, the trace of π is abb.*

Definition 4 (Context-Free Path Query)**.** *Given a data graph D and a context-free grammar G, a* context-free path query *(CFPQ) Q is a set of pairs (x, A) where $x \in V$ and $A \in N$. The evaluation of a context-free path query Q produces the set of all vertices y such that there exists a path from x to y whose trace s is A-derivable, that is:*

$$Eval(Q) = \{y \mid (x, A) \in Q\ \wedge\ \exists s\ .\ A \Rightarrow^* s\ \wedge\ s \in traces(paths(x, y))\}.$$

In our example, the query $(1, S)$ has answers $\{1, 3, 4\}$.

The next definition establishes closure rules for the data graph D, by adding edges labeled by non-terminal symbols of the grammar G. We refer to graphs that contain such non-terminal-labeled edges as *annotated graphs*.

Definition 5 (Closure Rules for CFG-Reachability)**.** *Let G be a context-free grammar and D a data graph. Given vertices $u, v \in V$ and a symbol $\alpha \in \Sigma \cup N$, the ternary reachability relation*

$$\mathcal{R}_{G,D} : V \times (\Sigma \cup N) \times V$$

defines the data graph containing the triples (u, α, v) that connects vertices u and v by an α-derivable path in D. This relation is recursively defined as follows:

1) *Every triple in D is in the closure:*

$$\frac{}{(u, l, v) \in \mathcal{R}_{G,D}} \quad \forall (u, l, v) \in D$$

In our example, $\{(1, a, 2), (1, a, 3), (2, b, 3)\} \subset \mathcal{R}_{G,D}$.

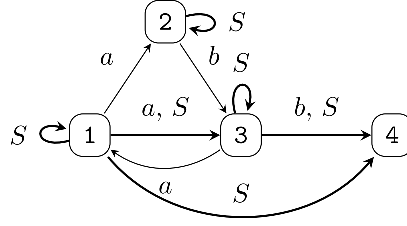


Figure 2.2: Annotated graph D' corresponding to the transitive closure of G and D

2) *For each rule $A \to \varepsilon \in P$, an A-labeled loop edge is added to every vertex in D:*

$$\frac{}{(v, A, v) \in \mathcal{R}_{G,D}} \quad \forall v \in V,\ A \to \varepsilon \in P$$

In our example, $\{(1, S, 1), (2, S, 2), (3, S, 3), (4, S, 4)\} \subset \mathcal{R}_{G,D}$, since $S \to \varepsilon$ is a production rule of the grammar.

3) *For each rule $A \to \alpha_1 \cdots \alpha_n \in P$, an A-labeled edge is added between nodes v_0 and v_n if there exist triples $(v_i, \alpha_i, v_{i+1}) \in \mathcal{R}_{G,D}$, for $0 \leq i \leq n$:*

$$\frac{(v_0, \alpha_1, v_1) \in \mathcal{R}_{G,D} \quad \cdots \quad (v_{n-1}, \alpha_n, v_n) \in \mathcal{R}_{G,D}}{(v_0, A, v_n) \in \mathcal{R}_{G,D}} \quad \forall A \to \alpha_1 \cdots \alpha_n \in P$$

In our example, $\{(1, S, 3), (1, S, 4), (3, S, 4)\} \subset \mathcal{R}_{G,D}$.

The graph D', which is D annotated with non-terminal edges from the closure of $\mathcal{R}_{G,D}$ is shown in Figure 2.2.

For each non-terminal symbol $A \in N$, there exists a triple $(x, A, y) \in \mathcal{R}_{G,D}$ if and only if $A \Rightarrow^* s$ and s defines a path in D. This is given by the following proposition [41].

Theorem 1. *Given a data graph D and a context-free grammar G,*

$$(x, A, y) \in \mathcal{R}_{G,D} \iff A \Rightarrow_G^* s \text{ and } s \text{ defines a path from } x \text{ to } y \text{ in } D.$$

Proof Sketch. This is a direct consequence of Definition 5. The "if" direction is by rule induction on $\mathcal{R}_{G,D}$, where the base cases come from Definitions 5.1 and 5.2, and the inductive step comes from Definition 5.3. The "only if" direction is by induction on the length of the string s, i.e., the length of the A-derivable path from x to y. □

Although regular path queries naturally fit Definition 4, we provide an alternative definition using automaton states.

Definition 6 (Regular Path Query)**.** *Given a data graph D and a regular expression exp, a* regular path query *Q is a set of pairs (x, q_0), where $x \in V$ and q_0 is the start state of the automaton A that recognizes exp. The evaluation of a regular path query Q produces the set of all vertices y such that there exists a path from x to y whose trace belongs to the language of exp.*

A standard technique for evaluating regular path queries is based on the construction of a *product automaton*. Let D be a graph, *exp* a regular expression and Q a set of query pairs. An automaton A is a set of states, containing one initial state and a number of final states, and transitions between them [6]. Automata can recognize regular languages. On its turn, a product automaton PA is formed by states, which are pairs (v, q), where v is a vertex in D and q is a state in A, and transitions between them. The construction of PA proceeds as follows:

1. every pair (s, q_0), where $s \in V$ and q_0 is the start state of A, is a start state in PA.

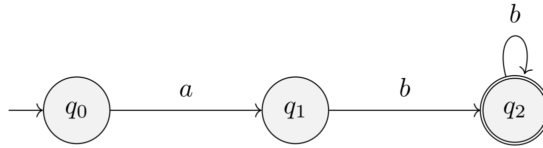


Figure 2.3: Automaton that recognizes the language given by `abb*`

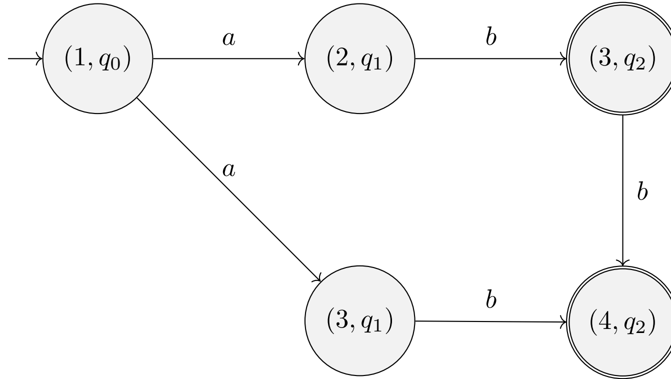


Figure 2.4: Product-automaton that recognizes regular paths in D that belong to the language of `abb*`

2. for each $(s, p, o) \in D$ and $(q_i, p, q_j) \in A$, if (s, q_i) is a state in PA, then $((s, q_i), p, (o, q_j)) \in PA$.
3. for each vertex v in D and final state q_f in A, if (v, q_f) is in PA, then (v, q_f) is a final state in PA.

In PA, a path from a start state (u, q_0) to a final state (v, q_f) corresponds to an *exp*-derivable path from u to v in D. Therefore, to answer a given regular path query Q, it suffices to start at every pair in Q and search for reachable final states.

Let `ab*` be regular a regular expression and let A be the automaton shown in Figure 2.3. The automaton A recognizes the language generated by `abb*`. The product automaton between A and the graph D from Figure 2.1 considering the query $Q = \{(1, q_0)\}$ is shown in Figure 2.4.

Chapter 3

State-of-Art

We gather in the next sections works related to non-regular expressions, context-free path queries and data minimization. These sections are based on Medeiros *et al.* [35], Medeiros *et al.* [40] and Medeiros *et al.* [36].

3.1 Non-Regular Expressions

The design of syntactic expressions to define non-regular languages is a topic that has been studied for decades. Differently from the case of regular expressions and languages, there is no consensus about a notation that may be used to describe context-free languages or, at least, a subset of them. We briefly analyze the proposals that inspired our work.

Cap Expressions [59] define context-free languages by building expressions that contain placeholder symbols that may be replaced by the expression itself. If $P(\alpha)$ is an expression describing a language, such that it does not contain the symbol $\hat{\alpha}$, the expression $\langle P(\hat{\alpha})\rangle_\alpha$ is a cap expression that describes a language. The language described by $\langle P(\hat{\alpha})\rangle_\alpha$ is formed by all the words obtained *(i)* from words in $P(\alpha)$ by any finite number of substitutions of α by words in $P(\alpha)$ and *(ii)* from words obtained in previous substitutions, also performing any number of substitutions of α by words in $P(\alpha)$. For example, the language a^nb^n is given by the cap expression $\langle\varepsilon \ \cup \ a\hat{\alpha}b\rangle_\alpha$. Any context-free language can be defined by expressions using concatenation, union and Cap operations [59]. Even if Cap Expressions can be seen as a step towards a simple notation to describe context-free languages, their use is not popularized.

Another notation for the definition of non-regular languages are *Linear Expressions* [50]. This class of expressions use the $_L$ and $_R$ annotations on the terminal symbols of a regular expression, to indicate the appearance of that symbol in the left or right-hand side repeated portion of a string belonging to a context-free language. For instance, the linear expression $(a_Lb_Rb_R)^*$ describes the language $\{a^ib^{2i} \mid i \geq 0\}$. The language described by the linear expression $(a_La_R + b_Lb_R)^*$ is $\{ww^r \mid w \in \mathcal{L}((a+b)^*)\}$ [50]. Linear expressions are capable of expressing linear languages, a proper subset of context-free languages. To the extent of our knowledge, these expressions have not been used in query or programming languages.

Nested Regular Expressions [45] are an extension of regular expressions that define query paths in nSPARQL, an extension of the SPARQL graph database query language. Nested regular expressions include a "[_]" operator to define branching. For instance, the expression $a[b*]c$ describes all paths $a\ c$ in the graph, such that there exists a path $b*$ departing from the node of the graph that is reached after the prefix path a. In this way, nSPARQL paths are defined by nested regular expressions and *navigational axes*, similar to those in XPATH [52].

Extended Regular Expressions [22] are an extension of regular expressions that use variables to refer to parts of the expression itself. These variables are also called backreferences. The notation describes a proper subset of context-sensitive languages that are not comparable with context-free languages. Despite their use in different implementations, Extended Regular Expressions are problematic, since, in some situations, they are provably undecidable even when they have just one variable.

In a previous work [37], we propose *Recursive Expressions*, which were intended to describe languages of matching parentheses. We extend regular expressions with a ternary operator to define pairs of matching

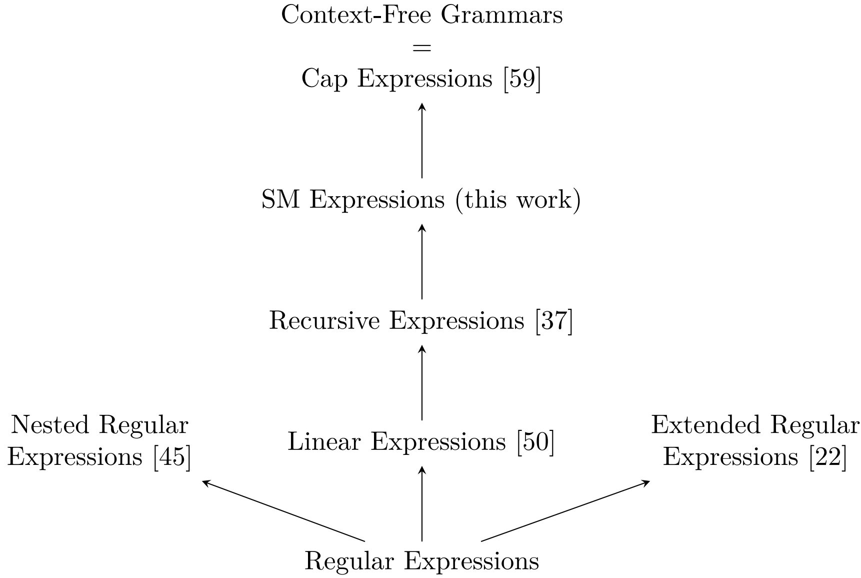


Figure 3.1: Summary of non-regular expressions

parentheses around a core set of strings [35]. Recursive Expressions define a class of non-regular languages that are a proper subset of context-free languages. Although these expressions are more expressive than regular expressions, they lack the capability of describing some useful languages.

Figure 3.1 summarizes the works presented in this section. The arcs indicate a "subset of" relationship between expressions.

3.2 Context-Free Path Queries

Current graph query languages support regular expressions for the definition of path patterns. However, in the last few years, Context-Free Path Queries (CFPQs) have received attention by the scientific community.

Hellings [26] proposes an algorithm to evaluate CFPQs based on the CYK parsing technique [25]. The input is a grammar in Normal Form and a data graph. The algorithm processes the whole graph by adding non-terminal-labeled edges to link vertices that are connected by a path derivable by a given non-terminal symbol. The worst-case time complexity is $\mathcal{O}(|N||E| + (|N||V|)^3)$, where N is the set of non-terminal symbols, V is the set of nodes of the graph and E is the set of edges.

In a posterior work, Hellings [27] studies two path-based semantics and provides algorithms for implementing them. For each pair of vertices connected by a path accepted by the given context-free grammar, evaluating a CFPQ under the *all-paths semantics* implies providing all paths linking them, whilst the *single-path semantics* implies providing only one of those paths. The algorithm implementing the all-paths semantics works by annotating grammar productions with graph vertices. It processes the entire graph for answering queries and has worst-case time complexity of $\mathcal{O}(|\mathbf{P'}_\lambda| + |\mathbf{P'}_\Sigma| + |E| + (|\mathbf{N}||V|)^3)$, where $\mathbf{P'}_\lambda$ is the set of empty production rules, $\mathbf{P'}_\Sigma$ is the set of unitary (terminal) production rules, E is the set of edges in the graph, $\mathbf{N}$ is the set of non-terminal symbols in the grammar and V is the set of vertices. The authors provide an algorithm for producing paths given an annotated grammar whose complexity is linear in terms of the length of the paths produced. Combining this algorithm with others that derive shortest strings or strings limited to a certain length, the author implements the single path semantics.

Pérez *et al.* [45] extend SPARQL to include *nested regular expressions* and *navigational axes* to be used as property paths. Nested regular expressions are regular expressions that include conditional expressions. Navigational axes are a flexible notation for navigating triple elements. The language proposed by the authors, called nSPARQL, combines these two concepts to add flexibility to the formulation of queries. This work does not deal with CFPQs. The worst-case runtime complexity of this algorithm for a single source vertex is $O(|D| * |exp|)$, where $|D|$ is the number of edges of the adjacency list representing the graph and $|exp|$ is the size of the expression. This algorithm is roughly quadratic in terms of the number

of vertices in the graph for a single source vertex, but it might reach cubic complexity if all vertices are used as sources.

Zhang *et al.* [61] propose the query language cfSPARQL. The language supports not only queries defined by context-free grammars, but also nested expressions and navigational axes [45]. The evaluation mechanism of cfSPARQL is an adaptation of the algorithm in [26] and presents the same complexity.

Grigorev and Ragozina [24] propose an LL-based approach to evaluate CFPQs. Their approach uses the GLL [48] parsing technique to define an algorithm for querying data graphs with worst-case time complexity of $\mathcal{O}(|V|^3 max_{v \in V}(deg^+(v)))$, where V is the set of vertices and $deg^+(v)$ is the outdegree of vertex v. Notice that for complete graphs this runtime complexity reaches $\mathcal{O}(|V|^4)$.

Azimov and Grigorev [8] propose an algorithm that uses a matrix representation of the graph where each cell contains the set of edges between two vertices, represented by line and column. The proposal is based on Valiant's parsing algorithm [56]. Their implementation uses an efficient, GPU-based computation of the transitive closure of that matrix to answer queries. The algorithm calculates all possible non-terminal labelled edges between nodes of the graph. The time complexity is $\mathcal{O}(|V|^2|N|^3(BMM(|V|)+BMU(|V|)))$, where V is the set of vertices, N is the set of non-terminal symbols and BMM and BMU refer to the number of elementary operations needed in the matrix multiplication. A subsequent paper [43] compares the performance of different implementations for this algorithm. Their results show the efficacy of using GPU for matrix multiplication. Derivations of this technique include reducing CFPQ evaluation to solving systems of equations over real numbers [53] and implementations of single-path and all-paths semantics [54, 7].

Santos *et al.* [47] design a query algorithm based on the LALR[1] parsing [2]. The proposal extends Tomita's algorithm and the GSS data structure [55] to simultaneously discover context-free paths on a data graph. The proposed algorithm does not need to pre-process the whole graph to answer the query. The time complexity of this algorithm is given by $\mathcal{O}(|V|^{4+k} \cdot |I|^{1+k} \cdot |\Sigma| \cdot |N|)$, where k is the maximum size of the right-hand side of the production rules in the grammar and I is the number of lines of the LALR(1) parsing table.

Medeiros *et al.* [39] propose a query processing algorithm based on the LL parsing technique[2] [2]. For queries of the form (x, S), where x is a vertex of the graph and S is a non-terminal symbol, the algorithm proceeds in a top-down manner, trying to discover S-generated paths from x. The worst-case time complexity of their algorithm is $\mathcal{O}(|V|^3 \cdot |P|)$, where P is the set of production rules of the grammar.

Kuijpers *et al.* [29] compare the CFPQ evaluation methods in [8, 47, 27]. The authors perform experiments with several data sets, including real and synthetic ones. These methods were implemented in Java, using Neo4j's graph store API to represent data and running a single execution thread. Because of that, the matrix-based algorithm, which should make use of the GPU for high performance computation, does not present a very competitive execution time in some cases.

Table 3.1 summarizes the path query algorithms mentioned in this section.

[1] Look-ahead, left-to-right, rightmost derivation: a simplified LR technique that is more efficient than traditional LR parsers in some situations.

[2] Left-to-right, leftmost derivation: top-down parsing technique that uses tokens of look-ahead to predict what grammar rules should be used to recognize a given string.

Work	Technique/ Algorithm	Time Complexity	CPU/ GPU	Language
Hellings [26]	CYK	$\mathcal{O}(\|N\|\|E\| + (\|N\|\|V\|)^3)$	CPU	Context-Free
Hellings [27]	CYK, Grammar Annotation	$\mathcal{O}(\|\mathbf{P'}_\lambda\| + \|\mathbf{P'}_\Sigma\| + \|E\| + (\|\mathbf{N}\|\|V\|)^3)$	CPU	Context-Free
Pérez *et al.* [45]	Product Automaton	$\mathcal{O}(\|V\|^3 * \|exp\|)$	CPU	Nested Regular Expressions + Navigational Axes
Zhang *et al.* [61]	CYK	$\mathcal{O}(\|N\|\|E\| + (\|N\|\|V\|)^3)$	CPU	Context-Free + Navigational Axes
Grigorev and Ragozina [24]	GLL	$\mathcal{O}(\|V\|^3 max_{v \in V}(deg^+(v)))$	CPU	Context-Free
Azimov and Grigorev [8]	Matrix Multiplication	$\mathcal{O}(\|V\|^2\|N\|^3(BMM(\|V\|) + BMU(\|V\|)))$	GPU	Context-Free
Santos *et al.* [47]	LALR, GSS	$\mathcal{O}(\|V\|^{4+k}\|I\|^{1+k}\|\Sigma\|\|N\|)$	CPU	Context-Free
Medeiros *et al.* [39]	LL, CYK	$\mathcal{O}(\|V\|^3\|P\|)$	CPU	Context-Free
This work	Trace-Items-based	$\mathcal{O}(\|V\|^3\|P\|^2k^2)$	CPU	Context-Free

Table 3.1: Summary of path query evaluation algorithms

3.3 Minimum-Weight Satisfiability

The Minimum-Weight Satisfiability (MINW-SAT) problem consists of, given a formula composed of boolean variables, finding a mapping from variables to boolean values that satisfies the formula with the smallest number of variables set to true. This problem is NP-hard and (as we show in Chapter 6) is closely related to the graph minimization problem we propose to solve. Variations of MINW-SAT, e.g. the minimum weight exact satisfiability problem [46] or the weighted partial minimum satisfiability problem [30], are beyond the scope of this research.

The main difference from MINW-SAT to the problem we propose to solve in this work is that we are also concerned about paths in the graph respecting a given formal language. At a first glance, traditional graph algorithms like those for building minimum spanning trees or finding shortest paths between vertices might seem sufficient for solving that problem. However, this is not the case. The problem of computing a minimum spanning tree does not consider directed graphs, nor a set of queries and a grammar as input, and also its solution does not contain cycles. Algorithms for computing shortest paths between all vertices in a graph (assuming they can take a formal language as input [9]) do not guarantee overall minimum weight, since longer overlapping paths might provide more economical solutions. For instance, consider the following instance of the all shortest paths problem. Let the input graph be an a-labeled complete graph (where all pairs of vertices are connected by an a-labeled edge) with equal weight for all edges and the language a^*. The shortest path between any pair of vertices in the graph is the direct edge between them. The solution is therefore the complete graph itself, which is far from an optimum solution (a cycle connecting all vertices).

Since boolean satisfiability problems are, in general, NP-hard, approximative algorithms have been proposed to solve them, or special cases of theirs, in viable time [60]. A popular algorithm that provides exact solutions to boolean satisfiability problems is DPLL [20], which follows a *branch-and-bound* strategy. A branch-and-bound algorithm enumerates candidate solutions in the form of a rooted tree, where the root contains all variables. Each branch of this tree is a subset of its parent. To avoid exhaustive enumeration of all candidate solutions of a branch, the algorithm estimates upper and lower bounds using efficient functions and discards branches that cannot produce a better solution than the one found so far. Among other techniques to solve this class of problems, we can cite randomized approximation, linear programming [18] and simulated annealing [28].

Liberatore [33] designs a DPLL-based exact algorithm for solving an equivalent problem to the MINW-SAT problem. The author develops a prototype of his algorithm and compares it to a local-search maximum satisfiability solver called MAXWALKSAT.

Li [31] studies the MINW-SAT and some related problems, providing a hierarchical classification for them. The author designs and implements algorithms for solving those problems and validate their performance with experiments.

Sebastiani *et al.* [49] proposes a MINW-SAT solver for goal graph problems. The authors implement tools capable of deciding whether there exists a set of input goals that satisfies the output goals of the goal graph and, if such solution exists, finding a minimum cost one. The authors evaluate the performance of the developed tool using a goal graph from a case study involving public transportation services.

Fu and Malik [23] design a MINW-SAT solver, called MinCostChaff. The MinCostChaff solver is a DPLL-based [20] SAT solver inspired by its popular implementation Chaff [44]. The performance of MinCostChaff and some variations of it are validated with benchmarks.

Anciaux et al. [5] formulate the *minimum exposure* approach. Their goal is to minimize the information provided by clients to companies when such information is strictly required. Health insurance or loan companies, for example, require applicants to feed them personal information to classify their situation according to a set of rules or conditions. In the minimum exposure approach, *service providers* supply their *collection rules* so that *users* can locally compute the offers they can obtain, using all information available. The user can then run a minimum exposure algorithm to identify the minimum information needed to achieve those offers, and return it to the service provider. In this way, the company receives only the information needed to satisfy the conditions for that offer. The authors argue that the minimum exposure approach is good for both client and company, because the user exposes less information, and the

company retains less information that could be disclosed in a future data breach. The minimum exposure problem is NP-hard. The authors propose an heuristic to solve it in polynomial time. Experiments assess the exposure reduction ratio and scalability of the technique. This approach can be combined with anonymization techniques to guarantee privacy protection.

Another strategy for solving complex combinatorial problems is to design algorithms that might take exponential time in some cases, but are as fast as possible in most cases. In these approaches, data preprocessing is often applied to reduce the size of the input. *Kernelization* algorithms transform a given problem instance into an equivalent, smaller one, in polynomial time. The smaller instance can then be solved faster and its answer converted back to the original one [15]. Many algorithms might benefit of graph kernelization to solve graph modification problems more efficiently [12, 14, 13, 19, 21].

Chapter 4

Standard Matching-Choice Expressions for Defining Path Queries

Most graph query languages use *regular expressions* to define path queries. However, regular expressions cannot specify some important properties over graph databases, such as same generation queries [1]. This kind of query characterizes a CFPQ. The definition of specification languages for CFPQs is important since it impacts the practical usage of graph databases. The next sections are an adaptation of Medeiros *et al.* [35]. We introduce *Standard Matching-Choice (SM) Expressions* for the specification of non-regular path queries and demonstrate important properties.

Our proposal is inspired by the family of Standard Matching-Choice Sets, a meaningful subset of context-free languages presented by Yntema [58]. We define the syntax and semantics of SM Expressions, and we formalize the translation of SM Expressions into a set of rules of a context-free grammar. The translation from SM Expressions to context-free grammars fills the gap between a more natural specification language for querying graph databases and the existing query engines for CFPQs.

4.1 Matching-Choice Languages

Yntema [58] defines an inclusion hierarchy of classes of context-free languages that can be defined using operations over sets of strings. Given an alphabet Σ, the building blocks for defining those languages are *(i)* a set of pairs of strings $S \subseteq \Sigma^* \times \Sigma^*$ and *(ii)* a set of strings $C \subseteq \Sigma^*$.

The set of pairs S can be seen as a generalization of open and close parentheses. The set of strings C contains strings that will be surrounded by those in S. The languages built from these sets contain strings $\alpha\beta\gamma$ such that $(\alpha, \gamma) \in S$ and $\beta \in C$. Given sets of pairs of strings S_1 and S_2, the author introduces *Matching-Choice Sets* defined over S_1 and S_2 as follows:

$$S_1 \oplus S_2 = \{(x, y) \mid (x, y) \in S_1 \vee (x, y) \in S_2\} \tag{4.1}$$

$$S_1 S_2 = \{(xz, wy) \mid (x, y) \in S_1 \wedge (z, w) \in S_2\} \tag{4.2}$$

$$S_1* = \{(\varepsilon, \varepsilon)\} \cup \{(x_1 \cdots x_n, y_n \cdots y_1) \mid (x_i, y_i) \in S_1, i \leq n, n \in \mathbb{N}^+\} \tag{4.3}$$

Notice that these operations define pairs of matching strings. We can now present the next definition.

Definition 7 (Matching-Choice Languages [58]). *Given a matching choice set S (i.e., a set of pairs built using the operations above) and a set of strings C, the* Matching-Choice Language $S \circ C$ *is defined as:*

$$S \circ C = \{xzy \mid (x, y) \in S \wedge z \in C\}$$

Given a finite number of sets of strings A_i, B_i ($1 \leq i \leq n$), we can build a set S of pairs of strings by *(i)* defining the sets of pairs $A_i \times B_i = S_i$, and *(ii)* recursively applying the union, sequence and closure operations over the sets S_i. The sets of strings A_i, B_i and C are called the *underlying sets* of the expression $S \circ C$. Notice that the elements in S can be seen as pairs of matching parentheses while the

strings in C correspond to strings placed between them. The union (4.1), sequencing (4.2) and star (4.3) operations ensure that the set $S \circ C$ is formed by strings containing well-formed parenthesized expressions. The depth of the innermost $\circ$ operator defines the *rank* of an SM Expression.

Yntema [58] defines the class of *Standard Matching-Choice Languages*, a proper subset of context-free languages that can be described using the above-mentioned operations. In the next section, we define a set of expressions to denote Standard Matching-Choice Sets.

4.2 Standard Matching-Choice Expressions

In this section, we propose SM Expressions to define Standard Matching-Choice Languages [58]. We provide the semantics of these expressions by means of SM Languages and show that the class of languages defined by SM Expressions is exactly the class of Standard Matching-Choice Sets. We also provide a way of converting an SM Expression into a context-free grammar.

Regular expressions are the basis for the definition of SM Expressions. Given an alphabet $\Sigma = \{t_1, \dots, t_n\}$, the set of regular expressions over Σ is the set of strings defined by $R \to \texttt{()} \mid t_1 \mid \dots \mid t_n \mid \texttt{(}R\texttt{)} \mid RR \mid R\texttt{|}R \mid R\texttt{*}$.

Definition 8 (Syntax of SM Expressions)**.** *The set of Standard Matching Choice Expressions over an alphabet Σ is inductively defined as follows:*

1. *Any regular expression E over Σ is an SM Expression.*

2. *If E, E', E_0 are SM Expressions over Σ, then $<E>E_0<E'>$ is an SM Expression:*

$$\frac{E, E_0, E' \text{ are SM Expressions}}{<E>E_0<E'> \text{ is an SM Expression}}$$

   This case defines the base case of SM Expressions.

3. *If $<P_1>E_0<P_1'>$ and $<P_2>E_0<P_2'>$ are SM Expressions, then $<P_2.P_1>E_0<P_1'.P_2'>$ is an SM Expression.*

$$\frac{<P_1>E_0<P_1'> \text{ is an SM Expression} \qquad <P_2>E_0<P_2'> \text{ is an SM Expression}}{<P_2.P_1>E_0<P_1'.P_2'> \text{ is an SM Expression}}$$

   This case defines SM Expressions that contains sequences of nested parentheses.

4. *If $<P_1>E_0<P_1'>$ and $<P_2>E_0<P_2'>$ are SM Expressions, then $<P_2 + P_1>E_0<P_1' + P_2'>$ is an SM Expression.*

$$\frac{<P_1>E_0<P_1'> \text{ is an SM Expression} \qquad <P_2>E_0<P_2'> \text{ is an SM Expression}}{<P_2 + P_1>E_0<P_1' + P_2'> \text{ is an SM Expression}}$$

   The SM Expressions defined above contain choice of parentheses.

5. *If $<P>E_0<P'>$ is an SM Expression, then $<: P :>E_0<: P' :>$ is an SM Expression.*

$$\frac{<P>E_0<P'> \text{ is an SM Expression}}{<: P :>E_0<: P' :> \text{ is an SM Expression}}$$

   The pair of $:_:$ operators denotes a Kleene star operator over matching parentheses.

Notice that the strings P_i in the cases above are, in general, not *SM expressions, but they denote a language of opening parentheses. Analogously, P_i' represent sets of closing parentheses. These strings may contain characters like ".", "+" and ":", that are used to build the languages that denote opening and closing parentheses.*

The next definition establishes the semantics of SM Expressions as a set of strings over an alphabet Σ. The language defined for each expression is an SM Set [58].

Definition 9 (Semantics of SM Expressions)**.** *The language $\mathcal{L}(M)$ denoted by an SM Expression M is inductively defined by the following rules:*

1. *The Standard Matching Choice language defined by a regular expression is the (regular) language denoted by that expression:*

$$\frac{}{\mathcal{L}(R) = \mathcal{L}_{RE}(R)}$$

   The function $\mathcal{L}_{RE}(R)$ defines the language denoted by a regular expression. Notice that this language is a Standard Matching-Choice set of rank 0 [58].

2. *The language defined by an SM Expression $<E>E_0<E'>$ over Σ is defined in terms of the languages defined by the SM Expressions E, E', and E_0, as follows:*

$$\frac{L = \mathcal{L}(E), \qquad L_0 = \mathcal{L}(E_0), \qquad L' = \mathcal{L}(E')}{\mathcal{L}(<E>E_0<E'>) = (L \times L') \circ L_0}$$

   The languages L, L' and L_0 are the underlying sets of $\mathcal{L}(<E>E_0<E'>)$. If n is the maximum rank among the ranks of L, L' and L_0, then $\mathcal{L}(<E>E_0<E'>)$ is a Standard Matching-Choice set of rank $n+1$.

3. *The language denoted by an SM Expression $<P_2.P_1>E_0<P'_1.P'_2>$ containing sequences of nested parentheses is defined as follows:*

$$\frac{(S_1 \times S'_1) \circ C = \mathcal{L}(<P_1>E_0<P'_1>), \qquad (S_2 \times S'_2) \circ C = \mathcal{L}(<P_2>E_0<P'_2>)}{\mathcal{L}(<P_2.P_1>E_0<P'_1.P'_2>) = (S_2\ S_1 \times S'_1\ S'_2) \circ C}$$

   where S_iS_j represents the string concatenation of languages S_i and S_j.

4. *The language denoted by an SM Expression containing choices of parentheses is defined as:*

$$\frac{(S_1 \times S'_1) \circ C = \mathcal{L}(<P_1>E_0<P'_1>), \qquad (S_2 \times S'_2) \circ C = \mathcal{L}(<P_2>E_0<P'_2>)}{\mathcal{L}(<P_2 + P_1>E_0<P'_1 + P'_2>) = ((S_1 \times S'_1) \oplus (S_2 \times S'_2)) \circ C}$$

   We can verify that

$$((S_1 \times S'_1) \oplus (S_2 \times S'_2)) \circ C = ((S_1 \times S'_1) \circ C) \cup ((S_2 \times S'_2) \circ C).$$

5. *The language denoted by an SM Expression containing the synchronized repetition of matching parentheses is given by the rule:*

$$\frac{(S \times S') \circ C = \mathcal{L}(<P>E_0<P'>)}{\mathcal{L}(<:P:>E_0<:P':>) = (S \times S') * \circ\ C}$$

We can see that, for all SM Expressions M, the set $\mathcal{L}(M)$ is an SM language. The following property states that every language denoted by an SM Expression is a Standard Matching-Choice set [58].

Theorem 2 ($\mathcal{L}(SMExp) \subseteq SMLang$)**.** *Given an SM Expression M, there exists an SM language Y such that $\mathcal{L}(M) = Y$.*

Proof. This result is immediate, by structural induction on SM Expressions M and the languages $\mathcal{L}(M)$ defined for them. □

The next property shows that there exists at least one SM Expression that defines each Standard Matching-Choice set.

Theorem 3 ($SMLang \subseteq \mathcal{L}(SMExp)$)**.** *Given a Standard Matching-Choice language L [58], there exists an SM Expression M such that $\mathcal{L}(M) = L$.*

Proof. The proof of this property proceeds in two cases:

Case 1: L is a regular language. In this case, there exists a regular expression R such that $\mathcal{L}_{RE}(R) = L$.

In this case, the SM Expression that defines L is also R (see Definition 9).

Case 2: L is a Standard Matching Choice set of rank $n > 0$. In this case we have that $L = T \circ C$, where T and C are built from Standard Matching Choice sets of rank of at most n.

The proof of this case is by induction on the rank of L:

Base Case ($n = 1$), we have that the underlying sets from which $L = T \circ C$ is built are regular languages.

In this case, we have that $L = (L_1 \times L_2) \circ C$, being L_1, L_2, C regular languages. Let be R, R', R_0, respectively, regular expressions that define L_1, L_2, and C. By Definitions 9.1 and 9.2, we can see that $L = \mathcal{L}(<R>R_0<R'>)$.

Inductive Hypothesis ($n \leq k$). Let us suppose that for any Standard Matching Choice set L of rank $n \leq k$ there exists an SM Expression E such that $L = \mathcal{L}(E)$.

Inductive case ($n = k + 1$).

Since L is a Standard Matching-Choice set of rank $n > 1$, then $L = T \circ C$, such that T and C are built using Standard Matching Choice sets of ranks less than n. In this way, we can proceed by cases on the construction of T:

Suppose the Standard Matching Choice sets $L_1 = T_1 \circ C$ and $L_2 = T_2 \circ C$. By the induction hypothesis, there exist SM Expressions $S_1 = <P_1>S_0<P_1'>$ and $S_2 = <P_2>S_0<P_2'>$ such that $\mathcal{L}(S_1) = T_1 \circ C$ and $\mathcal{L}(S_2) = T_2 \circ C$.

Case $T = (T_1\ T_2)$: We need to build an SM Expression for $(T_1\ T_2) \circ C$. By Definition 9.3, we can see that the language associated to $M = <P_1.P_2>S_0<P_2'.P_1'>$ is indeed $(T_1\ T_2) \circ C$.

Case $T = (T_1 \oplus T_2)$: By Definition 9.4, we can verify that the language associated to the SM Expression $M = <P_2 + P_1>S_0<P_1' + P_2'>$ is $(T_1 \oplus T_2) \circ C$.

Case $T = T_1*$: We need to build an SM Expression for $T * \circ C$. By Definition 9.5, we can verify that $\mathcal{L}(<: P_1 :>R_0<: P_1' :>) = T * \circ C$. □

These results allow us to enunciate the following property:

Corollary 3.1. *The class of SM Sets is denoted by SM Expressions.*

Proof. Immediate from Theorems 2 and 3. □

Let us now define context-free grammars to generate the language denoted by an SM Expression. The function φ_{SM} takes an SM Expression and returns a set of production rules. For the sake of clarity, we denote as $\{S \to \alpha, \dots\}$ a set of production rules of a grammar such that the symbol S is the start non-terminal symbol. We also use the function φ_{RE}, to obtain a grammar from a regular expression.

Definition 10 (Obtaining a Grammar from SM Expressions). *We inductively define the function φ_{SM}, taking an SM expression M and producing a context-free grammar G, generating the language $\mathcal{L}(M)$.*

The rules for the starting symbol of the grammars obtained below have the right-hand side consisting of three non-terminal symbols (for prefix, inner, and suffix expressions). Greek on the right-hand side of production rules represent arbitrary strings.

1. *Given a Regular Expression R_1 and a grammar $G_1 = \{S_1 \to \alpha, \dots\}$, such that $\mathcal{L}(G_1) = \mathcal{L}_{RE}(R_1)$, then*

$$\frac{\varphi_{RE}(R_1) = \{S_1 \to \alpha, \dots\}, \qquad S, X, Y \text{ are new} \quad G = \{S \to X S_1 Y,\ X \to \varepsilon,\ Y \to \varepsilon\} \cup G_1}{\varphi_{SM}(R_1) = G}$$

2. *The grammar defined for the combinations of SM Expressions can be obtained by the rule:*

$$\frac{\varphi_{SM}(E) = \{S_1 \to \alpha, \dots\}, \quad \varphi_{SM}(E') = \{S_2 \to \beta, \dots\}, \quad \varphi_{SM}(E_0) = \{S_0 \to \gamma, \dots\} \quad S \text{ is new} \quad G = \{S \to S_1 S_0 S_2\} \cup \varphi_{SM}(E) \cup \varphi_{SM}(E') \cup \varphi_{SM}(E_0)}{\varphi_{SM}(<E>E_0<E'>) = G}$$

3. Translation for SM Expressions containing sequences of nested parentheses:

$$\frac{\begin{array}{c} G_1 = \varphi_{SM}(\texttt{<}P_1\texttt{>}E_0\texttt{<}P_1'\texttt{>}) = \{S_1 \to XS_0Y, \dots\} \\ G_2 = \varphi_{SM}(\texttt{<}P_2\texttt{>}E_0\texttt{<}P_2'\texttt{>}) = \{S_2 \to WS_0Z, \dots\} \\ G_1' = G_1 - \{w \mid w = S_1 \to \delta \in G_1\} \\ G_2' = G_2 - \{w \mid w = S_2 \to \theta \in G_2\} \qquad S, S_1', S_2' \textit{ are new} \\ G_3 = \{S_1' \to WX \mid S_1 \to XS_0Y \in G_1,\ S_2 \to WS_0Z \in G_2\} \\ \cup\{S_2' \to YZ \mid S_1 \to XS_0Y \in G_1,\ S_2 \to WS_0Z \in G_2\} \\ G = \{S \to S_1'S_0S_2'\} \cup G_1' \cup G_2' \cup G_3 \end{array}}{\varphi_{SM}(\texttt{<}P_2.P_1\texttt{>}E_0\texttt{<}P_1'.P_2'\texttt{>}) = G}$$

4. The grammar for SM Expressions containing the choice operator is:

$$\frac{\begin{array}{c} G_1 = \varphi_{SM}(\texttt{<}P_1\texttt{>}E_0\texttt{<}P_1'\texttt{>}) = \{S_1 \to XS_0Y, \dots\} \\ G_2 = \varphi_{SM}(\texttt{<}P_2\texttt{>}E_0\texttt{<}P_2'\texttt{>}) = \{S_2 \to WS_0Z, \dots\} \\ G_1' = G_1 - \{w \mid w = S_1 \to \delta \in G_1\} \\ G_2' = G_2 - \{w \mid w = S_2 \to \theta \in G_2\} \qquad S \textit{ is new} \\ G_3 = \{S \to XS_0Y \mid S_1 \to XS_0Y \in G_1\} \cup \{S \to WS_0Z \mid S_2 \to WS_0Z \in G_2\} \\ G = G_1' \cup G_2' \cup G_3 \end{array}}{\varphi_{SM}(\texttt{<}P_2 + P_1\texttt{>}E_0\texttt{<}P_1' + P_2'\texttt{>}) = G}$$

5. The grammar for SM Expressions containing the closure operator

$$\frac{\begin{array}{c} G_1 = \varphi_{SM}(\texttt{<}P\texttt{>}E\texttt{<}P'\texttt{>}) = \{S_1 \to XS_0Y, \dots\} \\ G_1' = G_1 - \{w \mid w = S_1 \to \delta \in G_1\}, \qquad S, W, Z \textit{ are new} \\ G_2 = G_1' \cup \{S \to WS_0Z\} \cup \{W \to \varepsilon, Z \to \varepsilon\} \\ G = G_2 \cup \{S \to XS\,Y \mid S_1 \to XS_0Y \in G_1\} \end{array}}{\varphi_{SM}(\texttt{<}:P:\texttt{>}E\texttt{<}:P':\texttt{>}) = G}$$

We now need to prove that the class of SM expressions and of matching choice languages is that of the languages defined by our grammars.

Theorem 4 (Standard Matching-Choice Grammars and Languages)**.** *Given an SM Expression M, $\mathcal{L}(M) = \mathcal{L}(\varphi_{SM}(M))$.*

Proof. By Rule Induction on M. □

4.3 Defining Query Patterns with SM Expressions

In this section, we show how SM Expressions can be used to build non-regular path queries. We adapt rcfSPARQL [39] to support SM expressions without using a context-free grammar in the query.

Let us present the language by means of examples. The following example (adapted from [39]) illustrates the use of SM expressions in a query.

Same-Generation Queries. This kind of query [1] looks for nodes that are *(i)* equidistant to a common ancestor, and *(ii)* have some given property.

In Figure 4.1, we depict a database D containing data about employees of a company. In the following, we rewrite the query from Medeiros *et al.* [39] by using SM Expressions. This query selects employees having the same job, but different salaries:

```
1 SELECT ?job, ?emp1, ?sal1, ?emp2, ?sal2
2 FROM D
3 WHERE {
4   ?emp1 <: boss :><: boss^-1 :> ?emp2 .
5   ?emp1 job ?job .
```

```
6     ?emp2 job ?job .
7     ?emp1 salary ?sal1 .
8     ?emp2 salary ?sal2 .
9     FILTER (?sal1 > ?sal2)
10 }
```

Query 1: CF-SPARQL query retrieving users on the same level of the hierarchy, with the same job, but different salaries

Query 1 defines a relation formed by 5-tuples. The variables at line 1 define the attributes of this relation. The path pattern at line 4 defines a path between employees (`?emp1` and `?emp2`). Notice that this query uses an SM Expression to look for paths between employees at the same level of the hierarchy. These paths are formed by nested `boss` and $\texttt{boss}^{-1}$ edges. We use the $^{-1}$ notation to express the inversion of an edge.

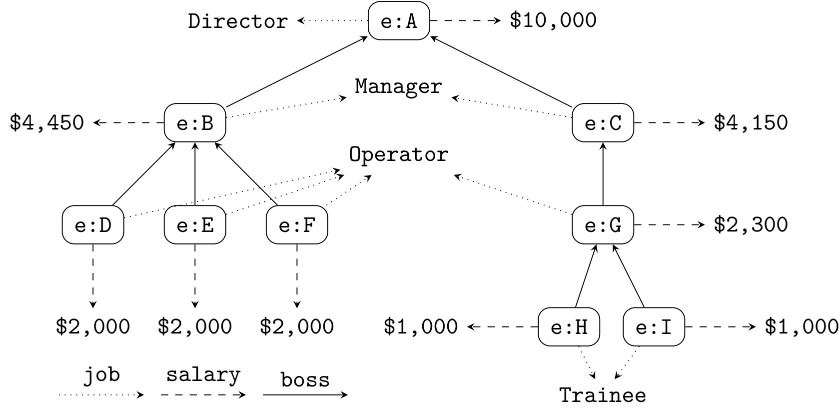


Figure 4.1: Example of hierarchy database D [39].

Lines 5-8 of the query look, respectively, for the jobs and salaries of each pair of employees identified at line 4. Notice that there is just one `?job` variable, stating that the two employees have the same job in the company. Line 9 filters the pairs of employees that have different salaries.

The SM Expression at line 4 of the query defines paths in the graph initiating with employee nodes. Departing from these nodes, the answer is defined as those employee nodes connected to the initial one by a sequence of `boss` edges followed by a sequence of equal number of $\texttt{boss}^{-1}$, thus linking employees at the same level in the hierarchy. We can verify that the SM Expression $\texttt{<: boss :><: boss}^{-1}\texttt{ :>}$ denotes the SM set $(\{boss\} \times \{boss^{-1}\}) * \circ\{\varepsilon\}$=$\{\texttt{boss}^n\ \texttt{boss}^{-1n} \mid n \geq 0\}$. We can check that the language generated by the SM Expression is the same as the one generated by the grammar $\{S \rightarrow \texttt{boss}\ S\ \texttt{boss}^{-1}, S \rightarrow \epsilon\}$.

Equal number of a's and b's. The language $\{\alpha \in \{a, b\}^* \mid \#_a(\alpha) = \#_b(\alpha)\}$ contains strings from an alphabet Σ whose number of a's is equal to the number of b's.

The equivalent SM expression for this language is `(<:a + b:><:a + b:>)*`. Notice here the use of the choice operator, which ensures that for every a, there will be a b in some subsequent position of the input string or trace, and vice-versa. The Matching-Choice set describing this language is $(\{(a, b), (b, a)\} * \circ\{\epsilon\})^*$.

Double-length right parentheses. A classic example of a non-regular context-free language is $a^n b^n$. A variation of that language is $b^n\ a\ b^{2n}$. This language is given by the Matching-Choice set $\{(b, bb)\} * \circ\{a\}$. The equivalent SM expression is `<:b:>a<:bb:>`.

Class/Type hierarchy. The RDFS vocabulary is used to define the schema of an RDF database. The terms *type* and $subClassOf$ from that vocabulary state, respectively, the type of a resource and a sub-class relationship between two types. The language of balanced pairs of $subClassOf$ and *type* edges was used in a related work [61]. It retrieves concepts on the same level of a class/type hierarchy. This language is given by the Matching-Choice set $\{(sc, sc^{-1}), (t, t^{-1})\} * \circ\{sc\ sc^{-1}, t\ t^{-1}\}$. In terms of SM expressions, the same language can be expressed as `<:sc+t:>(sc sc-1)|(t t-1)<:t-1+sc-1:>`.

A similar language was also defined in that work. The language of balanced $subClassOf$ edges, followed by an extra $subClassOf^{-1}$ retrieves concepts on adjacent levels of the class hierarchy. This

language is the same as the Matching-Choice set $(\{(sc, sc^{-1})\} * \circ \{\varepsilon\})\ sc^{-1}$. The equivalent SM expression is `<:sc:><:sc-1:>sc-1`.

Queries Over Synthetic Social Networks. The language $a^{n+1}b^m c^m d^{n+1}$, defined by Kuijpers *et al.* [29], is used to query synthetic social networks. It corresponds to the Matching-Choice set $a\{(a, d) * \circ(\{(b, c)\} * \circ\{\varepsilon\})\}d$. The SM Expression `a<:a:>(<:b:><:c:>)<:d:>d` is equivalent to it.

Recursion Inside Matching Parentheses. A more complicated language, presented by Medeiros *et al.* [37], is $(a^n\ b\ c^n)^k\ d\ (e^m\ f^m)^k$. It involves recursion inside the external matching parentheses. The Matching-Choice set $(\{(a, c)\} * \circ \{b\} \times \{(e, f)\} * \circ \{\varepsilon\}) * \circ \{d\}$ corresponds to that language. In terms of SM expressions, we can define that language as `<:(<:a:>b<:c:>):>d<:(<:e:><:f:>):>`.

4.4 Conclusions

We presented a notation to specify languages of the family of Standard Matching-Choice Sets [58]. This family of languages is a subset of context-free languages, built around the notion of parenthesizing strings. SM Expressions are an alternative notation to context-free grammars to define context-free path queries. We demonstrate how SM Expressions can be directly translated into rules of context-free grammars. This allows them to be used in graph query languages with most context-free evaluation engines available in the literature. To the extent of our knowledge, there is no previous attempt at the proposal of non-regular expressions based on Yntema's Matching-Choice sets.

Our work can be extended by considering different perspectives: (*i*) analyzing the asymptotic costs associated to the translation of SM Expressions into context-free grammars; (*ii*) identifying the class of context-free path queries that cannot be expressed with SM Expressions; (*iii*) comparing the expressiveness of SM Expressions with other proposals of non-regular expressions; (*iv*) investigating the usage of SM Expressions as part of other non-regular specification languages; and (*v*) assessing the usability of SM Expressions via experiments with students and professionals from Computer Science and related areas.

Chapter 5

Context-Free Path Query Evaluation

This chapter derives from Medeiros *et al.* [41]. We present our approach for the evaluation of Context-Free Path Queries (CFPQs). Our algorithm takes as input a grammar, a data graph and a query, and recognizes context-free paths in the data graph. The goal of the algorithm is to identify pairs of vertices connected by paths whose traces are strings in the language of the grammar.

In some traditional parsing techniques, *grammar items* guide the parsing process. Those items mark the parsing progress in a given state by adding a dot on the right-hand side of a production rule. The dot separates what has been recognized so far from what is yet to be. In the setting of a path query evaluation, there might be several paths satisfying a given pattern, and therefore several traces to be recognized. Thus, to control the evaluation process, we extend grammar items by (1) replacing the dot by a set of vertices and (2) allowing one set for each position on the right-hand side. The next definition captures this idea.

Definition 11 (Trace Item)**.** *Given a context-free grammar $G = (N, \Sigma, P, S)$ and a data graph D, a* trace item *is a pair formed by a production rule and a function associating a set of graph nodes to each position of the right-hand side of the rule. Formally, a trace item is defined as the pair $(A \rightarrow \alpha, f)$, where $A \rightarrow \alpha \in P$ and $f : \{0, \ldots, |\alpha|\} \rightarrow \mathcal{P}(V)$. Trace items are unique concerning their start vertex and production rule.*

The trace item $(A \rightarrow \alpha_1, \ldots, \alpha_n, f)$, where $f = \{0 \mapsto C_0, \ldots, n \mapsto C_n\}$ will be noted as $[A \rightarrow C_0\ \alpha_1\ C_1\ ...\ \alpha_n\ C_n]$. The sets $C_1, \ldots, C_n$ will be called position sets.

The first position set C_0 in a trace item is a singleton. Given two position sets C_1, C_2 and a grammar symbol α, a sequence $C_1\ \alpha\ C_2$ on the right-hand side of an item indicates that every vertex in C_2 is reachable from at least one vertex in C_1 via an α-derivable path. For instance, the trace item $[\ S \rightarrow \{1\}\ a\ \{2, 3\}\ S\ \{4\}\ b\ \{\ \}\ ]$ indicates that the parsing process is in a stage where a-derivable paths connecting vertex 1 to vertices 2 and 3 in the data graph have been identified. As for vertex 4 in the third position set, since there the second position set contains vertices 2 and 3, there might be an S-derivable path from either or even both of them. The data graph must be consulted to solve such ambiguity.

Our algorithm is defined by a transition system that transforms a set of trace items to parse paths in the graph. Given a grammar G, a data graph D, and a query Q, we can define the *initial* set of trace items as:

$$I_0 = \{[A \rightarrow \{w\}\ \alpha_1\ \{\ \}\ \ldots\ \alpha_n\ \{\ \}] \mid A \rightarrow \alpha_1 \ldots \alpha_n \in P \wedge (w, A) \in Q\},$$

For each pair $(w, A) \in Q$, this set contains one trace item for vertex w and each production rule $A \rightarrow \alpha_1 \ldots \alpha_n \in P$. The trace items contain empty sets for all positions on the right-hand side of the rule, except for the first one. The first position set is the singleton $\{w\}$.

The following rules define a step relation between sets of trace items. These rules are designed to implement the closure of I_0 for given G, D, Q. The notation $I \rightarrow_{G,D} I'$ means that the set of items I becomes I' in one step.

Definition 12 (Trace Item Set Transformation)**.** *Given a grammar G and a data graph D, the relation $I \rightarrow_{G,D} I'$ defines transformation steps between sets of trace items.*

1) *This rule covers the case when we have an l-labeled edge between nodes x and y in D. For adjacent position sets C_1 and C_2 in trace item i, if C_1 contains x and precedes l, we add y to C_2. The substitution notation $I[i'/i]$ is used to define a set of items that is equal to I, but replacing the item i by i'.*

$$\frac{\begin{array}{c} i = [\ A \rightarrow \ldots\ C_1\ l\ C_2\ \ldots\ ] \in I, \qquad l \in \Sigma, \qquad x \in C_1, \\ C_2' = C_2 \cup \{y \mid (x, l, y) \in D\}, \qquad i' = [\ A \rightarrow \ldots\ \ C_1\ l\ C_2'\ \ldots\ ] \end{array}}{I \rightarrow_{G,D} I[i'/i]}$$

Ex.: Given $(1, a, 2), (1, a, 3) \in D$, the trace item $i = [\ S \rightarrow \{1\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$ becomes $i' = [\ S \rightarrow \{1\}\ a\ \{2, 3\}\ S\ \{\ \}\ b\ \{\ \}\ ]$.

2) *This rule creates new trace items to look for B-paths from a node x. For a position set C_1 preceding B, new trace items are created for each production rule of B to look for paths starting at x, if such trace items do not exist yet.*

$$\frac{\begin{array}{c} i = [\ A \rightarrow \ldots\ C_1\ B\ C_2\ \ldots\ ] \in I, \qquad B \rightarrow \beta_1 \ldots \beta_k \in P, \qquad x \in C_1, \\ I' = \{[\ B \rightarrow \{x\}\beta_1 \ldots \beta_k\{\ \}\ ] \mid [B \rightarrow \{x\}\ \beta_1 \ldots \beta_k\ C_k\ ] \notin I\} \end{array}}{I \rightarrow_{G,D} I \cup I'}$$

Ex.: Given $i = [\ S \rightarrow \{1\}\ a\ \{2, 3\}\ S\ \{\ \}\ b\ \{\ \}\ ]$, new trace items $I' = \{[\ S \rightarrow \{2\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ], [\ S \rightarrow \{2\}\ ]\}$ (analogous for trace items starting at vertex 3) are added to I.

3) *This rule identifies a derivation by a non-terminal symbol A, provided that there exists a path between vertices x and y whose trace is a string derivable from A. In this case, vertex y is added to set C_2 in every trace item $[B \rightarrow \ldots\ C_1\ A\ C_2\ \ldots\ ] \in I$, where $x \in C_1$ and $y \notin C_2$.*

$$\frac{\begin{array}{c} i = [\ A \rightarrow \{x\}\ \cdots\ C\ ] \in I, \qquad I' = \{i' \mid i' = [\ B \rightarrow \ldots\ C_1\ A\ C_2\ \ldots\ ] \in I\} \\ I'' = \{i'' \mid i'' = [\ B \rightarrow \ldots\ C_1\ A\ C_2 \cup C\ \ldots\ ]\ \wedge\ [\ B \rightarrow \ldots\ C_1\ A\ C_2\ \ldots\ ] \in I\} \\ C - C_2 \neq \{\ \} \end{array}}{I \rightarrow_{G,D} (I - I') \cup I''}$$

Ex.: Given $i = [\ S \rightarrow \{1\}\ a\ \{2, 3\}\ S\ \{2\}\ b\ \{3\}\ ]$, the trace item $i' = [\ \{3\}\ a\ \{1\}\ S\ \{\ \}\ b\ \{\ \}\ ]$ becomes $i'' = [\ \{3\}\ a\ \{1\}\ S\ \{3\}\ b\ \{\ \}\ ]$.

Let us now present our algorithm using the rules above. This algorithm recognizes paths using trace items. The parsing of traces is performed in an incremental way. For each pair $(v, A) \in Q$, our algorithm identifies all A-derivable paths starting at v.

Let us define I^* as the set of trace items obtained as the closure of the relation $\rightarrow_{G,D}$ for the initial set of trace items I_0:

$$I_0 \rightarrow^*_{G,D} I^*.$$

This closure always exists since the sets that compose D and G are finite and rule 2 above only adds new items to I when they do not exist already in the set. Given a set of trace items I^*, we can now define the data graph:

$$D' = D \cup \{(x, A, y) \mid [\ A \rightarrow \{x\}\ \cdots\ C\ ] \in I^* \wedge y \in C\}.$$

which annotates the data graph D with non-terminal-labeled edges. For query Q, the answer is given by:

$$Eval(Q) = \{y \mid (x, A, y) \in D', (x, A) \in Q\}.$$

The next proposition relates the set of trace items I^* to paths in the data graph D'.

Theorem 5. *Given G, D, Q, and I_0, I^* calculated as above, for each item $i = [\ X \rightarrow \ldots\ C_1\ B\ C_2\ \ldots\ ] \in I^*$ such that $x \in C_1$ and there is a path from x to y in D whose trace is a string $s \in \Sigma^*$ and $B \Rightarrow^* s$, then $y \in C_2$.*

Proof. By induction on the derivation $B \Rightarrow^* s$.

- Base: $B \Rightarrow s$: This means that there exists a rule $B \to s \in P$.

  1. If $s = \varepsilon$, then, by Definition 12, there is an item $[B \to \{x\}] \in I^*$, so $x = y$ since the set $\{x\}$ is, at the same time, the first and last position set of the item, so y is added to C_2 (Rule 12.2).
  2. If $s = \sigma_1 \ldots \sigma_m \in \Sigma^*$, for $m > 0$, then the successive application of Rule 12.1 to item $[B \to \{x\}\sigma_1\{\ \}\ldots\sigma_m\{\ \}]$ traverses the s-generated path from x to y in D, until we have $[B \to \{x\}\sigma_1 \ldots \sigma_m\{y, \ldots\}] \in I^*$. In this way, by rule 3, when $x \in C_1$, we have $y \in C_2$.

- Inductive hypothesis: The property holds for all derivations with $k < n$ steps: for each item $i' = [\ X' \to \ldots\ C'_1\ B'\ C'_2\ \ldots\ ] \in I^*$ such that $x' \in C'_1$ and there is a path from x' to y' in D whose trace is a string $s' \in \Sigma^*$ and $B' \Rightarrow^k s'$, then $y' \in C'_2$.

- Inductive step: $B \Rightarrow^n s$, with $n > 1$. This means a production rule $B \to \beta_1 \ldots \beta_j \in P$ was applied as the first step of the derivation, so that we have $B \Rightarrow \beta_1 \ldots \beta_j \Rightarrow^{n-1} s$. For simplicity, we suppose that this is a leftmost derivation of s, from B, which means that we can decompose $B \Rightarrow^n s$ into

  $$B \Rightarrow \beta_1 \ldots \beta_j \Rightarrow^* s_1\beta_2 \ldots \beta_j \Rightarrow^* \cdots \Rightarrow^* s_1 \ldots s_{j-1}\beta_j \Rightarrow^* s_1 \ldots s_j$$

  where for each $1 \leq m \leq j$, derivations $\beta_m \Rightarrow^* s_m$ have length less than n and $s = s_1 \ldots s_j$. Notice that the substrings $s_1 \ldots s_j$ define subpaths of the s-path from x to y via intermediate vertices in D.

  As $x \in C_1$ in trace item i, we must have an item $i' \in I^*$ such that $i' = [B \to \{x\}\beta_1 C_1 \ldots C_{j-1}\beta_j C'_j] \in I^*$ (recall that, during the construction of I', if such an item does not belong to the current set of trace items, rule 2 in Definition 12 adds it to the set of trace items).

  Given the item $i' = [B \to \{x\}\beta_1 \ldots \beta_j C'_j] \in I^*$, and as $\beta_1 \Rightarrow^* s_1$ (with a derivation of length less than n), the induction hypothesis ensures that the vertex at the end of the s_1-path will be in position set C_2.

  Using a similar reasoning for each symbol of the right-hand side of the rule $B \to \beta_1 \ldots \beta_j$, we can see that $y \in C_j$.

  By Rule 12.3, we can conclude that $y \in C_2$ in i.

In this way, we can conclude that $y \in C_2$ for any derivations $B \Rightarrow^* s$. ☐

The next proposition shows that our algorithm is consistent, *i.e.*, for every pair of vertices $x, y \in D$, if they belong to successive position sets in any trace item in I^*, then y is G-reachable from x.

Theorem 6. *Given $i = [\ A \to \ldots\ C_1\ \alpha\ C_2\ \ldots\ ] \in I^*$, such that $\alpha \in N \cup \Sigma$, $x \in C_1$ and $y \in C_2$ then $(x, \alpha, y) \in \mathcal{R}_{G,D}$ (see Definition 5).*

Proof. By induction on the sequence of steps in the closure $I_0 \Rightarrow^* I^*$ to obtain i.

- Base: If i was created in the construction of I^0 and was not changed or substituted during the calculation of I^*, then i must be of the form $[A \to \{x\}]$, for a given $A \to \varepsilon \in P$. Otherwise, i would have been substituted by another item during the calculation of I^*. By Definition 5.2, we have $(x, A, x) \in \mathcal{R}_{G,D}$.

- Inductive Hypothesis: For any $k < n$, if $i = [\ A \to \ldots\ C_1\ \alpha\ C_2\ \ldots\ ]$, such that $\alpha \in N \cup \Sigma$, $x \in C_1$ and $y \in C_2$, was built in k steps, then $(x, \alpha, y) \in \mathcal{R}_{G,D}$.

- Inductive Step: Let $i = [\ A \to \ldots\ C_1\ \alpha\ C_2\ \ldots\ ]$, such that $\alpha \in N \cup \Sigma$, $x \in C_1$ and $y \in C_2$, was built in n steps (and will not be further changed by the calculation of I^*).

  The trace item i was built by using one of the three rules of Definition 12.

  1. Case i was built by Rule 12.1: In this case, i has the form $[\ A \to \ldots\ C_1\ \alpha\ C'_2\ \ldots\ ]$, where $C'_2 \supseteq \{y \mid (x, l, y) \in D\}$, so $(x, l, y) \in \mathcal{R}_{G,D}$ by Definition 5.1.

2. Case i was built by Rule 12.2: This case is similar to the base case.
3. Case i was built by Rule 12.3: In this case, $i = [\ B \to \dots\ C_1\ A\ C_2 \cup C\ \ \dots\ ] \in I''$ of Rule12.3, which adds vertices $y \in C$ to a previously computed item, provided that there exists an (also previously computed) item $[A \to \{x\} \dots C]$. By the induction hypothesis, we have that $(x, A, y) \in \mathcal{R}_{G,D}$.

From the items above, we can conclude that the property holds. ☐

Corollary 6.1. *Given* G, D, Q*,*

$$(x, s, y) \in D' \iff (x, s, y) \in \mathcal{R}_{G,D}$$

Proof. Immediate from Theorems 5, and 6. ☐

5.1 The Trace-Item-Based Algorithm

Let us now present Algorithm 1 for the evaluation of CFPQs based on trace items. This algorithm uses special marks $^\bullet$ and $^\circ$, respectively, for processed and unprocessed vertices in position sets, to keep track of what vertices have already been processed. We omit these marks when such distinction is unnecessary. The $\Join$ operator is used to perform unions between sets of processed and unprocessed vertices, and is defined as:

$$C \Join \{x^\circ\} \quad = \quad \begin{cases} C, & \textit{if } x^\bullet \in C \\ C \cup \{x^\circ\}, & \textit{otherwise} \end{cases}$$

where C is a position set and x° is an unmarked vertex. If vertex x has already been processed, it is conserved marked in position set C. Otherwise, it is added unmarked.

ALGORITHM 1 : The Trace-Item-based Algorithm [40]

```
   Input : A grammar G, a query Q, and a graph D
   Output : An annotated graph D' ⊆ V × (Σ ∪ N) × V and a set of trace items I
1  function eval
2      I := {[A → {w°} α1 { } ... αn { }] | A → α1...αn ∈ P ∧ (w, A) ∈ Q}
3      D' := D
4      while ∃i, x s.t. i = [A → ... {x°, ...} ...] ∈ I do
5          switch i
6          case i = [A → ... {x°, ...} αk Ck ...] do
7              if αk ∈ Σ ∨ [αk → {x} ...] ∈ I then
8                  Ck := Ck ⋈ {y° | (x, αk, y) ∈ D'}
9              else
10                 I := I ∪ {[αk → {x°} β1 { } ... βn { }] | αk → β1 ... βn ∈ P}
11         case i = [A → {w} ... {x°, ...}] do
12             D' := D' ∪ {(w, A, x)}
13             foreach [B → ... {w•, ...} A C ...] ∈ I do
14                 C := C ⋈ {x°}
15         mark(x, i)
16     return D', I
```

Algorithm 1 manipulates two data structures: a set of trace items I and a data graph D' containing the original data graph D incrementally annotated with new, non-terminal-labeled edges. To compute the answers to queries in Q, the algorithm starts by creating trace items from its pairs of vertices and grammar rules: for each pair $(v, A) \in Q$, it creates one trace item for each production rule of non-terminal A with v in its first position set (line 2). That prepares the algorithm to enter the main loop that processes unmarked vertices in items in I.

There are two cases when processing unmarked vertices:

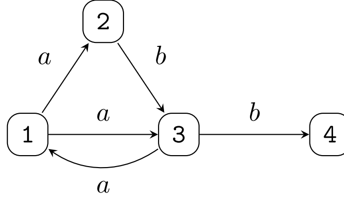


Figure 5.1: Example Graph (reproduction of Figure 2.1).

1. In the first case of the main loop (lines 6-10), given the trace item $i = [A \to C_0\ \alpha_1\ C_1\ ...\alpha_n\ C_n]$, x° belongs to a position set C_{k-1} that is not in the last position of the trace item. There are three sub-cases:

   (a) If $\alpha_k \in \Sigma$, all vertices y° such that there exists an edge $(x, \alpha_k, y) \in D'$ are added to C_k (line 8);

   (b) If $\alpha_k \in N$ and $[\ \alpha_k \to \{x\} \dots\ ] \in I$, all vertices y° such that there is an edge $(x, \alpha_k, y) \in D'$ are added to C_k (also at line 8);

   (c) If $\alpha_k \in N$ and there is no trace item $[\ \alpha_k \to \{x\ ...\}\ ...\ ]$, Algorithm 1 initiates the search for α_k-derivations beginning at x. This is done by creating new trace items $\alpha_k \to \{x^\circ\} \dots$ and adding them to I (line 10).

2. In the second case of the main loop (lines 11 to 14), vertex x belongs to the last position set of a trace item. The trace item $i = [A \to \{w\} \dots \{x^\circ, ...\}]$ states that there is a path from vertex w to x in D'. So, Algorithm 1 generates an A-labeled edge connecting those two vertices (line 12). After this operation, all position sets C such that $[\ B \to \dots \{w, \dots\}\ A\ C \dots\ ] \in I$ are updated with x° (line 14).

At the end of the main loop's body, vertex x° is marked (line 15). The stop condition of that loop is the absence of unmarked vertices in all position sets. The annotated graph D' is returned at the end (line 10). Let us now explain this process with an example.

Example 1. *Let us consider a grammar G with production rules $P = \{S \to a\ S\ b, S \to \varepsilon\}$ and the data graph given in Figure 5.1.*

Given the query $Q = \{(1, S), (3, S)\}$, our algorithm goes through paths starting at vertices 1 and 3 whose trace is generated by S. In this way, all the production rules of S will be investigated for paths starting at each of these vertices.

We start the parsing process by creating trace items. For each query pair $(v, A) \in Q$, we create one trace item for each production rule of A with v in its first position set. For the query Q we build the trace items:

$$[\ S \ \to\ \{1^\circ\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ] \tag{5.1}$$
$$[\ S \ \to\ \{1^\circ\}\ ] \tag{5.2}$$
$$[\ S \ \to\ \{3^\circ\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ] \tag{5.3}$$
$$[\ S \ \to\ \{3^\circ\}\ ] \tag{5.4}$$

Our algorithm picks unprocessed vertices in an arbitrary order. Let us start with vertex 1 from trace item (5.1). This vertex appears in a position set before the terminal symbol a. We must walk from vertex 1 to all its neighbors linked by an a-labeled edge in D. The neighbor vertices 2 and 3 must then be added to the next position set in the trace item. Doing so, our item will become $[\ S \to \{1^\bullet\}\ a\ \{2^\circ, 3^\circ\}\ S\ \{\ \}\ b\ \{\ \}\ ]$. Notice that vertex 1° has changed to $1^\bullet$ to signal that it has been processed. New vertices are added as unprocessed by using the mark $^\circ$.

Now we may pick vertex 2 for the next step. This vertex is in a position set before the non-terminal symbol S. That indicates that we have to look for S-derivable paths starting at vertex 2. We build the

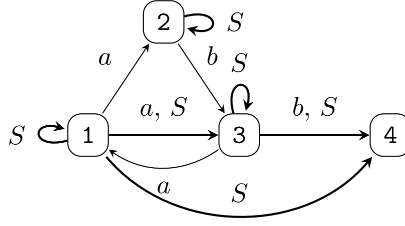


Figure 5.2: Result graph for the query of Example 1.

following new items:

$$[\ S \quad \to \quad \{2^\circ\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ] \tag{5.5}$$
$$[\ S \quad \to \quad \{2^\circ\}\ ] \tag{5.6}$$

Now item (5.1) becomes $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{\ \}\ b\ \{\ \}\ ]$ *and we have to pick another vertex to process. Picking vertex* 2 *from item (5.5), we verify that there is no a-labeled edge going from vertex* 2 *to any other vertex in the graph. That means that there is no a-derivable path from this vertex. Item (5.5) then becomes* $[\ S \to \{2^\bullet\}\ a\ \{\}\ S\ \{\}\ b\ \{\}\ ]$.

Let us now pick vertex 2 *from item (5.6). This item was built from an* ε*-rule. As vertex* 2 *belongs to the first and last position set of this item, that means there is an* S*-derivable path from vertex* 2 *to itself (the empty path). So, we augment the data graph with an* S*-labeled edge. The new edge is the loop right to vertex* 2 *in Figure 5.2 (new edges are shown in* ***bold****).*

Now, item (5.6) becomes $[\ S \to \{2^\bullet\}\ ]$*. The addition of the new,* S*-labelled edge to the data graph triggers a modification to the existing items: we add the unprocessed vertex* 2 *to any position set* C *appearing in a trace item matching the pattern* $[\ \ldots \{2, \ldots\}\ S\ C\ \ \ldots\ ]$*. In our case, item (5.1) becomes* $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\circ\}\ b\ \{\ \}\ ]$.

We may now pick the newly added vertex 2° *in item (5.1). Now we have a vertex in a position set before the terminal b. As we did before, we look for b-labeled edges going out from* 2 *in the data graph. There is only one such edge, which arrives at vertex* 3*. Item (5.1) then becomes* $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet\}\ b\ \{3^\circ\}\ ]$.

Now we pick the newly added vertex 3 *in the last position set of item (5.1). As this vertex is in the last position set of the item, we infer that there is an* S*-valid path from vertex* 1 *to vertex* 3*. As* $(1, S) \in Q$*, we have found one answer for our query. Item 5.1 then becomes* $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet\}\ b\ \{3^\bullet\}\ ]$*. Then, the data graph is augmented with a new* S*-labelled edge from* 1 *to* 3*, which is shown in Figure 5.2.*

This process is repeated until there are no more unprocessed vertices. The complete step-to-step process is presented in Table 5.1 and will result in the following set of items:

$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\bullet\}\ S\ \{2^\bullet, 3^\bullet, 4^\bullet\}\ b\ \{3^\bullet, 4^\bullet\}\ ]$, $[\ S \to \{1^\bullet\}\ ]$,
$[\ S \to \{2^\bullet\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$, $[\ S \to \{2^\bullet\}\ ]$,
$[\ S \to \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{1^\bullet, 3^\bullet, 4^\bullet\}\ b\ \{4^\bullet\}\ ]$, $[\ S \to \{3^\bullet\}\ ]$

The solutions computed by our algorithm are shown as ***bold*** *arrows, labeled by non-terminals, in Figure 5.2. Those arrows connect the node in the first position set of each trace item to the nodes in their last position set.*

5.2 Complexity

In this section, we analyze the time and space complexity of Algorithm 1.

Theorem 7 (Worst-Case Space Complexity)**.** *The worst-case space complexity of Algorithm 1 is* $\mathcal{O}(|V|^2 \cdot |P| \cdot k)$.

Proof. The maximum size that D' and I may reach is:

D': The algorithm increments graph D' with non-terminal-labeled edges, so it uses at most:

$$|D'| \quad = \quad |V| \cdot |N \cup \Sigma| \cdot |V| \tag{5.7}$$

#	Operation	Updated items
1	line 2	$[\ S \to \{\underline{1^\circ}\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$, $[\ S \to \{\underline{1^\circ}\}\ ]$, $[\ S \to \{\underline{3^\circ}\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$, $[\ S \to \{\underline{3^\circ}\}\ ]$
2	line 8	$[\ S \to \{\underline{1^\bullet}\}\ a\ \{\underline{2^\circ}, \underline{3^\circ}\}\ S\ \{\ \}\ b\ \{\ \}\ ]$
3	line 10	$[\ S \to \{1^\bullet\}\ a\ \{\underline{2^\bullet}, 3^\circ\}\ S\ \{\ \}\ b\ \{\ \}\ ]$, $[\ S \to \{\underline{2^\circ}\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$, $[\ S \to \{\underline{2^\circ}\}\ ]$
4	line 8	$[\ S \to \{\underline{2^\bullet}\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ]$
5	lines 12, 14	$[\ S \to \{\underline{2^\bullet}\}\ ]$, $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{\underline{2^\circ}\}\ b\ \{\ \}\ ]$
6	line 8	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{\underline{2^\bullet}\}\ b\ \{\underline{3^\circ}\}\ ]$
7	lines 12, 14	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet\}\ b\ \{\underline{3^\bullet}\}\ ]$
8	lines 12, 14	$[\ S \to \{\underline{1^\bullet}\}\ ]$
9	lines 12, 14	$[\ S \to \{\underline{3^\bullet}\}\ ]$ $[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet, \underline{3^\circ}\}\ b\ \{3^\bullet\}\ ]$
10	line 8	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet, \underline{3^\bullet}\}\ b\ \{3^\bullet, \underline{4^\circ}\}\ ]$
11	lines 12, 14	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\circ\}\ S\ \{2^\bullet, 3^\bullet\}\ b\ \{3^\bullet, \underline{4^\bullet}\}\ ]$
12	line 8	$[\ S \to \{\underline{3^\bullet}\}\ a\ \{\underline{1^\circ}\}\ S\ \{\ \}\ b\ \{\ \}\ ]$
13	line 8	$[\ S \to \{3^\bullet\}\ a\ \{\underline{1^\bullet}\}\ S\ \{\underline{1^\circ}, \underline{3^\circ}, \underline{4^\circ}\}\ b\ \{\ \}\ ]$
14	line 8	$[\ S \to \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{1^\circ, 3^\circ, \underline{4^\bullet}\}\ b\ \{\ \}\ ]$
15	line 8	$[\ S \to \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{1^\circ, \underline{3^\bullet}, 4^\bullet\}\ b\ \{\underline{4^\circ}\}\ ]$
16	lines 12, 14	$[\ S \to \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{1^\circ, 3^\bullet, 4^\bullet\}\ b\ \{\underline{4^\bullet}\}\ ]$
17	line 8	$[\ S \to \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{\underline{1^\bullet}, 3^\bullet, 4^\bullet\}\ b\ \{4^\bullet\}\ ]$
18	line 10	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, \underline{3^\bullet}\}\ S\ \{2^\bullet, 3^\bullet, \underline{4^\circ}\}\ b\ \{3^\bullet, 4^\bullet\}\ ]$
19	line 8	$[\ S \to \{1^\bullet\}\ a\ \{2^\bullet, 3^\bullet\}\ S\ \{2^\bullet, 3^\bullet, \underline{4^\bullet}\}\ b\ \{3^\bullet, 4^\bullet\}\ ]$

Table 5.1: Step-by-step behavior of Algorithm 1 (underlined vertices mean they were either marked or added on that step).

what is $\mathcal{O}(|V|^2 \cdot |N \cup \Sigma|)$.

I: Set I contains generalized items, which are annotated production rules with a single vertex at the start of the right-hand side. So we have at most:

$$|I| \quad = \quad |V| \cdot |P| \tag{5.8}$$

For each trace item, the number of position set sets depends on the size of the right-hand side of a production rule. Assuming that k denotes the greatest size of the right-hand side of the rules in G, each trace item may have k position sets of size at most $|V|$ (notice that the first position set on each trace item is a singleton).

In this context, the worst-case space complexity for I is:

$$|V| \cdot |P| \cdot k \cdot |V|.$$

what is $\mathcal{O}(|V|^2 \cdot |P| \cdot k)$.

We can now estimate the worst-case space complexity as:

$$\mathcal{O}(|V|^2 \cdot (|N \cup \Sigma| + |P| \cdot k)) \tag{5.9}$$

□

Theorem 8 (Worst-Case Time Complexity)**.** *The worst-case time complexity of Algorithm 1 is* $\mathcal{O}(|V|^3 \cdot |P|^2 \cdot k^2)$.

Proof. The main loop iterates until there are no more unmarked vertices x°. The maximum number of unmarked vertices is $|I| \cdot k \cdot |V|$, where k is the maximum number of possible position sets for rules of the grammar (the greatest size of a right-hand side of the rules in G). So, as $|I| = |V| \cdot |P|$, we have at most $|V|^2 \cdot |P| \cdot k$ possible vertices x°.

For each iteration, the form of trace item i guides the operation to be performed. The tests at lines 6 and 11 have constant cost.

There are two cases to be considered inside the **switch** command:

- The evaluation of the condition at line 7 requires searching over the set of trace items I. The cost of this operation is constant (supposing that we use a matrix representation).

  Line 8 is the case where the algorithm advances one step on a path by looking for edges $(x, \alpha, y) \in D'$. As there are at most $|V|$ possible destination vertices, the algorithm performs at most $|V|$ operations in this case.

  At line 10, the algorithm adds new trace items to I in order to start a new derivation. This line ensures that the algorithm only creates at most one trace item for each production rule in G for a fixed vertex x. So, in this case, the algorithm performs at most $|P|$ constant time operations.

  In this way, the overall cost of the case spanning from line 6 to 10 is bounded by $\max(|V|, |P|)$.

- The second case of the **switch** command adds non-terminal-labeled edges to the graph. Such edges are created at line 12 in constant time.

  The appearance of a new edge triggers the update of position sets by the iteration at line 13. We have at most $|V| \cdot |P| \cdot k$ position sets. Assuming, again, a matrix representation, locating each set C in a trace item, requires constant time. Thus, line 14 will be executed $|V| \cdot |P| \cdot k$ times in the worst case.

  In this way, the overall cost of the case spanning from line 11 to 14 is bounded by $|V| \cdot |P| \cdot k$.

Therefore, the worst-case time complexity of Algorithm 1 is $\mathcal{O}(|V|^3 \cdot |P|^2 \cdot k^2)$. □

5.3 Experiments

In this section, we analyze the viability of our technique in a series of experiments. Each of these experiments is designed to allow us to verify the behavior of the Algorithm 1 in general and specific situations. Our experiments are built to observe its time and memory consumption as data size scales.

We have five implementations that differ in the following aspects: (*i*) the implemented algorithm; (*ii*) programming language; (*iii*) data representation; (*iv*) grammar form supported; and (*v*) support for nested expressions and navigational axes. They are summarized in Table 5.2. The source-code as well as the datasets used in the experiments described in this section are publicly available in our online repository[1].

ID	Algorithm	Programming Language	Data Representation	Grammar Form	Support for Nested Expressions and Navigational Axes
I1	Algorithm 1	Go	Bit-arrays	Any	Yes
I2	Algorithm 1	Go	Hash Tables	Any	Yes
I3	Algorithm 1	Python	Hash Tables	Any	No
I4	LL-based [39]	Python	Hash Tables	NF	No
I5	CYK-based [26]	Python	Hash Tables	NF	No

Table 5.2: Implementations used in the experiments.

We implemented two versions of Algorithm 1 in the Go programming language: I1 uses a lower-level (bit-array) representation, while I2 uses a higher-level representation (native hash tables). Both versions extend Algorithm 1 with support for navigational axes and nested expressions [45]. The remaining implementations are all written in Python and use native hash tables to index triples. To allow us to analyze the influence of the programming language and present fair comparisons with the algorithms proposed by Medeiros *et al.* [39] and Hellings [26], we implemented Algorithm 1 in Python, named I3. We include an implementation for the LL-based algorithm [39], named I4, and an implementation for a standard CFPQ evaluation algorithm, named I5. The implementations I4 and I5 only accept grammars in Normal Form.

In the next sections, we present experiments for evaluating the performance of all implementations. Since I4 and I5 only accept grammars in Normal Form, they only present performance data in experiments with such grammars. We observe how performance is affected by: (*i*) underlying data structures; (*ii*) characteristics of the grammar, such as number of symbols, ambiguity or length of its production rules; *(iii)* density, topology and size of the data graph. Unless otherwise stated, the queries used in our setting are defined as $Q = \{(x, S) \mid x \in V\}$, where V is the set of vertices in the input graph, and S is the start symbol of the grammar. Notice that this query makes the algorithms start at *every vertex* in the graph and follow all S-derivable paths to find the reachable vertices.

All the experiments were performed on a Ubuntu 20.04, 8GB RAM, Intel Core i5-8250U CPU @1.60GHz $\times$ 8, 64 bits. The results presented in the next sections correspond to the average time and memory consumption of 10 runs to minimize the interference of the operation system. Missing values indicate that the experiment was aborted due to a timeout of 10 minutes.

5.3.1 Ambiguity and Normal Form

This experiment evaluates how the topology of the graph database, as well as grammar form and ambiguity, impact performance. We consider linear and complete graphs, as in Medeiros *et al.* [39]. In complete graphs, all vertices connect to all the others, including itself, by a- and b-edges (that is, a graph with n vertices has n^2 a-labeled edges and n^2 b-labeled edges). Linear graphs with n vertices define paths whose traces are of form $a^{\frac{n}{2}} b^{\frac{n}{2}}$.

In the queries of this experiment, we look for paths formed by nested, balanced pairs of a and b edges. We define four grammars that generate the same language. They are given as G_1, G_2, G_3 and G_4

[1] Online git repository containing the source-code and datasets used in the experiments: `https://gitlab.com/ciromoraismedeiros/rdf-ccfpq`.

in Table 5.3. Grammars G_1 and G_3 are ambiguous, where G_3 is in Chomsky Normal Form. Grammars G_2 and G_4 are unambiguous, where G_4 is in Chomsky Normal Form.

Grammar	Production Rules	Description
G_1	$S \to S\ S \mid a\ S\ b \mid \varepsilon$	Ambiguous grammar. Generates balanced pairs of a's and b's [24, 8, 61, 39]
G_2	$S \to a\ S\ b\ S \mid \varepsilon$	Unambiguous grammar, generates the same language as G_1 [24, 8, 61, 39]
G_3	$S \to S\ S \mid a\ A \mid \varepsilon$, $A \to S\ b$	Ambiguous grammar in Chomsky Normal Form, generates the same language as G_1.
G_4	$S \to a\ A \mid \varepsilon$, $A \to S\ B$, $B \to b\ S$	Unambiguous grammar in Chomsky Normal Form, generates the same language as G_1.

Table 5.3: Summary of grammars used for the experiments with complete and linear graphs.

Complete Graphs. Figure 5.3 shows time (left column) and memory (right column) consumption of all implementations for complete graphs.

In this experiment, I3 presented better results when compared to I4and I5. Results for I5 are not depicted because it was not able to process the minimum of 50 vertices. On the other hand, I4 timed out for 100 vertices.

Concerning the programming language, Go (I2) performed far better than Python (I3) for grammars G_3 and G_4. Notice that the execution of I3 was interrupted for more than 100 vertices due to the timeout of 10 minutes. For grammars G_1 and G_2, the difference of performance was not noticeable both for time and memory.

Implementations I1 and I2 explore different representations of data. Notice that I1 outperforms I2, indicating the saving of time and memory of the bit-array representation.

We conclude that for grammars not in Normal Form (G_1, G_2), the ambiguity reduced time and memory consumption. We believe that this fact is a consequence of the smaller size of the right-hand side of production rules. The smaller the number of symbols on the right-hand side, the smaller the number of position sets to be maintained. The ambiguity of grammar has not shown a visible impact on the performance with Normal Form grammars (G_3, G_4), given they have the same number of production rules. Notice that grammars in Normal Form (G_3, G_4) present better results both in terms of time and memory consumption.

Linear Graphs. Time (left column) and memory (right column) consumption for linear graphs are shown in Figure 5.4.

All implementations presented linear time and memory consumption, as expected due to the structure of linear graphs and the language being recognized. Algorithm 1 (I3) presented the best time performance. Python (I3) clearly outperformed Go (I2) in all cases. The bit-array representation (I1) performed slightly worse than the hash table representation (I2). This was expected due to the sparseness of linear graphs.

Different from what we observed for complete graphs, performance was negatively affected by both grammar ambiguity and normal form. Linear graphs in CFPQs simulate strings in the context of compilers, where larger and ambiguous grammars tend to negatively affect performance. The explanation comes from the fact that the structure of linear graphs preclude the algorithms from taking advantage of using such grammars to find alternative shorter paths.

5.3.2 Length of Derivations

In this experiment, we use two grammars to investigate the impact of the number of derivation steps on the algorithms' performance. Grammars G_5 and G_6 [27] in Table 5.4 describe, respectively, the languages generated by the regular expressions a^+ and a^*. Notice that, for non-empty strings, the number of steps required to generate the string using grammar G_5 is smaller than that of grammar G_6. These grammars are respectively referred to as 'dense' and 'sparse' by Hellings [27].

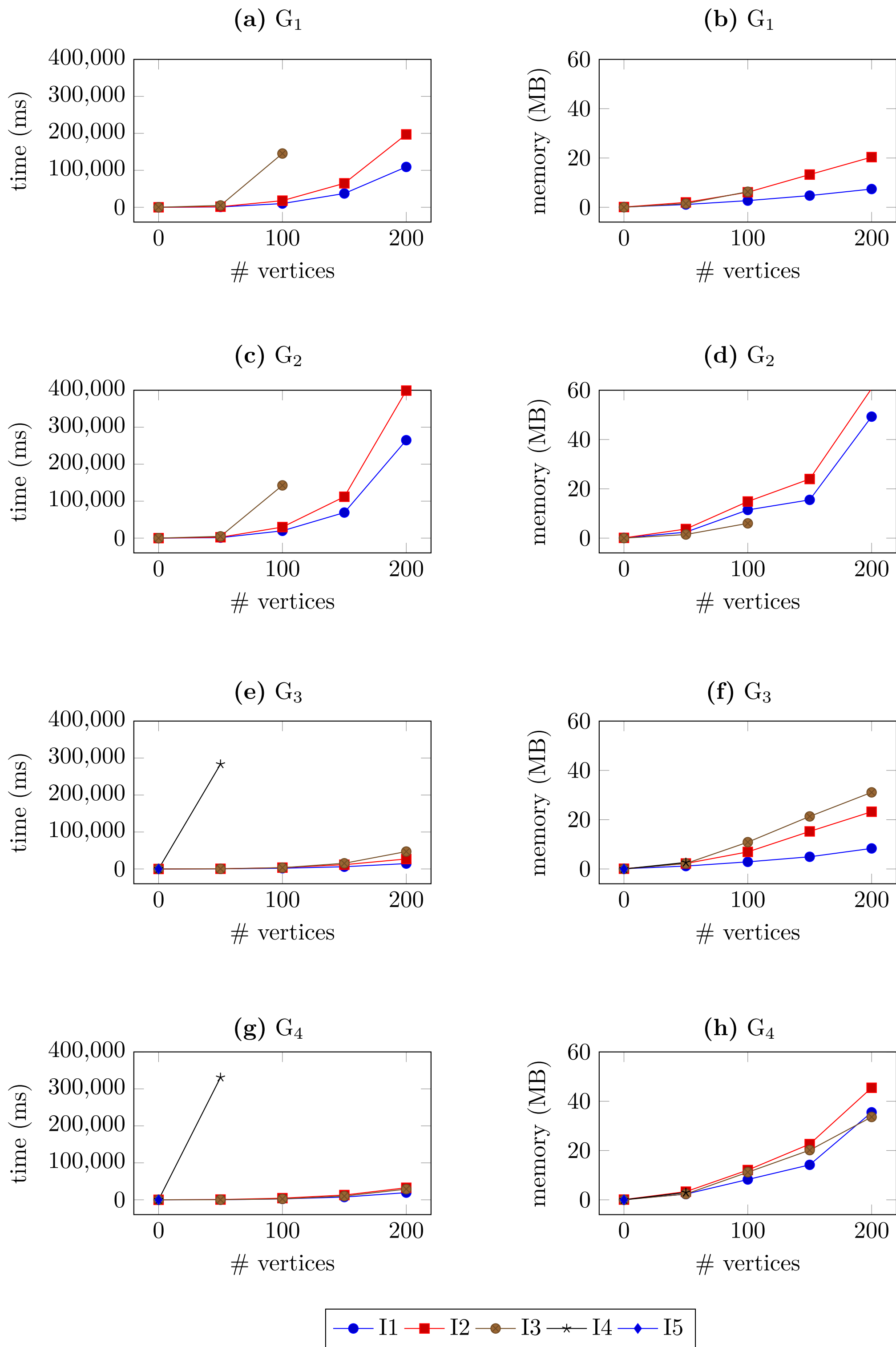


Figure 5.3: Experiments with *ab*-complete graphs and grammars G_1, G_2, G_3 and G_4.

Grammar	Production Rules	Description
G_5	$A \to A\ A \mid a$	Dense grammar recognizing the language given by a^+ [27]
G_6	$B \to B\ A \mid A\ B \mid \varepsilon$, $A \to a$	Sparse grammar recognizing the language given by a^* [27]

Table 5.4: Dense and Sparse grammars

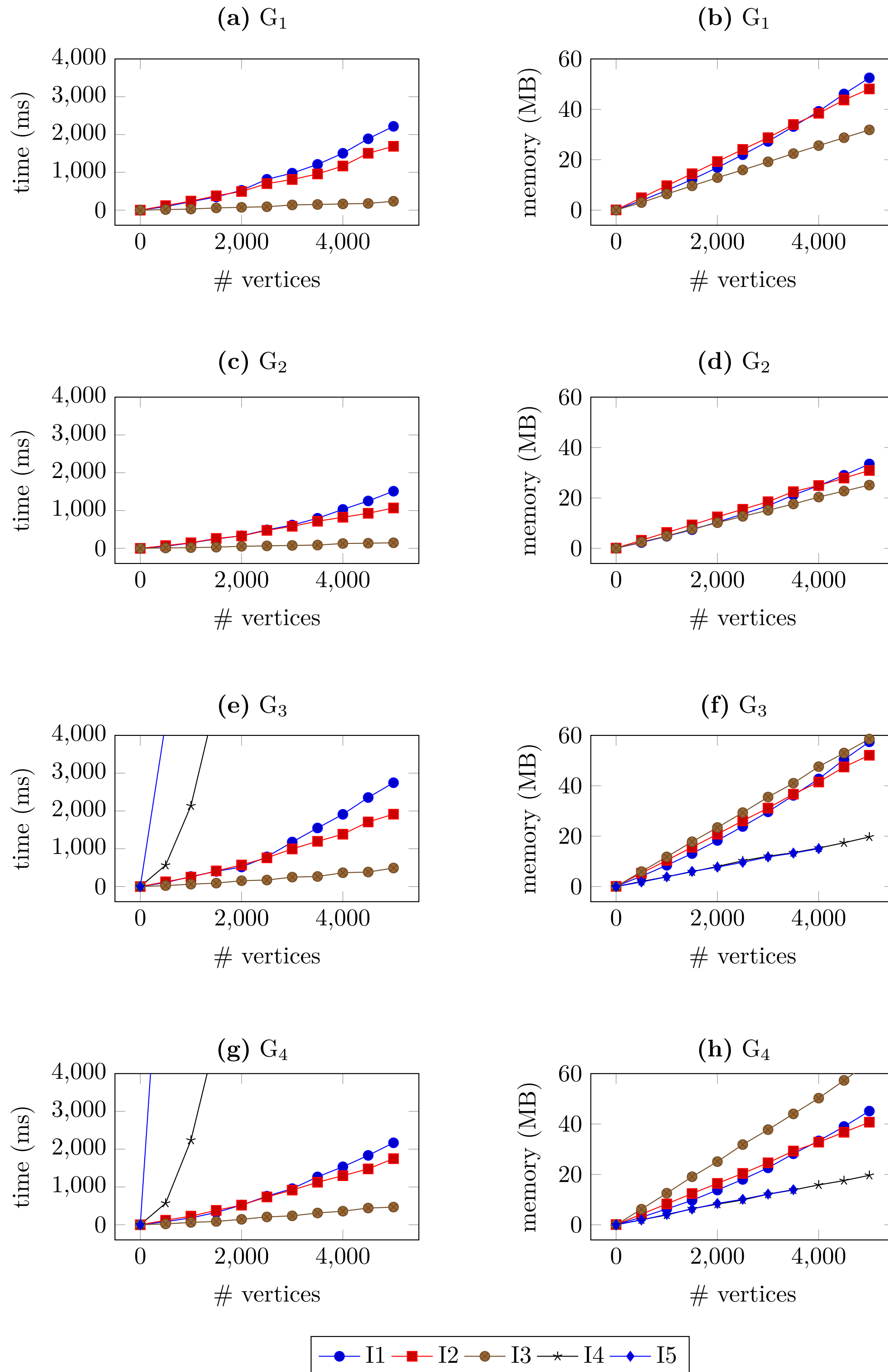


Figure 5.4: Experiments with *ab*-linear graphs using grammars G_1, G_2, G_3, G_4.

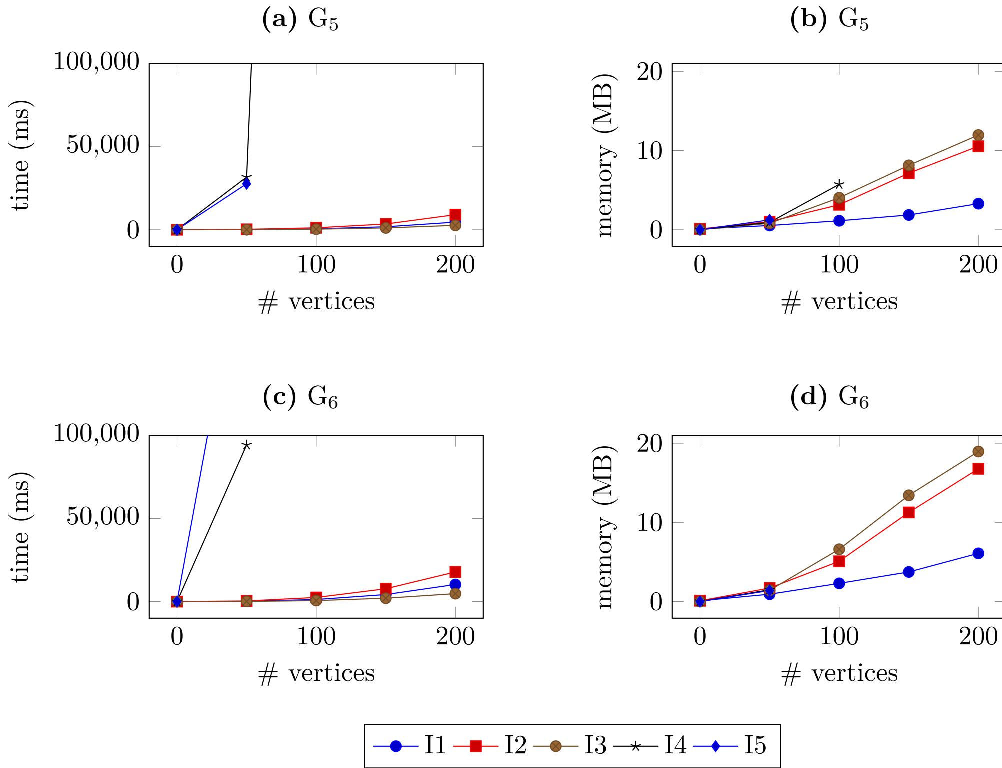


Figure 5.5: Experiments with a-complete graphs and grammars G_5 and G_6.

Querying a-Complete Graphs. Figure 5.5 shows time (left column) and memory (right column) consumption of all implementations when using grammars G_5 and G_6 on complete graphs containing a-labeled edges only. We notice that Algorithm 1 (I3) showed good scalability. The time and memory consumption of I4 and I5 prevents us from exploring graphs with more than 100 vertices for grammar G_5 and 50 vertices for grammar G_6. Python (I3) performed slightly better than Go (I2). The bit-array representation presented better memory consumption than the hash table representation. This was expected, since the bit-array representation is tuned for working with denser graphs. The use of G_5 provided better time and memory performance than that of G_6, since the former takes less derivation steps and has fewer production rules than the latter.

Querying a-Cycle and a-Path Graphs. The behavior of the implementations concerning a-cycle and a-path graphs is presented in Figures 5.6 and 5.7. In this experiment, we use grammars G_5 and G_6 to query cycle and linear graphs whose edges are labeled a. Implementation I3 performed better than I4 and I5, which timed out for medium-size or even small graphs. For both cycle and linear graphs, Go (I2) provided better performance using G_5, whilst Python (I3) provided better performance using G_6. Although both linear and cycle graphs are sparse, they grow non-linearly in size as the algorithms compute more answers. That made it possible for I1 to perform better than I2. Opposingly to the previous case, with a-complete graphs, the use of G_5 provided worse results than G_6. This might indicate that G_5 is more adequate for denser graphs, while G_6 is more adequate for sparser graphs.

5.3.3 Synthetic Social Networks

The next experiments were proposed in [29] and use random, synthetic graphs that simulate social networks. Given the order n of the graph and a constant $k \leq n$, the generator function $\mathcal{G}(n, k)$ starts with a clique of k vertices and iteratively adds lots of k edges to the graph until the number n of vertices is reached [3]. The edge labels are randomly chosen a, b, c or d. The probability of an edge being added to a destination vertex increases with the in-degree of that vertex. In this way, more popular vertices tend to gain more connections than less popular ones.

The grammars for this experiment are presented in Table 5.5. Both grammars G_7 and G_8 define the same language, being G_8 in Normal Form.

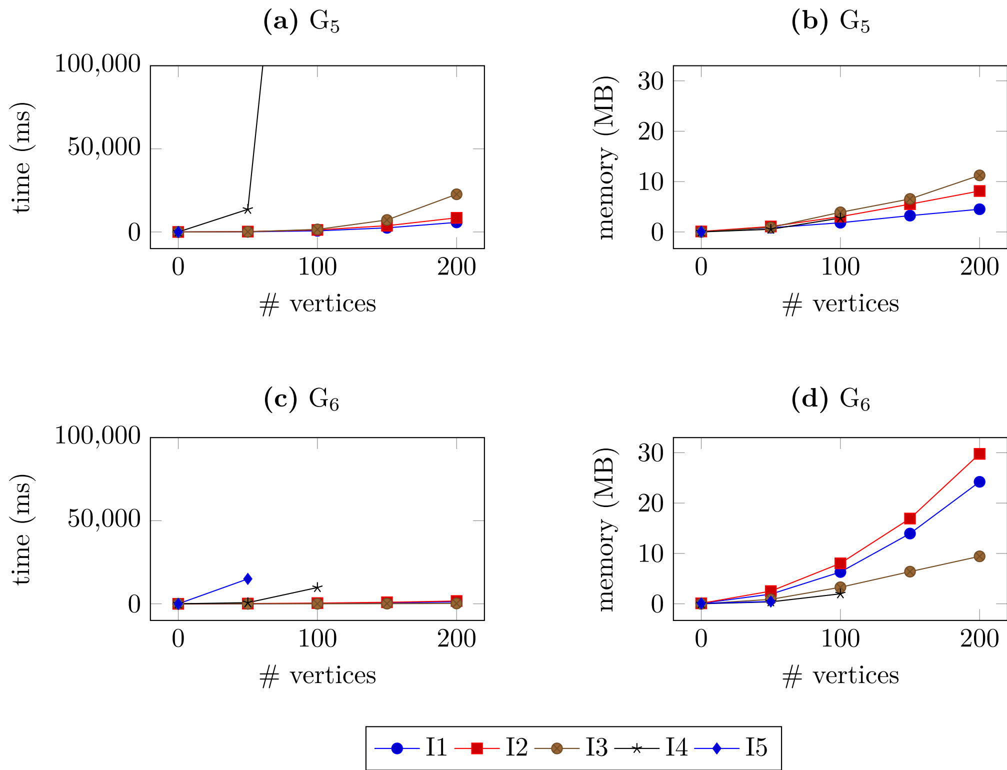


Figure 5.6: Experiments with a-cycle graphs grammars G_5 and G_6.

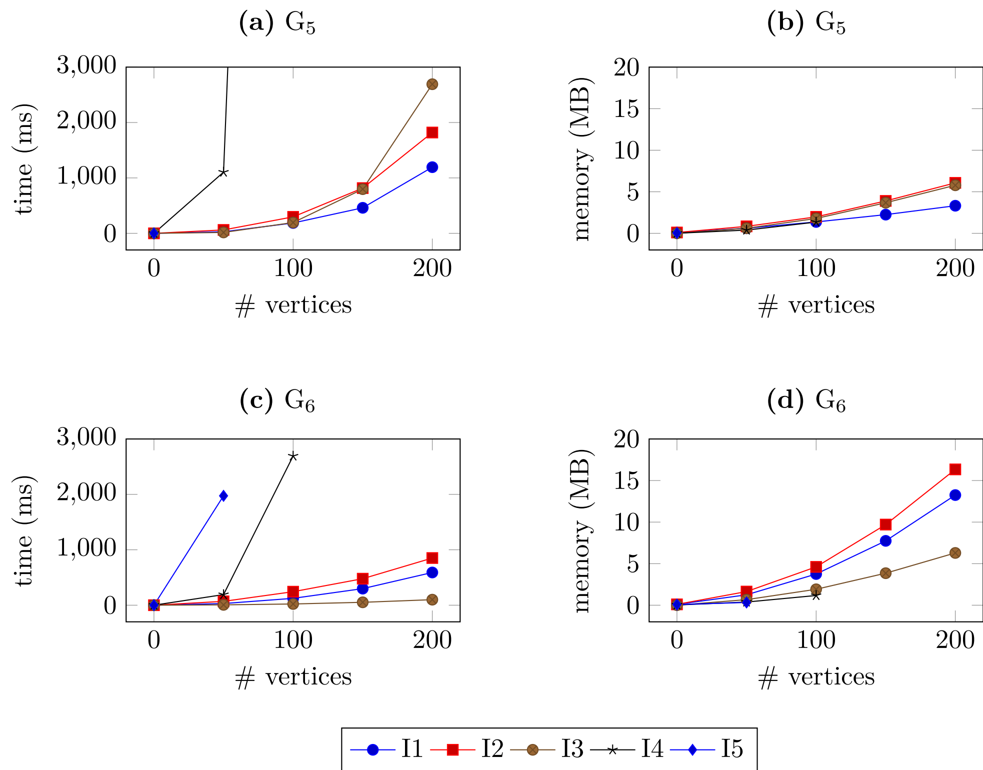


Figure 5.7: Experiments with a-path graphs grammars G_5 and G_6.

Grammar	Production Rules	Description
G_7	$S \to a\ S\ d \mid a\ X\ d$, $X \to b\ X\ c \mid \varepsilon$	Grammar generating the language $a^n b^m c^m d^n$ [29]
G_8	$S \to a\ W \mid a\ Y$, $W \to S\ d$ $Y \to X\ d$ $X \to b\ Z \mid X\ c \mid \varepsilon$	Normal form grammar generating $a^n b^m c^m d^n$

Table 5.5: Grammars for querying a simulated social network.

In Figures 5.8 and 5.9, we present time (left column) and memory (right column) consumption of querying graphs of up to 10,000 nodes (n), and initial clique sizes $k = 1, 5, 10, 15$ and 20. Implementations I4 and I5 are not shown because they did not achieve comparable performance.

Although more costly in terms of memory, I3 performed far better in time than I4 and I5. Python (I3) presented an overall better performance than Go (I2), with a few exceptions. The bit-array representation showed better performance for both time and memory for cases where $k \geq 5$, in which graphs are denser. When compared to G_7, grammar G_8 negatively affected memory consumption, especially for I3, but provided better time results for $k \geq 15$.

5.3.4 Ontologies

In order to explore a more realistic scenario, we investigate the behavior of Algorithm 1 by querying a set of popular ontologies, publicly available on the Internet. The dataset and grammars are the same used in previous works [24, 8, 61, 39]. The grammars explored in this experiment are summarized in Table 5.6. Grammar G_9 retrieves concepts on the same level of the RDFS (RDF Schema) *subClassOf*/*type* hierarchy. Grammar G_{10} retrieves concepts in adjacent levels of the RDFS *subClassOf* hierarchy. The grammars G_{11} and G_{12} are normal form versions of G_9 and G_{10}, respectively.

The experiment consists of performing a *same generation query* [1]: for each vertex, the query looks for all vertices on the same level of the subclass/type hierarchy. The results for this experiment are presented in Tables 5.7 and 5.8.

Grammar	Production Rules	Description
G_9	$S \to sc\ S\ sc^{-1} \mid t\ S\ t^{-1} \mid sc\ sc^{-1} \mid t\ t^{-1}$	Retrieves concepts on the same level of a class hierarchy [24, 8, 61, 39]
G_{10}	$S \to B\ sc^{-1}$, $B \to sc\ B\ sc^{-1} \mid \varepsilon$	Retrieves concepts on adjacent levels of a class hierarchy [24, 8, 61, 39].
G_{11}	$S \to sc\ S_2 \mid t\ S_3 \mid sc\ sc^{-1} \mid t\ t^{-1}$, $S_2 \to S\ sc^{-1}$, $S_3 \to S\ t^{-1}$	G_9 in Normal Form.
G_{12}	$S \to B\ sc^{-1}$, $B \to sc\ B_2 \mid \varepsilon$, $B_2 \to B\ sc^{-1}$	G_{10} in Normal Form.

Table 5.6: Summary of grammars used for the experiments with RDF Ontologies.

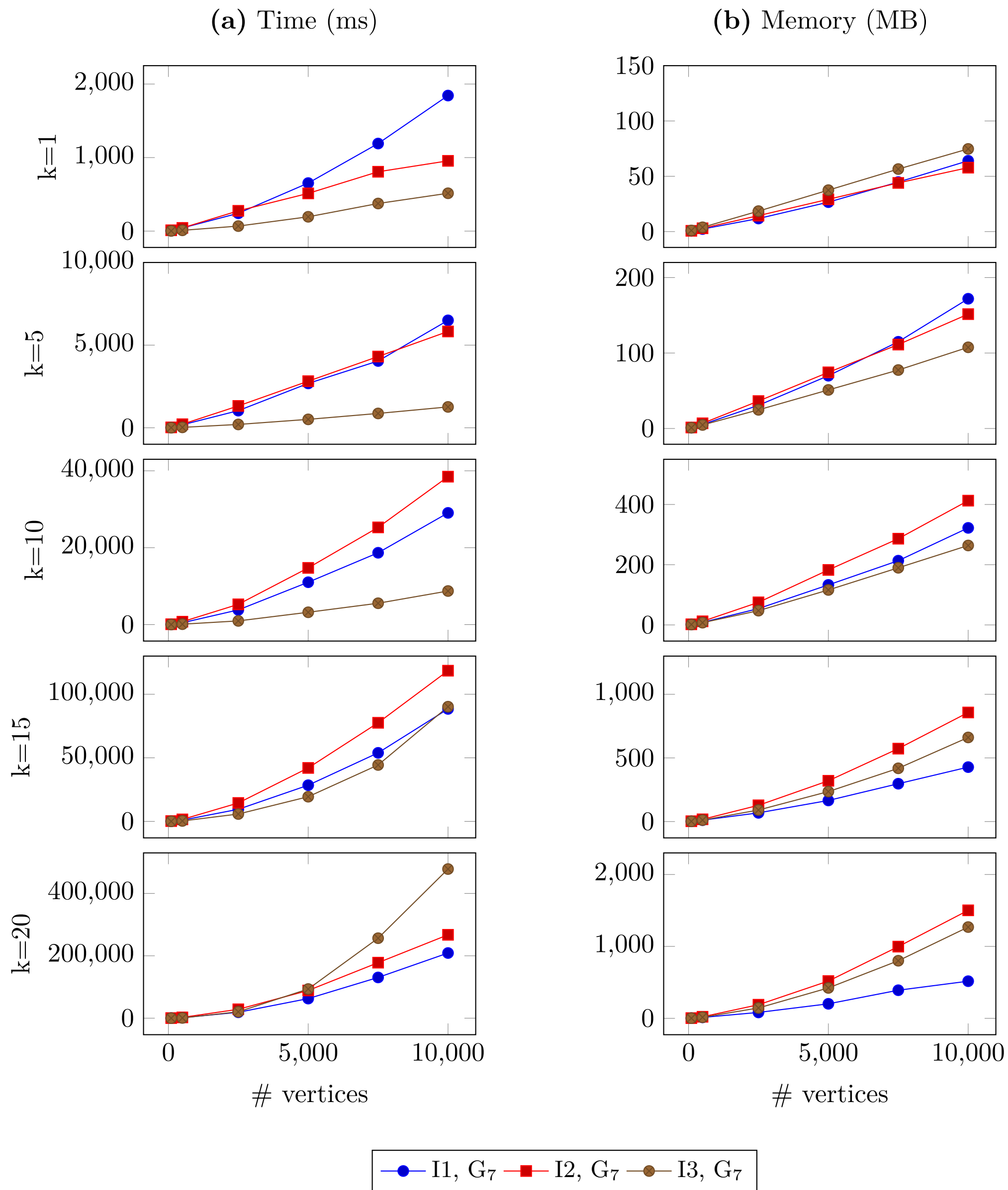


Figure 5.8: Experiment with synthetic social networks and grammar G_7 [29].

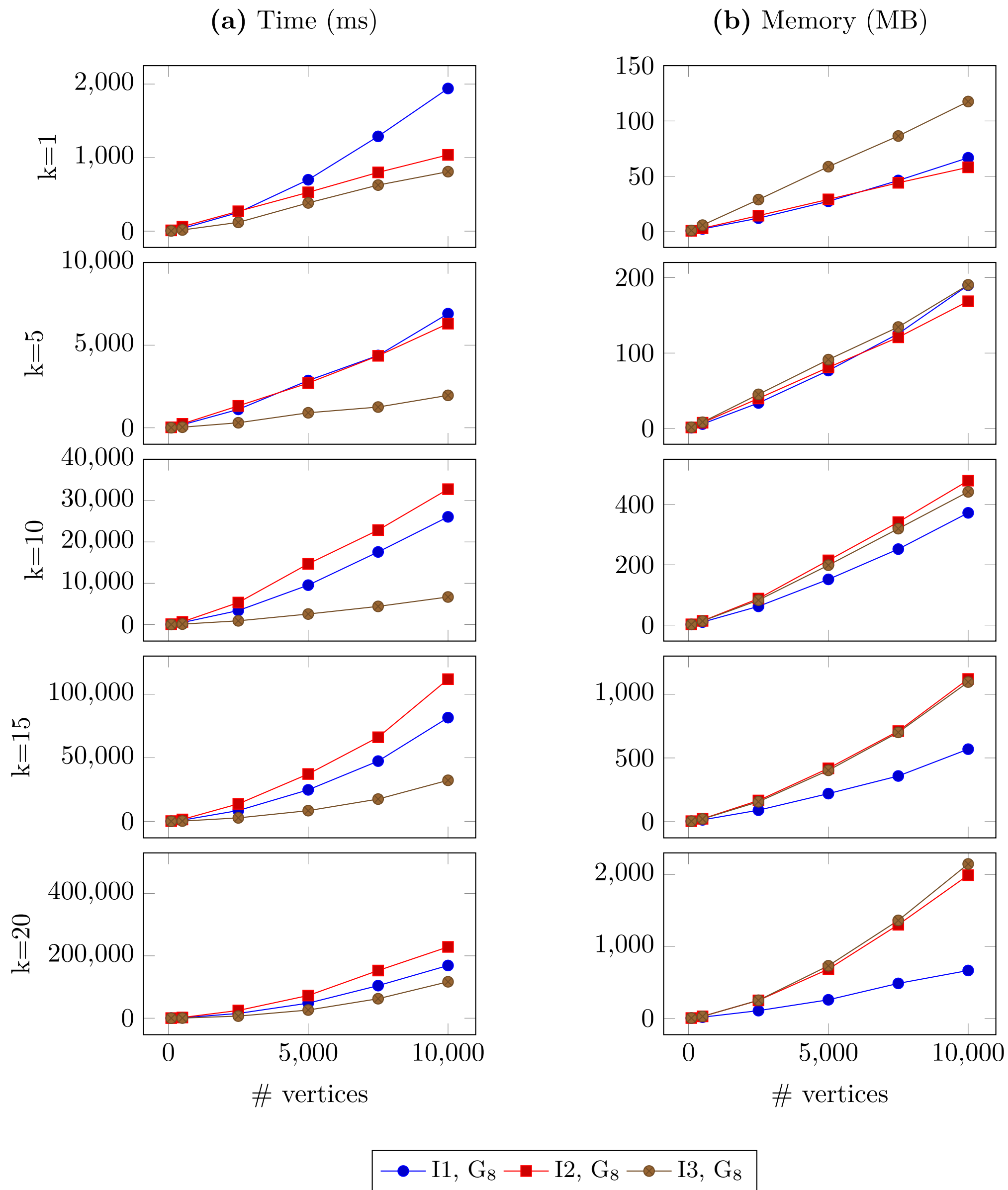


Figure 5.9: Experiment with synthetic social networks and grammar G_8 [29].

Grammar, Graph	**|V|**	**# Results**	**I1 Time**	**I1 Mem**	**I2 Time**	**I2 Mem**	**I3 Time**	**I3 Mem**
G_9, atom	142	15454	191	3,34	407	6,84	41	4,79
G_9, biomedical	134	15156	162	1,49	368	6,89	36	3,58
G_9, foaf	93	4118	26	2,00	60	1,35	7	1,26
G_9, funding	272	17634	147	3,25	281	6,24	33	5,43
G_9, generations	82	2164	14	1,40	36	2,30	4	0,88
G_9, people_pets	163	9472	78	2,91	151	3,08	17	2,92
G_9, pizza	359	56195	403	4,49	698	10,01	128	10,85
G_9, skos	43	810	8	0,60	18	1,10	2	0,50
G_9, travel	92	2499	24	2,30	61	3,30	7	1,26
G_9, univ-bench	90	2540	20	2,30	46	3,23	7	1,27
G_9, wine	468	66572	453	5,37	750	14,76	101	12,59
G_{10}, atom	124	122	97	2,60	294	3,69	26	2,57
G_{10}, biomedical	123	2871	57	2,42	123	2,23	15	1,77
G_{10}, foaf	13	10	1	0,20	2	0,22	0	0,00
G_{10}, funding	93	1158	38	2,65	86	1,45	8	1,01
G_{10}, generations	0	0	0	0,00	0	0,00	0	0,00
G_{10}, people_pets	44	37	6	0,60	10	0,90	1	0,50
G_{10}, pizza	261	1262	88	3,38	124	3,80	15	2,39
G_{10}, skos	2	1	0	0,00	0	0,00	0	0,00
G_{10}, travel	32	63	4	0,50	9	0,70	1	0,25
G_{10}, univ-bench	42	81	6	0,60	9	0,90	2	0,50
G_{10}, wine	163	133	15	1,80	25	2,80	5	1,00

Table 5.7: Performance evaluation on RDF databases with grammars G_9 and G_{10} *(Time in ms, Memory in Mb).*

Grammar, Graph	**|V|**	**# Results**	**I1**		**I2**		**I3**		**I4**		**I5**	
			Time	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**
G_{11}, atom	142	15454	112	1,40	326	4,30	27	5,11	34478	4,92	–	–
G_{11}, biomedical	134	15156	112	1,50	275	4,50	31	5,10	28994	4,80	–	–
G_{11}, foaf	93	4118	24	0,80	86	1,40	9	1,76	1207	0,89	235339	1,03
G_{11}, funding	272	17634	125	2,32	264	4,50	33	7,13	20829	4,85	–	–
G_{11}, generations	82	2164	20	0,70	31	1,10	6	1,38	231	0,55	43426	0,89
G_{11}, people_pets	163	9472	62	1,40	154	2,60	19	4,17	4689	3,07	–	–
G_{11}, pizza	359	56195	463	3,43	863	10,20	113	16,80	240902	12,47	–	–
G_{11}, skos	43	810	9	0,40	13	0,60	3	0,75	66	0,42	5657	0,38
G_{11}, travel	92	2499	32	0,90	50	1,34	9	2,01	644	0,85	228001	1,02
G_{11}, univ-bench	90	2540	19	0,90	61	1,30	9	1,89	711	0,97	194313	1,03
G_{11}, wine	468	66572	425	4,12	824	10,81	101	18,65	254174	14,77	–	–
G_{12}, atom	124	122	102	1,00	212	3,05	26	3,99	5830	3,18	–	–
G_{12}, biomedical	123	2871	56	1,33	123	2,31	16	2,53	3018	1,81	–	–
G_{12}, foaf	13	10	2	0,20	4	0,30	1	0,25	1	0,04	12	0,00
G_{12}, funding	93	1158	30	0,98	54	1,42	9	1,64	636	0,94	155616	1,02
G_{12}, generations	0	0	0	0,10	0	0,10	0	0,00	0	0,04	0	0,00
G_{12}, people_pets	44	37	5	0,50	8	0,60	3	0,75	9	0,29	201	0,25
G_{12}, pizza	261	1262	96	2,45	131	4,00	20	4,13	2357	1,72	556070	1,75
G_{12}, skos	2	1	0	0,10	0	0,10	0	0,00	0	0,04	0	0,00
G_{12}, travel	32	63	10	0,40	9	0,50	2	0,50	11	0,29	263	0,25
G_{12}, univ-bench	42	81	5	0,42	15	0,60	2	0,50	15	0,29	467	0,25
G_{12}, wine	163	133	21	1,40	32	1,90	8	1,75	65	0,79	1827	0,89

Table 5.8: Performance evaluation on RDF databases with grammars G_{11} and G_{12} *(Time in ms, Memory in Mb)*.

Our algorithm (I3) performed significantly faster with a little increase in the memory costs when compared to I4 and I5. Notice that the execution of I5 was not able to produce results for some cases due to the time limit of the experiments.

Comparing programming languages, we observe that the Go version (I2) is consistently much more time consuming than the Python version (I3), regardless the form of the grammars. Concerning the memory costs, the Go version (I2) is, in general, more expensive than the Python one (I3) for grammars not in Normal Form. For grammars in Normal Form, the memory costs are reduced for the Go version (I2).

Table 5.7 presents performance data of the hash- and bit-oriented implementations of Algorithm 1 for grammars G_9 and G_{10}, which are not in Normal Form. The data shows that, in all cases, I1 takes approximately half the time required by I2. Also the overall memory cost was improved by I1, except for some small graphs (this behavior can be explained by the baseline cost of I1). In Table 5.8, similar results can be observed for grammars G_{11} and G_{12}, which are in Normal Form. However, the memory consumption of I1 is always smaller than that of I2.

The data presented in Tables 5.7 and 5.8 are not conclusive with regard to the impact of the form of the grammars over time and memory costs. The grammars in Normal Form are more beneficial in several cases, but the influence of the Normal Form is not sufficient to reduce costs in a general manner. In special, notice that memory consumption increased for grammars in Normal Form for implementation I3.

5.3.5 Phylogenetic Trees

In this experiment we analyse the applicability of Algorithm 1 in another realistic scenario that benefits from context-free path queries. This experiment explores similarities in phylogenetic trees [16]. These trees are used in Biology-related domains such as Bioinformatics and Phylogenetics. A node in a phylogenetic tree may represent a singular species or group of individuals. Each species can be described by a number of features, that may be physical, biochemical, behavioral, or molecular.

We build synthetic phylogenetic trees to simulate evolutionary relationships. Given a set of nodes, the common ancestor is given by the root of the sub-tree containing these nodes, its descendants. Branches represent the evolution of one species into another, by means of changes in their features. In our setting, each individual species is modeled as a sequence of 100 features, described as a combination of five possible values (A, B, C, D, E).

The procedure we use to produce a synthetic phylogenetic tree T is guided by three parameters: the desired height h of the tree, the maximum arity m of each internal node, and the number n of species that appear in the phylogenetic tree. Our experiments evaluate the scalability of Algorithm 1 by varying the number of nodes n, using a random number of mutations m for each species, ranging from 0 to 2, and tree height defined as $h = \lfloor (n + \lceil log(n, m) \rceil)/2 \rfloor$.

The tree generation algorithm is as follows: (i) initially we generate a tree of height h whose nodes have arity one; (ii) after that, we add new children to randomly chosen internal nodes while the number of vertices is not reached, respecting height and arity constraints.

In our RDF representation, a mutation in a phylogenetic tree is given by five triples. Figure 5.10 shows our RDF representation of the mutation of species s_i into s_j. The mutation m_1 changes the feature at position 1 of the string representing s_i, from A to B. Notice that for each new mutation we have 5 new nodes on the data graph representation of a phylogenetic tree.

The experiment consists of looking for species such that:

(i) they are equidistant to a common ancestor species, and

(ii) they are the result of mutating their feature at the first position from A to B.

The grammars used for the queries in the experiment are given in Table 5.9. In Grammar G_{13}, the non-terminal symbol S checks that the first and last steps of a path both present the desired mutation (from A to B, at the initial position of the feature sequence of both species). The non-terminal symbol S_2 checks for the existence of an equidistant common ancestor. The grammar uses nested expressions and navigational axes [45] to check the desired mutation. For instance, the nested expression $[I]$ defines

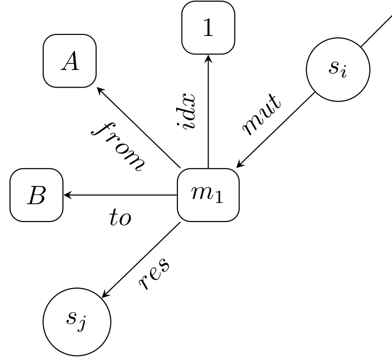


Figure 5.10: RDF representation of the mutation of species s_i into species s_j.

#	ProductionRules		Description
G$_{13}$	$S \to res^{-1}[I][F][T]\ mut^{-1}\ S_2\ mut\ [I][F][T]\ res$, $S_2 \to res^{-1}\ mut^{-1}\ S2\ mut\ res$, $S_2 \to \varepsilon$, $I \to idx$ self::1, $F \to from$ self::A, $T \to to$ self::B		Retrieves descendants equidistant from a common ancestor that suffered an "A1B" mutation.
G$_{14}$	$S \to res^{-1}\ X_1$, $X_2 \to [F]\ X_3$, $X_4 \to mut^{-1}\ X_5$, $X_6 \to mut\ X_7$, $X_8 \to [F]\ X_9$, $S_2 \to res^{-1}\ Y_1$, $Y_2 \to S_2\ Y_3$, $S_2 \to \varepsilon$, $I \to idx$ self::1, $F \to from$ self::A, $T \to to$ self::B	$X_1 \to [I]\ X_2$, $X_3 \to [T]\ X_4$, $X_5 \to S2\ X_6$, $X_7 \to [I]\ X_8$, $X_9 \to [T]\ res$, $Y_1 \to mut^{-1}\ Y_2$, $Y_3 \to mut\ res$,	G$_{13}$ in normal form.

Table 5.9: Summary of grammars used for the experiments with synthetic phylogenetic trees.

a condition described by the non-terminal symbol I. In this case, the query requires that the mutation occurs at the position with index 1 for the node being traversed, suppose m_1. Then, the expression $[I]$ requires the algorithm to look for nodes that are reachable from m_1 by paths whose traces are strings derivable from I. New edges are created if such paths are found and the query evaluation is resumed from m_1. Grammar G$_{14}$ is in normal form and generates the same language as G$_{13}$.

This experiment considers only implementations I1 and I2, since the other implementations do not support nested expressions and navigational axes. Time and memory consumption of I1 and I2 for trees containing up to 8,000 species and grammars G$_{13}$, G$_{14}$ are given in Figure 5.11.

We can observe that I1 is faster than I2 for both grammars. On the other hand, the I1 is more memory consuming. This indicates the advantage of performing fast operation over arrays of bits, but also the memory overhead of this representation for sparse data graphs.

The time and memory costs related to G$_{14}$ are similar to those obtained for G$_{13}$, with a little increase. This behavior can be explained by the number of production rules in G$_{14}$. The bigger the number of production rules, the bigger the number of trace items to be investigated.

Notice that this is not the only scenario in Biology where context-free path queries may be applicable. Sevon and Eronen [51] use context-free queries to identify similarities between evolutionary patterns (by means of identifying sub-graph similarities).

5.3.6 Single Source Queries

In this section, we present an experiment that explores the advantage of algorithms that traverse the data graph considering only a subset of vertices. In practical situations, it is common for one to be interested in querying not the whole graph database, but only a subgraph in it. Our algorithm presents this capability

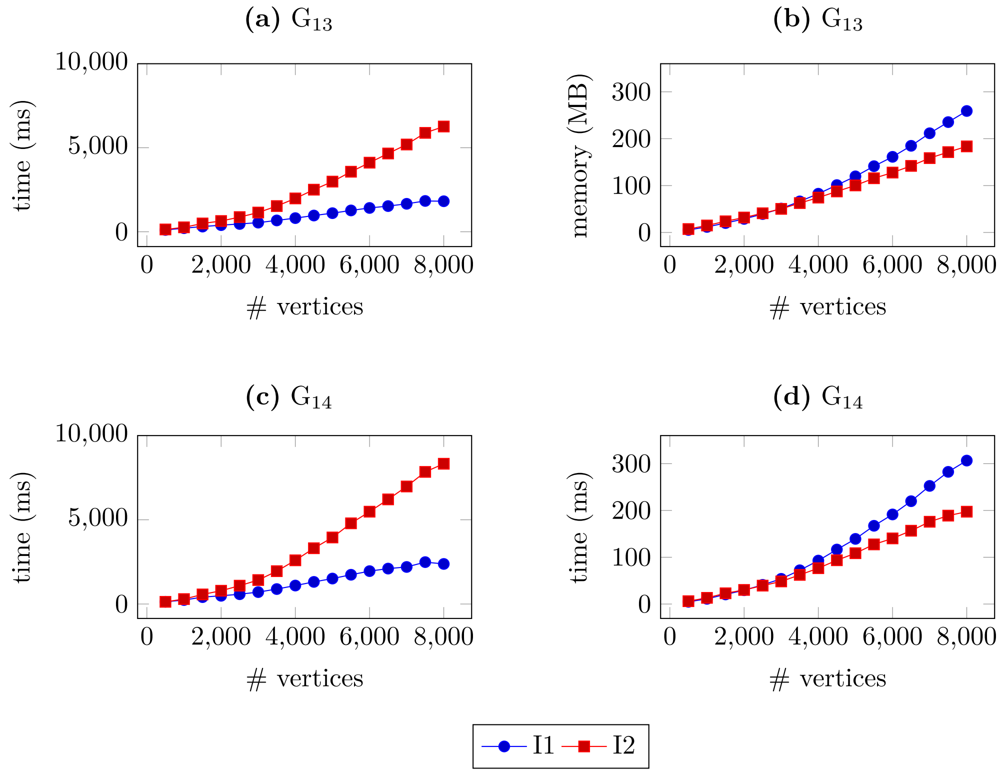


Figure 5.11: Experiments with phylogenetic trees

and so it may drastically reduce the costs of processing queries.

Suppose that we have a data graph containing a path π formed by the concatenation of paths π_1, π_2, π_3 and π_4 (*i.e.*, $\pi = \pi_1.\pi_2.\pi_3.\pi_4$) such that: (i) $\pi_1 = (v_1, l_1, v_2), \ldots, (v_{n-1}, l_{n-1}, v_n)$, $\pi_2 = (v_n, l_n, v_{n+1}), \ldots, (v_{2n-1}, l_{2n-1}, v_{2n})$, and they have the same trace a^n; and (ii) $\pi_3 = (v_{2n}, l_{2n}, v_{2n+1}), \ldots, (v_{3n-1}, l_{3n-1}, v_{3n})$, $\pi_4 = (v_{3n}, l_{3n}, v_{3n+1}), \ldots, (v_{4n-1}, l_{4n-1}, v_{4n})$, and they have the same trace b^n. In the case that we are interested in paths whose traces are of form $a^k b^k$, our algorithm is able to look for paths starting at vertex v_n and to find the path from this vertex to v_{3n}, without the need of traversing the whole data graph. The experiment we describe in this section uses ab-linear graphs and grammars G_3 and G_4 as defined in Section 5.3.1 to explore this scenario.

Figure 5.12 shows time and memory consumption of all implementations. Notice that I1 outperforms all the others implementations with regard to execution time, both for grammars G_3 and G_4. However, I1 is the most memory consumptive implementation. This can be explained by the representation of data as a bit-array, which is efficient for operations over vertices but suffers when applied over sparse data graphs. Since we are dealing with a linear graph in this experiment, we have a very low degree of connectivity among vertices. Although I4 presents the better behavior concerning memory consumption, its time explodes for a small number of vertices. In the case of I5, this implementation timed out for the smallest number of vertices considered in the experiment.

When considering both time and memory costs, implementation I2 (hash-oriented, Go version) seems to present the best trade-off for the setting of the experiment.

5.4 Conclusions

We presented an algorithm for the evaluation of context-free path queries for RDF databases, analyzed its correctness, as well as its worst-case runtime and space complexity. We validated our work by using both synthetic and real-life data bases, showing that our prototype outperforms other ones found in the literature.

Our experiments show that several factors affect its performance. Those factors are:

- The form of the grammar used in the formulation of queries: this affects the number and size of the right-hand side of production of rules;

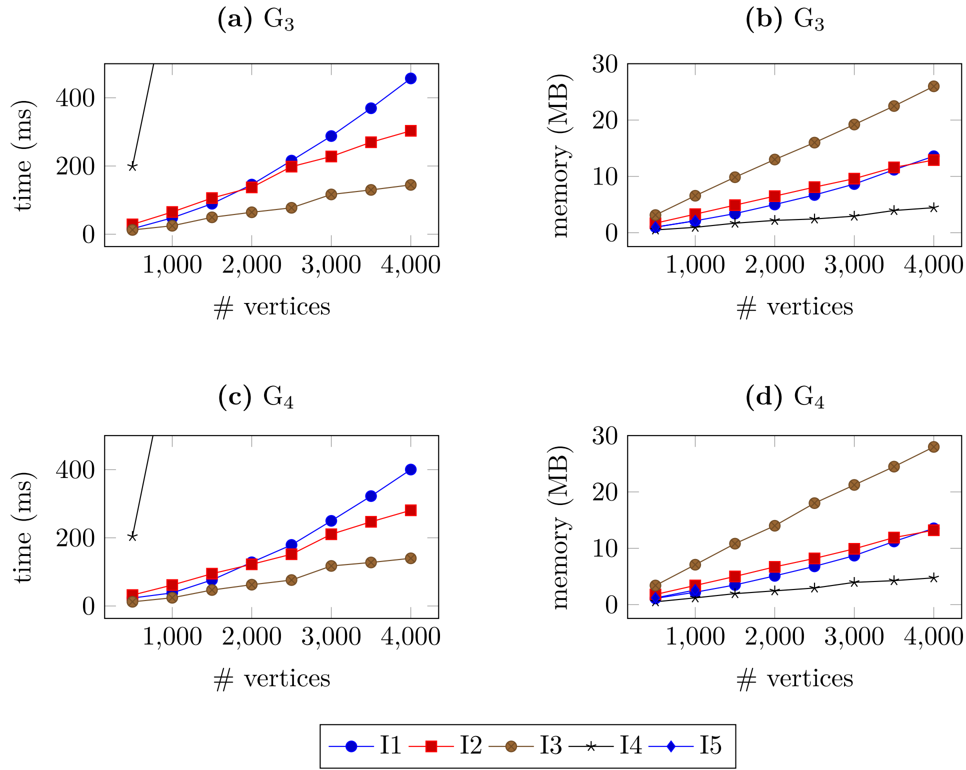


Figure 5.12: Experiments with a single start vertex over linear graphs

- The density of edges of the graph being queried (where complete graphs are considered the densest ones);
- The size of the graph;
- The data structure used.

It is worth noticing that those factors are not independent of each other. As a general rule, we can conclude that smaller grammars (in the number of production rules and non-terminal symbols) are preferable when querying bigger, denser graphs. For sparse graphs, such as lists or cycles, the performance figures show less impact from the form of the grammar. Also, we noticed that the use of ambiguous grammars in the query does not have a significant impact on performance nor memory consumption.

As future work, we intend to investigate a parallel version of Algorithm 1. This may improve its performance and scalability, since the treatment of unmarked vertices in position sets may be done in parallel. Better management of large graphs that do not fit in the memory is also a desired improvement. Another desirable feature is to use the information on trace items to reconstruct paths.

We intend to work on the definition of benchmarking data sets for algorithms for the evaluation of CFPQs. This will make it possible to have more accurate data to compare the different algorithms that implement those queries. This benchmark should be based on real-life problems [42, 62, 16].

Chapter 6

Formal-Language-Constrained Graph Minimization

In this chapter, we present the Formal-Language-Constrained Graph Minimization (FLGM) problem. This chapter is an extension of our previous work Medeiros *et al.* [36]. We propose solutions for the regular and context-free versions of that problem.

The FLGM problem consists of deleting from an input data graph as many triples as possible, in such a way that the answers to a user-defined utility query are preserved. The FLGM problem arises in graph minimization scenarios, such as network minimization, source-code analysis and others.

Simplifying the project of networks (such as traffic or communication networks) minimizes implementation and maintenance costs. FLGM algorithms can be used, for instance, to assist in the project of a traffic network connecting touristic attractions in a city. Edges labeled with the kind of transportation means connect those attractions. A formal language would define the pattern of paths that users would take to travel among attractions.

Many tasks in source-code analysis resume to graph reachability problems whose paths match some context-free language. Pointer assignments, function calls, and property reads/writes are often modeled as interleaving Dyck languages. Dyck languages are languages of the form $a^n b^n$. Simplifying the underlying graph that corresponds to those operations can improve the performance of algorithms for source-code analysis [32].

Another instance of the FLGM problem is privacy protection. Organizations that keep user data might have their databases attacked. Preserving the minimum sensitive information possible for the organization's purposes minimizes the damage in the case of a possible data breach. Also, such minimization reduces data storage and maintenance costs.

6.1 Problem Definition

Minimizing a graph D consists of identifying a subset $D^- \subseteq D$ whose weight (or cost) is minimum, and there remains at least one valid path for every answer of a given query. This query set is used to preserve data utility.

Definition 13 (Formal-Language-Constrained Graph Minimization Problem)**.** *Given a grammar G, a data graph D, a set of queries Q and a weight (or cost) function f, the* FLGM (Formal-Language-Constrained Graph Minimization) *problem consists of identifying a subgraph $D^- \subseteq D$ such that for all queries $(a, X) \in Q$, it holds that $(a, X, y) \in \mathcal{R}_{G,D^-} \iff (a, X, y) \in \mathcal{R}_{G,D}$, and $f(D^-)$ is minimum.*

This optimization problem is NP-hard, and its decision version is NP-complete for polynomial-time decidable languages. Inspired by Anciaux *et al.* [5], we demonstrate these statements by reducing an instance of the APMWS (All-Positive Minimum Weighted Boolean Satisfiability) problem [4] to an instance of the FLGM problem.

Theorem 9. *The APMWS problem is reducible to the FLGM problem.*

Proof. An instance of the APMWS problem consists of, given a formula in the conjunctive normal form $F = \bigwedge_j(\bigvee_k b_{j,k})$ and a weight function $f(h) = \sum_{h(b_{j,k})=true} w_{j,k}$, finding an assignment of truth-values to variables $h : b_{j,k} \mapsto \{true, false\}$ such that $h(F) = true$, and $f(h)$ is minimum.

The FLGM problem can be represented by a boolean formula similar to that of the APMWS problem as $F' = \bigwedge_j(\bigvee_k(\bigwedge_m b_{j,k,m}))$, where:

1. a variable $b_{j,k,m}$ is true if its corresponding triple $t_{j,k,m}$ is in D^-, and $false$ otherwise (it can occur that $t_{j,k,m} = t_{j',k',m'}$ for $j \neq j', k \neq k', m \neq m'$.);

2. given $A \in N$ and $u, v \in V$, the inner conjunction corresponds to a set of triples that form an A-derivable path from u to v;

3. the disjunction corresponds to the all A-derivable paths from u to v;

4. the outer conjunction corresponds to the answers for the queries in Q;

Solving this problem means finding an assignment $h' : b_{j,k,m} \mapsto \{true, false\}$ such that $h'(F') = true$, and $f(h')$ is minimum. Considering triple sets of size 1 in the inner conjunction of the FLGM problem, an APMWS variable $b_{j,k}$ can be rewritten as $b_{j,k,1}$. We can then solve an APMWS instance by solving its equivalent FLGM instance. ☐

Corollary 9.1. *The FLGM* optimization *problem is NP-hard.*

Proof. From Theorem 9 and from the fact that the APMWS optimization problem is NP-hard [4], it follows that the FLGM problem is NP-hard as well. ☐

Usually, optimization problems can be recast as decision problems by providing a constant k and verifying if, for a given solution h, $f(h) \leq k$ [18]. The next corollary establishes the complexity of the FLGM decision problem.

Corollary 9.2. *If input grammar G belongs to a class of polynomial-time decidable grammars, the FLGM* decision *problem is NP-complete.*

Proof. Given a solution to an instance of the FLGM decision problem, which includes a constant k, we can verify the solution in polynomial time by (1) summing the weight of all edges in D^- and comparing it to k; and (2) running a polynomial-time algorithm to evaluate the path queries Q. Therefore, FLGM $\in$ NP. From that, and from Corollary 9.1, we have that the FLGM decision problem is NP-complete for polynomial-time decidable grammars. ☐

Notice that such restriction on the class of the input grammar is necessary, since the sole task of recognizing a string using a non-polynomial-time decidable grammar is, by definition, not in P.

In practice, it is frequently sufficient to trade algorithms that find optimum solutions for complex problems in non-polynomial time by approaches that find near-optimal solutions in polynomial time. *Heuristics* can be used to find such near-optimal (approximate) solutions [18]. The next corollary establishes the approximability of the FLGM decision problem.

Corollary 9.3. *The FLGM problem is not in APX and is in 0-DAPX.*

Proof. From Theorem 1, and assuming $P \neq NP$, it is a consequence of the fact that the APMWS problem is not in APX [4] and has differential approximation of 0-DAPX [10]. ☐

In other words, Corollary 9.3 states that there cannot exist an algorithm with constant approximation ratio for the FLGM problem (because it is not in APX) and any polynomial-time algorithm will return the worst solution in at least one of their instances in its worst case (because of its differential approximation of 0-DAPX).

In the next sections, we present heuristic-based algorithms for solving the graph minimization problem for a subset of languages. Our techniques are composed of two stages: path query evaluation and minimum graph construction. Since string recognition is itself a complex task, we restrict the grammars used to

define paths to the classes of regular and context-free grammars. These classes allow us to cover not only the wide range of applications that use regular languages, but also those that require context-free ones [62, 16, 42]. Thus we present a solution to the *Regular*- and *Context-Free-Language-Constrained Graph Minimization* problems.

6.2 Regular-Language-Constrained Graph Minimization

Algorithm 2 is our regular-language-constrained graph minimization algorithm. This algorithm starts by building the product automaton PA. Then, it proceeds to the main loop, where it randomly picks paths from start states (a, q_0) to final states (b, q_f) performing a depth-first search. For each answer b, only one path is added to D^-. Example 2 illustrates how this algorithm works.

ALGORITHM 2 : The RGM Algorithm

Input : A regular expression *exp*, a set of queries Q, and a graph D
Output : A minimized graph D^-

```
1  function minimize
2  | A := automaton corresponding to exp
3  | PA := product automaton D × A
4  | D⁻ := { }
5  | foreach (a, q0) in PA do
6  | | foreach randomly chosen path π from (a, q0) to a final state (b, qf) do
7  | | | if no path from (a, q0) to any final state (b, qf') has been added then
8  | | | | foreach ((x, qi), p, (y, qj)) ∈ π do
9  | | | | | D⁻ := D⁻ ∪ {(x, p, y)} // Add edges in π to D⁻
10 | return D⁻
```

Example 2. *Let D be the graph in Figure 6.1a, and let exp be the regular expression a^+. Algorithm 2 constructs the automaton A for the regular expression exp (line 2, Figure 6.1b). The automaton A is used in the construction of the product automaton $PA = D \times A$ (line 3, Figure 6.1c). Algorithm 2 starts at every start state in PA and performs a depth-first search for final states (lines 6-9).*

Since the order of picking paths is random, in the next steps, we illustrate one possible sequence. Starting at state $(2, q_0)$, the algorithm walks through edge $((2, q_0), a, (3, q_1))$. Because $(3, q_1)$ is a final state, the algorithm adds edge $(2, a, 3)$ to D^-. Continuing through edge $((3, q_1), a, (1, q_1))$, the algorithm reaches another final state, $(1, q_1)$, so it adds the edge $(3, a, 1)$ to D^- (it is not necessary to add $(2, a, 3)$ again). The algorithm continues through edge $((1, q_1), a, (2, q_1))$ and adds edge $(1, a, 2)$ because $(2, q_1)$ is also final state. Walking through $((2, q_1), a, (3, q_1))$, it reaches the final state $(3, q_1)$, which was reached previously, so the search backtracks and continues. This behavior is repeated for all start states backtracking only at states visited in the current iteration. The minimized graph D^- presented in Figure 6.1d is a possible optimum solution for this example.

Next, we state the correctness and worst-case time and space complexities of Algorithm 2.

Theorem 10 (Correctness of Algorithm 2)**.** *Let w and w^- be the weights of the input graph D and output minimized graph D^- respectively. Algorithm 2 produces a minimized graph D^- such that the answers to all queries in Q are preserved and $w^- \leq w$.*

Proof. Given a product automaton $PA = D \times A$, where D is the input graph and A is the automaton corresponding to input expression *exp*, it is a known fact that if two states $(a, q_0), (b, q_f) \in PA$, where a, b are the start and destination vertices and q_0, q_f are the initial and a final state of A respectively, are connected via a path in PA, then there exists a path from a to b in D that is in the language of *exp* [45].

Since for each query $(a, q_0) \in Q$ (line 5) Algorithm 2 finds a path to every final state (b, q_f) (line 6), all answers are preserved. Also, it discards paths whose destination has already been reached (line 7), so the number of edges added to D^- and consequently its total weight w^- tend to decrease, and therefore $w^- \leq w$. □

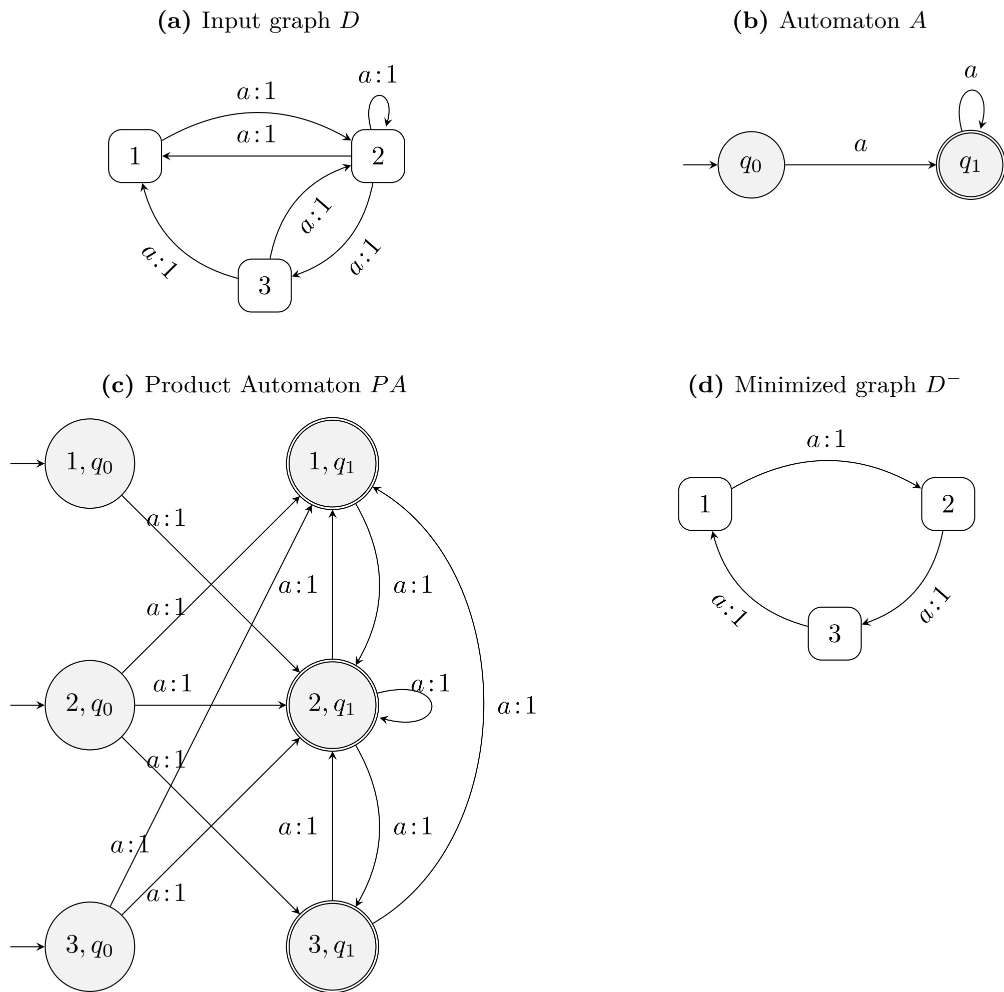


Figure 6.1: Example execution of the RGM algorithm

Theorem 11 (Space Complexity of Algorithm 2)**.** *The worst-case space complexity of Algorithm 2 is* $\mathcal{O}(|D| * |A|)$.

Proof. The only data structures that Algorithm 2 creates are the automaton A, the product automaton PA, and the minimized graph D^-, which is a subset of D. Therefore, the largest data structure is $PA = D \times A$, so the worst-case space complexity is $\mathcal{O}(|D| * |A|)$. □

Theorem 12 (Time Complexity of Algorithm 2)**.** *The worst-case time complexity of Algorithm 2 is* $\mathcal{O}(|Q| * |D| * |A|)$.

Proof. The basic operation of Algorithm 2 is line 6, where it iterates over all paths π from (a, q_0) to (b, q_f), for each $(a, q_0) \in Q$. Although we refer to π as a path (*sequence* of edges), we can instead represent it as a subgraph (*set* of edges). Therefore, the worst-case time complexity of Algorithm 2 is $\mathcal{O}(|Q| * |D| * |A|)$. □

6.3 Context-Free-Language-Constrained Graph Minimization

The context-free-language-constrained graph minimization algorithm is presented in Algorithms 3 and 4. It uses the output of Algorithm 1. For each edge $(a, X, b) \in D'$, where $(a, X) \in Q$, the minimization algorithm randomly picks a subgraph that matches an X-derivable path from a to b from trace items in I using the auxiliary function *get_subgraph* in Algorithm 4. Paths are chosen using depth-first search, starting at vertex b in any trace item of the form $X \to \{a\} \dots \{b, \dots\}$, and walking from right to left towards the start vertex a. The *color* variable (Alg. 3, l. 3) is used to guarantee that the search stops. A new color is chosen at each iteration (Alg. 3, l. 10). Since the algorithm picks only one subgraph (line 5) for each edge (a, X, b), the number of edges in D^- and consequently its total weight tend to decrease. The iteration of position sets from right to left avoids following paths that do not reach a vertex in the last position set. This process is applied recursively for non-terminal symbols between position sets. Subgraphs that match valid paths found in this process are stored in the mapping *answers*. Doing so, the algorithm avoids recomputing paths and getting stuck in infinite recursion. Subgraphs are effectively added to D^- in function *add_subgraph*.

ALGORITHM 3 : The CFGM Algorithm

```
   Input : A grammar G, a set of queries Q, an annotated graph D', and a set of trace items I
   Output : A minimized graph D⁻
 1 function minimize
 2 |  D⁻ := { }
 3 |  color := 1
 4 |  let answers : V × (Σ ∪ N) × V ↦ P(D')
 5 |  foreach a, X, b s.t. (a, X) ∈ Q ∧ (a, X, b) ∈ D' ∧ no X-derivable path from a to b has been
 |     added do
 6 |  |  foreach randomly picked trace item it = (X → x1 ... xm, f) such that a is in the first
 |  |     position set C0 and b is in the last position set Cm do
 7 |  |  |  if get_subgraph(b, it, m, color) ≠ null then
 8 |  |  |  |  break
 9 |  |  add_subgraph(a, X, b)
10 |  |  color := color + 1
11 |  return D⁻
```

Algorithm 4 contains two auxiliary subprograms that are used by Algorithm 3. The function *get_subgraph* concentrates the mechanics of the recursive search over trace items. This function returns a subgraph that links the vertex a in the first position set to the vertex b in the i-th position set (lines 5, 10, 29). Although this function always finds a subgraph that answers a given query (if such answer exists), it might return **null** sometimes. For instance, if the search reaches a given vertex in the last position set of a trace item that is painted (line 6), i.e., that given vertex in that same position set has already been visited in a previous call in that search, and no answer was found so far, it does not make sense to continue the search from that point, so the algorithm returns **null**. This indicates that it was not possible to find a

valid subgraph following that path, so another path in that or another trace item must be tried. The function *add_subgraph* (lines 31-36) recursively adds the terminal edges stored in the mapping *answers* in the function *get_subgraph* to the minimized graph D^-.

Example 3 presents an example run of Algorithms 3 and 4.

ALGORITHM 4 : Auxiliary subprograms for Algorithm 3

```
1  function get_subgraph(b, it = X → {a}x_1 C_1 ... C_{m-1} x_m C_m, i, color)
2  │ if i = 0 then
3  │ │ if m = 0 then
4  │ │ │ answers(b, X, b) = { }
5  │ │ return { }
6  │ else if i = m ∧ b is painted then
7  │ │ res := null
8  │ │ if (a, X, b) ∈ dom(answers) then
9  │ │ │ res := answers(a, X, b)
10 │ │ return res
11 │ paint(b, it, i, color)
12 │ foreach w s.t. w ∈ C_{i-1} ∧ (w, x_i, b) ∈ D' (pick firstly w's such that (w, x_i, b) ∈ D^-) do
13 │ │ paint(a, it, i − 1)
14 │ │ edges := null
15 │ │ if x_i ∈ Σ then
16 │ │ │ edges := {(w, x_i, b)}
17 │ │ else if x_i ∈ N then
18 │ │ │ foreach it2, b s.t. it2 = x_i → {w} ... C_{m2} ∧ b ∈ C_{m2} do
19 │ │ │ │ if get_subgraph(b, it2, m2, color) ≠ null then
20 │ │ │ │ │ edges = {(w, x_i, b)}
21 │ │ │ │ │ break
22 │ │ if edges ≠ null then
23 │ │ │ edges2 := { }
24 │ │ │ if i > 1 then
25 │ │ │ │ edges2 := get_subgraph(b, it, i − 1, color)
26 │ │ │ if edges ≠ null then
27 │ │ │ │ if (w, x_i, b) ∉ dom(answers) then
28 │ │ │ │ │ answers(w, x_i, b) = edges ∪ edges2
29 │ │ │ │ return edges ∪ edges2
30 │ return null
31 procedure add_subgraph(a, X, b)
32 │ foreach (s, p, o) ∈ answers(a, X, b) do
33 │ │ if p ∈ Σ then
34 │ │ │ D^- := D^- ∪ {(s, p, o)}
35 │ │ else if p ∈ N then
36 │ │ │ add_subgraph(s, p, o)
```

Example 3. *Let D be the graph in Figure 6.2a and G the grammar given by $S \to a\ S, S \to \varepsilon$. Algorithm 3 uses Algorithm 1 to construct the annotated graph D' and the set of trace items I respectively presented in Figures 6.2b and 6.2c. In its main loop, the algorithm uses the set of trace items I and the annotated graph D' to randomly pick subgraphs that form paths that answer queries in Q. It starts at vertices in the last position set of a trace item and performs a depth-first search for the vertex in the first position set.*

Since the order that the algorithm iterates over trace items is random, in the next steps, we follow one possible sequence, illustrated in Figure 6.2d. Let us pick $(a, X, b) = (1, S, 3)$ first (Alg. 3, l. 5). Start at vertex 3 in the last position set of trace item $S \to \{1\}\ a\ \{2\}\ S\ \{1, 2, 3\}$ (Alg. 3, l. 6). The algorithm starts the search for a path that answers $(1, S, 3)$ (Alg. 3, l. 7) using function get_subgraph (Alg. 3, l. 1-30). Since the symbol S between the last and the second position set is non-terminal (Alg. 4, l. 17), the algorithm has to recursively pick a path for $(2, S, 3)$.

The only item capable of providing a path for that non-terminal-labeled edge is $S \to \{2\}\ a\ \{1, 2, 3\}\ S\ \{1, 2, 3\}$ (Alg. 4, l. 18). Up to this moment, no edge has been added to the mapping answers or to D^-. The algorithm picks edge $(3, S, 3)$ (Alg. 4, l. 12), which is guaranteed by item $S \to \{3\}$ (Alg. 4, l. 18). The mapping $(3, S, 3) \mapsto \{\ \}$ is now added to answers (Alg. 4, l. 4), so, from now on, whenever an S-derivable path

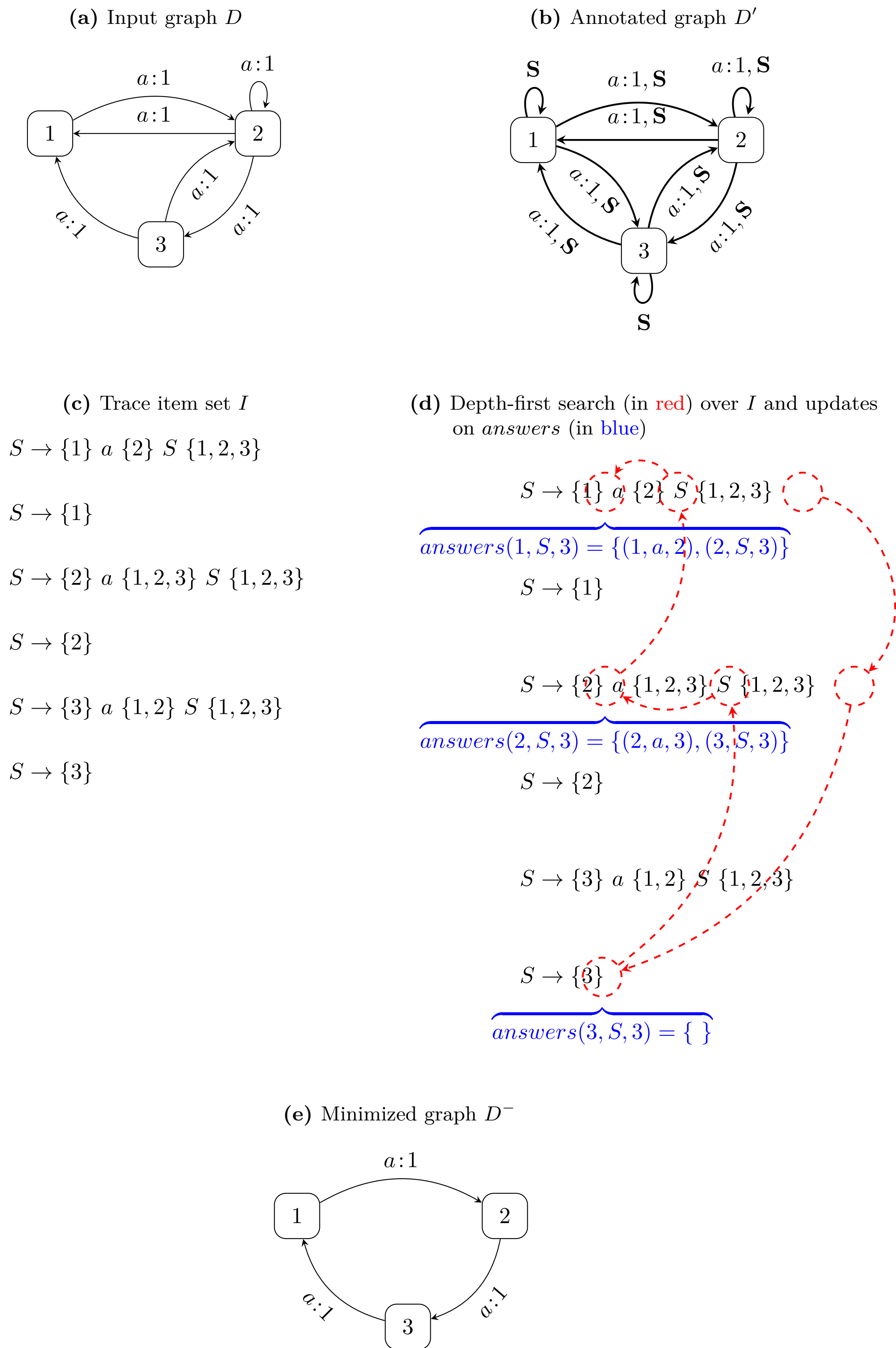


Figure 6.2: Example execution of the CFGM algorithm

from 3 to itself is needed, the value in answers will be used. The next edge picked is $(2, a, 3)$*, which is a terminal-labeled edge in D (Alg. 4, l. 15) and it is added to* $answers(2, S, 3)$ *along with* $(3, S, 3)$ *(Alg. 4, l. 28). With edge* $(2, S, 3)$ *guaranteed, the algorithm returns to the first trace item and picks edge* $(1, a, 2)$ *(Alg. 4, l. 12), which is also a terminal-labeled edge in D (Alg. 4, l. 15).*

Recursively adding all terminal edges to D^- *(Alg. 3, l. 9, Alg. 4, l. 31-36) using answers, the edge* $(1, S, 3)$ *is guaranteed to exist in* D^-*. This behavior is repeated for non-terminal edges* $(a, X, b) \in D'$ *such that* $(a, X) \in Q$*. The minimized graph* D^- *presented in Figure 6.2e is a possible optimum solution for this example.*

Next, we state the worst-case time and space complexities of Algorithm 3.

Theorem 13 (Space Complexity of Algorithm 3)**.** *The worst-case space complexity of Algorithm 3 is* $\mathcal{O}(|V|^2)$.

Proof. Algorithm 3 creates the minimized graph D^- and the mapping *answers*. The minimized D^- is a subset of D, so it may use at most $\mathcal{O}(|D|)$ space. The definition of *answers* is a mapping from edges to sets of edges. These sets of edges in the range of*answers*, however, are of limited size. They can only reach a size equal to the length of the trace used to produce them. Considering that the length of grammar rules, and therefore of trace items, is insignificant if compared to the size of the input graph, we can consider this length to be 1. So *answers* may be as big as $|V \times (\Sigma \cup N) \times V| * 1$, which is $\mathcal{O}(|D|)$ too. Since we are considering the grammar to be much smaller than the graph and the edge labels are restricted to the grammar's alphabet, we can consider only the size of the set of vertices in D. Therefore, the worst-case space complexity of Algorithm 3 is $\mathcal{O}(|V^2|)$. ☐

Theorem 14 (Time Complexity of Algorithm 3)**.** *The worst-case time complexity of Algorithm 3 is* $\mathcal{O}(|V|^3)$.

Proof. The basic operation of Algorithm 3 is line 7, where it will iterate over all X-derivable paths π from a to b, for each $(a, X) \in Q$. That operation is the function *get_subgraph* in Algorithm 4, which iterates trace items recursively saving subgraphs into the mapping *answers* to avoid redoing recursion for a given (a, X, b). On its turn, the basic operation of *get_subgraph* is line 18, which is executed once for each vertex w in the previous position set (line 12) ($|V|$ times in the worst-case) times the number of trace items starting at w and containing the vertex b in the its last position set (line 18) ($|V| * |P|$ times in the worst-case), totalizing $|V| * |N| * |D| * |P|$ times in the worst-case. Considering the size of the grammar insignificant comparing to the size of the graph, we have that the worst-case time complexity of Algorithm 2 is $\mathcal{O}(|V| * |N| * |D| * |P|) = \mathcal{O}(|V| * |V|^2) = \mathcal{O}(|V|^3)$. ☐

6.4 Experiments

In this section, we present experiments with prototypes of Algorithms 2 (RGM) and 3 (CFGM). The source-code as well as the databases used in the experiments are publicly available in our online repository[1]. In the charts presented, algorithms RGM and CFGM are painted red and blue respectively. Missing values for CFGM in some executions are due to program halting on reaching Python's maximum recursion depth. We use a variety of graphs, languages and algorithm behaviors. The graphs vary in density, representing the worst and a better case[2]. For the purpose of comparison, all these languages are regular, since only CFGM is capable of recognizing context-free languages. We define the queries as the product between all vertices and the start state of the automaton, for the RGM algorithm, or the start symbol of the grammar, for the CFGM algorithm. We developed three distinct behaviors for both algorithms:

- Alphabetical-sort behavior – vertices, edges, states and trace items are iterated in alphabetical order (for trace items, only the production rule is considered). For instance, the edges $(v1, b, v3), (v1, a, v2), (v2, a, v1), (v10, a$

[1]Online git repository containing the source-code and datasets used in the experiments: `https://gitlab.com/ciromoraismedeiros/rdf-ccfpq`.

[2]The best case would be linear graphs. However, we did not perform experiments with such graphs because they do not simulate realistic situations.

would be iterated in the order $(v1, a, v2), (v1, b, v3), (v10, a, v3), (v2, a, v1)$. Due to the fact that we distribute the weight of the edges in complete graphs in ascending order regarding their label, iterating over edges in this order simulates a greedy heuristic, where the algorithm chooses cheapest edges first. Since the order is consistent, this behavior tends to reuse edges.

- Implementation-dependant behavior – vertices, edges, states and trace items are iterated according to Python's built-in iteration order. This order is defined by the Python interpreter according to the current organization of data inside the data structures. There is no guarantee that it preserves insertion order, but it tends to repeat the same order between iterations over an unchanged data structure. Since the order tends to be consistent, this behavior tends to reuse edges.
- Random behavior – vertices, edges, states and trace items are iterated randomly. This behavior simulates the most naïve approach, which chooses edges in a completely random manner. Since the order is not consistent, this behavior tends not to reuse edges.

6.4.1 Complete Graphs

The first experiment we performed involves complete graphs and a number of regular languages. The edges in these complete graphs are labeled either a, b, c, d, e and have weight $1, 2, 3, 4, 5$ respectively. The languages used are $a^*, (a|b)^*, (a|b|c)^*, (a|b|c|d)^*$ and $(a|b|c|d|e)^*$. The size of the graphs ranges from 2,500 edges, in experiments using graphs with 50 vertices and the language a^*, to 450,000 edges, in experiments using graphs with 300 vertices and the language $(a|b|c|d|e)^*$. Since the cheapest edges are the ones labeled a (cost 1), a^* is a subset of all languages, and all vertices are reachable from every other, the exact solution is always a cycle of edges labeled a connecting all vertices. The cost of this solution is equal to the number of vertices in the graph.

Figure 6.3 presents the results of this experiment. We can see six series of data in each chart, with an extra series of data indicating the approximation ratio for the optimum solution, which is always 1, painted green in the charts in Figure 6.3a.

Concerning the approximation ratio to the optimum solution (Figure 6.3a), both algorithms present a constant approximation ratio for alphabetical-sort and implementation-dependant behaviors, where algorithm RGM shows overall better results, reaching the optimum weight in most executions. We can see a clear discrepancy between the experiments with the random behavior and these mentioned above.

For the random behavior, both algorithms achieve a roughly linear approximation ratio, reaching worse values by orders of magnitude. Those results indicate that, for the future development of heuristics, the reuse of edges should take the same attention as their cost. Regarding time (Figure 6.3b), the algorithms present competitive results. For both algorithms, the implementation-dependant behavior shows better performance. This result was expected since this behavior does not spend time sorting or shuffling lists of vertices, edges, states or trace items. Last, for memory consumption (Figure 6.3c), we can see that RGM presents uniform results between behaviors, and consumes less than CFGM in all executions. For CFGM, the random behavior presents worse performance than the alphabetical-sort and implementation-dependant ones.

6.4.2 Fractal Triangles

The second experiment we performed involves fractal triangles (a.k.a. Sierpinski triangles) and the same regular languages from the previous experiment regular languages. A fractal triangle is composed by a number of recursive subdivisions. Figure 6.4 illustrates examples of fractal triangles. Fractal triangles simulate a better case than complete graphs, where the graph is sparser but all vertices are still reachable from each other. As in the previous experiment, the edges in these graphs are labeled either a, b, c, d, e and have weight $1, 2, 3, 4, 5$ respectively, and the languages used are $a^*, (a|b)^*, (a|b|c)^*, (a|b|c|d)^*$ and $(a|b|c|d|e)^*$. The size of the graphs ranges from 6 edges, in experiments using triangles with 0 subdivisions and the language a^*, to 7,290 edges, in experiments using triangles with 5 subdivisions and the language $(a|b|c|d|e)^*$. We do not know the optimum solution for these graphs, so we present the percentage value of the weight of the solution concerning its original weight. Figure 6.5 presents the results of this experiment.

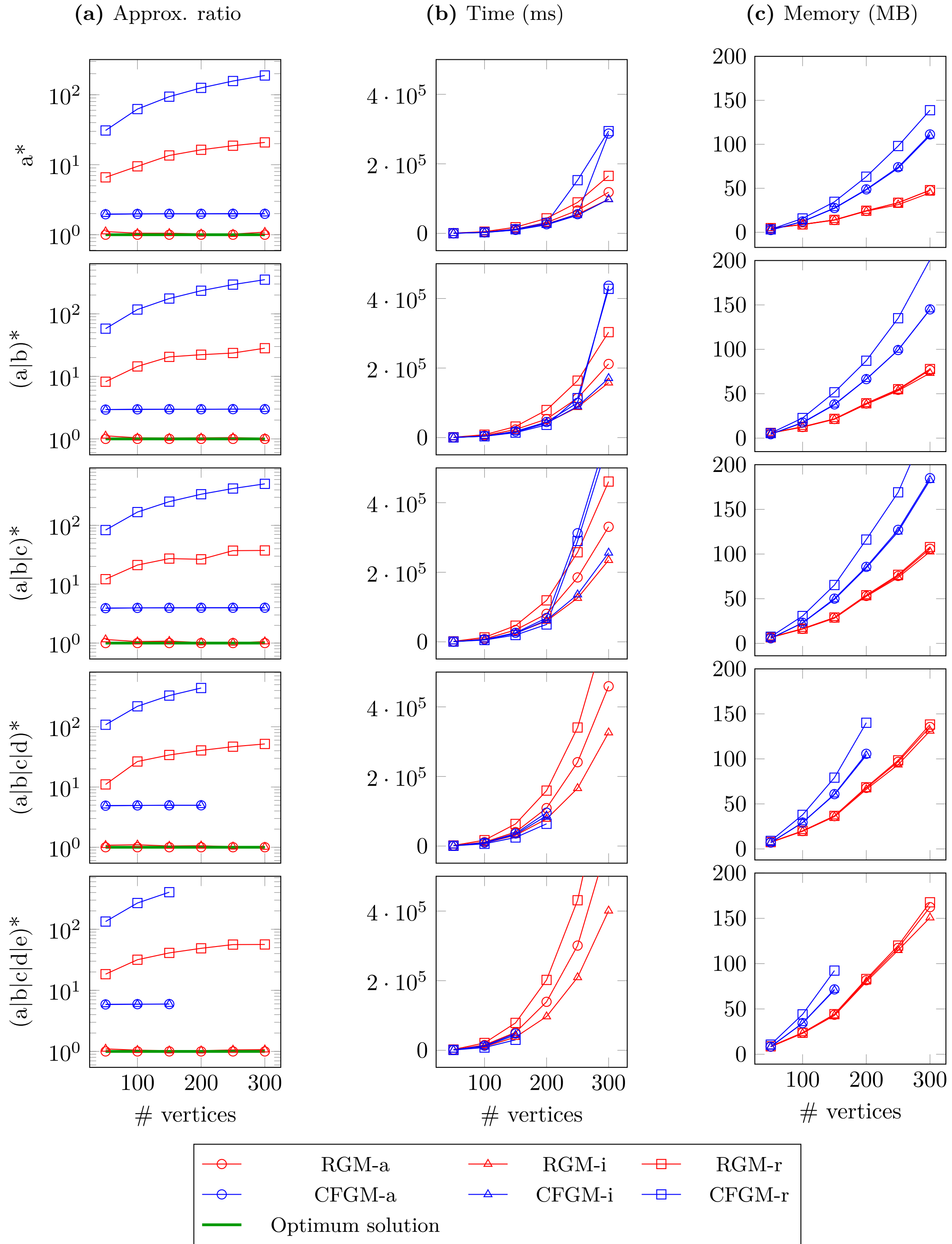
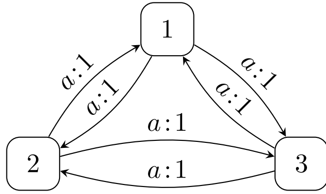
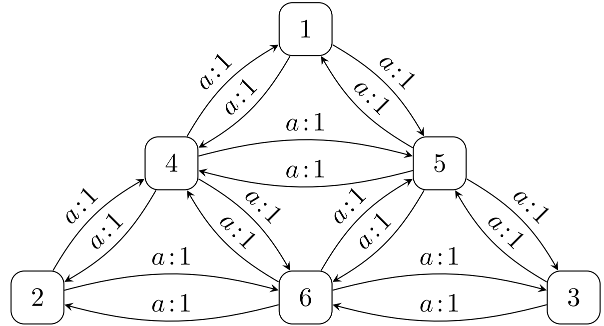


Figure 6.3: Experiments with complete graphs and alphabetical-sort (-a), implementation-dependant (-i), and random (-r) behaviors

(a) *a*-labeled fractal triangle with 0 subdivisions



(b) *a*-labeled fractal triangle with 1 subdivision



(c) *a*-labeled fractal triangle with 2 subdivisions

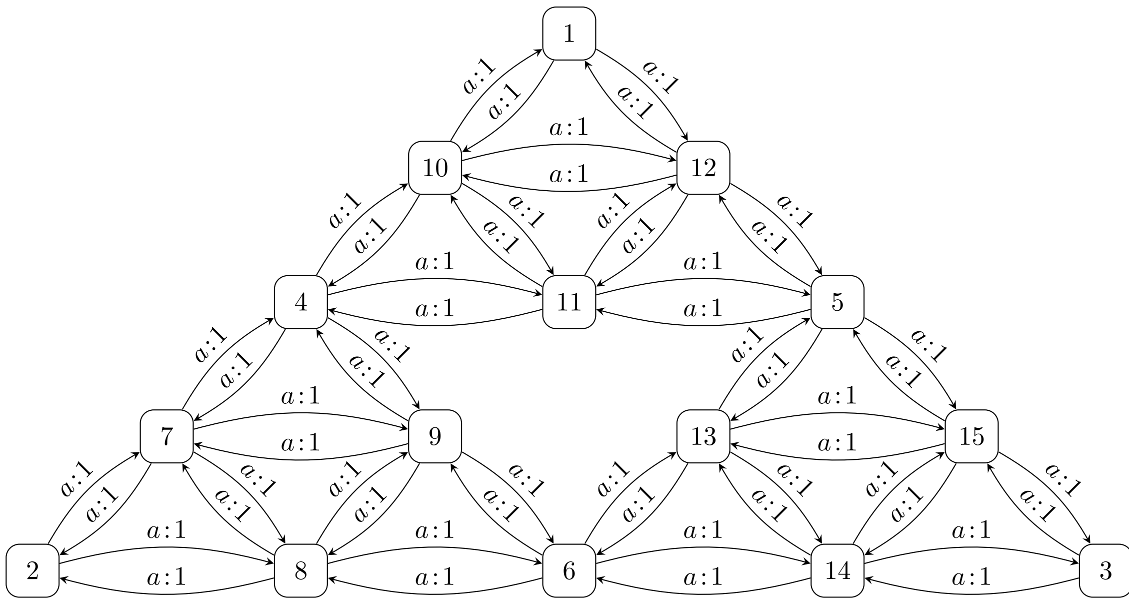


Figure 6.4: Example fractal triangles

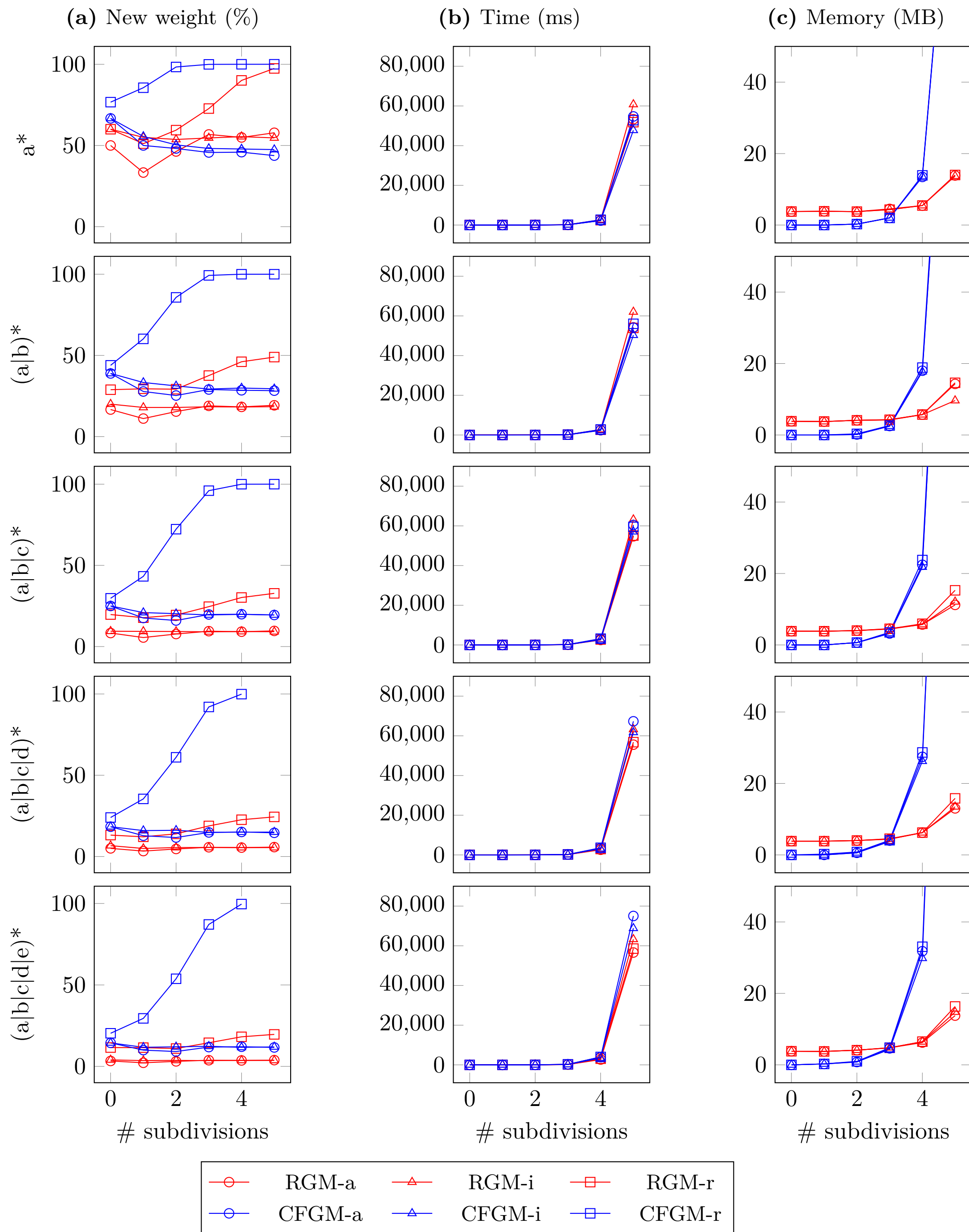


Figure 6.5: Experiments with fractal triangles and alphabetical-sort (-a), implementation-dependant (-i), and random (-r) behaviors

We can see six series of data in each chart. Concerning the percentage value of the solution's weight (Figure 6.5a), the alphabetical-sort and implementation-dependant behaviors produce better solutions than the random one. It is interesting to observe that the bigger the expression, the more the graph can be minimized. This is due to the fact that the graphs used in the experiments contain only the labels in the alphabet of that language, so more symbols mean that more edges can be deleted. Concerning time (Figure 6.5b), both algorithms using any of the implementations tend to behave similarly. The non-linear behavior, evidenced in experiments with triangles with 5 subdivisions, is due to the non-linear growth of the triangle graphs according to the linear increase in their number of subdivisions. Concerning memory consumption (Figure 6.5c), algorithm RGM presents a clear advantage over CFGM as the graphs scale.

6.5 Conclusions

We introduced and formalized a complex graph optimization problem and provided solutions for special cases. The Formal-Language-Constrained Graph Minimization problem consists of computing a minimal subgraph that preserves utility path queries defined with grammars. The FLGM problem is NP-hard and its decision version is NP-complete for regular and context-free grammars.

We presented two solutions to treat the regular and context-free graph minimization problems in polynomial-time. The idea behind these solutions is to iteratively add paths to the minimized graph until all utility queries are satisfied. We have developed prototypes to assert the feasibility of our techniques via experiments. The prototypes can behave in three different ways that simulate the use of heuristics. The experiments show that behaviors that tend to reuse edges strongly contribute for better solutions, with no substantial difference if they prefer cheaper edges or not. The CFGM (Algorithm 3) algorithm is more memory-consumptive than RGM (Algorithm 2). This is an expected behavior since, in string recognition, context-free recognizers generally use more memory than regular recognizers because they require a stack, which grows with the length of the recognition. The CFGM algorithm combines trace items with recursive function calls to simulate the stack.

As future work we want to improve the CFGM algorithm and its implementation, as well as find more real situations where the FLGM problem arises. We aim at developing a non-recursive version to avoid reaching Python's maximum recursion depth. Instead of relying on the function's recursive mechanism, we think of using stacks and trees to simulate such mechanism inside an iterative function. We are also thinking of a combination of the trace-items-based (Algorithm 1) and the CFGM algorithm to minimize the graph *during* the query stage. This combination should be able to return results faster and use less memory.

Thinking in terms of what the algorithm can do, we want to extend it to take as input not only formal languages and start vertices but also graph patterns with bound variables, similar to SPARQL queries. The minimization of graphs will be done to minimally match the graph patterns defined in the conjunctive query, which will include the language-based path patterns it already solves. This will require reasoning on dealing with homomorphisms to keep the less data possible that preserves the query's answers.

Better studying applications that can be modeled as the FLGM problem will give us clues on what needs to be improved in our solutions and provide us feedback on the design of realistic benchmarks. Higher theoretical complexity techniques might perform better for application-specific graph structures than those with lower theoretical complexity for general-purpose. Another question is on the graph model we are using. Undirected graphs, for example, might fit better in some situations.

Chapter 7

Final Remarks

In this thesis, we deal with three problems related to graphs and context-free languages: expressing context-free languages using alternative notations, context-free path querying, and minimizing graphs constrained to context-free languages. In the next sections, we make a summary of publications and future work derived from this research.

7.1 Publications Resulting from This Research

Four conference papers (including workshop and short papers) and one journal article resulted from this research:

In Medeiros *et al.* [37], we propose an extension of regular expressions, called recursive expressions, to support the definition of a subset of context-free languages. This work was later continued in Medeiros *et al.* [35], where we introduce the syntax of SM Expressions and present the semantics of their constructs in terms of Standard Matching-Choice Sets. Combined with a graph query language, SM Expressions can define path queries restricted to SM languages, a subset of context-free languages. We demonstrate how to obtain context-free grammars from SM Expressions and illustrate the usage of SM Expressions with examples.

In Medeiros *et al.* [40], we develop an original context-free path query evaluation mechanism that uses trace items. Trace items are grammar items annotated with graph vertices that allow us to recognize paths efficiently. We state the mechanism's time and space complexity, which are both asymptotically optimal when compared to other techniques from the literature, and demonstrate its correctness. We conduct a series of experiments to show its efficiency and applicability in practical cases. An extended version of this work [41] was published as a journal paper later on. The journal paper includes an extension of the prototype to support navigational axis and nested expressions [45] and a wide range of experiments.

In Medeiros *et al.* [36] we define the Formal-Language-Constrained Graph Minimization problem. This graph optimization problem consists of computing a minimal subgraph that preserves user-defined utility queries. We demonstrate its NP-hardness and NP-completeness for special cases. We extend this work by designing non-exact polynomial-time algorithms for the Regular- and Context-Free-Language-Constrained Graph Minimization problems. We demonstrate their time and space complexity and give an intuition on their correctness. Also, we implement prototypes to evaluate the feasibility of the techniques proposed via experiments. We are preparing an article to publish the new content.

7.2 Future Work

In this section, we present directions for future work for each of the problems treated in this thesis. These directions involve deeper investigations on the properties and usability of SM Expressions, improvements on the performance of the path query evaluation and graph minimization algorithms proposed, and the development of realistic benchmarks.

SM expressions still need formalization on some properties and assessing their usability. Analyzing

the complexity of the translating SM Expressions into context-free grammars, identifying the class of context-free languages that cannot be expressed with SM Expressions and comparing them with other proposals of non-regular expressions. It is equally important to run experiments with students and professionals from Computer Science and related areas to assess the usability of such expressions.

The performance of our trace-items-based algorithm (Algorithm 1) can be improved for dealing with larger graphs. Parallelism, better indexing techniques and better management of data to make use of disk memory to reduce the consumption of the main memory are desired improvements. These may improve performance and scalability of our prototype in many aspects: the treatment of unmarked vertices in position sets may be done in parallel; keeping the more data possible in the disk will allow it dealing with large graphs that do not fit in the main memory. Also, more investigation on how to use the information on trace items to reconstruct paths will allow it to implement single-path and all-path semantics efficiently.

The work on graph minimization is plenty of possibilities to be continued. Improving the algorithms and their corresponding prototypes will allow them to deal with larger graphs consuming less time and memory. It is important that the algorithms treat conjunctive queries to make it possible to express more complex utility queries. The minimization of graphs will be done to minimally match the graph patterns defined in the conjunctive query, which will include the language-based path patterns it already solves. This will allow new applications to make use of these graph minimization mechanisms.

Those directions open field for more research branches and for new query languages and applications making use of the new technologies developed to emerge.

Bibliography

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Alternative eText Formats Series. ADDISON WESLEY Publishing Company Incorporated, 2007.

[3] Albert, R., and Barabási, A.-L. Statistical mechanics of complex networks. *Rev. Mod. Phys. 74* (Jan 2002), 47–97.

[4] Alimonti, P., Ausiello, G., Giovaniello, L., and Protasi, M. On the complexity of approximating weighted satisfiability problems (extended abstract), 1997.

[5] Anciaux, N., Nguyen, B., and Vazirgiannis, M. Miminum Exposure in Classification Scenarios. Research report, 2011.

[6] Appel, A., and Palsberg, J. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002.

[7] Azimov, R., Epelbaum, I., and Grigorev, S. Context-free path querying with all-path semantics by matrix multiplication. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (New York, NY, USA, 2021), GRADES-NDA '21, Association for Computing Machinery.

[8] Azimov, R., and Grigorev, S. Context-free path querying by matrix multiplication. In *Proc. of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (New York, NY, USA, 2018), GRADES-NDA '18, ACM.

[9] Barrett, C., Jacob, R., and Marathe, M. Formal-language-constrained path problems. *SIAM Journal on Computing 30*, 3 (2000), 809–837.

[10] Bazgan, C., and Paschos, V. T. Differential approximation for optimal satisfiability and related problems. *European Journal of Operational Research 147*, 2 (2003), 397–404.

[11] Belhajjame, K. Lineage-preserving anonymization of the provenance of collection-based workflows. In *EDBT* (2020), pp. 229–240.

[12] Bessy, S., Fomin, F. V., Gaspers, S., Paul, C., Perez, A., Saurabh, S., and Thomassé, S. Kernels for feedback arc set in tournaments. *Journal of Computer and System Sciences 77*, 6 (2011), 1071–1078.

[13] Bessy, S., Paul, C., and Perez, A. Polynomial kernels for 3-leaf power graph modification problems. *Discrete Applied Mathematics 158*, 16 (2010), 1732–1744.

[14] Bessy, S., and Perez, A. Polynomial kernels for proper interval completion and related problems. vol. 231, pp. 89–108. Fundamentals of Computation Theory.

[15] Bodlaender, H. L. Kernelization: New upper and lower bound techniques. In *Parameterized and Exact Computation* (Berlin, Heidelberg, 2009), J. Chen and F. V. Fomin, Eds., Springer Berlin Heidelberg, pp. 17–37.

[16] Chattopadhyay, S., Paul, S., Dykhuizen, D. E., and Sokurenko, E. V. Tracking recent adaptive evolution in microbial species using timezone. *Nature Protocols 8*, 4 (2013), 652–665.

[17] Chomsky, N. Formal properties of grammars. *Handbook of Math. Psychology 2* (1963), 328–418.

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[19] Crespelle, C., Gras, B., and Perez, A. Completion to chordal distance-hereditary graphs: A quartic vertex-kernel. In *International Workshop on Graph-Theoretic Concepts in Computer Science* (2021), Springer, pp. 156–168.

[20] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[21] Dumas, M., Perez, A., and Todinca, I. Polynomial kernels for strictly chordal edge modification problems. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)* (2021), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[22] Freydenberger, D. Extended regular expressions: Succinctness and decidability. *Theory of Computing Systems 53*, 2 (2012), 159–193.

[23] Fu, Z., and Malik, S. Solving the minimum-cost satisfiability problem using sat based branch-and-bound search. In *2006 IEEE/ACM International Conference on Computer Aided Design* (2006), IEEE, pp. 852–859.

[24] Grigorev, S., and Ragozina, A. Context-free path querying with structural representation of result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia* (New York, NY, USA, 2017), CEE-SECR '17, ACM, pp. 10:1–10:7.

[25] Grune, D., and Jacobs, C. *Parsing Techniques: A Practical Guide.* Monographs in Computer Science. Springer New York, 2007.

[26] Hellings, J. Conjunctive context-free path queries. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014* (2014), N. Schweikardt, V. Christophides, and V. Leroy, Eds., OpenProceedings.org, pp. 119–130.

[27] Hellings, J. Path results for context-free grammar queries on graphs. *CoRR abs/1502.02242* (2015).

[28] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *science 220*, 4598 (1983), 671–680.

[29] Kuijpers, J., Fletcher, G., Yakovets, N., and Lindaaker, T. An experimental study of context-free path query evaluation methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management* (New York, NY, USA, 2019), SSDBM '19, Association for Computing Machinery, p. 121–132.

[30] Li, C.-M., Zhu, Z., Manya, F., and Simon, L. Minimum satisfiability and its applications. In *Twenty-Second International Joint Conference on Artificial Intelligence* (2011).

[31] Li, X. Y. *Optimization algorithms for the minimum-cost satisfiability problem.* PhD thesis, North Carolina State University, 2004.

[32] Li, Y., Zhang, Q., and Reps, T. Fast graph simplification for interleaved dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020), pp. 780–793.

[33] Liberatore, P. Algorithms and experiments on finding minimal models. Tech. rep., Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", 1999.

[34] Linz, P. *An Introduction to formal languages and automata (The fifth edition)*. Jones & Bartlett Publishers, 2012.

[35] Medeiros, C., Costa, U., and Musicante, M. Standard matching-choice expressions for defining path queries in graph databases. In *New Trends in Database and Information Systems* (Cham, 2021), L. Bellatreche, M. Dumas, P. Karras, R. Matulevičius, A. Awad, M. Weidlich, M. Ivanović, and O. Hartig, Eds., Springer International Publishing, pp. 97–108.

[36] Medeiros, C., Musicante, M., and Halfeld-Ferrari, M. The formal-language-constrained graph minimization problem. In *European Conference on Advances in Databases and Information Systems (Workshop)* (2021), Springer, pp. 139–145.

[37] Medeiros, C. M., da Costa, U. S., Grigorev, S. V., and Musicante, M. A. Recursive expressions for SPARQL property paths. In *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium* (2020), vol. 1260 of *Communications in Computer and Information Science*, Springer, pp. 72–84.

[38] Medeiros, C. M., Musicante, M. A., and Costa, U. S. Efficient evaluation of context-free path queries for graph databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2018), SAC '18, ACM, pp. 1230–1237.

[39] Medeiros, C. M., Musicante, M. A., and Costa, U. S. LL-based query answering over rdf databases. *Journal of Computer Languages 51* (2019), 75 – 87.

[40] Medeiros, C. M., Musicante, M. A., and Costa, U. S. An algorithm for context-free path queries over graph databases. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity* (New York, NY, USA, 2020), SBLP '20, Association for Computing Machinery, p. 40–47.

[41] Medeiros, C. M., Musicante, M. A., and Costa, U. S. Querying graph databases using context-free grammars. *Journal of Computer Languages 68* (2022), 101089.

[42] Miao, H., and Deshpande, A. Understanding data science lifecycle provenance via graph segmentation and summarization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (Los Alamitos, CA, USA, apr 2019), IEEE Computer Society, pp. 1710–1713.

[43] Mishin, N., Sokolov, I., Spirin, E., Kutuev, V., Nemchinov, E., Gorbatyuk, S., and Grigorev, S. Evaluation of the context-free path querying algorithm based on matrix multiplication. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (New York, NY, USA, 2019), GRADES-NDA'19, Association for Computing Machinery.

[44] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference* (2001), pp. 530–535.

[45] Pérez, J., Arenas, M., and Gutierrez, C. nsparql: A navigational language for RDF. *Web Semantics: Science, Services and Agents on the World Wide Web 8*, 4 (2010), 255 – 270. Semantic Web Challenge 2009User Interaction in Semantic Web research.

[46] Porschen, S. Solving minimum weight exact satisfiability in time o (2 0.2441n). In *International Symposium on Algorithms and Computation* (2005), Springer, pp. 654–664.

[47] Santos, F. C., Costa, U. S., and Musicante, M. A. A bottom-up algorithm for answering context-free path queries in graph databases. In *Web Engineering* (Cham, 2018), T. Mikkonen, R. Klamma, and J. Hernández, Eds., Springer International Publishing, pp. 225–233.

[48] Scott, E., and Johnstone, A. Gll parsing. *Electronic Notes in Theoretical Computer Science 253*, 7 (2010), 177 – 189. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[49] Sebastiani, R., Giorgini, P., and Mylopoulos, J. Simple and minimum-cost satisfiability for goal models. In *International Conference on Advanced Information Systems Engineering* (2004), Springer, pp. 20–35.

[50] Sempere, J. On a class of regular-like expressions for linear languages. *Journal of Automata, Languages and Combinatorics 5* (01 2000), 343–354.

[51] Sevon, P., and Eronen, L. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics (JIB) 5*, 2 (2008), 157–172.

[52] Spiegel, J., Dyck, M., and Robie, J. XML path language (XPath) 3.1. W3C recommendation, W3C, Mar. 2017. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[53] Susanina, Y. Context-free path querying via matrix equations. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 2821–2823.

[54] Terekhov, A., Khoroshev, A., Azimov, R., and Grigorev, S. Context-free path querying with single-path semantics by matrix multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (New York, NY, USA, 2020), GRADES-NDA'20, Association for Computing Machinery.

[55] Tomita, M. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[56] Valiant, L. G. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci. 10*, 2 (1975), 308–315.

[57] Winter, J., Bonsangue, M. M., and Rutten, J. Context-free languages, coalgebraically. In *International Conference on Algebra and Coalgebra in Computer Science* (2011), Springer, pp. 359–376.

[58] Yntema, M. Inclusion relations among families of context-free languages. *Information and Control 10*, 6 (1967), 572 – 597.

[59] Yntema, M. Cap expressions for context-free languages. *Information and Control 18*, 4 (1971), 311–318.

[60] Zhang, L., and Malik, S. The quest for efficient boolean satisfiability solvers. In *International conference on computer aided verification* (2002), Springer, pp. 17–36.

[61] Zhang, X., Feng, Z., Wang, X., Rao, G., and Wu, W. Context-free path queries on RDF graphs. In *International Semantic Web Conference (1)* (2016), vol. 9981 of *Lecture Notes in Computer Science*, pp. 632–648.

[62] Zheng, X., and Rugina, R. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, Association for Computing Machinery, p. 197–208.

Améliorations de Requêtes de Chemin Dans les Graphes : Expression, Évaluation et Satisfiabilité de Coût Minimal

Ciro MORAIS MEDEIROS

Directeur : M. MUSICANTE Martin (UFRN)
Co-Directrice: Mme HALFELD-FERRARI Mirian (UO)

Résumé Substantiel

1 Introduction Générale

Nous traitons trois problèmes liés aux bases de données en graphes et aux langages hors-contexte : exprimer des langages hors-contexte à l'aide de notations alternatives, évaluer des requêtes de chemins hors-contexte et minimiser les graphes contraints à des requêtes de chemins hors-contexte.

Les requêtes de chemin définissent des modèles pour faire correspondre les chemins dans un graphe étiqueté et orienté, de sorte que la chaîne formée par la concaténation des étiquettes appartient à un langage donné. Les langages de requête de graphe prennent généralement en charge les expressions régulières pour la définition des requêtes de chemin. Cependant, les expressions régulières ne suffisent pas à spécifier certaines propriétés importantes, comme les requêtes de même génération, où l'on veut trouver des paires de sommets au même niveau d'une hiérarchie donnée.

Pour définir des requêtes de chemins non réguliers, on peut utiliser des langages hors-contexte. Les langages hors-contexte sont généralement spécifiés par des grammaires hors-contexte, mais celles-ci sont en quelque sorte complexes et ne sont pas aussi populaires que les expressions régulières.

Les requêtes de chemins hors-contexte définissent des modèles de chemin en termes de grammaires hors-contexte. Ce type de requête est intéressant dans des domaines tels que la génétique, la science des données et l'analyse de code source. Les requêtes de chemins hors-contexte peuvent répondre, par exemple, à des requêtes de même génération.

Une réponse à une requête de chemins peut provenir de plusieurs sous-graphes de la base de données. Une telle redondance est parfois indésirable, que ce soit pour des questions de stockage, de confidentialité ou de coût. La satisfaisabilité de poids / coût minimum dans les bases de données de graphes peut réduire la quantité de données de manière à préserver les réponses à certaines requêtes données.

Dans ce travail, nous: (1) proposons une notation alternative pour exprimer des langages hors-contexte; (2) développons un algorithme d'évaluation de requêtes de chemins hors-contexte; e (3) développons des algorithmes pour la minimisation de bases de données en graphe constraint à un langage hors-contexte et régulier.

Nous réalisons des expériences avec des prototypes des algorithmes développés ici. Le code source ainsi que les bases de données utilisés dans les expériences sont accessibles au public dans notre référentiel en ligne[1].

[1] Référentiel git en ligne contenant le code source et les ensembles de données utilisés dans les expériences : `https://gitlab.com/ciromoraismedeiros/rdf-ccfpq`.

2 Formalisme

Avant de présenter les contributions de notre travail, nous présentons les définition des concepts les plus importants dans ce contexte.

2.1 Langages Formels

Un *language* est un ensemble de chaînes de caractères construit à partir d'un alphabet fini. Un langage peut être defini par une *grammaire*.

Definition 1 (Grammaire). *Une* grammaire *est une quadruple $G = (N, \Sigma, P, S)$ où N est l'ensemble de symboles non-terminels, Σ est l'ensemble de symboles terminels (alphabet), P est l'ensemble de règles de production dans la forme $\alpha \to \beta$, où $\alpha \in (N \cup \Sigma)^+$ et $\beta \in (N \cup \Sigma)^*$, et $S \in N$ est le symbole initial.*

Étant données une chaîne $\alpha\delta\gamma$, où $\delta \in (N \cup \Sigma)^+$, et une règle de production $\delta \to \beta$, on peut appliquer cette règle pour produire la chaîne $\alpha\beta\gamma$, noté par $\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$. Si, pour une chaîne s donnée, on peut appliquer des règles de production successivement et générer une chaîne s', noté par $s \Rightarrow^* s'$, on dit que s' est dérivable depuis s.

Étant donné une grammaire et une chaîne, la *reconnaissance de chaînes* consiste à vérifier si la chaîne appartient au langage généré par la grammaire. Il existe une large gamme d'algorithmes pour la reconnaissance de chaînes, variant en complexité et en puissance de calcul. Chaque algorithme de reconnaissance est adapté à une classe de langages.

Dans ce travail, les classes de grammaires hors-contexte et régulières reçoivent plus d'attention. Les règles de production dans une grammaire *hors-contexte* sont de la forme $X \to \gamma$, où X est un symbole non-terminel et γ est une chaîne posiblement vide (noté par ε) de symboles terminels et non-terminels. Les règles de production dans une grammaire *régulière* sont de la forme $X \to a$ ou $X \to a\ Y$, où Y est un symbole non-terminel et a, terminel. La règle vide n'est autorisée que si X est le symbole de début. Les grammaires régulières sont moins expressives, mais sont plus simples et peuvent être évaluées plus rapidement. Les expressions régulières sont une notation alternative répandue pour les grammaires régulières. Les grammaires hors-contexte sont plus expressives que les grammaires régulières, mais elles augmentent la puissance de calcul nécessaire à la reconnaissance des langages associés. Bien que certaines initiatives proposent des notations plus pragmatiques pour exprimer des langages hors-contexte, la notation la plus populaire est les grammaires hors-contexte (CFG).

2.2 Graphes

Dans cette section, nous formalisons les concepts liés aux graphes et aux requêtes de chemin. Pour illustrer ces définitions, nous nous référons au graphe D de la Figure 1 et à la grammaire G donnée par $\{S \to a\ S\ b, S \to \varepsilon\}$, qui définit le langage $a^n b^n$.

Definition 2 (Graphe). *Un* graphe *(aussi appelé* graphe de données *ou* base de données en graphe*) est un ensemble de triplets dans $V \times L \times V$, où V est un ensemble de sommets et L est un ensemble*

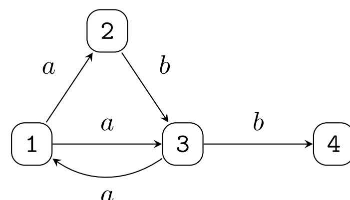


Figure 1: Graphe exemple

d'étiquettes d'arêtes. Nous appelons sous-graphe *tout sous-ensemble d'un graphe.*
Le graphe de la Figure 1 est $D = \{(1, a, 2), (1, a, 3), (2, b, 3), ...\}$.

Les graphes peuvent avoir des arêtes étiquetées non terminales. Par exemple, l'arête (a, S, b) indique qu'il existe un chemin dérivable depuis S entre a et b. Nous nous référons aux graphes qui contiennent de telles arêtes étiquetées non terminales comme des *graphes annotés*.

Les triples peuvent être organisées en sequence pour former un *chemin*. Par exemple, le chemin $\pi = ((1, a, 2), (2, b, 3), (3, b, 4))$ sur la Figure 1 lie les sommets 1 et 4. La concaténation des étiquettes d'un chemin forme une chaîne qui peut appartenir à un langage. Une grammaire peut être utilisée pour interroger des chemins dans un graphe. Cela nous donne la définition suivante.

Definition 3 (Requête de chemins hors-contexte)**.** *Étant donné un graphe D et une grammaire hors-contexte G, une* requête de chemins hors-contexte *(CFPQ) Q est un ensemble de paires (x, A) où $x \in V$ et $A \in N$. L'évaluation d'une requête de chemins hors-contexte Q produit l'ensemble de tous les sommets y tels qu'il existe un chemin de x à y dérivable depuis A.*
Dans notre exemple, la requête $(1, S)$ a des réponses $\{1, 3, 4\}$.

Les prochaînes sections présentent les travaux développés dans cette thèse.

3 Expressions SM pour la Définition de Requêtes de Chemin

3.1 Introduction

La plupart des langages de requête en graphes utilisent des *expressions régulières* pour définir des requêtes de chemin. Cependant, les expressions régulières ne peuvent pas spécifier certaines propriétés importantes sur les bases de données de graphes, telles que les requêtes de même génération. Cela caractérise une requête de chemins hors-contexte. La définition des langages de spécification pour les requêtes de chemins hors-contexte est importante car elle impacte l'utilisation pratique des bases de données en graphes. Nous présentons les *Expressions de Choix Correspondants Standard (Expressions SM)* pour la spécification des requêtes de chemins non réguliers.

Nos expressions sont inspirées par la famille des Ensembles de Choix Correspondants Standard, un sous-ensemble significatif des langages hors-contexte, présenté par Yntema [2]. Nous définissons la syntaxe et la sémantique des expressions SM, et nous formalisons la traduction des expressions SM en un ensemble de règles d'une grammaire hors-contexte. La traduction des expressions SM en grammaires hors-contexte comble le vide entre un langage de spécification plus naturel pour interroger les bases de données en graphes et les moteurs de requête existants pour les CFPQ.

Yntema définit une hiérarchie d'inclusion de classes de langages hors-contexte qui peuvent être définis à l'aide d'opérations sur des ensembles de chaînes. Étant donné un alphabet Σ, les blocs de construction pour définir ces langages sont *(i)* un ensemble de paires de chaînes $S \subseteq \Sigma^* \times \Sigma^*$ et *(ii)* un ensemble de chaînes $C \subseteq \Sigma^*$.

L'ensemble des paires S peut être vu comme une généralisation des parenthèses ouvrantes et fermantes. L'ensemble de chaînes C contient des chaînes qui seront entourées par celles de S. Les langages construits à partir de ces ensembles contiennent des chaînes $\alpha\beta\gamma$ telles que $(\alpha, \gamma) \in S$ et $\beta \in C$. Étant donné des ensembles de paires de chaînes S_1 et S_2, l'auteur introduit les *Matching-Choice Sets* définis sur S_1 et S_2

[2]Yntema, M. *Inclusion relations among families of context-free languages.* Information and Control10, 6 (1967), 572 – 597

comme suit :

$$S_1 \oplus S_2 = \{(x, y) \mid (x, y) \in S_1 \vee (x, y) \in S_2\} \tag{1}$$
$$S_1 S_2 = \{(xz, wy) \mid (x, y) \in S_1 \wedge (z, w) \in S_2\} \tag{2}$$
$$S_1* = \{(\varepsilon, \varepsilon)\} \cup \{(x_1 \cdots x_n, y_n \cdots y_1) \mid (x_i, y_i) \in S_1, i \leq n, n \in \mathbb{N}^+\} \tag{3}$$

À l'aide de ces opérations, nous pouvons maintenant présenter les langages de choix correspondants standard.

Definition 4 (Langages de Choix Correspondants Standard)**.** *Étant donné un ensemble de choix correspondant S (c'est-à-dire, un ensemble de paires construites à l'aide des opérations ci-dessus) et un ensemble de chaînes C, le* Langage de Choix Correspondants Standard *$S \circ C$ est défini comme suit :*

$$S \circ C = \{xzy \mid (x, y) \in S \wedge z \in C\}$$

Étant donné un nombre fini d'ensembles de chaînes A_i, B_i ($1 \leq i \leq n$), nous pouvons construire un ensemble S de paires de chaînes par *(i)* définissant les ensembles de paires $A_i \times B_i = S_i$, et *(ii)* appliquant récursivement les opérations d'union (1), de séquence (2) et de fermeture (3 sur les ensembles S_i. Notez que les éléments dans S peuvent être vus comme des paires de parenthèses correspondantes tandis que les chaînes dans C correspondent à des chaînes placées entre eux. Les opérations union (1), séquence (2) et étoile (3) garantissent que l'ensemble $S \circ C$ est formé par des chaînes contenant des expressions entre parenthèses bien formées.

Yntema définit la classe des langages SM, un sous-ensemble approprié de langages hors-contexte qui peuvent être décrits à l'aide des opérations mentionnées ci-dessus. Dans la section suivante, nous définissons un ensemble d'expressions pour désigner les ensembles de choix correspondants standard.

3.2 Expressions de Choix Correspondants Standard

Dans cette section, nous proposons les expressions SM pour définir les Langages de Choix Correspondants Standard. Les expressions régulières sont à la base de la définition des expressions SM. Étant donné un alphabet $\Sigma = \{t_1, \ldots, t_n\}$, l'ensemble des expressions régulières sur Σ est l'ensemble des chaînes défini par $R \rightarrow () \mid t_1 \mid \ldots \mid t_n \mid (R) \mid RR \mid R|R \mid R*$.

Definition 5 (Syntaxe des expressions SM)**.** *L'ensemble des expressions SM sur un alphabet Σ est défini de manière inductive comme suit :*

1. *Toute expression régulière E sur Σ est une expression SM.*

2. *Expression SM base :*

$$\frac{E, E_0, E' \text{ sont des expressions SM}}{<E>E_0<E'> \text{ est une expression SM}}$$

3. *Parenthèses imbriquées :*

$$\frac{<P_1>E_0<P_1'> \text{ est une expression SM} \qquad <P_2>E_0<P_2'> \text{ est une expression SM}}{<P_2.P_1>E_0<P_1'.P_2'> \text{ est une expression SM}}$$

4. *Choix de parenthèses :*

$$\frac{<P_1>E_0<P_1'> \text{ est une expression SM} \qquad <P_2>E_0<P_2'> \text{ est une expression SM}}{<P_2 + P_1>E_0<P_1' + P_2'> \text{ est une expression SM}}$$

Description	**Ensembles SM**	**Expression SM**
Nombre égal de a's et de b's.	$(\{(a,b),(b,a)\} * \circ\{\epsilon\})^*$	`(<:a+b:><:a+b:>)*`
$b^n ab^{2n}$	$\{(b,bb)\} * \circ\{a\}$	`<:b:>a<:bb:>`
Langage $\{\texttt{boss}^n\ \texttt{boss}^{-1n} \mid n \geq 0\}$, pour une requête de même génération.	$(\{boss\} \times \{boss^{-1}\}) * \circ\{\varepsilon\}$	$\texttt{<: boss :><: boss}^{-1}\texttt{ :>}$
$a^{n+1}b^m c^m d^{n+1}$	$a\{(a,d) * \circ(\{(b,c)\} * \circ\{\varepsilon\})\}d$	`a<:a:>(<:b:><:c:>)<:d:>d`

Table 1: Exemples d'Expressions SM

5. *Récursion :*

$$\frac{\texttt{<}P\texttt{>}E_0\texttt{<}P'\texttt{>} \text{ est une expression SM}}{\texttt{<:}\ P\ \texttt{:>}E_0\texttt{<:}\ P'\ \texttt{:>} \text{ est une expression SM}}$$

Notez que les chaînes P_i *dans les cas ci-dessus ne sont, en général,* pas *des expressions SM, mais elles dénotent un langage d'ouverture de parenthèses. De manière analogue,* P_i' *représentent des ensembles de parenthèses fermantes. Ces chaînes peuvent contenir des caractères comme "."*, *"+" et ":", qui sont utilisés pour construire les langages qui dénotent les parenthèses ouvrantes et fermantes.*

Le Tableau 1 réunit des exemples d'expressions SM et leur ensemble SM correspondant.

3.3 Conclusion

Nous avons présenté une notation pour spécifier les langages de la famille des Ensembles de Choix Correspondants Standard. Cette famille de langages est un sous-ensemble des langages hors-contexte, construit autour de la notion du parenthésage des chaînes. Les expressions SM sont une notation alternative aux grammaires hors-contexte pour définir des requêtes de chemins hors-contexte. Les expressions SM peuvent être directement traduites en règles de grammaires hors-contexte, ce qui leur permet d'être utilisés avec la plupart des moteurs d'évaluation de requêtes de chemins hors-contexte disponibles dans la littérature.

Notre travail peut être prolongé en considérant différentes perspectives : (*i*) analyser les coûts asymptotiques associés à la traduction d'expressions SM en grammaires hors-contexte ; (*ii*) identifier la classe des requêtes de chemins hors-contexte qui ne peuvent pas être exprimées avec des expressions SM ; (*iii*) comparer l'expressivité des expressions SM avec d'autres propositions d'expressions non régulières ; (*iv*) enquêter sur l'utilisation des expressions SM dans le cadre d'autres langages de spécification non réguliers ; et (*v*) évaluer l'utilisabilité des expressions SM via des expériences avec des étudiants et des professionnels de l'informatique et des domaines connexes.

4 Évaluation des Requêtes de Chemins Hors-Contexte

4.1 Introduction

Dans cette section, nous présentons notre approche pour l'évaluation des requêtes de chemins hors-contexte (CFPQs). Notre algorithme prends comme paramètre d'entrée une grammaire, un graphe et une requête, et reconnaît les chemins hors-contexte dans le graphe. Le but de l'algorithme est d'identifier des paires de sommets reliés par des chemins qui forment des chaînes dans le langage de la grammaire.

Notre algorithm utilise une structure de données specifique pour contrôler le processus d'évaluation de chemins, presentée dans la définition suivante.

Definition 6 (Item de Trace)**.** *Étant donné une grammaire hors-contexte $G = (N, \Sigma, P, S)$ et un graphe D, un* item de trace *est une paire formé par une règle de production et une fonction associant un ensemble de nœuds de graphe à chaque position du côté droit de la règle. Formellement, un item de trace est défini comme la paire $(A \to \alpha, f)$, où $A \to \alpha \in P$ et $f\ : \{0, \dots, |\alpha|\} \to \mathcal{P}(V)$. Les items de trace sont uniques en ce qui concerne leur sommet de départ et leur règle de production.*

Nous noterons l'item de trace $(A \to \alpha_1, \dots, \alpha_n, f)$, où $f = \{0 \mapsto C_0, \dots, n \mapsto C_n\}$ comme $[A \to C_0\ \alpha_1\ C_1\ ...\ \alpha_n\ C_n]$. Les ensembles $C_1, \dots, C_n$ seront appelés ensembles de position.

Le premier ensemble de position C_0 dans un item de trace est un singleton. Étant donné deux ensembles de position C_1, C_2 et un symbole grammatical α, une séquence $C_1\ \alpha\ C_2$ sur le côté droit d'un item indique que chaque sommet dans C_2 est accessible depuis au moins un sommet dans C_1 via un chemin dérivable depuis α. Par exemple, l'item de trace $[\ S \to \{1\}\ a\ \{2, 3\}\ S\ \{4\}\ b\ \{\ \}\ ]$ indique que le processus d'analyse est à un stade où des chemins dérivables depuis a reliant le sommet 1 aux sommets 2 et 3 dans le graphe ont été identifiés.

4.2 L'Algorithm Basée Sur les Items de Trace

Dans cette section, nous présentons l'Algorithm 1 pour l'évaluation des CFPQs basés sur les items de trace. Cet algorithme utilise des marques spéciales $^\bullet$ et $^\circ$, respectivement, pour les sommets traités et non traités dans les ensembles de positions, pour garder l'histoire des sommets qui ont déjà été traités. Nous omettons ces marques lorsqu'une telle distinction n'est pas nécessaire. L'opérateur $\Join$ est utilisé pour effectuer des unions entre des ensembles de sommets traités et non traités, et il est défini comme :

$$C \Join \{x^\circ\} \quad = \quad \begin{cases} C, & \textit{si } x^\bullet \in C \\ C \cup \{x^\circ\}, & \textit{au cas contraire} \end{cases}$$

où C est un ensemble de positions et x° est un sommet non marqué. Si le vertex x a déjà été traité, il est conservé marqué dans l'ensemble de position C. Sinon, il y est ajouté non marqué.

ALGORITHM 1 : L'Algorithme Basé Sur les Items de Trace

Input : Une grammaire G, une requête Q, et un graphe D
Output : Un graphe annoté $D' \subseteq V \times (\Sigma \cup N) \times V$ et un ensemble d'items de trace I

```
1  function eval
2    I := {[A → {w°} α_1 { } ... α_n { }] | A → α_1...α_n ∈ P ∧ (w, A) ∈ Q}
3    D' := D
4    while ∃i, x s.t. i = [A → ... {x°, ...} ...] ∈ I do
5      switch i
6      case i = [A → ... {x°, ...} α_k C_k ...] do
7        if α_k ∈ Σ ∨ [α_k → {x} ...] ∈ I then
8          C_k := C_k ⋈ {y° | (x, α_k, y) ∈ D'}
9        else
10         I := I ∪ {[α_k → {x°} β_1 { } ... β_n { }] | α_k → β_1 ... β_n ∈ P}
11     case i = [A → {w} ... {x°, ...}] do
12       D' := D' ∪ {(w, A, x)}
13       foreach [B → ... {w•, ...} A C ...] ∈ I do
14         C := C ⋈ {x°}
15     marquer(x, i)
16   return D', I
```

L'algorithme 1 manipule deux structures de données : un ensemble d'items de trace I et un graphe D' contenant le graphe d'origine D annoté de manière incrémentielle avec de nouvelles arêtes non-terminelles. Pour calculer les réponses aux requêtes dans Q, l'algorithme commence par créer des items de trace à

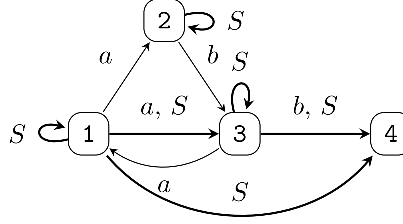


Figure 2: Graphe Résultat

partir de ses paires de sommets et de règles de grammaire : pour chaque paire $(v, A) \in Q$, il crée un item de trace pour chaque règle de production du non-terminal A avec v dans le premier ensemble de position (line 2). Cela prépare l'algorithme à entrer dans la boucle principale qui traite les sommets non marqués dans les éléments de I.

Il existe deux cas lors du traitement de sommets non marqués :

1. Dans le premier cas de la boucle principale (lines 6-10), étant donné l'item de trace $i = [A \rightarrow C_0\ \alpha_1\ C_1\ ...\alpha_n\ C_n]$, x° appartient à un ensemble de positions C_{k-1} qui n'est pas dans la dernière position de l'item de trace. Il existe trois sous-cas :

   (a) Si $\alpha_k \in \Sigma$, tous les sommets y° tels qu'il existe une arête $(x, \alpha_k, y) \in D'$ sont ajoutés à C_k (ligne 8);

   (b) Si $\alpha_k \in N$ et $[\ \alpha_k \rightarrow \{x\} \dots\ ] \in I$, tous les sommets y° tels qu'il y ait une arête $(x, \alpha_k, y) \in D'$ sont ajoutés à C_k (ligne 8) ;

   (c) Si $\alpha_k \in N$ et il n'y a pas d'item de trace $[\ \alpha_k \rightarrow \{x\ ...\}\ ...\ ]$, Algorithm 1 lance la recherche de dérivations depuis α_k commençant à x. Cela se fait en créant de nouveaux items de trace $\alpha_k \rightarrow \{x^\circ\} \dots$ et en les ajoutant à I (ligne 10).

2. Dans le second cas de la boucle principale (lignes 11 à 14), le sommet x appartient au dernier ensemble de position d'un item de trace. L'item de trace $i = [A \rightarrow \{w\} \dots \{x^\circ, ...\}]$ indique qu'il existe un chemin du sommet w à x en D'. Ainsi, Algorithm 1 génère une arête étiquetée A reliant ces deux sommets (ligne 12). Après cette opération, tous les ensembles de position C tels que $[\ B \rightarrow \dots \{w, \dots\}\ A\ C \dots\ ] \in I$ sont mis à jour avec x° (ligne 14).

À la fin du corps de la boucle principale, le sommet x° est marqué (ligne 15). La condition d'arrêt de cette boucle est l'absence de sommets non marqués dans tous les ensembles de positions. Le graphe annoté D' est retourné à la fin (ligne 16).

Étant données la grammaire G avec des règles de production $P = \{S \rightarrow a\ S\ b, S \rightarrow \varepsilon\}$, le graphe D dans la Figure 1 et la requête $Q = \{(1, S), (3, S)\}$, notre algorithme calcule les solutions représentées par des flèches en **gras** et étiquetées par des non-terminaux dans la Figure 2. L'ensemble d'items de trace final est :

$$[\ S \rightarrow \{1^\bullet\}\ a\ \{2^\bullet, 3^\bullet\}\ S\ \{2^\bullet, 3^\bullet, 4^\bullet\}\ b\ \{3^\bullet, 4^\bullet\}\ ], \quad [\ S \rightarrow \{1^\bullet\}\ ],$$
$$[\ S \rightarrow \{2^\bullet\}\ a\ \{\ \}\ S\ \{\ \}\ b\ \{\ \}\ ], \quad [\ S \rightarrow \{2^\bullet\}\ ],$$
$$[\ S \rightarrow \{3^\bullet\}\ a\ \{1^\bullet\}\ S\ \{1^\bullet, 3^\bullet, 4^\bullet\}\ b\ \{4^\bullet\}\ ], \quad [\ S \rightarrow \{3^\bullet\}\ ]$$

Les complexités en space et en temps, respectivement, de l'Algorithm 1 dans le pire des cas est de $\mathcal{O}(|V|^2 \cdot |P| \cdot k)$ et $\mathcal{O}(|V|^3 \cdot |P|^2 \cdot k^2)$.

4.3 Conclusion

Nous avons présenté un algorithme pour l'évaluation des requêtes de chemins hors-contexte pour les bases de données en graphe. Nous avons analysé sa complexité de temps et en espace dans le pire des cas.

Nous avons conduit une série d'expériences pour valider la viabilité de notre technique. Nous avons développé cinq implémentations qui diffèrent par l'algorithme implémenté, le langage de programmation, la représentation des données, entre autres aspects. Nos expériences montrent que notre algorithme surpasse d'autres algorithmes disponibles dans la littérature.

Comme travaux futurs, nous avons l'intention d'étudier une version parallèle de notre algorithm. Une meilleure gestion des grands graphes qui ne rentrent pas dans la mémoire est également une amélioration souhaitée. Une autre caractéristique souhaitable consiste à utiliser les informations sur les items de trace pour reconstruire les chemins.

Nous avons aussi l'intention de travailler sur la définition d'un benchmarking pour les algorithmes d'évaluation des CFPQs. Cela permettra d'avoir des données plus précises pour comparer les différents algorithmes qui implémentent ces requêtes.

5 Minimisation de Graphes Contrainte par un Langage Formel

5.1 Introduction

Dans ce chapitre, nous présentons le problème de la Minimisation de Graphes Contrainte par un Langage Formel (FLGM). Nous proposons des solutions pour les versions régulières et hors-contexte de ce problème.

Ce problème consiste à supprimer d'un graphe d'entrée autant de triplets que possible, de manière à ce que les réponses à une requête d'utilité définie par l'utilisateur soient préservées. Le problème FLGM apparaît dans les scénarios de minimisation de graphes, tels que la minimisation de réseaux, l'analyse de code source et autres.

Minimiser un graphe D consiste à identifier un sous-ensemble $D^- \subseteq D$ dont le poids (ou le coût) est minimum, et il reste au moins un chemin valide pour chaque réponse d'une requête donnée. Cet ensemble de requêtes est utilisé pour préserver l'utilité des données.

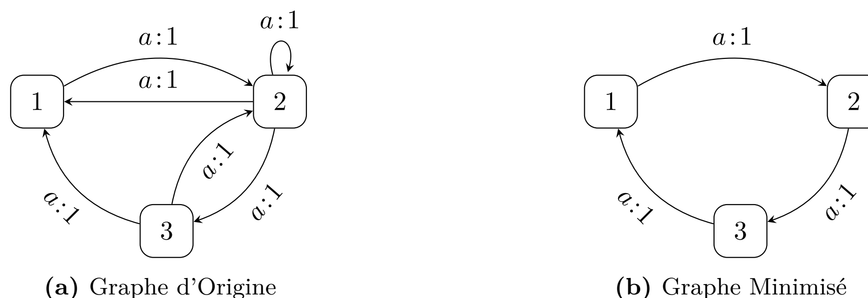
Definition 7 (Problème de Minimisation de Graphe Contrainte par un Langage Formel)**.** *Étant donné une grammaire G, un graphe D avec des poids sur les arêtes, un ensemble de requêtes Q et une fonction de poids (ou de coût) f, le problème de la* Minimisation de Graphe Contrainte par un Langage Formel *(FLGM) consiste en identifier un sous-graphe $D^- \subseteq D$ tel que pour toutes les requêtes $(a, X) \in Q$, aient leur réponses préservées et $f(D^-)$ est le minimum.*

Ce problème d'optimisation est NP-difficile, et sa version de décision est NP-complète pour les langages décidables en temps polynomial.

Dans les sections suivantes, nous présentons des algorithmes heuristiques pour résoudre le problème de minimisation de graphes pour un sous-ensemble de langages. Nos techniques sont composées de deux étapes : l'évaluation d'une requête de chemin et la construction d'un graphe minimum. Puisque la reconnaissance de chaînes est elle-même une tâche complexe, nous restreignons les grammaires utilisées pour définir les chemins aux classes de grammaires régulières et hors-contexte. Ces classes nous permettent de couvrir non seulement le large éventail d'applications qui utilisent des langages réguliers, mais aussi celles qui nécessitent des langages hors-contexte. Ainsi, nous présentons une solution aux problèmes de la *Minimisation de Graphes Contrainte à un Langage Régulier* et *Hors-Contexte*. Par des contraintes d'espace, nous ne présentons pas leur pseudo-code.

5.2 Algorithms pour le Problème FLGM

Nos algorithmes de minimisation de graphes contrainte par un langage régulier et hors-contexte s'appellent, respectivement, *RGM* et *CFGM*. Ces algorithmes se comportent de façon similaire.



(a) Graphe d'Origine **(b)** Graphe Minimisé

Figure 3: Exemple de Minimisation de Graphe Contrainte à Un Langage Formel

Les algorithmes RGM et CFGM créent des structures de données auxiliaires qui décrivent les chemins dans le graphe. Ces structures de données leur permetent de trouver, pour chaque réponse aux requêtes, des chemins qui appartiennent au langage formel spécifié. L'algorithme RGM construit un automate produit, lors que l'algorithme CFGM construit un ensemble d'items de trace.

Construite leur structure de données respective, les algorithmes choisissent un chemin pour chaque réponse aux requêtes et ajoutent les arêtes de ce chemin au graphe minimum. De cette façon, il est garanti que toutes les réponses aux requêtes séront préservées, puisqu'il y a au moins un chemin valide pour chaque réponse.

Il peut y avoir plusieurs solutions de coût minimial pour une instance du problème FLGM. Comme ce problème est NP-complet, un algorithme ne peu pas garantir de trouver la meilleure solution. Nos algorithmes utilisent une heuristique gloutonne. Ça veut dire que, à chaque instant, ils choisissent les chemins moins chèrs pour trouver des solutions les plus proches possibles des meilleures.

Nous présentons un exemple d'instance du problème FLGM. Soit D le graphe de la Figure 3a, G la grammaire donnée par $S \to a\ S, S \to \varepsilon$, $Q = \{(1, S), (2, S), (3, S)\}$ l'ensemble des requêtes, et f une fonction de somme simple des poids des arêtes. Tous les sommets sont accessibles à partir de tous les autres via un chemin dérivable depuis S. Une possible solution pour cette instance est présenté dans la Figure 3b. Notez que tous les sommets restent toujours accessibles à partir de tous les autres via un chemin dérivable depuis S. Les réponses sont donc préservées, malgré l'effacement de plusieurs arêtes du graphe.

Les complexités de temps et d'espace de l'algorithme RGM sont respectivement $\mathcal{O}(|Q| * |D| * |A|)$, et $\mathcal{O}(|D| * |A|)$, où Q est l'ensemble des requêtes, D est le graphe et A est l'automate construit à partir de la grammaire donnée. Les complexités de temps et d'espace de l'algorithme CFGM sont respectivement $\mathcal{O}(|V|^3)$ et $\mathcal{O}(|V|^2)$, où V est l'ensemble des sommets du graphe D.

5.3 Conclusion

Nous avons défini le problème FLGM et développé des algorithmes pour des cas spéciaux. Le problème FLGM est NP-difficile, et sa la version de décision est NP-complète pour des langages hors-contexte.

Nous avons développé deux algorithmes pour résoudre ce problème quand les langages sont réguliers ou hors-contexte. Nous avons conduit des expéciences avec des prototypes des deux algorithmes. Les résultats montrent que, dans nos expériences, l'algorithme RGM atteint un facteur d'approximation de 1 (solution optimale) pour les graphes complets, où il y a, pour chaque étiquette, une arête entre toutes les paires de sommets. Le facteur d'approximation de l'algorithme CFGM varie entre 2 et 6. En termes de performances de temps et de consomation de mémoire, l'algorithme RGM est aussi supérieur. Par contre, nous savons que l'algorithme CFGM est capable de processer une classe de langages plus grande que celle de l'algorithme RGM.

Comme travaux futurs, nous voulons améliorer l'algorithme CFGM et son implémentation, ainsi que

trouver des situations plus réelles où le problème FLGM se pose. En pensant en termes de ce que les algorithmes peuvent faire, nous voulons les étendre pour prendre en entrée non seulement des langages formels et des sommets de départ, mais aussi des modèles de graphe avec des variables, similaire aux requêtes SPARQL. Cela nécessitera un raisonnement sur le traitement des homomorphismes pour conserver le moins de données possible qui préserve les réponses de la requête.

Une meilleure étude des applications qui peuvent être modélisées comme le problème FLGM nous donnera des indices sur ce qui doit être amélioré dans nos solutions et nous fournira des commentaires sur la conception de benchmarks réalistes. Les techniques de complexité théorique plus élevée pourraient mieux fonctionner pour les structures de graphes spécifiques à l'application que celles avec une complexité théorique inférieure à usage général. Une autre question concerne le modèle graphique que nous utilisons. Les graphes non orientés, par exemple, peuvent mieux s'adapter dans certaines situations.

6 Conclusion Générale

Dans cette thèse, nous traitons trois problèmes liés aux graphes et aux langages hors-contexte : l'expression de langages hors-contexte à l'aide de notations alternatives, l'évaluation de requêtes de chemins hors-contexte et la minimisation de graphes contraints aux langages hors-contexte.

Les expressions SM ont encore besoin d'être formalisées sur certaines propriétés et d'évaluer leur utilisabilité. Analyser la complexité de la traduction des expressions SM en grammaires hors-contexte, identifier la classe des langages hors-contexte qui ne peuvent pas être exprimés avec des expressions SM et les comparer avec d'autres propositions d'expressions non régulières. Il est également important de mener des expériences avec des étudiants et des professionnels de l'informatique et des domaines connexes pour évaluer l'utilisabilité de telles expressions.

Les performances de notre algorithme basé sur les items de trace (Algorithm 1) peuvent être améliorées pour traiter des graphes plus grands. Le parallélisme, de meilleures techniques d'indexation et une meilleure gestion des données pour utiliser la mémoire disque afin de réduire la consommation de la mémoire principale sont des améliorations souhaitées. Ceux-ci peuvent améliorer les performances et l'évolutivité de notre prototype à bien des égards : le traitement des sommets non marqués dans les ensembles de positions peut être effectué en parallèle ; conserver le plus de données possible sur le disque lui permettra de traiter de graphes plus grands qui ne tiennent pas dans la mémoire principale.

Le travail sur la minimisation des graphes est plein de possibilités à poursuivre. L'amélioration des algorithmes et de leurs prototypes correspondants leur permettra de traiter des graphes plus grands consommant moins de temps et de mémoire. Il est important que les algorithmes traitent des requêtes conjonctives avec des variables pour qu'ils puissent être utilisés dans une gamme plus grande d'applications.

Ces orientations ouvrent le champ à davantage de branches de recherche et à de nouveaux langages d'interrogation et applications utilisant ces nouvelles technologies développées pour émerger.

Ciro MORAIS MEDEIROS

Améliorations de Requêtes de Chemin Dans les Graphes : Expression, Évaluation et Satisfiabilité de Coût Minimal

Résumé :

Nous traitons trois problèmes liés aux requêtes de chemin en graphes. La plupart des langages de requête en graphes actuels prennent en charge les requêtes de chemins réguliers. Cependant, certaines applications telles que l'analyse du code source et la génétique nécessitent des requêtes de chemins hors-contexte. Les requêtes de chemins hors-contexte utilisent des langages hors-contexte. Il n'y a pas de notation standard pour les langages hors-contexte plus simple que les grammaires hors-contexte. L'évaluation d'une requête de chemins hors-contexte est plus complexe qu'une requête de chemins réguliers. Encore, dans certaines applications, on souhaite avoir le graphe minimum qui préserve les réponses à une requête de chemins donnée.

Pour résoudre chacun de ces problèmes, nous : (1) développons une notation alternative pour exprimer des langages hors-contexte ; (2) développons et expérimentons un algorithme d'évaluation de requête de chemins hors-contexte ; et (3) formalisons le problème de minimisation de graphes contrainte par un langage formel, pour lequel nous développons des solutions pour les cas où le langage formel est régulier ou hors-contexte.

Mots-clés: langages hors-contexte, requêtes de chemins dans les graphes, minimization de graphes.

Improvements on Graph Path Queries: Expression, Evaluation, and Minimum-Weight Satisfiability

Abstract :

We deal with three problems related to graph path queries. Most current graph query languages support regular path queries. However, some applications such as source-code analysis and genetics require context-free path queries. Context-free path queries use context-free languages. There is no standard notation for context-free languages simpler than context-free grammars. The evaluation of a context-free path query is more complex than a regular path query. Moreover, in some applications, it is desired to have the minimum graph that preserves answers to a given path query.

To address each of those problems, we: (1) develop an alternative notation for expressing context-free languages; (2) design, implement and experiment with a context-free path query evaluation algorithm; and (3) formalize the formal-language-constrained graph minimization problem, for which we design solutions for the cases where the formal language is regular or context-free.

Keywords: context-free languages, graph path queries, graph minimization.

LIFO - Laboratoire d'Informatique Fondamentale d'Orléans

Bâtiment 3IA, rue Léonard de Vinci, B.P. 6759 45067

ORLEANS cedex 2, FRANCE

DIMAp - Departamento de Informática e Matemática Aplicada

Campus Universitário, Lagoa Nova, 59.078-970, Natal, RN,

BRASIL