EFFICIENT EVALUATION OF PATH QUERIES OVER GRAPH DATABASES

BENJAMÍN F. FARÍAS VALDÉS

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Advisor:
DOMAGOJ VRGOC, PH.D

Santiago de Chile, July 2024

© 2024, BENJAMÍN FARÍAS
EFFICIENT EVALUATION OF PATH QUERIES OVER GRAPH DATABASES

BENJAMÍN F. FARÍAS VALDÉS

Members of the Committee:
DOMAGOJ VRGOC, PH.D
JUAN REUTTER, PH.D
AIDAN HOGAN, PH.D
MATÍAS NEGRETE, PH.D

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Santiago de Chile, July 2024

© 2024, BENJAMÍN FARÍAS
With all my love to my family and my friends.
ACKNOWLEDGEMENTS

First of all I want to offer my deepest thanks to Domagoj, for accompanying and guiding me through this entire process, not only in the academic aspect, but also in the human one. This work would not have been possible without your supervision. I also want to express my gratitude to Carlos Rojas, for his continuous support when working with the MILLENNIUMDB team.

I am grateful to Cristián Riveros for inviting me to work in the MILLENNIUMDB project, as well as Adrián Soto and Juan Reutter for introducing me to the field of graph databases. I credit all of you with kick-starting my journey in this area of research.

To my family: I would not be at this stage of my life without the unconditional love and support that you give me. Felipe, Michelle, Emi and Elo, I am forever grateful for everything. To all other members of my big family, I treasure all the moments we have together, and thank you for always encouraging me to do my best.

To my close friends, both from university and from my school days, I appreciate all the fun experiences we share together, they have shaped me into the person I am today and serve as inspiration for my goals.

Finally, I cannot close this section without thanking myself, for never giving up on my objectives, always putting out all the effort I can provide in order to chase my dreams and live a happy and meaningful life.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv  
LIST OF TABLES vii  
LIST OF FIGURES viii  
RESUMEN ix  
ABSTRACT x  

1. INTRODUCTION 1  

2. BACKGROUND 7  
   2.1. Graphs and Paths 7  
   2.2. RPQs Over Graph Databases 8  
   2.3. Related Work 11  

3. ALGORITHMS FOR GQL SEMANTICS 13  
   3.1. Product Graph 13  
   3.2. ANY (SHORTEST) WALKS 14  
   3.3. ALL SHORTEST WALKS 19  
   3.4. ALL Restricted Paths 23  

4. IMPLEMENTATION DETAILS 29  
   4.1. Engine and Data Access 29  
   4.2. Pipelined Execution 31  
   4.3. Bidirectional Search 37  

5. EXPERIMENTAL EVALUATION 46  
   5.1. Experimental Setup 46  
   5.2. Pokec: Scaling the Path Length 49  
   5.3. Wikidata: The Effect of Big Graphs 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4. Diamond: Scaling the Number of Paths</td>
<td>53</td>
</tr>
<tr>
<td>5.5. Exploring Optimizations</td>
<td>56</td>
</tr>
<tr>
<td>5.6. Experimental Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>6. CONCLUSIONS AND FUTURE WORK</td>
<td>60</td>
</tr>
</tbody>
</table>

61
LIST OF TABLES

1.1 Support for path queries in graph database engines. .......................... 4
LIST OF FIGURES

1.1 A sample graph database. ........................................ 3

3.1 Visited after running Algorithm 1 in Example 3.1. .......... 18

3.2 Visited after running Algorithm 2 in Example 3.2. .......... 22

3.3 Visited after running Algorithm 3 in Example 3.3. .......... 26

4.1 A graph database and a CSR for the label $a$. .............. 30

4.2 Example of the bidirectional search strategy. ............... 38

5.1 Graph database with exponentially many paths. ............ 47

5.2 Runtimes over the Pokec dataset. ............................. 49

5.3 More runtimes over the Pokec dataset. ....................... 50

5.4 Runtimes for the Wikidata experiment. ....................... 52

5.5 WALK queries on the graph of Figure 5.1. .................... 54

5.6 TRAIL queries on the graph of Figure 5.1. .................... 55

5.7 Runtimes for the index experiment. .......................... 57

5.8 Runtimes for the bidirectional search experiment. .......... 58
Las consultas de caminos regulares (RPQs) son una característica central de todos los lenguajes y estándares de consulta de grafos modernos, como SPARQL, Cypher, SQL/PGQ y GQL. Si bien SPARQL devuelve puntos finales de RPQs, en Cypher, SQL/PGQ y GQL es también posible devolver los caminos completos. En esta tesis, presentamos el primer marco de trabajo para devolver caminos que coincidan con RPQs bajo los quince modos prescritos en los estándares SQL/PGQ y GQL. El núcleo de nuestro enfoque es la construcción del grafo de producto, combinada con una forma de representar de forma compacta un número potencialmente exponencial de resultados que pueden coincidir con una RPQ. A lo largo de la tesis, describimos cómo opera este enfoque a nivel conceptual y brindamos garantías de tiempo de ejecución para evaluar RPQs. También desarrollamos una implementación de referencia sobre un motor de procesamiento de grafos de código abierto, mostrando así cómo se puede integrar en cadenas de procesamiento de grafos existentes, y realizamos un análisis detallado sobre la ejecución de RPQs en conjuntos de datos relevantes para evaluar la utilidad de nuestros métodos en un escenario realista. En comparación con varios motores de consulta modernos, obtenemos mejoras de órdenes de magnitud y un rendimiento notablemente estable, incluso para clases de RPQs teóricamente intratables.

**Palabras Claves**: bases de datos de grafos, consultas de caminos regulares, grafos de propiedades, algoritmos de búsqueda.
ABSTRACT

Path queries are a central feature of all modern graph query languages and standards, such as SPARQL, Cypher, SQL/PGQ, and GQL. While SPARQL returns endpoints of path queries, it is possible in Cypher, SQL/PGQ, and GQL to return entire paths. In this thesis, we present the first framework for returning paths that match regular path queries under all fifteen modes prescribed in the SQL/PGQ and GQL standards. At the core of our approach is the product graph construction combined with a way to compactly represent a potentially exponential number of results that can match a path query. Throughout the thesis, we describe how this approach operates on a conceptual level and provide runtime guarantees for evaluating path queries. We also develop a reference implementation on top of an existing open-source graph processing engine, thus showing how it can be integrated into existing graph processing pipelines, and perform a detailed analysis of path querying over relevant data sets to gauge the usefulness of our methods in a real-world scenario. Compared to several modern graph query engines, we obtain order-of-magnitude speedups and remarkably stable performance, even for theoretically intractable classes of path queries.

Keywords: graph databases, regular path queries, property graphs, search algorithms.
1. INTRODUCTION

In current society, great technological advancements have made all sorts of information readily available and accessible throughout the world, which translates to large volumes of data that require reliable database systems to be able to store, manage, retrieve and analyze this data.

The most common type of databases seen today are known as *relational databases*, since they store their data inside pre-defined relationships that are organized as tables. This allows them to be very reliable and consistent in terms of data storage and management. These systems make use of a standardized query language, known as the Structured Query Language (SQL), as a consistent method of querying and interacting with their information. Despite the aforementioned advantages, relational models also come with important limitations. Specifically, they lack flexibility and scalability, as a product of their pre-defined structure. Due to the highly varied nature of data that is present in the world nowadays, relational databases are not well suited for the job in many situations.

This is why another type of database has gained a significant amount of popularity lately, namely, *graph databases* (Angles et al., 2018; Sakr et al., 2021), which this thesis focuses on. This type of system models data using a network-like structure: a collection of nodes connected via edges, where these edges represent semantic relationships between the data. The intuitive and conceptual nature of this approach makes it ideal for handling massive amounts of dynamic and highly interconnected data. Graph databases are currently used in a wide array of contexts, including areas such as Knowledge Graphs (Hogan et al., 2022), Biology (Jumper, 2021), and The Semantic Web (Angles et al., 2017).

With the proliferation of different database engines and query languages supporting graph-oriented features (Webber, 2012; T. Team, 2021; S. Team, 2021; A. N. Team, 2021; J. Team, 2021; Oracle, n.d.; ten Wolde, Singh, Szárnyas, & Boncz, 2023), there has been an increasing need for a standard language for expressing graph queries. This has led to several efforts such as SPARQL (Harris, Seaborne, & Prud’hommeaux, 2013),
a W3C (World Wide Web Consortium) standard for querying edge-labeled graphs, and SQL/PGQ (Deutsch et al., 2022), a recent ISO (International Organization for Standardization) initiative to extend SQL with pattern matching for property graphs. Currently, ISO is still working on the Graph Query Language (GQL), which is planned to become a standard for querying property graphs (Deutsch et al., 2022). Both SQL/PGQ and GQL are heavily influenced by the Cypher language (Francis et al., 2018).

Path queries are a core feature of all these graph query languages. In SPARQL these are supported through property paths, which are a variant of regular path queries (RPQs) that are well-studied in the literature (Calvanese, Giacomo, Lenzerini, & Vardi, 2002; Mendelzon & Wood, 1989; Baeza, 2013; Cruz, Mendelzon, & Wood, 1987). Intuitively, an RPQ is an expression (?x, regex, ?y), where regex is a regular expression, and ?x, ?y are variables. When evaluated over an edge-labeled graph $G$, it extracts all node pairs $(n_1, n_2)$ such that there is a path in $G$ linking $n_1$ to $n_2$, and whose edge labels form a word that matches regex.

However, RPQs only return endpoints of paths. In important applications such as money laundering detection or investigative journalism, where one is trying to understand the actual connections between entities, it is desirable to also see the full paths. This shortcoming of RPQs was recognized by the ISO standardization committee by making paths first-class citizens. Indeed, GQL and SQL/PGQ enrich RPQs with the ability to return the matching paths, filter on the type of such paths, limit the number of such paths, and provide many other features for path manipulation.

To illustrate these features, consider the graph in Figure 1.1. Here we have node identifiers (such as Joe or Rome), edge identifiers (such as $e_1$), and edge labels (for instance follows for $e_7$). A natural task would be to explore the influence network of, say, Joe, i.e., finding the people that Joe follows, then the people they follow, and so on, transitively traversing follows-edges. This can be written as an RPQ of the form $(Joe, follows^+, ?x)$,

---

1We consider a slightly simplified model of property graphs in this thesis to keep the discussion concise. Our results transfer verbatim to the real-world setting.
signaling that we wish to start at Joe, and traverse any non-zero number of follows-edges. In this case, the query returns all the people in the database. If we also wish to return paths witnessing these connections, we might run into some issues. Most notably, given the cycle formed by the edges $e_1$ and $e_2$, there is an infinite number of paths that start with Joe and return to Joe. To ensure the number of returned paths to be finite, GQL and SQL/PGQ (Deutsch et al., 2022) provide the so-called path modes.

Two of these path modes found in GQL are SIMPLE, which requires the paths to not repeat any nodes (apart from the first and last, which are allowed to be the same), and TRAIL, which requires the paths to not repeat any edges. If we choose to adhere to any of these modes, there is precisely one path that leaves Joe and comes back to Joe; namely, through the edge $e_2$ and then back via $e_1$. This effectively eliminates the infinite loop explained previously, given the restrictions provided by the mentioned path modes.

Another way of restricting the number of paths is by selecting only the shortest among them, or even non-deterministically choosing a single shortest path. In GQL and SQL/PGQ, the ANY SHORTEST mode will return a single shortest path for each unique pair of endpoints in the answer, whereas ALL SHORTEST will return all such shortest paths. Consider the RPQ $(Joe, \text{follows}^* \cdot \text{works}, ?x)$, which looks for the working place of people.
that Joe follows. In the graph of Figure 1.1, there are three shortest paths linking Joe to ENS Paris (traced by the edges $e_3 \rightarrow e_5 \rightarrow e_{11}$, $e_4 \rightarrow e_7 \rightarrow e_{10}$, and $e_3 \rightarrow e_6 \rightarrow e_{10}$, respectively), and we might wish to retrieve all of them.

While GQL and SQL/PGQ recognize the need to allow full regular path queries and return paths at the same time, the support for these features in modern graph database engines is lacking. Table 1.1 reviews path querying features in several leading graph systems. We see that, apart from SPARQL engines, which cannot return any paths, RPQs are only partially supported in property graph engines. More precisely, all engines under review only support regular expressions where the kleene star operator (“*”) is placed over a single edge label, or a disjunction of labels. NEO4J (Webber, 2012) is a bit more general, but does not support more complex expressions such as $(follows \cdot knows)^*$. In summary, no engine supports all regular expressions in RPQs. Similarly, when it comes to returning different types of paths, engines usually pick a single path type to support. The most powerful in this sense is NEBULA (Vesoft Inc/Nebula, 2023), which can return trails and acyclic paths, and can detect the presence of arbitrary paths (called “walks”), but cannot return walks. In general, none of the modern graph engines fully support RPQs with the path modes required by GQL and SQL/PGQ.

The lack of full support for RPQs is not surprising. In order to deal with them in a scalable fashion, one needs to find a way to cope with the potentially exponential number of paths that matches an RPQ in the data (Martens et al., 2023). In fact, most engines suffer from performance issues when returning paths, especially as the length of these paths grows. We illustrate this with a simple experiment on the Pokec dataset (1.6M

<table>
<thead>
<tr>
<th></th>
<th>RPQs</th>
<th>WALK</th>
<th>TRAIL</th>
<th>SIMPLE</th>
<th>ACYCLIC</th>
<th>SHORTEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAZEGRAPH (Thompson, Personick, &amp; Cutcher, 2014)</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>JENA (J. Team, 2021)</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>VIRTUOSO (Erling, 2012)</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>NEO4J (Webber, 2012)</td>
<td>partial</td>
<td>✔️</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>✔️</td>
</tr>
<tr>
<td>NEBULA (Vesoft Inc/Nebula, 2023)</td>
<td>partial</td>
<td>partial</td>
<td>✔️</td>
<td>–</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>KUZU (Jin, Feng, Chen, Liu, &amp; Salihoglu, 2023)</td>
<td>partial</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✔️</td>
</tr>
<tr>
<td>DUCKPGQ (ten Wolde et al., 2023)</td>
<td>partial</td>
<td>✔️</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✔️</td>
</tr>
<tr>
<td>MILLENIUMDB (Our Engine)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
nodes, 30M edges) from the SNAP graph collection (Leskovec & Krevl, 2014), which contains a real social network that is similar in structure to the one in Figure 1.1. In our experiment we select the node that is median in terms of centrality,² and explore its influence network while also returning up to 100,000 paths that witness the connections. We do so for paths of length 1 to 12, since the diameter of the Pokec network is capped at 11. The experimental results can be checked in Chapter 5 of this thesis, Figure 5.3. The figure shows that no tested engine performed well when scaling the path size, apart from MILLENNIUMDB (our own engine).

GQL and SQL/PGQ justify the need for efficient algorithms that can both support all RPQs and all prescribed path modes, which are currently lacking in leading graph systems, both in terms of performance and supported features. In this thesis we aim to address this problem, which is quite fundamental, since RPQ query answering is at the very core of path processing in the GQL and SQL/PGQ standards.

**Hypothesis and Contributions**

**Hypothesis.** Our working hypothesis is that the product graph construction, from the early theoretical literature on the subject, can be adapted to the context of returning paths specified by GQL path modes in an efficient manner, enabling competitive or even improved runtimes against state of the art graph database systems.

**Contributions.** Through this work, we provide the following contributions.

(i). We develop new algorithms for evaluating path queries, for all RPQs and all path modes prescribed by GQL and SQL/PGQ. Specifically, we build three main algorithms; one for the ANY (SHORTEST) WALK semantics, another one for the ALL SHORTEST WALK semantics, and the third one for the ALL (SHORTEST) restrictor semantics, where restrictor can be TRAIL, SIMPLE or ACYCLIC. From the ideas presented in these main

²Meaning that we computed the number of edges in which each node participates and selected one whose count is the median for the entire dataset.
algorithms, we can construct specific algorithms for all possible combinations that make up the path modes in GQL and SQL/PGQ.

(ii). We focus on making these algorithms as theoretically efficient as possible, and show that they all achieve output-linear delay. This means that our algorithms are able to output path query answers in an optimal manner, with no unnecessary steps when enumerating these paths.

(iii). We work inside MILLENIUMDB, a modern graph database query engine, and implement all our algorithms with a linear iterator interface that allows for pipelined evaluation. We do this with the purpose of returning answers to the user (or next operator in the query plan) as quickly as possible.

(iv). Compared to other systems that partially support RPQs with path modes or even to systems that cannot return paths, we show in an extensive experimental study that our implementations in MILLENIUMDB scale drastically better.

(v). We explore different optimizations that can further improve the performance of our algorithms, both at the data storage layer and at the logical level.

The rest of this thesis is organized as follows: In Chapter 2 we define graph databases and formalize the RPQ fragment of GQL and SQL/PGQ, alongside related work. Algorithms for the WALK, TRAIL and SIMPLE modes are all studied, from a theoretical standpoint, in Chapter 3. Chapter 4 explains all implementation details, including the pipeline versions for each algorithm and the optimizations that we experimented with. Experimental results are reviewed and discussed in Chapter 5. Finally, Chapter 6 provides concluding remarks and future lines of work.
2. BACKGROUND

2.1. Graphs and Paths

Graph Databases. Let Nodes be a set of node identifiers and Edges be a set of edge identifiers, with Nodes and Edges being disjoint. Additionally, let Lab be a set of labels. Following the research literature (Angles et al., 2017; Deutsch et al., 2022; Francis et al., 2023; Martens et al., 2023), we define graph databases as follows.

Definition 2.1. A graph database $G$ is a tuple $(V, E, \rho, \lambda)$, where:

- $V \subseteq \text{Nodes}$, is a finite set of nodes.
- $E \subseteq \text{Edges}$, is a finite set of edges.
- $\rho : E \rightarrow (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ means that $e$ is a directed edge going from $v_1$ to $v_2$.
- $\lambda : E \rightarrow \text{Lab}$, is a total function assigning a label to an edge.

Similarly as in (Martens et al., 2023), for our graph database model, we opt for a simplified version of property graphs (Deutsch et al., 2022), where we only consider nodes, edges and edge labels. Most importantly, we omit properties (with their associated values) that can be assigned to nodes and edges, as well as node labels. This is done since the type of queries considered in this work only require the features of this minimalist version of property graphs. However, all of our results transfer verbatim to the full version of property graphs. We also remark that our results apply directly to RDF graphs (Pérez, Arenas, & Gutiérrez, 2009) and edge-labeled graphs (Mendelzon & Wood, 1989; Baeza, 2013), which do not use explicit edge identifiers.

Paths. A path in a graph database $G = (V, E, \rho, \lambda)$ is a sequence $p = v_0e_1v_1e_2v_2\cdots e_nv_n$ where $n \geq 0$, $e_i \in E$, and $\rho(e_i) = (v_{i-1}, v_i)$ for $i = 1, \ldots, n$. If $p$ is a path in $G$, we write $\text{lab}(p)$ for the sequence of labels $\text{lab}(p) = \lambda(e_1)\cdots\lambda(e_n)$ occurring on the edges of $p$. We write $\text{src}(p)$ for the starting node $v_0$ of $p$, and $\text{tgt}(p)$ for the end node $v_n$ of $p$. The length
of a path $p$, denoted $\text{len}(p)$, is defined as the number $n$ of edges it uses. We will say that a path $p$ is a:

- **WALK**, for any $p$.\(^1\)
- **TRAIL**, if $p$ does not repeat any edges. That is, $e_i \neq e_j$ for all pairs of edges $e_i, e_j$ in $p$ with $i \neq j$.
- **ACYCLIC**, if $p$ does not repeat any nodes. That is, $v_i \neq v_j$ for all pairs of nodes $v_i, v_j$ in $p$ with $i \neq j$.
- **SIMPLE**, if $p$ does not repeat any nodes, except that possibly $\text{src}(p) = \text{tgt}(p)$. That is, $v_i \neq v_j$ for all pairs of nodes $v_i, v_j$ in $p$ with $(i, j) \in \{0, \ldots, n\}^2 - \{(0, n)\}$ and $i \neq j$.

Additionally, given a set of paths $P$ over a graph database $G$, we will say that $p \in P$ is a **SHORTEST** path in $P$, if it holds that $\text{len}(p) \leq \text{len}(p')$, for each $p' \in P$. We will use $\text{Paths}(G)$ to denote the (potentially infinite) set of all paths in a graph database $G$.

### 2.2. RPQs Over Graph Databases

**Regular Path Queries in GQL and SQL/PGQ.** We study *regular path queries* (RPQs), which form the basis of navigation in GQL and SQL/PGQ (Deutsch et al., 2022). For us, a regular path query will be an expression of the form:

$$\text{selector? restrictor \ (v, regex, ?x)}$$

where $v \in \text{Nodes}$ is a node, $\text{regex}$ is a regular expression, and $?x$ is a variable. Following GQL and SQL/PGQ (Deutsch et al., 2022), we use selectors and restrictors to specify which paths are to be returned by the RPQ. The grammar of these is as follows:

$$\text{restrictor : WALK — TRAIL — SIMPLE — ACYCLIC}$$

$$\text{selector : ANY — ANY SHORTEST — ALL SHORTEST}$$

---

\(^1\)The term *path* is used in the database literature to denote what is called a *walk* in graph theory. GQL and SQL/PGQ use WALK as a keyword for denoting any path.
Traditionally (Angles et al., 2017), the \((v, regex, ?x)\) part of an RPQ tells us that we wish to find all the nodes \(v'\) of our graph \(G\) for which there is a path \(p\) from \(v\) to \(v'\), such that \(\text{lab}(p)\) is a word in the language of the regular expression \(regex\).\(^2\) Since the set of all such paths can be infinite (Angles et al., 2017), restrictors allow us to specify which paths are considered valid, while selectors filter out results from a given set of valid paths. Next, we formally define the semantics of an RPQ.

Let \(G\) be a graph database and \(q\) an RPQ of the form:

\[
\text{restrictor} (v, regex, ?x)
\]

namely, we omit the optional selector part for now. We use the notation \(\text{Paths}(G, \text{restrictor})\) to denote the set of all paths in \(G\) that are valid according to \(\text{restrictor}\). For example, \(\text{Paths}(G, \text{TRAIL})\) is the set of all trails in \(G\). We then define the semantics of \(q\) over \(G\), denoted \([q]_G\), where \(q = \text{restrictor} (v, regex, ?x)\), as follows:

\[
[q]_G = \{ (p, v') \mid p \in \text{Paths}(G, \text{restrictor}), \\
\text{src}(p) = v, \text{tgt}(p) = v', \\
\text{lab}(p) \in \mathcal{L}(regex) \}
\]

Here \(\mathcal{L}(regex)\) denotes the language of the regular expression \(regex\). Intuitively, for an RPQ “\( \text{TRAIL} (v, regex, ?x) \)” the semantics return all pairs \((p, v')\) such that \(p\) is a TRAIL in our graph that connects \(v\) to \(v'\) and \(\text{lab}(p) \in \mathcal{L}(regex)\). The semantics of selectors are defined on a case-by-case basis. For this, we will use \(q\) to denote the selector-free RPQ \(q = \text{restrictor} (v, regex, ?x)\). We now have:

- \([\text{ANY restrictor} (v, regex, ?x)]_G\) returns, for each node \(v'\) reachable from \(v\) by a path \(p\) with \((p, v') \in [q]_G\), a single pair \((p, v') \in [q]_G\), chosen non-deterministically.

\(^2\)Notice that we assume that the starting node in an RPQ is fixed. RPQs can generally also have a variable in the place of \(v\), but for simplicity we consider the more limited case.
• \( [\text{ANY SHORTEST restrictor } (v, \text{regex}, ?x)] G \) returns, for a node \( v' \) reachable from \( v \) by a path \( p \) with \( (p, v') \in \llbracket q \rrbracket G \), a single pair \( (p, v') \in \llbracket q \rrbracket G \), where \( p \) is SHORTEST among all paths \( p' \) for which \( (p', v') \in \llbracket q \rrbracket G \).

• \( [\text{ALL SHORTEST restrictor } (v, \text{regex}, ?x)] G \) will return, for each \( v' \) reachable from \( v \) by a path \( p \) with \( (p, v') \in \llbracket q \rrbracket G \), the set of all pairs \( (p, v') \in \llbracket q \rrbracket G \) with \( p \) SHORTEST among paths \( p' \) for which \( (p', v') \in \llbracket q \rrbracket G \).

Intuitively, we can think of paths being grouped by \( v' \) before the \textit{selector} is applied. Notice that the semantics of ANY and ANY SHORTEST are non-deterministic when there are multiple (shortest) paths connecting some \( v' \) with the starting node \( v \). While the \textit{selector} is optional in our RPQ syntax, GQL and SQL/PGQ prohibit the WALK restrictor to be present without any selector attached to it, in order to ensure a finite result set. Therefore, in this thesis we will assume that queries using the WALK restrictor will always have an associated selector. This gives rise to 15 total combinations of query prefixes to specify the type of path(s) that are to be returned for each node reachable by the \((v, \text{regex}, ?x)\) part of the query.

**Output-Linear Delay.** Returning paths can result in a large number of outputs. Therefore, to measure the efficiency of our algorithms, we will use the paradigm of \textit{enumeration algorithms} (Bagan, 2006; Segoufin, 2013; Florenzano, Riveros, Ugarte, Vansummeren, & Vrgoc, 2020; Martens et al., 2023). Such algorithms work in two phases: a \textit{pre-processing phase}, which constructs a data structure allowing us to output the solutions; and the \textit{enumeration phase}, which enumerates these solutions \textit{without repetitions}. The efficiency of an enumeration algorithm is measured by the complexity of the pre-processing phase, and the delay between any two outputs produced during the enumeration phase. We will say that an enumeration algorithm works with \textit{output-linear delay}, when the delay is linear in the size of each output element. This means that the time needed to output a single path is linear in the number of nodes in the path, after which we immediately start to output the next path. Notice that this is optimal, in the sense that we will always have to at least write
down each element of the path when generating the output, making output-linear delay the best-case scenario for the enumeration phase of an algorithm.

**Regular Expressions and Automata.** We assume basic familiarity with regular expressions and finite state automata (Sakarovitch, 2009). If `regex` is a regular expression, we will denote by \( L(regex) \) the language of `regex`. We use \( A = (Q, \Sigma, \delta, q_0, F) \) to denote a non-deterministic finite automaton (NFA). Here \( Q \) is a set of states, \( \Sigma \) a finite alphabet of edge labels, \( \delta \) the transition relation over \( Q \times \Sigma \times Q \), \( q_0 \) the initial state, and \( F \) the set of final states, respectively. A `regex` can be converted into an equivalent NFA of size linear in \(|regex|\) (size of the regular expression) (Sakarovitch, 2009). Additionally, an NFA is called unambiguous if it has at most one accepting run for every word. In this thesis, we assume that the automaton has a single initial state and that no \( \varepsilon \)-transitions are present.

### 2.3. Related Work

Our work builds on top of (Martens et al., 2023), where a method to construct a compact representation of a set of paths witnessing an RPQ query answer under some of the GQL path modes is presented. In contrast, our work focuses on returning paths to the user. Both (Martens et al., 2023) and our approach leverage the product construction (Baeza, 2013) (see Chapter 3), but we extend the approach of (Martens et al., 2023) in several important ways. First, we present algorithms that run on top of the product graph and return paths witnessing RPQ query answers in a pipelined fashion, instead of computing the entire result set as in (Martens et al., 2023). Second, we support returning trails, simple paths, and acyclic paths, and we also implement the non-deterministic ANY variant of all GQL path modes, unlike (Martens et al., 2023). Third, our focus is on algorithms that are easily implementable in a real-world context, whereas (Martens et al., 2023) mostly focuses on theoretical guarantees. Other lines of work that relate to our proposal are described next.

**RPQ Evaluation.** Some of the most representative works in this area are (Yakovets, Godfrey, & Gryz, 2016; Gubichev, Bedathur, & Seufert, 2013; Gubichev, 2015; Fionda,
Pirrò, & Gutiérrez, 2015; Baier, Daroch, Reutter, & Vrgoč, 2017), where the focus is on finding nodes reachable by an RPQ-conforming path, and not on returning paths. Most of these works propose methods similar to Algorithm 1 (see Chapter 3). In (Gubichev & Neumann, 2011) the authors focus on retrieving paths, but not conforming to RPQ queries. An approach for finding top-$k$ shortest paths in RPQ answers is explored in (Savenkov, Mehmood, Umbrich, & Polleres, 2017), using a BFS-style search. Here the first $k$ paths discovered by BFS are retrieved, so some non-shortest paths might be returned, unlike in the GQL and SQL/PGQ semantics we use.

Interesting optimizations for the base BFS-style algorithm are presented in (Kaufmann, Then, Kemper, & Neumann, 2017; Then et al., 2014; Li, Zhao, Ntarmos, Cao, & Buneman, 2023), where vectorized execution of BFS with multiple starting points is explored. Regarding the simple path semantics, (Wadhwa, Prasad, Ranu, Bagchi, & Bedathur, 2019) proposes a sampling-based algorithm which is an efficient approximate answering technique, but will not necessarily find all answers.

There is a rich body of theoretical work on RPQ evaluation (Baeza, 2013). Interestingly, in (Casel & Schmid, 2021) Casel and Schmidt proved that, under the BMM conjecture, one requires $|G||q|$ time just to decide if a given node pair is the answer of an RPQ $q$ on $G$, which corresponds with the theoretical complexity of the product construction approach that we use in Chapter 3 to not only decide, but also return answers of RPQs.

**Reachability Indexes.** There is extensive work on speeding-up graph traversal algorithms via reachability indexes (Su, Zhu, Wei, & Yu, 2016; Yu & Cheng, 2010; Trißl & Leser, 2007; Yildirim, Chaoji, & Zaki, 2010; Seufert, Anand, Bedathur, & Weikum, 2013). Of these, (Gubichev et al., 2013) incorporated (Seufert et al., 2013) to allow finding nodes reachable by paths conforming to an RPQ. In terms of indexes developed for regular path queries, apart from some early theoretical work (Milo & Suciu, 1999), there is also a recent proposal (Zhang, Bonifati, Kapp, Haprian, & Lozi, 2022), that allows checking whether two nodes are connected by an RPQ-conforming path. In contrast, we design algorithms for retrieving paths over base data.
3. ALGORITHMS FOR GQL SEMANTICS

3.1. Product Graph

The basis for the algorithms that we designed lies in constructing a compact representation for all the paths in the query answer (Martens et al., 2023), using the product construction (Mendelzon & Wood, 1989; Sakarovitch, 2009). Given a graph database $G = (V, E, \rho, \lambda)$, and an expression of the form $q = (v, \text{regex}, ?x)$\footnote{Notice that the product construction does not depend on 
selectors or restrictors, since it was traditionally not used to return paths.}, the first step is converting the regular expression $\text{regex}$ into an equivalent non-deterministic finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. Then, we construct the product graph as shown next.

**Definition 3.1.** The product graph is a graph database $G_\times = (V_\times, E_\times, \rho_\times, \lambda_\times)$, where:

- $V_\times = V \times Q$
- $E_\times = \{(e, (q_1, a, q_2)) \in E \times \delta \mid \lambda(e) = a\}$
- $\rho_\times(e, d) = ((x, q_1), (y, q_2))$ if:
  - $d = (q_1, a, q_2)$
  - $\lambda(e) = a$
  - $\rho(e) = (x, y)$
- $\lambda_\times((e, d)) = \lambda(e)$.

Intuitively, the product graph is the graph database obtained by the cross product of the original graph $G$ and the automaton for regex $\mathcal{A}$. Each node of the form $(u, q)$ in $G_\times$ corresponds to a node $u$ in $G$ and, furthermore, each path $P$ of the form $(v, q_0), (v_1, q_1), \ldots, (v_n, q_n)$ in $G_\times$ corresponds to a path $p = v, v_1, \ldots, v_n$ in $G$ that (a) has the same length as $p$ and (b) brings the automaton from state $q_0$ to $q_n$. As such, when $q_n \in F$, then this path in $G$ matches $\mathcal{L}(\text{regex})$. Therefore all such $v$'s can be found by using any standard graph search algorithm (BFS/DFS) on $G_\times$ starting at the node $(v, q_0)$.
It is important for efficiency that we only construct the subgraph of $G_x$ that is needed for query answers. In (Martens et al., 2023) it is described how to efficiently construct the trim subgraph, which is the subgraph of $G_x$ that contains the nodes on a path from $(v, q_0)$ to some node of the form $(v', q_n)$ with $q_n \in F$. These nodes are exactly the ones we need to be able to answer any query under the WALK semantics (Martens et al., 2023).

We now take this a step further by only building $G_x$ on-the-fly as we are exploring the product graph. Additionally, we show how the product graph can be used to return paths.

### 3.2. ANY (SHORTEST) WALKS

We first treat the WALK restrictor combined with the ANY and ANY SHORTEST selectors, that is, queries of the form:

$$q = \text{ANY (SHORTEST)? WALK } (v, \text{regex}, ?x)$$  \hspace{1cm} (3.1)

The idea is that, given a graph database $G$ and a query $q$ as described above, we can perform a classical graph search algorithm such as BFS or DFS starting at the node $(v, q_0)$ of the product graph $G_x$, built from $G$ and the automaton $A$ for $\text{regex}$. Since both BFS and DFS also support reconstructing a single (shortest in the case of BFS) path to any reached node, we obtain the desired semantics for RPQs of the form (3.1). Query evaluation is presented in Algorithm 1.

The basic object we will be manipulating is a search state, which is simply a quadruple of the form $(n, q, \text{edge}, \text{prev})$, where $n$ is a node of $G$ we are currently exploring, $q$ is the state of $A$ in which we are currently located, while $\text{edge}$ is the edge of $G$ we used to reach $n$, and $\text{prev}$ is a pointer to the search state we used to reach $(n, q)$ in $G_x$. Intuitively, the $(n, q)$-part of the search state allows us to track the node of $G_x$ we are traversing, while

\[\text{Importantly, the trim subgraph of } G_x \text{ can be computed in optimal time, i.e., in the same time than it would take to decide if there exists a path labeled with some word in } L(\text{regex}) \text{ from } v \text{ to a given node } v', \text{ if the BMM conjecture holds (Casel & Schmid, 2021).}\]
Algorithm 1 Evaluation of \( \text{query} = \text{ANY} \) (SHORTEST)? \( \text{WALK} \) \((v, \text{regex}, ?x)\).

1: function \( \text{ANYWALK}(G, \text{query}) \)
2: \( \mathcal{A} \leftarrow \text{Automaton}(\text{regex}) \) \( \triangleright q_0 \text{ initial state, } F \text{ final states} \)
3: Open.init(); Visited.init(); ReachedFinal.init()
4: startState \( \leftarrow (v, q_0, \text{null}, \bot) \)
5: Visited.push(startState); Open.push(startState)
6: while Open \( \neq \emptyset \) do
7: \( \text{current} \leftarrow \text{Open}.\text{pop()} \) \( \triangleright \text{current} \leftarrow (n, q, \text{edge}, \text{prev}) \)
8: if \( q \in F \) and \( n \notin \text{ReachedFinal} \) then
9: \( \text{ReachedFinal}.\text{add}(n) \)
10: \( \text{Solutions}.\text{add}(\text{current}) \)
11: for each \((n', q', \text{edge}') \in \text{Neighbors(\text{current}, G, A)}\) do
12: if \( (n', q') \notin \text{Visited} \) then
13: \( \text{newState} \leftarrow (n', q', \text{edge}', \text{current}) \)
14: Visited.push(newState); Open.push(newState)

\( \text{edge} \), together with \( \text{prev} \), are necessary to reconstruct the path from \((v, q_0)\) that we used to reach \((n, q)\). The algorithm then requires the following data structures:

- **Open**, which is a queue (for BFS) or stack (for DFS) of search states, with their usual \text{push}() and \text{pop}() methods to add and remove elements, respectively.
- **Visited**, which is a dictionary of search states we have already visited in our traversal, maintained so that we do not end up in an infinite loop. We assume that \((n, q)\) can be used as a search key to check if some \((n, q, \text{edge}, \text{prev}) \in \text{Visited} \).
  We remark that \( \text{prev} \) always points to a state stored in \text{Visited}.
- **Solutions**, which is a set containing (pointers to) search states in \text{Visited} that encode a solution path to be returned; and
- **ReachedFinal**, a set containing nodes we already returned as query answers, in case we re-discover them via a different end state (recall that an NFA can have several end states).

The algorithm can explore the product \( G \times \mathcal{A} \) of \( G \) and \( \mathcal{A} \) using either BFS or DFS, starting from \((v, q_0)\). In line 4 we initialize a search state based on \((v, q_0)\), and add it to \text{Open} and \text{Visited}. The main loop of line 6 is the classical BFS/DFS algorithm that pops an element from \text{Open} (line 7), and starts exploring its neighbors in \( G \times \mathcal{A} \). When
exploring each state \((n, q, \text{edge}, \text{prev})\) in \(\text{Open}\), we scan all the transitions \((q, a, q')\) of \(\mathcal{A}\) that originate from \(q\), and look for neighbors of \(n\) in \(G\) reachable by an \(a\)-labeled edge (line 11). Here, writing \((n', q', \text{edge}') \in \text{Neighbors}((n, q, \text{edge}, \text{prev}), G, \mathcal{A})\) simply means that \(\rho(\text{edge}') = (n, n')\) in \(G\), and that \((q, \lambda(\text{edge}'), q')\) belongs to the transition relation of \(\mathcal{A}\). If the pair \((n', q')\) has not been visited yet, we add it to \(\text{Visited}\) and \(\text{Open}\), which allows it to be expanded later on in the algorithm (lines 12–14). When extracting a node from \(\text{Open}\) to be expanded, we first check if \(q\) is a final state, and if this is the case and \(n\) was not yet added to \(\text{ReachedFinal}\) (line 8), we found a new solution; meaning a \text{WALK} from \(v\) to \(n\) whose label is in the language of \text{regex}, so we add it to \(\text{Solutions}\) (line 10). This walk can then be reconstructed using the \text{prev} part of the search states stored in \(\text{Visited}\), via a standard linked list traversal.

Finally, we need to explain why the \(\text{ReachedFinal}\) set is used to store each previously undiscovered solution in line 9. Basically, since our automaton is non-deterministic and can have multiple final states, two things can happen:

(i) The automaton might be ambiguous, meaning that there could be two different runs of the automaton that accept the same word in \(\mathcal{L}(\text{regex})\). This, in turn, could result in the same path being returned twice, which is incorrect.

(ii) There could be two different paths \(p\) and \(p'\) in \(G\) that link \(v\) to some \(n\), and such that both \(\text{lab}(p) \in \mathcal{L}(\text{regex})\) and \(\text{lab}(p') \in \mathcal{L}(\text{regex})\), but the accepting runs of \(\mathcal{A}\) on these two words end up in different end states of \(\mathcal{A}\). This could result in multiple paths to \(n\) being returned, which is not allowed in the ANY \text{WALK} semantics.

Both of these problems are solved by using the \(\text{ReachedFinal}\) set, which stores a node the first time it is returned as a query answer (i.e. as an endpoint of a path reaching this node). Then, each time we try to return the same node again, we do so only if the node was not returned previously (line 8). This basically means that for each node that is a query answer, only a single path is returned, without any restrictions on the automaton used for modeling the query.
The described procedure continues until *Open* is empty, meaning that there are no more states to expand and all reachable nodes have been found with a WALK from the starting node $v$. Finally, the solutions are enumerated by traversing the *prev* pointers defined by each search state, in the same way as one would enumerate all the elements in a linked list. This step has linear complexity over the number of nodes in the path, meaning that it takes time proportional to each path to output it, thus achieving output-linear delay. It is also important to mention that, when choosing to use BFS, this algorithm has the added benefit of returning the SHORTEST WALK to every reachable node, since by design it always prioritizes the shortest paths when traversing the product graph. For the case of DFS, arbitrary WALKS are returned. To illustrate how Algorithm 1 works, we present an example.

**Example 3.1.** Consider again the graph $G$ in Figure 1.1, and let

$$q = \text{ANY SHORTEST WALK} \ (John, \ follows^+ \cdot \ lives, ?x).$$

Namely, we wish to find places where people that *John* recursively follows live. Looking at the graph in Figure 1.1, we see that *Rome* is such a place, and the shortest path reaching it starts with *John*, and loops back to him using the edges $e_1$ and $e_2$, before reaching *Rome* (via $e_8$), as required. To compute the answer, Algorithm 1 first converts the regular expression $follows^+ \cdot \ lives$ into the following automaton:

![Diagram of automaton]

To find shortest paths, we use the BFS version of Algorithm 1 and explore the product graph starting at $(John, q_0)$. The algorithm then explores the only reachable neighbor $(Joe, q_1)$, and continues by visiting $(John, q_1), (Paul, q_1)$ and $(Lily, q_1)$. When expanding $(John, q_1)$, the first solution, *Rome*, is found and recorded in *Solutions*. The algorithm continues by reaching $(Anne, q_1)$ and $(Jane, q_1)$ from $(Paul, q_1)$. When the $(Lily, q_1) node
is then expanded, it would try to reach \((Jane, q_1)\) again, which is blocked in line 12. Expanding \((Anne, q_1)\) would try to revisit Rome, but since this solution was already returned, we ignore it. The structure of Visited upon executing the algorithm is illustrated in Figure 3.1. Here we represent the pointer prev as an arrow to other search states in Visited, and annotate the arrow with the edge witnessing the connection. Notice that we can revisit a node of \(G\) (e.g. John), but not a node of \(G \times (e.g. (Jane, q_1))\). Since Solutions contains only \((Rome, q_F)\), we enumerate a single path traced by the edges \(e_1 \rightarrow e_2 \rightarrow e_8\).

\[\square\]

We can summarize the results about Algorithm 1 as follows:

**Theorem 3.1.** Let \(G\) be a graph database, and \(q\) the query:

\[
\text{ANY (SHORTEST)? WALK (v, regex, ?x).}
\]

If \(A\) is the automaton for regex, then Algorithm 1 correctly computes \([q]_G\) with \(O(|A| \cdot |G|)\) pre-processing time and output-linear delay.

**Proof.** The \(O(|A| \cdot |G|)\) factor is the cost of running the standard BFS/DFS algorithm on \(G \times\). Basically, we will visit each node \((n, q)\) in \(G \times\) at most twice, and each edge \((e, d)\) of \(G \times\) at most once. The correctness follows from the observation that each \(v'\) reachable from \(v\) by a path conforming to regex will be added to ReachedFinal once. The output-linear delay is explained from the fact that the enumeration of each path is performed as a linked list traversal, which is linear in the size of the path. \(\square\)
3.3. ALL SHORTEST WALKS

To fully cover the WALK semantics of RPQs, we next show how to evaluate queries of the form:

\[ q = \text{ALL SHORTEST WALK} \left( v, \text{regex}, ?x \right) \] (3.2)

For this, we will extend the BFS version of Algorithm 1 in order to support finding all shortest paths between a pair \( (v, v') \) of nodes, instead of a single one. The intuition here is: upon reaching \( v' \) from \( v \) by a path conforming to regex for the first time, the BFS algorithm will do so using a shortest path. The length of this path can then be recorded (together with \( v' \)). When a new path reaches the same, already visited node \( v' \), if its length is equal to the recorded length for \( v' \), then this path is also a valid answer to our query. We refer to Algorithm 2 for details.

As before, we use \( \mathcal{A} \) to denote the NFA for regex. This time, we will additionally assume that \( \mathcal{A} \) is unambiguous, to prevent duplicated solutions. The main difference to Algorithm 1 is in the search state structure. A search state is now a quadruple of the form \( (n, q, \text{depth}, \text{prevList}) \), where:

- \( n \) is a node of \( G \) and \( q \) a state of \( \mathcal{A} \);
- \( \text{depth} \) is the length of any shortest path to \( (n, q) \) from \( (v, q_0) \);
- \( \text{prevList} \) is a list of pointers to any previous search state that allows us to reach \( (n, q) \) via a shortest path.

In our design, \( \text{prevList} \) is a linked list, initialized as empty, and accepting sequential insertions of pairs \( \langle \text{searchState}, \text{edge} \rangle \) through the \( \text{add()} \) method. Intuitively, \( \text{prevList} \) will allow us to reconstruct all the shortest paths reaching a node, since there can be multiple ones. When adding a pair \( \langle \text{searchState}, \text{edge} \rangle \), we assume \( \text{searchState} \) to be a pointer to a previous search state, and \( \text{edge} \) will be used to reconstruct the path passing through the node in this previous search state. This procedure is then performed in a
Algorithm 2 Evaluation of \( \text{query} = \text{ALL SHORTEST WALK} \ (v, \text{regex}, ?x) \).

1: function \( \text{ALL SHORTEST WALK} (G, \text{query}) \)
2: \( A \leftarrow \text{UnambiguousAutomaton(\text{regex})} \) \quad \triangleright \text{\( q_0 \) initial state, \( F \) final states}
3: \( \text{Open.init(); Visited.init(); ReachedFinal.init()} \)
4: startState \( \leftarrow (v, q_0, 0, \bot) \)
5: Visited.push(startState); Open.push(startState)
6: while \( \text{Open} \neq \emptyset \) do
7: \( \text{current} \leftarrow \text{Open.pop()} \) \quad \triangleright \text{\( \text{current} \leftarrow (n, q, \text{depth}, \text{prevList}) \)}
8: if \( q \in F \) then
9: \( \quad \text{if } n \notin \text{ReachedFinal then} \)
10: \( \quad \quad \text{ReachedFinal.add}((n, \text{depth})) \)
11: \( \quad \quad \text{Solutions.add} \text{\( (\text{current}) \)} \)
12: \( \quad \text{else if } \text{ReachedFinal.get}(n).\text{depth} = \text{depth} \) then
13: \( \quad \quad \text{Solutions.add} \text{\( (\text{current}) \)} \)
14: \( \quad \text{for each } (n', q', \text{edge'}) \in \text{Neighbors(current, } G, A) \) do
15: \( \quad \quad \text{if } (n', q') \in \text{Visited then} \)
16: \( \quad \quad \quad (n', q', \text{depth'}, \text{prevList'}) \leftarrow \text{Visited.get}(n', q') \)
17: \( \quad \quad \quad \text{if } \text{depth} + 1 = \text{depth'} \) then \quad \triangleright \text{new shortest path to } (n', q')
18: \( \quad \quad \quad \quad \text{prevList'}.\text{add}((\text{current}, \text{edge'})) \)
19: \( \quad \text{else} \)
20: \( \quad \quad \text{prev} \leftarrow \text{prevList.init()} \)
21: \( \quad \quad \text{prev}.\text{add}((\text{current}, \text{edge'})) \)
22: \( \quad \quad \text{newState} \leftarrow (n', q', \text{depth} + 1, \text{prev}) \)
23: \( \quad \text{Visited.push(newState); Open.push(newState)} \)

recursive manner, with a standard backtracking algorithm, to enumerate all paths that connect \((v, q_0)\) with a specific \((n, q)\).

Similar to Algorithm 1, we have that \( \text{Visited} \) is a dictionary of search states, with the pair \((n, q)\) being the search key. Namely, there can be at most one tuple \((n, q, \text{depth}, \text{prevList})\) in \( \text{Visited} \) with the same pair \((n, q)\). By using \( \text{Visited.get}(n, q) \), we will obtain the unique search state having \( n \) and \( q \) as the first two elements, or a null pointer if no such search state exists. Finally, \( \text{ReachedFinal} \) is now a dictionary which contains pairs \((n, \text{depth})\), where \( n \) is a node reachable from \( v \) and \( \text{depth} \) is the length of the shortest path between \( v \) and \( n \), conforming to \( \text{regex} \). This dictionary allows \( n \) as a search key to obtain the associated \( \text{depth} \).
Algorithm 2 explores the product graph $G_x$ of $G$ and $A$ using BFS, since it needs to find shortest paths. Therefore, we assume Open to be a queue. The execution is very similar to Algorithm 1, with a few key differences. First, if a node $(n', q')$ of the product graph $G_x$ has already been visited with a shortest path (line 15), we do not directly discard the new path, but instead choose to keep it if and only if it is also shortest (line 17). In this case, the prevList for $(n', q')$ is extended by adding the new path (line 18). If a new pair $(n', q')$ is discovered for the first time, a fresh prevList is created (lines 20–23).

The second difference to Algorithm 1 is that ReachedFinal is now used to check if a group of new solutions are optimal for a reached node $n$ (lines 9–13). This is done since there can be multiple solutions for each node $n$ reachable from $v$, but not all of them are necessarily optimal, due to the automaton potentially having multiple final states. The following example illustrates what happens when there are multiple shortest paths between two nodes.

**Example 3.2.** Consider again the graph $G$ in Figure 1.1, and let

$$q = \text{ALL SHORTEST WALK} (Joe, \text{follows}^* \cdot \text{works}, ?x).$$

In the introductory chapter, we showed there are three paths in $[[q]]_G$, all three linking Joe to ENS Paris. The three paths are traced by the edges $e_3 \rightarrow e_5 \rightarrow e_{11}$, $e_4 \rightarrow e_7 \rightarrow e_{10}$, and $e_3 \rightarrow e_6 \rightarrow e_{10}$, respectively. To compute $[[q]]_G$, Algorithm 2 first converts the regular expression $\text{follows}^* \cdot \text{works}$ into the following automaton $A$:

```
follows

$q_0$ \hspace{1cm} \text{works} \hspace{1cm} q_F
```

Algorithm 2 then starts traversing the product graph $G_x$ by pushing the node $(Joe, q_0)$ to both Open and Visited (with depth = 0, and an empty prevList). When this search state is popped from the queue, its neighbors in $G_x$, namely $(Paul, q_0)$, $(Lily, q_0)$ and $(John, q_0)$.
are pushed to both Visited and Open, with depth = 1 and prevList pointing to \((Joe, q_0)\). The algorithm proceeds by visiting \((Anne, q_0)\) from \((Paul, q_0)\). Similarly, \((Jane, q_0)\) is visited from \((Paul, q_0)\). Something interesting happens in the next step when \((Lily, q_0)\) is the node being expanded to its neighbor \((Jane, q_0)\), which is already present in Visited. Here we trigger lines 15–18 of the algorithm for the first time, and update the prevList for \((Jane, q_0)\), instead of ignoring this path as Algorithm 1 does. When we try to explore neighbors of \((John, q_0)\) we revisit \((Joe, q_0)\), so lines 15–18 are triggered again. This time the depth test in line 17 fails (we already visited \(Joe\) with a path that has depth 0), so this path is abandoned. We then explore the node \((ENS, q_F)\) in \(G \times \) by traversing the neighbors of \((Anne, q_0)\). Finally, \((ENS, q_F)\) will be revisited as a neighbor of \((Jane, q_0)\) on a previously unexplored shortest path.

Figure 3.2 shows Visited upon executing the algorithm. Here we represent prevList as a series of arrows to other states in Visited, and only draw \((n, q, depth)\) in each node. For instance, \((Jane, q_0, 2)\) has two outgoing edges, representing two pointers in its prevList. The arrow is also annotated with the edge witnessing the connection (as stored in the search state).

Finally, given the final search state; namely the one with \((ENS, q_F, 3)\) in the first three components, we can correctly enumerate all three shortest paths connecting Joe to ENS Paris, using a standard backtracking algorithm.
For Algorithm 2 to work correctly, we crucially need $A$ to be unambiguous, since otherwise there could potentially be duplicated solutions. However, this does not pose a real problem, since the vast majority of RPQs in practice are, in fact, unambiguous (Bonifati, Martens, & Timm, 2020). Summing up, we obtain:

**Theorem 3.2.** Let $G$ be a graph database, and $q$ the query:

\[ \text{ALL SHORTEST WALK} (v, \text{regex}, ?x). \]

If the automaton $A$ for regex is unambiguous, then Algorithm 2 correctly computes $\langle q \rangle_G$ with $O(|A| \cdot |G|)$ pre-processing time and output-linear delay.

**Proof.** The $O(|A| \cdot |G|)$ factor stems from the fact that Algorithm 2 does precisely the same number of steps as Algorithm 1. Namely, Algorithm 2 is similar to running traditional BFS over $G \times$, since the nodes of $G \times$ we revisit (lines 15–18) do not get added to the queue $Open$ again. The output-linear delay is also achieved, since each path is enumerated by traversing its nodes only once, starting from the final search state and using a backtracking search algorithm in a recursive fashion. □

### 3.4. ALL Restricted Paths

In this section, we devise algorithms for supporting the rest of the restricted path types, namely, TRAIL, SIMPLE and ACYCLIC. It is well established in the research literature that even checking whether there is a single path between two nodes that conforms to a regular expression and is a simple path, acyclic path, or a trail is NP-complete (Baeza, 2013; Cruz et al., 1987; Martens, Niewerth, & Trautner, 2020; Bagan, Bonifati, & Groz, 2013). Therefore, we know that, in essence, the “best” known algorithm for finding such paths is a brute-force enumeration of all possible candidate paths in the product graph. Intuitively, we will be exploring an “unraveling” of the product graph. Our algorithms will follow this intuition and prune the search space whenever possible. Of course, in the
worst case all such algorithms will be exponential; however, as we later show, on real-world graphs, the number of paths will not be so large, and the pruning technique will ensure that all dead-ends are discarded. We begin by showing how to find all trails and simple/acyclic paths.

Returning All Paths. We start by dealing with queries of the form:

$$\text{q} = (\text{ALL SHORTEST})? \text{restrictor} (v, \text{regex}, ?x)$$

where restrictor is TRAIL, SIMPLE, or ACYCLIC. In Algorithm 3 we present a solution for computing the answer of q over a graph database G. Intuitively, when run over G, Algorithm 3 will first construct the automaton A for regex with the initial state q0. It will then start enumerating all paths in the product graph of G and A starting at (v, q0), and discarding ones that do not satisfy the restrictor of q. To ensure correctness, we will need the automaton to be unambiguous, and will keep the following auxiliary structures:

- **search state**, as a tuple \((n, q, \text{depth}, \text{edge}, \text{prev})\), where \(n\) is a node, \(q\) an automaton state, \(\text{depth}\) the length of a path reaching \((n, q)\) in \(G \times A\), \(\text{edge}\) an edge used to reach the node \(n\); and \(\text{prev}\) a pointer to another search state stored in Visited.
- **Visited**, as a set storing already explored search states.
- **Solutions**, which is a set containing (pointers to) search states in Visited that encode a solution path to be returned.
- **ReachedFinal**, which is a dictionary of pairs \((n, \text{depth})\), where \(n\) is a node reached in some query answer, and \(\text{depth}\) the length of the shortest path to this node. Here \(n\) is the dictionary key, so ReachedFinal.get(\(n\)) returns the pair \((n, \text{depth})\).

Algorithm 3 explores the product graph by enumerating all paths starting at \((v, q_0)\). For this, we can use a breadth-first (Open is a queue), or a depth-first (Open is a stack) strategy. In the case of the ALL SHORTEST selector, a queue needs to be used. The execution is very similar to Algorithm 1, but Visited is not used to discard solutions. Instead, every time a state from Open is expanded (lines 18–19), we check if the resulting path satisfies
Algorithm 3 Evaluation of $\text{query} = (\text{ALL SHORTEST})? \text{restrictor} (v, \text{regex}, x)$. Here \text{restrictor} is a string and ALL SHORTEST is a boolean value.

1: function \text{ALLRESTRICTEDPATHS}(G, \text{query})
2:   $\mathcal{A} \leftarrow \text{UnambiguosAutomaton}(\text{regex})$ \hspace{40pt} $\triangleright q_0$ initial state, $F$ final states
3:   \text{Open.init(); Visited.init(); ReachedFinal.init()}
4:   \text{startState} \leftarrow (v, q_0, 0, \text{null}, \bot)
5:   \text{Visited.push(startState); Open.push(startState)}
6: while $\text{Open} \neq \emptyset$
7:     current $\leftarrow \text{Open.pop()}$ \hspace{20pt} $\triangleright$ current $\leftarrow (n, q, \text{depth}, \text{edge}, \text{prev})$
8:     if $q \in F$ then
9:         if $\text{ALL SHORTEST} = \text{False}$ then
10:            \text{Solutions.add(current)}
11:        else if $n \notin \text{ReachedFinal}$ then
12:            \text{ReachedFinal.add}($\langle n, \text{depth} \rangle$)
13:            \text{Solutions.add(current)}
14:        else
15:            \text{optimal} $\leftarrow \text{ReachedFinal.get}(n).\text{depth}$
16:            if $\text{depth} = \text{optimal}$ then
17:                \text{Solutions.add(current)}
18:     \text{for each} $(n', q', \text{edge}') \in \text{Neighbors(current, G, A)}$ do
19:         if $\text{isValid}(\text{current}, (n', q', \text{edge}'), \text{restrictor})$ then
20:             \text{newState} $\leftarrow (n', q', \text{depth} + 1, \text{edge}', \text{current})$
21:             \text{Visited.push(newState); Open.push(newState)}

the \text{restrictor} of $q$ via the $\text{isValid}$ function. The function traverses the path defined by the search states stored in $\text{Visited}$ recursively. Here we are checking whether the path in the original graph $G$ satisfies the \text{restrictor}, and not the path in the product graph. If the explored neighbor allows to extend the current path according to the \text{restrictor}, we add the new search state to $\text{Visited}$ and $\text{Open}$ (lines 20–21).

If we reach a final state of the automaton (line 8), we record the solution. If the ALL SHORTEST selector is not present, we simply add the newly found solution (lines 9–10). In the presence of the selector, we need to make sure to add only shortest paths to the solution set. The dictionary $\text{ReachedFinal}$ is used to track discovered nodes, and stores the length of the shortest path to each of them. If the node is seen for the first time, the dictionary is updated, and a new solution added (lines 11–13). Upon discovering the same node again, a new solution is added only if it is shortest (lines 14–17). Once all paths have
been explored ($\text{Open} = \emptyset$), we can enumerate the solutions with the same procedure as in Algorithm 1, but now using the extended search states. We now illustrate the execution of this algorithm with a brief example.

**Example 3.3.** Consider again the graph $G$ in Figure 1.1, and

$$q = \text{SIMPLE} \ (\text{John}, \text{follows}^+ \cdot \text{lives}, ?x).$$

Namely, we wish to use the same regular pattern as in Example 3.1, but now allowing only simple paths. The same automaton as in Example 3.1 is used in Algorithm 3. The algorithm will start by visiting $(\text{John}, q_0)$, followed by $(\text{Joe}, q_1)$. After this we will visit $(\text{John}, q_1)$, $(\text{Paul}, q_1)$ and $(\text{Lily}, q_1)$. In the next step we will try to expand $(\text{John}, q_1)$, but we detect that this leads to a path which is not simple. We will continue exploring neighbors, building the Visited structure depicted in Figure 3.3. In Figure 3.3 we use the same notion for $\text{prev}$ pointers as in previous examples. For brevity, we do not show $\text{depth}$, but this is simply the length of the path needed to reach $(\text{John}, q_0)$ in Figure 3.3. We remark that the node $(\text{Jane}, q_1)$ appears twice since it will be present in the search state $(\text{Jane}, q_1, 3, e_6, \text{prev})$ and in $(\text{Jane}, q_1, 3, e_7, \text{prev}')$.

Just like with Algorithm 2, the correctness of this algorithm crucially depends on the fact that $\mathcal{A}$ is unambiguous, since otherwise we could record the same solution multiple times. Termination is assured by the fact that eventually all paths that are valid according to the restrictor will be explored, and no new search states will be added to $\text{Open}$.
Unfortunately, since we will potentially enumerate all the paths in the product graph, the complexity is exponential. More precisely:

**Theorem 3.3.** Let $G$ be a graph database, and $q$ the query:

$$(\text{ALL SHORTEST})? \text{ restrictor } (v, \text{regex}, ?x),$$

where restrictor is TRAIL, SIMPLE, or ACYCLIC. If the automaton $A$ for regex is unambiguous, then Algorithm 3 correctly computes $\mathbb{L}_q^G$ with $O(|A| \cdot |G|^{|G|})$ pre-processing time and output-linear delay.

**Proof.** The algorithm needs to iterate over all paths in the product graph. The stopping criterion is that the path needs to be SIMPLE, TRAIL or ACYCLIC in the original graph. Therefore, the longest possible path length will be $|G|$, and the total number of all paths in the worst case will then be $(|A| \cdot |G|^{|G|})$, since we are exploring paths of length $|G|$ over the product graph. As for the output-linear delay, it follows from using the same method for path enumeration as in Algorithm 1.

**ANY and ANY SHORTEST.** To treat queries of the form:

$$q = \text{ANY (SHORTEST)}? \text{ restrictor } (v, \text{regex}, ?x)$$

where restrictor is TRAIL, SIMPLE, or ACYCLIC, minimal changes to Algorithm 3 are required. Namely, for this case we would assume that ReachedFinal is a set instead of a dictionary, and that it only stores nodes $n$ reachable from $v$ by a path conforming to restrictor, and whose label belongs to $L(\text{regex})$. We would then replace lines 8–17 of Algorithm 3 with the following:

```plaintext
if $q \in F$ and $n \notin \text{ReachedFinal}$ then
    ReachedFinal.add($n$)
    Solutions.add(current)
```
Basically, when \( n \) is discovered as a solution for the first time, we record a path associated with it, and never return a path reaching \( n \) as a solution again. To make ANY SHORTEST work correctly, we need to use BFS (i.e. Open is a queue), while for ANY we can use either BFS or DFS. Unfortunately, due to the aforementioned results of (Cruz et al., 1987) stating that checking the existence of a simple path between a fixed pair of nodes is NP-complete, we cannot simplify the brute-force search of Algorithm 3. One interesting feature is that, due to the fact that ReachedFinal is now a set, and therefore for each solution node only a single path is returned, we no longer need the requirement that \( A \) be unambiguous. That is, we obtain:

**Theorem 3.4.** Let \( G \) be a graph database, and \( q \) the query:

\[
\text{ANY (SHORTEST)? restrictor } (v, \text{regex}, ?x),
\]

where restrictor is TRAIL, SIMPLE, or ACYCLIC. If \( A \) is the automaton for regex, then we can compute \( [q]_G \) with \( O(|A| \cdot |G|) \) pre-processing time and output-linear delay.

**Proof.** The same explanation from Theorem 3.3 applies here. \( \Box \)
4. IMPLEMENTATION DETAILS

4.1. Engine and Data Access

The algorithms of Chapter 3 were implemented inside of MILLENIUMDB (Vrgoč et al., 2021), a recent open-source graph database engine developed at IMFD. MILLENIUMDB provides the infrastructure necessary to process generic queries such as RPQs, and takes care of parsing, generation of execution plans, and data storage. By default, MILLENIUMDB stores graph data on disk using B+trees, and loads the necessary pages into a main memory buffer during query execution. MILLENIUMDB indexes several relations, however, our algorithms only use the one shown here:

\textbf{EDGES(NODEFROM, LABEL, NODETO, EDGEID)}

This relation allows us to find neighbors of a certain node reachable by an edge with a specific label, as used when calling the \textit{Neighbors} function in the algorithms discussed in Chapter 3. All B+trees in MILLENIUMDB are accessed using the standard linear iterator interface used in relational databases (Ramakrishnan & Gehrke, 2000). Additionally, all of the proposed algorithms can be easily extended with the ability to traverse edges backwards, as required, for instance, by C2RPQs (Calvanese et al., 2002) and SPARQL property paths (Harris et al., 2013). To support this, MILLENIUMDB also includes the inverse \textit{EDGES} relation, with the following schema:

\textbf{EDGES−(NODETO, LABEL, NODEFROM, EDGEID)}.

We can then easily extend all of the described algorithms to allow RPQs with backward edges, and use the \textit{EDGES−} table whenever looking for a neighbor accessed via a backward edge.

\textbf{Compressed Sparse Row}. In addition to B+trees, we also support data access via the \textit{Compressed Sparse Row (CSR)} representation (Buluç, Fineman, Frigo, Gilbert, & Leiserson, 2009), which is a popular in-memory index for fast access to edge data (ten Wolde et
Figure 4.1. A graph database and a CSR for the label $a$.

al., 2023). Basically, a CSR allows us to compress the standard matrix representation of a graph, which is of size $n \times n$ for a graph of $n$ vertices, to a more compact representation of roughly size $|E|$, where $E$ is the set of edges, while still allowing fast access to each node’s neighbors. For us, a CSR consists of three arrays:

- **src**, an array of source nodes, ordered by their identifier;
- **index**, an array of integers, where the number in position $i$ tells us where the neighbors of the $i$th source node start in the array **tgt**; and
- **tgt**, an array of target nodes.

As an illustration, Figure 4.1 shows a graph database (left), and its CSR for the edges labeled with $a$ (right). Notice that the node $n_3$ is never mentioned in the CSR since it has no $a$-labeled edges associated with it. The intuition in this example is that, for instance, $n_4$ is the second source node for $a$-labeled edges. Therefore, in the second position of the **index** array, we store the index of the **tgt** array where we start listing the neighbors of $n_4$ that can be reached by an $a$-labeled edge. In the example, these are $n_1$ and $n_2$.

Given that in regular path queries we only need to access neighbors reachable by specific labels, we support creating CSRs on-the-fly for these particular labels as needed by the query, which results in structures that can be much smaller than the CSR of the entire graph. We remark that our CSRs are created from the B+tree data residing on disk, and they support both backward and forward looking edges (each with their separate CSR). To avoid unnecessary overhead in the query runtime, we store our CSRs in cache after
construction, allowing for further use without having to build them again. In Chapter 5, we experiment with CSRs to determine their performance in a practical scenario.

4.2. Pipelined Execution

In MILLENNIUMDB, all the algorithms are implemented in a pipelined fashion using linear iterators. In particular, this means that each solution is returned to the user as soon as it is encountered. This requires each algorithm to be able to pause its execution to return a path, and then resume it when the next solution is expected. The main benefit of the pipelined execution is that paths can be encountered on demand, and the entire solution set need not be constructed in advance. All these operators are implemented as standard linear iterators, providing the following three methods:

(i) **BEGIN**(*G*, *query*), which initializes our query, and positions itself just before the first output tuple, without returning anything.

(ii) **NEXT**(*G*, *query*), which is called each time we wish to access the following tuple in the query answer. Notice that this implies that each algorithm is “paused” upon finding a solution, and then resumed when **NEXT**(*G*, *query*) is called again. This is done in order to combine several operators in a pipelined fashion.

(iii) **SEARCH**(*n*, *transition*, *G*), which allows us to search inside the database *G*, and locate all neighbors of node *n* that are connected via an edge with the same label as *transition*. This method returns an iterator object, capable of sequentially retrieving the necessary data (neighbor nodes and the edges that connect them to *n*) from a list of matching tuples that is stored on the database index of choice (in our case, B+trees and CSRs).

The structure of the automaton is modeled as an array, where each element in the array represents an automaton state, and contains another array with all the outgoing transitions from said state. Each of these transitions store the destination state in the automaton and the edge label for the connection. For the sake of simplicity, we omit implementations
Algorithm 4 Pipeline evaluation of \( query = \text{ANY SHORTEST WALK} (v, \text{regex}, ?x) \).

1: function \textsc{Begin}(G, query)
2: \( A \leftarrow \text{Automaton}(\text{regex}) \) \hfill \( \triangleright q_0 \text{ initial state, } F \text{ final states} \)
3: Open.init(); Visited.init(); ReachedFinal.init()
4: startState \( \leftarrow (v, q_0, \text{null, } \perp) \)
5: Visited.push(startState); Open.push(startState)
6: \( \text{iter} \leftarrow \text{null} \) \hfill \( \triangleright \text{index iterator for neighbors} \)
7: \( \text{firstNext} \leftarrow \text{True} \) \hfill \( \triangleright \text{first time calling NEXT} \)

8: function \textsc{Next}(G, query)
9: \( \text{if firstNext then} \) \hfill \( \triangleright \text{check if initial state is solution} \)
10: \( \text{firstNext} \leftarrow \text{False} \)
11: \( \text{if } q_0 \in F \text{ then} \)
12: ReachedFinal.add(\( v \))
13: return \( v \)
14: while Open \( \neq \emptyset \) do
15: current \( \leftarrow \text{Open.front()} \) \hfill \( \triangleright \text{current} \leftarrow (n, q, \text{edge, prev}) \)
16: reached \( \leftarrow \text{EXPANDANY}(\text{current}) \)
17: \( \text{if reached} \neq \text{null then} \) \hfill \( \triangleright \text{new solution found} \)
18: return \( \text{GETPATH}(\text{reached}) \)
19: \( \text{else} \) \hfill \( \triangleright \text{state was fully expanded} \)
20: \( \text{iter} \leftarrow \text{null} \)
21: Open.pop()
22: return \( \text{null} \) \hfill \( \triangleright \text{no more solutions} \)

based on DFS, since they are very similar to the ones that use BFS. The only differences between the two versions are the use of a stack (DFS) instead of a queue (BFS) for \text{Open}, and the need for DFS to store the iterator returned by \text{SEARCH}(n, \text{transition}, G) and the currently explored transition of the automaton inside each search state, given that, unlike the BFS strategy, each search state will not necessarily be fully expanded before deciding to explore a different one. Now we present a pipelined version for each of the algorithms explained in Chapter 3.

ANY SHORTEST WALKS. As shown in Algorithm 4, the idea is to search for query solutions using a BFS traversal strategy until one is found (line 17) or there are no more possible results (line 22). Most of the process occurs inside the \text{EXPANDANY} function (see Algorithm 5), which is constructed in a way that allows the operator to \textit{pause} the
Algorithm 5 EXPANDANY function for ANY SHORTEST WALK pipeline evaluation.

1: function EXPANDANY(state = (n, q, edge, prev))
2:     if iter = null then ▷ first time state is explored
3:         if |A.transitions(q)| = 0 then
4:             return null
5:     else
6:         transitionIdx ← 0
7:         transitionInfo ← A.transitions(q)[transitionIdx]
8:         iter ← SEARCH(n, transitionInfo, G) ▷ set the index iterator
9:     while transitionIdx < |A.transitions(q)| do
10:        transition ← A.transitions(q)[transitionIdx]
11:        q′ ← transition.to ▷ next automaton state
12:        while iter.next() ≠ null do ▷ iter ← (n′, edge′)
13:            if (n′, q′) ∉ Visited then
14:                newState ← (n′, q′, edge′, state)
15:                Visited.push(newState); Open.push(newState)
16:            if q′ ∈ F and n′ ∉ ReachedFinal then
17:                ReachedFinal.add(n′)
18:                return newState
19:        transitionIdx++ ▷ next transition
20:     if transitionIdx < |A.transitions(q)| then
21:        transitionInfo ← A.transitions(q)[transitionIdx]
22:        iter ← SEARCH(n, transitionInfo, G)
23: return null

execution when returning a solution, and resume the search when NEXT is called again. To remember the state of the search each time this happens, we store the current transition and iterator inside variables. Since the complete expansion of a single search state can now take multiple calls to NEXT, whenever we access the top state from Open, we use the front() method to keep the state inside the queue (line 15), and only apply pop() after said state has been fully expanded (line 21).

ALL SHORTEST WALKS. Algorithm 6 extends Algorithm 4 to handle the ALL SHORTEST WALK semantics, as seen in Chapter 3. One important detail is the fact that we now use an additional queue-like structure, ReachedSolutions, to store a batch of paths found via the GETNEWPATHS function (line 21). This allows us to return a group of results one by one, each time NEXT is called (lines 10–11). An important consideration here is that
Algorithm 6 Pipeline evaluation of query = ALL SHORTEST WALK \((v, \text{regex}, ?x)\).

1: function \(\text{BEGIN}(G, \text{query})\)
2: \(A \leftarrow \text{UnambiguousAutomaton(\text{regex})}\) \(\triangleright q_0 \text{ initial state, } F \text{ final states}\)
3: \(\text{Open.init(); Visited.init(); ReachedFinal.init()\)\)
4: \(\text{startState} \leftarrow (v, q_0, 0, \perp)\)
5: \(\text{Visited.push(startState); Open.push(startState)\)\)
6: \(\text{iter} \leftarrow \text{null}\) \(\triangleright \text{index iterator for neighbors}\)
7: \(\text{ReachedSolutions.init()}\)
8: \(\text{firstNext} \leftarrow \text{True}\) \(\triangleright \text{first time calling NEXT}\)

9: function \(\text{NEXT}(G, \text{query})\)
10: while \(\text{ReachedSolutions} \neq \emptyset\) do \(\triangleright \text{enumerate solutions}\)
11: \(\text{return } \text{ReachedSolutions.pop()}\)
12: if \(\text{firstNext}\) then \(\triangleright \text{check if initial state is solution}\)
13: \(\text{firstNext} \leftarrow \text{False}\)
14: if \(q_0 \in F\) then
15: \(\text{ReachedFinal.add}((v, 0))\)
16: \(\text{return } v\)
17: while \(\text{Open} \neq \emptyset\) do \(\triangleright \text{current} \leftarrow (n, q, \text{depth}, \text{prevList})\)
18: \(\text{current} \leftarrow \text{Open.front()}\)
19: \(\text{reached} \leftarrow \text{EXPANDALLSHORTEST(current)}\)
20: if \(\text{reached} \neq \text{null}\) then \(\triangleright \text{new solutions found}\)
21: \(\text{ReachedSolutions} \leftarrow \text{GETNEWPATHS(reached)}\)
22: \(\text{return } \text{ReachedSolutions.pop()}\)
23: else \(\triangleright \text{state was fully expanded}\)
24: \(\text{iter} \leftarrow \text{null}\)
25: \(\text{Open.pop()}\)
26: return \(\text{null}\) \(\triangleright \text{no more solutions}\)

for a given pair \((n, q)\), the \(\text{GETNEWPATHS}\) function could potentially be called multiple times to obtain solutions for \(n\), so it needs to take this into consideration and skip paths that have already been found and returned in previous iterations. This can be achieved by backtracking only from the last edge that reached the state \((n, q)\), ignoring the previous pointers in its \(\text{prevList}\) that were already used to reconstruct other shortest paths. The \text{EXPANDALLSHORTEST} function is displayed separately in Algorithm 7, and follows the same logic as \text{EXPANDANY}, but adapted to the ALL SHORTEST case, basically applying the same structure from Algorithm 2, where we search for solutions while encoding multiple shortest paths inside the \(\text{prevList}\) array of each search state.
```
Algorithm 7 EXPANDALLSHORTEST function for ALL SHORTEST WALK pipeline evaluation.

1: function EXPANDALLSHORTEST(state = (n, q, depth, prevList))
2:     if iter = null then ▷ first time state is explored
3:         if |A.transitions(q)| = 0 then
4:             return null
5:     else
6:         transitionIdx ← 0
7:         transitionInfo ← A.transitions(q)[transitionIdx]
8:         iter ← SEARCH(n, transitionInfo, G) ▷ set the index iterator
9:     while transitionIdx < |A.transitions(q)| do
10:         transition ← A.transitions(q)[transitionIdx]
11:         q' ← transition.to ▷ next automaton state
12:         while iter.next() ≠ null do ▷ iter ← (n', edge')
13:             if (n', q') ∈ Visited then
14:                 (n', q', depth', prevList') ← Visited.get(n', q')
15:                 if depth + 1 = depth' then
16:                     prevList'.add((state, edge'))
17:                     if q' ∈ F then
18:                         optimal ← ReachedFinal.get(n').depth
19:                         if depth + 1 = optimal then
20:                             return Visited.get(n', q')
21:     else
22:         prev ← prevList.init()
23:         prev.add((state, edge'))
24:         newState ← (n', q', depth + 1, prev)
25:         Visited.push(newState); Open.push(newState)
26:         if q' ∈ F then
27:             if n' ∉ ReachedFinal then
28:                 ReachedFinal.add((n', depth + 1))
29:                 return newState
30:         else
31:             optimal ← ReachedFinal.get(n').depth
32:             if depth + 1 = optimal then
33:                 return newState
34:         transitionIdx++ ▷ next transition
35:     if transitionIdx < |A.transitions(q)| then
36:         transitionInfo ← A.transitions(q)[transitionIdx]
37:         iter ← SEARCH(n, transitionInfo, G)
38: return null
```
Algorithm 8 Pipeline evaluation of \( \text{query} = \text{restrictor} (v, \text{regex}, ?x) \).

1: function \( \text{BEGIN}(G, \text{query}) \)
2: \( A \leftarrow \text{UnambiguousAutomaton}(\text{regex}) \) \( \triangleright q_0 \) initial state, \( F \) final states
3: Open.init(); Visited.init()
4: startState \( \leftarrow (v, q_0, \text{null}, \bot) \)
5: Visited.push(startState); Open.push(startState)
6: iter \( \leftarrow \text{null} \) \( \triangleright \) index iterator for neighbors
7: firstNext \( \leftarrow \text{True} \) \( \triangleright \) first time calling NEXT

8: function \( \text{NEXT}(G, \text{query}) \)
9: if firstNext then \( \triangleright \) check if initial state is solution
10: firstNext \( \leftarrow \text{False} \)
11: if \( q_0 \in F \) then
12: return \( v \)
13: while Open \( \neq \emptyset \) do
14: current \( \leftarrow \text{Open.front}() \) \( \triangleright \) current \( \leftarrow (n, q, \text{edge}, \text{prev}) \)
15: reached \( \leftarrow \text{EXPANDALL}(\text{current}) \)
16: if reached \( \neq \text{null} \) then \( \triangleright \) new solution found
17: return \( \text{GETPATH}(\text{reached}) \)
18: else \( \triangleright \) state was fully expanded
19: iter \( \leftarrow \text{null} \)
20: Open.pop()
21: return \( \text{null} \) \( \triangleright \) no more solutions

ALL Restricted Paths. In the case of \( \text{restrictor} \) based semantics, Algorithm 8 computes ALL paths that satisfy a specific \( \text{restrictor} \). The \textsc{ExpandAll} function is presented in Algorithm 9, and follows the same ideas as the previous pipeline implementations, with the addition of the \textsc{IsValid} function, which checks if the expanded path is allowed under the restricted semantics (line 13 of Algorithm 9). This is done by traversing the entire path and judging if it satisfies the conditions of the restrictor of choice (shown in Chapter 2).

An interesting observation to be made is the fact that \( \text{ReachedFinal} \) is not used by this algorithm, since we are looking for \( \text{ALL} \) possible restricted paths from \( v \) to each reachable node \( n \), without being concerned by the length of these paths. For the sake of brevity, we omit the optional \text{ALL SHORTEST} selector for this algorithm, since it follows the same ideas as shown in Algorithm 6.
Algorithm 9 EXPANDALL function for restrictor pipeline evaluation.

```plaintext
function EXPANDALL(state = (n, q, edge, prev))

if iter = null then ▷ first time state is explored
    if |A.transitions(q)| = 0 then
        return null
    else
        transitionIdx ← 0
        transitionInfo ← A.transitions(q)[transitionIdx]
        iter ← SEARCH(n, transitionInfo, G) ▷ set the index iterator
    while transitionIdx < |A.transitions(q)| do
        transition ← A.transitions(q)[transitionIdx]
        q' ← transition.to ▷ next automaton state
        while iter.next() ≠ null do ▷ iter ← (n', edge')
            if IS VALID(state, iter, restrictor) then
                newState ← (n', q', edge', state)
                Visited.push(newState); Open.push(newState)
                if q' ∈ F then
                    return newState
            transitionIdx++ ▷ next transition
        if transitionIdx < |A.transitions(q)| then
            transitionInfo ← A.transitions(q)[transitionIdx]
            iter ← SEARCH(n, transitionInfo, G)
    return null
```

4.3. Bidirectional Search

Throughout this thesis, we assumed that the RPQ part of our query takes the form (v, regex, ?x), where v is a node identifier and ?x is a variable. Namely, we were searching for all nodes reachable from a fixed point v. Based on this, it is straightforward to extend our algorithms to patterns of the form (v, regex, v'), where both endpoints of the path are known. That is, we can run any of the algorithms as described previously, and check whether v' is a query answer, returning only the solutions that reach v' and ignoring the rest.

However, when dealing with this type of query, we can leverage from the fact that both endpoints of the path are pre-defined, and explore a different approach. This new strategy consists of running the same BFS search used by all our algorithms, but instead
of just starting from the initial state \((v, q_0)\) and advancing forward with the automaton transitions, we now also start from all possible final states of the form \((v', q_F)\), where \(q_F \in F\), and advance backwards using the inverted automaton transitions. The intuition behind this is that if we run the search from both endpoints of the query and advance in opposite directions, we will eventually find all paths connecting \(v\) to \(v'\), exactly at those points where the search algorithms visit the same state from both directions of traversal. In other words, we are using a bidirectional version of multi-source BFS, which is now possible due to \(v'\) being a known node instead of a variable.

The potential advantage of this technique lies in the fact that each search will have to expand less states to find the solutions, since they will be “meeting in the middle” as they advance through the product graph (see Figure 4.2). As a result, each run of BFS will be able to ignore an exponential number of paths that would have never reached a solution, making it significantly more efficient than simply adapting the algorithms for the general case. Additionally, if the query has no valid solutions, then this bidirectional approach may be able to discard all states from one of the directions, stopping the algorithm early instead of continuing the search in the other direction in a futile effort to find non-existent solutions.

Looking to explore this search strategy, we designed a new version for some of the pipeline algorithms, for the case where the query is of the form \((v, \text{regex}, v')\). The results we found are analyzed later in Chapter 5.
Algorithm 10 Pipeline evaluation of \( \text{query} = \text{ANY SHORTEST WALK} (v, \text{regex}, v') \), using bidirectional search.

1: function \( \text{Begin}(G, \text{query}) \)
2: \( A \leftarrow \text{Automaton}(\text{regex}) \) \( \triangleright q_0 \) initial state, \( F \) final states
3: \( \text{Open.init(); Visited.init()} \)
4: \( \text{startState} \leftarrow (v, q_0, \text{null}, \bot, \text{forward}) \)
5: \( \text{Visited.push(startState); Open.push(startState)} \)
6: for each \( q_F \in F \) do \( \triangleright \) final states that expand backwards
7: \( \text{finalState} \leftarrow (v', q_F, \text{null}, \bot, \text{backward}) \)
8: \( \text{Visited.push(finalState); Open.push(finalState)} \)
9: \( \text{iter} \leftarrow \text{null} \)
10: \( \text{firstNext} \leftarrow \text{True} \)

11: function \( \text{Next}(G, \text{query}) \)
12: if \( \text{firstNext} \) then
13: \( \text{firstNext} \leftarrow \text{False} \)
14: if \( q_0 \in F \) and \( v = v' \) then
15: \( \text{Open.clear()} \)
16: return \( v \)
17: while \( \text{Open has forward and backward states} \) do
18: \( \text{current} \leftarrow \text{Open.front()} \) \( \triangleright \) \( \text{current} \leftarrow (n, q, \text{edge}, \text{prev}, \text{dir}) \)
19: \( \text{reached} \leftarrow \text{EXPANDANY2D} \text{(current)} \)
20: if \( \text{reached} \neq \text{null} \) then
21: \( \text{Open.clear()} \) \( \triangleright \) stop searching for solutions
22: return \( \text{GETPATH}(\text{reached}) \)
23: else
24: \( \text{iter} \leftarrow \text{null} \)
25: \( \text{Open.pop()} \)
26: return \( \text{null} \)

ANY SHORTEST WALKS. As displayed in Algorithm 10, the general structure is similar to the other pipeline algorithms, minus some important differences. To start with, each search state now holds a new \( \text{dir} \) attribute that indicates in which direction the state is being expanded (forward or backward), and we start the search by running BFS forwards from the initial state and backwards from the final states, as explained before (lines 4–8). The algorithm will then enter the main loop, stopping when any of the directions has no more search states to expand (line 17), since it would mean that no more states can provide a path to connect both directions anymore.
Algorithm 11 EXPANDANY2D function for ANY SHORTEST WALK pipeline evaluation, using bidirectional search.

1: function EXPANDANY2D(state = (n, q, edge, prev, dir))
2:   if dir == forward then ▷ transitions follow the current direction
3:       transitions ← A.transitions(q)
4:   else
5:       transitions ← A.reverseTransitions(q)
6:   if iter == null then
7:     if |transitions(q)| = 0 then
8:       return null
9:   else
10:      transitionIdx ← 0
11:      transitionInfo ← transitions(q)[transitionIdx]
12:      iter ← SEARCH(n, transitionInfo, G)
13:   while transitionIdx < |transitions(q)| do
14:      transition ← transitions(q)[transitionIdx]
15:      q′ ← transition.to
16:      while iter.next() ≠ null do ▷ iter ← (n′, edge′)
17:        if (n′, q′) ∉ Visited then
18:           newState ← (n′, q′, edge′, state, dir)
19:           Visited.push(newState); Open.push(newState)
20:        else if Visited.get(n′, q′).dir ≠ dir then ▷ converge to solution
21:           return MERGESTATES(state, edge′, Visited.get(n′, q′))
22:      transitionIdx++
23:   if transitionIdx < |transitions(q)| then
24:      transitionInfo ← transitions(q)[transitionIdx]
25:      iter ← SEARCH(n, transitionInfo, G)
26:   return null

The rest of the work is done inside the EXPANDANY2D function (Algorithm 11), which decides if the automaton transitions need to be inverted or not, according to the direction of the search (lines 2–5). When a state that has already been visited from a certain direction is then visited from the other direction, we have found a solution, and the path is obtained by simply merging the sub-paths from each direction into a single full path from v to v′ (lines 20–21). The rest of the logic follows the same steps as Algorithm 4. Notice how instead of checking for final states (states with q ∈ F), we now detect solutions solely by finding states where both directions meet, since this implies that one of the directions comes from the initial state and the other from one of the final states.
Algorithm 12 Pipeline evaluation of \( query = \text{ALL SHORTEST WALK} (v, regex, v') \), using bidirectional search.

1: function \textsc{Begin}(G, query)  
2: \( A \leftarrow \text{UnambiguousAutomaton}(regex) \) \(\triangleright q_0 \text{ initial state, } F \text{ final states} \)  
3: \text{Open.init(); Visited.init()}  
4: startState \( \leftarrow (v, q_0, 0, \bot, \text{forward}) \)  
5: \text{Visited.push(startState); Open.push(startState)}  
6: \textbf{for each } \( q_F \in F \textbf{ do} \) \(\triangleright \text{final states that expand backwards} \)  
7: \text{finalState} \( \leftarrow (v', q_F, 0, \bot, \text{backward}) \)  
8: \text{Visited.push(finalState); Open.push(finalState)}  
9: \text{iter} \leftarrow \text{null}  
10: \text{ReachedSolutions.init()}  
11: \text{firstNext} \leftarrow \text{True}  
12: \text{optimalDir} \leftarrow \text{null} \(\triangleright \text{direction of optimal solution} \)  

13: function \textsc{Next}(G, query)  
14: \textbf{while } \text{ReachedSolutions} \neq \emptyset \textbf{ do} \(\triangleright \text{enumerate solutions} \)  
15: \textbf{return } \text{ReachedSolutions.pop()}  
16: \textbf{if } \text{firstNext} \textbf{ then}  
17: \text{firstNext} \leftarrow \text{False}  
18: \textbf{if } q_0 \in F \textbf{ and } v = v' \textbf{ then}  
19: \text{Open.clear()}  
20: \textbf{return } v  
21: \textbf{while } \text{Open has forward and backward states do}  
22: \text{current} \leftarrow \text{Open.front()} \(\triangleright \text{current} \leftarrow (n, q, \text{depth}, \text{prevList}, \text{dir}) \)  
23: \textbf{if } \text{optimalDir} \neq \text{null and optimalDir} \neq \text{current.dir} \textbf{ then}  
24: \text{Open.clear()} \(\triangleright \text{no more optimal solutions} \)  
25: \textbf{return } \text{null}  
26: \text{reached} \leftarrow \text{EXPANDALLSHORTEST2D(current)}  
27: \textbf{if } \text{reached} \neq \text{null} \textbf{ then}  
28: \text{ReachedSolutions} \leftarrow \text{GETALLPATHS(reached)}  
29: \textbf{return } \text{ ReachedSolutions.pop()}  
30: \textbf{else}  
31: \text{iter} \leftarrow \text{null}  
32: \text{Open.pop()}  
33: \textbf{return } \text{null}  

\textbf{ALL SHORTEST WALKS.} Algorithms 12 and 13 show the procedure, which follows the same guidelines as with the \text{ANY SHORTEST WALK} semantics, but with the structure of Algorithm 2. There is an important factor to mention here; when the first shortest path is found by one of the directions of the search, we can infer that all shortest paths will also
Algorithm 13 EXPANDALLSHORTEST2D function for ALL SHORTEST WALK pipeline evaluation, using bidirectional search.

```
1: function EXPANDALLSHORTEST2D(state = (n, q, depth, prevList, dir))
2:     if dir = forward then ▷ transitions follow the current direction
3:         transitions ← A.transitions(q)
4:     else
5:         transitions ← A.reverseTransitions(q)
6:     if iter = null then
7:         if |transitions(q)| = 0 then
8:             return null
9:     else
10:         transitionIdx ← 0
11:         transitionInfo ← transitions(q)[transitionIdx]
12:         iter ← SEARCH(n, transitionInfo, G)
13:     while transitionIdx < |transitions(q)| do
14:         transition ← transitions(q)[transitionIdx]
15:         q' ← transition.to
16:         while iter.next() ≠ null do ▷ iter ← (n', edge')
17:             if (n', q') ∈ Visited then
18:                 (n', q', depth', prevList', dir') ← Visited.get(n', q')
19:                     if dir ≠ dir' then ▷ converge to solutions
20:                         if optimalDir = null then
21:                             optimalDir ← dir ▷ direction of all optimal solutions
22:                             return MERGESTATES(state, edge', Visited.get(n', q'))
23:                     else
24:                         if depth + 1 = depth' then
25:                             prevList'.add((state, edge'))
26:                     else
27:                         prev ← prevList.init()
28:                         prev.add((state, edge'))
29:                         newState ← (n', q', depth + 1, prev, dir)
30:                         Visited.push(newState); Open.push(newState)
31:         transitionIdx++
32:     if transitionIdx < |transitions(q)| then
33:         transitionInfo ← transitions(q)[transitionIdx]
34:         iter ← SEARCH(n, transitionInfo, G)
35:     return null
```

be found in this same direction, and this will happen before the algorithm explores the next state that expands in the opposite direction. This comes from the notion that, every time the search switches directions, the length of the paths to be found increases by one.
Moreover, this notion stems from a logical argument; if the path length did not increase when switching directions, then we could have found this same path when expanding the previous direction, which falls into a contradiction; hence the path length has to increase (and per the nature of BFS, this increment is of only one edge). Thanks to this realization, the algorithm can be stopped early when all shortest paths have already been found, using the `optimalDir` variable (lines 23–25 of Algorithm 12).

When obtaining available solutions with the `GETALLPATHS` function (line 28 of Algorithm 12), we essentially need to calculate the cross-product of all paths contained by the pair of states that are being merged from both directions, which results in all the combinations between paths that can be constructed through backtracking using the `prevList` attribute of each search state. This step will yield a batch of shortest paths that connect \( v \) to \( v' \), which can then be enumerated in the same way as the original algorithm.

**ANY SHORTEST Restricted Paths.** We cover the restricted version of the ANY SHORTEST semantics to gauge how the bidirectional search strategy fares when dealing with paths that are more complex. We define Algorithms 14 and 15, which are very similar to the bidirectional algorithm designed for the ANY SHORTEST WALK semantics, but with the `restrictor` considerations. The main difference here is that in the restricted semantics, `Visited` is only a set (see Chapter 3), so there can be many different states that share the same pair \( (n, q) \), unlike the algorithms shown for other semantics.

To deal with this complication, we now use a function called `GETCOMPATIBLESTATE`, after checking the validity of the path (lines 17–18 of Algorithm 15). This new function will first gather all the states from `Visited` that share the same pair \( (n, q) \) that is being reached through expanding the current state, and keep only those that are coming from the opposite direction. Then, from this group of `candidate` states, the function will check if any of them can be merged with the current path without invalidating the conditions imposed by the `restrictor`, in which case the first one that is capable of this will then be merged with the current state and the resulting path returned as the solution (lines 19–20 of Algorithm 15). If none of the candidates can return a valid solution, then we simply add
Algorithm 14 Pipeline evaluation of $query = \text{ANY SHORTEST restrictor} \ (v, regex, v')$, using bidirectional search.

```plaintext
1: function BEGIN($G, query$) 
2: \hspace{1em} $A \leftarrow \text{Automaton}(regex)$ \hspace{1em} $\triangleright q_0$ initial state, $F$ final states 
3: \hspace{1em} Open.init(); Visited.init() 
4: \hspace{1em} startState $\leftarrow (v, q_0, \text{null}, \bot, \text{forward})$ 
5: \hspace{1em} Visited.push(startState); Open.push(startState) 
6: \hspace{1em} for each $q_F \in F$ do \hspace{1em} $\triangleright$ final states that expand backwards 
7: \hspace{2em} finalState $\leftarrow (v', q_F, \text{null}, \bot, \text{backward})$ 
8: \hspace{2em} Visited.push(finalState); Open.push(finalState) 
9: \hspace{1em} iter $\leftarrow \text{null}$ 
10: \hspace{1em} firstNext $\leftarrow \text{True}$ 
11: function NEXT($G, query$) 
12: \hspace{1em} if firstNext then 
13: \hspace{2em} firstNext $\leftarrow \text{False}$ 
14: \hspace{2em} if $q_0 \in F$ and $v = v'$ then 
15: \hspace{3em} Open.clear() 
16: \hspace{3em} return $v$ 
17: \hspace{1em} while Open has forward and backward states do 
18: \hspace{2em} current $\leftarrow$ Open.front() \hspace{1em} $\triangleright$ current $\leftarrow (n, q, edge, prev, dir)$ 
19: \hspace{2em} reached $\leftarrow$ \text{EXPANDANYRESTRICTED2D}(current) 
20: \hspace{2em} if reached $\neq \text{null}$ then 
21: \hspace{3em} Open.clear() \hspace{1em} $\triangleright$ stop searching for solutions 
22: \hspace{3em} return \text{GETPATH}(reached) 
23: \hspace{2em} else 
24: \hspace{3em} iter $\leftarrow \text{null}$ 
25: \hspace{3em} Open.pop() 
26: \hspace{2em} return $\text{null}$ 
```

The reached state to Visited and Open, then resume the standard execution of the algorithm (lines 21–23 of Algorithm 15).
Algorithm 15 EXPAND\textsc{AnyRestricted2D} function for ANY SHORTEST restrictor pipeline evaluation, using bidirectional search.

\begin{algorithmic}[1]
\Function{EXPAND\textsc{AnyRestricted2D}}{(state = (n, q, edge, prev, dir))}
\If{\textit{dir} = \textit{forward}} \Comment{transitions follow the current direction}
\State transitions $\leftarrow \mathcal{A}.\text{transitions}(q)$
\Else
\State transitions $\leftarrow \mathcal{A}.\text{reverseTransitions}(q)$
\EndIf
\If{\textit{iter} = \text{null}}
\If{|transitions(q)| = 0}
\State \Return null
\Else
\State transitionIdx $\leftarrow$ 0
\State transitionInfo $\leftarrow$ transitions(q)[transitionIdx]
\State iter $\leftarrow$ SEARCH(n, transitionInfo, G)
\While{transitionIdx < |transitions(q)|}
\State transition $\leftarrow$ transitions(q)[transitionIdx]
\State $q'$ $\leftarrow$ transition.to
\While{iter.next() \neq \text{null}} \Comment{iter $\leftarrow (n', edge')$}
\If{\text{IS\textsc{Valid}}(state, iter, restrictor)}
\State compatible $\leftarrow$ GET\textsc{CompatibleState}(state, iter, restrictor)
\If{compatible \neq \text{null}} \Comment{converge to solution}
\State \Return MERGE\textsc{States}(state, edge', compatible)
\Else \Comment{no solution, keep expanding}
\State newState $\leftarrow (n', q', edge', \text{state}, \text{dir})$
\State Visited.push(newState); Open.push(newState)
\EndIf
\State transitionIdx++
\EndIf
\EndWhile
\EndWhile
\EndIf
\EndIf
\EndFunction
\end{algorithmic}
5. EXPERIMENTAL EVALUATION

5.1. Experimental Setup

We empirically evaluate the algorithms from Chapter 4 using MILLENNIUMDB and show that our approach scales on a broad range of real-world and synthetic data sets and queries. We perform three main sets of experiments:

- **Pokec**, which tests the effect of path length on performance;
- **Wikidata**, where we test the performance over a large real-world graph and user supplied queries; and
- **Diamond**, where we test the effect of having a large number of paths in the graph.

Next we describe each set of experiments in more detail.

(1) **Pokec.** This experiment uses the Pokec social network graph from SNAP (Leskovec & Krevl, 2014). Pokec is a Slovakian social network which records (directed) user connections, similar to the graph of Figure 1.1, but with a single type of edge label (we call it follows). The graph contains around 1.6 million nodes and 30 million edges. As described in the introduction, we fix a node that is median in terms of centrality, and traverse follows-labeled edges from this node. We explore paths of length 1 through \( k \), where \( k \) ranges from 1 to 12. Longer paths are uninteresting, since the graph’s diameter is 11. We pair these queries with the path modes described in Chapter 2. The idea behind this experiment is to test what happens with query performance as we seek longer paths in a real-world graph of intermediate size.

(2) **Wikidata.** Here, we want to check performance over a large real-world graph. For this we use Wikidata (Vrandecic & Krötzsch, 2014), and queries from its public SPARQL query log (Malyshev, Krötzsch, González, Gonsior, & Bielefeldt, 2018). Specifically, we use WDBench (Angles, Aranda, Hogan, Rojas, & Vrgoč, 2022a), a recently proposed Wikidata SPARQL benchmark. WDBench provides a curated version of the data set based on the truthy dump of Wikidata (Foundation, 2021), which is an edge-labeled graph with
364 million nodes and 1.257 billion edges, using more than 8,000 different edge labels. The data set is publicly available (Angles, Aranda, Hogan, Rojas, & Vrgoč, 2022b). WD-Bench provides multiple sets of queries extracted from the Wikidata’s public endpoint query log. We use the Paths query set, which contains 659 2RPQs patterns. From these, 592 have a fixed starting point or ending point (or both), while 67 have both endpoints as free variables. We note that these queries require general regular expressions which cannot be expressed in some of the tested systems. The 659 patterns are then used in our tests under the restrictor and selector options described in Chapter 2.

(3) Diamond. In our final experiment we test what happens when there is a large number of paths present in our graph. The database we use, taken from (Martens et al., 2023), is presented in Figure 5.1. The queries we consider look for paths between start and end using $a$-labeled edges. Notice that all such paths are, at the same time, shortest, trails and simple paths, and have length $2n$. Furthermore, there are $2^n$ such paths, while the graph only has $3n + 1$ nodes and $4n$ edges. We test our query with the path modes from Chapter 2, while scaling $n$ (and thus path length) from 1 to 40. While returning all these paths is unfeasible for any algorithm, we test whether a portion of them (100,000 in our experiments) can be retrieved efficiently.

Tested Systems. MILLENIUMDB supports both BFS and DFS traversal (when paths need not be shortest). We denote the two versions as MDB-BFS and MDB-DFS, respectively. When there is only one algorithm (e.g. for all shortest walks), we simply write MILLENIUMDB. All the versions assume data to be stored on disk and being buffered into main memory as required.
In order to compare with state of the art in the area, we selected six publicly available graph database systems that allow for benchmarking with no legal restrictions. These are:

- Neo4J version 4.4.12 (Webber, 2012) (NEO4J);
- NebulaGraph version 3.5.0 (Vesoft Inc/Nebula, 2023) (NEBULA);
- Kuzu version 0.0.6 (Jin et al., 2023) (KUZU);
- Jena TDB version 4.1.0 (J. Team, 2021) (JENA);
- Blazegraph version 2.1.6 (Thompson et al., 2014) (BLAZEGraph); and
- Virtuoso version 7.2.6 (Erling, 2012) (VIRTUOSO).

From the aforementioned systems, Neo4J and Nebula use the ALL TRAIL semantics by default. Neo4J and Kuzu support the ANY SHORTEST WALK and ALL SHORTEST WALK modes. Kuzu also supports the ALL WALK semantics, but to assure finite answers, all paths are limited to a length of at most 30. In terms of SPARQL systems (Jena, Blazegraph, Virtuoso), these do not return paths, but do support arbitrary RPQs and according to the SPARQL standard (Harris et al., 2013), detect pairs of nodes connected by an arbitrary walk. A brief summary of supported features can be found in Table 1.1 in the introduction. Other systems we considered are DUCKDB (ten Wolde et al., 2023), Oracle Graph Database (Oracle, n.d.) and Tiger Graph (T. Team, 2021), which support (parts of) SQL/PGQ. Unfortunately, (Oracle, n.d.) and (T. Team, 2021) are commercial systems with limited free versions, while the SQL/PGQ module for DUCKDB (ten Wolde et al., 2023) is still in development.

**Hardware Setup.** The experiments were run on a commodity server with an Intel®Xeon® Silver 4110 CPU, and 128GB of DDR4/2666MHz RAM, running Linux Debian 10 with the kernel version 5.10. The hard disk used to store the data was a SEAGATE model ST14000NM001G with 14TB capacity. Note that this is a classical HDD, and not an SSD. Custom indexes for speeding up the queries were created for Neo4J, Nebula and Kuzu, and the four systems were run with the default settings and no limit on RAM usage. Jena, Blazegraph, Virtuoso and MillenniumDB were assigned 64GB of RAM for buffering. Since we run large batches of queries, these are executed in succession, in
order to simulate a realistic load to a database system. *All queries were run with a limit of 100,000 results and a timeout of 1 minute.*

### 5.2. Pokec: Scaling the Path Length

Here we take a highly connected graph of medium size and test what happens if we ask for paths of increasing length. All tested systems were able to easily load this data set. Given that SPARQL systems cannot return paths, we compare with them when the system is only asked to retrieve the reachable nodes (the ENDPOINTS experiment). Also, given that other systems only support semantics based on TRAILS and WALKS, we retrieve paths according to these two modes. Our results are presented in Figure 5.2 and Figure 5.3.

**ENDPOINTS.** Results for retrieving reachable nodes are presented in Figure 5.2 (left). As we can see, SPARQL systems handle this use case relatively well (only BLAZEGRAH timed out for the largest path length), while NEO4J also shows decent performance with no timeouts. In contrast, NEBULA and KUZU start timing out for paths of length 5 and 6, respectively. MILLENNIUMDB shows superior performance with a stable runtime. From length 4 onwards, MILLENNIUMDB stabilizes due to pipelined execution.
**Walks.** Results for ANY SHORTEST WALK are given in Figure 5.2 (right). As we can see, KUZU has a highly performant algorithm that handles this use case well, while NEO4J times out rather quickly. MILLENIUMDB is the clear winner, with rather stable performance for longer lengths. We note that the spike in MILLENIUMDB for lengths 3 and 4 is due to loading the data from disk, which then stays in the buffer for longer length paths and stabilizes (recall that we run the queries one after another).

The case of ALL SHORTEST WALK is shown in Figure 5.3 (right). The picture here is similar, with no system being able to handle paths of length 6 or more, while MILLENIUMDB scales very well. The data loading spike is again present for length 3 and 4, but it still results in fast performance. We also conducted an experiment where we look for paths of length precisely \( k \), with \( k = 1 \ldots 12 \), with identical results, so we omit the plot for this case. Overall, we can conclude that MILLENIUMDB offers stable performance, and unlike the other systems, does not get into issues as the allowed path length increases.

**Trails.** The results for ALL TRAIL are presented in Figure 5.3 (left). The performance of NEO4J is much better here than for SHORTEST WALKS, with timeouts occurring
much later. On the other hand, NEBULA could not handle length 5 paths. When it comes to MILLENNIUMDB, the results show the performance of the BFS version of Algorithm 8. For the DFS version (not shown in the figure), the picture is similar. As in other experiments, we see that MILLENNIUMDB can handle the query load with no major issues.

5.3. Wikidata: The Effect of Big Graphs

Here we test whether evaluating path queries in big real-world graphs is feasible. We encountered significant issues when loading the data set into some engines. For NEBULA, we ran into a well documented storage issue (Vesoft Inc., 2023), which we could not resolve, while KUZU ran out of memory while loading the data set. We even tried splitting the data set into smaller chunks, with one file for each distinct edge label. In this case we only managed to load the ten biggest edge sets into KUZU, but this amounted to less than a third of the total number of edges, so we excluded KUZU from this experiment. Apart from MILLENNIUMDB, the systems that could load the data were: NEO4J, BlazeGRAPH, JENA, and VIRTUOSO. Given that NEO4J supports the WALK and TRAIL restrictors, we ran our queries under these two path modes. Out of 659 queries, 20 could not be expressed in NEO4J since they were complex RPQs. Results are depicted in Figure 5.4.

WALKS. In Figure 5.4 (left) we show the results for the WALK restrictor. Since we ran 659 queries, the results are presented as box plots, to allow for a more comprehensive analysis. The first two boxes represent the BFS and DFS version of Algorithm 4 in MILLENNIUMDB, which corresponds to the ANY (SHORTEST) WALK mode. Compared to SPARQL engines (the next three boxes), which do not return paths, we can see a marked improvement. This is also reflected in the number of timeouts. Here MDB-BFS had 8 and MDB-DFS 9 timeouts. In contrast, JENA timed out 95 times, and BlazeGRAPH and VIRTUOSO 86 and 24 times, respectively. We remark that while NEO4J supports the ANY SHORTEST WALK mode, it timed out in 657 out of 659 queries, so we do not include

\(^1\)https://docs.nebula-graph.io/3.5.0/20.appendix/0.FAQ/
Figure 5.4. Runtimes for the Wikidata experiment.

it in the graphs. The final box in Figure 5.4 (left) shows the performance of MILLENNIUMDB for the ALL SHORTEST WALK mode, using Algorithm 6. Interestingly, despite requiring a more involved algorithm, finding 100,000 paths under this path mode shows almost identical performance to finding a single shortest path for each reached node. The number of timeouts here was only 7. Again, while NEO4J does support this path mode, it could only complete 2 out of 659 queries. Overall, we can conclude that MILLENNIUMDB presents a stable strategy for finding WALKS in all tested instances.

**TRAILS.** Results for the TRAIL semantics are shown in Figure 5.4 (right). The first two boxes correspond to ANY SHORTEST TRAIL and ANY TRAIL in MILLENNIUMDB. The performance here is almost identical to the ANY WALK case, with only 25 and 24
timeouts for the BFS and DFS version, respectively. The following three boxes correspond to the ALL TRAIL path mode, supported by MDB-BFS, MDB-DFS, and Neo4J. Comparing MillenniumDB to Neo4J, we can see an order of magnitude improvement in performance. This is reflected in the number of timeouts, with 134 for Neo4J, and only 9 for MDB-BFS and 10 for MDB-DFS. The final box corresponds to the ALL SHORTEST TRAIL mode in MillenniumDB, which again shows similar performance to other TRAIL-based modes, with only 21 timeouts. Overall, we can see that MillenniumDB shows remarkably stable performance when returning trails.

Interestingly, while the theoretical literature classifies the TRAIL mode as an intractable problem (Baeza, 2013), and algorithms proposed in Chapter 3 take a brute-force approach to solving them, over real-world data they do not seem to fare significantly worse than algorithms for the WALK restrictor. This is most likely due to the fact that they can either detect 100,000 results rather fast, or because the data itself permits no further graph exploration. We remark that we also ran the experiments for the SIMPLE and ACYCLIC restrictors in MillenniumDB, with identical results as in the TRAIL case, showing that Algorithm 8 is indeed a good option for real-world use cases.

5.4. Diamond: Scaling the Number of Paths

Here we test the performance of a query looking for paths between the nodes start and end in the graph of Figure 5.1. We scale the size of the database by setting $n = 1, \ldots, 40$. This allows us to test how the algorithms perform when the number of paths is large, i.e. $2^n$. For each value of $n$ we will look for the first 100,000 results. To compare with other engines, we focus on the WALK restrictor and the TRAIL restrictor. All the other path modes in MillenniumDB, which is the only system supporting them, have identical performance as in the TRAIL case, since they are all derivatives of Algorithm 8. Given that SPARQL systems cannot return paths, we exclude them from this experiment.
SHORTEST WALKS. The runtimes for the ANY SHORTEST WALK and ALL SHORTEST WALK modes are presented in Figure 5.5. Here we compare with NEO4J and KUZU, since NEBULA only supports the TRAIL restrictor. Due to the small size of the graph, we run each query twice and report the second result. This is due to minuscule runtimes which get heavily affected by initial data loading. As we can observe, for ANY SHORTEST WALK (Figure 5.5 (left)), all the engines perform well (we also pushed this experiment to $n = 1000$ with no issues). Notice that KUZU only works up to $n = 15$ since the longest path its implementation supports is of length 30.

In the case of ALL SHORTEST WALK (Figure 5.5 (right)), NEO4J times out for $n = 16$. Same as before, KUZU stops at $n = 15$ with a successful execution. Overall, MILLENIUMDB seems to have a linear time curve in this experiment, while the other engines grow exponentially, showing the full power of Algorithm 6 when returning 100,000 paths. We tried scaling to $n = 1000$ and the results were quite similar.

TRAILS. Apart from comparing with other systems, this experiment also allows us to determine which traversal strategy (BFS or DFS) is better suited for Algorithm 8, in extreme cases such as the graph of Figure 5.1. We present the results for the ANY TRAIL and ALL TRAIL modes in Figure 5.6. Considering first the ANY TRAIL case, which is
only supported by MILLENIUMDB, the BFS-based algorithm will time out already for \( n = 26 \), which is to be expected, since it will construct all paths of length 1, 2, \ldots, 25, before considering the first path of length 26. In contrast, DFS will find the required paths rather fast, since it will follow each branch of the search directly towards the end node, finding many solutions without having to expand an exponential amount of nodes.

When it comes to the ALL TRAIL mode, the situation is rather similar. Here we also compare with other engines that find trails. As we can see, no engine apart from MDB-DFS could handle the entire query load as they all show an exponential performance curve. This illustrates that for a huge number of trails, DFS is the strategy of choice. Overall, this experiment allows us to pinpoint problems that BFS-based approaches might have in Algorithm 8, particularly when there is a large number of paths that merge, and when the paths are of large length. Interestingly, this behavior never showed up in the real-world data used for the previous experiments.
5.5. Exploring Optimizations

Here we experiment with the potential optimizations that were mentioned in Chapter 4, regarding the use of more efficient indexes to access data, as well as a bidirectional search strategy for algorithms that deal with queries where both endpoints are fixed nodes.

Efficient Indexes. We compare the performance of different indexes under the ANY WALK and ALL SHORTEST WALK modes, using the same Wikidata graph and queries that were used for the large real-world graphs experiment. The indexing methods implemented by MILLENNIUMDB are the following:

- **B+tree** version, which is the default;
- **CSR-cache**, which creates a CSR for each edge label as needed by the query, but caches the already created CSRs to save time later; and
- **CSR-full** version, which assumes CSRs for all edge labels to already be available in main memory.

Results are shown in Figure 5.7. As expected, the CSR-based approaches are, in general, faster than the default B+tree index, since they are able to skip loading pages from disk into memory. For the CSR-cache version, the improvement is not that pronounced, given the fact that it has to spend additional time constructing and loading CSRs into memory every time a new edge label that has not appeared yet requires it. The CSR-full version avoids this limitation, since it already has all the CSRs pre-loaded into memory from the start.

For a realistic scenario, CSR-cache would be the best choice of index, if the available memory in the system running the engine was enough to store all the necessary CSRs. Otherwise, the default B+tree index performs pretty well comparatively, without requiring the construction of data structures mid-evaluation. Finally, the CSR-full index, even when it performs the best, is not suited to practical use, since it would require the construction of all CSRs prior to evaluation, which may not be plausible for large volumes of data.
Bidirectional Search. We compare the performance of the bidirectional algorithms presented in Chapter 4 with their standard counterparts. For this, we also use the large Wiki-data graph, but consider a new set of 204 query patterns of the form \((v, \text{regex}, v')\), which was adapted from the original WDBench queries to be sufficiently challenging. This new set was constructed by extracting the most demanding queries of the original set, and replacing the \(?x\) variable of each query with some fixed node \(v'\) that is reachable from \(v\) under said query. Results are displayed in Figure 5.8.

For the ANY WALK semantics, we observe great improvement in terms of performance using the bidirectional version, to the point where even the hardest queries are answered in a similar time than the easiest ones for the standard version of the algorithm.
This can be interpreted as the bidirectional strategy being very effective at finding solutions before expanding too deeply with BFS, as hypothesized in Chapter 4.

On the other hand, for the ALL SHORTEST WALK and ANY TRAIL semantics, the difference in runtimes is not that meaningful, and in fact, is detrimental in the case of ALL SHORTEST WALK. This means that for these path modes, the cost of some of the steps added to implement the new strategy; such as obtaining the cross-product of paths or finding compatible states according to a restrictor; are simply adding too much overhead to the algorithm, ultimately negating the performance improvements obtained through the bidirectional strategy in the first place. Considering these results, we can conclude that the bidirectional strategy is a very powerful approach when answering query patterns of
the form: ANY (SHORTEST) WALK \((v, \text{regex}, v')\), but falls short for other more complex semantics.

5.6. Experimental Conclusions

Based on our experiments, we believe that with the proposed algorithms, MILLENNIUMDB offers a sound strategy for dealing with path queries. It is highly performant on all the query loads we considered, and runs faster than any other system in every scenario we tested. This is particularly true for the WALK semantics, which runs very fast and with few timeouts, even on huge datasets such as Wikidata.

When it comes to TRAILS, one has to be careful in selecting either the BFS or DFS strategy, with the former being a good candidate for highly connected graphs with few hops, and the latter being able to better handle a huge number of paths.

We also found that under certain conditions, and for certain semantics, the performance can be improved even further via optimizations at the storage level of MILLENNIUMDB (CSR Indexes), and also at the algorithmic level (Bidirectional Search).
6. CONCLUSIONS AND FUTURE WORK

In this thesis we develop algorithms and implement them inside MILLENIUMDB, creating a unifying framework for efficiently returning paths in RPQ query answers. We believe MILLENIUMDB to be the first system that allows returning paths under all path modes prescribed by the GQL and SQL/PGQ query standards (Deutsch et al., 2022), and our experimental evaluation shows the approach to be highly competitive and scalable on realistic workloads. This confirms our working hypothesis, and makes it a highly relevant proposal, with respect to the current lack of support for RPQs in modern graph database engines.

In future work, we would like to extend MILLENIUMDB to cover additional features of interest, such as the GQL top-\(k\) semantics, combinations of different path modes within a single query, and supporting path features in GQL and SQL/PGQ that go beyond RPQs.

Additionally, we plan to test how optimization techniques such as reachability indexes (Su et al., 2016; Zhang et al., 2022), or vectorized and parallel execution of BFS and DFS (Kaufmann et al., 2017; Then et al., 2014) can speed up the algorithms we proposed.
REFERENCES


Vesoft Inc. (2023). NebulaGraph Documentation. Retrieved from https://docs.nebula-graph.io/3.5.0/20.appendix/0.FAQ/


