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

AN RDF DATABASE COMPACT REPRESENTATION FOR TIME- AND SPACE-EFFICIENT REGULAR PATH QUERIES

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

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

JOSEFA IGNACIA ROBERT PARRA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
IVANA BACHMANN ESPINOZA
CECILIA HERNÁNDEZ RIVAS

Este trabajo ha sido parcialmente financiado por FONDECYT Grant 1-230755

SANTIAGO DE CHILE
2024

RESUMEN DE LA TESIS PARA OPTAR
AL GRADO DE MAGÍSTER EN CIENCIAS
DE LA INGENIERÍA, MENCIÓN COMPUTACIÓN
Y MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN
POR: JOSEFA IGNACIA ROBERT PARRA
FECHA: 2024
PROF. GUÍA: GONZALO NAVARRO BADINO

UNA REPRESENTACIÓN COMPACTA DE BASES DE DATOS RDF PARA LA RESOLUCIÓN EFICIENTE EN TIEMPO Y ESPACIO DE RPQS

Esta tesis contribuye al problema de manejar y consultar bases de datos tipo Resource Description Framework (RDF) de manera eficiente, con un enfoque en las Two-way Regular Path Queries (2RPQs). La primera contribución de este trabajo es la implementación de una representación compacta para grafos introducida por Navarro, junto con un conjunto de métodos para extraer información valiosa del grafo. Desarrollamos, sobre esta representación, un algoritmo para resolver 2RPQs, que utiliza autómatas de Glushkov para manejar las expresiones regulares y que simula el autómata en el grafo para identificar los caminos válidos. Para llevar a cabo esto, se utilizan en gran medida los métodos implementados anteriormente para el grafo. Evaluaciones empíricas muestran mejoras en la velocidad de procesamiento de las consultas y en el uso del espacio en comparación con soluciones existentes, particularmente para 2RPQs que involucran una única variable. En general, esta tesis presenta un enfoque competitivo para resolver 2RPQs y establece un rumbo para futuras investigaciones en el área.

AN RDF DATABASE COMPACT REPRESENTATION FOR TIME- AND SPACE-EFFICIENT REGULAR PATH QUERIES

This thesis contributes to the problem of efficiently managing and querying Resource Description Framework (RDF) databases, with a focus on Two-way Regular Path Queries (2RPQs). The first contribution is the implementation of a compact representation for graphs introduced by Navarro, along with a body of methods for extracting valuable information from the graph. We develop, over this representation, a novel algorithm for solving 2RPQs, which uses Glushkov automata for handling regular expression and simulates the automaton on the graph to pinpoint matching paths. Carrying out this relies heavily on the previously implemented methods for the graph. Experimental evaluations demonstrate improvements in query processing speed and space usage when compared to existing solutions, particularly for 2RPQs involving a single variable. Overall, this thesis presents a competitive approach to solving 2RPQs and sets a course for future research in the field.

Acknowledgments

Mi más profundo agradecimiento y reconocimiento a Navarro por su apoyo como director de tesis. Su profesionalismo y minuciosidad son admirables.

Adrián, gracias por la paciencia, por responder todas mis preguntas y por ayudarme con los experimentos. ¡Perdón por los mini infartos! Jajaja.

Sandra, muchas gracias por su ayuda con los asuntos administrativos.

Gracias a I. Bachmann, B. Bustos y C. Hernández por formar parte del comité y por sus valiosos comentarios.

A Basti, mi familia y mis amigos, soy tremendamente afortunada de tenerlos en mi vida.

Un especial agradecimiento a todas las personas anónimas de internet que comparten sus conocimientos desinteresadamente.

Table of content

Introduction

The Resource Description Framework (RDF) was developed by the World Wide Web Consortium to facilitate data exchange on the web. These databases rely on triples, each consisting of a subject, a predicate, and an object, which together form what is known as an RDF graph. The strengths of RDF databases lie in their flexibility and ability to efficiently integrate and query data from varied sources. Due to this, RDF databases have become relevant beyond the semantic web, in fields such as bio-informatics, linguistics and geographic information systems.

SPARQL (a recursive acronym for SPARQL Protocol and RDF Query Language) is the standard query language for RDF databases. One of its important features is that it allows users to define patterns that RDF triples must match, through so-called Property Path queries. These types of queries can be intricately combined and allow for the definition of complex relationships between resources.

Today's massive growth of digital information poses challenges not only in storing and efficiently managing RDF graphs, but also in fully exploiting the value of the data itself, that is, being able to perform queries on the data that return relevant results in reasonable time. To address these issues, numerous indexes for RDF graphs have been proposed, focusing on providing different operations and trade-offs between time efficiency and space consumption.[1]

One set of operations of interest, which are particular instances of Property Path queries, are Regular Path Queries (RPQs). RPQs are used in graph databases to find paths between nodes that match patterns specified by regular expressions. They are especially useful in scenarios where either the path's structure or length is not known in advance, or when one wants to study the graph's topology. RPQs can also be enhanced to traverse edges in both forward and backward directions, known as two-way regular path queries or 2RPQs. The problems related to efficiently handling 2RQPs and RQPs in the context of graph databases are under active research. [19, 30, 6]

This work intends to address the research question: How can we efficiently solve 2RPQs on RDF databases in terms of space and time? To this end, our main objective is to implement a compact structure to represent RDF graphs and use it to develop a competitive algorithm for solving 2RPQs.

Our starting point is the compact index for labelled directed graphs that has been recently introduced by Navarro [26]. This index is inspired by the Burrows-Wheeler transform for data compression, as it is based on encoding the graph as sequences of symbols that follow certain special orderings akin to those of this technique. This approach avoids explicitly storing all the nodes of the graph and results in asymptotically optimal space usage.

The sequences composing the index are supported using succinct structures such as bitvectors [10] and GMR-arrays [17]. These structures are equipped with efficient Rank, Select and Access (RSA) operations, while requiring negligible additional space beyond the optimum. Based on this, we implement the body of methods designed by Navarro to retrieve valuable information from the database, such as those for identifying neighboring nodes and extracting the neighbors of a given node.

The next step in this work is to develop an algorithm that solves 2RPQs for this particular representation of the database. We follow the approach of Arroyuelo *et al.* [3] and rely on two central ideas. The first one is the handling of the regular expressions in the 2RQPs using Glushkov automata, which has the advantage of a parallel traversal. Then, solving a 2RPQ corresponds to computing the pair of nodes that can be simultaneously traversed by a path in the graph and by an accepted path in the automaton. The second important idea is that the traversal in the graph can be efficiently carried out using the methods that we implemented in the first part of the thesis. We hypothesize that the index's methods, along with this parallel product graph approach, will lead to a competitive solution for 2RPQS

We conduct several experiments for testing the time performance and space usage of our algorithm. These are based on a popular real world RDF database: Wikidata [31]. The competing algorithms that we consider are among the most popular stores systems for RDF databases –Jenna, Virtuoso and Blazegraph–, along with the store of Arroyuelo *et al.* which our approach is based on [3].

We propose the following specific objectives:

(1) Implement the compact structure for representing RDF graphs introduced by Navarro [26].

(2) Implement a body of methods over the representation for an efficient retrieval of the data.

(3) Implement an algorithm for solving 2RPQs based on a product graph approach that integrates the aforementioned methods and a parallel traversal.

(4) Conduct experimental evaluations to measure and compare the time performance and space efficiency of our solution using real-world RDF datasets.

Organization

We begin with Chapter 1, which introduces key concepts and reviews relevant literature on RDF, SPARQL, RPQs, and compact data structures.

In Chapter 2, we present the proposed compact graph representation model, detailing its structure, construction, and the operations it supports.

We describe, in Chapter 3, the algorithms for solving 2RPQs, covering both single-variable and double-variable 2RPQs, utilizing the previously introduced representation.

In Chapter 4 we present the experiments conducted on a real-world RDF database, comparing the model against existing benchmarks in query processing and space efficiency.

Chapter 5 concludes the document with a summary of the main findings and suggests directions for further research.

Chapter 1

Background and Related Work

This chapter is devoted to introduce the preliminaries that will be useful for the understanding of the upcoming chapters.

1.1 Resource Description Framework

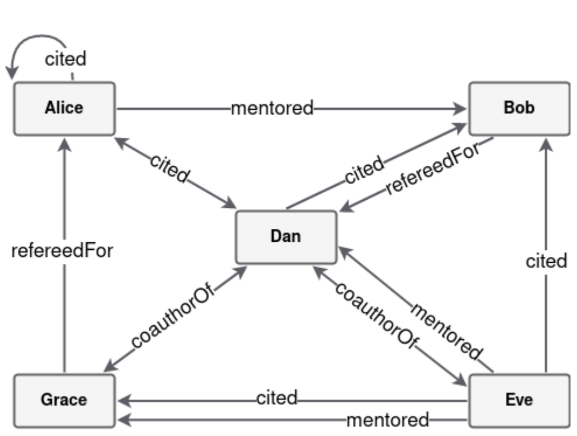
The *Resource Description Framework* [11], or RDF for short, is a database model that has as atomic statements triples of the form $(subject, predicate, object)$, where:

- the subject is an Internationalized Resource Identifier (IRI) [12] or a blank node, and corresponds to an entity (e.g., a person or a building),
- the predicate is an IRI corresponding to a characteristic of the entity (e.g., profession or location),
- the object is an IRI, a literal or a blank node, and corresponds to the value of the predicate for the entity in question (e.g., musician or Santiago).

Here, IRIs represent resource identificators, literals represent strings and datatype values, and blank nodes are for existential variables. IRIs, literals and blank nodes are collectively known as *RDF terms* and are distinct and distinguishable. We call a set of RDF triplets an *RDF graph*. Figure 1.1 illustrate a sample RDF graph, showcasing a network of academics both in graphical format and represented as triples.

The simplicity and extensibility of this model have contributed to its widespread adoption in various domains, including medicine, linguistics, and geography. Moreover, RDF is one of the fundamental components of the semantic web, facilitating the interconnection of diverse data sources across the Internet.

Large-scale projects encompassing hundreds of millions of triples, like YAGO [29] and Wiki-Data [31], have motivated initiatives aimed at designing space and time efficient representations for RDF data and the algorithms that query such data. Luo *et. al.* [22] classify these efforts according to three perspectives: relational, entity-centric, and graph-based approaches.



Subject	Predicate	Object
Alice	mentored	Bob
Alice	cited	Alice
Alice	cited	Dan
Bob	refereedFor	Dan
Dan	cited	Alice
Dan	coauthorOf	Grace
Dan	coauthorOf	Eve
Dan	cited	Bob
Eve	cited	Grace
Eve	mentored	Grace
Eve	mentored	Dan
Eve	cited	Bob
Eve	coauthorOf	Dan
Grace	refereedFor	Alice
Grace	coauthorOf	Dan

Figure 1.1: The table on the right describes an RDF database of researchers and their relationships. The labeled directed graph on the left is a graphical representation of this database. For simplicity, the terms are not prefixed with IRIs as it is usually the case.

The relational perspective represents the data by mapping it to relational database structures. Here, much of the literature focuses on adapting techniques previously developed for this class of databases. A straightforward example along these lines is a relational table where the subjects, predicates and objects of the triples are the columns. Additional schemes that follow this approach can be found in the works by Sakr *et al.* [28] and Luo *et al.* [22].

The entity perspective takes a node-centric view and interprets RDF subjects and objects as entities with properties, where the latter correspond to the predicates. For this type of representation, it is common to use inverted index data structures in conjunction with techniques from the area of information retrieval [1].

Finally, the graph-based perspective, as its name suggests, represents the information as a graph, where subjects and objects are nodes and predicates are the labels of directed edges. For this method, it is common to adapt techniques used in other graph-oriented databases. TurboHOM++ [20], DOGMA [9] and gStore [35] are examples of graph-based RDF stores.

1.1.1 SPARQL

Among the various query languages for RDF (such as RQL, RDQL or SeRQL), SPARQL -a recursive acronym for SPARQL Protocol and RDF Query Language- has been accepted by the community as the standard querying language [1].

The fundamental building blocks of SPARQL queries are *basic graph patterns*. A basic graph pattern is essentially a group of triple patterns that are matched against the RDF data graph. A triple pattern specifies a condition that data must satisfy. It has the form of an RDF triple with the difference that one or more variables can substitute the values of subject, predicate and object.

```
SELECT   ?mentor ?mentee
WHERE {
    ?mentor mentored ?mentee .
    ?mentee refereedFor ?person .
}
```

?mentor	?mentee
Eve	Grace
Alice	Bob

Figure 1.2: SPARQL query utilizing a basic graph pattern to retrieve pairs of individuals where one has mentored the other, and the mentee served as a referee for someone. The evaluation of this query against the RDF data of Figure 1.1 results in the two pairs showed in the table.

Figure 1.2 shows a SPARQL query for a basic graph pattern and its resulting evaluation. The SPARQL syntax is similar to that of the SQL language, and the results obtained can be returned in the form of tables or in a template to generate new data.

Property paths

Property paths queries are a feature added in SPARQL 1.1 that permits defining patterns of relationships between resources by matching arbitrary length paths in the RDF graph. A trivial case is a property path of length exactly 1, which is a triple pattern. The end of a property path may be an RDF term or a variable. Variables cannot be used as part of the path itself, only at the ends.

We follow the work of Kostylev *et al.* [21], and describe a property path query as follows. Let I be a set of IRIs, L the set of literals, B the blank nodes, and V an infinite set of variables. We set $T = I \cup L \cup V$. A *property path expression* is an expression of the grammar

$$e := a;\ \hat{e}; e_1/e_2; e_1|e_2;\ e^+;\ e^*;\ e?;\ !a_1, \dots, a_k;\ !\hat{a}_1, \dots \hat{a}_k,$$

with $a, a_1, \dots, a_k \in I$. Here, $\hat{e}$ denotes the inverse expression, / is the concatenation operator, | is the logical disjunction, + is the Kleene plus, $*$ is the Kleene star, and $!a_1, \dots, a_k$ is the expression that avoids all the a_i. Then, a *property path pattern* is a tuple in $(I \cup B \cup V) \times E \times (T \cup V)$, where E is the set of property path expressions. Finally, a *property path query* is a SPARQL query with at least one property path pattern.

An example of a property path for the database from Figure 1.1 is the following:

$$\texttt{(?x, }\widehat{\texttt{mentored}}\texttt{/!(mentored|cited), ?y)}$$

This query asks for all tuples (x, y) in which x has been mentored by someone who has a relationship with y that is not mentoring or citing. The only pair that satisfies these conditions is (Bob, Grace).

Another important type of query that allows pattern matching and traversal through graph data structures are *regular path queries* (RPQ). While the syntax and applications of RPQs and property paths differ, solving RPQs is roughly equivalent to solving property paths that do not use inverses ($\hat{e}$) or negations ($!a$). The focus of our work will be to develop methods for handling RPQs. We introduce them in the next section.

1.2 Regular Path Queries

Regular path queries, or just RPQs, are a fundamental concept for graph databases [24]. It allows searching for paths within a graph that match a certain regular expression condition over the edge or node labels. RPQs are instrumental in a wide range of applications where the relationships between the entities and therefore the topology of the graph are as important as other data attached to the entities, such as social networks, biological networks or semantic web data.

Definitions

We formalize RPQs as follows. Let $G = (V, E)$ be a graph, where $E \subseteq V \times \mathcal{L} \times V$ is a set of labeled edges and V the set of nodes. A path ρ in G is a sequence of edges $(e_1, ..., e_n)$, with $n \geq 0$ and $e_i = (s_i, p_i, o_i)$, such that $o_j = s_{j+1}$, for $j \in \{1, \ldots, n-1\}$. In other words, each edge within a path initiates from the node where the preceding edge ends. If $n = 0$, then we call ρ the *empty path*. The labels of ρ is the sequence $\text{labels}(\rho) = (p_1, ..., p_n)$. Finally, we say that s_1 is the *starting node* of ρ and o_n is the *ending node* of ρ.

A *regular expression* over an alphabet $\mathcal{L}$ is obtained as follows:

(1) Any element of $\mathcal{L}$ is a regular expression.

(2) The element ϵ is a regular expression, called the *empty path*.

(3) If R, R_1, and R_2 are regular expressions, then the Kleene star R^*, the concatenation R_1/R_2, and the disjunction $R_1|R_2$ are regular expressions.

We abbreviate R^*/R as $R+$ and $\epsilon|R$ as $R?$.

A *regular path query* Q is an expression of the form (x, R, y) where x, y are either in V or are variables representing an element of V, and R is a regular expression over $\mathcal{L}$. We define the *evaluation* of Q on G as the set of pairs (s, o) such that there exists a path ρ starting at s, ending at o, and where $\text{labels}(\rho)$ matches R.

To illustrate the definitions, let us consider again the RDF graph of Figure 1.1. Imagine we wish to identify the individuals linked through a sequence of citations starting from Alice, and then obtain a list of their students. The RPQ that encodes this query is

```
(Alice, cited+/mentored, ?x),
```

which, in this case, yields one solution tuple: (Alice, Bob).

If we now want to obtain the mentors of those who are linked through a sequence of citations starting from Alice, instead of their students, we need to traverse certain edges in the opposite direction. To address this issue, RPQs are often augmented with the capability to traverse edges both forwards and backwards. The queries obtained in this way are called *two-way regular path queries*, or *2RPQs* for short.

We formalize a 2RPQ as follows. Let $\hat{\mathcal{L}} = \{\hat{s} : s \in \mathcal{L}\}$ be the set of formal inverses of the symbols in $\mathcal{L}$, and let $\mathcal{L}^{\leftrightarrow} = \mathcal{L} \cup \hat{\mathcal{L}}$. The inverses satisfy $\mathcal{L} \cap \hat{\mathcal{L}} = \emptyset$ and $s = \hat{\hat{s}}$. We denote by $\hat{G} = \{(y, \hat{p}, x) : (x, p, y) \in G\}$ the inverse of the graph G, and by $G^{\leftrightarrow} = G \cup \hat{G}$ the *completion* of G or the *extended* G.

A 2RPQ is a regular expression over $\mathcal{L}^{\leftrightarrow}$. Note that the inverse $\hat{R}$ of a 2RPQ R, which is defined as R read backwards and with each symbol inverted, is also a 2RPQ. A 2RPQ is an RPQ (x, R, y) on $G^{\leftrightarrow}$, and its evaluation on G is the same as its evaluation on $G^{\leftrightarrow}$ as an RPQ over $\mathcal{L}^{\leftrightarrow}$. Thus, 2RPQs are essentially the same as property path queries that do not use negations.

The 2RPQ that modifies the prior example and asks for the mentors of those who are linked through a sequence of citations starting from Alice, is the following:

$$\texttt{(Alice, cited+/}\widehat{\texttt{mentored}}\texttt{, ?x)}$$

The evaluation of this query results in the pairs (Alice, Alice) and (Alice, Eve).

Solving RPQs

Efficiently retrieving results for RPQs has motivated the development of various strategies. These efforts can be broadly categorized into datalog-based, search-based, index-based, automata-based strategies or a combination of them. We summarize the approaches in the following.

(1) *Datalog-based* strategies involve rewriting the RPQs using Datalog rules, for example by translating the Kleene star operator into recursive Datalog or SQL programs, and the subsequent utilization of Datalog engines for the query evaluation over the graph data. These approaches particularly excels in handling extensive, distributed graph databases, although its expressive power is limited to the Datalog/SQL-based methods.

(2) *Search-based* approaches mainly rely on graph traversal algorithms to find the paths complying with the RPQ. Common techniques encompass pattern matching, BFS, DFS or using an auxiliary distance matrix. As an example, Baier *et al.* [6] use the classic AI search A^*-algorithm.

(3) The *index-based* approaches, as their name suggests, employ auxiliary indexes for speeding up specific operations. Some examples are 3-Dimensional distance matrices, d-Path indexes, or Join-ahead pruning. The main challenge of this method is the potential massive index size. For instance, the computational cost associated with the indexes from [19, 30, 34], is $O(|V|^2)$, where $|V|$ represents the number of nodes in the graph database. This increase is not practical when dealing with massive databases, especially when working with in-memory storage.

(4) *Automata-based* approaches involve converting the regular expression and the graph into nondeterministic or deterministic finite automata, for example using Thompson's

construction, to subsequently navigate them to guide the search for matching paths. It is also common to intersect the resulting automata, as that yields the subgraph specified by the query. The downside is that automata-based methods may lead to a runtime complexity of $O(|V| \cdot |S|)$, where $|V|$ is the number of nodes in the graph and $|S|$ is the number of automaton states. This means mapping every state onto every node, which can be inefficient.

The Ring

Let us briefly describe the work of Arroyuelo *et. al.* [3], which is an important motivation to our approach for solving RPQs and the main benchmark we use in Chapter 4.

Originally developed to optimize join queries [5], the ring is an in-memory compressed RDF graph representation that uses almost the same amount of storage as a basic representation of graph triples.

The ring, inspired by the Burrows-Wheeler Transform (BWT), adapts a previously dictionary-encoded RDF graph into a bidirectional, circular index that covers all triple permutations. Specifically, the n RDF triples are rotated according to the patterns (s, p, o), (p, o, s), and (o, s, p), where s denotes the subjects, p predicates, and o objects. For each of these rotation patterns, the triples are sorted lexicographically and the last term of the triples are stored into three sequences: L_o, L_s, and L_p. The concatenation of the resulting sequences is what is akin to the Burrows–Wheeler Transform (BWT) of the concatenation of all triples. Formally:

- $L_o[1 \ldots n]$ contains the objects o from the list of lexicographically sorted triples (s, p, o).
- $L_s[1 \ldots n]$ contains the subjects s from the list of lexicographically sorted triples (p, o, s).
- $L_p[1 \ldots n]$ contains the predicates p from the list of lexicographically sorted triples (o, s, p).

In a later article [3], the authors used the ring to design a novel algorithm that evaluates RPQs. We give an overview of their approach.

The case with double-variables queries is handled by reducing it to a set of single-variable queries. Furthermore, the authors reduce queries of the form (s, R, x) to the case $(x, \hat{R}, s)$, where R is the inverted regular expression. This reduction is possible as they build the ring on an extended version of the RDF graph that includes the inverted edges. This also has the advantage of enabling 2RPQs, but it has the drawback of using twice the storage of a compact data representation.

Let us now describe their method for solving a single-variable RPQ of the form (y, R, o), where y is the variable. They first build the bit-parallel Glushkov automaton for the regular expression R, as presented in Subsection 1.3. Then, they simultaneously, traverse the Glushkov automaton and the ring, starting from the final states of the automaton and from the vertex o. Each step consists of three parts:

(1) Identify the predicates associated to the current object o, which are the elements of an interval in L_p containing all edges with object o.

(2) For each obtained predicate p, identify the subjects s of edges of the form (s, p, o), which are the elements of an interval in L_s.

(3) Reinterpret each subject s as an object o, and for each of them create a new branch that starts from part 1.

Branches are discarded after parts 1 and 2 if the resulting ranges in L_p or L_s become empty, or if the automaton becomes inactive. The current node s is reported as a solution when the initial state of the automaton is reached.

One of the advantages of the ring is that obtaining the predicates and subjects in part 1 and 2, respectively, can be computed in batch using the ring's so-called backward search. Furthermore, they use wavelet trees [18] to store the sequences, which have the ability to work on ranges of symbols to carry out steps 1 and 2 in such a way that optimal time is spent in identifying the resulting predicates and subjects.

As a result of the previous two advantages and the use of the bit-parallel navigation of the automaton, the authors achieved on average 2.8 times faster query performance than their competitors, and 3–5 times less storage compared to the state-of-the-art graph databases designed for RPQs [3].

1.3 Glushkov Automata

We proceed to describe an efficient method for determining if a word satisfies a given regular expression. The starting point is the description of a regular expression as a nondeterministic finite automaton (NFA), which we now define.

Let R be a regular expression. An *automaton describing* R is a quintuple $G_R = (Q, \mathcal{L}_R, \Delta, q_0, F)$, where:i need to put a — inside a

(1) Q is a set of states with $|Q|$ elements,

(2) $\mathcal{L}_R$ is the set of symbols occurring in R together with, maybe, the empty word ε,

(3) Δ are the transitions,

(4) q_0 is the initial state,

(5) F is a binary word of length $|Q|$ that indicates the set of final states,

such that a word w satisfies R if and only if there is a path in G_R reading w from the initial state to a final state.

There are several constructions for an automaton describing R. The one that best fits our needs is the Glushkov Automaton [14]. This is a type of NFA with interesting properties that makes it useful in the context of formal language theory and regular expressions. Its main properties are stated in Theorem 1.1 below.

Theorem 1.1 ([7, 14]) *Let R be a regular expression on m symbols. There is an automaton $G_R = (Q, \mathcal{L}_R, \Delta, q_0, F)$ describing R,* –Glushkov automaton– *such that:*

(1) There are no ε-transitions, i.e., $\varepsilon \notin \mathcal{L}_R$.

(2) There are exactly $m+1$ states, that is, $|Q| = m+1$.

(3) All the transitions arriving at a state have the same label.

The automaton can be constructed in $O(m^2)$ time and uses $\Theta(m^2)$ space.

The properties in Theorem 1.1 permit to define procedures to traverse the Glushkov automaton non-deterministically. In the rest of this section, we present the *bit-parallel traverse* of a Glushkov automaton, which has the advantage of allowing fast, non-deterministic traversal of the automaton, both forwards and backwards.

Let R be a regular expression with m symbols in $\mathcal{L}$ and $G_R = (Q, \mathcal{L}_R, \Delta, q_0, F)$ be its Glushkov automaton. We start with the following observation: The properties in Theorem 1.1 imply that

($\square$) In G_R, the states reached in one step from a set X of states by symbol c are the intersection of those reached from X in one step and those states reached by c from any state.

Let us write, for simplicity, $\mathcal{L}_R = \{1, \ldots, \sigma\}$, $Q = \{1, 2, \ldots, m+1\}$ and $q_0 = 1$. We denote by

(1) D a binary word containing $m+1$ bits, which represents a non-deterministic state of G_R, *i.e.*, a subset of Q,

(2) $B[1, \ldots, \sigma]$ a table containing binary words of length $m+1$, in which $B[c]$ indicates with 1s the states targeted by transitions labeled c,

(3) $T[0, \ldots, 2^{m+1}-1]$ a table storing in $T[X]$, where X is a $(m+1)$-bit argument representing a subset of Q, the set of states reachable from X in one step using any symbol.

We use the symbol & to denote the bit-wise and operator. Then, ($\square$) ensures that

($\triangle$) $T[D]$ & $B[c]$ is the binary array of length $m+1$ indicating the states of G_R that are reachable from D using a transition labeled c.

Then, the non-deterministic forward traversal of G_R can be done as follows:

(1) Set $D \leftarrow 2^{m+1}$ to activate the initial state q_0.

(2) If D & $F \neq 0$, then we have reached a final state, and the word that has been read so far is accepted.

(3) If $D = 0$, then there are no more active states, and the procedure ends.

(4) For each input symbol c, we use ($\triangle$) to update D as

$$D \leftarrow T[D] \ \& \ B[c].$$

(5) Return to the second step.

It is also possible to read the input text in reverse. For this, we first build a table $T'[0, \ldots, 2^{m+1}-1]$ such that $T'[X]$ marks with 1s the states that reach some state in X in one step. Then, use the following procedure:

(1) Set $D \leftarrow F$ to activate the final states.

(2) If $D \ \& \ 2^{m+1} \neq 0$, then we have reached a final state, and the word that has been read so far is accepted.

(3) If $D = 0$, then there are no more active states, and the procedure ends.

(4) For each input symbol c, we update D using the formula

$$D \leftarrow T'[D \ \& \ B[c]].$$

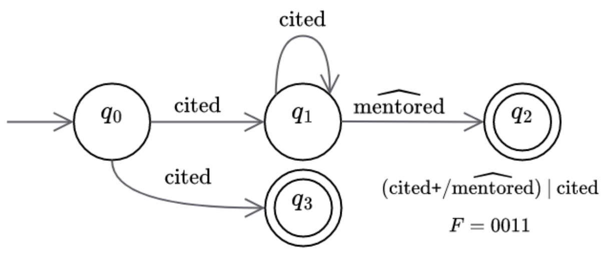
(5) Return to the second step.

The space complexity of the forward or backward traversal is of $O(2^m+\sigma)$ bits. Precomputing B and T (or T') takes $O(2^m)$ time.

As an example, consider the set of symbols consisting of the predicates from Figure 1.1. The regular expression

$$\texttt{(cited+/}\widehat{\texttt{mentored}}\texttt{)|cited} \tag{1.1}$$

can be represented by the Glushkov automaton depicted in Figure 1.3.



B	
cited	0101
$\widehat{\text{mentored}}$	0010
$\mathcal{L}^{\leftrightarrow} \backslash \{\widehat{\text{mentored}}, \text{cited}\}$	0000

T	
1000	0101
0100	0110
1100	0111
1010	0101
...	...

T'	
0001	1000
0010	0100
0100	1100
0011	1100
...	...

Figure 1.3: Glushkov Automaton for the regular expression from (1.1). Note that, in this case, $m = 3$.

1.4 Compact Data Structures

Amidst the rapid growth in data storage and processing, the demand for efficient and space-saving data structures has become increasingly important. Compact structures address this need by aiming to prioritize minimal memory usage while upholding robust data representation and functionality.

Three basic operations when working with compact structures are rank, select and access (or RSA). We define them as follows.

Let $S[1,n]$ be a sequence on the alphabet $\Sigma = \{1,\ldots,\sigma\}$ and $a \in \Sigma$ a letter, then:

- $\mathsf{access}(S,i)$ returns the letter in $S[i]$, for any $1 \leq i \leq n$.
- $\mathsf{rank}_a(S,i)$ returns the number of occurrences of a in $S[1,i]$, for any $1 \leq i \leq n$. In particular, we assume $\mathsf{rank}_a(S,0) = 0$.
- $\mathsf{select}_a(S,j)$ returns the position of the j-th occurrence of a in S, for $j \geq 0$. We assume $\mathsf{select}_a(S,0) = 0$ and $\mathsf{select}_c(S,j) = n+1$ if $j > \mathsf{rank}_a(S,n)$.

We write by default rank_1 and select_1 as rank and select, respectively.

Two important algorithms that are enabled by the RSA operations are the *predecessor* and the *successor* functions. The predecessor $\mathsf{pred}_a(S,i)$ gives the position of the last occurrence of a in $S[1\ldots i]$, and can be computed using the formula

$$\mathsf{pred}_a(S,i) = \mathsf{select}_a(S, \mathsf{rank}_a(S,i)).$$

Similarly, $\mathsf{succ}_c(S,i) = \mathsf{select}_a(S, \mathsf{rank}_a(S,i-1)+1)$ is the position of the first occurrence of c in $S[i\ldots n]$.

We now present the compact data structures for storing binary and general sequences that we will use in later chapters.

1.4.1 Bitvectors

A *bitvector* (also known as bitmask or bit array) B is a structure that describes a sequence of size n over the alphabet $\Sigma = \{0,1\}$. Bitvectors are the basis of many compact structures, so numerous representations have been developed. There is usually a trade-off between availability and time complexity of the RSA operations and space complexity of the structure. For instance, a length-n bitvector B can be stored using $n \cdot H_0(B)$ bits, where H_0 is the zero-order entropy, at the same time that rank and select can be done in $O(1)$ and $O(\log n)$ time, respectively. In turn, if we increase the store complexity to $n + o(n)$, then we can implement the so-called *plain bitvector*, which supports constant time for the three RSA operations.

In this work, we will use *plain bitvectors*. Their main properties are summarized in the theorem below.

Theorem 1.2 ([10]) *Let B be a binary sequence of length n. Then, it is possible to store B in such a way that:*

(1) $n + o(n)$ bits are used, and

(2) rank*,* select *and* access *require $O(1)$ time.*

We briefly describe the structure of binary sequences used in Theorem 1.2 and how their RSA algorithms achieve $O(1)$ time.

The discrete interval $\{1, 2, \ldots, k\}$ is denoted by $[k]$. Let B be a binary sequence of length n. We use n bits to store it as a regular array, which permits computing access in constant time. For each of rank and select, we will introduce additional structures of $o(n)$ bits to ensure $O(1)$ time for those operations.

Let us start with rank. We divide B into *super-blocks* B_i of length $p = (\log n)^2/2$. We store in an array $S[1, n/p]$ the answers to $\mathsf{rank}_1(B, i)$ for all positions i that indicate the end of a super-block B_i. Since $S[1, n/p]$ only contains integers in $[p]$, we require no more than $(n/p) \cdot \log p = o(n)$ bits to store $S[1, n/p]$.

Next, we divide each super-block B_i into *blocks* $B_{i,j}$ of length $q = (\log n)/2$. As with super-blocks, we store in an array $R_i[1, p/q]$ the results of $\mathsf{rank}_1(B_i, k)$ for all positions k that indicate the end of a block $B_{i,j}$. This time, we use $(p/q) \cdot \log(p/q)$ bits for each R_i; since there are n/p arrays R_i, the total used space is of at most $n \cdot (2 \log\log n)/\log n$ bits, which is $o(n)$.

The last step of the construction is storing in a table $T \colon [2^q] \times [q] \to [q]$ the results of rank for all binary sequences C of length q (note that C is seen as an element of $[2^q]$). This uses at most $2^q \cdot q \cdot \log(q) = O(n^{1/2} \log\log n)$ bits, which is $o(n)$ space complexity.

The final step is to recover rank from S, R and T. Note that, for binary sequences, $\mathsf{rank}_1(B, i) = i - \mathsf{rank}_0(B, n)$, so it is enough to implement $\mathsf{rank}_1(B, i)$. For $i \in [n]$, we decompose i in terms of the lengths of the hierarchical decomposition that we have defined for B. More precisely, we can uniquely write $i = j \cdot p + k \cdot q + r$ where $j \in [n/p]$, $k \in [p/q]$ and $r \in [q]$. Then,

$$\mathsf{rank}_1(B, i) = S[j] + R[j \cdot (p/q) + k] + T[C][r],$$

where $C = B[j \cdot p + k \cdot q, j \cdot p + (k+1) \cdot q]$. The decomposition of i is $O(1)$ time, so rank ends up having $O(1)$ time.

It rests to implement select. This can be done with the same idea that we used for rank, but with a modification: instead of using constant-length blocks (super-blocks), we choose their lengths in such a way that the numbers of 1s in each block (super-block) is constant. This will require $o(n)$ additional bits and yield $O(1)$ time for select.

1.4.2 GMR Sequences

We now turn to consider sequences over general alphabets. We will focus on the compact data structure introduced by Golynski, Munro and Rao [17], which we call *GMR sequences.* The following theorem summarizes the main aspects of this representation.

Theorem 1.3 (Item (1), Theorem 2.2, [17]) *Let $\Sigma = \{1, 2, \ldots, t\}$ be a finite alphabet and*

$x \in \Sigma^n$ be a sequence over Σ of length $n \geq t$. The GMR representation of x is such that

(1) it can be stored using $n\log(t) + o(n\log(t))$ bits, and

(2) it supports rank *and* access *in $O(\log\log(t))$ time, and* select *in $O(1)$ time.*

We review, in the rest of this subsection, the construction of the GMR sequences. Let $x \in \Sigma^n$ be a sequence over $\Sigma = \{1, \ldots, t\}$. The construction used in Theorem 1.3 has three parts. First, we decompose x into *chunks* and *blocks*, and define efficient *block operations*, based on bitvectors, to access to them. Then, we treat each chunk independently and introduce *local operations* for them. Finally, the two types of operations are combined to get support for rank, access and select.

We start with some preliminaries. Let us write $[k]$ for the discrete interval $\{1, 2, \ldots, k\}$. As $t \leq n$, we can assume for simplicity that t divides n.

Let $T\colon [t] \times [n] \to \{0, 1\}$ be the binary table such that

$$T[c, i] = 1 \text{ if and only if } c \text{ occurs at position } i \text{ of } x.$$

We define the bitvector B as the concatenation of the rows of $T[c, 1]T[c, 2] \ldots T[c, n]$ in increasing order. Note that B has length tn. Also, there is a simple relationship between the operations $\mathsf{rank}_c(x, j)$ $\mathsf{select}_c(x, i)$ on x and $\mathsf{rank}_1(B, j)$ and $\mathsf{select}_1(B, i)$ on B:

$$\begin{aligned} \mathsf{rank}_c(x, j) &= \mathsf{rank}_1(B, (c-1)n + i) - \mathsf{rank}_1(B, (c-1)n), \\ \mathsf{select}_c(x, i) &= \mathsf{select}_1(B, \mathsf{rank}_1(B, (c-1)n) + i)) - (c-1)n. \end{aligned}$$

We now describe the main part of the construction.

Part 1. *Partitioning x into chunks.* We decompose x into *chunks* $C_i = x[it + 1]x[it + 2] \ldots x[(i + 1)t]$ of length t. Similarly, B is partitioned into *blocks* B_i of length t. Note that there are n/t chunks and n blocks. Observe also that B is obtained from x in the same way that $B_i B_{i+n/t} \ldots B_{i+(t-1)n/t}$ is obtained from C_i. Thus, to obtain access to the chunks C_i, it is enough to have adequate algorithms for B. We provide such algorithms in the following.

Define the *block operations*:

$$\mathsf{rank\text{-}b}(it) = \mathsf{rank}_1(B, ti) \text{ and } \mathsf{select\text{-}b}(i) = \left\lfloor \frac{\mathsf{select}_1(B, i)}{t} \right\rfloor. \tag{1.2}$$

We have that $\mathsf{rank\text{-}b}(it)$ is the number of 1s seen in $B_1 B_2 \ldots B_i$, and that $\mathsf{select\text{-}b}(it)$ is the block number to which the i-th 1 of B belongs.

We implement the operations in (1.2) as follows. Set $k_i = \mathsf{rank\text{-}b}(it) - \mathsf{rank\text{-}b}((i-1)t)$ and $B' = 1^{k_1}01^{k_2}0 \ldots 1^{k_n}0$. Observe that the block operations in (1.2) can be computed using RSA algorithms on B':

$$\begin{aligned} \mathsf{rank\text{-}b}(it) &= \mathsf{rank}_1(B', \mathsf{select}_0(B', i)), \\ \mathsf{select\text{-}b}(i) &= \mathsf{rank}_0(B', \mathsf{select}_1(B', i)). \end{aligned}$$

Thus, as B' has length $2n$, a plain bitvector implementation for B' (see Subsection 1.4.1) yields $2n + o(n)$ bit-space complexity and $O(1)$ time complexity for both block operations $\mathsf{rank\text{-}b}(it)$ and $\mathsf{select\text{-}b}(i)$.

Part 2. *Local operations on chunks.* The local operations are based on the efficient representation of permutation of Munro *et al.* [25]. Let C_i be a chunk of x, $\ell_{i,j}$ be the number of times that j occurs in C_i, and $k_{i,j}(1), k_{i,j}(2), \ldots, k_{i,j}(\ell_{i,j})$ the (possibly empty) list of positions at which j occurs in C_i, written in increasing order. We define the sequence

$$\pi_i = k_{i,1}(1), \ldots, k_{i,1}(\ell_{i,1}), k_{i,2}(1), \ldots, k_{i,2}(\ell_{i,2}) \ldots k_{i,t}(1), \ldots, k_{i,t}(\ell_{i,t}).$$

Then, π_i is a permutation of $[t]$. We encode π_i using the representation of Munro *et al.* [25], which has the following properties:

(1) it uses $t\log(t) + o(t\log t)$ bits of space, and

(2) it supports computing π in $O(1)$ time and π^{-1} in $O(\log\log(t))$ time.

We also define the bitvector

$$D_i = 01^{\ell_{i,1}}01^{\ell_{i,2}}\ldots 01^{\ell_{i,t}}.$$

As D_i has length $2t$, using the plain bitvector implementation from Subsection 1.4.1 yields $2t + o(t)$ bit-space complexity and $O(1)$ time for RSA operations. This bitvector facilitates the computation of the *local operations* access and select on C_i:

$$\mathsf{access}(C_i, j) = \mathsf{rank}_0(D_i, \mathsf{select}_1(D_i, \pi_i^{-1}(j))),$$
$$\mathsf{select}_c(C_i, j) = \pi_i(\mathsf{select}_0(D_i, c) + j - c).$$

Remark that these formulas yield the time complexities $O(\log\log(t))$ and $O(1)$, respectively.

It rests to implement the local operation rank on D_i. For $j \in [t]$, we store the positions $E_{i,j}$ corresponding to $\log(t)$-th occurrence of j in C_i within a *y-fast trie* $F_{i,j}$ [33]. As j occurs $k_{i,j}$ times in C_i, there are at most $k_{i,j}/\log(t)$ elements stored in $F_{i,j}$. Hence, $F_{i,j}$ satisfies the following:

(1) it uses $O(k_{i,j})$ bits;

(2) it supports $\mathsf{rank}_1(F_{i,j}, k)$ in $O(\log\log(t))$ time.

Note that $\mathsf{rank}_1(F_{i,j}, k)$ permits to approximate $\mathsf{rank}_1(C_i, k)$ using that

$$\log(t)\cdot\mathsf{rank}_1(F_{i,j}, k) \leq \mathsf{rank}_1(C_i, k) < \log(t)\cdot\mathsf{rank}_1(F_{i,j}, k) + \log(t). \qquad (1.3)$$

Then, a binary search between the positions $\log(t)\cdot\mathsf{rank}_1(F_{i,j}, k)$ and $\log(t)\cdot\mathsf{rank}_1(F_{i,j}, k) + \log(t)$ of $E_{i,j}$ yields the value $\mathsf{rank}_1(C_i, k)$.

The time cost for computing the approximation (1.3) is $O(\log\log(t))$, which is also the cost of a binary search in $E_{i,j}$ in an interval of length at most $\log(t)$. Therefore, this implementation of $\mathsf{rank}_1(C_i, k)$ has a time cost of $O(\log\log(t))$. To estimate the space complexity, we first note that there are n/t permutations π_i, n/t bitvectors D_i, and that the sum of the $k_{i,j}$ is exactly the length n of x. Hence, we have used no more than

$$(n/t) \cdot (t\log(t) + o(t\log(t))) + (n/t) \cdot (2t + o(t)) + \sum_{i,j} O(k_{i,j}) = n\log(t) + o(n\log t) \text{ bits.}$$

Part 3. *From block and local operations to global ones.* It is not difficult to check that

$$\begin{aligned} \mathsf{rank}_c(x,j) &= \mathsf{rank\text{-}b}(it) + \mathsf{rank}_c(C_i, j - it), \text{ and} \\ \mathsf{access}(x,j) &= \mathsf{access}(C_i, j - it), \end{aligned}$$

where $i = \lfloor j/t \rfloor$, and that

$$\mathsf{select}_c(x,j) = \mathsf{select}_c(C_i, j - \mathsf{rank\text{-}b}(it - t)) + (i-1)t,$$

where $i = \mathsf{select\text{-}b}(j)$. It then follows from the computations from Parts 1 and 2 that $\mathsf{rank}_c(x,j)$, $\mathsf{access}(x,j)$ and $\mathsf{select}_c(x,j)$ can be computed in $O(\log\log(t))$, $O(\log\log(t))$ and $O(1)$ time, respectively.

1.4.3 SDSL: Succinct Data Structure Library

The Succinct Data Structure Library (SDSL) [15] is a C++ library that provides a collection of compressed data structures and algorithms for efficiently managing and manipulating large volumes of data, and has been shown in practice to obtain spaces and times close to the theoretical ones.

We will implement the graph representation from Chapter 2 by relying on the SDSL library. Specifically, for the plain bitvectors we use the class `bit_vector` with the methods `rank_support_v` and `select_support_mcl`, and, for the GMR sequences, we use the class `wt_gmr`.

Chapter 2

Graph Representation

This chapter is devoted to describing the implementation of the compact graph structure introduced by G. Navarro [26] and the operations it supports. We also analyze its theoretical time- and space-complexities and present its building algorithm.

2.1 Description of the Data Structure

Let G be a directed labeled graph over a set of labels $\mathcal{L}$. We represent G as a pair $G = (V, E)$, where V is the set of nodes and $E \subseteq V \times \mathcal{L} \times V$ is the set of edges. We write the edges of G as (s, l, o) and use the language of graph databases, that is, we call the elements in $\mathcal{L}$ *predicates* or *labels*, and the elements in V *subjects* or *objects*. The cardinalities of V, E and $\mathcal{L}$ are denoted by n, e and λ, respectively. For simplicity, we assume that $V = \{1, \ldots, n\}$ and $\mathcal{L} = \{1, \ldots, \lambda\}$.

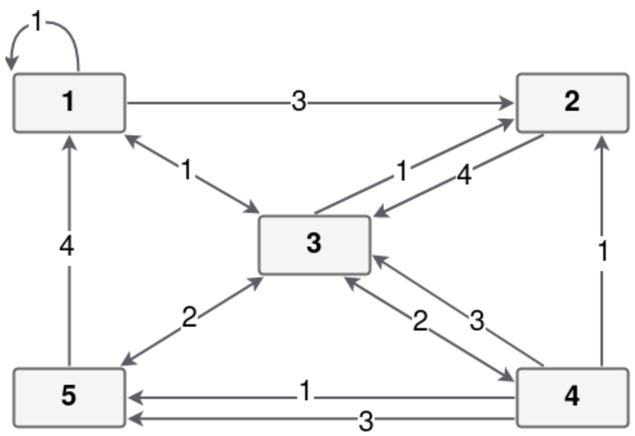
Our representation consists of sequences L, B_L, N and B_N, that we define as follows.

Consider the set $\{(s_i, l_i, o_i) : 1 \leq i \leq e\}$ of all edges, lexicographically sorted, first with respect to s, then o and finally l. Then, we define L as the sequence $[l_1, l_2, \ldots, l_e]$. In this way, L can be seen as an ordered concatenation of subsequences L_s (for $s \in V$), which contain the labels for all the triples with subject s. Similarly, for each L_s, we have smaller divisions $L_{s,o}$ representing all the edges with subject s and object o.

To identify the length of each segment L_s, we define the binary sequence $B_L = 10^{|L_1|}10^{|L_2|}1 \cdots 0^{|L_n|}$. Note that $|B_L| = n + e$.

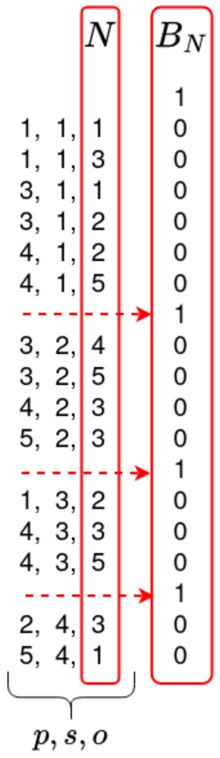
The sequences N and B_N are defined similarly. Let $\{(s_i, l_i, o_i) : 1 \leq i \leq e\}$ be the set of all edges, lexicographically sorted, first with respect to l, then s and finally o. We set $N = [o_1, \ldots, o_e]$. Observe that N is the ordered concatenation of the subsequences N_l (for $l \in \mathcal{L}$) containing the objects o_i such that $l_i = l$, and that, for each N_l, we have smaller divisions $N_{l,s}$ representing all the edges with predicate l and subject s. Based on the lengths of the N_l segments, we define $B_N = 10^{|N_1|}10^{|N_2|}1 \cdots 0^{|N_\lambda|}$. We have that $|B_N| = \lambda + e$.

Figure 2.1 shows the sequences N, L, B_L, and B_N for the database from Figure 1.1.



predicates	
cited	1
coauthorOf	2
mentored	3
refereedFor	4

subjects/objects	
Alice	1
Bob	2
Dan	3
Eve	4
Grace	5



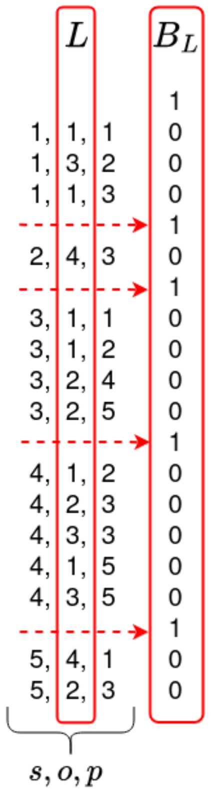


Figure 2.1: The figure is divided into two parts: On the left, we present the numerically encoded version of the graph from Figure 1.1, along with its corresponding translation to the original terms. The diagram on the right shows how N is derived from the triples sorted by p, then s, then o, with the red arrows highlighting transitions in the predicates. These transitions mark the beginnings of the N_l segments and lead to the 1s in B_N. Similarly, the equivalent figure for L and B_L is illustrated using the triples under the other order.

Note that our representation does not explicitly store the subjects of the edges, as they are only implicitly associated with the L_s segments. To retrieve the original triplets and other useful information about the graph, we can use RSA operations on the sequences L, B_L, N and B_N. This will be detailed in the next section.

2.2 Basic Graph Operations

The first group of methods that our representation supports corresponds to classical operations from the literature: to obtain, for a given node, its indegree, outdegree, neighbors set and reverse neighbors set; and to decide whether two nodes are adjacent. We formalize these operations as follows.

- $\textsc{adj}(G, u, v)$: Returns 1 if there exists an edge from u to v; equivalently, whether $(u, l, v) \in G$ belongs to G for some l.
- $\textsc{neigh}(G, u)$: returns the list of neighbors of u, *i.e.*, $\{v : \exists l \text{ s.t. } (u, l, v) \in G\}$.
- $\textsc{rneigh}(G, v)$: returns the list of reverse neighbors of v, *i.e.*, $\{u : \exists l \text{ s.t. } (u, l, v) \in E\}$.
- $\textsc{outdegree}(G, u)$: returns the number of neighbors of u, *i.e.*, $|\textsc{neigh}(u)|$.
- $\textsc{indegree}(G, v)$: returns the number of reverse neighbors of v, *i.e.*, $|\textsc{rneigh}(v)|$.

We also aim to support the analogous operations $\textsc{adj}_l$, $\textsc{neigh}_l$, $\textsc{rneigh}_l$, $\textsc{outdegree}_l$ and $\textsc{indegree}_l$, which do the same as the ones above but work on the subgraph G_l consisting of all the edges with label l.

Let us start by computing the outdegree of a node v. Note that a segment L_v is associated with all the triplets having that fixed v. Thus, the length of L_v is equal to the outdegree of v. We can obtain the beginning and end of L_v using $\textsc{select}(B_L, v)$ and $\textsc{select}(B_L, v+1)$, respectively. To obtain the indegree of a node v, we can simply count the occurrences of v as an object in N using $\textsc{rank}_v(N, e)$. The methods for outdegree and indegree are shown in Algorithm 1.

Algorithm 1

```
1: procedure Outdegree($G, v$)
2:     $b_1 \leftarrow \textsc{select}(B_L, v)$
3:     $b_2 \leftarrow \textsc{select}(B_L, v+1)$
4:     return $b_2 - b_1 - 1$

5: procedure Indegree($G, v$)
6:     return $\textsc{rank}_v(N, e)$
```

We now explain how to compute $\textsc{Neigh}(G, v)$. The strategy is the following. Let $j \geq 1$. We first find the j-th element l of L_v, which is associated with a certain edge $e \in E$. Then, we search the segment N_l for the object corresponding to e. Finally, to obtain the neighbors set of v, we iterate the previous procedure over the all the indexes $j \in [1, \textsc{Outdegree}(v)]$.

The starting point of L_v is equal to $\textsc{select}(B_L, v) - v$; then, the position of l in L_v is obtained by including the offset j, that is,

$$l = L[p], \text{ with } p = \textsc{select}(B_L, v) - v + j.$$

Next, we can identify the beginning of the segment N_l as $\textsc{select}(B_N, l) - l$. It only remains to add the offset in N_l of the occurrences of the labels l that appear before, in L, than the label of the j-th neighbor of v. This can be obtained by counting the occurrences of l in $L[1, \ldots, p]$ using $\textsc{rank}_l(L, p)$. Therefore, the j-th neighbor of v is equal to

$$N[\textsc{select}(B_N, l) - l + \textsc{rank}_l(L, p)].$$

Algorithm 2 details this method. Note that there is no additional cost in returning the label associated with each neighbor.

Algorithm 2

```
1: procedure Neigh($G, v$)
2:     for each $j \in [1 \ldots \texttt{OUTDEGREE}(v)]$ do
3:         $p \leftarrow \textsc{select}(B_L, v) - v + j$
4:         $l \leftarrow \textsc{access}(L, p)$            ▷ label associated with the $j$-th neighbor
5:         $q \leftarrow \textsc{select}(B_N, l) - l$
6:         $m \leftarrow \textsc{rank}_l(L, p)$
7:         $S[j] \leftarrow \textsc{access}(N, q + m)$
8:     return $S$
```

We continue with $\textsc{rneigh}$. Let $j \geq 1$. We first consider the j-th occurrence of v in N, which is associated with a certain edge $e \in E$ and is contained in a segment N_l. Then, we search the segment L for the label corresponding to e. This occurrence is contained in a segment L_s, and s is reported as the j-th reverse neighbor of v. Finally, the reverse neighbors set of v is obtained by iterating the previous procedure over the all the indexes $j \in [1, \textsc{Indegree}(v)]$.

The j-th occurrence of v in N is at position $p = \textsc{select}_v(N, j)$. Then, p is contained in the segment N_l, where $l = \textsc{select}_0(B_N, p) - p$. Now, the segment N_l in N starts at $q = \textsc{select}(B_N, l) - l + 1$. Thus, the j-th occurrence of v in N corresponds to the $(p-q+1)$-th e labeled l, and therefore to the position $r = \textsc{select}_l(L, p - q + 1)$ of L. We finally obtain our target node as $s = \textsc{select}_0(B_L, r) - r$. The pseudocode of this method is given in Algorithm 3.

Algorithm 3

```
1: procedure Rneigh($G, v$)
2:     for each $j \in [1 \ldots \texttt{INDEGREE}(v)]$ do
3:         $p \leftarrow \textsc{select}_v(N, j)$
4:         $l \leftarrow \textsc{select}_0(B_N, p) - p$       ▷ label associated with the $j$-th reverse neighbor
5:         $q \leftarrow \textsc{select}(B_N, l) - l + 1$
6:         $r \leftarrow \textsc{select}_l(L, p - q + 1)$
7:         $S[j] \leftarrow \textsc{select}_0(B_L, r) - r$
8:     return $S$
```

One of the disadvantages of our representation is that there is no direct way of checking if two nodes are adjacent. The only way of doing so is by iterating the method $\textsc{Adj}_l$ for each label l, which we now define.

Let $l \in [1, \ldots, \lambda]$. To decide whether there is an edge from v to u in the subgraph G_l, we identify the subsegment of N_l corresponding to the subject v (recall that the lexicographical order we used to define N was with respect to p, then s, and finally o) and check if u occurs in it. To do this, we first locate the beginning of the segment N_l using $r = \textsc{select}(B_N, l) - l$. Then, to get the offset at which the objects associated with v begin, we must count the number of edges having predicate l and whose subjects are less than v. This can be done using the formula

$$q_1 = \textsc{rank}_l(L, p_1), \text{ where } p_1 = \textsc{select}(B_L, v) - v.$$

Note that p_1 is the beginning of the segment L_v and that $r + q_1$ is the beginning of the subsegment of N_l for the subject v. Similarly, p_2 is the end of L_v and $r + q_2$ is the end the subsegment of N_l for the subject v, where

$$p_2 = \textsc{select}(B_L, v + 1) - (v + 1) \text{ and } q_2 = \textsc{rank}_l(L, p_2).$$

Then, there is an edge from v to u if and only if

$$\textsc{rank}_u(N, r + q_2) - \textsc{rank}_u(N, r + q_1) \text{ is equal to 1.}$$

We refer the reader to Algorithm 4 for the pseudocodes of $\textsc{Adj}(v, u)$ and $\textsc{Adj}_l(v, u, l)$.

Algorithm 4

```
 1: procedure Adj($G, u, v$)
 2:     for each $l \leftarrow 1 \ldots \lambda$ do
 3:         if Adj$_l(G, u, v, l)$ then
 4:             return True
 5:     return False

 6: procedure Adj$_l(G, v, u)$
 7:     $r \leftarrow \textsc{select}(B_N, l) - l$
 8:     $p_1 \leftarrow \textsc{select}(B_L, v) - v$
 9:     $p_2 \leftarrow \textsc{select}(B_L, v + 1) - (v + 1)$
10:     $q_1 \leftarrow \textsc{rank}_l(L, p_1)$
11:     $q_2 \leftarrow \textsc{rank}_l(L, p_2)$
12:     if $\textsc{rank}_N(r + q_2, u) - \textsc{rank}_N(r + q_1, u) = 1$ then
13:         return True
14:     else
15:         return False
```

Let us now implement $\textsc{Outdegree}_l$ and $\textsc{Indegree}_l$. To get the outdegree of a node v in the subgraph G_l, we can count all the edges l in the interval $L_1 \ldots L_v$ and then subtract the number of those in $L_1 \dots L_{v-1}$. The ends of these intervals correspond to the beginning and end of L_v, so they are given by

$$p_1 = \textsc{select}(B_L, v) - v \text{ and } p_2 = \textsc{select}(B_L, v+1) - (v+1), \text{ respectively.}$$

Finally, the answer is given by $\textsc{rank}_l(L, p_2) - \textsc{rank}_l(L, p_1)$.

Similarly, $\textsc{Indegree}_l$ is equal to the difference between the number of times that v appears in $N_1 \dots N_{l-1}$ and in $N_1 \dots N_l$. Note that ends of these intervals are given by

$$r_1 = \textsc{select}(B_N, l) - l \text{ and } r_2 = \textsc{select}(B_N, l+1) - (l+1), \text{ respectively.}$$

So, $\textsc{Indegree}_l = \textsc{rank}_v(N, r_2) - \textsc{rank}_v(N, r_1)$.

Algorithm 5 presents the pseudocode for $\textsc{Outdegree}_l$ and $\textsc{Indegree}_l$.

Algorithm 5

```
1: procedure $\textsc{Outdegree}_l(G, v)$
2:     $p_1 \leftarrow \textsc{select}(B_L, v) - v$
3:     $p_2 \leftarrow \textsc{select}(B_L, v+1) - (v+1)$
4:     return $\textsc{rank}_l(L, p_2) - \textsc{rank}_l(L, p_1)$

5: procedure $\textsc{Indegree}_l(G, v)$
6:     $r_1 \leftarrow \textsc{select}(B_N, l) - l$
7:     $r_2 \leftarrow \textsc{select}(B_N, l+1) - (l+1)$
8:     return $\textsc{rank}_v(N, r_2) - \textsc{rank}_v(N, r_1)$
```

We end the section by describing the algorithms for $\textsc{Neigh}_l$ and $\textsc{Rneigh}_l$.

The neighbors of v in the subgraph G_l are the objects in the subsegment $N_{l,v}$ of N_l that corresponds to the subject v. Recall that we determined the subsegment $N_{l,v}$ when we presented the algorithm for $\textsc{Adj}_l$. In particular, we know that its beginning is at $r+q$, where

$$r = \textsc{select}(B_N, l) - l,\ q = \textsc{rank}_l(L, p) \text{ and } p = \textsc{select}(B_L, v) - v.$$

Then, the neighbors set of v in G_l is given by

$$\{N[r+q+j] : j \in [1 \dots \textsc{Outdegree}_l(v)]\}.$$

We now describe how to obtain the j-th reverse neighbor of v in G_l. Recall that N_l starts at position $r = \textsc{select}(B_N, l) - l$ of N. Thus, the position in N of the j-th occurrence of v in N_l can be calculated as

$$t = \textsc{select}_v(N, p), \text{ where } p = \textsc{rank}_v(N, r) + j.$$

The position t in N corresponds to the position $q = \textsc{select}_l(L, t-r)$ of L. Then, q is a position inside a subinterval L_s, and we report s as the j-th neighbor.

The last two procedures are detailed in Algorithm 6. Illustrative examples for $\textsc{Neigh}_l$ and $\textsc{Rneigh}_l$ are depicted in Figures 2.2 and 2.3, respectively.

Algorithm 6

```
1: procedure $\text{NEIGH}_l(G, v)$
2:     $p \leftarrow \text{SELECT}(B_L, v) - v$
3:     $q \leftarrow \text{RANK}_l(L, p)$
4:     $r \leftarrow \text{SELECT}(B_N, l) - l$
5:     for each $j \in [1 \ldots \text{OUTDEGREE}_l(v)]$ do
6:         $S[j] \leftarrow \text{ACCESS}(N, r + q + j)$
7:     return $S$

8: procedure $\text{RNEIGH}_l(G, v)$
9:     $r \leftarrow \text{SELECT}(B_N, l) - l$
10:    for each $j \in [1 \ldots \text{INDEGREE}_l(v)]$ do
11:        $p \leftarrow \text{RANK}_v(N, r) + j$
12:        $t \leftarrow \text{SELECT}_v(N, p)$
13:        $q \leftarrow \text{SELECT}_l(L, t - r)$
14:        $S[j] \leftarrow \text{SELECT}_0(B_L, q) - q$
15:    return $S$
```

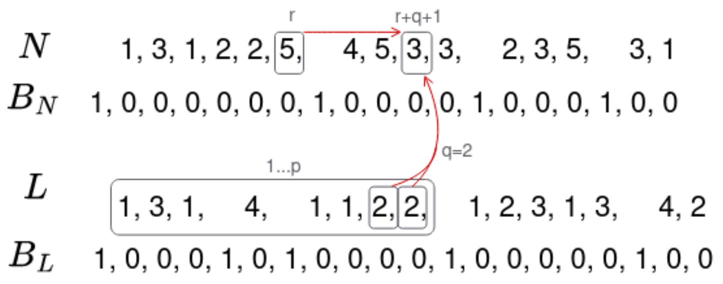

r
r+q+1
N 1, 3, 1, 2, 2, 5, 4, 5, 3, 3, 2, 3, 5, 3, 1
B_N 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0
q=2
1...p
L 1, 3, 1, 4, 1, 1, 2, 2, 1, 2, 3, 1, 3, 4, 2
B_L 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0

Figure 2.2: Illustration of the computation of $\text{NEIGH}_2(G, 4)$ for the graph from Figure 2.1. This query is equivalent to retrieving the neighbors of Eve who are connected by the `coauthorOf` relationship. The variables r, q and p correspond to those in Algorithm 6.

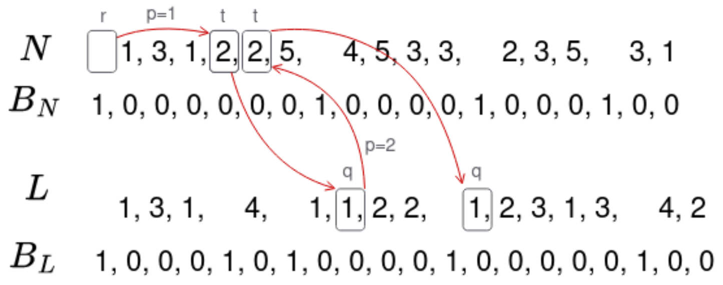

r
p=1
t
t
N 1, 3, 1, 2, 2, 5, 4, 5, 3, 3, 2, 3, 5, 3, 1
B_N 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0
p=2
q
q
L 1, 3, 1, 4, 1, 1, 2, 2, 1, 2, 3, 1, 3, 4, 2
B_L 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0

Figure 2.3: Illustration of the computation of $\text{RNEIGH}_1(G, 2)$ for the graph from Figure 2.1. This query asks for the reverse neighbors of Bob that are connected to him by an edge whose label is `cited`. The variables r, p, t and q correspond to those in Algorithm 6.

2.3 Additional Graph Operations

Due to the nature of our representation, other operations that provide additional information about the graph can be implemented efficiently. We describe them in this section.

- $\textsc{access}_l(G, j)$: returns the j-th edge with label l in G_l.
- $\textsc{count}_l(G)$: returns the number of edges with label l, *i.e.,* $|\{(u, v) : (u, l, v) \in E\}|$.
- $\textsc{sources}_l(G)$: returns the nodes u that are the origin of an edge with label l, *i.e.*, $\{u : \exists v \text{ s.t. } (u, l, v) \in E\}$.
- $\textsc{targets}_l(G)$: returns the nodes v that are the target of an edge with label l, *i.e.*, $\{v : \exists u \text{ s.t. } (u, l, v) \in E\}$.

In the rest of this section, we describe algorithms for implementing the previous operations.

We can straightforwardly implement $\textsc{count}_l$ using $\textsc{rank}_l(L, e)$. See Algorithm 7.

Algorithm 7

```
1: procedure $\textsc{count}_l(G)$
2:     return $\textsc{rank}_l(L, e)$
```

Next, we consider $\textsc{access}_l(G, j)$. Let $e = (v, l, u)$ be the edge associated with the j-th occurrence of l in L. That occurrence of l in L is at position $p = \textsc{select}_l(L, j)$. Then, L_v is the subsegment of L containing p, where $v = \textsc{select}_0(B_L, p) - p$. Finally, to obtain the position of u in N, we compute the start of the subsegment N_l using $\textsc{select}(B_N, l) - l$ and then add the offset j, that is,

$$u = \textsc{access}(N, q), \text{ where } q = \textsc{select}(B_N, l) - l + j.$$

The pseudocode is shown in Algorithm 8.

Algorithm 8

```
1: procedure $\textsc{access}_l(G, j)$
2:     $p \leftarrow \textsc{select}_l(L, j)$
3:     $v \leftarrow \textsc{select}_0(B_L, p) - p$
4:     $q \leftarrow \textsc{select}(B_N, l) - l + j$
5:     $u \leftarrow \textsc{access}(N, q)$
6:     return $(v, l, u)$
```

To implement $\textsc{sources}_l$, we start by computing the first occurrence of l in L with the formula $p = \textsc{select}_l(L, 1)$. We report the subject s of the L_s interval to which $L[p]$ belongs. To avoid adding duplicates (which occurs if the same subject is connected to two different objects by the same label), we move forward until we reach the end of L_s at $c = \textsc{succ}_1(B_L, s{+}p) - (s{+}1)$. Note that the last occurrence of l in L_s is the $\textsc{rank}_l(L, c)$-th one. We repeat this procedure by looking for the next occurrence of l (which occurs at position $p = \text{SELECT}_l(L, \text{RANK}_l(L, c)+1)$) until we get to the last occurrence of l in L. The pseudocode is given in Algorithm 9 .

Algorithm 9

```
1: procedure SOURCES$_l(G)$
2:     $c \leftarrow 0$
3:     next $\leftarrow 1$
4:     while $true$ do
5:         $p \leftarrow \text{SELECT}_l(L, \texttt{next})$
6:         $s \leftarrow \text{SELECT}_0(B_L, p) - p$
7:         APPEND$(S, s)$
8:         $c \leftarrow \text{SUCC}_1(B_L, s + p) - (s + 1)$
9:         next $\leftarrow \text{RANK}_l(L, c) + 1$
10:        if next $> \text{COUNT}_l(L)$ then                ▷ If we have found them all.
11:            return $S$
```

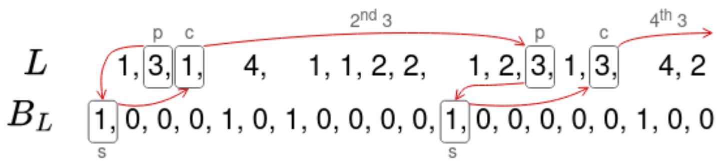


Figure 2.4: Example for $\text{SOURCES}_3(G)$, equivalent to retrieving the subjects of triples with *mentored* as a predicate. The variables s, p, and, c correspond to those in Algorithm 9.

It is left to implement $\text{TARGETS}_l(G)$. Note that the objects of the edges with label l are stored consecutively in N_l. So, it is enough to find the beginning and the end of this segment and then remove the duplicates. The ends of N_l are at

$$p_1 = \text{SELECT}(B_N, l) - l + 1 \text{ and } p_2 = \text{SELECT}(B_N, l + 1) - (l + 1).$$

We remove the duplicates using a classic sort algorithm with duplicates removal.

There are array data structures that permit removing duplicates more efficiently, such as the wavelet tree [18], which implement an *intersect* method efficiently. However, this is not the case for GMR arrays.

We refer the reader to Algorithm 10 for the pseudocode.

Algorithm 10

```
1: procedure TARGETS$_l(G)$
2:     $p_1 \leftarrow \text{SELECT}(B_N, l) - l + 1$
3:     $p_2 \leftarrow \text{SELECT}(B_N, l + 1) - (l + 1)$
4:     for each $k$ in $[p_1, \ldots, p_2]$ do
5:         APPEND$(S, \text{ACCESS}(N, k))$
6:     return SORTANDREMOVEDUPLICATES$(S)$
```

2.4 Time and Space Complexities

In this section we present the time and space complexities of the structures and methods developed.

Recall that G has n nodes, e edges and that there are λ distinct labels. We assume that $\lambda \leq n \leq e$, which is almost always the case in practical situations.

We use the plain bitvector implementation from Subsection 1.4.1 for B_N and B_L, and the GMR arrays from Section 1.4.2 for L and N. Tables 2.5 and 2.6 summarize the resulting space and time complexities for the structures and their methods. Note that the total space used by our representation is

$$(1 + o(1))(e \log(\lambda n) + n + \lambda) \text{ bits}$$

which is asymptotically optimal.

Given the pseudocodes provided previously, it is a routine computation obtaining the time complexities of the algorithms from Sections 2.2 and 2.3. We refer the reader to Table 2.7 for an overview of these times.

Structure	**Space Complexity**
N	$e \log n + o(e \log n)$
L	$e \log \lambda + o(e \log \lambda)$
B_N	$e + \lambda + o(e + \lambda)$
B_L	$e + n + o(e + n)$

Figure 2.5: Space complexity of storing B_L, B_N, N and L.

Method	**Time Complexity**
Any RSA for B_L or B_N	$O(1)$
RANK and ACCESS for N	$O(\log \log n)$
RANK and ACCESS for L	$O(\log \log \lambda)$
SELECT for N and L	$O(1)$

Figure 2.6: Time complexities of the RSA operations for B_L, B_N, N and L.

Method	Time Complexity
$\textsc{outdegree}(G, v)$	$O(1)$
$\textsc{indegree}(G, v)$	$O(\log \log n)$
$\textsc{neigh}(G, v)$	$\textsc{outdegree}(G, u) \cdot O(\log \log n)$
$\textsc{rneigh}(G, v)$	$\textsc{indegree}(G, v) \cdot O(1)$
$\textsc{adj}(G, u, v)$	$\lambda \cdot O(\log \log \lambda)$
$\textsc{adj}_l(G, u, v)$	$O(\log \log \lambda)$
$\textsc{outdegree}_l(G, v)$	$O(\log \log \lambda)$
$\textsc{indegree}_l(G, v)$	$O(\log \log n)$
$\textsc{neigh}_l(G, v)$	$\textsc{outdegree}_l(G, v) \cdot O(\log \log n)$
$\textsc{rneigh}_l(G, v)$	$\textsc{indegree}_l(G, v) \cdot O(\log \log n)$
$\textsc{count}_l(G)$	$O(\log \log \lambda)$
$\textsc{access}_l(G, v, j)$	$O(\log \log n)$
$\textsc{sources}_l(G)$	$\lvert\textsc{sources}_l(G)\rvert \cdot O(\log \log \lambda)$
$\textsc{targets}_l(G)$	$\textsc{count}_l(G) \cdot O(\log \log n + \log \textsc{count}_l(G))$ [1]

Figure 2.7: Time complexities of the algorithms from Sections 2.2 and 2.3.

2.5 Construction Algorithm

A naive implementation of the compact representation (B_L, B_N, N, L) requires performing two sorts on an array of length e, which may be overly time-consuming. This final section is devoted to describing how this representation of G can be computed using just one $O(e \log e)$-time sort, along with other additional $O(e)$ time operations.

Let us write $E = \{(s_i, l_i, o_i) : i = 1, \ldots, e\}$ for the set of edges of G. We start by sorting E lexicographically according to the pattern (s, o, l). Let $\mathsf{objs} = [o_1, o_2, \ldots, o_e]$ and $\mathsf{preds} = [l_1, l_2, \ldots, l_e]$.

The next step is to build two auxiliary arrays for each of objs and preds. We explain the procedure only for objs, as for preds it is analogous. Let N_{acc} be the array of length $n + 1$ that contains in its position v the total number of occurrences of nodes $v' < v$ in objs. Then, B_N satisfies

$$B_N = 10^{N_{\mathsf{acc}}[2]-N_{\mathsf{acc}}[1]}10^{N_{\mathsf{acc}}[3]-N_{\mathsf{acc}}[2]}10^{N_{\mathsf{acc}}[4]-N_{\mathsf{acc}}[3]} \ldots 10^{N_{\mathsf{acc}}[n+1]-N_{\mathsf{acc}}[n]}.$$

It is possible to compute N_{acc} and B_N in $O(e)$ time using the following procedure: First, we traverse objs and save in $N'_{\mathsf{acc}}[l]$ the number of times $l \in [1 \ldots \lambda]$ appears in objs. Observe that $N_{\mathsf{acc}}[l] = N'_{\mathsf{acc}}[1] + \cdots + N'_{\mathsf{acc}}[l-1]$ and that the 1s of B_N are at positions $N_{\mathsf{acc}}[l] + l$. Therefore, N_{acc} and B_N can be computed from N'_{acc} in λ steps, yielding a $O(e)$ total time.

Let L_{acc} and B_L be the arrays analogously obtained from preds. Remark that L_{acc} has length $\lambda + 1$. Since E was sorted according to the pattern (s, o, l), L is equal to preds.

It only rests to compute N. The central observation is the following: Suppose that, for $i \in [1 \ldots e]$, k_i is the number of times l_i occurs in $\mathsf{preds}[1 : i]$. Then,

$$N[N_{\mathsf{acc}}[l_i] + k_i] = o_i \text{ for every } i \in [1 \ldots e].$$

This completely determines N. Moreover, N can be computed in $O(e)$ time by iterating through $i \in [1 \dots e]$ at the same time we maintain updated variables k'_l, $l \in \mathcal{L}$, so that k'_l is the number of times l occurs in $\mathsf{preds}[1 : i]$.

Algorithm 11 gives the pseudocode of the procedure described above.

Algorithm 11 Pseudocode for the index building algorithm. The input $E = [(s_i, l_i, o_i) : i = 1, \dots, e]$ of Build-Index is an array containing the edges (s, l, o) of the graph, and the output are the sequences N, B_N, L and B_L that describe the compact representation of the graph. The procedure Build-Auxs receives an array A with values in $[1 \dots m]$ and computes two auxiliary arrays.

```
 1: procedure Build-Index($E$)
 2:     Sort $E$ lexicographically according to the pattern $(s, o, l)$
 3:     $\mathsf{objs} \leftarrow [o_1, o_2, \dots, o_e]$
 4:     $\mathsf{preds} \leftarrow [l_1, l_2, \dots, l_e]$
 5:     $N_{\mathrm{acc}}, B_N \leftarrow$ Build-Auxs($\mathsf{objs}, \lambda$)
 6:     $L_{\mathrm{acc}}, B_L \leftarrow$ Build-Auxs($\mathsf{preds}, n$)
 7:     $\mathsf{idx\_}B_N \leftarrow$ zero-initialized array of length $\lambda$
 8:     $N \leftarrow$ new array of length $e + \lambda$
 9:     for each $i$ in $[1 \dots e]$ do
10:         $(o, l) \leftarrow (\mathsf{objs}[i], \mathsf{preds}[i])$
11:         $N[N_{\mathrm{acc}}[l] + \mathsf{idx\_}B_N[l]] \leftarrow o$
12:         $\mathsf{idx\_}B_N[l] \leftarrow \mathsf{idx\_}B_N[l] + 1$
13:     $N \leftarrow$ Build-GMR-Array($N$)
14:     $L \leftarrow$ Build-GMR-Array($\mathsf{preds}$)
15: procedure Build-Auxs($A$, $m$)
16:     $A_{\mathsf{acc}} \leftarrow$ zero-initialized array of length $m + 1$
17:     for each $x$ in $A$ do
18:         $A_{\mathsf{acc}}[x + 1] \leftarrow A_{\mathsf{acc}}[x + 1] + 1$
19:     $B \leftarrow$ zero-initialized bitvector of length Length($A$) $+ m$
20:     for each $i$ in $[1 \dots m - 1]$ do
21:         $A_{\mathsf{acc}}[i + 1] \leftarrow A_{\mathsf{acc}}[i + 1] + A_{\mathsf{acc}}[i]$.
22:         $B[A_{\mathsf{acc}}[i] + i] \leftarrow 1$
23:     return $A_{\mathsf{acc}}$, $B$
```

We conclude this chapter with a practical consideration on the memory limitations of the machines utilized. It is not difficult to adjust Algorithm 11 to use, at any time, only the k needed bits for storing E, plus $o(k)$ bits. This can be achieved by immediately freeing memory once a variable is no longer needed, using an in-place sorting algorithm and by writing structures to disk during the construction.

Chapter 3

2RPQ Evaluation

The objective of this chapter is to provide to the compact data structure from Chapter 2 an algorithm that efficiently solves 2RPQs. We start by defining the general notation that will be used throughout the sections, and then explain the strategy of the algorithm. The cases of single-variable and double-variable queries will be treated separately in Sections 3.1 and 3.2, respectively.

Notation

Let $G = (V, E)$ be a graph, where V is the set of nodes and $E \subseteq V \times \mathcal{L} \times V$ is a set of labeled edges. We consider 2RPQs of the form (s, R, x), (y, R, o) and (x, R, y), where $s, o \in V$, R is a regular expression over $\mathcal{L}^{\leftrightarrow} = \mathcal{L} \cup \hat{\mathcal{L}}$ and x, y are variables representing an element of V. The first two types of queries correspond to *single-variable queries*, while the last one is a *double-variable query*.

We denote by G_R the Glushkov automaton of the regular expression R with m symbols, as it was introduced in Section 1.3. Recall that G_R is a non-deterministic finite automaton, which we describe as a quintuple $G_R = (Q, \mathcal{L}_R, \Delta, q_0, F)$. Let us write, for simplicity, $\mathcal{L}_R = \{1, 2, \ldots, \lambda\}$. We will use the forward and backward bit-parallel simulation of G_R from Section 1.3. To this end, we use the tables D, B, T and T' introduced therein. Recall that for forward traversal starting from state D and a label c we update

$$D \leftarrow T[D] \;\&\; B[c], \tag{3.1}$$

and that for backward traversal, we use

$$D \leftarrow T'[D \;\&\; B[c]]. \tag{3.2}$$

3.1 Single-variable 2RPQs

We start by discussing a symmetry in the queries : A pair (s, o) is a solution for the 2RPQ (s, R, x) if and only if (o, s) is a solution for $(y, \hat{R}, s)$, where $\hat{R}$ is the inverse of R. Hence, when solving single-variable queries, we can freely choose the position of the variable. Queries of the form (s, R, x) involve traversing G_R forwards, otherwise backwards. This gives us an important degree of freedom: we can choose to traverse G_R forwards or backwards.

Each traversal direction yields a slightly different algorithm.

We first present the algorithm for a query with right-side variable and then show how it can be modified to solve the other case. Consider a query of the form (s, R, x). The base idea is the following: The solutions are the ends of paths ρ in G starting at s such that labels(ρ) is accepted by G_R. Thus, the problem boils down to simultaneously traversing G_R and G. To carry out this idea, we encode each step of the traversal as an *inductive pair* (D, v), with D representing a (non-deterministic) state in G_R and v being a node of G.

Then, a jump in G_R is executed by first choosing a label l as one of the outgoing edges from D, and then computing the new state D' using Formula (3.1). The equivalent jump in G is to the nodes v' obtained using the methods $\textsc{neigh}_l(G, v)$ or $\textsc{rneigh}_l(G, v)$ from Section 2.2, depending on whether l is an inverse label or not. By performing these jumps, for each initial choice of l, we obtain a set of inductive pairs (D', v'), which we add to an induction stack.

There are two critical checks that we make when considering an inductive pair (D, v). First, we have to determine whether v is part of the solution. This is done by verifying that D is an accepting state. Second, we need to check if the algorithm is looping. To that end, we maintain a structure that keeps track of the previously seen inductive pairs.

Let us now explain the exact computations required in each substep. We will denote by (D, v) the current inductive pair of the substeps.

A. Initializing the induction stack The traversal of G_R is forwards, so we start from the initial state q_0; and, in G, we start from s to move to the solutions represented by y. Hence, the induction stack has, at first, (q_0, s).

B. Avoiding redundancies and loops If we have previously seen an inductive pair of the form (D', v), where D' shares active states with D, then the intersection $D\&D'$ gives rise to branches that have already been considered. This is inefficient and can even cause the algorithm to loop.

To avoid redundancy and loops, we employ a table $\textsc{seen}[1 \ldots, n]$ containing in $\textsc{seen}[v]$ the bitvector of length $m + 1$ that represents all the states that have been activated by a previously seen pair of the form (D', v). During Step B, we remove from D the states $\textsc{seen}[v]$, as they have already been considered. We then update $\textsc{seen}[v]$ by adding the new states from D. Formally, we update:

$$D \leftarrow D \ \& \sim\!\textsc{seen}[v] \text{ and } \textsc{seen}[v] \leftarrow \textsc{seen}[v] \mid D,$$

where $\sim$ is the bitwise logical NOT.

C. Deciding to report a solution To check whether the path traversed so far represents a solution to the query, we must check if the current active states of G_R contain one of the final states of the automaton, *i.e.*, if $D \ \& \ F$ is not equal to zero. Note, however, that v may already be in the solution set if it is reachable from multiple accepted paths. To avoid reporting duplicated answers, we must verify before adding v that there is no previously seen pair (D', v), where D' contains an accepted state. This can be done by verifying that $\textsc{seen}[v] \& F$ is equal to zero.

D. Jumping to new inductive pairs It rests to perform the simultaneous jump in G and G_R.

First, we find the set P_D of all the elements $l \in \mathcal{L}_R$ that are the label of an edge in G_R leaving D. Note that, by (3.1), l is in P_D if and only if

$$T[D] \ \& \ B[l] \neq 0. \tag{3.3}$$

Hence, we can obtain P_D by iterating through all the labels $l \in \mathcal{L}_R$ and keeping those for which (3.3) holds. Since we may encounter D multiple times throughout the inductive process, it is convenient to define a table $P[0, \ldots, 2^{m+1} - 1]$ and store P_D in the coordinate D of it as a bitvector of length $m + 1$.

For a fixed $l \in P_D$, we jump to the new state D' reached from D by l using Formula (3.1). The corresponding jump in G is to the neighbors of v with label l if $l \in \mathcal{L}$ and to the reverse neighbors of v with label $\hat{l}$ if $l \in \hat{\mathcal{L}}$. These neighbors are computed using the methods $\textsc{neigh}_l(v)$ and $\textsc{rneigh}_{\hat{l}}(v)$ from Section 2.2.

In this way, every $l \in P_D$ yields a new state D' and a set of neighbors v' of v. For each resulting pair (D', v'), a new branch of the algorithm is created by adding (D', v') to the induction stack.

The previous procedure can be modified to solve a left-variable query (y, R, o). The steps are the same but we traverse G_R backwards. This means that F and q_0 take the opposite roles and that the jump in G_R is backwards. More precisely:

(1) In substep A we initialize the stack with the inductive pair (F, o).

(2) We report v as a solution, in substep C, if $D \,\&\, q_0 \neq 0$ and $\textsc{seen}[v] \& q_0$ is equal to zero.

(3) In substep D, $P[D]$ now corresponds to the *ingoing* predicates, which are determined as those $l \in \mathcal{L}_R$ such that $D \,\&\, B[l] \neq 0$. To obtain the new state D', Formula (3.2) is used, and the jump in G is to the reverse neighbors of v with label l if $l \in \mathcal{L}$ and to its neighbors with label $\hat{l}$ if $l \in \hat{\mathcal{L}}$.

The precise procedure can be found in Algorithm 12.

Algorithm 12 Pseudocode of the algorithm that solves single-variable RPQs presented in Section 3.1, using a forward traversal of G_R. The comments on the right side give the necessary modifications for a backward traversal.

```
1: procedure SINGLE-VAR-RPQ(query)
2:     is_left ← true if query has its variable on its left side; false otherwise
3:     if is_left then                                              ▷ if ∼is_left
4:         query ← INVERSE(query)
5:     Parse query as $(s, R, x)$                                   ▷ Parse query as $(y, R, o)$
6:     $(\mathcal{L}_R, T, T', B, q_0, F)$ ← BUILDGLUSHKOV($R$)
7:     PUSH(induction_stack, $(q_0, s)$)                            ▷ PUSH(induction_stack, $(F, o)$)

8:     while induction_stack is not empty do
9:         $(D, v)$ ← POP(induction_stack)
10:        $D \leftarrow D \,\&\, \sim$seen$[v]$
11:        if $D \neq 0$ then
12:            seen$[v]$ ← seen$[v] \mid D$
13:            if $D \,\&\, F$ is not 0 then                        ▷ if $D \,\&\, q_0$ is not 0
14:                if seen$[v]\&F$ is 0 then                        ▷ if seen$[v] \,\&\, q_0$ is 0
15:                    ADD(solutions, $v$)
16:            if $P[D]$ has not been initialized then
17:                for each $l$ in $\mathcal{L}_R$ do
18:                    if $T[D] \,\&\, B[l]$ is not 0 then          ▷ if $D \,\&\, B[l]$ is not 0
19:                        APPEND($P[D], l$)
20:            for each $l$ in $P[D]$ do
21:                $D' \leftarrow T[D] \,\&\, B[l]$                  ▷ $D' \leftarrow T'[D \,\&\, B[l]]$
22:                if $l$ is not in $\hat{\mathcal{L}}$ then
23:                    $V' \leftarrow$ NEIGH$_l(v)$                  ▷ $V' \leftarrow$ RNEIGH$_l(v)$
24:                else
25:                    $V' \leftarrow$ RNEIGH$_{\hat{l}}(v)$         ▷ $V' \leftarrow$ NEIGH$_{\hat{l}}(v)$
26:                for each $v'$ in $V'$ do
27:                    PUSH(induction_stack, $(D', v')$)
```

3.2 Double-variable 2RPQs

It remains to treat double-variable queries (x, R, y). The naive approach is solving the single-variable queries (v, R, y) for all $v \in V$ (or, symmetrically, the queries (x, R, v)). This is highly inefficient as it may happen that many objects v do not lead to any solution to the original query. We improve this basic method by first obtaining a *feasible set* V', which is a (typically) small subset of V that contains all the nodes v for which (v, R, y) leads to at least one solution to (x, R, y).

The set V' is defined as follows. Let $\mathcal{L}'$ be the set of $l \in \mathcal{L}_R$ that are labels of edges in G_R

starting at q_0. Then, we set

$$V' = \bigcup_{l \in \mathcal{L}' \cap \mathcal{L}} \textsc{sources}_l(G) \cup \bigcup_{l \in \mathcal{L}' \cap \hat{\mathcal{L}}} \textsc{targets}_{\hat{l}}(G).$$

It is not difficult to check that all the subjects in the solutions given by the algorithm from Section 3.1 to the queries (x, R, o), $o \in V$, are in V'. Therefore, the solution set of (x, R, y) is equal to the collection of all solutions of the queries (v, R, y), $v \in V'$.

Let us now describe in detail how we get the elements of V'. First, the labels l connected to q_0, are those that lead to at least one active state when jumping from q_0, *i.e.*, those such that $T[q_0]\ \&\ B[l] \neq 0$. Then, for each such l, we need all subjects of triples with label l in $G^{\leftrightarrow}$. These are given by $\textsc{sources}_l(G)$ if l is not inverted and by $\textsc{targets}_{\hat{l}}(G)$ otherwise.

We present in Algorithm 13 the pseudocode of the described strategy.

Algorithm 13 Pseudocode of the algorithm that solves double-variable RPQs presented in Section 3.2, using a forward traverse of G_R.

```
1: procedure Get-Feasible-Set(query)
2:     Parse query as $(x, R, y)$
3:     $(\mathcal{L}_R, T, T', B, q_0, F) \leftarrow$ BuildGlushkov($R$)
4:     for each $l$ in $\mathcal{L}_R$ do
5:         if $T[D]\ \&\ B[l] \neq 0$ then
6:             if $l$ in $\mathcal{L}$ then
7:                 Push(feasible_set, sources($l$))
8:             else
9:                 Push(feasible_set, targets($\hat{l}$))
10:    return feasible_set
11: procedure Double-Var-RPQ(query)
12:    Parse query as $(x, R, y)$
13:    feasible_set $\leftarrow$ Get-Feasible-Set(query)
14:    for each $v$ in feasible_set do
15:        single_var_query $\leftarrow (v, R, y)$
16:        Push(solutions, Single-Var-RPQ(single_var_query))
17:    return solutions
```

Chapter 4

Experimental Evaluation

To evaluate the efficiency of the 2RPQ algorithm proposed, we have taken an empirical approach and measured its performance on a real-world RDF database. We use the Ring [3] as our primary benchmark due to the fact that our proposal is inspired by it. To this end, we replicate the experimental setup from [3] and compare with all the systems tested therein, focusing on contrasting our solution with the Ring.

The structure of the chapter is the following. Subsection 4.1.1 introduces the RDF database systems that we compare to our algorithm, and Subsection 4.1.2 the specific database and queries used in the benchmarks. Then, in Subsection 4.1.3, we discuss some implementation details, including hardware and software specifications and certain aspects of our algorithm that have not been precised. Finally, Section 4.2 is dedicated to presenting and analyzing the results of the experiments.

4.1 Benchmark and Implementation Details

4.1.1 Benchmark Systems

In addition to the Ring, we compare our algorithm to some well-known platforms for managing and querying RDF databases. We briefly describe them below.

(1) *Blazegraph* is the official SPARQL endpoint used by Wikidata (which is our benchmark database) and by other large customers [8].

(2) *Apache Jena* is a widely used graph database and the reference implementation of the SPARQL standard [2].

(3) *Virtuoso* is a multi-model database that accommodates RDF data, which hosts the public DBpedia endpoint, among others [13].

An important design detail of Jena, Blazergraph and Virtuoso is that they treat constant- and variable-length RPQs (*i.e.* queries with and without the operators $*$ and $+$) differently. For constant-length queries, they translate it into a SPARQL graph pattern without RPQs, and then evaluate it under bag semantics. The other queries are solved by applying set semantics, according to the SPARQL standard. Jena and Blazegraph use a BFS-style function, while Virtuoso employs a transitive closure operator.

The particular implementation of the algorithm over the Ring is the one provided by its creators [3], and can be downloaded from their GitHub page [4]. Jena, Blazegraph and Virtuoso are simply run according to their vendor configurations.

4.1.2 Benchmark Database

Wikidata, created by the Wikimedia Foundation and built upon the RDF framework, is a collaborative and multilingual repository of structured data about various topics [31], which contains $15,019,738,576$ triples as of August 2023. Hogan *et al.* argue that the raw database is not adequate for RPQ benchmarks since it contains information in several different languages and additional information that is not relevant for RPQs, among other things [1]. They propose instead the Wikidata Graph Pattern Benchmark (WGPB) as a benchmark, which is obtained by removing multilingual labels (and keeping only English labels), aliases and descriptions. The result is a graph having $e = 958,844,164$ edges, $|V| = 348,945,080$ nodes, $|S| = 106,736,662$ subjects, $|\mathcal{L}| = 5,419$ predicates, and $|O| = 295,611,216$ objects. This amounts to a total of 10.7 GB in plain form (with 32-bit integers for each triple component, and thus 12 bytes per tuple) and 7.9 GB in packed form (*i.e.*, using $\lceil \log |S| \rceil + \lceil \log |P| \rceil + \lceil \log |O| \rceil$ bits, or 8.63 bytes per tuple). Some of the advantages of the WGPB over other benchmarking databases is its high volume, that it is not synthetic, its interesting graph patterns (such as cycles), and its complex schemata (with over 5000 labels). This database can be downloaded from [32].

Queries

In order to get challenging, real-world RPQs, the authors of [3] extracted all the RPQs done to the Wikidata Query Service that threw timeout error, i.e. that needed more than 60 seconds, from the Wikidata Query Logs [23]. After filtering RPQs using Wikidata-specific features, mentioning constants not used in the dataset, having one label, normalizing variable names, and removing duplicates, this process yielded 1,952 unique queries.

Furthermore, they only keep the 1,583 queries with less than 1 million unique results for comparability reasons (as Virtuoso has a hard-coded limit of $2^{20} \approx 1$ million results). All queries are run with a timeout of 60 seconds under set semantics (using DISTINCT in the case of SPARQL).

We classify the RPQs according to their *pattern* by mapping nodes/variables to constant/variable types and erasing their predicates (that is, we keep only the regular expression operators). For instance, $(x, p_1/p_2^+, v)$ has the pattern $(v, /+, c)$. Table 4.1 presents the 20 most common RPQ patterns in our query set.

Additionally, we will categorize our results based on the query type, distinguishing between single-variable and double-variable queries, as well as by the number of solutions yielded. Table 4.2 shows the distribution of the query types. It is noteworthy that the queries are highly unbalanced, as the majority of them fall into the single-variable category with the constant to the right, and with over half of them producing a relatively small number of results (less than 3,000).

Query pattern	**Count**	**Query pattern**	**Count**
v /∗ c	450	v /? c	20
v ∗ c	421	v ∧ v	14
v + c	107	v | v	13
c ∗ v	101	v | c	9
c /∗ v	100	v ∗ v	8
v / c	48	c |∗ v	7
v ∗/∗ c	30	v /+ c	7
v |∗ c	30	v //∗ c	6
v ∗/∗/∗/∗/∗ c	28	v /| c	6
v / v	26	v ?/∗ c	5

Table 4.1: Top 20 patterns representing 90.7% of the total queries

Query type	**Number of Queries**
Double-variable	81
Single-variable	1502
Left-side constant	258
Right-side constant	1244

Table 4.2: Distribution of the queries according to their type

4.1.3 Implementation Details

The benchmarks for Jena, Virtuoso and Blazegraph were provided to us by the authors of [3]. To test the Ring and our algorithm, we closely replicate their experimental setup.

Our benchmarks were conducted on an isolated Intel(R) Xeon(R) CPU E5-2407 v2 running at 2.40GHz, with 10 MB of cache and 264 GB of RAM. The operating system is GNU/Linux Devuan 2.1, with kernel `4.9.0-18-amd64`. Our implementation is written in `C++11`, using the compiler `g++` version 6.3.0 and the flags `-std=c++11`, `-O3` and `-msse4.2`. All experiments are single-threaded. We used the SDSL library[16] to get support for bitvectors and GMR arrays, and the Glushkov automata implementation of [4]. The complete source code and the instructions for compiling it can be found in the repository [27].

Let us comment on an aspect of our index. As well as in the Ring, our representation assumes the nodes and labels are in intervals $\{1, \ldots, n\}$ and $\{1, \ldots, \lambda\}$, respectively (see the beginning of Chapter 2). Wikidata, and most real-world databases, do not satisfy this hypothesis, so we need to integer-encode beforehand the database and construct two dictionaries (one for nodes and the other for labels) to recover the original information. It has been observed that recovering the original values from the dictionaries has marginal space- and time-cost [3].

4.2 Experimental Results

Index construction

Constructing the dictionary-encoded database takes 5.2 hours using the algorithm provided by Arroyuelo *et at.* [3]. Once the encoded database is available, constructing the index takes 0.4 hours. The resulting index uses 6.87 GB, that is, 7.17 bytes per triple.

The running space used is 10.28 bytes per triple. This equates to the index space plus the size of the variables that only depend on the regular expression (such as those of the automaton, which are on the order of 10^{-5} bytes per triple and therefore negligible) plus the space used by the seen table, which uses 3.11 bytes per triple.

Table 4.3 presents the construction time and resulting space usage of our index compared to the other systems. Our index is the most space-efficient, with the Ring being the closest contender, using slightly more than twice the space. This is consistent with the fact that we do not duplicate the edges for dealing with the inverted predicates, as the Ring does.

In terms of index construction time, our index maintains a competitive edge with a completion time of 5.6 hours. While Virtuoso leads the group with the fastest index time of 3.0 hours, the Proposed Index's construction time is notably less than that of Ring, which takes 7.5 hours. It also far outpaces Jena and Blazegraph, which require 37.4 and 39.4 hours respectively.

	Proposed Index	Ring	Jena	Virtuoso	Blazegraph
Index space	7.17	16.41	95.83	60.07	90.79
Index time	5.6	7.5	37.4	3.0	39.4

Table 4.3: Index space in bytes per triple and construction time in hours for the different systems. The times for the Ring and the Proposed Index include the dictionary encoding time.

Comparison of variations

In Chapter 4, we discuss that the Glushkov automata can be traversed forwardly or backwardly. We implement both versions. Additionally, we explore further variations by omitting the lazily initialized table P in Algorithms 12 and 13. Recall that $P[D]$ is responsible for storing the labels of the edges outgoing a state D. Hence, omitting storing P offers an advantage in terms of memory efficiency, potentially saving up to $O(m \cdot 2^{m+1})$ bytes. However, this comes at the cost of having to compute the labels associated with D at each inductive step. Taking these variants into account, we end up with four distinct algorithms: fwd (forward with precomputed table P), bwd (backward with precomputed table P), fwd-noP (forward without precomputed table P), and bwd-noP (backward without precomputed table P).

Figure 4.1 shows the execution times for the variations according to two query types and the four most common query patterns. Table 4.4 details the exact median and average times.

Across the entire spectrum of queries, the average execution time for fwd-noP is approximately 3% higher than that of fwd. In contrast, bwd-noP has an average time increase of around 30% when compared to bwd. This difference is notably more pronounced in the case of backward traversal, especially for single-variable queries, where the noP variant is approximately 40% slower on average. A plausible explanation for this could be that the computation of $D \& B[l]$ is more expensive than that of $T[D] \& B[l]$.

The overall execution time differences between fwd and bwd fall within the microsecond range, making them almost negligible. Note that fwd exhibits marginally faster performance, especially when the execution times are low. This observation is consistent with three factors. One is that the number of traversals leading to a valid solution is the same irrespective of the starting point in the automaton, and therefore of the type of traversal. The second one is that the queries are imbalanced, predominantly favoring scenarios where the constant is on the right side and the predicates in the RPQ are non-negated. This imbalance results in a disparity in the frequency of calls to RNEIGH() and NEIGH(), both of which have a time complexity of $O(\log\log(n))$; however, the implied constants in each case are different. Lastly, the initialization of the table P in the backward process involves an additional "$T[D]$" step. This may not be negligible when there are few inductive steps (and therefore low execution times).

In the sequel, we fix fwd as our benchmark algorithm due to its slightly better performance.

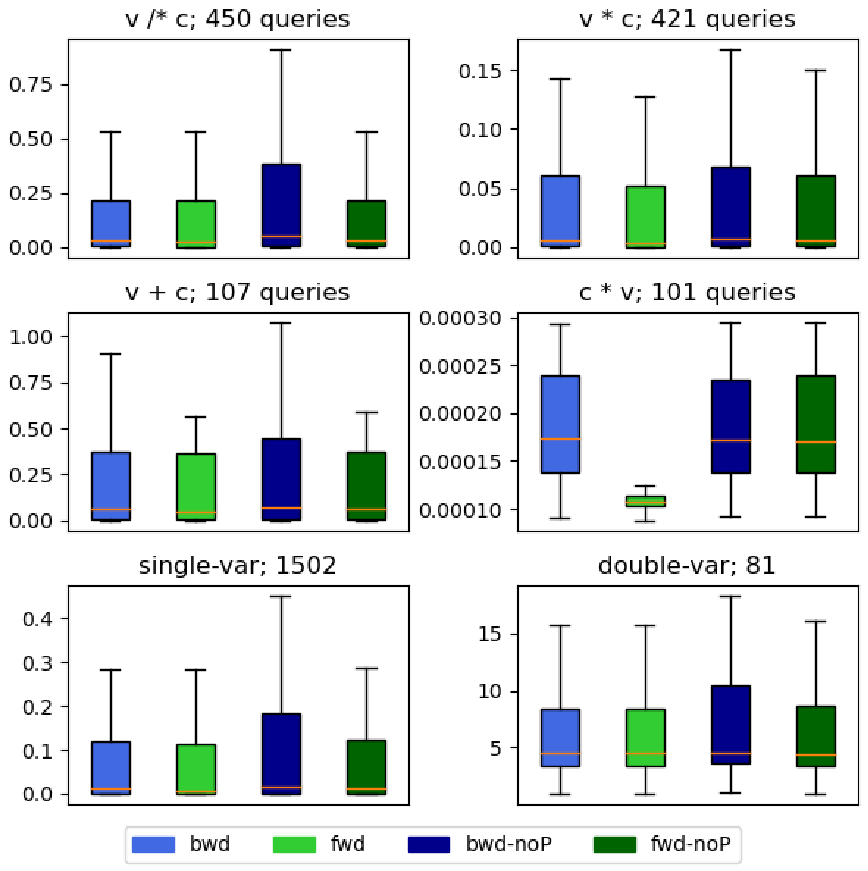


Figure 4.1: Execution times of the four algorithm variants for two query patterns and the four most common query types. Remark that in the case $c * v$, where fwd shows markedly superior performance, the execution time lies in the range of 10^{-4} seconds

		$v \; * \; c$	$v \; + \; c$	$v \; * \; c$	$c \; * \; v$	single-var	double-var
bwd	Mean	0.40	0.41	0.18	0.14	0.32	7.88
	Median	0.03	0.06	0.01	0.00	0.01	4.48
bwd-noP	Mean	0.61	0.48	0.21	0.14	0.45	9.38
	Median	0.05	0.07	0.01	0.00	0.01	4.49
fwd	Mean	0.39	0.42	0.17	0.14	0.32	7.88
	Median	0.03	0.05	0.00	0.00	0.01	4.48
fwd-noP	Mean	0.41	0.43	0.19	0.14	0.33	8.01
	Median	0.03	0.06	0.01	0.00	0.01	4.43

Table 4.4: Mean and median execution times in seconds for the cases depicted in Figure 4.1.

Comparing with other systems

We turn to present the results that compare our algorithm fwd with the Ring, Virtuoso, Blazegraph and Jenna. Figure 4.2 shows the performance of these algorithms for the 12 most popular query patterns, and Figure 4.3 does so for the different query types.

The results show that our algorithm fwd performs consistently well in single-variable queries and that offers competitive results across all query patterns. Notably, in simple patterns such as $v \; * \; c$, $v \; + \; c$, $v \; |* \; c$ and $v \; */* \; c$, fwd greatly outperforms its counterparts. For more complex query patterns, such as $v \; */*/*/*/* \; c$, Jenna, Virtuoso and Blazegraph have some of their worst times, suggesting some difficulty in dealing with the added complexity. In contrast, the Ring and fwd surpass the other algorithms, while performing similarly as in the other scenarios.

In the case of double-variable queries, fwd shows a notably poorer performance, characterized by significant variability and some of the highest time measurements. One possible reason for this performance could be the algorithm's naive approach of reducing double-variable queries to iterating over single-variable ones. This method heavily depends on finding a good initial set of possibilities, which, in the least favorable cases, might be as large as the number of nodes, with none of its elements leading to a solution.

Another factor might be the specialized optimizations integrated into the other algorithms for specific query types. For instance, the Ring treats queries such as $(s, p_1|p_2, x)$ by solving (s, p_1, x) and (s, p_2, x) separately, and then merging the results. Similarly, for queries of the form $(x, p_1/p_2, x)$, it divides them into (s, p_1, x) and (s, p_2, x) and then performs a *join* operation on the results. Blazegraph takes one step further and employs advanced strategies to convert every fixed-length RQPs (*i.e.*, those not involving $+$ and $*$) into join queries.

Regarding timeouts, Virtuoso recorded the least with just two, despite its volatile performance and above-average results. Our algorithm fwd and the Ring follow, with 13 and 18 timeouts, respectively. Blazegraph and Jena experienced considerably more timeouts, with 42 and 93 respectively.

	fwd	Ring	Jena	Virtuoso	Blazegraph
Average	0.65	0.83	1.41	2.23	1.81
Median	0.01	0.09	0.18	0.14	0.13
Timeout	13	18	93	2	42
Average $1v$	0.32	0.65	1.33	1.86	1.83
Median $1v$	0.01	0.08	0.17	0.11	0.13
Timeout $1v$	2	2	59	1	39
Average $2v$	7.9	5.1	1.8	10	2.0
Median $2v$	4.51	1.47	1.19	4.82	0.14
Timeout $2v$	11	16	34	1	3

Table 4.5: Performance comparison detailing number of timeouts (more than 60 seconds of execution time) and the average and median times for the different algorithms. The median and average times do not include the timeouts. The symbol $1v$ represents the set of single-variable queries, and $2v$ represents the set of double-variable queries.

Comparison with the Ring

Our algorithm shows similar performance to the Ring, in line with their theoretical similarities. In this subsection, we further examine the time differences between the two systems.

Figure 4.4 compares the execution times of the Ring and fwd, according to the number of results yielded by the queries and the query type. For single-variable queries, fwd performs better in almost all instances, especially as the number of results increases. With double-variable queries, the Ring tends to be more efficient, but the times of fwd relative to those of the Ring improve again as the queries yield more results.

The main issue with fwd seems to be that it incurs in a very high initial cost for queries having few results. This is consistent with what we discussed previously: our algorithm may frequently encounter feasible sets having many nodes that do not ultimately lead to solution tuples. Another factor that might explain this phenomenon is that the GMR arrays used in fwd have faster access times than the wavelet trees used by the Ring. Hence, as the number of results increases, more accesses are required to retrieve the nodes, further highlighting the time differences.

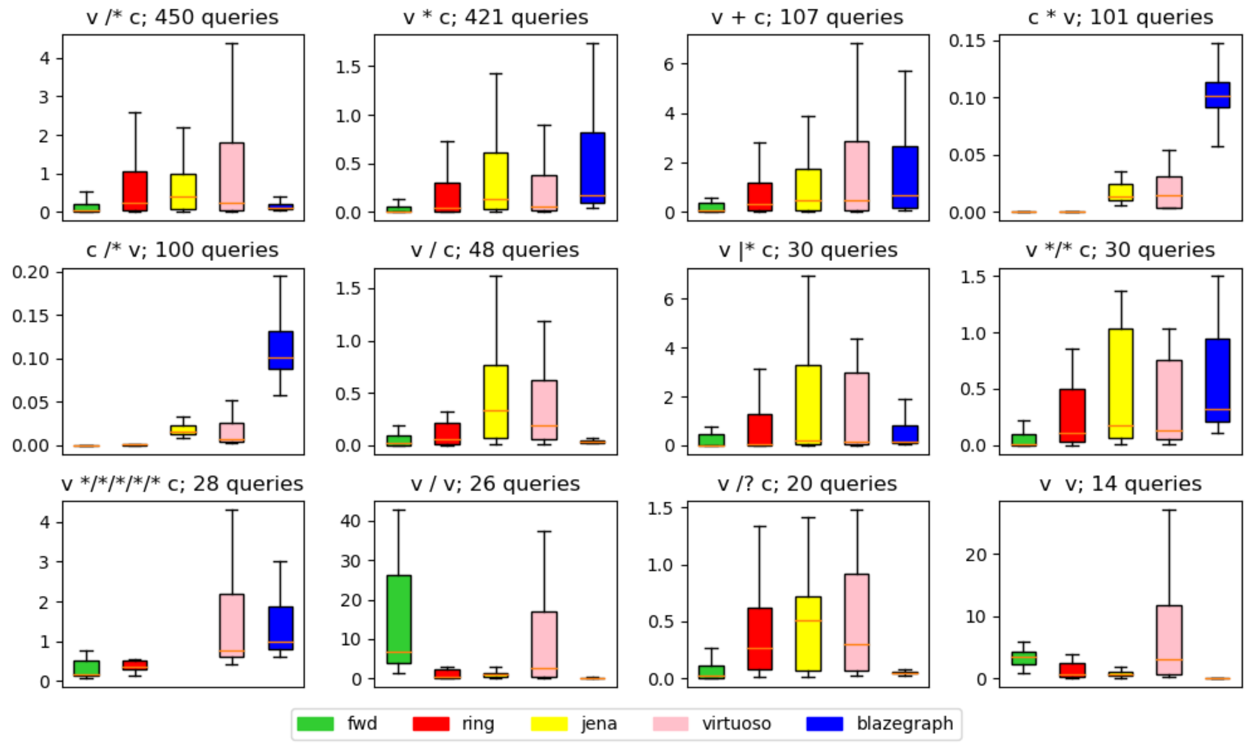


Figure 4.2: Time taken by the different algorithms across the most common query patterns. Note that the results for the query patterns $c\ *\ v$ and $c\ /*\ v$ of the Ring and fwd are very close to the x-axis, as it is the case for Blazegraph for the patterns $v\ /\ v$ and $v\ v$. In the results for the pattern $v\ */*/*/*/*\ c$, Jenna does not appear as it timed out on all the 28 queries.

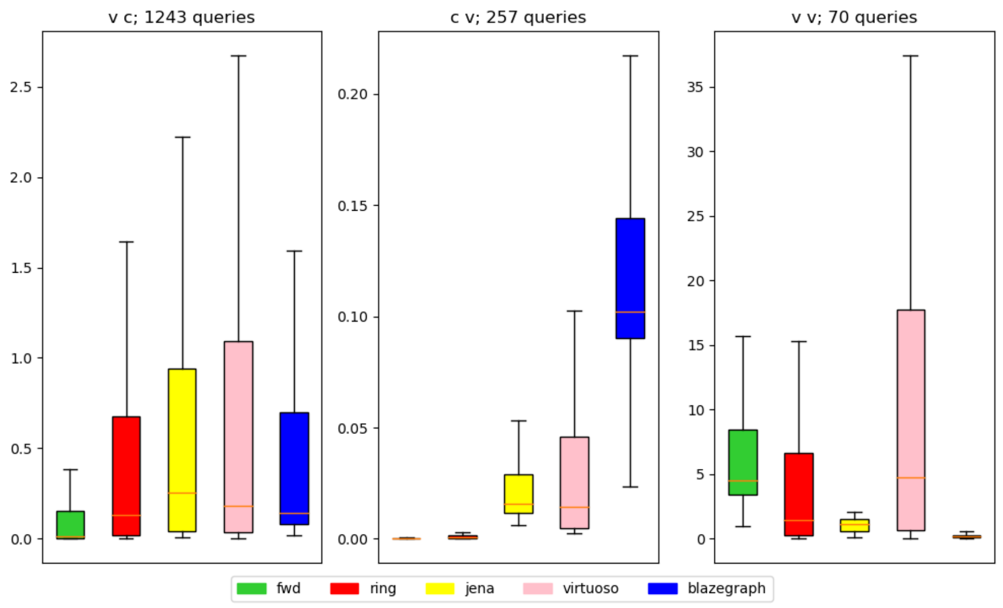


Figure 4.3: Time taken by the different algorithms across the three query types.

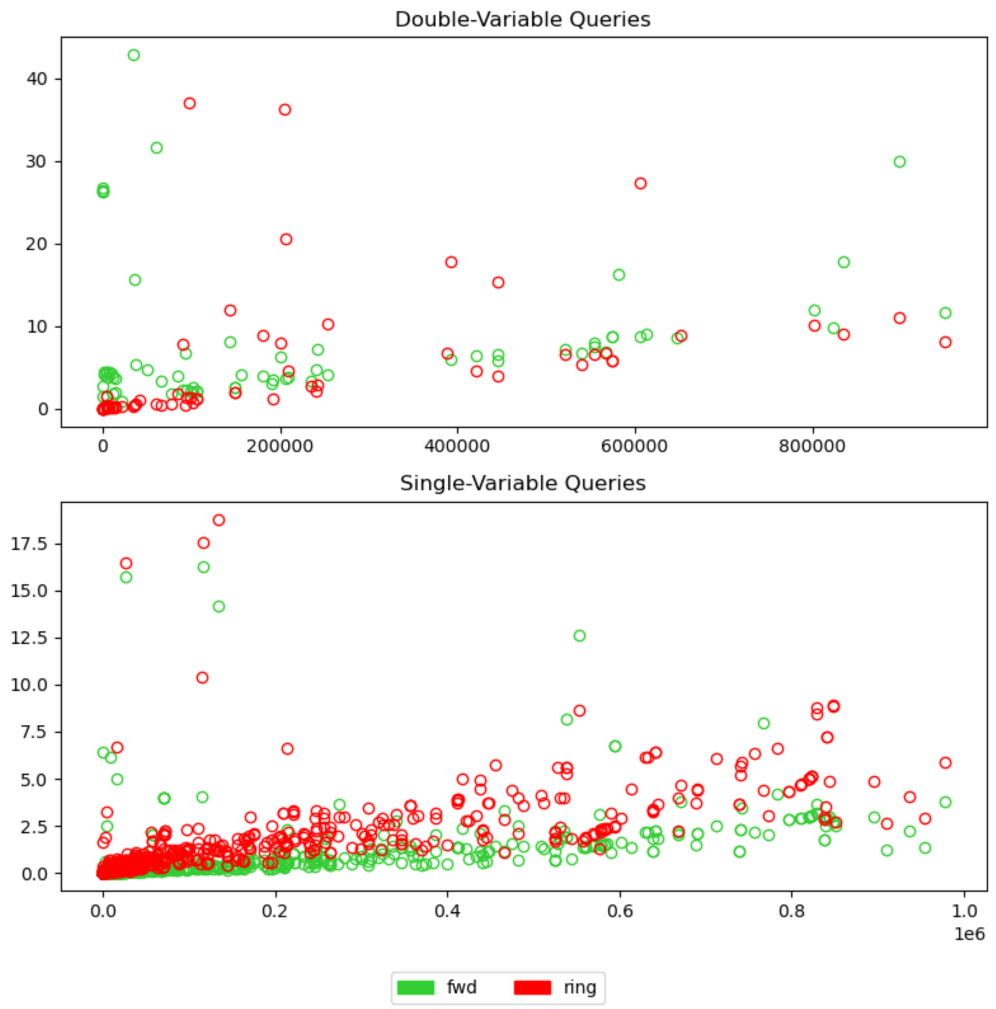


Figure 4.4: Execution times of the Ring and fwd (x-axis) according to the number of results yielded by the queries (y-axis). The figure above shows the results for double-variable queries, and the one below for single-variable queries.

Chapter 5

Conclusions and Future Work

RDF databases provide a simple scheme based on triples for structuring and linking data that has gained widespread adoption, notably by the Semantic Web. In response to the scalability challenges posed by the continuous increase of the data volumes, the community actively seeks space- and time-efficient algorithms for managing RDF databases.

In this thesis, we have implemented a compact index for RDF databases introduced by Navarro [26], which uses bitvectors and GMR-sequences as underlying data structures. Building upon this, we developed a novel algorithm for solving 2RPQs, a central query class for these databases. The resulting index and algorithm excel in storage efficiency and single-variable query processing, while maintaining competitive performance for double-variable queries.

The implemented index is inspired by the Burrows-Wheeler transform, a string compression algorithm. It is build by considering strategically chosen orderings of the RDF triples and storing, according to them, two arrays containing its subjects and objects, together with two space-negligible bitvectors that contain information about the orderings. This method avoids explicitly storing the objects, allowing asymptotically optimal space-usage. Methods for recovering useful data, such as neighboring nodes or all subjects for a fixed predicate, were implemented making use of the structure of the index.

One of the advantages of this index is its flexibility in the data structures that can be used for its implementation. Indeed, the methods for accessing the data only use RSA operations on the arrays and bitvectors. Our particular implementation employs plain bitvector and GMR-arrays, which were selected for their well-known good space and time performance.

Our algorithm for solving 2RPQs takes advantage of the index's retrieval functions, combined with an automaton-based approach for handling regular expressions. Specifically, it uses Glushkov automata due to its capability of non-deterministic traversal. This enabled us to navigate the graph and the automaton concurrently, with branching occurring only in response to interactions with the database.

We conducted a series of benchmarks using Wikidata to compare our algorithm with leading RDF systems –Blazegraph, Jena, and Virtuoso– as well as the Ring, an index with similar theoretical design to ours. These evaluations revealed that our approach is the most space-efficient, requiring only half the storage space of the Ring, its nearest competitor. The construction of our index was completed in approximately 5.6 hours, making it the second fastest, only surpassed by Virtuoso. For single-variable 2RPQs, our solution exhibited remarkable performance, consistently outperforming the other systems in every scenario, regardless of the constant's position in the query or the complexity of the regular expression. However, while still competitive, the performance of our algorithm for double-variable queries was found to be unsatisfactory, which indicates potential areas for further improvement.

Future work

The high variability in outcomes for double-variable 2RPQs highlights the need for further refinement and testing of this case. We propose the following concrete ideas for analyzing this problem:

- As our algorithm first computes a feasible set, it is important to check whether it tends to be much bigger than the optimum, *i.e.*, than the set of constants leading to at least one solution. In addition to a substantial discrepancy between these two sets, the process of discarding constants not shared by both may be excessively time-consuming.
- It is possible that merely computing the feasible set, which primarily involves calls to sources or targets followed by filtering duplicate nodes, constitutes a main time-consuming step.
- Considering that the other algorithms use specialized subroutines when dealing with certain query types (for instance, the queries of type v/v are usually solved using join operations), it is relevant to test the performance of our algorithm according/in line with/as per to the competitors' used subroutines. Depending on these results, it might be necessary to optimize our algorithm by query type, as the competitors do. We remark that this may also improve our single-variable query performance.

Another interesting idea for future research is that, since the experiments showed that our index is by far the most space-efficient one, it is viable to trade used space for query processing time. This can be done by either using heavier structures with better performance or by storing additional information about the database structure.

With respect to the first point, we can explore other data structures for handling sequences that may offer faster RSA times or additional methods, such as enhanced duplicate element management. For instance, without deviating far from the current structures, the article in which GMR arrays are introduced include a variant of it that swaps the time costs of select and access, while slightly increasing the time complexity of rank to $O(\log\log(\sigma)\log\log\log(\sigma))$. This yields a different interplay between the different methods used in our algorithm, and thus potentially different performance.

We can also store additional information of the database.A concrete example that can be found in the literature is storing node connectivity data, which helps to reduce queries to a small set of simpler ones by "splitting" the original query at a weakly connected node.

Lastly, our index offers several efficient functions for retrieving graph information, of which we primarily utilize only four. Future iterations of the algorithm can explore the remaining functions or focus on introducing new ones. For instance, given an inductive pair (D, v), we can derive the next step's predicates by examining those connected to v in the graph, rather than inspecting labels associated with D in the automaton. To this end, the outgoing predicates of v can be retrieved by locating and accessing the corresponding L_v segment of L, while the ingoing ones can be obtained by adapting the algorithm that implements $\mathsf{sources}$, replacing L with N, B_L with B_N, and select_l with select_v.

Bibliography

[1] Waqas Ali et al. "A survey of RDF stores and SPARQL engines for querying knowledge graphs". In: *The VLDB Journal* 31 (Nov. 2021). DOI: 10.1007/s00778-021-00711-3.

[2] *Apache Jena*. https://jena.apache.org/.

[3] Diego Arroyuelo et al. "Optimizing RPQs over a compact graph representation". In: *The VLDB Journal* (2023). ISSN: 0949-877X. DOI: 10.1007/s00778-023-00811-2. URL: http://dx.doi.org/10.1007/s00778-023-00811-2.

[4] Diego Arroyuelo et al. *Ring-RPQ*. https://github.com/darroyue/Ring-RPQ. 2022.

[5] Diego Arroyuelo et al. "Worst-Case Optimal Graph Joins in Almost No Space". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 102–114. DOI: 10.1145/3448016.3457256. URL: https://doi.org/10.1145/3448016.3457256.

[6] Jorge A. Baier et al. "Evaluating Navigational RDF Queries over the Web". In: *Proceedings of the 28th ACM Conference on Hypertext and Social Media, HT 2017, Prague, Czech Republic, July 4-7, 2017*. Ed. by Peter Dolog et al. ACM, 2017, pp. 165–174. DOI: 10.1145/3078714.3078731. URL: https://doi.org/10.1145/3078714.3078731.

[7] Gerard Berry and Ravi Sethi. "From regular expressions to deterministic automata". English (US). In: *Theoretical Computer Science* 48.C (1986), pp. 117–126. ISSN: 0304-3975. DOI: 10.1016/0304-3975(86)90088-5.

[8] *Blazegraph*. https://www.blazegraph.com/.

[9] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. "DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases". In: *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*. Ed. by Abraham Bernstein et al. Vol. 5823. Lecture Notes in Computer Science. Springer, 2009, pp. 97–113. DOI: 10.1007/978-3-642-04930-9_7. URL: https://doi.org/10.1007/978-3-642-04930-9%5C_7.

[10] V. Chandru and V. Vinay, eds. *Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, 1996, pp. 37–42. DOI: 10.1007/3-540-62034-6. URL: https://doi.org/10.1007/3-540-62034-6.

[11] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[12] Martin J. Dürst and Michel Suignard. "Internationalized Resource Identifiers (IRIs)". In: *RFC* 3987 (2005), pp. 1–46. DOI: 10.17487/RFC3987. URL: https://doi.org/10.17487/RFC3987.

[13] Orri Erling and Ivan Mikhailov. "RDF Support in the Virtuoso DBMS". In: *Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Tech-*

nologies and Semantic Systems. Ed. by Tassilo Pellegrini et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 7–24. ISBN: 978-3-642-02184-8. DOI: `10.1007/978-3-642-02184-8_2`.

[14] V M Glushkov. "The abstract theory of automata". In: *Russian Mathematical Surveys* 16.5 (Oct. 1961), pp. 1–53. DOI: `10.1070/rm1961v016n05abeh004112`. URL: `https://doi.org/10.1070/rm1961v016n05abeh004112`.

[15] Simon Gog et al. "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 2014, pp. 326–337. URL: `https://github.com/simongog/sdsl-lite`.

[16] Simon Gog et al. *SDSL - Succinct Data Structure Library*. `https://github.com/simongog/sdsl-lite`. 2016.

[17] A. Golynski, J. Munro, and S. Rao Satti. "Rank/Select Operations on Large Alphabets: a Tool for Text Indexing". In: Jan. 2006, pp. 368–373. DOI: `10.1145/1109557.1109599`.

[18] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 2003, pp. 841–850. URL: `http://dl.acm.org/citation.cfm?id=644108.644250`.

[19] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. "Sparqling kleene: fast property paths in RDF-3X". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. Ed. by Peter A. Boncz and Thomas Neumann. CWI/ACM, 2013, p. 14. DOI: `10.1145/2484425.2484443`. URL: `http://event.cwi.nl/grades2013/14-gubichev.pdf`.

[20] Jinha Kim et al. "Taming Subgraph Isomorphism for RDF Query Processing". In: *Proc. VLDB Endow.* 8.11 (2015), pp. 1238–1249. DOI: `10.14778/2809974.2809985`. URL: `http://www.vldb.org/pvldb/vol8/p1238-kim.pdf`.

[21] Egor V. Kostylev et al. "SPARQL with Property Paths". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. Ed. by Marcelo Arenas et al. Vol. 9366. Lecture Notes in Computer Science. Springer, 2015, pp. 3–18. DOI: `10.1007/978-3-319-25007-6\_1`. URL: `https://doi.org/10.1007/978-3-319-25007-6%5C_1`.

[22] Yongming Luo et al. "Storing and Indexing Massive RDF Datasets". In: *Semantic Search over the Web*. Ed. by Roberto De Virgilio, Francesco Guerra, and Yannis Velegrakis. Data-Centric Systems and Applications. Springer, 2012, pp. 31–60. DOI: `10.1007/978-3-642-25008-8\_2`. URL: `https://doi.org/10.1007/978-3-642-25008-8%5C_2`.

[23] Stanislav Malyshev et al. "Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph". In: *Proceedings of the 17th International Semantic Web Conference (ISWC'18)*. Ed. by Denny Vrandečić et al. Vol. 11137. LNCS. Springer, 2018, pp. 376–394.

[24] Wim Martens and Tina Trautner. "Dichotomies for Evaluating Simple Regular Path Queries". In: *ACM Trans. Database Syst.* 44.4 (2019), 16:1–16:46. DOI: `10.1145/3331446`. URL: `https://doi.org/10.1145/3331446`.

[25] J. Ian Munro et al. "Succinct Representations of Permutations". In: *Automata, Languages and Programming*. Ed. by Jos C. M. Baeten et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 345–356. ISBN: 978-3-540-45061-0.

[26] Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016, pp. 285–291. DOI: 10.1017/CBO9781316588284.

[27] Josefa Robert. *A compact graph structure for efficiently solving RPQs.* https://github.com/j-rparra/compactGraph. 2023.

[28] Sherif Sakr and Ghazi Al-Naymat. "Relational processing of RDF queries: a survey". In: *SIGMOD Rec.* 38.4 (2009), pp. 23–28. DOI: 10.1145/1815948.1815953. URL: https://doi.org/10.1145/1815948.1815953.

[29] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. "YAGO: A Core of Semantic Knowledge". In: *Proceedings of the 16th international conference on World Wide Web* (2007), pp. 697–706.

[30] Frank Tetzel et al. "An Analysis of the Feasibility of Graph Compression Techniques for Indexing Regular Path Queries". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017.* Ed. by Peter A. Boncz and Josep Lluis Larriba-Pey. ACM, 2017, 11:1–11:6. DOI: 10.1145/3078447.3078458. URL: https://doi.org/10.1145/3078447.3078458.

[31] Denny Vrandecic and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (2014), pp. 78–85. DOI: 10.1145/2629489. URL: https://doi.org/10.1145/2629489.

[32] *Wikidata Graph Pattern Benchmark.* http://compact-leapfrog.tk/. 2020.

[33] Dan E. Willard. "Log-logarithmic worst-case range queries are possible in space $\Theta(N)$". In: *Information Processing Letters* 17.2 (1983), pp. 81–84. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(83)90075-3. URL: https://www.sciencedirect.com/science/article/pii/0020019083900753.

[34] Lei Zou et al. "Efficient processing of label-constraint reachability queries in large graphs". In: *Inf. Syst.* 40 (2014), pp. 47–66. DOI: 10.1016/J.IS.2013.10.003. URL: https://doi.org/10.1016/j.is.2013.10.003.

[35] Lei Zou et al. "gStore: Answering SPARQL Queries via Subgraph Matching". In: *Proc. VLDB Endow.* 4.8 (2011), pp. 482–493. DOI: 10.14778/2002974.2002976. URL: http://www.vldb.org/pvldb/vol4/p482-zou.pdf.