UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A GENERAL APPROACH TO RECURSION IN G-CORE

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

VALENTINA BELÉN URZÚA PUENTES

PROFESOR GUÍA:
CLAUDIO GUTIERREZ GALLARDO

MIEMBROS DE LA COMISIÓN:
JOSE A. PINO URTUBIA
ÉRIC TANTER
DIEGO SECO NAVEIRAS

SANTIAGO DE CHILE
2021

Resumen

G-CORE es un lenguaje de consulta de bases de datos de grafos. Una de sus características más novedosas es que es un lenguaje cerrado, pues se consultan grafos y se retorna grafos. Además, tiene la capacidad de expresar caminos como ciudadanos de primera clase, es decir, se puede manipular, operar y retornar caminos.

Como muchos otros lenguajes de consulta, G-CORE no tiene recursión. Aunque las funcionalidades de caminos ayudan, todavía no son suficientes para expresar una amplia variedad de consultas, particularmente algoritmos de grafos clásicos como orden topológico, búsqueda en anchura (BFS), circuito Euleriano, árbol recubridor mínimo, etc.

Para abordar este problema en esta tesis se extiende G-CORE con funcionalidades recursivas. Esto se ha hecho en otros lenguajes de consulta, por ejemplo en SQL y SPARQL, donde se han obtenido buenos resultados al aumentar su poder expresivo de forma amigable.

Con ese objetivo, comenzamos por introducir una sintaxis y semántica para el nuevo operador recursivo. Dado que el modelo de datos de G-CORE es el de Path Property Graph, no es posible traducir directamente la recursión de SPARQL o de SQL. Lo anterior se debe a que la recursion de SQL y SPARQL está basada en la semantica de punto fijo, usando para ello la convergencia de una tabla auxiliar temporal que va creciendo hasta estabilizarse.

En el caso de The Path Property Graph, el avance del proceso recursivo no siempre se corresponde con un crecimiento de algún grafo. Luego no se puede saber de manera precisa cual métrica seguir para evaluar la convergencia del grafo temporal a un punto fijo. Para abordar esto, proponemos una recursión general a través de una estructura externa que pudiera definir distintas métricas para medir el crecimiento y convergencia del grafo temporal.

También mostramos como pueden ser codificadas las consultas clásicas de grafos añadiendo este nuevo operador recursivo. Finalmente repasamos las dificultades de implementar este nuevo operador en G-CORE y que otros operadores necesitan ser añadidos al lenguaje.

Abstract

G-CORE is a graph database query language. One of its most novel characteristics is that it is a closed language, since graphs are consulted and graphs are returned. In addition, it has the ability to express paths as first-class citizens, that is, it can manipulate, operate and return them.

Like many other query languages, G-CORE does not have recursion. Although the path functionalities help, they are not yet sufficient to express a wide variety of queries, particularly classical graph algorithms such as topological order, Breadth First Search (BFS), Eulerian circuit, minimal spanning tree, etc.

To address this problem in this thesis G-CORE is extended with recursive functionalities. This has been done in other query languages, for example in SQL and SPARQL, where good results have been obtained by increasing its expressive power in a friendly way.

With that goal, we start by introducing a syntax and semantics for the new recursive operator. Since the G-CORE data model is that of the Path Property Graph, it is not possible to directly translate the recursion from SPARQL or SQL. This is due to the fact that the recursion of SQL and SPARQL is based on fixed point semantics, meaning it uses the convergence of a temporary auxiliary table that grows until it stabilizes.

In the case of The Path Property Graph, the progress of the recursive process does not always correspond to a growth of some graph. Therefore, it is not possible to know precisely which metric to follow to evaluate the convergence of the temporal graph to a fixed point. To address this, we propose a general recursion through an external structure that could define different metrics to measure the growth and convergence of the temporal graph.

We also show how classic graph queries can be coded by adding this new recursive operator. Finally we review the difficulties of implementing this new operator in G-CORE and that other operators need to be added to the language.

A mi familia, mi Mamá y mi Papá que me inculcaron ir el amor por la ciencia.

Agradecimientos

Tengo que agradecer a mucha gente el poder haber realizado esta tesís. Fue un trabajo largo y en ocaciones tedioso pero que al final logró un buen resultado.

Primero quiero agradecer a mi familia, a mi Papá y Mamá que siempre me apoyaron en todo para que yo pudiese estudiar tranquila y que siempre me incentivaron a tomar desafíos. A mis abuelitos, la abuelita pilar y el chico Mario que siempre estuvieron ahí preocupandose por mí en este proceso. A mi gatita Chimu por todo el amor y el ronroneo y también a lupita y manchita por acompañarme siempre.

Luego agradecer al profesor Claudio Gutierrez por su paciencia y por siempre ayudarme para que yo pudiese terminar este trabajo, no me es posible retribuirle todo lo que ha hecho por mi pero espero que con este trabajo podamos sacar una publicación. Al profesor Aidan y a Adrian que me guiaron en mi tesís aportando ideas y revisando mi trabajo, gracias por ayudarme!

A mis amigos de la carrera de Licenciatura en Matemáticas, pues sin ellos no hubiese llegado hasta aquí, en especial a Luis y a Miguel que llegaron a ser tan importante como mi familia y que siempre estarán ahí no importa que pase.

A mis amigos del trabajo, a Benja, Franquito, Ignacio, Roberto. Por los buenos momentos y porque a pesar de que eramos compañeros de trabajo, forjamos una amistad profunda y bonita, espero seguir viendolos y riendonos!

A mis amigos que conocí en Beauchef: Alvaro, Jorge Ampuero, Lucas, Eduardo, Nelson, Sofía (mi primera amiga de la U) por siempre apoyarme y ayudarme en lo que necesitaba, son amistades muy valiosas y que lucharé por conservar. A Florencia y a Constanza, por su amistad única y por siempre creer en mi e incentivarme a hacer nuevos proyectos, muchas gracias amigas de mi alma.

A mis amigas de la vida, Karin y Vicky que siempre me apoyaron y me ayudaron en llevar bien este proceso. Gracias por la paciencia y el amor que esta presente hace muchos años. Espero seguir viviendo cosas maravillosas con ustedes.

Contents

Introduction **1**
Outline and Main Contributions 4

1 An introduction to G-CORE **5**
1.1 G-CORE Overview 5
1.2 Data Model 7
1.2.1 Path Property graphs 7
1.3 Basic notions of G-CORE 8

2 Related Work **13**
2.1 Recursion in Relational Database Query Languages 13
2.1.1 Datalog 13
2.1.2 Recursion in SQL 15
2.2 Recursion in Graph Query Languages 15
2.2.1 SPARQL 16
2.2.2 Cypher 16
2.2.3 PGQL 17
2.2.4 G-CORE 17

3 General Recursion **18**
3.1 The need for a general recursive operator 19
3.2 A proposal to generalize the expression of recursive queries 22
3.3 Syntax of general recursive queries in G-CORE 25
3.3.1 Nested Recursion 26
3.4 Semantics of general recursive queries in G-CORE 27
3.5 An Example: Nodes at k distance from the root 29
3.6 Data Complexity 31
3.6.1 Note: On Linear Recursion. 31

4 Graphs algorithms with recursive G-CORE **33**
4.1 Topological Sorting 33
4.2 Minimum Spanning Tree 37
4.3 Eulerian Circuit 40
4.4 Maximum Matching 48
4.5 Connected Components 53
4.6 Planarity Testing 55

5 Issues for the Implementation of recursion in G-CORE **57**
5.1 Current Implementation of G-CORE 57
5.2 Adding recursion to G-CORE 58
5.3 Problems with G-CORE operators 58

6 Conclusion and Future Work **60**

Bibliography **62**

List of Tables

3.1 Parent Table 19
3.2 Base Case Nodes at k distance 30
3.3 Recursive step 1 Nodes at k distance 30
3.4 Recursive step 2 Nodes at k distance 30

4.1 Base case topological sorting 35
4.2 Recursive step 1 topological sorting 35
4.3 Recursive step 2 topological sorting 36
4.4 Recursive step 3 topological sorting 36
4.5 Recursive step 4 topological sorting 37
4.6 Base step Minimum Spanning Tree 39
4.7 Recursive step 1 Minimum Spanning Tree 39
4.8 Recursive step 2 Minimum Spanning Tree 40
4.9 Base case Eulerian Circuit 42
4.10 Recursive step 1 Eulerian Circuit 43
4.11 Recursive step 2 Eulerian Circuit 43
4.12 Recursive step 3 Eulerian Circuit 44
4.13 Recursive step 4 Eulerian Circuit 44
4.14 Recursive step 6 Eulerian Circuit 45
4.15 Recursive step 7 Eulerian Circuit 46
4.16 Recursive step 8 Eulerian Circuit 46
4.17 Recursive step 9 Eulerian Circuit 47
4.18 Final graph Eulerian Circuit 47
4.19 Table representing state of edges in a labeled graph 48
4.20 Base case Maximum Matching 50
4.21 Recursive step 1 Maximum Matching 51
4.22 Recursive step 3 Maximum Matching 51
4.23 Recursive step 3 Maximum Matching 52
4.24 Table that represents the nodes of a graph 54
4.25 Final Result that represents de nodes that are connected 54

List of Figures

1.1 A PPG that represents a small social network. 8

3.1 Successive partial results given by the query above over the Parent table . . 20
3.2 Mapping from a graph G to a partially ordered structure 24
3.3 Schema of the different steps of the mappings from the data (graphs) to a partially ordered structure while the recursion is taking place. 24
3.4 The recursive process of evaluation of a query. Each q_r gives the next t_j based on the previous one and on G $(q_r(G, t_{j-1}))$. The map H gives in each step the corresponding element of the order. 27
3.5 A hierarchical company 29

4.1 A Directed Graph 34
4.2 G^1 topological Sorting 35
4.3 G^2 topological Sorting 35
4.4 G^3 topological Sorting 36
4.5 G^4 topological Sorting 36
4.6 G^5 topological Sorting 37
4.7 A weighted general graph 38
4.8 G^1 Minimum Spanning Tree 39
4.9 G^2 Minimum Spanning Tree 39
4.10 G^3 Minimum Spanning Tree 40
4.11 A generic Graph 41
4.12 G^1 Eulerian Circuit 42
4.13 G^2 Eulerian Circuit 43
4.14 G^3 Eulerian Circuit 43
4.15 G^4 Eulerian Circuit 44
4.16 G^5 Eulerian Circuit 44
4.17 G^6 Eulerian Circuit 45
4.18 G^7 Eulerian Circuit 45
4.19 G^8 Eulerian Circuit 46
4.20 G^9 Eulerian Circuit 46
4.21 G^{10} Eulerian Circuit 47
4.22 Final Graph Eulerian Circuit 47
4.23 G^1 Maximum Matching 50
4.24 G^2 Maximum Matching 51
4.25 G^3 Maximum Matching 51

4.26 G^4 Maximum Matching 52
4.27 A PPG that represents a social network. 53

5.1 Difference of edges 59

Introduction

Graphs are one of the most popular representations of data today. Its flexibility and simplicity have made them a favorite model in many fields.

For example in electrical engineering, graphs are used in probabilistic graphical models to explicitly show the probabilistic relationships between variables of a model. In physics and chemistry they are used to study molecules and in linguistics, they are used for parsing of a language tree[1].

But probably where this model is most popular is in representing data and phenomena from social networks. Also, in computer science, graphs have been used since long ago for solving many problems, such as finding the shortest path between entities, or to understand the importance of an entity which is part of a network.

There are many ways to represent a graph. For instance, representations with adjacency lists, with adjacency matrices, or with other specific data structures. In graph databases, currently the most common model for representing graphs is the property graph model [1]. Property graphs have nodes representing entities, edges representing relationships between nodes, and properties that add information about specific characteristic of those entities and relationships.

In the last decade many query languages and database systems have emerged for working with graph-like data. The use-cases for graph databases and the number of sources available have increased dramatically, and each day it is more natural to see this model as a natural complement to the relational setting. There are many graph query languages today. Among the most popular are SPARQL, Cypher and Gremlin [2].

In this work we present an extension for graph query languages that allows us to compute recursive queries. These kinds of queries were introduced in the last main revision of SQL, where the "WITH RECURSIVE" clause was introduced, mainly to have a way to deal with queries that code reachability between entities. Recursion has been widely studied in the case of relational databases [3] but not so much in graph databases. For graph query languages, forms of reachability and recursion were first introduced at a theoretical level in Datalog [3]. As for the first approach with the current wave of graph query languages, it was given by Reutter et al [4] with SPARQL. We follow closely their approach. Our extension is declarative, therefore, it is known what to extract (consult) but not how to do it. This model

[1] `http://www.its.caltech.edu/~matilde/ParsingTrees.pdf`

is found in the syntax of various graph query languages, for example: SPARQL [5], Cypher [6] and G-CORE.

Our research focuses in G-CORE (2018), which is a query language proposal designed by a working group belonging to the *Linked Data Benchmark Council* [7]. G-CORE has two main features:

1. The language is closed, that is, the input of a query is a graph and the output is also a graph.

2. Paths are treated as first class citizens [8]. This means that paths can be manipulated and expressed in a simple way.

Evolution of this thesis. The original objective of this thesis was to study the functionality (2) that is novel in query languages. We began by studying how relevant polynomial graph problems could be expressed with this new functionality. We believed that studying whether or not classical graph problems could be solved in a graph database language can give us clear ideas about its query power and help us to optimally design these types of languages.

These relevant polynomial problems in graph theory were chosen from the book [9]. Soon we realized that most of the algorithms that solved these problems used some type of structure that was storing previous results, that is, some sort of recursion that was difficult or impossible to express with the current functionalities of G-CORE. Clearly a recursion functionality was needed to express (to code) in a clean and simply manner these algorithms in G-CORE. That is, we found recursion to be a natural addition to this graph database language and a needed preliminary step to address the original goal of the thesis.

Given this, we shifted the objective of the thesis towards studying and adapting recursion from relational databases to graph databases, a problem that was interesting per se and posed several challenges. We needed to adapt the recursion. SPARQL had a different approach as it uses tables as output, and thus the recursion in SQL in this case could be adapted naturally. G-CORE on the contrary, uses property graphs as input and output. Then, to formally define a recursive operator in G-CORE, we study how it could be extended in the Property Graph database model. We show that the conversion of this recursive operator is not direct, since the fixed point is not reached in the same way as in the case of having tables that converge to the same result.

After formally defining this operator, we studied how the queries of graph problems presented at the beginning could be expressed. For this, we express the query using the best known algorithms for each problem and adapt them to the recursive operator using a base case and a recursive case and in each problem, explaining step-by-step how the response was constructed.

We also identify in each problem and for the recursive operator new operators and functionalities that needed to be made explicit in the language. Definitively G-CORE needs to be enriched in order to express basic problems in graph theory. We also found that more fine-grained operators need to be defined at the level of operators between graphs. Besides

these additions, we show that G-CORE is a sufficiently flexible language to permit incorporating this recursive operator in a simple way. Also we show the importance of implementing recursion since it allows for expressing classical graph algorithms in a declarative way.

Outline and Main Contributions

Chapter 1: We introduce a general G-CORE review, presenting its functionality and main clauses.

Chapter 2: We introduce the state of art of recursion in relational query languages and graph query languages.

Chapter 3: We give a formal definition of a recursive operator in G-CORE making syntax and semantics explicit.

Chapter 4: We show how polynomial graph algorithm problems can be coded with this new recursive operator.

Chapter 5: We show a small discussion of issues and considerations implementing the recursive operator in G-CORE.

Chapter 6: We present our conclusions and future investigations that could be explored based on the results of this work.

Publications

Preliminary parts of this work were published in 13th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2019) `http://ceur-ws.org/Vol-2369/short07.pdf`.

Chapter 1

An introduction to G-CORE

In this chapter we give a formal introduction to our object of study, the graph query language G-CORE. We first present in general terms what G-CORE is, then we formally define the notion of Path Property Graph which is the data model of G-CORE. Finally we review the syntax and semantics of the MATCH and WHERE clauses which are the base clauses of G-CORE. The official definition of G-CORE is presented in [8].

1.1 G-CORE Overview

G-CORE is a graph database query language designed by a working group belonging to *The Linked Data Benchmark Council* (**LDBC**) [7]. LDBC was founded in 2012 by members of academia and industry, and its goal is to set standards for new graph database management systems. G-CORE was proposed to address the need of a standard property graph query language (**The GQL manifesto** [10]). For the authors the main challenges were to have a composable query language with graph construction, to support path functionalities for navigational queries and to capture the core of available languages [8].

Like many graph database query languages, G-CORE is essentially a matching pattern query language, that is, its core functionality is to match a pattern against a graph, describing visual patterns using ASCII-Art syntax like Cypher [6]. Its main characteristics are that, first, it is a closed language, that is, the resulting query always returns a graph with a CONSTRUCT CLAUSE; and second, that it incorporates paths as first-class citizens [8].

Example 1.1.1 Example of a G-CORE query A classic graph database language query is to get all the friends of a person named "John". This query can be written in G-CORE as:

```
1   CONSTRUCT (n)
2     MATCH   (n:Person)-[:friendOf]-(m:Person) ON G
3     WHERE   m.name="John"
```

This query gives as results all the nodes in a graph G that are adjacent by a relation represented by the edge whose label is friendOf with a node m whose name is John.

Path queries are fundamental in graph database query languages since they can express navigational queries. As stated above, paths in G-CORE are first class citizens, which means that these paths can be returned, compared, operated and also stored like any other datatype. These functionalities are mainly based on the regular path expressions borrowing the syntax from PGQL [11].

Example 1.1.2 Example of a simple path query in G-CORE. We can extend the example 1.1.1 and make a reachability query, that is to get all persons that are connected with John via a path of arbitrary length of friendOf edges:

```
4   CONSTRUCT (n)
5     MATCH   (n:Person)-<:friendOf*>-(m:Person) ON G
6     WHERE   m.name="John"
```

In this case, G-CORE finds all the nodes that are connected through a path that satisfies a regular expression (in this case the Kleene star), therefore, it returns a set of isolated nodes. To compute this query G-CORE uses the semantics of the shortest path [12], which we will present in the next sections.

As in SPARQL [5], can use full regular expressions to express navigational and path queries. In addition, G-CORE allows a PATH clause to express complex path expressions, namely, regular queries, studied in [13].

Example 1.1.3 Example of stored paths query in G-CORE

```
7   CONSTRUCT (n), (n)-/@p:JohnContacts/->(m)
8     MATCH   (n:Person)-/<:friendOf*>/-(m:Person) ON G
9     WHERE   m.firstName="John"
```

In the query above we see an "@" prefix that indicates a stored path. A stored path is a path in the graph that is stored as data in the database model, therefore it can be used in a query. In this case the query is returning nodes and stored paths that are stored with the label "JohnContacts".

Example 1.1.4 Path clause in G-CORE

```
10   PATH newFriend = (x)-[e:knows]-(y)
11    WHERE e.date > '2010-01-01'
12   CONSTRUCT (m)
13    MATCH    (m)-/p<~newFriend*>/-(n)
14    WHERE    n.firstName = "John"
```

The above query uses the PATH clause in line 10, that defines an edge pattern with conditions. Thus we find all the people that can be found from John by following "knows" edges created after 2010. This example is based on the one given in [11].

1.2 Data Model

The Property Graph (PG)[1] is a data model in which vertices and edges in a graph can be associated with arbitrary properties as key-value pairs. For example, one can have an x node labeled as "University" and a property name whose value is "University of Chile". This is the data model of several graph query languages [12], and there are imperative languages like Green-Marl [14] designed to express graph algorithms in this model.

1.2.1 Path Property graphs

The data model used by G-CORE is an extension of the model described above and is called a `Path Property graph`. This model has the characteristic that each node or edge has property-value pairs and in addition, the paths in the graph can be explicitly handled. Here is a formal definition of the model *Path Property Graph* described in [8].

Definition 1.2.1. (Path Property Graph) Assume that:

- L : Infinite set of labels for nodes, edges and paths.
- K : Infinite set of property names.
- V : Infinite set of literals as variables that can be true or false, strings, etc.
- $FSET(X)$: All finite subsets of a set X including $\emptyset$.
- $FLIST(X)$: Set of all finite items lists of a set X (including the empty list).

Then a PPG (Path Property Graph) is a graph of the form $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ where N is the set of nodes of G , E is the set of edges and P is a set of paths identifiers of G (not all paths in G). On the other hand, ρ, δ, λ and σ are total functions, that is, they are defined over all the domain, such that:

1. $\rho : E \to (N \times N)$, where to each edge $e \in E$ assigned a pair of nodes $(a, b) \in N \times N$.
2. $\delta : P \to FLIST(N \cup E)$, where each path $p \in P$ it is assigned a list that represents the sequence of nodes and edges that conforms it.
3. $\lambda : (N \cup E \cup P) \to FSET(L)$, where each node, edge and path is assigned a set of labels.
4. $\sigma : (N \cup E \cup P) \times K \to FSET(V)$ where to each pair $< x, k > \in (N \cup E \cup P) \times K$ the set of values of the property k is assigned for the node, edge or path x.

Example 1.2.1 Consider the following PPG example from the article [8], representing a small social network:

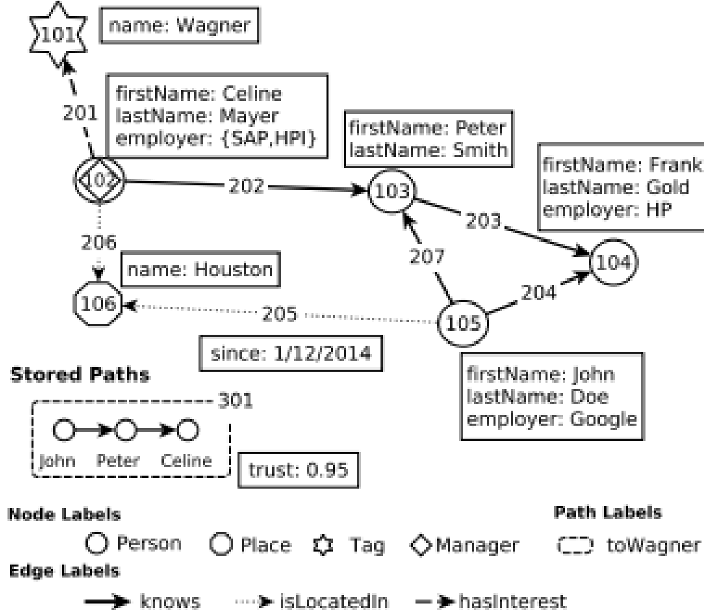


Figure 1.1: A PPG that represents a small social network.

We have:

$$N = \{101, 102, 103, 104, 105, 106\},$$
$$E = \{201, 202, 203, 204, 205, 206, 207\},$$
$$P = \{301\},$$

then, the identifiers of nodes, edges and paths are:

$$\rho = \{201 \to (102, 101), ..., 207 \to (105, 103)\},$$
$$\delta = \{301 \to [105, 207, 103, 202, 102]\},$$

and we have that the assignments of edges and paths are respectively:

$$\lambda = \{101 \to \{\text{Tag}\}, 102 \to \{\text{Person,Manager}\}, ..., 201 \to \{\text{hasInterest}\}, ..., 301 \to \{\text{toWagner}\}\},$$

$$\sigma = \{(101, \text{name}) \to \{\text{Wagner}\}, ..., (205, \text{since}) \to \{1/12/2014\}, ..., (102, \text{employer}) \to \{SAP, HPI\}\}.$$

1.3 Basic notions of G-CORE

In this section we will review the semantics of queries in G-CORE, concentrating mainly on the clause `MATCH` and `WHERE`.

Paths conforming to regular expressions: Unlike Cypher [6], which defines the semantics of the expressions of its paths based on simple paths, G-CORE is based on arbitrary paths.

These arbitrary paths are expressed as strings that satisfy a regular expression, where G-CORE computes **the shortest path** according to the lexicographical order [15] in the nodes, because this can be done in polynomial time [8]. Let $\mathcal{N}$, $\mathcal{E}$, $\mathcal{P}$ denote infinite sets of node identifiers, edges, and path variables respectively.

Definition 1.3.1. A regular expression r, used in G-CORE, is defined as follows:

$$r := \quad _ \quad | \quad l \quad | \quad l^{-} \quad | \quad !l \quad | \quad (r+r) \quad | \quad (rr) \quad | \quad (r)^{*},$$

where l is in an alphabet L. The expressions l and l^{-} represent a label of an edge, while $!l$ represents the label of a node. On the other hand, the expression _ represents any edge tag, $l \in L$, and therefore $(-)^{-}$ represents any edge tag of the form l^{-}.

Definition 1.3.2. Let $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ be a PPG; we say that a path $p \in P$ satisfies a regular expression r if there is a string that belongs to the language generated by r, such that there is a correspondence between the node and edge labels along the way with the string.

Also, if the path length of p, $length(p) = n$, then p is the shortest path from a node a_1 to a_{n+1} that satisfies r, if for every other path p' from a_1 to a_{n+1} that satisfies r, it holds $length(p) \leq length(p')$.

Definition 1.3.3 (substitutions)**.** Let $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ be a PPG. Then a mapping μ over G is defined as a partial function $\mu : (\mathcal{N} \cup \mathcal{E} \cup \mathcal{P}) \rightarrow (N \cup E \cup P)$, such that $\mu(x) \in N$, if $x \in \mathcal{N}$, $\mu(y) \in E$, if $y \in \mathcal{E}$ and $\mu(z) \in P$, if $z \in \mathcal{P}$.

Furthermore, it is assumed that $dom(\mu)$ is a finite set that corresponds to the domain of the function μ. The previous definition states that the function μ assigns each node, arc or path its respective identifier in the graph in which we want to make the queries.

MATCH clause

First we will start presenting the G-CORE `MATCH` clause. It is based in the notion of pattern, whose basic component is the basic graph pattern.

Definition 1.3.4. (Basic graph pattern) A basic pattern is specified in the following grammar:

$$\begin{aligned} basicGraphPattern &= nodePattern \quad | \quad edgePattern \quad | \quad pathPattern \\ nodePattern &= (x), \\ edgePattern &= x \xrightarrow{z} y \\ pathPattern &= x \xRightarrow{\text{@w in r}} y \quad | \quad x \xRightarrow{\text{w in r}} y \end{aligned}$$

where $x, y \in \mathcal{N}$, $w \in \mathcal{P}$, r is a regular expression and $z \in \mathcal{E}$.

Definition 1.3.5 (Evaluation of basic graph pattern)**.** Let α be a basic graph pattern. The evaluation of α on a PPG, G denoted by $\|\alpha\|_G$ is inductively defined as follows:

1. If α is a node (x) then $\|\alpha\|_G = \{\mu | dom(\mu) = \{x\}$ and $\mu(x) \in N\}$

2. If α is an edge $x \xrightarrow{z} y$, then $\|\alpha\|_G = \{\mu | dom(\mu) = \{x, y, z\}$ and $\mu(x) \in N, \mu(y) \in N, \mu(z) \in E$ and $\rho(\mu(z)) = (\mu(x), \mu(y))\}$

3. If α is a path $x \xRightarrow{\text{w in r}} y$, then $\|\alpha\|_G = \{\mu | dom(\mu) = \{x, y, w\}$,$\mu(x) \in N, \mu(y) \in N$ and $\mu(w)$ is a new path, which does not belong to P and is associated with the shortest path L that goes from $\mu(x)$ to $\mu(y)$ over G that conforms to r.

Example 1.3.1 Considering Figure 1, for Definition 1.3.4, we have the following cases:

1. $\alpha = (x)$, that is , if α is a node we have:

$$\|(x)\|_G = \{\mu_1, \mu_2, \mu_3, ..., \mu_6\}$$

where

$$\begin{aligned} \mu_1 &= \{x \to 101\} \\ \mu_2 &= \{x \to 102\} \\ \mu_3 &= \{x \to 103\} \\ \mu_4 &= \{x \to 104\} \\ \mu_5 &= \{x \to 105\} \\ \mu_6 &= \{x \to 106\} \end{aligned}$$

2. $\alpha = x \xrightarrow{z} y$, that is, if α is an edge we have:

$$\|x \xrightarrow{z} y\|_G = \{\mu_1, \mu_2, \mu_3, \mu_4, ..., \mu_6\}$$

where

$$\begin{aligned} \mu_1 &= \{x \to 102, z \to 201, y \to 101\} \\ \mu_2 &= \{x \to 102, z \to 202, y \to 103\} \\ \mu_3 &= \{x \to 103, z \to 203, y \to 104\} \\ \mu_4 &= \{x \to 105, z \to 204, y \to 104\} \\ \mu_5 &= \{x \to 105, z \to 207, y \to 103\} \\ \mu_6 &= \{x \to 105, z \to 205, y \to 106\} \\ \mu_7 &= \{x \to 102, z \to 206, y \to 106\} \end{aligned}$$

3. $\alpha = x \xRightarrow{\text{w in r}} y$, that is, it is a path that satisfies a regular expression, for example $r = knows^*$, we have:

$$x \xRightarrow{\text{w in r}} y = \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5\} \tag{1.1}$$

where:

$$\mu_1 = \{x \to 102, y \to 101, w \to knows\}$$
$$\mu_2 = \{x \to 102, y \to 103, w \to knows\}$$
$$\mu_3 = \{x \to 105, y \to 103, w \to knows\}$$
$$\mu_4 = \{x \to 105, y \to 104, w \to knows\}$$
$$\mu_5 = \{x \to 103, y \to 104, w \to knows\}$$

Note that the path from node (102) to node (104) is not delivered, since it has length 2 and is not the minimum. In addition w could be a more **complex regular expression**. For instance, $(knows + knows^{-})^{*}$ is a two-way regular path.

Definition 1.3.6. (Graph patterns in different graphs) In G-CORE the basic patterns can be evaluated in different graphs. The operator ON is used to specify in which graph to evaluate them and it is explained as follows:

```
basicGraphPatternLocation ::= basicGraphPattern |
                              basicGraphPattern ON location

location::= gid |fullGraphQuery
```

Where gid is an identifier of a graph and $fullGraphQuery$ is a combination of basic queries to a graph under the union, intersection and subtraction operations.
A complete pattern is defined as a sequence of basic patterns separated by commas:

```
fullGraphPattern ::= basicGraphPatternLocation |
                      basicGraphPatternLocation,
                     fullGraphPattern
```

WHERE Clause

Another fundamental structural feature of G-CORE is the `WHERE` clause that allows for filtering results from Boolean conditions.

Definition 1.3.7. The clause syntax `MATCH` with conditions is:

```
matchClause ::= MATCH fullGraphPattern |
                MATCH fullgraphPattern  WHERE BooleanCondition
```

where BooleanCondition is an **expression** which is evaluated as true or false.

Definition 1.3.8. An **expression** that we will denote by ξ is formally defined, which corresponds to a Boolean-valued expression, that is, an expression that evaluates true or False as follows:

$$\xi := x \quad | \quad x.k \quad | \quad x : l \quad | \quad \diamond\xi \quad | \quad \xi \odot \xi \quad | \quad f(\xi, \xi, ...) \quad | \quad \sum(\xi) \quad | \quad \texttt{EXISTS q}$$

where:

- x is a variable.
- $k \in K$ a property name.
- $l \in L$ a label.
- $\diamond$ is an operator corresponding to Boolean negation `NOT`, arithmetic negation -1, etc.
- $\odot$ corresponds to some binary operator for example: Arithmetic operations on numbers and strings: $+$, -, *, $/$, etc. Boolean operators like `AND` and `OR` or comparisons like: $\neq$, $>$, texttt IN, etc.
- f defined as standard functions inherited from relational query languages, `Count`, `Min`, `Max`, `Sum`, `Avg`, etc.
- $\sum$ is an aggregation function.
- `EXISTS q`, where `q` is a subquery such that true is evaluated if the evaluation of `q` is a non-empty *Path Property graph* and is false otherwise.

Subqueries. In G-CORE there are existential subqueries, which provides some type of negation of graph patterns and enable more powerful pattern filtering.

Suppose we want to know all the people who are friends of John's friends who do not work in a telephone company for example "Claro". This can be done with the following query

Example 1.3.2 Example of a subquery in G-CORE

```
15 CONSTRUCT (n)
16   MATCH   (n:Person)-<:friendOf*>-(m:Person) ON G
17   WHERE   m.name="John"
18   AND
19   WHERE NOT EXISTS (
20               CONSTRUCT ()
21               MATCH (n)-[:isEmployeeOf ]->(c:Company)
22               WHERE c.name = "Claro")
```

Now in order to extend G-CORE with recursive functionalities we need to study how it has been done before in other query languages. In the next chapter we will review the state of the art regarding adding recursion to database query languages putting special emphasis on Datalog, SQL and SPARQL.

Chapter 2

Related Work

This chapter presents a review of the state-of-the-art research on recursion in query languages for relational databases as well as for graph databases.

Recursion in database query languages has been extensively studied both from a theoretical [16] as well as from a practical point of view (e.g. SQL, SPARQL [12, 4]). For our purposes in this thesis, the classical case of SQL and the the graph-like character of SPARQL are the most relevant and we will concentrate on them.

2.1 Recursion in Relational Database Query Languages

The paradigms of relational database query languages are relational algebra and relational calculus, which provide the language with algebraic and logical operations, based on algebra and first order calculus respectively [16]. Although these languages contain what are considered fundamental operations for database query languages, there are simple but useful operations that cannot be performed in this framework, for example, transitive closure. In order to study these types of operators, the formalism of Datalog was introduced. Datalog is a standard and flexible formalism where to study the more general notion of recursion that is what interest us here.

2.1.1 Datalog

Datalog was introduced by Chandra and Harel in [17] and it can be viewed as logic programming without function symbols. A Datalog program consists of a collection of rules, each of which is a conjunctive query [16]. Recursion is introduced by allowing the same predicate symbol in both the head and the body of a rule or by allowing "cycles" in the order of calls of a predicate. The semantics for these recursive Datalog programs is based on the theory of fixpoints. Recursion in general needs a notion of order. In Datalog it uses the order (by set inclusion) conformed by the set of facts that the program derives from the original dataset. One can define thus an operator usually called the immediate consequence operator, that produces new facts starting from known facts. The operator uses the semantic of the smallest fixed point which is the smallest solution of a fixpoint equation involving that operator.

This solution can be obtained constructively. In this manner, standard relational algebra can be extended with a while operator that allows us to iterate an algebraic expression while some condition holds.

Datalog with negation The standard semantics for recursive Datalog uses the semantic of the least fixed point, thus it is only defined for monotonous queries. Therefore, it cannot express queries such as the complement of the transitive closure. Combining negation with recursion in queries seems natural, but the fact that the resulting query could be not positive can present problems. A natural idea would be to extend the fixed-point semantics. However, with non-monotonous queries the consequence operator may not converge to any fixed point or there may be more than one fixed point.

In order to define a consistent semantics for Datalog with recursion and negation, different semantics have been proposed. The most relevant are the inflationary semantics, the stratified semantics and the well founded semantics. These three semantics are the most relevant to understanding the complexity of adding negation to recursion. Since algorithms that use negation are very important in practice (see the examples of graph algorithms in the following chapters), it is necessary to relax the restrictions that avoid negation (as in SQL and SPARQL) and define a more approachable semantics. We will return to this topic in the recursion chapter.

The inflationary semantics [16] has a fixed point approach and it uses iteration based on an "inflation of tuples". In other words, the fixed point operator is cumulative, which means that the previous results are added to the result of the current evaluation and it stops when no news elements are added (the inflationary fixed point).

Stratified Negation has a different approach based on allowing negation only in the strata where there is no recursion. Essentially it is an extension of semi-positive Datalog [16], where negation occurs in predicates in rule bodies and not in IDB (Intensional Database) predicates. The main idea is that a Datalog program P can be written as a sequence of semi-positive Datalog programs (strata). Then, at each step of the sequence evaluation the predicates of the current term in the sequence are the EDB (Extensional Database) predicates of the following terms. Unfortunately, not every Datalog program can be stratified. This is the reason to introduce the well-founded semantics.

Well-founded semantics is based on the use of negation using a three-value model [16], where the truth values are not only true and false as in Boolean algebra but can also be unknown. It is used for Datalog programs that are not stratified and it can be computed via the alternating fix point procedure [18]. The idea behind the alternating fix point procedure is to build up a set of negative facts until the least fix point is reached. This is done in two steps, the first step the operator transforms an underestimation of the set of negative conclusions into an overestimation of these and in the second step the operator transforms the overestimation into a new underestimation, therefore it is alternating, finally the composition of the two steps (monotonic and antimonotonic) is a monotonous operator. For more details on this procedure the reader can consult [16, 18, 3].

2.1.2 Recursion in SQL

SQL is the most widely used relational database language. Conceptually it is based on relational algebra extended with many functionalities beyond it. Since relational algebra cannot express recursive queries, it was assumed that practical (implemementations of) SQL cannot express them either. This was formally proved in a study by Libkin in [19], where a formal proof that SQL cannot define recursive queries is given. The novelty was to show the standing conjecture that no matter what aggregate functions and arithmetic operations are allowed SQL cannot express transitive closure (and thus recursion).

Given this important limitation on its expressiveness, recursion was included in the SQL-99 standard [12], mainly based on the semantics of Datalog. It was developed via common table expressions (CTE 'S) that have a base SELECT statement and a recursive SELECT statement. This implementation allows to express reachability queries like DFS, BFS, topological sort, connected components, etc. In section 3.2.1 we present the sintaxis and examples of recursion in SQL.

Linear recurive queries The recursive operator implemented in SQL-99 has a constraint: it only can handle linear recursive queries. This implies that the recursive relation can be named only once within the recursive procedures. Some other features are not supported, e.g. mutual recursion and query solving involving an except clause. In general, termination consists of detecting that there are no further opportunities to add new tuples and cycles are manually controlled by limiting the number of iterations. Since queries must be linear and many interesting queries are not [2], optimizations have been proposed in [20, 21] (r-sql proposal)[16]. R-SQL proposal stands out, where they proposed an extension for recursion in SQL, adding the stratification techniques and fixpoint computations used for instance in Datalog. These techniques that consist of building a dependency graph and compute by fixed-point iteration stratum by stratum allow a legal mix with negation and therefore make recursion more expressive.

With recursion in SQL several known and useful graph algorithms were implemented in SQL. The graphs algorithms BFS and DFS were implemented with the recursive SQL operator [22]. The topological order was studied in [23], making a BFS starting from the node whose outdegree is zero. In the case of the connected components it was shown in [24] how to obtain them adding recursion, by building a table with all the connections of the nodes and choosing the minimum id as a connected component.

2.2 Recursion in Graph Query Languages

Despite having being researched since at least 30 years ago, graph query languages had not achieved significant visibility yet. Today, due to the emergence of non-relational databases (*NoSQL data bases*) and the graph structures that populate the Web, this panorama has changed and interest in studying its properties has grown. Graph databases have a basic model where information is expressed by nodes representing entities and edges representing relationships between them. A relevant point in this line of research and what sets it apart from the relational systems is that graph databases allow other types of queries by specifying graph patterns and navigation patterns. These functionalities, that are the building blocks

of G-CORE, were studied in depth in [2] where syntax and semantics were compared in three modern graph database languages: SPARQL[5], Cypher[6] and Gremlin[25].

2.2.1 SPARQL

SPARQL is the standard language for querying RDF graphs[26]. Given the growing interest in being able to express more complex queries, in particular inherently recursive queries and navigational queries, in SPARQL 1.1 were introduced Property Paths [5]. The idea of property paths is to allow the language to compute paths of arbitrary lengths using regular expressions. Although this allows the user to compute queries related to paths, the expressive power of this addition falls short in that it does not capture all functionalities of recursion. This has been studied by Barcelo et al. [27, 28, 29, 30], where it was shown that property paths cannot express navigational queries. A paradigmatic case is the following example: match all pairs of nodes connected by a path where each intermediate node belonging to the path satisfies some condition.

One of the first approaches for combining Property Paths and Recursion can be found in [31], where the authors propose an implementation of Property Paths using Recursive SQL. Since the main idea of recursion is to compute a query several times where the query uses a dataset that changes from iteration to iteration, one needs a closed language for defining this kind of queries. Since SPARQL queries are applied over RDF graphs, but the result is a set of mappings (which is analogous to a table), it was necessary to define Recursive Queries in terms of the CONSTRUCT operator, which is used to return RDF graphs instead. Reutter et al. [4] use this approach to give a formal definition of a Recursive Operator for SPARQL.

Also, Reutter et al. show queries that cannot be expressed with Property Paths (as the query defined in the example above), but can be expressed with the Recursive Operator. Finally they show that General Recursive Queries do not perform well in practice, defining then a fragment of the language that compute Linear Recursive Queries, the same fragment that is supported by SQL. They provide an implementation of Linear Recursion in SPARQL that was compared to the existing approach, showing empirical evidence that Linear Recursive Queries perform better than Property Paths in several use-cases.

Another attempt to add navigation to RDF was done in 2010 by Peréz et al [32], where they proposed nSPARQL that uses nested regular expressions, which is an extension of property paths. However, it was shown in [33] that like property paths, nested regular expressions cannot express important navigational queries.

2.2.2 Cypher

Cypher is a SQL-inspired declarative language for graph databases. It originated from the graph database *Neo4j*, which in turn was created in 2000 by the founders of Neo4j, and was designed with the goal of having a user-friendly graph database. In Cypher the queries are a composition of different clauses that are executed in a linear order, that is, from the beginning to the end of the text. Unlike SQL that specifies the answer of a query with `SELECT` at the beginning of it, Cypher specify answers and the end with the `RETURN` clause. The basis of Cypher's queries is found in the `MATCH` clause, which matches graph patterns.

Graph patterns in Cypher are expressed visually with basic patterns of the form *ASCII art* as $(a) - [r]- > (b)$ that expresses an arc r from node a to node b. Repeating edges and nodes are not allowed when matching the pattern in the graph. The `MATCH` clause returns a table that provides link information that matches the pattern. Recursion in Cypher has not been implemented directly. However, Cypher allows to express paths queries with the Kleene star operator, which can be applied to an edge property/value pair and it uses no-repeated-node/edge paths semantics. Despite of this functionality, currently it is not possible to express more complex regular expressions [2].

2.2.3 PGQL

PGQL is a new language for Property Graph data model (PG). Like Cypher and G-CORE, it uses graph pattern matching, it also follows the SQL structure and has path functionalities [11].

A basic PGQL query is made up of three clauses: SELECT, FROM and WHERE. In addition, it includes operators for graph pattern: edge matching, vertex macthing and path macthing, these operators are located in the WHERE clause. The result of a PGQL query forms a tabular result set with variables and their bindings. However, it is possible as in G-CORE to return a graph with a CONSTRUCT clause, also it allows groups of graph solutions with the SELECT clause and an UNION operator.

Currently, recursion is not directly implemented in PGQL there is no syntactic or semantic definition of a recursive operator. Despite that, in PGQL there are path functionalities. As in SPAQRL, you can use regular path queries to express navigational and path queries. However, since the data model is PG, PQL extends regular paths queries with general expression, allowing to compare vertices and edges along paths.

2.2.4 G-CORE

G-CORE has the standard approach of graph query languages, using matching and filter as the basic machine for evaluation. The syntax and semantics of G-CORE is inspired by current graph query language.

Being a proposal for a graph query language, G-CORE has not been fully implemented. G-CORE does not have a recursive operator designed or implemented. However, it has path functionalities that allow you to have a limited form of recursion. As discussed in chapter 1 section 1.1.1, G-CORE treats paths as first-class citizens. Therefore, paths can be manipulated (eg. filtered, searched, stored) and like PGQL regular path queries are allowed using the PATH clause to express more complex regular path expressions.

Given the definitions and applications of recursion in Datalog and SPARQL we are ready to give a formal definition of recursion in G-CORE.

Chapter 3

General Recursion

In this chapter we introduce the concept of a general recursive operator for G-CORE based both on the recursive SPARQL operator defined by Reutter et al [4] and the recursive SQL operator [12]. The idea is to allow the user of the query language to explicitly define an order on which to base the recursive query and so provide more flexibility when writing queries. This would permit to code in a more natural and simple manner several algorithms.

In the cases of SPARQL and SQL's recursion, the order that guides the recursive process is hooked to the set-order of sets of partial results and thus prevents intuitive codification of various problems, in particular certain graph algorithms. The general idea that we present is to separate the order that guides the recursion from the successive results that the query gives in the process of its recursive evaluation.

We will first review notions of order and fixed point that are at the base in the recursion process. Then, we will see how the syntax and semantics of recursion are defined in SQL and we will explain the limitations that this given set-based recursion principle presents. Finally we will extend the definition of SQL to define a recursive operator in G-CORE, defining an abstract structure that follows an order and that can easily encode classic graph queries.

In the following, we briefly review concepts about partial orders taken from the literature [34].

Partial Order Let P be a set. A partial order on P is a binary relation $\leq$ on P such that, for all $x, y, z \in$ P,

- $x \leq x$
- $x \leq y$ and $y \leq x$ imply $x = y$
- $x \leq y$ and $y \leq z$ imply $x \leq z$

This ordered set P will be denoted as $(P, \leq)$.

Fixed Points A definition of a recursive procedure or function has two basic parts. The base step where one specifies the value of the function for the first terms and the recursive

step where one defines a rule for finding subsequent values using previous values. A fixed point for a recursive function $F : D \rightarrow D$ is an element $d \in D$ such that $F(d) = d$. That is, it is a point in the domain of the function that remains stationary.

The fixed point theory has a fundamental role in the definition of recursive operators in database languages. Usually the recursive operator q is a fixed point operator, such that there is an output table T such as $q(T) = T$ (fixed point). This concept is key to recursion in SQL, as we will see in the next section.

Our recursive operator proposal is based on the one defined in SQL and SPARQL.

3.1 The need for a general recursive operator

In this section we will show the need to define a general recursive operator for queries. We will begin by recalling how recursion is defined in SQL and SPARQL, and based on it we will argue the advantages that a more general notion could bring.

Recursion in SQL

Recursion in SQL was introduced in SQL-99 standard out of the need to be able to express queries based on transitive closure, which are not expressible in relational algebra [3]. Recursion in SQL is based on the notion of common table expression (CTE). Generally speaking, a common table expression is a temporary (auxiliary) table that is built by means of queries that accumulate rows based on certain relationships with the rows that are already in the table of the results of the main query.

Let us consider the following example:

Example 3.1.1 Consider the following parent's table that establishes the father-son and/or mother-son relationship:

parent	**child**
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

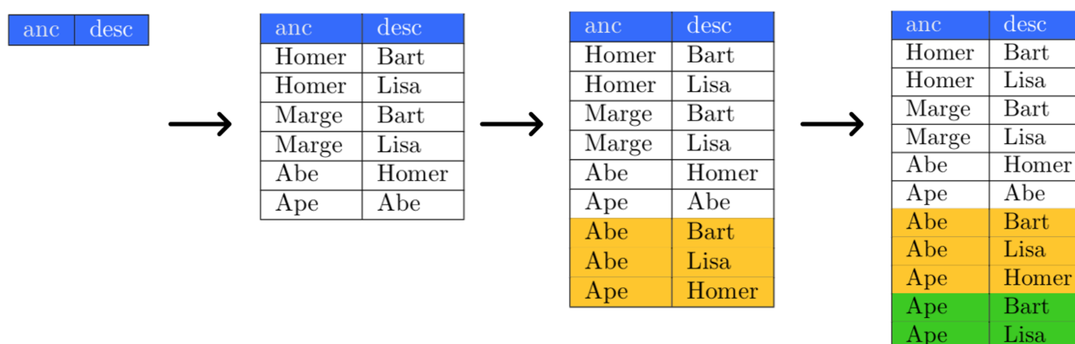
Table 3.1: Parent Table

If we know the schema of the table and want to find all the ancestors of somebody, in principle (i.e. if we do not know the instance table in advance) this cannot be obtained using a basic

SQL query. Thus a recursive query is needed, like the following recursive query: [1]

```
1  WITH RECURSIVE Ancestor(anc,desc) AS (
2   SELECT parent,child FROM Parent
3  UNION
4   SELECT Ancestor.anc, Parent.child
5   FROM  Parent INNER JOIN  on
6   Ancestor.desc = Parent.parent)
```

A recursive query is made up of a base case (line 2 of the query) and another query that defines the recursive relationship (lines 4, 5 and 6). In this case, our base case builds all the "direct" relationships of kinship, e.g. father-son. Then the recursive case deduces more ancestors by taking into account the ones already added in the previous steps as seen in the following figure:



anc	desc

anc	desc
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

anc	desc
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer

anc	desc
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer
Ape	Bart
Ape	Lisa

Figure 3.1: Successive partial results given by the query above over the Parent table

It is important to note that we do not want the query to run indefinitely. This is why the semantics of this recursion is based on the fixed point semantics defined above, which in simple terms, means that the query will end when a fixed point of the successive set of tuples obtained by the recursive part of the query is reached. Note that we are assuming here that the recursive function (in the example the query in lines 4, 5 and 6) is monotonously increasing, that is, in every step the set of tuples obtained is a superset of the previous one, and reaches a fixed point (in this case given by the combinatorics of possible pairs given by the finite size of the table).

[1] Taken from `https://www2.cs.duke.edu/courses/fall14/compsci316/lectures/09-recursion-notes.pdf` (visited on January 4, 2020)

Recursion in SPARQL

Reutter et al [5] proposed a recursion operator based on SQL in 2015. They formalize the syntax and the semantic of the recursive operator and develop algorithms for evaluating it in practical scenarios.

Definition 3.1.1 (Syntaxis of a recursive query in SPARQL)**.** The syntax of a recursive query in SPARQL is defined as follows:

```
WITH RECURSIVE t AS {q1} q2
```

where t is an IRI from I, q1 is a c-query (construct query in SPARQL), and q2 is a recursive query.

Definition 3.1.2 (Semantics of a recursive query in SPARQL)**.** The answer of a recursive query q, denoted $ans(q, D)$, is equal to $ans(q_2, D_{LFP})$, where D_{LFP} is the least fixed point of the sequence:

$$D_{i+1} = D \cup \{\langle t, ans(q_2, D_i)\rangle\} \text{ for } i \geq 0.$$

Therefore, as in SQL, there is a temporary structure t that is built by joining D with the evaluation of q_1 over D_1 until $D_i = D_{i+1}$.

Example 3.1.2 Example of recursive query in SPARQL This is an example taken directly from [5]. Given an RDF G graph, we want to obtain all the nodes (a, b) (where each node represent scientific articles) such that a and b are linked by a path of $wasRevisionOf$ links and that they were edited by the same user. The query can be computed as follows:

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp AS {
CONSTRUCT {?x ?u ?y}
FROM NAMED <http://db.ing.puc.cl/temp>
WHERE{{
?x prov:wasRevisionOf ?y .
?x prov:wasGeneratedBy ?w .
?w prov:used ?y .
?w prov:wasAssociatedWith ?u}
UNION{
GRAPH <http://db.ing.puc.cl/temp> {?x ?u ?z} .
GRAPH <http://db.ing.puc.cl/temp> {?z ?u ?y}}}
}
SELECT ?x ?y
FROM <http://db.ing.puc.cl/temp>
WHERE {?x ?u ?y}
```

This query is made up of a base query and a recursive query. First build the triples $(x, u.y)$ where x e y are joined by an edge of $wasRevisionOf$ and u is the user generating this revision. Then comes the recursive part so if (x, u, z) is in the temporal graph and (z, u, y)

is in the temporal graph then (x, u, y) is added to the temporal graph.

A general recursion operator

Note that in SQL and SPARQL, the recursion is based on the "natural" set order given by the inclusion order relation. That is, in each iteration the order of subsets of tuples is used as the base order over which the recursive function operates and terminates. This way of defining recursion has some limitations. In fact, there are queries that with the semantic definition given in SQL recursion cannot be easily coded or queries whose intermediate set of results is not strictly non-decreasing. A natural idea to improve this situation would be to define an "external" order to serve as the basis for the recursion.

A simple example of this is a query that returns the set of nodes at depth k in a tree. In this case, the natural order for the SQL query is the size of the table that grows when the visited nodes are added. However, it seems more intuitive that the recursion be done on the depth of the nodes that are visited reaching the fixed point when the depth stabilize; we will see this example in more detail in the next section.

On the other hand, many graph algorithms use some form of negation (e.g. those we studied to express them in G-CORE, see chapter 4 section 2.2). Negation in recursion produces problems related to finding the fixed point [16]: The operator may not have a fixed point; or the operator could have several minimal fixed points; or the sequence of applying the operator may not converge or the sequence of operators; or could converge but the limit is not necessarily the fixed point. To avoid these problems, recursion in SQL and SPARQL do not allow negation in recursive queries. This constitutes a limitation in the expressive power of queries as negation is needed to code interesting graph algorithms.

Negation in recursion can be managed by adding inflationary fixed point or partial fixed point semantics. However, this usually makes the definition of recursion cumbersome both for users and developers. We claim that this burden can be avoided.

In the next section we will give an idea of how can we achieved the idea of an external order and how can be extended and thus be able to avoid the problems that currently exist with negation.

3.2 A proposal to generalize the expression of recursive queries

Well defined recursion, as we saw, is based on a partial order, a starting point and a mechanism to determine when to stop the process. The latter could be a limiting point when it is known in advance, or in general is the fixed point of the recursive procedure (i.e where the ordered function that the recursive query defines stabilizes).

The intuitive idea of current recursive queries goes as follow. A recursive query is a query that produces successive calls to queries over successive datasets. As any recursive procedure, it is is composed of two parts: a) the base query q_0, and b) a recursive query q_r. The process begins by querying the dataset to be queried D with the query q_0. The answer of $q_0(D)$,

say a dataset D_1, is then queried by q_r giving D_2, and so on. At some point it has to stop and produce the final answer. Usually this is done when the increasing sequence of datasets $D_1, \ldots, D_n$ reaches a fixed point.

The new idea we introduce here is to separate (to modularize) the order of the recursion from the sequence of partial datasets $D_1, \ldots, D_n$. In fact, essentially in the above process there are three structures involved:

1. The base query q_0.
2. The recursive query q_r.
3. A partial ordered set $(A, \leq)$ and mapping H from results to A.

In SQL one does not need to define explicitly (3) because the order set A in (3) is given by the set of results of the queries q_0 and q_r (possibly applied iteratively) over D and the order $\leq$ is subset inclusion $\subseteq$. Not making explicit the order, that in principle could be seen as a help to the developer [2], becomes an obstacle when more flexibility is needed to code recursive problems that do not fit easily in the order (Partial results of q_r, $\subseteq$).

What we are going to do is incorporate the feature to define (3) independently of the "natural" order given by successive results of q_r. In some sense, we will abstract the implicitly given partial order of the successive dataset results into an explicit (and hopefully simple) partial order set $(A, \leq)$ and a non-decreasing function F over this partial order. See the following figure 3.2:

[2] This seems to be an illusion. The fact is that developer writing an SQL recursive query cannot devise mentally the recursive process he/she is going to code. What we are doing in our proposal is to make this implicit thought explicit in the code.

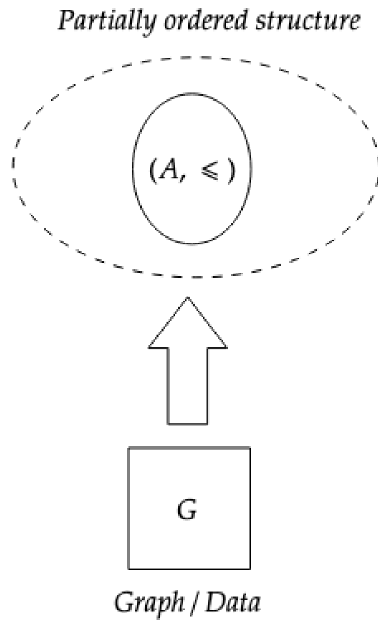


Figure 3.2: Mapping from a graph G to a partially ordered structure

To make clearer the idea of how this procedure will work (the syntax and semantics are presented in the next subsections), consider the diagram in Figure 3.3.

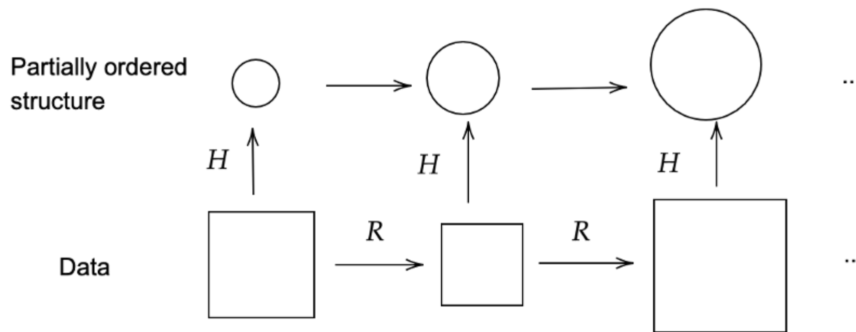


Figure 3.3: Schema of the different steps of the mappings from the data (graphs) to a partially ordered structure while the recursion is taking place.

In Figure 3.3 the sequence of squares below show the recursion operator R (defined by the query q_r). Classically this sequence will stop when the operator R finds its fixed point (i.e. the squares start repeating). In our proposal, we obtained (by means of the map H) from

each partial result an element in the abstract partial order $(A, \leq)$. With this setting, the recursive operator R is reflected in the partial order $(A, \leq)$. That is, there a map H from the partial result datasets to the order $(A, \leq)$ so that $H(D_i) \leq H(D_{i+1})$ for every relevant i.

Thus the developer, similarly to the classical case, has to code the query q_r (and of course the base case q_0); but now, instead of worrying about making monotonous the partial results of queries q_j, can rely on order $(A, \leq)$ that usually is much simpler. Of course this brings a cost, and now will have to code (to make explicit to the compiler) the order $(A, \leq)$ and how at each step the corresponding order for each result is obtained from that result (this is the map H). This will become clear with an example.

Example. Consider the following problem: Obtain the nodes at distance k from a tree T presented as a table with schema $Tree(Parent, Child)$. (Technically it does not need a recursive query, e.g. a join of k copies of T; but we may not know k in advance or it could be too big).

A. Our method gives the following simple and intuitive items to be coded:

1. q_0: a query to get the root of T.
2. q_r: a query that gets the children in T from the previous result $q_{r-1}(T)$.
3. The order $(A, \leq)$ that are the natural numbers with usual order. The map H gives the depth of each iteration. Stop at k.

B. Classically we would need to code only two items, but the second is not obvious at all:

1. q_0: a query to get the root of T.
2. q_r: the challenge is to find a query q_r that using the set order of its partial results gives the right answer. Not obvious in this example because with traditional recursion we use the order of subset so we can't reach a fixed point.

Before going on, let us point out one more problem of the classical view to code recursion in query languages that motivated our approach. Possibly because it was designed for tables, coding recursion gets much more complex for the developer when dealing with graph structures, and in particular, with property graph structures. We will try to argue this with the study cases in the next section.

Next we will formalize the syntax and semantics of this idea.

3.3 Syntax of general recursive queries in G-CORE

The following syntax is motivated by the one in SQL and the one Reutter *et al.* [4] gives for SPARQL. They specify the temporary object (table, graph) over which the recursion acts, and a base case and the recursive step.

A recursive query q will be defined as follows.

Definition 3.3.1 (Syntax of Recursive queries in G-CORE)**.** A recursive query in G-CORE is defined as:

```
(R)      WITH RECURSIVE t AS (
            ( BASE q0
              RECURSIVE qr
            )
            OVER ORDER ( a )
         )
```

where each of the terms in the code are defined as follows:

1. t is a temporary PPG (a name to the partial results that will be required when defining the recursive query q);

2. q_0 is a usual (standard) G-CORE query that computes the base case of the recursive query;

3. q_r is a usual (standard) G-CORE query that can use the temporary graph t,

Until here we only changed the syntax of SQL and SPARQL by expliciting that q_0 is the base query and q_r is the recursive query.[3]. Now, what follows is new, not just a change of notation:

4. a is the variable that represent $H(t)$ in the code (inside q_r). Recall that H is the map that gets the abstract order from a given t. It should be of the type of A where $(A, \leq)$ is the partial order.

Notes. Let us call the attention to the next steps that could help the reader understand what is going on:

1. The dynamic (auxiliary) graph t will be instantiated recursively as follows when the query (`R`) is issued over a graph (database) G:

   - Base case: $q_0(G)$
   - Recursive case: $q_r(G, t)$ (the recursive step can consult two data-graphs: the original graph G and the temporary graph t.)
   - The map H and the order $(A, \leq)$ do not show up in the code, but must be clear when defining the recursive query. It is represented by the variable a.

3.3.1 Nested Recursion

Note that in the syntax of recursion presented above, one would not be able to express nested recursion procedures, because it is needed that the base case q_0 and the recursive step q_r be

[3]In SQL and in SPARQL recursion by Reutter et al. this is done by the convention that what comes after the `"WITH .. AS RECURSIVE ("` is the base query followed by a `UNION` operator and then the recursive query; we avoided the confused message that the `"UNION"` installs, that in our case –as we generalized the order– does not make sense.

standard G-CORE queries.

The general syntax thus will be a slight update as follows:

```
(NR)        WITH RECURSIVE t AS  (
               ( BASE q0
                 RECURSIVE qr
               )
               OVER ORDER ( a )
            ) q2
```

where everything is as in Definition 3.3.1, and q_2 is a recursive G-CORE query as in (NR), a simple G-CORE query, or nothing.

3.4 Semantics of general recursive queries in G-CORE

Let G be a PPG (the database to be queried) and q a recursive G-CORE query as in Definition 3.3.1 (the nested case is a straightforward extension).

Classically, the answer $\mathrm{ans}(q, G)$ of a recursive query would corresponds to the fixed point of the sequence $t_0, \ldots, t_n$ of instantiations of the temporary graph t given by:

$$t_0 = q_o(G), \tag{3.1}$$

$$t_{n+1} = q_r(G, t_n). \tag{3.2}$$

In our proposal, the answer $\mathrm{ans}(q, G)$ correspond to the last element of the sequence $t_0, \ldots, t_n$ obtained when the sequence $a_0, \ldots, a_n$ (where each $a_j = H(t_j)$ and $a_j \leq a_{j+1}$ in the order $(A, \leq)$) stops (finds its fixed point of a limit given by the user). We recall, and this is one of the main advantages of our approach, namely, that we do not impose any condition on the sequence $t_0, \ldots, t_n$, i.e. that it should be increasing, etc. We only rely for termination on the order $a_0, \ldots, a_n$ over $(A, \leq)$.

This can be visualised in the following diagram:

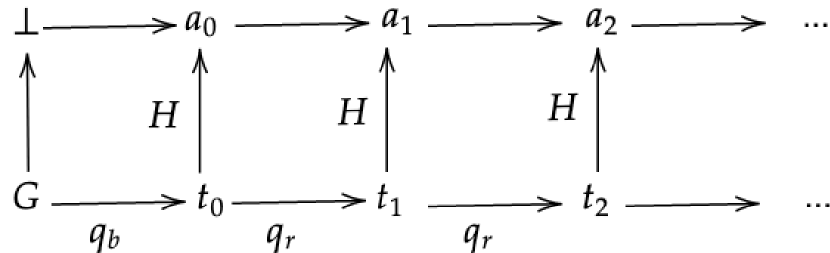


Figure 3.4: The recursive process of evaluation of a query. Each q_r gives the next t_j based on the previous one and on G ($q_r(G, t_{j-1})$). The map H gives in each step the corresponding element of the order.

The pseudo-code of the algorithm to compute ans(q, G) is the following:

Algorithm 1: Computing the answer of a recursive query in G-CORE

```
Data: A database PPG G; a recursive query q with components (q_0, q_r) (see
      Definition 3.3.1)
1 Variables t of type PPG; a and a.old of type A
2 Set t = q_0(G)
3 Set a = H(t)
4 Set a.old = ⊥
5 while a <> a.old do
6     Set t = ans(q_r, t + G)
7     Set a.old = a
8     Set a = H(t)
9 return t
```

When the order $(A, \leq)$ is the one of the results $t_1, \ldots, t_n$, then we can get rid of H because $H(t) = t$ and the order is $\subseteq$. In this case the code is more simple:

Algorithm 2: Computing the answer of a recursive query in G-CORE

```
Data: A database PPG G; a recursive query q with components (q_0, q_r) (see (R))
1 Variables t and t.old of type PPG
2 Set t.old = ⊥
3 Set t = q_0(G)
4 while t.old <> t do
5     Set t.old = t
6     Set t = ans(q_r, t + G)
7 return t
```

Note that were using for the order $(A, \leq)$ built-in orders (e.g. that of natural numbers, strings, etc.). One could allow the user to define more complex orders (i.e., lexicographic, etc.) over which the recursion will be based. In that case it should be enforced some way of assuring that the linear order is correct (i.e. there are no cycles, etc.) to avoid infinite loops. A simple such idea is defining a maximum number of iterations for the recursive query. This idea has been used previously, in the recursion in SPARQL [4], where a clause "MAXRECURSION K" was proposed, with K a natural number. This bound limits the number of iterations and the query is guaranteed to terminate their execution.

3.5 An Example: Nodes at k distance from the root

Let us consider a toy example to illustrate the idea. Suppose we have a company represented in the following figure as a tree where some people are developers, other project managers and the root is the company manager:

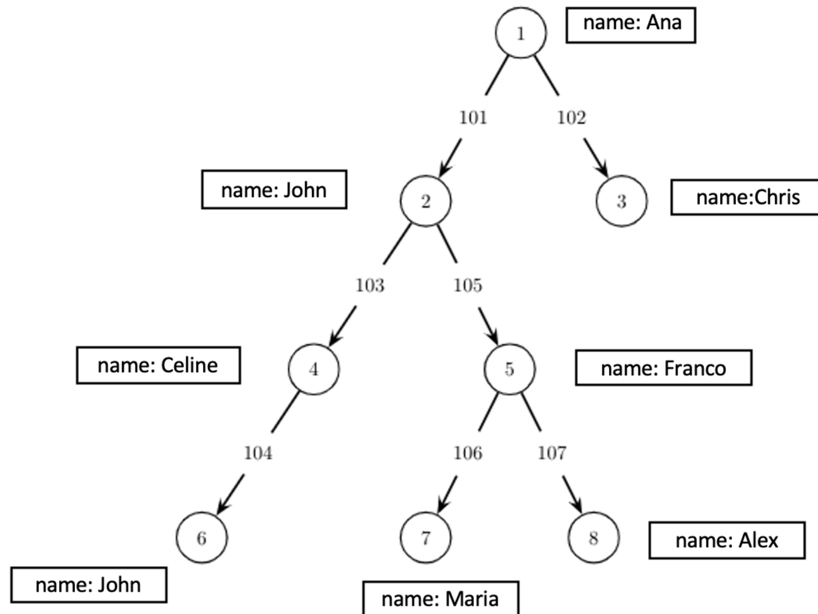


Figure 3.5: A hierarchical company

If we want to know all the developers of the company, that is, all the nodes that are at distance 2 from the root; we can first consult the root node and then go through the graph by levels. So we have the following recursive query: We can traverse these graph with the following query using general recursion:

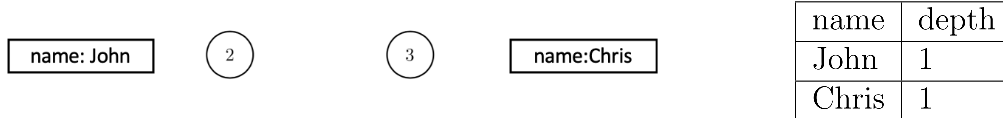
```
WITH RECURSIVE t as (
 BASE   CONSTRUCT (n) SET n.depth:=0
        MATCH (n) on G
        WHERE n.name='Ana'
 RECURSIVE (
       CONSTRUCT (n)
       SET n.depth = if x.depth+1=k  then x.depth else x.depth+1 end
       MATCH (x)-[:BossOf]->(n)  on G
             (x) on t
 )
OVER ORDER ( SELECT MAX(n.depth) FROM t )
```

First the query base is evaluated once and gets the root of the tree:

name	depth
Ana	0

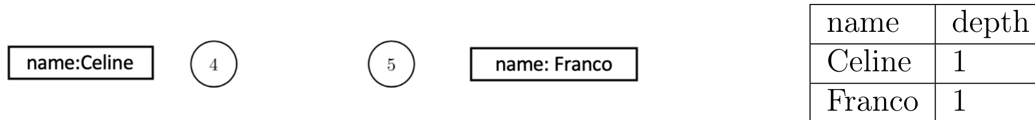
Table 3.2: Base Case Nodes at k distance

Then it continues until the depth property does not change:



name	depth
John	1
Chris	1

Table 3.3: Recursive step 1 Nodes at k distance



name	depth
Celine	1
Franco	1

Table 3.4: Recursive step 2 Nodes at k distance

In this case, the function does not use any operator and uses only nodes built in each iteration. The associated structure are natural numbers that correspond to the depth of the node. In other words, we have to keep track of:

- G_i: set of nodes
- a_i: natural numbers that correspond to the depth of the nodes, example Ana corresponds to 0
- The order in this case is the usual order of the natural numbers.

In the first iteration, the root is constructed. Then the nodes that are at a distance of 1 from the root and saving the depth of each node as a property. Note that it will stop when the depth does not change anymore. We want this to happen when the nodes are at distance 2 from the root. This can be achieved by putting a condition in the SET clause of the construct, specifying the depth $k-1$ when the value k is reached in the next iteration.

This is a simple example of how the general recursion can be used; of course this problem can be solved with the recursion defined in SQL but we will see later with the matching problem that this type of recursion is needed to express that query.

3.6 Data Complexity

It was shown by Angles et al.[8] that G-CORE can be evaluated efficiently in data complexity. They show that given a fixed query q, evaluating q on a PPG G can be computed in polynomial time. For example, the case of having a fixed pattern ϕ on a PPG G can be computed in polynomial time by looking for all combinations of variables and identifiers of nodes and edges in G that match the pattern ϕ. In the case of a variable p that represents a path from a node u to a node v that forms a regular expression r, p is replaced with the shortest path from u to v applying standard automata theory techniques in conjunction with Dijkstra algorithm. They also show that subqueries can be evaluated efficiently.

Finally the clause CONSTRUCT on top G and the set of bindings generated by the match clause can be carried out in polynomial time given that the operations allowed with this clause are aggregation and grouping functions on top of bindings generated by relational algebra operations.

What is the data complexity of for a given (fixed) recursive G-CORE query q_r? Analyzing the Algorithm 1 of our recursive operator proposal, it can be straightforward checked that the size of the sequence of evaluations of the query q_r are:

$$|G| = d,\ p_0(d),\ p_1(p_0(d) + d),\ p_1(p_1(p_0(d) + d) + d),\ \ldots$$

where $p_j(x)$ are polynomials (because the data complexity of a fixed query in G-CORE is polynomial). That is, the k-element of the sequence is a polynomial of degree k in the size d of the original graph dataset G.

Thus, the data complexity of the Algorithm will depend on: (a) The number of iterations being performed; and (b) the size of the intermediate results, that is, the data complexity of q_r. For example to have a polynomial data complexity, we would need a constant number of iterations (i.e. that do not depend on the datasets to be queried), or a polynomial number (in the original dataset).

3.6.1 Note: On Linear Recursion.

Traditionally, in SQL and SPARQL, there is a distinction between recursion in general and linear recursion. This last one occurs when in the j-step of the recursive procedure the query q_r only uses plus the original graph to be queried G, plus the difference $t_j - t_{j-1}$ (instead of the whole t_j). This makes sense because in SQL and SPARQL the output of each step, namely t also works as the "guiding" order for the recursion (and thus it should be an increasing sequence $t_1, \ldots, t_n$).

In [35] we explored a preliminary linear version of recursion in G-CORE. It followed the lines of classical recursion philosophy (that is, without an additional order that guides the recursion). It followed pretty much the philosophy and methodology of recursion in Reutter et al. [4].

In the approach we follow here, it does not matter the order-behaviour of the temporal t_j because as the "guiding" order for the recursion we use another order (the $(A, \leq)$ given by the map H). So the question arises: What does "linear recursion" mean in our case?

First, we would like to point out that we do not need to distinguish different syntax (and a different way to compile) linear versus non-linear queries. Second, in our approach one should not be worried about "doubling" each step the size of the data because we do not union the current result with the previous one in order to keep the "monotonicity" of the sequence of results.

Thus, in our case, a good estimator is to define that a query q will be linear if the t is used (occurs) only once in q_r, that is, it is called only once in the code. This is the standard baseline to determine if a query is linear [16, 12, 4, 24].

In our case we can be more relaxed. In fact, a query can be considered linear, if the size of the elements of the sequence of intermediate results $t_1, \ldots, t_n$ is approximately that of G, that is, it does not increase in a non-linear way (e.g. is a subgraph of G, or a construction that keeps similar size of that of the source, etc.).

Now that we have defined recursion we can now see how it can be used to express more complex graph problems, the next chapter we will show how using this new operator one can code classical graph algorithms.

Chapter 4

Graphs algorithms with recursive G-CORE

In this chapter we show how the general and linear recursion syntax defined in the previous chapter works by applying it to classical graph problems. Graph algorithms have enough complexity and interest to expose different features of the recursive formalism we introduced. We chose graph algorithms that can be solved in polynomial time included in the book [9].

We start by giving a general definition of each problem. Next, we choose a known algorithm that solves the problem in polynomial time and present an intuitive idea of how it solves the problem. Finally we show the coding of the respective recursive query that solves the problem, and develop an example, explaining at each step how the answer is constructed and which order follows the general recursive operator.

4.1 Topological Sorting

Topological order in a directed graph without cycles (DAG) is a linear order of its nodes. One of its main applications is the ordering of tasks to be executed, tasks whose precedence is given by means of a graph.

Definition 4.1.1. Let G be a directed graph without cycles. A topological order of G is a linear order $\leq$ of its vertices such that if the edge (u, v) belongs to the graph, then the vertex $u \leq v$ in the order.

Intuitively it is not evident how to express topological sort in G-CORE. We will show how to write this query with the proposed linear recursive operator presented in the previous chapter.

Intuitive idea Suppose you have the following PPG shown in Figure 4.1. To find the topological sorting of a graph, we first will find all the nodes that do not have incident edges and we are going to set them with depth 0. This will be the base case. Then, we are going to assume we have a set of nodes at depth k, and from them, compute the children of this

set and set their depths at $k + 1$. Clearly in this case, the abstract structure is the set of depths (i.e. natural numbers) and in this case we choose to compute maximum depth of the graph constructed so far.

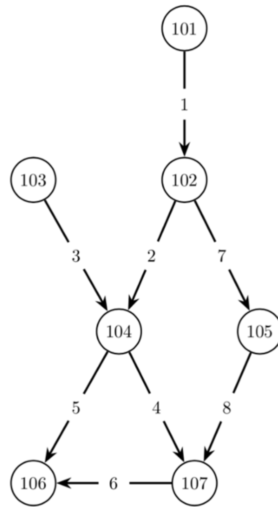


Figure 4.1: A Directed Graph

The following query illustrates the codification in G-CORE of this algorithm for topological sort.

```
 1    WITH RECURSIVE t AS (
 2    BASE (CONSTRUCT (x) SET x.depth :=0,
 3             MATCH (x) ON G,
 4             WHERE NOT EXISTS (CONSTRUCT (),
 5                                 MATCH  (y)-[:Label]->(x) ON G))
 6
 7    RECURSIVE (CONSTRUCT t UNION (x) SET x.depth:= 1 +
 8                                          (SELECT MAX(z.depth) MATCH (z) ON t)
 9               MATCH (x) ON G,
10                      (n) ON t
11               WHERE EXISTS (CONSTRUCT (),
12                              MATCH   (n)-[:Label]->(x) ON G)))
13
14    OVER ORDER (SELECT MAX(x.depth) MATCH (x) ON t)
```

```
SELECT z.id,
MATCH (z) ON t,
ORDER BY MAX(z.depth) DESC
```

The above query first looks for those nodes which have no in-going edges (base case) and are

assigned with depth equal to 0 as a property so in this case we will have:

G^1_{temp}

$a_1 = 0$

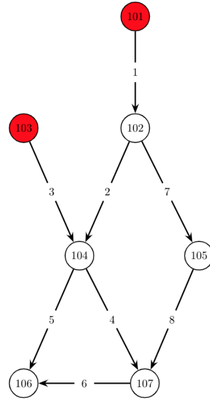


Figure 4.2: G^1 topological Sorting

x.id	x.depth
101	0
103	0

Table 4.1: Base case topological sorting

We have the temporary graph G^1_{temp} which turns out to be two nodes whose depth is set to 0.

Then comes the recursive case and the nodes which have in-coming edges from the nodes obtained in the base case are added with property depth equal to 1 as shown in the following figure:

G^2_{temp}

$a_2 = 1$

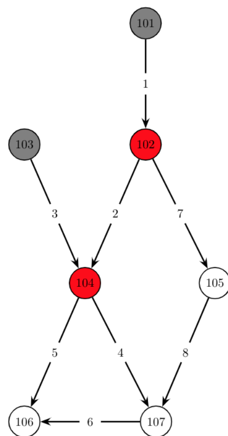


Figure 4.3: G^2 topological Sorting

x.id	x.depth
102	1
104	1

Table 4.2: Recursive step 1 topological sorting

Then we evaluate the recursive query over the temporary graph G_{temp}. Note that the next element in the abstract structure is now $a_2 = 1$ which is the max depth of the property depth of the graph constructed so far.

And so on, we proceed with the following step-by-step evaluation of the recursive graph:

G^3_{temp}

$a_3 = 2$

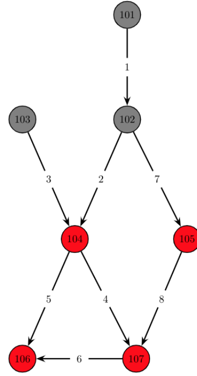


Figure 4.4: G^3 topological Sorting

x.id	x.depth
105	2
104	2
107	2
106	2

Table 4.3: Recursive step 2 topological sorting

G^4_{temp}

$a_4 = 3$

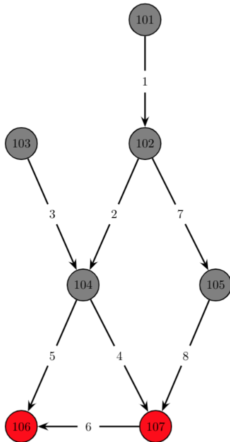


Figure 4.5: G^4 topological Sorting

x.id	x.depth
107	3
106	3
107	3
106	3

Table 4.4: Recursive step 3 topological sorting

G^5_{temp}

$a_5 = 4$

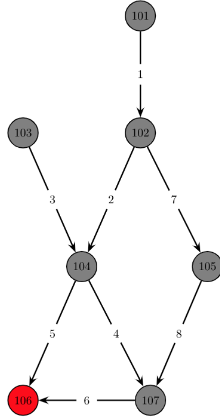


Figure 4.6: G^5 topological Sorting

x.id	x.depth
106	4
106	4

Table 4.5: Recursive step 4 topological sorting

Finally we have $a_5 = a_6$ and we stop in the fifth iteration. That is, because the node 106 does not have any outgoing edge, therefore in the sixth iteration G^6_{temp} is the empty graph thus the max depth of the temporary graph G_{temp} does not change (and so does the value `(SELECT MAX(x.depth) MATCH (x) ON t)` and we got a fix point. Then we can obtain for each node its depth property, which tells us its place in the topological sorting.

Topological sorting has been coded with recursion in SQL [23], where an idea similar to the procedure presented here is followed. The interesting thing about our approach is that we use an order, namely the depth of the nodes, that is not based on inclusion of results of intermediate results.

4.2 Minimum Spanning Tree

The next problem we will code is the problem of finding the minimum spanning tree in a graph with weights. This problem is important in graph theory given its various applications in real life, for example in the field of electrical, telephone, communication, etc. The following is a formal definition of spanning tree and minimum spanning tree respectively:

Definition 4.2.1 (Spanning Tree)**.** A spanning tree of a graph G is a subgraph of G that includes all nodes of G and that is a tree.

Definition 4.2.2 (Minimum Spanning Tree (MST))**.** A minimum spanning tree is a spanning tree whose weight is as small as possible.

Consider the following PPG that represents a graph with weights:

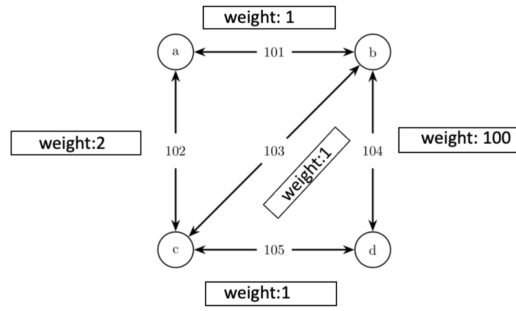


Figure 4.7: A weighted general graph

Intuitive idea. There are several algorithms that solve the problem of finding a MST. We will implement the algorithm of Prim [36], whose basic recursive idea is to choose at each step a new edge with the minimum weight to add the tree under construction.

To express the query in G-CORE we will use the general recursion operator defined in the previous chapter. The idea is to build a temporary graph with the union operator and in each iteration a new edge is joined to the graph until all the nodes are included. For this query we will use the difference operator that G-CORE offers in its documentation so that in each iteration we can obtain the nodes that are not yet contained in the temporal graph t. In general terms, for an edge to be joined to the temporal graph, one of its nodes must belong to the temporary graph and there must not be another edge that has less weight among the possible candidates. We choose to set the numbers of edges in the graph as the elements of the abstract structure.

The query we propose to find the MST of the graph is as follows:

```
1  WITH RECURSIVE t as (
2   BASE (CONSTRUCT (a)-[:e1]->(x)
3           MATCH  (a)-[:e1]->(x) ON G
4           WHERE  e1.weight =<
5                        (SELECT MIN(e.weight) MATCH (a)-[:e]->(b) ON G
6           LIMIT 1)
7
8   RECURSIVE( CONSTRUCT t UNION (a)-[:e1]->(x)
9               MATCH  (a)-[:e1]->(x) ON G
10                      (a)           ON t
11                     (x)           ON G \ t
12              WHERE e1.weight =<
13                        (SELECT MIN(e.weight) MATCH (a)-[:e]->(x) ON G
14              LIMIT 1 )
15  OVER ORDER (SELECT COUNT(e1) from t)
```

In the example, first we have the base case that constructs the first step (a tree of two nodes,

a and b) whose edge has minimum weight in G; that is, we obtain:

G^1_{temp}

$H(G_1) =$ Number of edges in G_1

$a_1 = 1$

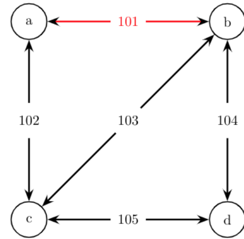


Figure 4.8: G^1 Minimum Spanning Tree

x.id	e.id	y.id
a	101	b

Table 4.6: Base step Minimum Spanning Tree

Then comes the recursive step, where an edge of minimum weight is added such that one of the nodes belongs to the temporary graph and the other does not. This is done with a sub query that, for the edges that match the pattern, chooses one of minimal weight in the original graph (e.g. without considering the edges of the temporary). Then we obtain:

G^2_{temp}

$H(G_2) =$ Number of edges in G_2

$a_2 = 2$

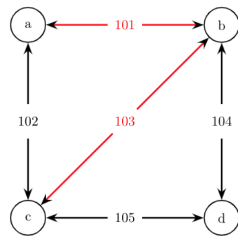


Figure 4.9: G^2 Minimum Spanning Tree

x.id	e1.id	y.id
a	101	b
b	103	c

Table 4.7: Recursive step 1 Minimum Spanning Tree

G^3_{temp}

$H(G_3) =$ Number of edges in G_3

$a_3 = 3$

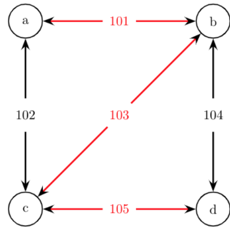


Figure 4.10: G^3 Minimum Spanning Tree

x.id	e1.id	y.id
a	101	b
b	103	c
c	105	d

Table 4.8: Recursive step 2 Minimum Spanning Tree

Finally we have $G^4_{temp} = \emptyset$ and $a_4 = a_3 = 3$. Therefore, we get all the edges that belong to the MST.

This problem has not been studied in other languages with recursion and we believe that the fact that you have to check conditions on nodes and / or edges makes it necessary to have negation. However, as we pointed out, SPARQL and SQL do not allow it in their recursive operators.

4.3 Eulerian Circuit

This problem is one of the fundamental polynomial problems in Graph theory. The problem was inspired by the city of Königsberg, which had two island connected by seven bridges. The question was if there was a way to walk around the city crossing each bridge exactly once.

Definition 4.3.1. Eulerian Circuit o Cycle A circuit is a walk that starts and ends at a same vertex, and contains no repeated edges. A Eulerian circuit in a graph G is a circuit that includes all vertices and edges of G.

We will show how an algorithm to solve Eulerian circuit problem can be expressed with our general recursive operator. We note that expressing the Eulerian path in Graph data was studied by Barceló et al. [27] but using regular expression with memory rather than recursion.

The existence of a Eulerian circuit in a digraph depends on the degree of its vertices, being sufficient that the in-degree and out-degree of each vertex be equal. Thus in G-CORE you can write the following query to collect these values (in degree and outdegree of each node) and then just have to compare them for each node.

```
CONSTRUCT (n) SET n.in:=COUNT(x), n.out:=COUNT(y)
MATCH (x)-[e1:]->(n),(n)-[e2:]->(y) ON G
```

In what follows, we will implement the algorithm for this problem proposed by Fleury [37]. It proceeds by repeatedly removing edges from the graph in such a way, that the graph remains Eulerian.

1. Choose any vertex v of G and set current vertex equal to v and current trail equal to the empty sequence of edges;

2. Select any edge e incident with the current vertex but choosing a bridge only if there is no alternative;

3. Add e to the current trail and set the current vertex equal to the vertex at the "other end" of e;

4. Delete e from the graph. Delete any isolated vertices.

5. Repeat steps 2 – 4 until all edges have been deleted from G.

Let us consider the following graph to fix ideas.

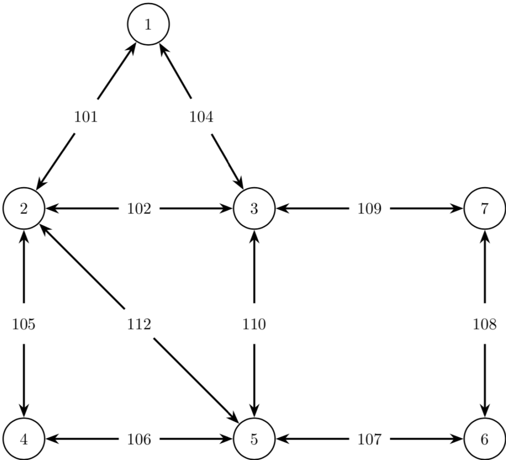


Figure 4.11: A generic Graph

Intuitive idea. To find an Eulerian circuit in the previous graph, we will use the Fleury algorithm explained above. Our base query constructs an edge, one of its nodes has $id = 101$, and marks the edge with property $order = 1$ and the other node (which is the current vertex) with property $order = 1$. Then a sub-query is made to ensure that the chosen edge is not a bridge, that is, that there exists a path between the two nodes that does not include the edge. In the recursive case, we are building edges and marking them with an order, based on the order of the current vertex, which we obtain by calculating the node that has the maximum value of the order property. In this case, as in the case of the minimum spanning tree, the order is calculated as the number of edges of the temporary graph in each iteration.

The next query shows how to find an Eulerian circuit in a graph using G-CORE syntax and the general recursive operator:

```
1 WITH RECURSIVE t as (
2 BASE (CONSTRUCT (a)-[:e1]->(x) SET x.mark:=1, e1.order=1
3       MATCH  (a)-[:e1]->(x) ON G
4       WHERE EXISTS ((CONSTRUCT ()
5                     MATCH (a)-/p/->(x)
6                     WHERE e1 is not in edges(p) )
7             AND
8                a.id=1)
9       LIMIT 1 )
10
11 RECURSIVE (
     CONSTRUCT t UNION (a)-[:e1]->(x) SET x.mark:= 1 + max(a.mark),
12                                         e1.order:= 1 + a.mark
13    MATCH (a) ON t
14          (x) ON G
15          (a)-[:e1]->(x) ON G-t
16    WHERE EXISTS (CONSTRUCT ()
17                  MATCH (a)-/p/->(x) ON G-t
18                  WHERE e2 is not in edges(p)
19                  OR
20                  Length(p)<=1 )
21          AND
22            a.mark=(SELECT max(mark) FROM t)
23    LIMIT 1)
24 OVER ORDER (SELECT COUNT(e1) FROM t)
```

In our example, the base case selects any edge that does not disconnect the graph G and we obtain:

G^1_{temp}

$H(G_1) =$ Number of edges in G_1

$a_1 = 1$

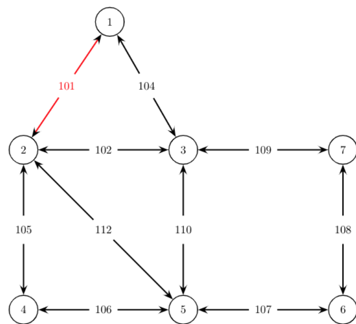


Figure 4.12: G^1 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1

Table 4.9: Base case Eulerian Circuit

Then, using the recursive part, we continue adding edges to the temporal graph such that the added edge is not in the temporal graph and does not disconnect the graph:

G^2_{temp}

$H(G_2) =$ Number of edges in G_2

$a_2 = 2$

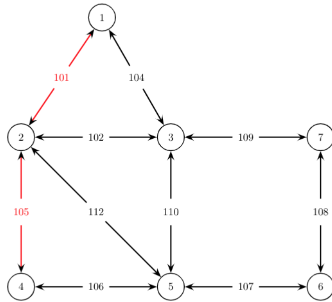


Figure 4.13: G^2 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2

Table 4.10: Recursive step 1 Eulerian Circuit

G^3_{temp}

$H(G_3) =$ Number of edges in G_3

$a_3 = 3$

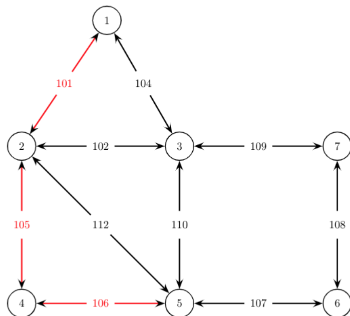


Figure 4.14: G^3 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3

Table 4.11: Recursive step 2 Eulerian Circuit

G^4_{temp}

$H(G_4) =$ Number of edges in G_4

$a_4 = 4$

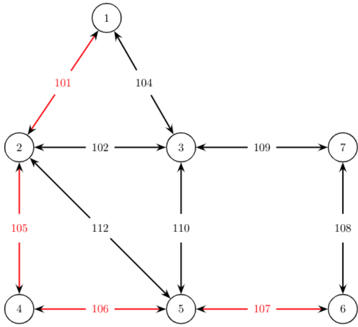


Figure 4.15: G^4 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4

Table 4.12: Recursive step 3 Eulerian Circuit

G^5_{temp}

$H(G_5) =$ Number of edges in G_5

$a_5 = 5$

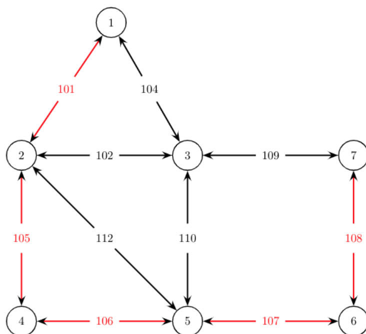


Figure 4.16: G^5 Eulerian Circuit

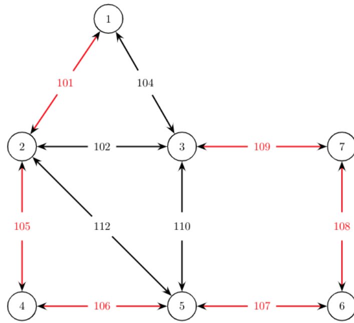
x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5

Table 4.13: Recursive step 4 Eulerian Circuit

G^6_{temp}

$H(G_6) =$ Number of edges in G_6

$a_6 = 6$



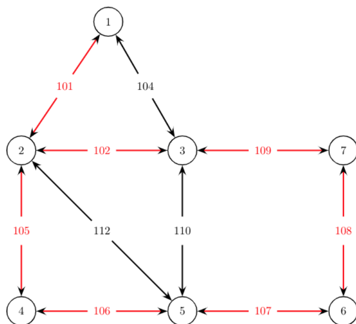
x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6

Figure 4.17: G^6 Eulerian Circuit

G^7_{temp}

$H(G_7) =$ Number of edges in G_7

$a_7 = 7$



x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6
3	2	102	7	7

Table 4.14: Recursive step 6 Eulerian Circuit

Figure 4.18: G^7 Eulerian Circuit

G^8_{temp}

$H(G_8) =$ Number of edges in G_8

$a_8 = 8$

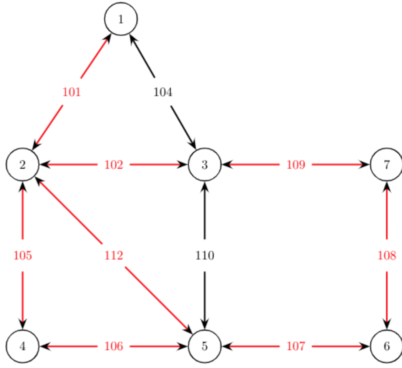


Figure 4.19: G^8 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6
3	2	102	7	7
2	5	112	8	8

Table 4.15: Recursive step 7 Eulerian Circuit

G^9_{temp}

$H(G_9) =$ Number of edges in G_9

$a_9 = 9$

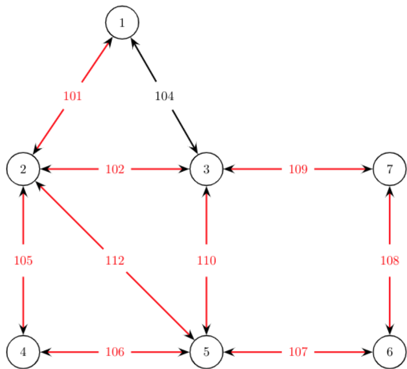


Figure 4.20: G^9 Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6
3	2	102	7	7
2	5	112	8	8
5	3	110	9	9

Table 4.16: Recursive step 8 Eulerian Circuit

G^{10}_{temp}

$H(G_{10})$ = Number of edges in G_{10}

$a_{10} = 10$

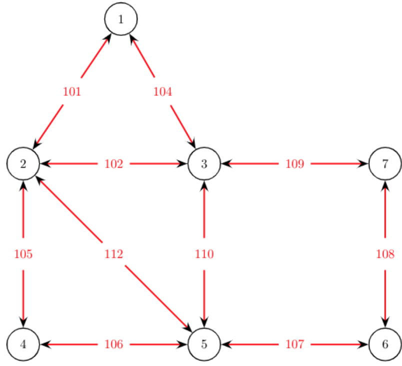


Figure 4.21: G^{10} Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6
3	2	102	7	7
2	5	112	8	8
5	3	110	9	9
3	1	104	10	10

Table 4.17: Recursive step 9 Eulerian Circuit

Finally we get the following table as final result and we can order each edge by its order property to find the Eulerian circuit:

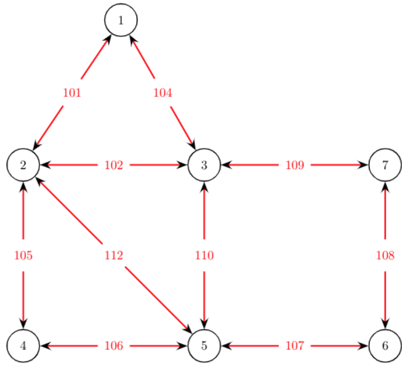


Figure 4.22: Final Graph Eulerian Circuit

x.id	y.id	e1.id	y.mark	e1.order
1	2	101	1	1
2	4	105	2	2
4	5	106	3	3
5	6	107	4	4
6	7	108	5	5
7	3	109	6	6
3	2	102	7	7
2	5	112	8	8
5	3	110	9	9
3	1	104	10	10

Table 4.18: Final graph Eulerian Circuit

This problem has not been studied with the recursive operator in SQL or SPARQL. However, as mentioned above it has been studied in [27] using regular expressions with memory, that is, given a graph it establishes pairs of nodes that are joined by a path that satisfies a regular expression c allowing some values to be able to be saved and compared, so using a REM formula it checks if there is a path between that passes through all the nodes where no edges are repeated. To simulate the Fleury algorithm we need to know if an edge is not a "bridge", that is to say that when removing it does not disconnect the graph, for this we use paths and the fact that the edge is not contained in the edges of those paths, a functionality that is not supported in SQL.

4.4 Maximum Matching

The last problem we would like to present to illustrate the potential of our approach for coding recursion, is finding a maximum matching in an undirected graph. A **matching** is a set of independent edges, that is, that have no vertices in common.

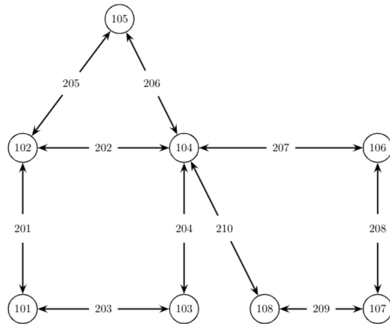
Definition 4.4.1. A matching M of G is said to be maximum if is a matching with maximum cardinality among all matchings of G.

This problem is important in at least two directions. First, it is a good example that indicates the difficulties one can find when having as recursion-order only the subset relation among partial solutions. And second, it points to operators that one would like to have in a query language (in this case symmetric difference) that are not currently present.

To begin with, let us consider the following lemma to find the maximum matching:

Lemma 1 (Berge's Lemma)**.** A matching M in a graph G is maximum (contains the largest possible number of edges) if and only if there is no augmenting path (a path that starts and ends on free (unmatched) vertices, and alternates between edges in and not in the matching) in M.

To fix these ideas, consider the following PPG G, and a table indicating the value of the property of each edge that represents whether or not the edge is in the matching. That is, n represents that the edge is not in the matching and m represents that the edge is in the matching. Initially we have no edges in the matching:



e.id	e.state
201	n
202	n
203	n
204	n
205	n
206	n
207	n
208	n
209	n
210	n

Table 4.19: Table representing state of edges in a labeled graph

The following query checks if there is an augmenting path in a PPG G (when marked with n and m as indicated above):

```
CONSTRUCT (x)-/p/->(y)
MATCH (x) - p<:n(mn)*> -> (y) ON G
WHERE
NOT EXISTS( CONSTRUCT (z)
```

```
                  MATCH (z)-[:e1]->(x)
                  WHERE e1.state='m')
AND NOT EXISTS(CONSTRUCT (c)
                  MATCH (c)-[:e2]->(y)
                  WHERE e2.state='m')
```

Note that the regular expression $n(mn)*$ ensures that the edges are alternated between belonging and not belonging to the matching. Then a sub-query is made that verifies that there are no incident edges to the nodes x, y that are in the matching, that is, they are free vertices.

Intuitive idea. Now that we have a simple algorithm to check if there is an augmenting path, the Blossom algorithm to find a maximum matching using Berge's Lemma works as follows. Mark an edge. Check if there is an augmenting path. If there is one, a well known result [38] shows that the symmetric difference of M with the already constructed match yields a matching having one more edge than M.

Thus the Blossom algorithm can be coded as follows:

Algorithm 3: Maximum Matching Algorithm

1 initialization;
2 $M \leftarrow \emptyset$
3 **while** *exists an augmenting path* P **do**
4 | $M = M \oplus P$
5 **end**
6 **return** M

The algorithm iterates while there is an augmenting path in the graph. For this we are going to "mark" the edges with an operator $\oplus$ that codes the symmetric difference between two sets (it does not exist in current G-CORE; we will denote it as OPLUS and will define it in the next chapter.) Finally, we finish the iterations when there is no longer an augmenting path in the graph. Note that in this case, the crucial trick is that we are iterating over the number of edges in the maximum matching (the "subset-order" of solutions does not work here because the set of nodes or edges involves could change).

Thus, the recursive query that we propose to solve this problem is:

```
1 WITH RECURSIVE t as(
2 BASE CASE (CONSTRUCT G, (a)-[:e1]->(x) SET e1.state:='m'
3            MATCH  (a)-[:e1]->(x) ON G
4            WHERE e1.id=207
5            )
6 RECURSIVE CASE (
7          CONSTRUCT t OPLUS (w)-/p/->(z)
8           MATCH (w)-p<:n(mn)*->(z) on t
9            WHERE
10                NOT EXISTS( CONSTRUCT (z)
```

```
11                    MATCH (z)-[:e1]-(w)
12                    WHERE e1.state='m')
13              AND
14                  NOT EXISTS(CONSTRUCT (c)
15                    MATCH (c)-[:e2]-(z)
16                    WHERE e2.state='m')
17             )
10 OVER ORDER (SELECT COUNT(e) FROM t WHERE e.state='m')
```

In this case we will use general recursion defined in the previous chapter, our recursion will be based on the number of edges that are marked with state m until there are no more edges to mark.

In our example: in the base case an edge of the graph G is chosen (in this case 207) and it is marked with state m:

G^1_{temp}

$H(G_1) =$ Number of edges in G_1 with state equals to "m"

$a_1 = 1$

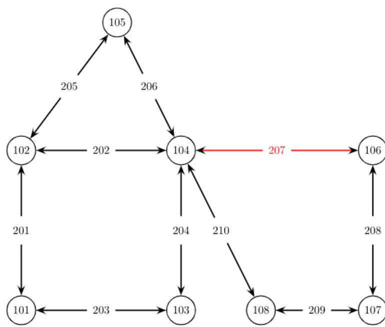


Figure 4.23: G^1 Maximum Matching

e.id	e.state
201	n
202	n
203	n
204	n
205	n
206	n
207	m
208	n
209	n
210	n

Table 4.20: Base case Maximum Matching

Then in the recursive case we find an augmenting path in the graph and we do the oplus operator and mark the new edges that belong to the matching. In this case an augmenting path is the path $103 - 204 - 104 - 207 - 106 - 107$, therefore the edges 204 and 208 are marked and the edge 207, originally in the matching, is unmarked:

G^2_{temp}

$H(G_2)$ = Number of edges in G_2 with state equals to "m"

$a_2 = 2$

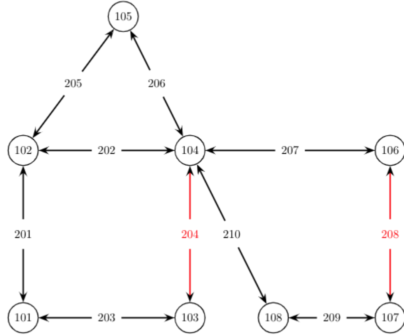


Figure 4.24: G^2 Maximum Matching

e.id	e.state
201	n
202	n
203	n
204	m
205	n
206	n
207	n
208	m
209	n
210	n

Table 4.21: Recursive step 1 Maximum Matching

Then, the augmenting path for this matching is searched again and the following matching is obtained:

G^3_{temp}

$H(G_3)$ = Number of edges in G_3 with state equals to "m"

$a_3 = 3$

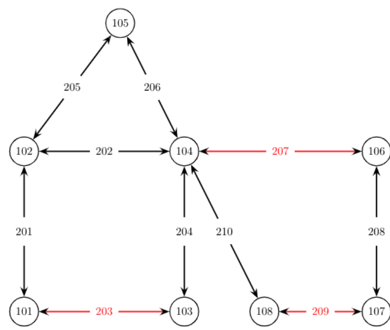


Figure 4.25: G^3 Maximum Matching

e.id	e.state
201	n
202	n
203	m
204	n
205	n
206	n
207	m
208	n
209	m
210	n

Table 4.22: Recursive step 3 Maximum Matching

Now we have 2 unmatched vertex 102 and 105, where the edge 205 is an augmenting path with respect to M.

G^4_{temp}

$H(G_4)$ = Number of edges in G_4 with state equals to "m"

$a_4 = 4$

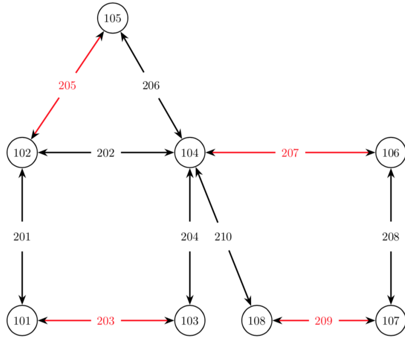


Figure 4.26: G^4 Maximum Matching

e.id	e.state
201	n
202	n
203	m
204	n
205	m
206	n
207	m
208	n
209	m
210	n

Table 4.23: Recursive step 3 Maximum Matching

Finally, there is no other augmenting path with this matching, therefore $a_4 = a_5 = 4$ so the edges with property state m are set that is the maximum matching.

This problem has not been studied with recursion neither in SPARQL, nor in G-CORE. The query about whether there is an increasing path with respect to the M matching could be expressed in other languages without recursion (e.g. Cypher). However, to find the result it is necessary to iterate until no other matching is found, which is achieved with a fixed point in the external order. In this query, paths are built which are not supported in SQL and SPARQL.

In this work we study how to express problems of graphs that can be solved in polynomial time [9] in G-CORE. In the previous section we showed how to express them in G-CORE with general recursion. In some cases, polynomial graph problems could be solved with the current machinery of G-CORE, particularly simulating the recursion via regular path expressions. We will show two such cases in what follows.

4.5 Connected Components

A connected component of an undirected graph is a maximal set of nodes that are connected to each other, that is, two nodes u, v are the same connected component if they are connected by a path.

Verifying if a graph is connected or has connected components can be done with the following query:

```
CONSTRUCT (x),
MATCH (x),(y) ON G,
WHERE NOT EXISTS(CONSTRUCT (x),
                 MATCH (x)-/<:Knows*>/-> (y) ON G)
```

The above query has a sub query that constructs two nodes that are not connected to each other in the graph. If this query is evaluated as true the graph is not connected, otherwise the graph is connected.

In the case the graph is not connected, one could ask how to find the connected components. To perform this query, we will construct a query on related components based on SQL connected components recursive solution[24]. Let us consider the following graph:

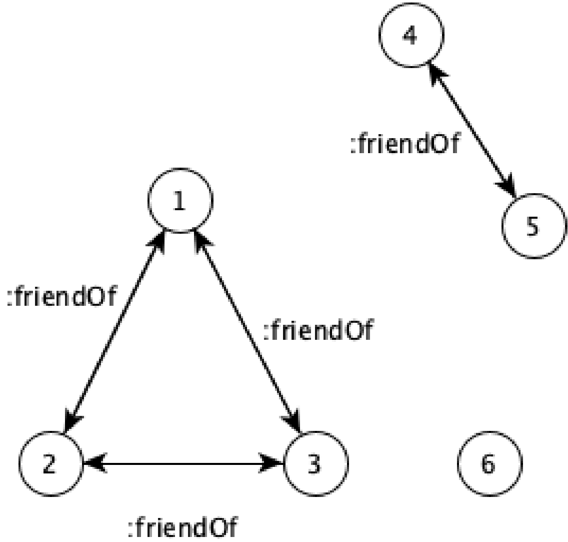


Figure 4.27: A PPG that represents a social network.

The graph in the figure has 3 connected components. To obtain them we first write the following query in G-CORE:

```
GRAPH H as(
          CONSTRUCT (n) SET n.comp=m.id,
          MATCH (n)-/<:label*>/->(m) ON G)
```

For simplicity we will see the result as a table:

n.id	n.comp
1	2
1	3
2	1
2	3
3	1
3	2
4	5
5	4

Table 4.24: Table that represents the nodes of a graph

Therefore, we have a table with all edges among the nodes. Now we need to add to each node their id as component because we need to consider the nodes that are not connected to any other node, therefore, we do the following G-CORE query:

```
GRAPH T as (
           CONSTRUCT (n) SET n.comp:=n.id,
           MATCH (n) ON G)
```

Finally we perform a union between the graphs H and T and make the following query:

```
SELECT x.name + MIN(x.comp) AS component
MATCH (x) ON (T UNION H)
GROUP BY x.name
```

Basically, we group all the ids and for each one of them we select the minimum id with to which it is connected. This query will give us as a result the following table:

n.id	component
1	1
2	1
3	1
4	4
5	4
6	6

Table 4.25: Final Result that represents de nodes that are connected

Therefore, in this case we have 3 connected components which are 1,4 and 6 and we can get

the connected component of each node.

4.6 Planarity Testing

Another important problem included in [9] that can be expressed in G-CORE without recursion is planarity testing.

Definition 4.6.1. A planar graph G is a graph that can be embedded in the plane.

Kuratowski's Theorem. A graph G is planar if only if it contains neither K_5 nor $K_{3,3}$ as topological minor.

Definition 4.6.2. A graph H is a topological minor of G if, when its edges are replaced by internally disjoint paths, the resulting graph occurs as a subgraph of G. Two paths from u to v in G are *internally disjoint* if they have no common internal vertex.

Given these results [39] and G-CORE's ability to manipulate and express paths, we show in the following example how to construct the K_5 pattern in a query.

```
CONSTRUCT (v1)
MATCH   (v1)-/p1/-(v2),
        (v1)-/p2/-(v3),
        (v1)-/p3/-(v4),
        (v1)-/p4/-(v5),
        (v2)-/p5/-(v3),
        (v2)-/p6/-(v4),
        (v2)-/p7/-(v5),
        (v3)-/p8/-(v4),
        (v3)-/p9/-(v5),
        (v4)-/p10/-(v5) ON G

WHERE
    nodes(p1)-{v1} INTER nodes(p2)-{v1}
    INTER nodes(p3)-{v1}
    INTER nodes(p4)-{v1} = empty
AND
    nodes(p1)-{v2} INTER nodes(p5)-{v2}
    INTER nodes(p6)-{v2}
    INTER nodes(p7)-{v2} = empty
AND
    nodes(p2)-{v3} INTER nodes(p5)-{v3}
    INTER nodes(p8)-{v3}
    INTER nodes(p9)-{v3} = empty
AND
    nodes(p10)-{v4} INTER nodes(p8)-{v4}
    INTER nodes(p3)-{v4}
    INTER nodes(p6)-{v4} = empty
```

```
AND
    nodes(p10)-{v5} INTER nodes(p9)-{v5}
    INTER nodes(p7)-{v5}
    INTER nodes(p4)-{v5} = empty
```

Therefore, the five vertices that form K_5 can be found in a pattern such that for each node the four incident paths are internally disjoint.

The same can be done for $K_{3,3}$. Then let P pattern to find $K_{5,5}$ and Q the pattern to find $K_{3,3}$. Then the query in G-CORE has the form:

```
CONSTRUCT (v)
MATCH (v) ON G
WHERE(
        CONSTRUCT (),
        MATCH P ON G,
         OR
        CONSTRUCT (),
        MATCH Q ON G
)
```

From a theoretical point of view, G-CORE and the recursive operator can express complex graph queries, however, since G-CORE is not yet fully implemented there are some limitations and aspects to consider in the implementation of this recursive operator that we will review. in the next chapter.

Chapter 5

Issues for the Implementation of recursion in G-CORE

In this chapter, we will address the possible implementation of the recursive operator defined in chapter 3. This will be mainly a theoretical discussion because currently there are no complete implementations of G-CORE. First we will discuss the current implementation of G-CORE. Next, we will discuss how difficult it is to add the recursive operator syntax to the current syntax. Finally we will discuss the operators that would be desirable to be added and some issues regarding some of the current operators.

5.1 Current Implementation of G-CORE

G-CORE is a language that has not yet a complete implementation. The most systematic current development of G-CORE we are aware of is the interpreter on Spark 2.2.0 that can be found in this Github projet [40]. These are the functionalities tested, implemented and designed in this project:

- Components: The nodes and edges are tested and the paths implemented (not tested). Each node and edge can have properties; however,checking that a path variable has properties is not supported yet.
- Properties : Currently the properties in nodes and edges are tested. However, it is not yet supported to have multi properties, that is, $prop = \{val1, val2, val3\}$ cannot be performed yet.
- Clauses: The CONSTRUCT, MATCH, WHERE clauses have been tested and the clauses have been implemented: CREATE GRAPH, DROP GRAPH, GRAPH VIEW and GRAPH AS. The PATH clause is designed but not implemented like the SELECT AND FROM clause that extends the language to return tables. The MATCH clause can currently be tested with one or more ON clauses which allows querying patterns from different graphs.
- Operators: Most of the operators are tested. Unfortunately, the most important operator in our example queries that allows us to make subqueries, the EXISTS operator, is not yet implemented. The graph operations, that is, union, intersection and difference

are not yet implemented.

For more details of G-CORE implementation one can consult [41].

5.2 Adding recursion to G-CORE

Syntax

In the case of the recursive operator syntax, we use the keywords `WITH RECURSIVE AS` based on the SQL syntax. In addition, we add the keywords `OVER ORDER` so that it is explicit that it defines the order that we are going to use in the query.

Implementation

For a possible implementation of a recursive operator in G-CORE, it must be considered that the linear version has already been made in the case of SPARQL [42]. In both cases we note that only one recursive operator is added and all the other components that already exist in the system are not modified. In the rec-SPARQL case, which is the most similar case to G-CORE, they only added an operator and did not modify anything in the database system. The G-CORE operators that we propose are based on the algorithm specified in section 3, consisting of:

- To have an empty temporal graph
- To perform a base query on the original graph
- To append the result to the temporal graph
- To perform a recursive query on original graph and / or temporal graph
- To append the result to the temporal graph
- For each iteration, to calculate the element of the abstract structure and thus calculate the order
- To repeat until the order in each iteration does not change

This operation does not modify the query engine because it only does the while statement until the element that represents how we are comparing the graphs is not modified. Finally, it is easy to add recursion as it is an algorithm that iterates and the parser needs to be changed depending on the system.

5.3 Problems with G-CORE operators

In section 4 we needed graph operators that are not in the design of G-CORE [8]. Among the most relevant are the following:

- **Difference of edges in the Eulerian circuit problem**: In the Eulerian circuit problem, in each iteration the algorithm must choose edges that are in the original graph but not in the temporary graph. Thus, an operator that subtracts graph edges is needed. Currently the G-CORE difference operator is restricted only to subtracting

nodes, so it seems natural to add the edge subtraction operator. This can be defined as follows.

Let us suppose that we have a PPG G_1 and edge e; we define the edge difference operator as $G_1 - e = (N, E, P, \rho, \delta, \lambda, \sigma)$ where $N = N_1$, $E = E_1 - \{e\}$, $P = \{p \in P_1 | edges(p) \subseteq E\}$, $\rho = \rho_1|_E$, $\delta = \delta_1|_P$, $\lambda = \lambda_1|_{(N \cup E \cup P)}$, $\sigma = \sigma_1|_{(N \cup E \cup P)xK}$.

The following simple example illustrates the features needed:

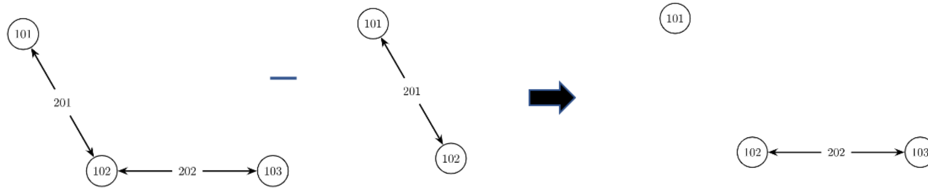


Figure 5.1: Difference of edges

The result we expect in this case is simply to delete the edge with identifier 201 from G_1. We need a new substract operation because the difference operator that currently exists first subtracts the nodes. Thus in this case, the node 101 and 102 would be deleted along with the edge 201, so that only the node 103 would remain.

- **Symmetric Difference in Maximum Matching Problem**: Another operator that we need to implement and we believe it is important is the one used in the maximum matching problem in section 4, which denoted "<>". This operator allows us to make a symmetric difference between the edge labels. That is, if we have a path p in a Graph G_1 with the set of edges $\{e_1, e_2, e_3\}$ with $\sigma(e_1, k) = n$, $\sigma(e_2, k) = m$ and $\sigma(e_3, k) = n$, the result of doing $G <> p$ would be the Graph G_1 with edges $\{e_1, e_2, e_3\}$ where $\sigma(e_1, k) = m$, $\sigma(e_2, k) = n$ and $\sigma(e_3, k) = m$.
- **SQL clauses**: At some point we defined an order on a query in SQL. Thus it would be desirable to implement in G-CORE the basic SQL clauses. As we saw in the previous section, the clauses that extend G-CORE to be able to query and manipulate tables are designed but not implemented.
- **If expressions**: If statements are not currently defined in G-CORE. There is a similar clause called Case and it allows to build rows based on a condition. This clause is implemented in Cypher and we use it in the previous examples (chapter 3 section 3.5). In this case we want to build a property of a node based on the value of the property of another node that is joined to it by an edge (match pattern). For the query we use a generic form of Case [1], and we believe that it can be easily added to G-CORE since Cypher and G-CORE are similar languages.

[1] https://neo4j.com/docs/cypher-manual/current/syntax/expressions/

Chapter 6

Conclusion and Future Work

Initially, this study aimed to investigate G-CORE's capacity to express graph problems. Specifically, we wanted to analyze graph problems that could be solved in polynomial time. Addressing this initial goal, we realized that recursion was fundamental to express most polynomial graph algorithms, and that expressing recursion in G-CORE was by itself a challenging research problem. Given the thesis timeline, we focused on solving the latter.

SQL and SPARQL recursion were our main inspiration for this work. The main difficulty we had in adapting the recursive operator to G-CORE was the Property graph data model, since recursion in SQL did not apply directly. SQL and SPARQL recursive operators both base their fixed point convergence on their data models cardinality. SQL's tables data model cardinality is given by the number of rows of the table, as in SPARQL's RDF graph data model it's given by the number of triples. In the case of G-CORE's Property Graphs, being a more complex data model with metadata as properties on its edges and nodes, translating the recursion from SQL or SPARQL was not straightforward.

In addition, in traditional recursion the temporal graph was built based on the union of the previous ones, which leaves out other types of queries that are also recursive. This was due to the fact that the results of the intermediate steps of the recursion, compared under a set theoretical order, were used as a "measure" of the advance of the recursion. So in our approach we redefine this general architecture, by separating the intermediate results from the order that guided the recursion. Indeed, we use an "external" (and simpler) order structure that in each iteration would reflect the "order" of the corresponding partial result data in that iteration. In our case, this measure can be chosen by the user and usually is a simple measure (number of nodes, depth of a tree, number of edges with some mark, etc.).

We realized that in any query language it is convenient to isolate and separate recursion in to iteration mechanism and the order/recursion mechanism. This offers more flexibility and simplicity to the programmer coding the problem, especially when dealing with graph models. This way, it's easier to code problems that have negation and queries that appear non-monotonous in traditional recursion but are monotonous in our recursive definition.

This separation can be designed and implemented very easily, although it requires attention

by the user to express the order correctly in the query. Small changes are necessary to implement it on a data base system; is includes having a temporal graph structure and an While algorithm (as discussed in chapter 3).

We also learned that there are operators that are not specified in G-CORE syntax and are necessary to express graph queries (for instance, edges substractions in chapter 5). We discovered that some operators are defined inconsistently (in a very restrictive way). For instance, the substraction operator which subtracts the sets of nodes from each graph and information is lost.

Although we believe it is important to add recursion to G-CORE, this language has valuable path functionalities, and for example, the planarity testing problem and connected components problems could be expressed using only this paths feature.

From our research, we conclude that further investigation is needed in the following topics:

- In depth studies on path properties to compare and integrate recursion and paths as first class citizen. This is relevant and is a novel problem since there were no languages with paths as first class citizens.
- Find out if all graph polynomial algorithms can be expressed in this new version of G-CORE plus recursion.
- To extend the recursion mechanisms defined for G-CORE to Cypher and PGQL, since G-CORE is based on these two languages.

As general conclusions regarding this thesis, we consider that by adding a recursive operator we were able to fulfill the objective of expressing graph algorithms in G-CORE.

Finally, we believe that G-CORE is a flexible enough language to implement the recursive operator and its possible implementation remains as future work.

Bibliography

[1] Renzo Angles. The Property Graph Database Model. In *AMW*, 2018.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50:1–40, 09 2017.

[3] Todd Green, Shan Shan Huang, Boon Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 2, 01 2012.

[4] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. Recursion in SPARQL. pages 19–35, Berlin, Heidelberg, 2015. Springer-Verlag.

[5] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF - W3C Recommendation, 2008. `https://www.w3.org/TR/rdf-sparqlquery/`.

[6] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.

[7] LBDC. `http://ldbcouncil.org/`.

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar Van, Alex Averbuch, Hassan Chaa, Irini Fundulaki, Alastair Green, Josep Lluis, Larriba Pey, Jan Michels, Raquel Pau, Arnau Prat, Tomer Sagi, and Yinglong Xia. G-CORE A Core for Future Graph query Languages Designed by the LDBC Graph query Language Task Force *. *In: SIGMOD*, 2017.

[9] Ludek Kucera. *Combinatorial Algorithms.* IOP Publishing Ltd. Adam Hilger Editorial, Praga, Checoslovaquía, 1990.

[10] The GQL Manifesto. `https://gql.today/`.

[11] Oskar Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a Property Graph Query Language. pages 1–6, 06 2016.

[12] Jim Melton and Alan R Simon. *SQL: 1999-Understanding Relational Language Com-*

ponents. Morgan Kaufmann, 2001.

[13] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular Queries on Graph Databases. *Theor. Comp. Sys.*, 61(1):31–83, July 2017.

[14] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A DSL forEasy and Efficient Graph Analysis. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 47, 04 2012.

[15] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, second edition, 2006.

[16] Richard Hull Serge Abiteboul and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[17] Ashok K Chandra and David Harel. Horn clause queries and generalizations. *The Journal of Logic Programming*, 2(1):1–15, 1985.

[18] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, page 1–10, New York, NY, USA, 1989. Association for Computing Machinery.

[19] Leonid Libkin. Expressive Power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, March 2003.

[20] Gabriel Aranda, Susana Nieva, Fernando Sáenz-Pérez, and Jaime Sánchez-Hernández. Formalizing a broader recursion coverage in SQL. pages 93–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] Carlos Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering*, 22, 2010.

[22] Devin W. Homan. Transitive Closure in SQL. `http://dwhoman.com/blog/sql-transitive-closure.html`.

[23] FusionBox. Graph Algorithms in a Database . `https://www.fusionbox.com/blog/detail/graph-algorithms-in-a-database-recursive-ctes-and-topological-sort-with-po` `620/`.

[24] Torsten Grust. Advanced SQL. `https://db.inf.uni-tuebingen.de/staticfiles/teaching/ss17/advanced-sql/slides/advanced-sql-05.pdf`.

[25] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language. *The 15th Symposium on Database Programming Languages*, pages 1–10, 2015.

[26] RDF Graph Literals and Named Graphs. `https://www.w3.org/2009/07/NamedGraph.html`. último acceso 2018-07-22.

[27] Pablo Barcelo, Gaelle Fontaine, and Anthony Lin. Expressive path queries on graph with data. *Logical Methods in Computer Science*, 11(4), Oct 2015.

[28] Pablo Barceló, Jorge Perez, and Juan Reutter. Relative expressiveness of nested regular expressions. *CEUR Workshop Proceedings*, 866, 01 2012.

[29] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems*, 37(4):1–46, 2012.

[30] Pablo Barceló. Querying Graph Databases. *In Principles of Database Systems (PODS)*, page 175–188, 2013.

[31] N Yakovets, P Godfrey, and J Gryz. Evaluation of SPARQL property paths via recursive SQL. *CEUR Workshop Proceedings*, 1087, 01 2013.

[32] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. NSPARQL: A Navigational Language for RDF. *Web Semant.*, 8(4):255–270, November 2010.

[33] Leonid Libkin, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. Trial: A navigational algebra for rdf triplestores. *ACM Trans. Database Syst.*, 43(1), March 2018.

[34] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

[35] Valentina Urzua and Claudio Gutiérrez. Linear recursion in G-CORE. In Aidan Hogan and Tova Milo, editors, *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*, volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[36] H N Gabow, Z Galil, T Spencer, and R E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, January 1986.

[37] *Journal de mathématiques élémentaires [et spéciales]*. Number v. 5. C. Delagrave, 1881.

[38] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[39] Saeed Akhoondian Amiri, Ali Golshani, Stephan Kreutzer, and Sebastian Siebertz. Vertex disjoint paths in upward planar graphs. In Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K. Vereshchagin, editors, *Computer Science - Theory and Applications*, pages 52–64, Cham, 2014. Springer International Publishing.

[40] LDBC. Implementation of G-CORE Graph Query Language on Spark. `https://github.com/ldbc/gcore-spark`.

[41] G-CORE implementation. `https://g-core.org/wp/`.

[42] SPARQL implementation. https://adriansoto.cl/RecSPARQL/index/.